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
Stratsec senior consultant

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)
Heaps 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
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)
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)

<table>
<thead>
<tr>
<th>ListHint Size: 0x7f</th>
<th>FreeList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flink: 1 Blink: 0x00</td>
<td>Chunk: 1 Size: 0x120 Flink: 2</td>
</tr>
<tr>
<td></td>
<td>Chunk: 2 Size: 0x120 Flink: 1 Blink: 3</td>
</tr>
<tr>
<td></td>
<td>Chunk: 3 Size: 0x131 Flink: 2 Blink: 4</td>
</tr>
<tr>
<td></td>
<td>Chunk: 4 Size: 0x140 Flink: 3 Blink: 5</td>
</tr>
<tr>
<td></td>
<td>Chunk: 5 Size: 0x140 Flink: 4 Blink: 6</td>
</tr>
<tr>
<td></td>
<td>Chunk: 6 Size: 0x140 Flink: 5 Blink: 7</td>
</tr>
<tr>
<td></td>
<td>Chunk: 7 Size: 0x144 Flink: 6 Blink: 8</td>
</tr>
<tr>
<td></td>
<td>Chunk: 8 Size: 0x145 Flink: 7 Blink: 9</td>
</tr>
<tr>
<td></td>
<td>Chunk: 9 Size: 0x146 Flink: 9 Blink: 10</td>
</tr>
<tr>
<td></td>
<td>Chunk: 10 Size: 0x152 Flink: -</td>
</tr>
</tbody>
</table>
Windows 7 LFH heap data structures

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

<table>
<thead>
<tr>
<th>_HEAP_ENTRY</th>
<th>encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x00 FunctionIndex/Size</td>
<td>Yes</td>
</tr>
<tr>
<td>+0x02 Flags/ContextValue</td>
<td>Yes</td>
</tr>
<tr>
<td>+0x03 SmallTagIndex</td>
<td>Yes</td>
</tr>
<tr>
<td>+0x04 PreviousSize/UnusedBytesLength</td>
<td>No</td>
</tr>
<tr>
<td>+0x06 LFHFlags/SegmentOffset</td>
<td>No</td>
</tr>
<tr>
<td>+0x07 UnusedBytes/ExtendedBlockSignature</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header</th>
<th>Size</th>
<th>Flags</th>
<th>SmallTagIndex</th>
<th>PrevSize</th>
<th>SegOffset</th>
<th>UnusedBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextOffset</td>
<td>Userdata</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Users can overwrite the NextOffset of an allocated chunk
Example:

- Which Bin should you use for an allocation size of 232?
  \[ \frac{232}{8} = 0x1d \]

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

18 consecutive allocations 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:

```c
for (i=0; i<0x13; i++){
    chunks[i] = (char*)HeapAlloc(myheap,0,0x20);
}
```
## Windows 7 LFH allocation/free

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x02 NextOffset: 0x08</th>
<th>+0x08 NextOffset: 0x0e</th>
<th>+0x0e NextOffset: 0x14</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+0x1a NextOffset: 0x20</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

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.
### UserBlocks

<table>
<thead>
<tr>
<th>Offset</th>
<th>NextOffset: 0x0e</th>
<th>NextOffset: 0x14</th>
<th>NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x1a</td>
<td>+0x20</td>
<td>+0x26</td>
<td>+0x2c</td>
</tr>
<tr>
<td>+0x10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**aggrExchg.Depth = 0x29**

**aggrExchg.FreeEntryOffset = 0x40 (0x08 * 8)**

Next chunk: UserBlocks + FreeEntryOffset

Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.
### Windows 7 LFH allocation/free

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NextOffset: 0x14</td>
<td>NextOffset: 0x1a</td>
</tr>
<tr>
<td>+0x1a</td>
<td>NextOffset: 0x20</td>
<td>NextOffset: 0x26</td>
</tr>
<tr>
<td></td>
<td>NextOffset: 0x2c</td>
<td>NextOffset: 0x32</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x02</th>
<th>+0x0e</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>NextOffset:</strong> 0x0e</td>
<td><strong>NextOffset:</strong> 0x14</td>
<td><strong>NextOffset:</strong> 0x1a</td>
</tr>
<tr>
<td>+0x1a</td>
<td><strong>NextOffset:</strong> 0x20</td>
<td><strong>NextOffset:</strong> 0x26</td>
<td><strong>NextOffset:</strong> 0x2c</td>
</tr>
<tr>
<td></td>
<td><strong>NextOffset:</strong> 0x20</td>
<td><strong>NextOffset:</strong> 0x26</td>
<td><strong>NextOffset:</strong> 0x32</td>
</tr>
</tbody>
</table>

**aggrExchg.Depth** = **0x29**

**aggrExchg.FreeEntryOffset** = **0x10 (0x02 * 8)**

Next chunk: UserBlocks + FreeEntryOffset

Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.
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).

Set the contextValue to 0x000000002
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`).

<table>
<thead>
<tr>
<th>Header</th>
<th>0x0000</th>
<th>0x00</th>
<th>0x02</th>
<th>0x4141</th>
<th>0x0a</th>
<th>0x05</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NextOffset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Userdata</td>
</tr>
</tbody>
</table>

Let's assume we are using UserBlock bin 0x5. We set the segment offset for 2 chunks behind...
**Ben Hawkes #1 technique**

<table>
<thead>
<tr>
<th>UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated Object</td>
</tr>
<tr>
<td>+0x1a NextOffset: 0x20</td>
</tr>
</tbody>
</table>

We are supposed 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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>Allocated Object</th>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a</td>
<td>+0x20</td>
<td>0x26</td>
<td>0x2c</td>
<td>0x32</td>
</tr>
<tr>
<td>NextOffset:0x20</td>
<td>NextOffset:0x26</td>
<td>NextOffset:0x2c</td>
<td>NextOffset:0x32</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \text{aggrExchg.Depth} = 0x27 \]
\[ \text{aggrExchg.FreeEntryOffset} = 0xd0 (0x1a * 8) \]
Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.
Ben Hawkes #1 technique

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x02 NextOffset: 0x14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
<th>Allocated chunk</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

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, “AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x02\x00\x00\x00\x42\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,"AAAAAAAAAAAAAAAAAAAAAMMMMMMMMMMMM",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/struct
### FreeEntryOffset Overwrite

<table>
<thead>
<tr>
<th>Header</th>
<th>Size</th>
<th>Flags</th>
<th>SmallTagIndex</th>
<th>PrevSize</th>
<th>SegOffset</th>
<th>UnusedBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x000e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FreeEntryOffset Overwrite

<table>
<thead>
<tr>
<th>Header</th>
<th>Size</th>
<th>Flags</th>
<th>SmallTagIndex</th>
<th>PrevSize</th>
<th>SegOffset</th>
<th>UnusedBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x4242</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x08 NextOffset: 0x0e</th>
<th>+0x0e NextOffset: 0x14</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

\[
\text{aggrExchg.Depth} = 0x29 \\
\text{aggrExchg.FreeEntryOffset} = 0x40 (0x08 \times 8) \\
\text{Next chunk: UserBlocks + FreeEntryOffset Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.}
\]
### Windows 7 LFH allocation/free

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x08</th>
<th>+0x0e</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextOffset:</td>
<td>0xffff</td>
<td>0x14</td>
<td>0x1a</td>
</tr>
<tr>
<td>+0x1a</td>
<td>+0x20</td>
<td>0x26</td>
<td>0x2c</td>
</tr>
<tr>
<td>NextOffset:</td>
<td>0x20</td>
<td>0x26</td>
<td>0x2c</td>
</tr>
<tr>
<td>0x20</td>
<td>0x26</td>
<td>0x2c</td>
<td>0x32</td>
</tr>
</tbody>
</table>

| **aggrExchg.Depth** = 0x29  |

| **aggrExchg.FreeEntryOffset** = 0x40 (0x08 * 8) |

Next chunk: UserBlocks + FreeEntryOffset

Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.
Windows 7 LFH allocation/free

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e NextOffset: 0x14</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
</tr>
<tr>
<td>+0x2c NextOffset: 0x32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

aggrExchg.Depth = 0x28
aggrExchg.FreeEntryOffset = 0x7fff8 (0xffffff * 8)

Next chunk: Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.
Windows 7 LFH allocation/free

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e NextOffset: 0x14</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
</tr>
<tr>
<td>NextOffset: 0x20</td>
<td></td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

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,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\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,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",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

<table>
<thead>
<tr>
<th>Header</th>
<th>Size</th>
<th>Flags</th>
<th>SmallTagIndex</th>
<th>PrevSize</th>
<th>SegOffset</th>
<th>UnusedBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000c</td>
<td>Userdata</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e NextOffset: 0x14</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a</td>
<td>+0x20 NextOffset: 0x20</td>
<td>0x26 NextOffset: 0x2c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

\[\text{aggrExchg.Depth} = \text{0x28}\]

\[\text{aggrExchg.FreeEntryOffset} = \text{0x70 (0x0e * 8)}\]

**Next chunk:** Next chunk: UserBlocks + FreeEntryOffset

Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.
Offset match attack

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e NextOffset: 0x0e</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>+0x20 NextOffset: 0x26</td>
<td>0x26 NextOffset: 0x2c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>NextOffset: 0x0e</td>
<td>NextOffset: 0x1a</td>
</tr>
<tr>
<td>+0x20 NextOffset: 0x26</td>
<td>NextOffset: 0x2c</td>
<td>NextOffset: 0x32</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e NextOffset: 0x0e</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0x20 NextOffset: 0x26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26 NextOffset: 0x2c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2c NextOffset: 0x32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NextOffset: 0x0e</td>
<td>NextOffset: 0x1a</td>
</tr>
<tr>
<td>+0x1a</td>
<td>NextOffset: 0x20</td>
<td>NextOffset: 0x26</td>
</tr>
</tbody>
</table>

Next chunk: Next chunk: UserBlocks + FreeEntryOffset
Start at 0x02 so the manager accommodates the UserBlocks header of 16 bytes.

aggrExchg.Depth = 0x26
aggrExchg.FreeEntryOffset = 0x70 (0x0e * 8)

Points to & returns 0x0151ab60
Offset match attack

<table>
<thead>
<tr>
<th>UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>+0x1a \nNextOffset: \n0x20</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>UserBlocks</th>
<th>+0x0e NextOffset: 0x0e</th>
<th>+0x14 NextOffset: 0x1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x1a NextOffset: 0x20</td>
<td>0x20 NextOffset: 0x26</td>
<td>0x2c NextOffset: 0x32</td>
</tr>
</tbody>
</table>

**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
```
**Offset match attack**

Sample code triggering state using a double free...

```c
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
```
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, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 32);

// Call the function pointer from the struc
control_flow->get_eip();
Offset match attack
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 : _HEAP_SUBSEGMENT
    +0x004 Reserved : _HEAP_VIRTUAL_ALLOC_ALIGNMENT
    +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
```

BusyBitmap.Buffer->UserBlocks.BitmapData
Windows & 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

```assembly
MOV EAX, DWORD PTR FS:[18] ; load the current teb
MOVZ X ECX, WORD PTR DS:[EAX+FAA] ; \texttt{arrayindex} = \_teb
MOVZX ESI, BYTE PTR DS:[ECX+773EFF40] ; \texttt{rand\_index} =
; \texttt{RtlpLowFragHeapRandomData}
; [\texttt{arrayindex}]

MOV EAX, DWORD PTR FS:[18] ; load the current teb
MOV EBX, DWORD PTR SS:[EBP-2C] ; load the \_heap\_subsegment
INC ECX ; \texttt{arrayindex}++
AND ECX, OFF ; \texttt{arrayindex} &\texttt{=} 0xff
MOV WORD PTR DS:[EAX+FAA], CX ; \_teb->\texttt{LowFragHeapDataSlot}
; = \texttt{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
```
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)
Reset the Offset and Depth

OR ECX,EAX

MOV EAX,DWORD PTR SS:[EBP-20]

MOV DWORD PTR DS:[EAX],ECX

; calculate a new
; AggregateExchg.OffsetAndDepth
; load the current
; AggregateExchg.OffsetAndDepth
; set the new
; AggregateExchg.OffsetAndDepth
**Windows 8 LFH allocation**

```assembly
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 *
ADD EAX,EBX ; UserBlocks::BitmapData
ADD ECX,EAX ; next_chunk += UserBlocks
TEST BYTE PTR DS:[ECX+7],3F ; next_chunk += UserBlocks
               ; ->FirstAllocationOffset
JNZ ntdll.77392598 ; is the chunk free?
               ; jump if it isn’t
TEST ECX,ECX ; test to see if it returns a
               ; value...
JE ntdll.7730EFEB ; jump if it doesn’t
```
That calculation again:

\[
\text{next\_chunk} = \text{UserBlocks}\cdot\text{FirstAllocationOffset} + ((\text{UserBlocks}\cdot\text{BlockStride} \times \text{UserBlocks}\cdot\text{BitmapData}) + \text{UserBlocks})
\]
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 & 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
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 vs Windows 8

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uninitialized UserBlocks</th>
<th>Uninitialized UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
## Determinism

### Windows 7

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

### Windows 8

<table>
<thead>
<tr>
<th>Uninitialized UserBlocks</th>
<th>Uninitialized UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
## Determinism

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uninitialized UserBlocks</th>
<th>Uninitialized UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
### Determinism

#### Windows 7

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Windows 8

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

#### Uninitialized UserBlocks

<table>
<thead>
<tr>
<th>Uninitialized UserBlocks</th>
<th>Uninitialized UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
## Determinism

<table>
<thead>
<tr>
<th>Windows 7</th>
<th>Windows 8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UserBlocks 0x40</strong></td>
<td><strong>UserBlocks 0x40</strong></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Uninitialized UserBlocks</strong></th>
<th><strong>Uninitialized UserBlocks</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
## Determinism

### Windows 7

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

### Windows 8

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

### Uninitialized UserBlocks

<table>
<thead>
<tr>
<th>Uninitialized UserBlocks</th>
<th>Uninitialized UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
## Determinism

**Windows 7**

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

**Windows 8**

<table>
<thead>
<tr>
<th>Uninitialized UserBlocks</th>
<th>Uninitialized UserBlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
## Determinism

### Windows 7

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

### Windows 8

<table>
<thead>
<tr>
<th>UserBlocks 0x40</th>
<th>UserBlocks 0x40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
UserBlocks overwrite

<table>
<thead>
<tr>
<th>Windows 7</th>
<th>Windows 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserBlocks 0x40</td>
<td>UserBlocks 0x40</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Unknown distance between last free chunk and next UserBlocks
UserBlocks overwrite

Sized bin: 0x40

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00E72700</td>
<td>UserBlocks0 -&gt; 08 chunks – No Guard</td>
</tr>
<tr>
<td>0x00e73710</td>
<td>SubSegment0 -&gt; UserBlocks0</td>
</tr>
<tr>
<td>0x00e73738</td>
<td>SubSegment1 -&gt; UserBlocks1</td>
</tr>
<tr>
<td>0x00e73760</td>
<td>SubSegment2 -&gt; UserBlocks2</td>
</tr>
<tr>
<td>0x00e73788</td>
<td>SubSegment3 -&gt; UserBlocks3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00E73b00</td>
<td>UserBlocks1 -&gt; 19 chunks – No Guard</td>
</tr>
<tr>
<td>0x00e75b00</td>
<td>UserBlocks2 -&gt; 19 chunks – No Guard</td>
</tr>
<tr>
<td>0x00e77b00</td>
<td>UserBlocks3 -&gt; 19 chunks – No Guard</td>
</tr>
</tbody>
</table>

We must not damage the _lfh_block_zone
**UserBlocks overwrite**

**Sized bin: 0x40**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00E72700</td>
<td>UserBlocks0 -&gt; 08 chunks – No Guard</td>
</tr>
<tr>
<td>0x00e73710</td>
<td>SubSegment0 -&gt; UserBlocks0</td>
</tr>
<tr>
<td>0x00e73738</td>
<td>SubSegment1 -&gt; UserBlocks1</td>
</tr>
<tr>
<td>0x00e73760</td>
<td>SubSegment2 -&gt; UserBlocks2</td>
</tr>
<tr>
<td>0x00e73788</td>
<td>SubSegment3 -&gt; UserBlocks3</td>
</tr>
<tr>
<td>0x00E73b00</td>
<td>UserBlocks1 -&gt; 19 chunks – No Guard</td>
</tr>
<tr>
<td>0x00e75b00</td>
<td>UserBlocks2 -&gt; 19 chunks – No Guard</td>
</tr>
<tr>
<td>0x00e77b00</td>
<td>UserBlocks3 -&gt; 19 chunks – No Guard</td>
</tr>
</tbody>
</table>

*Overwrite starting From UserBlocks1*
UserBlocks overwrite

Sized bin: 0x40

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
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

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\cdot FirstAllocationOffset +
((UserBlocks\cdot BlockStride \ast UserBlocks\cdot BitmapData) +
UserBlocks)

Sequential overflow against the UserBlocks will work...
FirstAllocationOffset = 0xXXYY (any value)
BlockStride = 0xXXYY (any value)
UserBlocks\cdot BitmapData = 0x00000000 (any value)
Setting BitmapData pointer to NULL seems best...
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:
  - \((0x40 \times 0x8) / 0x10\) requires an even number of chunks
  - \((0x41 \times 0x8) / 0x10\) 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/