X'con 2005

How to exploit Windows kernel memory pool

kinvis@hotmail.com
SoBeIt
Beihang University

Memory pool: mechanism and algorithm

- **Memory pool - mechanism:**
  - Overview
  - Introduction of PoolDescriptor
- **Memory pool – request algorithm:**
  - Handling differently based on request size
  - LookAsideList and algorithm on top of it

Memory pool: mechanism and algorithm

-- mechanism

- The memory pool is used for kernel memory allocation, the same as user-mode heap. The routines are ExAllocatePool() and ExFreePool() separately.
- The memory pool is managed by PoolDescriptor, we'll mention it later.
- The memory pool has two categories: NonPagedPool and PagedPool, the former is swappable, while the latter must reside in memory.

Memory pool: mechanism and algorithm

-- mechanism

- NonPagedPool has two parts, determined by (MmNonPagedPoolStart, MnNonPagedPoolEnd) and(MmNonPagedPoolExpansionStart, MmNonPagedPoolExpansionEnd), mostly located at 0x8xxxxxxx and 0xfxxxxxxx-0xffbe0000。
- PagedPool is determined by (MmPagedPoolStart, MmPagedPoolEnd), mostly located at 0xexxxxxxx。

Memory pool: mechanism and algorithm

--mechanism

Pool is managed by PoolDescriptor, its structure is:

```
typedef struct _POOL_DESCRIPTOR {
    POOL_TYPE PoolType;
    ULONG PoolIndex;
    ULONG RunningAllocs;
    ULONG RunningDeAllocs;
    ULONG TotalPages;
    ULONG TotalBigPages;
    ULONG Threshold;
    PVOID LockAddress;
    LIST_ENTRY ListHeads[POOL_LIST_HEADS];
} POOL_DESCRIPTOR, *PPOOL_DESCRIPTOR;
```

Memory pool: mechanism and algorithm

--mechanism

PoolDescriptor has several key member:

- PoolType:Type of memory pool, can be PagedPool, NonPagedPool, NonPagedPoolMust etc, in fact it's the index of PoolVector.
- PoolIndex:Applied to PagedPool, it's index value of PoolDescriptor in PagedPool array.
- ListHeads: The allocation grain of pool, 32 bytes at least. In order to manage these chunks, the free chunks of the same size are in the same double-linked list. So number of lists is 4096 / 32 = 128, which is the value of POOL_LIST_HEADS.

Memory pool: mechanism and algorithm

-- mechanism

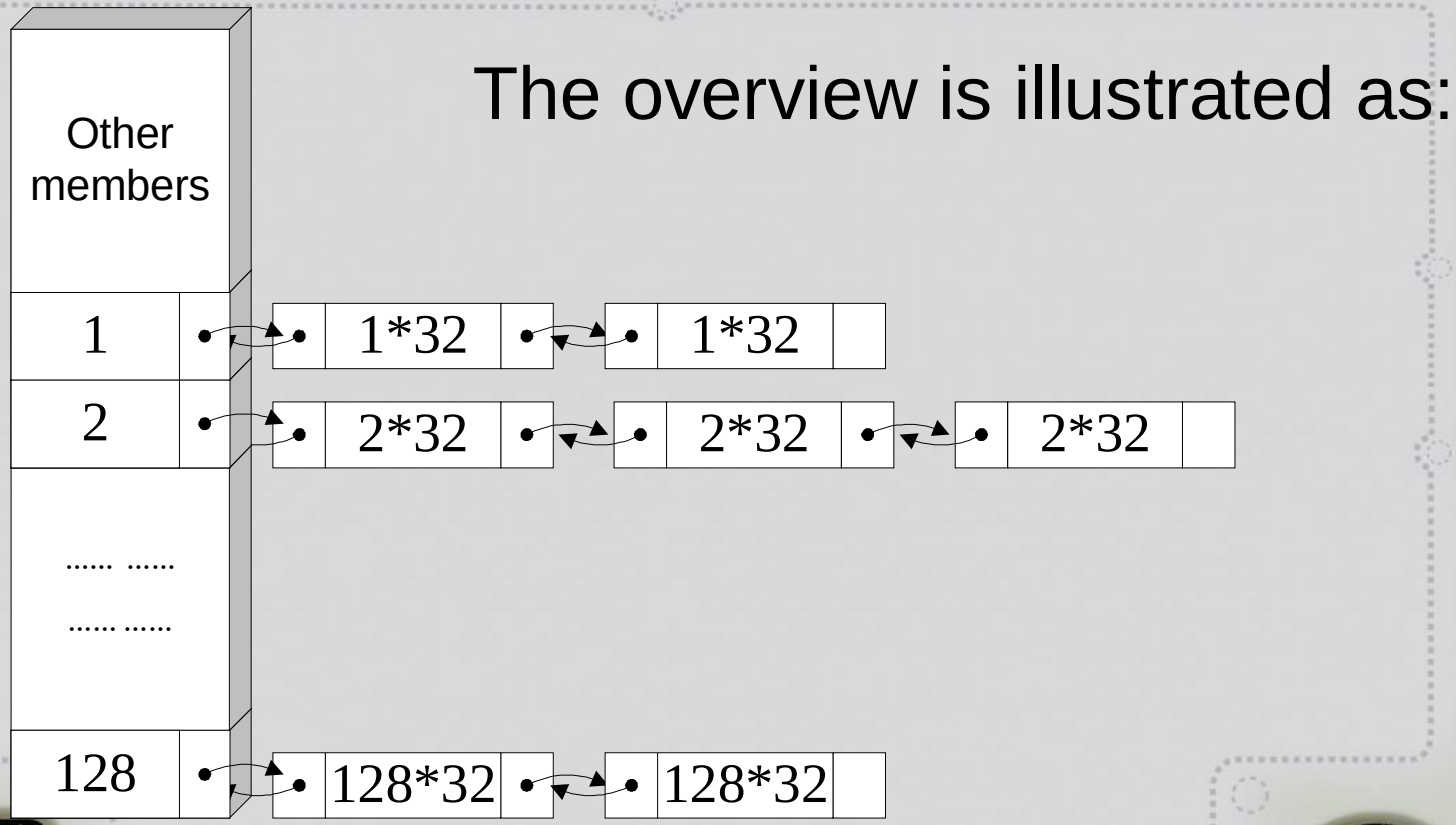
PoolDescriptor is managed by global array PoolVector, includuing 3 membrers:

Two pointers pointing to two statically allocated descriptor NonPagedPoolDescriptor and NonpagedPoolDescriptorMS, and one pointer pointing to PagedPoolDescriptor array.

Memory pool: mechanism and algorithm

-- mechanism

The overview is illustrated as:



Memory pool: mechanism and algorithm

-- mechanism

```
typedef struct _POOL_HEADER {
   union {
      struct {
         UCHAR PreviousSize;
         UCHAR PoolIndex;
         UCHAR PoolType;
         UCHAR BlockSize;
      };
      ULONG Ulong1;
   };
   union {
      EPROCESS *ProcessBilled;
      ULONG PoolTag;
      struct {
         USHORT
     AllocatorBackTraceIndex;
         USHORT PoolTagHash;
      };
   };
} POOL_HEADER, *PPOOL_HEADER;
```

Just like heap, each requested pool has a management structure, which definition is on the left:

Memory pool: mechanism and algorithm -- mechanism

- PreviouSize: Size of previous chunk, the value should be the result of division by 32. In case of the 1st chunk for each page, the value should be 0.
- PoolIndex: For PagedPool, it is allocated from PagedPoolDescriptor in a loop, PoolIndex is the index of PagedPoolDescriptor the chunk belongs to. For free chunks, PoolIndex is the actually index, while for allocated chunks, PoolIndex equals actually index plus 0x80. When freeing, system will use value & 0x80 to determine whether this pool chunk is freed.
- PoolType:0 when free, pool type plus 1 when allocated. When freeing and merging, system determines whether it's free based on this member from its neighbors.
- BlockSize:Size of current chunk, it equals requested size plus 8 bytes(management struct) aligned by 8.
- PoolTag:Normal chunk requests, it's 4 bytes chars, differs based on requested type.

Memory pool: mechanism and algorithm

-- algorithm

3 cases based on requested size:

Case 1:

When requested size is large than 0xfb8, i.e. page size – size of pool management struct – size of chunk (4096-8-32), allocate aligned one or several pages through MiAllocatePoolPage.

Memory pool: mechanism and algorithm

-- algorithm

Case 2:

When requested size is between `0x100` and `0xfd8`, a proper chunk from ListHeads linked list of PoolDescriptor is returned. The algorithm is similar but simple than buddy algorithm. By walking through the linked list, the system will find a suitable chunk, get if off, then cut it to the right size, insert the remaining chunk to the correspondent list. When freed, the adjacent chunks will merge if possible, and inserted to the correspondent list.

Memory pool: mechanism and algorithm

-- algorithm

Case 2(Cont.):

When allocated from PagedPool, the allocation algorithm use Round-Robin to obtain the lock of some pool descriptor except item 0. When lock is obtained, memory will be allocated from this pool. Next time, the lock is requested from the next pool descriptor. So two ExAllocatePool calls will get memory from different pool

Memory pool: mechanism and algorithm

-- algorithm

Case 3:

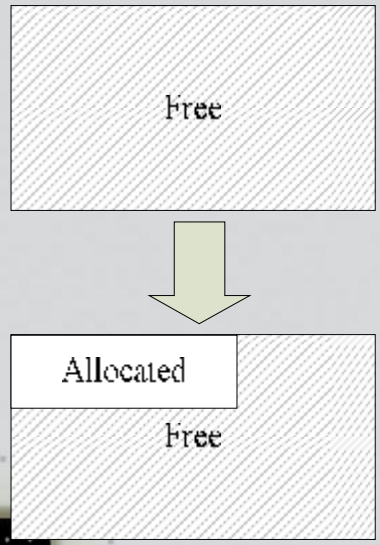
When requested size is equal or less than 0x100, for such frequent chunk allocation, the system will use Lookaside linked list for the reason of efficiency. Lookaside is a heap data based on linked list, located at KPCR. PagedPool and NonPagedPool has 8 PP_LOOKASIDE_LIST separately, ranged from 32 to 256. Each structure has 2 linked lists, represented by auto-balanced binary tree.

Memory pool: mechanism and algorithm

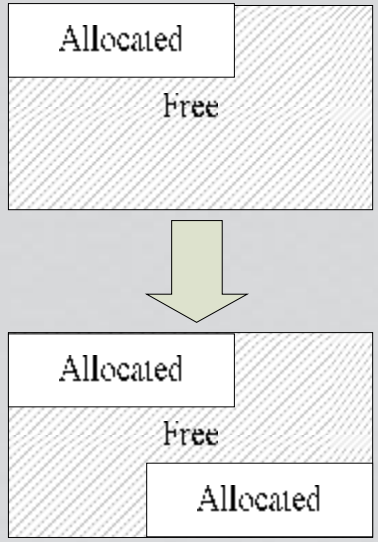
-- algorithm

Allocation order:

Allocation of free pages:
---begin from the head



Allocation of non-free pages:
---begin from the tail



Memory pool: mechanism and algorithm

-- algorithm

LookasideList and algorithm on top of it

LookasideList is based on pool allocator. By calling ExAllocatePool to allocate frequent-used size, the system will directly pick up a pool chunk from the list. ExAllocatePool is managed by one-way linked list, which is auto-balanced. On frequent pickups, Lookaside will call ExAllocatePool; when there're many pool chunk, ExFreePool will be called.

Memory pool: mechanism and algorithm

-- algorithm

LookasideList and algorithm on top of it (Cont):

You can call ExInitializePagedLookasideList or ExInitializeNPagedLookasideList to setup LookasideList of PagedPool and NonPagedPool, and specify the chunk size. Later you can call ExAllocateFromPagedLookasideList or ExAllocateFromNPagedLookasideList to get pool chunk you want from LookasideList, and call ExFreeToPagedLookasideList or ExFreeToNPagedLookasideList to release the chunk back to LookasideList.

The system has several self-used LookasideList in PPLookasideList array of KPCR, this array has 16 items, but only 7 items are used.

Exploit difficulty compared with heap overflow

- No default heap per process, no way to build its own heap in user-mode. All kernel-mode applications shared those pools, which adds uncertainty of allocated addresses. For PagedPool, since the allocation is from two pool descriptors one by one, it's almost IMPOSSIBLE to control the allocation addresses. So most methods for heap overflow are not useful.
- When overflow occurs in pool, you can't use a newly-created heap as the default heap in user-mode. You have to repair the pool manually, so, try to ruin the pool descriptors as less as possible.
- For pool overflow, there's no accurate way to locate shellocde just like heap overflow(you can create a heap marked as LAST_ENTRY).

Exploit difficulty compared with heap overflow

- The heap overflow is in kernel-mode, IRQL is likely to be DISPATCH_LEVEL, you can control the system but it's after the exception. The kernel-mode exception is critical than user-mode, if not handled correctly, BLUE SCREEN! So, a careful restore is necessary.
- Which pointers to overflow if you want to take control?

Exploit method

We can overflow KiDebugRoutine, which is a built-in interface of kernel debugging. When each exception occurs, KiDispatchException will see whether KiDebugRoutine is NULL, then call it if possible. By overwriting this pointer, we can take control and return to normal. The exception is triggered when system frees the faked pool or the pool next to the faked pool.

Exploit method - I

--- build free pool(not recommended)

By building a free pool chunk behind the overflowed pool. When the overflowed pool is freed, the merge occurs, so we can overwrite any 4 bytes. After overwriting KiDebugRoutine function pointer, we can take control. Since the overflowed pool address is in heap, we can use a jump instruction heading for this address.

Exploit method - I

--- build free pool(not recommended)

- Pros: Can be applied to PagedPool and NonPagedPool
- cons: Can't be the last pool chunk, otherwise no merge afterwards. The distance between overflowed pool address and current heap address is far, so, it's not easy to find the correct jump instruction.

Exploit method - Ⅱ

----merge free pool across the page (recommended)

Build a free pool chunk after the overflowed pool, the two pools are larger than one page. So after overwrite 4 bytes, the address AddListTail inserts is under our control, we can overwrite another 4 bytes, then we can locate our shellcode accurately.

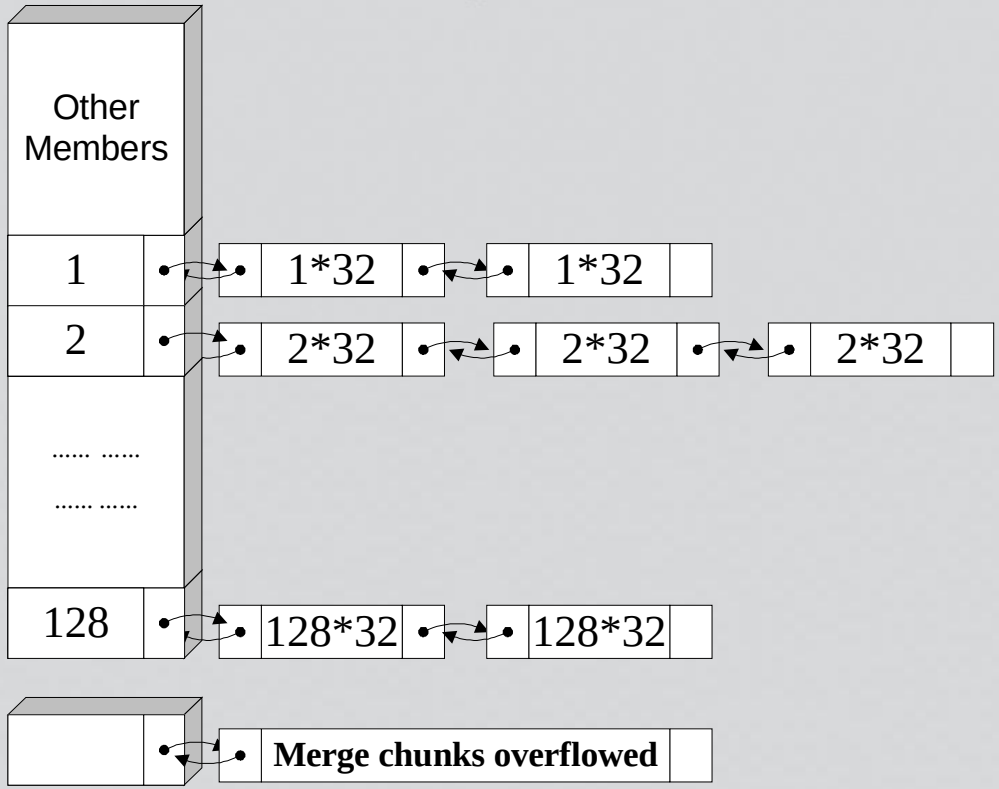
Exploit method - Ⅱ

----merge free pool across the page (recommended)

The list head address will be written to the address after overflowed pool structure. So the last byte of this address is 0xx0 or 0xx8. We can use the following jumps:0xe0(loopnz 0xxxxxxxxx)、0x70(jo 0xxxxxxxxxx)、0x78(js 0xxxxxxxxxx)。

Exploit method - Ⅱ

----merge free pool across the page (recommended)



Exploit method - Ⅱ

----merge free pool across the page (recommended)

- 1st pointer overwrite(RemoveEntry())

```
8046b6de 890a            MOV    [EDX],ECX
8046b6e0 895104          MOV    [ECX+0X4],EDX
```

- 2nd pointer overwrite(AddListTail())

```
8046b7b5 8b54cf1c        MOV
     EDX,[EDI+ECX*8+0x1c]
8046b7b9 8d44cf18        LEA
     EAX,[EDI+ECX*8+0x18]
8046b7bd 8d4e08          LEA    ECX,[ESI+0X8]
8046b7c0 89560c          MOV    [ESI+0XC],EDX
8046b7c3 8901            MOV    [ECX],EAX
8046b7c5 890a            MOV    [EDX],ECX
8046b7c7 894804          MOV    [EAX+0X4],ECX
```

Exploit method - Ⅱ

----merge free pool across the page (recommended)

- Pros:Locate shellcode accurately; easy to restore pool management structure.
- Cons:Only applied to statically allocated NonPagedPool, not dynamically allocated PagedPool. It can't be the last pool chunk, otherwise no merge afterwards. We must assume the pool in front of the overflowed pool is not free, otherwise no merge backwards.

Exploit method

Common disadvantage:

Not stable, dependent on version of system and SP.

Exploit method

----What does ShellCode do?

- (1) Repair pool chunk lists of PoolDescriptor. Initialize all lists of PoolDescriptor, enumerate all the pool chunks in the same page of overflowed pool, fix the chunk size based on its neighbors, set PoolIndex to 0x80, set PoolType to 1, so no merge both backwards and afterwards.
- (2) Search necessary functions from export tables of NTOSKRNL.
- (3) Search processes having SYSTEM priorities such as lsass.exe,csrss.exe,serverice.exe, then get thread which is in Alertable state(It should have one!). Later, insert APC of user-mode shellcode to be executed to this thread, waiting for execution.
- (4) Restore exception. Call ZwYieldExcution in shellcode to stop exception dispatcher returning in this exception.

Demo

Any questions?

Thanks!