

# An Introduction to Win32 Heap Overflows

Lin0xx  
Noxusfiles.com  
Offensive Vulnerability Research

# What is a Heap?

- Dynamically allocated memory for arbitrary variables
- Can be a lot larger than the stack
- Used for when the stack isn't large enough to house data or just inefficient
- Process is given 1mb to start with – this grows as needed
- Some people think an overflow here is 'non-exploitable'

# Heap Management

- Each OS has its own heap implementation
  - Linux – dlmalloc
  - Windows - RtlHeap
- Most of us know the CRT functions: malloc(), realloc(), and free()
- These wrap RtlAllocateHeap(), RtlReAllocateHeap(), and RtlFreeHeap()
  - Inside of ntdll
- Side note: only SP1 is covered here

# Where is the Heap?

```
0:001> dt _PEB 7ffd4000
```

```
+0x000 InheritedAddressSpace : 0 "
```

```
+0x001 ReadImageFileExecOptions : 0 "
```

```
+0x002 BeingDebugged : 0x1 "
```

```
+0x003 SpareBool : 0 "
```

```
+0x004 Mutant : 0xffffffff
```

```
+0x008 ImageBaseAddress : 0x01000000
```

```
...
```

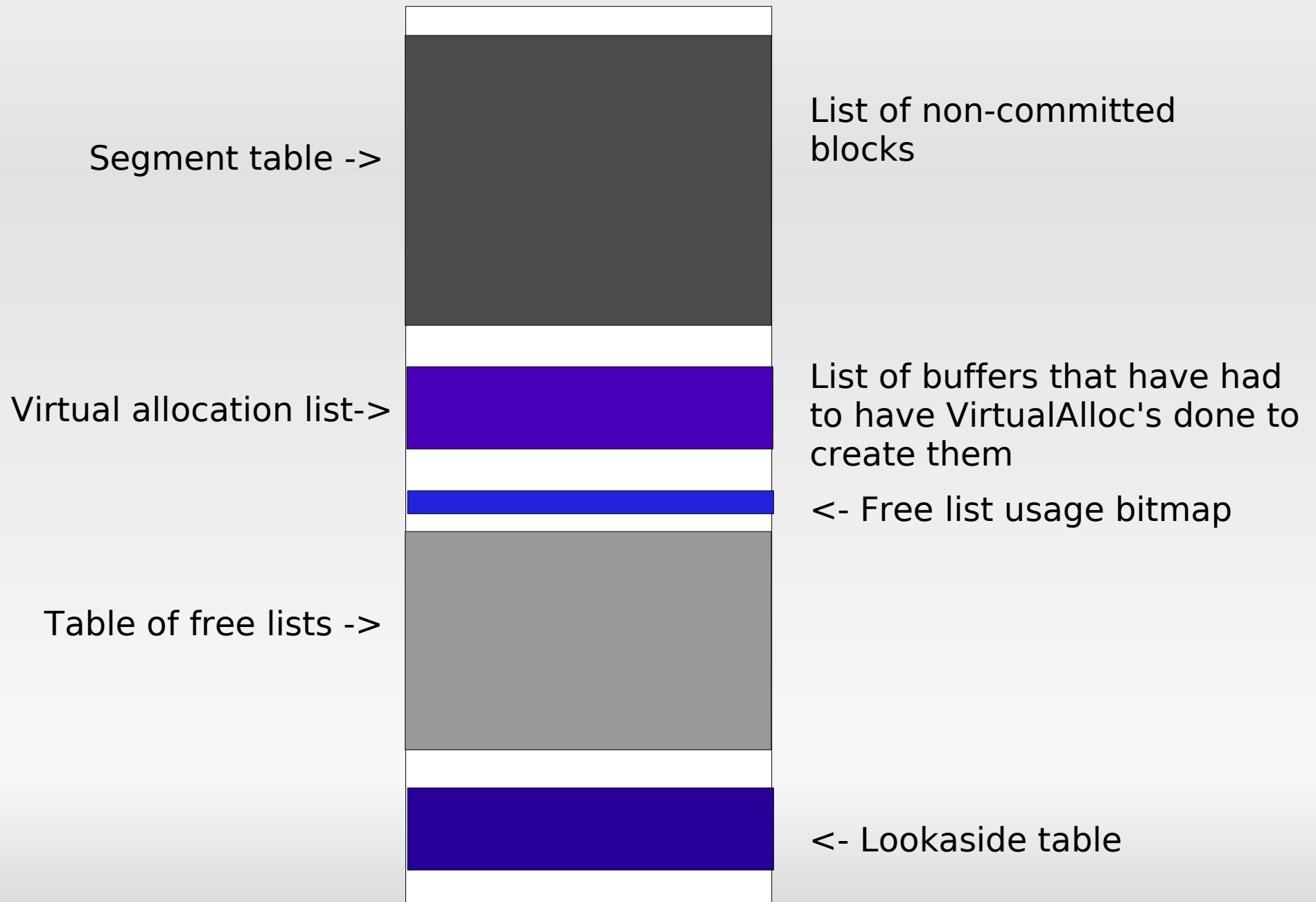
```
+0x018 ProcessHeap : 0x000a0000 <- default heap
```

```
+0x088 NumberOfHeaps : 0xa <- how many other heaps
```

```
+0x090 ProcessHeaps : 0x7c97de80 -> 0x000a0000
```

```
^Heap list
```

# Heap Diagram

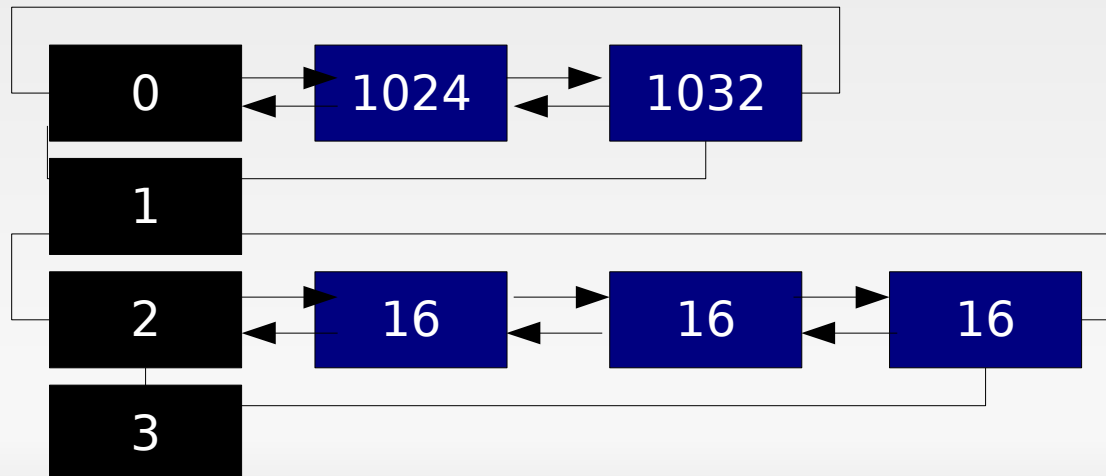


# Free List

- 128 entries – all tell the amount of 'allocation units' used
- Each entry starts a doubly linked list
  - This is really just the block being 8-byte aligned – all allocations are rounded up to this boundary.
- FreeList[1] does not exist due to size of heap management headers and FreeList[0] is used for free blocks with size  $> 1024$ 
  - These are sorted from smallest to greatest

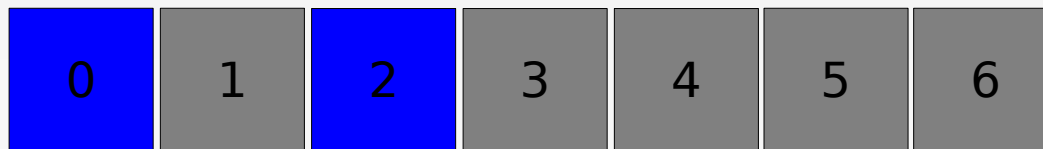
# Free List Diagram

- Each list entry consists of  
Flink – the forward link  
Blink – the backward link  
(this means it's a doubly linked list ;)



# Free List Bitmap

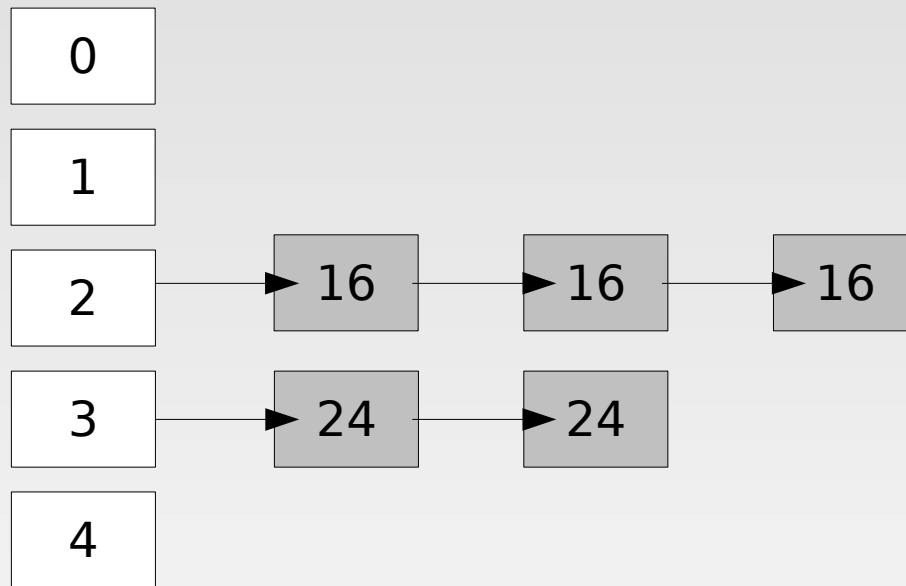
- 4 unsigned longs (128 bits in total)
  - 1 if the FreeList[x] contains data, 0 if it doesn't.
  - x is amount of allocation units
- Keeps track of whether a certain FreeList has blocks inside of it
- Used for quick searching



# The Lookaside List

- Size of 128 – starts empty
  - Singly linked list that is self balancing
- Speed increasing feature of RtlHeap
- When a certain size of block gets allocated / freed at a high frequency, the blocks are pointed to here
  - If “busy” chunks get less busy, they are removed.
- Is turned on if `HEAP_NO_SERIALIZE` and `HEAP_GROWABLE` flags are set (default)

# Lookaside List Diagram



# Heap Management Structure

Busy Block Header  
(HEAP\_ENTRY)

|               |       |               |              |
|---------------|-------|---------------|--------------|
| Size          |       | Previous Size |              |
| Seg.<br>index | Flags | Unused        | Tag<br>Index |

size -> (real\_block\_size + header) / 8  
previous size -> previous block size  
seg. index -> segment index where the  
block resides  
Flink -> link to next block  
Blink -> link to previous block

Free Block Header  
(LIST\_ENTRY)

|                       |       |               |              |
|-----------------------|-------|---------------|--------------|
| Size                  |       | Previous Size |              |
| Seg.<br>index         | Flags | Unused        | Tag<br>Index |
| Forward Link (Flink)  |       |               |              |
| Backward Link (Blink) |       |               |              |

# Pwning a Live Target

- Okay, so the heap basics are covered, now let's focus on offense.
  - We're using the `unlink()` method.
- These overflows occur like any other.
  - `strcpy()`, `memcpy()`, or other inlined / called memory copying function.
- Now that we know the layout of the heap, we can abuse it to our advantage.

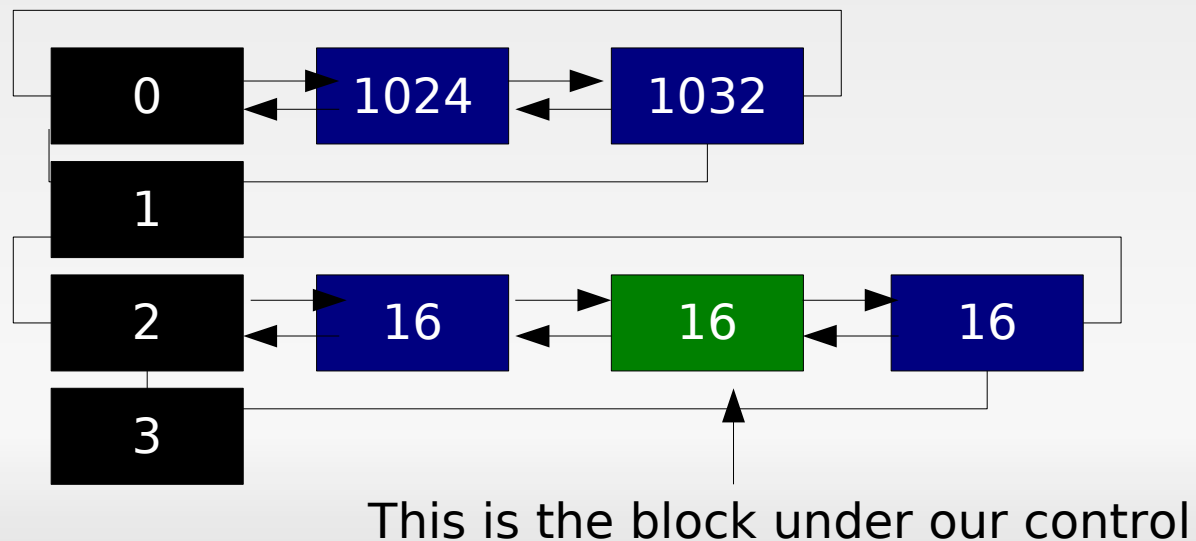
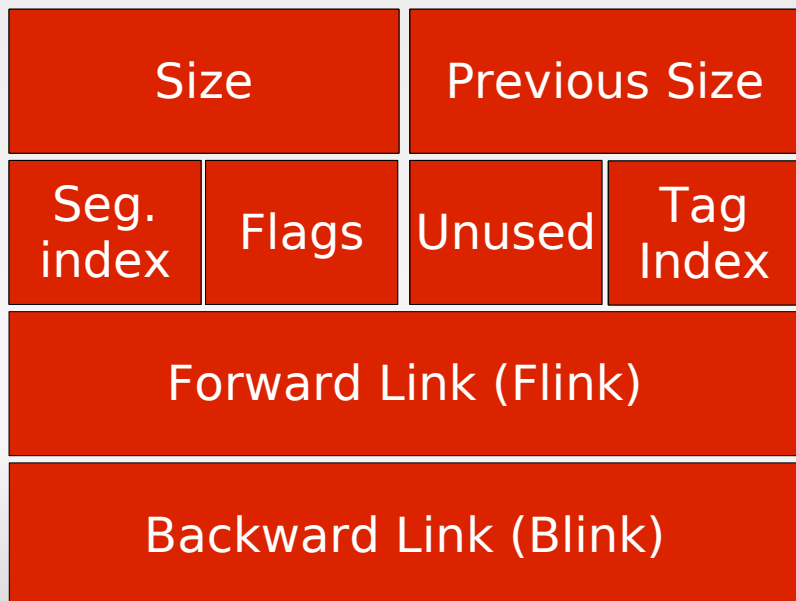
# Anatomy of an RtlAllocateHeap()

- `char * buf = ( * char) malloc(323);`
  - `total = align_to_8_bytes(323 + 8);`
- If `total < VirtualAlloc` value:
  - Check lists to see if there are free blocks:
    - Lookaside list
    - `FreeList[1-127]`
    - `FreeList[0]`
- If `total >= VirtualAlloc()` threshold:
  - Call `VirtualAlloc()` to allocate memory from the OS.

# Smashing the Heap

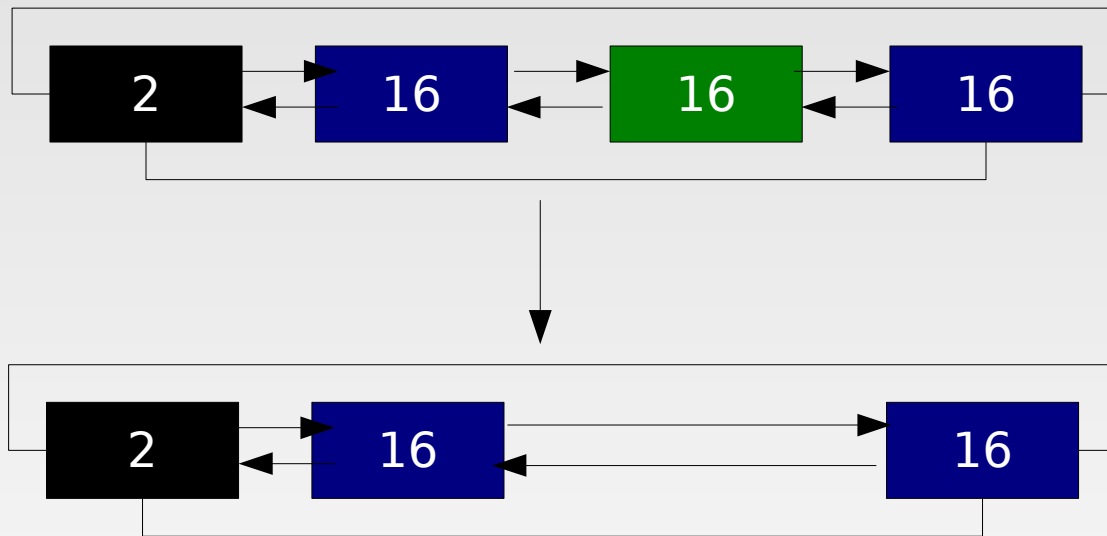
- If the heap block after your data is a free block...
  - And you have an overflow..
  - You can contaminate the heap block.

Free Block Header



# free() to Own

- When an `RtlAllocateHeap()` happens, the block must be removed from the FreeList



# The Unlink

- The process of removing the block from this list is called unlinking.

- Code:

```
MOV DWORD PTR DS:[EAX],ECX
```

```
MOV DWORD PTR DS:[ECX+4],EAX
```

- $EAX = \text{blink} \mid ECX = \text{flink}$
- We control both of these, thus we can now overwrite a dword of our choosing.
- The first overwrite will work, but the second one will crash the program.

# First Run

- Run `dc404-heap-test.rb` to find the offsets into the buffer that `eax` and `ecx` are located
- Uses the Rex pattern creation libraries
- Find the values of `ecx` and `eax`, then use `pattern_offset.rb` to tell their offsets
- `EAX offset = 212`, `ECX offset = 208`

# Now What?

- What do we overwrite?
- Assume XPSP1, so we can overwrite the Unhandled Exception Filter (UEF).
  - The UEF is the exception handler that gets called after all of the SEH chains are walked and none of them can handle the exception.
- We then point the UEF to our code.
- Our code will run because of the exception thrown by the 2<sup>nd</sup> write.

# Where is the UEF?

- Disassemble  
SetUnhandledExceptionFilter()

```
MOV ECX,DWORD PTR SS:[ESP+4]
```

```
MOV EAX,DWORD PTR DS:[77ED73B4]
```

```
MOV DWORD PTR DS:[77ED73B4],ECX
```

```
RETN 4
```

- The code simply moves the first argument passed to it into 0x77ED73B4, which one can assume is the location of the UEF.

# Stealing EIP

- After the “call eax,” we must get back to our shellcode.
- We must examine the stack and registers to see where a pointer to data we control lies.
- Windbg to the rescue!!! (Thanks Richard!)

```
0:000> ddp edi
```

```
0012f554 0012f640 c0000005...
```

```
0012f5c4 00360750 cccccccc <-our data!
```

- $0x12f5c4 - 0x12f554 = 70...$

# The Right Return

- We need a “call [edi+0x70]”
  - “\xff\x57\x70”
- Don't bother trying to use ollydbg to search for it.
  - The binary search function was written by someone smoking crack.
- Use memdump.exe and msfpescan
- msfpescan -x “\xff\x57\x70” -d <dir>
- Gives us... 0x7804bf8d in RPCRT4.dll

# Digging in

- We must jump to the top of the user data to have enough space to execute shellcode.
- Assemble a jump 200 bytes backwards
  - `\xE9\x33\xFF\xFF\xFF`
- Payload can be generated by metasploit
  - `./msfpayload windows/exec CMD=calc.exe EXITFUNC=process R | ./msfencode -b '\x00\x0a' -t ruby`
  - `EXITFUNC=process` so we don't crash the app

# Pwnage in a Can

- Run the ruby file, we win.
- Thanks to
  - Rolf Rolles and Richard Johnson for explaining things
  - Skape for helping fix dll problems during compilation
- Books/papers used
  - *Exploiting Software* by Hoglund
  - *The Shellcoder's Handbook*
  - Matt Connover's 2004 Blackhat Speech

# Oh Yeah...

- SP2 / Vista will be covered later.