- Sitemap

« EncFSGui – GUI Wrapper around encfs for OSX

Please consider donating: https://www.corelan.be/index.php/donate/

11,708 views | This page as PDF

Windows 10 x86/wow64 Userland heap

Published July 5, 2016 | By Corelan Team (corelanc0d3r)

Introduction

Hi all,

Over the course of the past few weeks ago, I received a number of "emergency" calls from some relatives, asking me to look at their computer because "things were broken", "things looked different" and "I think my computer got hacked".� I quickly realized that their computers got upgraded to Windows 10.

We could have a lengthy discussion about the strategy to force upgrades onto people, but in any case it is clear that Windows 10 is gaining market share.� This also means that it has become a Windows version that is relevant enough to investigate.

In this post, I have gathered some notes on how the userland heap manager behaves for 32bit processes in Windows 10.� The main focus of my investigation is to document the similarities and differences with Windows 7, and hopefully present some ideas on how to manipulate the heap to increase predictability of its behavior.�� More specifically, aside from documenting behavior, I am particularly interested in getting answers to the following questions:

1. How does the back-end allocator behave?
2. What does it take to activate the LFH?
3. What are the differences in terms of LFH behavior between Win7 and Win10, if any?
4. What options do we have to create a specific heap layout that involves certain objects to be adjacent in memory?� (objects of the same size, objects of different sizes)
5. Can we perform a "reliable" and precise heap spray?

As I am a terrible reverse engineer, it’s worth noting that the notes below are merely a transcript of my observations, and not backed by disassembling/decompiling/reverse engineering ntdll.dll.� Furthermore, as the number of tests were limited (far below what would be needed to provide some level of statistical certainty), I am not 100% sure that my descriptions can be considered a true representation of reality.�� In any case, I hope the notes will inspire people to do more tests, to perform reverse engineering on some of the heap management related code, and share more details on how things were implemented.

This post assumes that you already have experience with the Windows 7 heap and its front-end and back-end allocators, and that you understand the output of various !heap commands.��

In my test applications, I'll be using the default process heap.� It would be fair to assume that the notes below apply to all windows heaps and applications that rely on the windows heap management system.

Throughout this post, I will use the following terminology:

- chunk: a contiguous piece of memory
- block: size unit, referring to 8 bytes of memory.� (Don't be confused when you see the word "block" in WinDBG output. WinDBG uses the term "block" to reference a chunk.� I am using different terminology.
- virtualallocdblock: a chunk that was allocated through RtlAllocateHeap, but is larger than the VirtualMemoryTHreshold (and thus is not stored inside a segment, but rather as a separate chunk in memory), and managed through the VirtualAllocdBlocks list. (offset 0x9c in the heap header)
- segment: the heap management unit, managed by a heap, used to allocate & manage chunks.� The segment list is managed inside the heap header (offset 0xa4)
- SubSegment: the LFH management unit used to manage LFH chunks

Test environment

My test environment consists of the following components:

- Windows 10 Enterprise x64, fully patched� (running as a virtual machine inside VirtualBox) with 2 CPUs and 1.8Gb of RAM
- Visual Studio Express 2015 for desktop : https://go.microsoft.com/fwlink/?LinkId=691984&clcid=0x409
- WinDBG (for Windows 10) : https://go.microsoft.com/fwlink/p/?LinkId=536682.� (Make sure to run a recent version of WinDBG to make sure all heap related structures are properly parsed and represented)

I have set up Symbol support for WinDBG by creating a system environment variable:

- **Variable**:� _NT_SYMBOL_PATH
- **Value**: srv*c:\symbols*http://msdl.microsoft.com/download/symbols

The source code (Visual Studio C++ projects) for all test cases used in this post can be found here:� https://github.com/corelan/win10_heap�

The Heap

Similar to previous Windows versions, the Windows 10 heap sits at an ASLR-influenced ("random") address and starts with a header. As the base address of the heap management unit sits at a non-static address, you'll see different heap base addresses throughout this post.� In order to avoid confusion, I'll use the term "address of default process heap" to refer to this base address.� Similarly, any address you'll see in the post will be different on your machine.�

Dumping the header contents of a heap, we can oberve the following fields:

```
0:003> dt _HEAP 00a40000
```

```
ntdll!_HEAP
+0x000 Segment : _HEAP_SEGMENT
+0x000 Entry : _HEAP_ENTRY
+0x008 SegmentSignature : 0xffeeffee
+0x00c SegmentFlags : 2
+0x010 SegmentListEntry : _LIST_ENTRY [ 0xa400a4 - 0xa400a4 ]
+0x018 Heap : 0x00a40000 _HEAP
+0x01c BaseAddress : 0x00a40000 Void
+0x020 NumberOfPages : 0xff
+0x024 FirstEntry : 0x00a40498 _HEAP_ENTRY
+0x028 LastValidEntry : 0x00b3f000 _HEAP_ENTRY
+0x02c NumberOfUnCommittedPages : 0xe9
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved : 0
+0x038 UCRSegmentList : _LIST_ENTRY [ 0xa55ff0 - 0xa55ff0 ]
+0x040 Flags : 2
+0x044 ForceFlags : 0
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask : 0x100000
+0x050 Encoding : _HEAP_ENTRY
+0x058 Interceptor : 0
+0x05c VirtualMemoryThreshold : 0xfe00
+0x060 Signature : 0xeeffeeff
+0x064 SegmentReserve : 0x100000
+0x068 SegmentCommit : 0x2000
+0x06c DeCommitFreeBlockThreshold : 0x800
+0x070 DeCommitTotalFreeThreshold : 0x2000
+0x074 TotalFreeSize : 0x462
+0x078 MaximumAllocationSize : 0x7ffdefff
+0x07c ProcessHeapsListIndex : 1
+0x07e HeaderValidateLength : 0x248
+0x080 HeaderValidateCopy : (null)
+0x084 NextAvailableTagIndex : 0
+0x086 MaximumTagIndex : 0
+0x088 TagEntries : (null)
+0x08c UCRList : _LIST_ENTRY [ 0xa55fe8 - 0xa55fe8 ]
+0x094 AlignRound : 0xf
+0x098 AlignMask : 0xfffffff8
+0x09c VirtualAllocdBlocks : _LIST_ENTRY [ 0xa4009c - 0xa4009c ]
+0x0a4 SegmentList : _LIST_ENTRY [ 0xa40010 - 0xa40010 ]
+0x0ac AllocatorBackTraceIndex : 0
+0x0b0 NonDedicatedListLength : 0
+0x0b4 BlocksIndex : 0x00a40260 Void
+0x0b8 UCRIndex : (null)
+0x0bc PseudoTagEntries : (null)
+0x0c0 FreeLists : _LIST_ENTRY [ 0xa4cd80 - 0xa53e70 ]
+0x0c8 LockVariable : 0x00a40248 _HEAP_LOCK
+0x0cc CommitRoutine : 0x4807219e long +4807219e
+0x0d0 FrontEndHeap : 0x003f0000 Void
+0x0d4 FrontHeapLockCount : 0
+0x0d6 FrontEndHeapType : 0x2 ''
+0x0d7 RequestedFrontEndHeapType : 0x2 ''
+0x0d8 FrontEndHeapUsageData : 0x00a43fe8 -> 0
+0x0dc FrontEndHeapMaximumIndex : 0x802
+0x0de FrontEndHeapStatusBitmap : [257] "???"
+0x1e0 Counters : _HEAP_COUNTERS
+0x23c TuningParameters : _HEAP_TUNING_PARAMETERS
```

The header starts with information about the segments associated with this heap.

Offset 0x4c (EncodeFlagMask) and 0x50 (Encoding) are used to store information about the chunk header encoding in the heap (= same offsets as in Windows 7).�� The actual key used to encode and decode the chunk header fields (XOR) is stored at offset 0x50.�� Fortunately the WinDBG !heap extension will perform all decoding for us.�� As you can imagine, this key is 1/ random for each process and 2/ different for each heap in the process, which means it is (still) quite effective at preventing heap header attacks.

Also, similar to Windows 7, the VirtualMemoryThreshold field (offset 0x5c) contains value 0xfe00.� As this value denotes the number of blocks, we need to multiply the value with 8 to get the actual number of bytes. (0x7F000 bytes).� In other words, we can still trigger VirtualAllocdBlocks by causing a regular (HeapAlloc) allocation of a size that is larger than this value.� Some of the heap spray techniques documented here and here were based on triggering VirtualAllocBlock chunks.� We'll see if the current implementation still allows for a precise heap spray.

Offset 0x0d6 (FrontEndHeapType) indicates what front-end allocator is being used.� Value 0x2 refers to LFH.�� The address of the FrontEndHeap "master" header structure can be found at the address referenced at offset 0xd0 (FrontEndHeap).� In the example above, the LFH header is stored at 0x003f0000 and contains the following information:

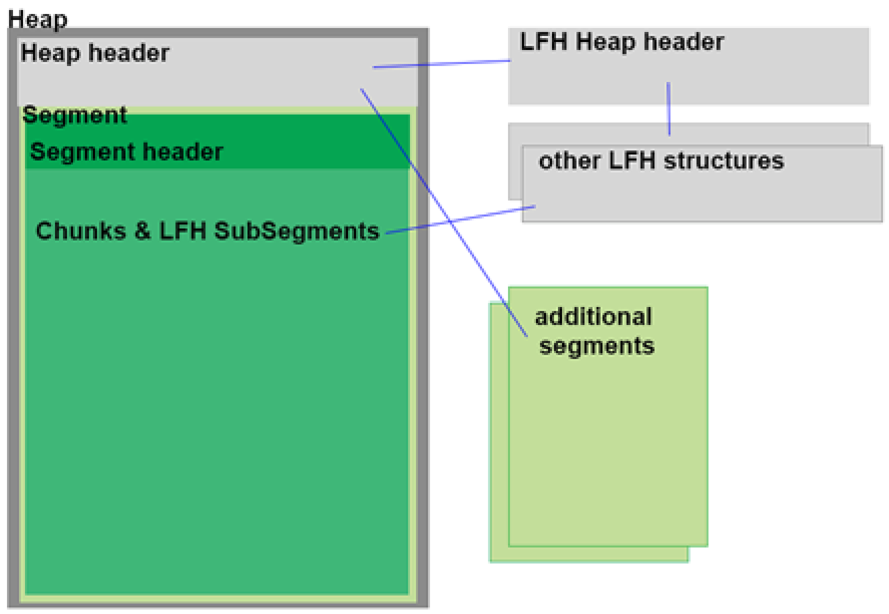
```
0:003> dt _LFH_HEAP 0x003f0000

ntdll!_LFH_HEAP
+0x000 Lock : _RTL_SRWLOCK
+0x004 SubSegmentZones : _LIST_ENTRY [ 0xa48910 - 0xa48910 ]
+0x00c Heap : 0x00a40000 Void
+0x010 NextSegmentInfoArrayAddress : 0x003f0a08 Void
+0x014 FirstUncommittedAddress : 0x003f1000 Void
+0x018 ReservedAddressLimit : 0x003f8000 Void
+0x01c SegmentCreate : 7
+0x020 SegmentDelete : 1
+0x024 MinimumCacheDepth : 0
+0x028 CacheShiftThreshold : 0
+0x02c SizeInCache : 0
+0x030 RunInfo : _HEAP_BUCKET_RUN_INFO
+0x038 UserBlockCache : [12] _USER_MEMORY_CACHE_ENTRY
+0x1b8 MemoryPolicies : _HEAP_LFH_MEM_POLICIES
+0x1bc Buckets : [129] _HEAP_BUCKET
+0x3c0 SegmentInfoArrays : [129] (null)
+0x5c4 AffinitizedInfoArrays : [129] (null)
+0x7c8 SegmentAllocator : (null)
+0x7d0 LocalData : [1] _HEAP_LOCAL_DATA
```

We can see references to various LFH terms, including "Bucket", "SubSegment" and "Heap local data".

Based on some quick tests, it looks like the LFH_HEAP header is (usually/always?) stored within the first segment of the heap on Windows 7, but on Windows 10 it seems to be stored outside of the memory range used by the first segment.

Putting things in a graphical (but very high-level and abstract) manner, the heap pretty much consists of the following building blocks:



The back-end allocator

BEA_Alloc1

On Windows 7, the back-end allocator (BEA) is the default/active mechanism used to manage freed chunks.� Based on my observations, this is still the case on Windows 10. It uses the chunks available inside the segment(s) and starts empty.� As soon as a chunk gets freed, it will "remember" these free chunks in some kind of list.� The free chunks are organized per size, and the mechanism uses a table to do so. Each entry in the table represents a size (increment of 8 bytes), and index 0 is used to manage chunks that are larger than the chunk size managed by index 127.

The FreeList table is now referenced at offset 0xc0 from the heap base.� I didn't check in detail, but I am assuming that the table stil consists of 128 elements, and that each entry is basically a Flink/Blink to the double linked list of free chunks managed in the table entry, or null pointers (to indicate there are no free chunks of the size that is managed by the corresponding table entry).� Again, I didn't verify in detail, as I'm more interested in how it behaves at this time.

In order to evaluate its behavior on Windows 10, we'll run some basic tests using a couple of example applications.� You can find the sourcecode and binary for this first test in the "BEA_Alloc1" folder in the github repository.

This first test application starts by allocating 2 chunks of 0x300 bytes each.� This should not be a common object size in my simple test application, and should be large enough to

- avoid that the back-end allocator already has chunks of that size on its freelist
- avoid that the back-end allocator already has chunks that are bigger that this size on its freelist
- avoid that the LFF is already active for the bucket that contains chunks of this size

(I don't really know if these 'conditions' will be met, but we'll find out)

In any case, the goal of these 2 allocations is to check where these 2 chunks will be allocated from (i.e. from a normal segment) and if they are placed next to each other.�� If that is the case, and if we free both chunks, we'll check if they get coalesced (merged together) by the BEA and managed in its free list.� Finally, if we then ask for a new allocation, let's see if the BEA is going to split the chunk and use it to satisfy new requests.

Let's go ahead & compile + run the binary.� The application will first print the address of the default process heap and will then wait for you to press return.

```
C:\Users\corelan\Desktop\vc++\win10\BEA_Alloc1\Release>BEA_Alloc1.exe

Default process heap found at 0x01550000
Press a key to start...
```

Attach WinDBG and check the state of the default process heap allocations at this point with !heap -p -h

.

As I am only interested in the "free" chunks at this point, I have removed all lines from the output that correspond with a "busy" chunk and/or are part of a LFH subsegment for a different bucket size.� (in case you're wondering how to recognize those: simply look for lines that are preceded by an asterisk (*), indicating a large busy/internal chunk, and followed by a list a smaller chunks of similar sizes within the address range of this larger busy/internal chunk.�� This larger one is the subsegment, the smaller ones are the LFH managed chunks inside that subsegment).

In other words, I'm only listing the chunks that are managed by the BEA and could potentially be used to satisfy the allocation requests caused by my test application.

```
0:003> !heap -p -h 0x01550000 _HEAP @ 1550000
_LFH_HEAP @ f80000
_HEAP_SEGMENT @ 1550000
CommittedRange @ 1550498
HEAP_ENTRY Size Prev Flags UserPtr UserSize - state

01551db0 0005 0045 [00] 01551db8 00020 - (free)

01552b00 0021 0012 [00] 01552b08 00100 - (free)
01552cb0 0015 0015 [00] 01552cb8 000a0 - (free)
015573a8 0008 000b [00] 015573b0 00038 - (free)
01557578 0002 0008 [00] 01557580 00008 - (free)
01559080 0003 0041 [00] 01559088 00010 - (free)
01559098 0003 0003 [00] 015590a0 00010 - (free)
015590e0 0003 0003 [00] 015590e8 00010 - (free)
01559110 0003 0003 [00] 01559118 00010 - (free)
01559128 0003 0003 [00] 01559130 00010 - (free)
01559140 0003 0003 [00] 01559148 00010 - (free)
01559158 0003 0003 [00] 01559160 00010 - (free)
01559170 0003 0003 [00] 01559178 00010 - (free)
01559188 0003 0003 [00] 01559190 00010 - (free)
015591a0 0003 0003 [00] 015591a8 00010 - (free)
015591d0 0003 0003 [00] 015591d8 00010 - (free)
015591e8 0003 0003 [00] 015591f0 00010 - (free)
01559200 0003 0003 [00] 01559208 00010 - (free)
01559218 0003 0003 [00] 01559220 00010 - (free)
01559230 0003 0003 [00] 01559238 00010 - (free)
0155d9b0 0018 000a [00] 0155d9b8 000b8 - (free)
0155dcc8 0012 000a [00] 0155dcd0 00088 - (free)
0155e538 002e 0021 [00] 0155e540 00168 - (free)
0155e6d0 0004 0081 [00] 0155e6d8 00018 - (free)
0155e6f0 0004 0004 [00] 0155e6f8 00018 - (free)
```

```
0155e710 0004 0004 [00] 0155e718 00018 - (free)
0155e730 0004 0004 [00] 0155e738 00018 - (free)
0155e750 0004 0004 [00] 0155e758 00018 - (free)
0155e770 0004 0004 [00] 0155e778 00018 - (free)
0155e790 0004 0004 [00] 0155e798 00018 - (free)
0155e7b0 0004 0004 [00] 0155e7b8 00018 - (free)
0155e7d0 0004 0004 [00] 0155e7d8 00018 - (free)
0155e7f0 0004 0004 [00] 0155e7f8 00018 - (free)
0155e810 0004 0004 [00] 0155e818 00018 - (free)
0155e830 0004 0004 [00] 0155e838 00018 - (free)
0155e890 0004 0004 [00] 0155e898 00018 - (free)
0155e990 0004 0004 [00] 0155e998 00018 - (free)
0155e9d0 0004 0004 [00] 0155e9d8 00018 - (free)
0155e9f0 0004 0004 [00] 0155e9f8 00018 - (free)
0155ea10 0004 0004 [00] 0155ea18 00018 - (free)
0155ea30 0004 0004 [00] 0155ea38 00018 - (free)
0155ea70 0004 0004 [00] 0155ea78 00018 - (free)
01561438 0175 0201 [00] 01561440 00ba0 - (free)
```

As you can see, there are no free chunks of exactly 0x300 bytes, however the last line in the output shows a free chunk of 0xba0 bytes (01561440 – 01561FE0).

This is quite normal, as this is the remaining space in the segment that has not been allocated yet. It shows up as a free chunk (because that's exactly what it is) and will be split to satisfy new allocation requests (= that's pretty much how the BEA works when it has to use a larger chunk to satisfy an allocation request).�

Since there are no free chunks of 0x300 bytes, a larger free chunk should be split (providing that the one in the list above is not being used by LFH… but as the flag is set to "free" and as it is not part of a larger subsegment, it shouldn't be related with LFH at all.� After all, a subsegment typically shows up as a larger chunk that is marked as "busy" and "internal").

Anyway, let's verify to be sure:

```
0:003> !heap -p -a 01561438

address 01561438 found in
_HEAP @ 1550000
HEAP_ENTRY

    Size Prev Flags     UserPtr  UserSize - state
01561438 0175 0000 [00] 01561440 00ba0 - (free)

0:003> !heap -x 01561438

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
01561438 01561440 01550000 01550000 ba8 1008 0 free
```

The "flags" indicate that this is not an LFH managed chunk.� (Otherwise, the output would have mentioned the word "LFH" as well)

Continue to run the application & see what happens:

```
Allocated chunk of 0x300 bytes at 0x01561440

Allocated chunk of 0x300 bytes at 0x01561748
Press return to continue
```

Based on the addresses returned by HeapAlloc, it looks like the final free chunk of 0xba0 bytes (at 0x1561440) was indeed split into pieces:

- *a chunk of 0x300 bytes, taken from the start of the larger free chunk, allocated at 0x01561440. The larger free chunk is now smaller, but still big enough to satisfy another request.�*
- *another piece of 0x300 bytes bytes, allocated at 0x01561748, taken from the (new) start of the larger free chunk.*

Both allocations sit next to each other, because allocations are taken from the start of a free chunk when splitting.� As both allocations are taken from the same larger chunk, it is expected from them to sit together. What is left of the original 0xba0 byte chunk is 0x590 bytes (as expected, showing up as a free chunk which starts at 01561a50 and ends at 01561FE0)

WinDBG: !heap -p -h

(output limited to the 3 chunks that are relevant to this exercise: the 2 "new" allocations, and the remaining space)

```
01561438 0061 0201 [00] 01561440 00300 - (busy)

01561740 0061 0061 [00] 01561748 00300 - (busy)

01561a48 00b3 0061 [00] 01561a50 00590 - (free)
```

Great, but we have not been really using the FreeList mechanisms of the BEA at this point.�� All we have been doing is allocate chunks from within a normal segment, consuming the free space that was there all the time.�� To trigger the BEA FreeList mechanism and see how it behaves, we'll have to "free" some chunks ourselves first.

BEA_Alloc2

In this second example, we'll create a series of allocations of 0x300 bytes.�� We'll free the last one and then cause 2 allocations of 0x100 bytes.�

The purpose of the exercise is to check where these 2 allocations will be placed.�

As we intend to evaluate the BEA mechanism, we have to avoid that the LFH gets triggered.� We know that the heap manager in Windows 7 will trigger the LFH when it sees 18 consecutive allocations of a size in the same bucket (i.e. the next request will be allocated from within a LFH managed subsegment).

For that reason, we'll only cause 10 allocations of 0x300 bytes, hopefully avoiding that the LFH will be used.

After causing the 10 allocations, let's examine the last one and see if LFH is on or off.

App:

```
C:\Users\corelan\Desktop\vc++\win10\BEA_Alloc2\Release>BEA_Alloc2.exe
Default process heap found at 0x009B0000
Press a key to start...
Allocated chunk of 0x300 bytes at 0x009BF1D0
Allocated chunk of 0x300 bytes at 0x009BF4D8
Allocated chunk of 0x300 bytes at 0x009BF7E0
Allocated chunk of 0x300 bytes at 0x009BFAE8
Allocated chunk of 0x300 bytes at 0x009BFDF0
Allocated chunk of 0x300 bytes at 0x009C00F8
Allocated chunk of 0x300 bytes at 0x009C0400
Allocated chunk of 0x300 bytes at 0x009C0708
Allocated chunk of 0x300 bytes at 0x009C0A10
Allocated chunk of 0x300 bytes at 0x009C0D18

Press return to continue
```

WinDBG:

```
0:003> !heap -x 0x009C0D18

Entry���� User����� Heap����� Segment������ Size� PrevSize� Unused��� Flags

-----------------------------------------------------------------------------

009c0d10� 009c0d18� 009b0000� 009b0000������ 308������ 308�������� 8� busy
```

So far so good, no sign of LFH at this point.�� Next, the application will free this last chunk:
App:

```
Free chunk at 0x009C0D18
Press return to continue
```

WinDBG:

```
0:003> !heap -x 0x009C0D18
Entry���� User����� Heap����� Segment������ Size� PrevSize� Unused��� Flags
-----------------------------------------------------------------------------
009c0d10� 009c0d18� 009b0000� 009b0000����� 12d0������ 308�������� 0� free
```

Before running the rest of the application, let's look at the current state of the heap, looking for "free" chunks of 0x300 bytes or larger.� The output below only contains the relevant lines:

WinDBG:

```
0:003> !heap -p -h 0x009B0000

_HEAP @ 9b0000

_LFH_HEAP @ 610000

_HEAP_SEGMENT @ 9b0000

CommittedRange @ 9b0498

HEAP_ENTRY Size Prev Flags��� UserPtr UserSize - state

009c0d10 025a 0061� [00]�� 009c0d18��� 012c8 - (free)
```

As we can see in the output, as the last chunk of 0x300 bytes was allocated right before the remaining free space in the segment, this now freed chunk got merged with the adjacent free space in the segment, and has now become a 0x12c8 byte free chunk. This larger free chunk starts at 009c0d10, which is also the start address of the 0x300 byte chunk that was freed.��
Our test application will now cause 2 allocations of 0x100 bytes each. Sure enough, this is large enough to satisfy 2 requests for 0x100 bytes, but in order to predict what exactly will happen, we need to look at the heap state again first.
The output of !heap -p -h shows the following lines that refer to free chunks of 0x100 bytes or more:
WinDBG:

```

0:003> !heap -p -h 0x009B0000
��� _HEAP @ 9b0000
����� _LFH_HEAP @ 610000
����� _HEAP_SEGMENT @ 9b0000
������ CommittedRange @ 9b0498
����� HEAP_ENTRY Size Prev Flags��� UserPtr UserSize - state
������� 009b2b00 0021 0012� [00]�� 009b2b08��� 00100 - (free)
������� 009bc330 0034 0089� [00]�� 009bc338��� 00198 - (free)
������� 009c0d10 025a 0061� [00]�� 009c0d18��� 012c8 - (free)
```

Based on the output, we can see

- *A free chunk of exactly 0x100 bytes (entry starts at 009b2b00, userptr is 009b2b08)�*
- *A free chunk of 0x198 bytes (entry starts at 009bc330, userptr is 009bc338)*
- *The remaining free space in the segment, at 009c0d10*

Let’s see what logic is applied by the BEA to reuse those freed chunks:
App:

```
Allocated chunk of 0x100 bytes at 0x009B2B08
Allocated chunk of 0x100 bytes at 0x009BC338
Done...
```

This makes sense.� The chunk that is exactly the size of the allocation request gets reused first.� Next, the next one that is larger gets split.� The remaining space near the end of the segment, which is now a combination of the original free space and the 0x300 byte chunk that was freed, does not get used.
The 2 chunks are not adjacent (despite the allocations happened right after each other).� None of the 2 chunks were able to take memory space back from one the 0x300 byte chunk that was freed earlier.
The behavior is absolutely normal, but this means that we cannot simply use the BEA to take a previously-allocated-and-now-freed chunk back… at least not without doing some "massaging".�
On the other hand, because the BEA does not keep chunks together based on buckets, and if we can improve control, we may be able to use a smaller or a larger allocation size to take a position back in memory that has been freed, regardless of the original size of that freed chunk.
Let’s try to increase control over what happens, increasing our chances that we can take a freed chunk back.

BEA_Alloc3

As demonstrated in the previous exercise, the fact that the BEA is managing freed chunks means that, by default, we may not be able to predict where a new allocation will be placed relative to another chunk or in relation with taking the space of a freed chunk (which would be useful in a Use After Free scenario).�
Let’s start with the latter scenario first.�� What does it take to take a freed chunk back?
The answer is quite simple: it depends on what else is free.��

A possible approach to control what else is free, is to simply reduce what is free at the time the free happens that you would like to take back. The idea is to exhaust the freelist as much as possible… and to do that, we need to cause allocations.
Additionally, we may want to check that the freed chunk (the one we're trying to take back… we'll call this one the 'vulnerable' freed chunk) does not sit after an already freed chunk.� This would cause both chunks to be merged, and subsequent allocation requests would start consuming the start of the new (bigger chunk).�
Depending on the size of the first part, it may be difficult to get perfect control over the contents.� On the other hand, if you can control the size of that first freed chunk (which will eventually get merged with the 'vulnerable' freed chunk), it might actually allow you to improve control over what bytes exactly you will manage if you don't control all bytes of the new allocation.�
Again, it all depends on the layout that you create, and your understanding of what is inside the free list at the time of the allocation that is supposed to take the place of the "vulnerable" freed chunk.
Anyway, let's look at an example.� BEA_Alloc3 contains the following scenario: we'll free a 0x58 byte chunk, and the goal is to use a 0x58 byte allocation that will take its position back
Obviously we have to make sure that LFH is never activated (as we are playing with the BEA), and that we can avoid that the 0x58 byte chunk gets merged with an adjacent free chunk.�� This means that we could try to surround the 0x58 byte chunk by 2 allocations that we created as well, but won't allow to be freed.��
In order for all 3 allocations to be adjacent, we have to make sure there is nothing on the free list already that will satisfy any of the 3 allocation requests.� In the example application, I am using 10 allocations for both sizes, but of course in reality, you'll need to check how many allocations you need to truly remove all free chunks; while being careful about avoiding LFH at the same time.
As the size of the 1st and 3rd allocation (the ones before and after the 0x58 byte chunk) is not so important, we can choose a size that is not on the freelist yet.� (Sizes are not important, because we don't have to take LFH bucket sizes into account. Again, we're playing with the BEA).� Anyway, I'll use 0x100 byte chunk to surround the 0x58 chunk.
Run the application, and check the heap after causing a series of allocations for 0x58 and 0x100 bytes, to exhaust the freelist for those sizes.
App:

```
C:\Users\corelan\Desktop\vc++\win10\BEA_Alloc3\Release>BEA_Alloc3.exe

Default process heap found at 0x016C0000
Press a key to start...

Allocated chunk of 0x58 bytes at 0x016C96D0
Allocated chunk of 0x100 bytes at 0x016C2B08
Allocated chunk of 0x58 bytes at 0x016CDF50
Allocated chunk of 0x100 bytes at 0x016CD328
Allocated chunk of 0x58 bytes at 0x016CD430
Allocated chunk of 0x100 bytes at 0x016CFFF8
Allocated chunk of 0x58 bytes at 0x016C2CB8
Allocated chunk of 0x100 bytes at 0x016D0100
Allocated chunk of 0x58 bytes at 0x016CDC18
Allocated chunk of 0x100 bytes at 0x016D0208
Allocated chunk of 0x58 bytes at 0x016CDC78
Allocated chunk of 0x100 bytes at 0x016D0310
Allocated chunk of 0x58 bytes at 0x016D0418
Allocated chunk of 0x100 bytes at 0x016D0478
Allocated chunk of 0x58 bytes at 0x016D0580
Allocated chunk of 0x100 bytes at 0x016D05E0
Allocated chunk of 0x58 bytes at 0x016D06E8
Allocated chunk of 0x100 bytes at 0x016D0748
Allocated chunk of 0x58 bytes at 0x016D0850
Allocated chunk of 0x100 bytes at 0x016D08B0

Press return to continue
```

WinDBG shows that the last 2 allocations are not part of LFH (which means LFH is not active yet).�� We can also see that these allocations are taken from the large free chunk at the end of the segment (thus no longer taking already freed chunks that might have been part of the freelist)

```
0:003> !heap -x 0x016D08B0

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
016d08a8 016d08b0 016c0000 016c0000 108 60 8 busy


0:003> !heap -x 0x016D0850

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
016d0848 016d0850 016c0000 016c0000 60 108 8 busy
```

Next, the application will cause the necessary layout.� As a segment doesn't necessarily want to keep chunks of the same bucket size together, and operates in a first ask, first serve manner (unlike the LFH), the 3 allocations should be placed right next to each other:
App:

```
Allocated chunk of 0x100 bytes at 0x016D09B8
Allocated chunk of 0x58 bytes at 0x016D0AC0, filled with 'A'
Allocated chunk of 0x100 bytes at 0x016D0B20

Press return to continue
```

Great! We can use WinDBG to confirm that the 3 chunks are adjacent indeed, and that the middle chunk (0x58) is filled with A's

(from output of !heap -p -h 0x016C0000)

```
016d09b0 0021 0021 [00] 016d09b8 00100 - (busy)
016d0ab8 000c 0021 [00] 016d0ac0 00058 - (busy)
016d0b18 0021 000c [00] 016d0b20 00100 - (busy)

0:001> dd 0x016d0ac0 L 0x58/4
016d0ac0 41414141 41414141 41414141 41414141
016d0ad0 41414141 41414141 41414141 41414141
016d0ae0 41414141 41414141 41414141 41414141
016d0af0 41414141 41414141 41414141 41414141
016d0b00 41414141 41414141 41414141 41414141
016d0b10 41414141 41414141
```

Next, the 0x58 byte chunk will be freed (which means it should end up on the freelist, and not merged with an adjacent free chunk as long as we don't free the adjacent 0x100 byte chunks ourselves).� Finally, a new allocation request for 0x58 is supposed to take that position back.� The example application will populate it with B's:

App:

```
Free chunk of 0x58 bytes at 0x016D0AC0
Allocated chunk of 0x58 bytes at 0x016D0AC0, filled with 'B'

Done...
```

WinDBG:

```
0:001> dd 0x016d0ac0 L 0x58/4
016d0ac0 42424242 42424242 42424242 42424242
016d0ad0 42424242 42424242 42424242 42424242
016d0ae0 42424242 42424242 42424242 42424242
016d0af0 42424242 42424242 42424242 42424242
016d0b00 42424242 42424242 42424242 42424242
016d0b10 42424242 42424242
```

Bingo.

BEA_Alloc4

Let’s take it one step further.

This time we’ll free a 0x58 byte chunk, and we’ll try to use a 0x80 byte allocation (where we only control the last 4 dwords), to control the exact 4 first dwords in the original 0x58 byte chunk. (I know, replacing one object with another object of a different size sounds kinky… but hey, who knows this could be useful one time … or not)

The approach is pretty much the same: we’ll start by (trying) to exhaust the freelist for the sizes that we are going to use, to make sure we have better control over what will be returned when we ask for a chunk of those sizes.�
What is different in this case, is the layout we need to allow magic to happen – i.e. to take the first 4 dwords of the 0x58 byte chunk using the last 4 dwords of the 0x80 byte allocation.
The idea is to try to force 2 freed chunks to merge (by placing them next to each other and then freeing both).�
The so-called "vulnerable" object that gets freed in our imaginary "use after free" scenario is 0x58 bytes, and the goal is to to control the first 4 dwords.� The object we can use to replace it with is 0x80 bytes, and we’ll pretend we can only control the last 4 dwords of that one.� In other words, we have to make sure that we position those 4 dwords with surgical precision, making sure that the "overlap" will happen in the right place.��
I generally suck at math, but after spending some time on the abacus, finger counting and scratching my head, I came up with the complex formula that results in trying to position a 0x80-4-4-4-4-8 = 0x68 byte object before the 0x58 byte object.� The 4 times 4 bytes are needed to cause overlap, and the additional 8 bytes are there to compensate for the chunk header of the 0x58 byte chunk that sits before the contents of the 0x58 byte chunk, and is thus also consuming space in memory.��
This means that we also have to make sure to remove 0x68 byte chunks from the free list in order to increase control over allocations of that size.�
We’ll create a layout that consists of 4 allocations:� our ‘magic’ 0x68 and 0x58 objects in the middle, surrounded by 2 other objects, to avoid that a merge of one of the 2 in the middle will cause us to lose control when things start to merge with the wrong objects. We’ll also cause a little bit more allocations to try to exhaust the heap, as some of these sizes may be popular sizes.�
(Again, the number of allocations you need to properly clear out the free list depends on the applicaiton, the context, basically on what is already free at that time)
Let’s see what it looks like:

App:

```
C:\Users\corelan\Desktop\vc++\win10\BEA_Alloc4\Release>BEA_Alloc4.exe

Default process heap found at 0x010D0000

Press a key to start...

<...snip...>

Allocated chunk of 0x68 bytes at 0x010E1D68
Allocated chunk of 0x58 bytes at 0x010E1DD8
Allocated chunk of 0x80 bytes at 0x010E1E38
Allocated chunk of 0x68 bytes at 0x010E1EC0
Allocated chunk of 0x58 bytes at 0x010E1F30
Allocated chunk of 0x80 bytes at 0x010E1F90
Allocated chunk of 0x68 bytes at 0x010E2018


Press return to continue
```

(Check the last 3 chunks in WinDBG (using !heap -x
) to make sure that none of them is LFH managed at this point.)
Next, the application creates the initial layout (0x80 // 0x68 // 0x58 // 0x80).� The first and last one are just there to prevent the 0x68 and 0x58 sized chunks to accidentally merge with the wrong one.
App:

```
Allocated start chunk (0x80 bytes) at 0x010E2088
Allocated first chunk (0x68 bytes) at 0x010E2110
Allocated second 'vulnerable' chunk (0x58 bytes) at 0x010E2180, filled with 'A'
Allocated end chunk (0x80 bytes) at 0x010E21E0

Press return to continue
```

WinDBG (output from !heap -p -h
):

```
������� 010e2080 0011 000e� [00]�� 010e2088��� 00080 - (busy)
������� 010e2108 000e 0011� [00]�� 010e2110��� 00068 - (busy)
������� 010e2178 000c 000e� [00]�� 010e2180��� 00058 - (busy)
������� 010e21d8 0011 000c� [00]�� 010e21e0��� 00080 - (busy)
```

Sure enough, the 4 chunks are in the right place.� When the 'vulnerable' 0x58 byte chunk gets freed, we'll also cause the first one to free ourselves.� This should merge the 2 together.
App:

```
Free chunk of 0x58 bytes at 0x010E2180
Free first chunk of 0x68 bytes at 0x010E2110

Press return to continue
```

WinDBG shows that we now have one big free chunk at 0x010E2110, of 0xc8 bytes.� If we dump it, we see the original 0x68 chunk followed by the vulnerable 0x58 byte chunk.

```
 0:003> !heap -p -a 0x010E2110
��� address 010e2110 found in
��� _HEAP @ 10d0000
����� HEAP_ENTRY Size Prev Flags��� UserPtr UserSize - state
������� 010e2108 001a 0000� [00]�� 010e2110��� 000c8 - (free)

0:003> dd 0x010E2110 L 0xc8/4
010e2110� 010e2268 010de858 00000000 00000000
010e2120� 00000000 00000000 00000000 00000000
010e2130� 00000000 00000000 00000000 00000000
010e2140� 00000000 00000000 00000000 00000000
010e2150� 00000000 00000000 00000000 00000000
010e2160� 00000000 00000000 00000000 00000000
010e2170� 00000000 00000000 0c00000c 00008812
010e2180� 010e2268 010de858 41414141 41414141
010e2190� 41414141 41414141 41414141 41414141
010e21a0� 41414141 41414141 41414141 41414141
010e21b0� 41414141 41414141 41414141 41414141
010e21c0� 41414141 41414141 41414141 41414141
010e21d0� 41414141 41414141
```

Finally, our 0x80 byte allocation is now supposed to take the first 0x80 bytes of this free chunk, overwriting the first 4 dwords inside the chunk with A's with the last 4 dwords of the 0x80 byte chunk.
App:

```

Allocated chunk of 0x80 bytes at 0x010E2110, filled with 'B'

Done...
```

WinDBG:

```

0:001> !heap -p -a 0x010E2110
��� address 010e2110 found in
��� _HEAP @ 10d0000
����� HEAP_ENTRY Size Prev Flags��� UserPtr UserSize - state
������� 010e2108 0011 0000� [00]�� 010e2110��� 00080 - (busy)


0:001> dd 0x010E2110 L 0xc8/4
010e2110� 42424242 42424242 42424242 42424242
010e2120� 42424242 42424242 42424242 42424242
010e2130� 42424242 42424242 42424242 42424242
010e2140� 42424242 42424242 42424242 42424242
010e2150� 42424242 42424242 42424242 42424242
010e2160� 42424242 42424242 42424242 42424242
010e2170� 42424242 42424242 42424242 42424242
010e2180� 42424242 42424242 42424242 42424242
010e2190� 6a309389 0000880d 010dee10 010d1db8
010e21a0� 41414141 41414141 41414141 41414141
010e21b0� 41414141 41414141 41414141 41414141
010e21c0� 41414141 41414141 41414141 41414141
010e21d0� 41414141 41414141
```

Damn, it looks like we missed our target with 8 bytes (as I said, I suck at math… but perhaps I should have used a 0x60 byte chunk instead of 0x68…)� anyway I'm sure you get the point…�
Nice – but all of this was only possible because none of the sizes involved were managed by the LFH already, and because we were able to avoid the LFH from being triggered.�
Perhaps this is the right time to look at the LFH under WIndows 10 and see how it behaves.� After all, we may not always have the luxury of avoiding the LFH in the first place.

The Front-End Allocator – LFH

LFH_Alloc1

In this first exercise, we'll examine if it still takes 0x12 (18) consecutive allocations for a size in the same bucket before the LFH will start taking care of allocations and frees of those sizes.

In order to avoid any influencing or assumptions, I'll use chunksizes that has not been allocated yet. (0x1500 bytes, 0x2100 bytes, 0x3000 bytes, 0x800 bytes)

I'll combine a couple of tests in this test:

1. *check how many allocations it takes before LFH kicks in*
2. *see if the LFH trigger is influenced when an allocation happens of a different bucket size during the series of allocations*
3. *see if the LFH trigger is influenced when a free happens during the allocations, of a chunk in a different bucket/of a different size*
4. *see if the LFH trigger is influenced when a chunk of the same bucket is freed again, during the series of allocations*

<u>Step1: how many allocations are needed to trigger LFH</u>

App:

```
C:\Users\corelan\Desktop\vc++\win10\LFH_Alloc1\Release>LFH_Alloc1.exe
Default process heap found at 0x00D30000 Press a key to start...


[1] Allocated chunk of 0x1500 bytes at 0x00D4CF68
[2] Allocated chunk of 0x1500 bytes at 0x00D4E470
[3] Allocated chunk of 0x1500 bytes at 0x00D4F978
[4] Allocated chunk of 0x1500 bytes at 0x00D50E80
[5] Allocated chunk of 0x1500 bytes at 0x00D52388
[6] Allocated chunk of 0x1500 bytes at 0x00D53890
[7] Allocated chunk of 0x1500 bytes at 0x00D54D98
[8] Allocated chunk of 0x1500 bytes at 0x00D562A0
[9] Allocated chunk of 0x1500 bytes at 0x00D577A8
[10] Allocated chunk of 0x1500 bytes at 0x00D58CB0
[11] Allocated chunk of 0x1500 bytes at 0x00D5A1B8
[12] Allocated chunk of 0x1500 bytes at 0x00D5B6C0
[13] Allocated chunk of 0x1500 bytes at 0x00D5CBC8
[14] Allocated chunk of 0x1500 bytes at 0x00D5E0D0
[15] Allocated chunk of 0x1500 bytes at 0x00D5F5D8
[16] Allocated chunk of 0x1500 bytes at 0x00D60AE0
[17] Allocated chunk of 0x1500 bytes at 0x00D61FE8
[18] Allocated chunk of 0x1500 bytes at 0x00D6B348
[19] Allocated chunk of 0x1500 bytes at 0x00D64A20
[20] Allocated chunk of 0x1500 bytes at 0x00D69E40
[21] Allocated chunk of 0x1500 bytes at 0x00D6C850
[22] Allocated chunk of 0x1500 bytes at 0x00D63518
```

```
[23] Allocated chunk of 0x1500 bytes at 0x00D6DD58
[24] Allocated chunk of 0x1500 bytes at 0x00D67430
[25] Allocated chunk of 0x1500 bytes at 0x00D6F260
[26] Allocated chunk of 0x1500 bytes at 0x00D65F28
[27] Allocated chunk of 0x1500 bytes at 0x00D70768
[28] Allocated chunk of 0x1500 bytes at 0x00D68938
[29] Allocated chunk of 0x1500 bytes at 0x00D71C70
[30] Allocated chunk of 0x1500 bytes at 0x00D77438
Press return to continue
```

If you pay close attention to the addresses, you can see a bigger gap between allocations 17 and 18.� This is a good indication that allocations are no longer individual chunks inside a normal segment, but are now being consumed from an LFH subsegment.� To be sure, let's validate the findings in WinDBG.

Simply run !heap -x� on all addresses (starting from the first one in the list), until you find the first one that has the "LFH" marker.

Allocation 17:

```
0:001> !heap -x 0x00D61FE8

Entry User Heap Segment Size PrevSize Unused Flags

-----------------------------------------------------------------------------

00d61fe0 00d61fe8 00d30000 00d30000 1508 1508 8 busy
```

Allocation 18:

```
0:001> !heap -x 0x00D6B348

Entry User Heap Segment Size PrevSize Unused Flags

-----------------------------------------------------------------------------

00d6b340 00d6b348 00d30000 00d3ae28 1508 - 8 LFH;busy
```

So – it looks like the LFH takes over with allocation 18.

<u>Step 2: will the LFH trigger be influenced if another allocation (of a size in a different bucket) occurs in the series of 18 allocations.</u>

App:

```
[1] Allocated chunk of 0x2100 bytes at 0x00D93500
[2] Allocated chunk of 0x2100 bytes at 0x00D95608
[3] Allocated chunk of 0x2100 bytes at 0x00D97710
[4] Allocated chunk of 0x2100 bytes at 0x00D99818
[5] Allocated chunk of 0x2100 bytes at 0x00D9B920
[6] Allocated chunk of 0x2100 bytes at 0x00D9DA28
[7] Allocated chunk of 0x2100 bytes at 0x00D9FB30
[8] Allocated chunk of 0x2100 bytes at 0x00DA1C38
[9] Allocated chunk of 0x2100 bytes at 0x00DA3D40
[10] Allocated chunk of 0x2100 bytes at 0x00DA5E48
Allocated chunk of 0x300 bytes at 0x00D49450
```

```
[11] Allocated chunk of 0x2100 bytes at 0x00DA7F50
[12] Allocated chunk of 0x2100 bytes at 0x00DAA058
[13] Allocated chunk of 0x2100 bytes at 0x00DAC160
[14] Allocated chunk of 0x2100 bytes at 0x00DAE268
[15] Allocated chunk of 0x2100 bytes at 0x00DB0370
[16] Allocated chunk of 0x2100 bytes at 0x00DB2478
[17] Allocated chunk of 0x2100 bytes at 0x00DB4580
[18] Allocated chunk of 0x2100 bytes at 0x00DC10D8
[19] Allocated chunk of 0x2100 bytes at 0x00DBAAC0
[20] Allocated chunk of 0x2100 bytes at 0x00DC32E0
Press return to continue
```

Focusing on allocation 17 and 18 (of 0x2100 bytes), we can see that alloc 17 was not LFH yet, but the 18th one did it again, despite the allocation of a 0x300 byte chunk in the middle of the series.

```
0:001> !heap -x 0x00DB4580

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00db4578 00db4580 00d30000 00d30000 2108 2108 8 busy



0:001> !heap -x 0x00DC10D8

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00dc10d0 00dc10d8 00d30000 00d3ae78 2208 - 3f LFH;busy
```

Interesting indeed :)

Step 3: will a free (of a different chunk size) influence the LFH trigger ?

App:

```
[1] Allocated chunk of 0x3000 bytes at 0x00DD6690

[2] Allocated chunk of 0x3000 bytes at 0x00DD9698
[3] Allocated chunk of 0x3000 bytes at 0x00DDC6A0
[4] Allocated chunk of 0x3000 bytes at 0x00DDF6A8
[5] Allocated chunk of 0x3000 bytes at 0x00DE26B0
[6] Allocated chunk of 0x3000 bytes at 0x00DE56B8
[7] Allocated chunk of 0x3000 bytes at 0x00DE86C0
[8] Allocated chunk of 0x3000 bytes at 0x00DEB6C8
[9] Allocated chunk of 0x3000 bytes at 0x00DEE6D0
[10] Allocated chunk of 0x3000 bytes at 0x00DF16D8
Freed chunk at 0x00D49C80
Freed chunk at 0x00D4A188
[11] Allocated chunk of 0x3000 bytes at 0x00DF46E0
[12] Allocated chunk of 0x3000 bytes at 0x00DF76E8
[13] Allocated chunk of 0x3000 bytes at 0x00DFA6F0
[14] Allocated chunk of 0x3000 bytes at 0x00DFD6F8
[15] Allocated chunk of 0x3000 bytes at 0x00E00700
[16] Allocated chunk of 0x3000 bytes at 0x00E03708
[17] Allocated chunk of 0x3000 bytes at 0x00E06710
[18] Allocated chunk of 0x3000 bytes at 0x00E0C748
[19] Allocated chunk of 0x3000 bytes at 0x00E15760
[20] Allocated chunk of 0x3000 bytes at 0x00E1B770
Done...
```

WinDBG (focus on allocation 17 and 18 again):

```
0:001> !heap -x 0x00E06710

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00e06708 00e06710 00d30000 00d30000 3008 3008 8 busy



0:001> !heap -x 0x00E0C748

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00e0c740 00e0c748 00d30000 00d3aea0 3008 - 8 LFH;busy
```

Interesting once more:)

<u>Step 4: free a chunk from the same bucket during the allocation series.</u>

```
[1] Allocated chunk of 0x800 bytes at 0x004EAE70

[2] Allocated chunk of 0x800 bytes at 0x0053E910
[3] Allocated chunk of 0x800 bytes at 0x0053F118
[4] Allocated chunk of 0x800 bytes at 0x0053F920
[5] Allocated chunk of 0x800 bytes at 0x00540128
[6] Allocated chunk of 0x800 bytes at 0x00540930
[7] Allocated chunk of 0x800 bytes at 0x00541138
[8] Allocated chunk of 0x800 bytes at 0x00541940
[9] Allocated chunk of 0x800 bytes at 0x00542148
[10] Allocated chunk of 0x800 bytes at 0x00542950
Freed chunk at 0x00542950
[11] Allocated chunk of 0x800 bytes at 0x00542950
[12] Allocated chunk of 0x800 bytes at 0x00543158
[13] Allocated chunk of 0x800 bytes at 0x00543960
[14] Allocated chunk of 0x800 bytes at 0x00544168
[15] Allocated chunk of 0x800 bytes at 0x00544970
[16] Allocated chunk of 0x800 bytes at 0x00545178
[17] Allocated chunk of 0x800 bytes at 0x00545980
[18] Allocated chunk of 0x800 bytes at 0x009F0070
[19] Allocated chunk of 0x800 bytes at 0x009F2090
[20] Allocated chunk of 0x800 bytes at 0x009F68D8
Done...
```

WinDBG (looking again at allocation 17 and 18)

```
0:004> !heap -x 0x00545980

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00545978 00545980 00450000 00450000 808 808 8 busy



0:004> !heap -x 0x009F0070
```

```
Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
009f0068 009f0070 00450000 00458c60 808 - 8 LFH;busy
```

Interestingly enough, the free did not really impact the LFH trigger at all.� 18 allocations still did the trick.

I guess more exhaustive testing is needed to confirm this behaviour, including checking what happens if there are more frees etc… but at least we are able to see some kind of pattern that indicates it may be more difficult to avoid that the LFH will be get enabled eventually, especially if you have no other option that to cause a certain number of allocations (more than 18) of chunks in the same bucket.�

If you know of a way to prevent this from happening, please let me know :)

LFH_Alloc2

In the second exercise, we'll see if the LFH still behaves the same way as under Windows 7 – i.e. returning freed chunks in a LIFO manner.�� We'll activate the LFH using 20 allocations of 0x500 bytes.� The last one will be freed, and then another allocation of 0x500 bytes will happen.

The goal is to see if the last one to be freed will be the first one to be returned again. (LIFO).

```
C:\Users\corelan\Desktop\vc++\win10\LFH_Alloc2\Release>LFH_Alloc2.exe

Default process heap found at 0x008D0000
Press a key to start...

[1] Allocated chunk of 0x500 bytes at 0x008E0440

[2] Allocated chunk of 0x500 bytes at 0x008E0948
[3] Allocated chunk of 0x500 bytes at 0x008E0E50
[4] Allocated chunk of 0x500 bytes at 0x008E1358
[5] Allocated chunk of 0x500 bytes at 0x008E1860
[6] Allocated chunk of 0x500 bytes at 0x008E1D68
[7] Allocated chunk of 0x500 bytes at 0x008E2270
[8] Allocated chunk of 0x500 bytes at 0x008E2778
[9] Allocated chunk of 0x500 bytes at 0x008E2C80
[10] Allocated chunk of 0x500 bytes at 0x008E3188
[11] Allocated chunk of 0x500 bytes at 0x008E3690
[12] Allocated chunk of 0x500 bytes at 0x008E3B98
[13] Allocated chunk of 0x500 bytes at 0x008E40A0
[14] Allocated chunk of 0x500 bytes at 0x008E45A8
[15] Allocated chunk of 0x500 bytes at 0x008E4AB0
[16] Allocated chunk of 0x500 bytes at 0x008E4FB8
[17] Allocated chunk of 0x500 bytes at 0x008E54C0
[18] Allocated chunk of 0x500 bytes at 0x008E7D28
[19] Allocated chunk of 0x500 bytes at 0x008E5EF8
[20] Allocated chunk of 0x500 bytes at 0x008E6E10
Press return to continue

Freed chunk at 0x008E6E10

Press return to continue

Allocated chunk of 0x500 bytes at 0x008E9148

Press return to continue
```

In WinDBG, we can see that the LFH was indeed activated, but it also looks like the LIFO is behavior from Windows 7 is no longer there.

```
0:003> !heap -x 0x008E6E10
Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
008e6e08 008e6e10 008d0000 008d8d90 508 - 0 LFH;free



0:003> !heap -x 0x008E9148
Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
008e9140 008e9148 008d0000 008d8d90 508 - 8 LFH;busy
```

Perhaps causing a series of allocations of 0x500 byte will allow us to take the freed chunk back.� Press return to cause another 20 allocations of 0x500 bytes and see what happens:

```
Allocated chunk of 0x500 bytes at 0x008E59F0

Allocated chunk of 0x500 bytes at 0x008E7318
Allocated chunk of 0x500 bytes at 0x008E8738
Allocated chunk of 0x500 bytes at 0x008E6400
Allocated chunk of 0x500 bytes at 0x008E8C40
Allocated chunk of 0x500 bytes at 0x008E6908
Allocated chunk of 0x500 bytes at 0x008E7820
Allocated chunk of 0x500 bytes at 0x008E8230
Allocated chunk of 0x500 bytes at 0x008E6E10
--- got it back ---
Allocated chunk of 0x500 bytes at 0x008EC740
Allocated chunk of 0x500 bytes at 0x008EEA78
Allocated chunk of 0x500 bytes at 0x008EBD30
Allocated chunk of 0x500 bytes at 0x008EC238
Allocated chunk of 0x500 bytes at 0x008EDB60
Allocated chunk of 0x500 bytes at 0x008EF488
Allocated chunk of 0x500 bytes at 0x008ECC48
Allocated chunk of 0x500 bytes at 0x008F0DB0
Allocated chunk of 0x500 bytes at 0x008E99F8
Allocated chunk of 0x500 bytes at 0x008EEF80
Allocated chunk of 0x500 bytes at 0x008EB828
Press return to continue
```

In this case, it took another 9 allocations to get the freed chunk back.� In fact, if you'd run the same application a couple of times, you'll see that the number of times it takes to get the freed chunk back, varies largely between 0 (sometimes you'll get it back LIFO style) and never (at least, not in the first 20 allocations or so)… but in most cases I got it back within the first 10 allocations.� (A lot more structured testing would be needed to find the sweet spot that would provide some sort of predictability. It'll probably never be 100% reliable, but it may not be too messy either.)

Update (7/7/2016) – I added "LFH_TakeBack" to the github repository, which will automate some statistic gathering.� For each chunksize between 8 and 0x4000, it will enable LFH, alloc a chunk and free it again, and then measure how many allocations are needed to take it back.� The application calculates an average, a minimum and maximum number of tries, and also keeps track how many times the object was not taken back within the first 2000 allocations.

After running the app, it looks like the maximum number of allocations needed sits around 50.

Anyway, looking at the addresses of the allocations, it also looks like the chunks are no longer adjacent (as they were in Windows 7, at least as long as the chunks are inside the same subsegment).

This will certainly make it more complex to create a specific layout/sequence of objects when the LFH is enabled.

LFH_Alloc3

Is LFH still limited to 0x4000 byte chunks (max)?

App:

```
C:\Users\corelan\Desktop\vc++\win10\LFH_Alloc3\Release>LFH_Alloc3.exe

Default process heap found at 0x00930000
Press a key to start...

[1] Allocated chunk of 0x4000 bytes at 0x00940FF8

[2] Allocated chunk of 0x4000 bytes at 0x00945000
[3] Allocated chunk of 0x4000 bytes at 0x00949008
[4] Allocated chunk of 0x4000 bytes at 0x0094D010
[5] Allocated chunk of 0x4000 bytes at 0x00951018
[6] Allocated chunk of 0x4000 bytes at 0x00955020
[7] Allocated chunk of 0x4000 bytes at 0x00959028
[8] Allocated chunk of 0x4000 bytes at 0x0095D030
[9] Allocated chunk of 0x4000 bytes at 0x00961038
[10] Allocated chunk of 0x4000 bytes at 0x00965040
[11] Allocated chunk of 0x4000 bytes at 0x00969048
[12] Allocated chunk of 0x4000 bytes at 0x0096D050
[13] Allocated chunk of 0x4000 bytes at 0x00971058
[14] Allocated chunk of 0x4000 bytes at 0x00975060
[15] Allocated chunk of 0x4000 bytes at 0x00979068
[16] Allocated chunk of 0x4000 bytes at 0x0097D070
[17] Allocated chunk of 0x4000 bytes at 0x00981078
[18] Allocated chunk of 0x4000 bytes at 0x0098D0B8
[19] Allocated chunk of 0x4000 bytes at 0x009950C8
[20] Allocated chunk of 0x4000 bytes at 0x009910C0
Press return to continue



[1] Allocated chunk of 0x4008 bytes at 0x009C7008

[2] Allocated chunk of 0x4008 bytes at 0x009CB018
[3] Allocated chunk of 0x4008 bytes at 0x009CF028
[4] Allocated chunk of 0x4008 bytes at 0x009D3038
[5] Allocated chunk of 0x4008 bytes at 0x009D7048
[6] Allocated chunk of 0x4008 bytes at 0x009DB058
[7] Allocated chunk of 0x4008 bytes at 0x009DF068
[8] Allocated chunk of 0x4008 bytes at 0x009E3078
[9] Allocated chunk of 0x4008 bytes at 0x009E7088
[10] Allocated chunk of 0x4008 bytes at 0x009EB098
[11] Allocated chunk of 0x4008 bytes at 0x009EF0A8
[12] Allocated chunk of 0x4008 bytes at 0x009F30B8
[13] Allocated chunk of 0x4008 bytes at 0x009F70C8
[14] Allocated chunk of 0x4008 bytes at 0x009FB0D8
[15] Allocated chunk of 0x4008 bytes at 0x009FF0E8
[16] Allocated chunk of 0x4008 bytes at 0x00A030F8
[17] Allocated chunk of 0x4008 bytes at 0x00A07108
```

```
[18] Allocated chunk of 0x4008 bytes at 0x00A0B118
[19] Allocated chunk of 0x4008 bytes at 0x00A0F128
[20] Allocated chunk of 0x4008 bytes at 0x00A13138
Press return to continue
```

WinDBG: 0x4000 bytes

```
0:001> !heap -x 0x00981078

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00981070 00981078 00930000 00930000 4008 4008 8 busy



0:001> !heap -x 0x0098D0B8

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
0098d0b0 0098d0b8 00930000 00938c48 4008 - 8 LFH;busy
```

WinDBG: 0x4008 bytes

```
0:001> !heap -x 0x00A07108

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00a07100 00a07108 00930000 00930000 4010 4010 8 busy


0:001> !heap -x 0x00A0B118

Entry User Heap Segment Size PrevSize Unused Flags
-----------------------------------------------------------------------------
00a0b110 00a0b118 00930000 00930000 4010 4010 8 busy
```

Answer:� YES, 0x4000 seems to be the maximum size (just like on Windows 7)

LFH_TakeBack2

The LFH_Alloc exercises demonstrate that chunks are no longer allocated in an consecutive manner inside a LFH subsegment. Of course, this complicates creating a specific layout within the subsegment.�� I still wanted to know if it would be possible to replace the memory space used by a LFH chunk by an LFH allocation of a size from a different bucket. As an example, can I take the space of a 0x58 byte chunk using a 0x88 byte allocation, within the LFH.*

As the LFH subsegments are used to keep chunks of the same bucket size together, this would require clearing out the entire subsegment used for storing the vulnerable object, and hopefully the Heap Manager will reuse those pages for another subsegment (allocations for a different bucket size).

LFH_TakeBack2 demonstrates if it works or not.�� The idea is to try to place the "vulnerable" object in a subsegment where you control all the other chunks.� As soon as the vulnerable object gets freed, you cause all the other chunks to be freed as well.� Hopefully, this will also release the entire subsegment and its pages, so they can be reused again (even for a subsegment associated with chunksizes that fall in a different bucket).

App:

```
C:\Users\corelan\Desktop\vc++\win10_heap\LFH_TakeBack2\Release>LFH_TakeBack2.exe

Vulnerable object of 0x00000058 bytes at 0x01136B98, filled with 'A'

Allocations done. Press return to start free process
```

WinDBG:

```
0:003> dd 0x01136B98
01136b98� 41414141 41414141 41414141 41414141
01136ba8� 41414141 41414141 41414141 41414141
01136bb8� 41414141 41414141 41414141 41414141
01136bc8� 41414141 41414141 41414141 41414141
01136bd8� 41414141 41414141 41414141 41414141
01136be8� 41414141 41414141 07fe9621 88011d00
01136bf8� 00000000 00000000 00000000 00000000
01136c08� 00000000 00000000 00000000 00000000


0:003> !heap -p -a 0x01136B98
��� address 01136b98 found in
��� _HEAP @ e30000
����� HEAP_ENTRY Size Prev Flags��� UserPtr UserSize - state
������� 01136b90 000c 0000� [00]�� 01136b98��� 00058 - (busy)

�
0:003> !heap -x 0x01136B98
Entry���� User����� Heap����� Segment������ Size� PrevSize� Unused��� Flags
-----------------------------------------------------------------------------
01136b90� 01136b98� 00e30000� 00e45638������� 60������ -����������� 8� LFH;busy
```

Continue with App:

```
Vulnerable object at 0x01136B98 was freed
Free done. Press return to start new allocations (size 0x00000088)

Allocations done.�� Check if 0x01136B98 contains 'B' now
```

WinDBG:

```
0:001> dd 0x01136B98
01136b98� 42424242 42424242 42424242 42424242
01136ba8� 42424242 42424242 42424242 42424242
01136bb8� 42424242 42424242 42424242 42424242
01136bc8� 42424242 42424242 42424242 42424242
01136bd8� 42424242 42424242 42424242 42424242
01136be8� 42424242 42424242 42424242 42424242
01136bf8� 42424242 42424242 42424242 42424242
01136c08� 42424242 42424242 07f296dd 8800bf00
```

```
0:001> !heap -x 0x01136B98
Entry���� User����� Heap����� Segment������ Size� PrevSize� Unused��� Flags
-----------------------------------------------------------------------------
01136b80� 01136b88� 00e30000� 00e38ca0������� 90������ -����������� 8� LFH;busy
```

Size is now 0x90.�� So, the mechanism still works (just like in older Windows versions). Of course, because the sequence of chunks inside a segment is not fully under our control, it will be difficult to control specific bytes of the freed object when you replace it with a different sized chunk.��

Large chunks

Large_Alloc1

What do virtualallocdblock chunks/allocations look like under WIndows 10?

In order to create such chunks, we need to cause HeapAlloc/RtlAllocateHeap allocations of a size that is larger than the VirtualMemoryThreshold value in the heap header.� In the example application, I am triggering 20 allocations of 0x7ffb0 bytes (which is larger than the 7ff00 byte threshold).

App:

```
C:\Users\corelan\Desktop\vc++\win10\Large_Alloc1\Release>Large_Alloc1.exe

Default process heap found at 0x00E20000
Press a key to start...

[1] Allocated chunk of 0x7ffb0 bytes at 0x00C48020

[2] Allocated chunk of 0x7ffb0 bytes at 0x01110020
[3] Allocated chunk of 0x7ffb0 bytes at 0x011A1020
[4] Allocated chunk of 0x7ffb0 bytes at 0x0123F020
[5] Allocated chunk of 0x7ffb0 bytes at 0x01326020
[6] Allocated chunk of 0x7ffb0 bytes at 0x013BA020
[7] Allocated chunk of 0x7ffb0 bytes at 0x0144C020
[8] Allocated chunk of 0x7ffb0 bytes at 0x014D2020
[9] Allocated chunk of 0x7ffb0 bytes at 0x0156D020
[10] Allocated chunk of 0x7ffb0 bytes at 0x015F4020
[11] Allocated chunk of 0x7ffb0 bytes at 0x01680020
[12] Allocated chunk of 0x7ffb0 bytes at 0x01712020
[13] Allocated chunk of 0x7ffb0 bytes at 0x017AB020
[14] Allocated chunk of 0x7ffb0 bytes at 0x0183B020
[15] Allocated chunk of 0x7ffb0 bytes at 0x018C9020
[16] Allocated chunk of 0x7ffb0 bytes at 0x01957020
[17] Allocated chunk of 0x7ffb0 bytes at 0x019EA020
[18] Allocated chunk of 0x7ffb0 bytes at 0x01A75020
[19] Allocated chunk of 0x7ffb0 bytes at 0x01B09020
[20] Allocated chunk of 0x7ffb0 bytes at 0x01B9D020
Press return to continue
```

WinDBG: (output of !heap -p -h

, limited to information related with VirtualAllocdBlocks)

```
VirtualAllocdBlocks @ e2009c
```

```
00c48018 10000 0004 [00] 00c48020 7ffb0 - (busy VirtualAlloc)
01110018 10000 0000 [00] 01110020 7ffb0 - (busy VirtualAlloc)
011a1018 10000 0000 [00] 011a1020 7ffb0 - (busy VirtualAlloc)
0123f018 10000 0000 [00] 0123f020 7ffb0 - (busy VirtualAlloc)
01326018 10000 0000 [00] 01326020 7ffb0 - (busy VirtualAlloc)
013ba018 10000 0000 [00] 013ba020 7ffb0 - (busy VirtualAlloc)
0144c018 10000 0000 [00] 0144c020 7ffb0 - (busy VirtualAlloc)
014d2018 10000 0000 [00] 014d2020 7ffb0 - (busy VirtualAlloc)
0156d018 10000 0000 [00] 0156d020 7ffb0 - (busy VirtualAlloc)
015f4018 10000 0000 [00] 015f4020 7ffb0 - (busy VirtualAlloc)
01680018 10000 0000 [00] 01680020 7ffb0 - (busy VirtualAlloc)
01712018 10000 0000 [00] 01712020 7ffb0 - (busy VirtualAlloc)
017ab018 10000 0000 [00] 017ab020 7ffb0 - (busy VirtualAlloc)
0183b018 10000 0000 [00] 0183b020 7ffb0 - (busy VirtualAlloc)
018c9018 10000 0000 [00] 018c9020 7ffb0 - (busy VirtualAlloc)
01957018 10000 0000 [00] 01957020 7ffb0 - (busy VirtualAlloc)
019ea018 10000 0000 [00] 019ea020 7ffb0 - (busy VirtualAlloc)
01a75018 10000 0000 [00] 01a75020 7ffb0 - (busy VirtualAlloc)
01b09018 10000 0000 [00] 01b09020 7ffb0 - (busy VirtualAlloc)
01b9d018 10000 0000 [00] 01b9d020 7ffb0 - (busy VirtualAlloc)
```

Yes, we can still trigger this kind of allocations.� Due to the nature of this type of allocations they are still placed at the start of a fresh new page (which is why we're seeing the start address alignments).� On the other hand, the gaps between 2 allocations seems to be larger than under Windows 7.�

This means that it will become harder to use this type of allocations to fill up a larger memory region as part of a heap spray.� There will be much bigger holes in between allocations, and the locations of the holes are also non-predictable, which means we may not be able to rely on absolute heap spray addresses as much as we could in Windows 7.

Perhaps a larger series of allocations is needed, and a larger number of runs, to find "sweet spots", addresses that are allocated more often than others.� Not sure what kind of percentage of predictability we may be able to obtain, but it might be worth the try.

Large_Alloc2

So… can we do a precise heap spray under Windows 10?

Well…. yes.� The key is to avoid LFH, and to avoid virtualallocdblocks as well.

Use a "sweet" size and a "sweet" number of allocs to get aligned consecutive allocations (starting at ????0048) as a normal chunk, inside a normal segment.� Perhaps the first few allocations won't start at that aligned address (because there are already some smaller allocations in the segment), but as soon as the allocations trigger the creation of another segment, and you manage to take the first spot, your allocations should be aligned.

App:

```
C:\Users\corelan\Desktop\vc++\win10\Large_Alloc2\Release>Large_Alloc2.exe

Default process heap found at 0x012B0000
Press a key to start...

[1] Allocated chunk at 0x012C0FF8
```

```
[2] Allocated chunk at 0x01300FF8
[3] Allocated chunk at 0x01340FF8
[4] Allocated chunk at 0x015B0048
[5] Allocated chunk at 0x015F0048
[6] Allocated chunk at 0x01630048
[7] Allocated chunk at 0x016B0048
[8] Allocated chunk at 0x016F0048
[9] Allocated chunk at 0x01730048
[10] Allocated chunk at 0x01770048
[11] Allocated chunk at 0x017B0048
[12] Allocated chunk at 0x017F0048
[13] Allocated chunk at 0x01830048
[14] Allocated chunk at 0x018B0048
[15] Allocated chunk at 0x018F0048
[16] Allocated chunk at 0x01930048
[17] Allocated chunk at 0x01970048
[18] Allocated chunk at 0x019B0048
[19] Allocated chunk at 0x019F0048
[20] Allocated chunk at 0x01A30048
[21] Allocated chunk at 0x01A70048
[22] Allocated chunk at 0x01AB0048

...
```

WinDBG:

```
0:003> d 0c0c0c0c

0c0c0c0c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c1c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c2c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c3c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c4c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c5c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c6c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0c0c0c7c 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

To turn this into a precise heap spray, we need to overcome the fact that we don't know the exact start address of the first one.� As we know that the start addresses will be aligned eventually to the start of a page, and we can control the size of the allocations, we simply have to repeat the same structure� (junk + ROP + shellcode + junk) every 0x1000 bytes inside each allocation (as explained in the heap spray tutorials on this site). This should allow you to put/find your content at a predictable address.� The same logic applies if you need to put specific values at specific places… simply repeat the layout every 0x1000 bytes and you should be fine.

In "Precise_Spray", the goal is to put marker "$$$$" (\x24\x24\x24\x24) at� 0x0c0c0c0c:

```
C:\Users\corelan\Desktop\vc++\win10_heap\Precise_Spray\Release>Precise_Spray.exe
Default process heap found at 0x00550000
Press a key to start...

Spray done, check 0x0c0c0c0c
>> Contents at 0x0c0c0c0c: 24242424
```

```
0:003> db 0c0c0c0c
```

```
0c0c0c0c� 24 24 24 24 20 20 20 20-20 20 20 20 20 20 20 20� $$$$������������
0c0c0c1c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
0c0c0c2c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
0c0c0c3c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
0c0c0c4c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
0c0c0c5c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
0c0c0c6c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
0c0c0c7c� 20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20�����������������
```

Of course, in a complex/multithreaded application, there will be 'noise' (other allocations and frees) at the same time your heapspray is running, and which could affect the placement of your allocations within the segment.��� A possible approach could be to cause some large allocations first (50 allocations of 0x1ff00 bytes or so… any big size, smaller than the chunk size you're using for the actual spray), each time followed by a small allocation (which we will keep allocated, to avoid that the big ones get coalesced),� and then free the large ones.� That way, the application can use those freed chunks, split them, consume them, without bothering your aligned spray at all.�

I've had good results with spraying using chunk sizes of 0x20000-8 bytes, and 0x40000-8 bytes, but I guess any similar aligned size that is a multiple of a page size will work.

Good luck y'all.� <3

Peter

Oh yeah, before I forget, please check out:

https://facebook.com/demandglobalchange // https://bit.ly/demandglobalchange_full // https://bit.ly/demandglobalchange

Read & share. Give people reasons to live, not to die for.� thank you.

Twitter | Reddit | Facebook 121 | Google | LinkedIn | Email | Pinterest

Related Posts:

- *Analyzing heap objects with mona.py*
- *Heap Layout Visualization with mona.py and WinDBG*
- *HITB2012AMS Day 2 – Ghost in the Allocator*
- *Exploit writing tutorial part 11 : Heap Spraying Demystified*
- *BlackHatEU2013 – Day2 – Advanced Heap Manipulation in Windows 8*
- *DEPS – Precise Heap Spray on Firefox and IE10*
- *Metasploit Bounty – the Good, the Bad and the Ugly*
- *Root Cause Analysis – Integer Overflows*
- *Root Cause Analysis – Memory Corruption Vulnerabilities*
- *Blackhat Europe 2010 Barcelona – Day 10*

Posted in Exploit Writing Tutorials, Windows Internals | Tagged back-end allocator, bea, block, breakpoint, C#, chunk, fea, front-end allocator, heap, heap management, heap spray, lfh, low fragmentation heap,