A short story by Nicolas Waisman

nicolas@immunityinc.com

http://twitter.com/nicowaisman

Who am i?

- Regional Manager at Immunity, Inc.
- Research and Development of reliable Heap Overflow exploitation for CANVAS attack framework
- Big fan of Guarana.

Once upon a time

- There was a bug on MS **NTTP server ...**
- MS04-036 was the starting point for finding new ways to exploit the heap.
- This is how this the bitmask heap technique born.

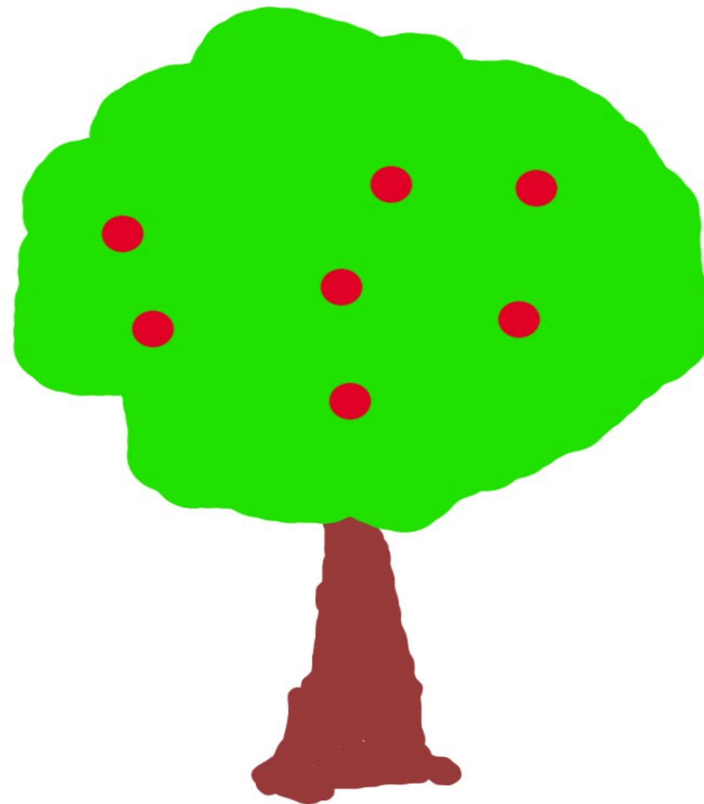
Public Exploits

VS

Commercial Exploits

Windows HEAP

The heap provides a mechanism for Allocating, Reallocation and freeing dynamic memory.



Windows HEAP

ntdll.dll

Why is *that* important?

Google - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back · Search · Favorites

Address http://www.google.com.ar/ · Go · Links

La Web Imágenes Noticias Grupos Libros Blogs Gmail Más ▾ · iGoogle | Acceder

Una forma más rápida de navegar la web.

Buscar con Google

Buscar en: la Web · página

Programas de publicidad - Soluciones Empre

Hacer de Google

©2009

http://www.google.com.ar/intl/es/options/

Log data

```
Address   Message
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0040E01C  RtlAllocateHeap(0x00030000, 0x00000000, 0x00000020) <- 0x0040e01c
0040262A  RtlAllocateHeap(0x00030000, 0x00000000, 0x0000002c) <- 0x0040262a
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02070440)
0BADF00D  RtlFreeHeap(0x00030000, 0x00000000, 0x02013590)
0BADF00D  ==============================================
          [09:59:28] Thread 000000CC terminated, exit code 0
```

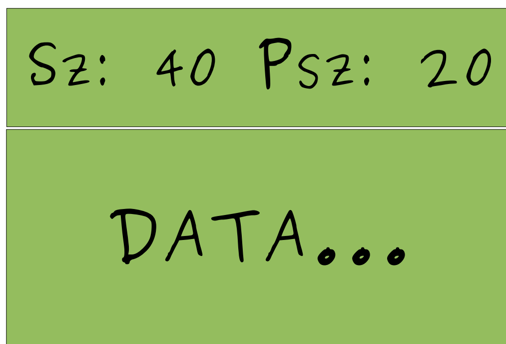
23587 heap functions executed

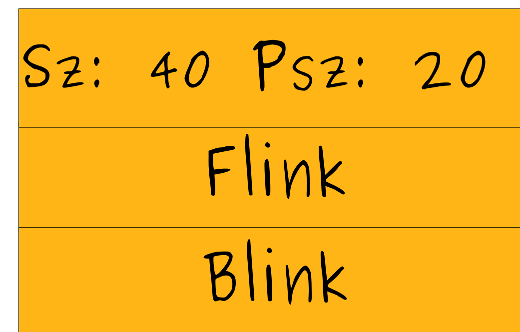
How was the heap exploited?

- In the past, research had taken advantage of the (in)famous “unlink” technique.
- This technique could allow an attacker to write 4 bytes, whenever they want.
- 4 bytes + some effort = Shellcode Execution

Learn heap exploitation in 3 minutes

- Every piece of heap dynamic memory, are represented as chunks.
- Chunks has headers





Learn heap exploitation in 3 minutes

- **Free chunks are connected forwardly and backwardly.**

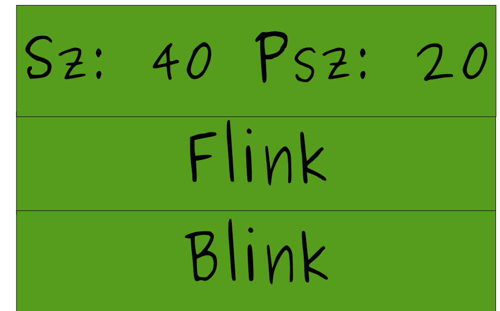
Sz: 40 Psz: 20
Flink
Blink

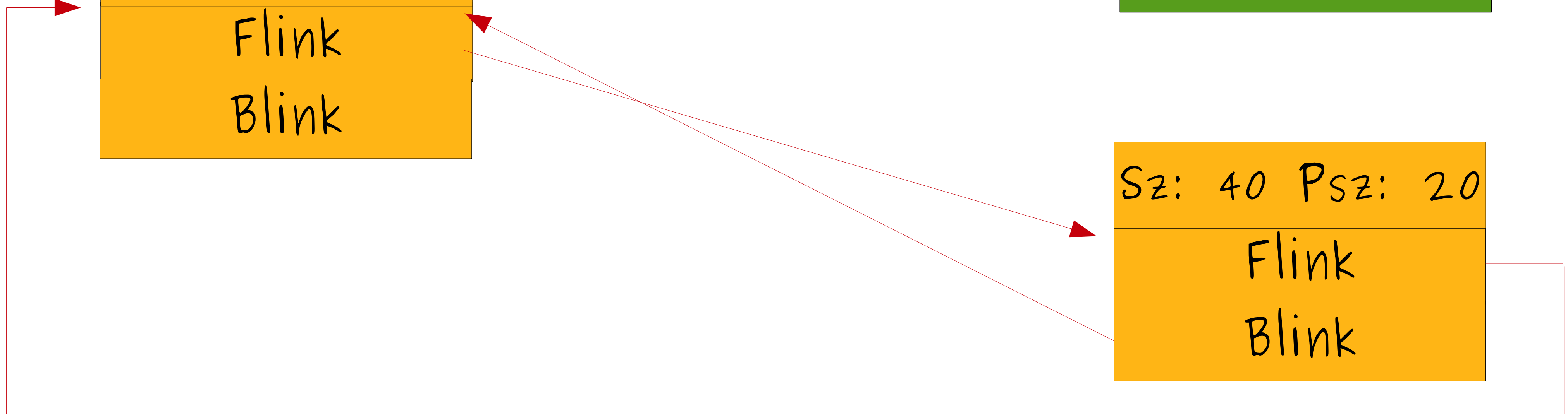
Sz: 40 Psz: 20
Flink
Blink

Sz: 40 Psz: 20
Flink
Blink

Learn heap exploitation in 3 minutes

- When memory is required, it will take one of the free chunk and unlink it.





What if we overflow the chunk before being freed?

Demo

Need three volunteer from the audience

FreeList

- For each “List” of connected chunks, there is a slot on the FreeList structure.
- This speed up the search process.

FreeList

0	BL	FL
1	BL	FL
2	BL	FL
3	BL	FL
4	BL	FL
5	BL	FL
n	BL	FL

1600 bytes
BL FL

2000 bytes
BL FL

24 bytes
BL FL

24 bytes
BL FL

n*8 bytes
BL FL

where $n < 128$

FreeListInUse

0 BL FL
1 BL FL
2 BL FL
3 BL FL
4 BL FL
5 BL FL
n BL FL

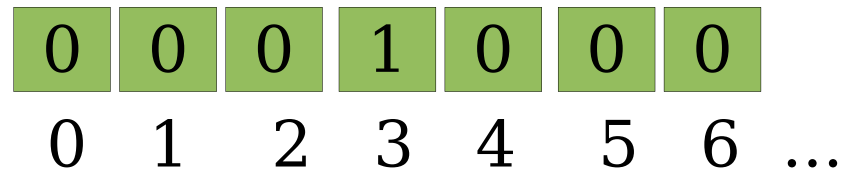
1600 bytes
BL FL

2000 bytes
BL FL

24 bytes
BL FL

24 bytes
BL FL

n*8 bytes
BL FL



Heap dump 0x00150000

Address	Chunks
0x00000000	### Immunity's Heapdump ###
0x00150000	Dumping heap: 0x00150000
0x00150000	Flags: 0x00000002 Forceflags: 0x00000000
0x00150000	Total Free Size: 0x00001d7b VirtualMemoryThreshold: 0x0000fe00
0x00000000	Segment[0]: 0x00150000
0x00000000	FreeListInUse 10011001011001110100110100000100 00000000000000000001000000000010
0x00000000	01000000000000000000000000000010 00000000000000000000000000000000
0x00150178	[000] 0x00150178 -> [0x001d07c0 | 0x00212c80]
0x00212c80	0x00212c80 -> [0x00150178 | 0x001f43c8] (00000471)
0x001f43c8	0x001f43c8 -> [0x00212c80 | 0x001bc008] (00000405)
0x001bc008	0x001bc008 -> [0x001f43c8 | 0x001d9dc8] (00000401)
0x001d9dc8	0x001d9dc8 -> [0x001bc008 | 0x001d5110] (000002a0)
0x001d5110	0x001d5110 -> [0x001d9dc8 | 0x001dd848] (00000202)
0x001dd848	0x001dd848 -> [0x001d5110 | 0x001e3008] (00000202)
0x001e3008	0x001e3008 -> [0x001dd848 | 0x001d0eb8] (00000150)
0x001d0eb8	0x001d0eb8 -> [0x001e3008 | 0x00204008] (00000141)
0x00204008	0x00204008 -> [0x001d0eb8 | 0x001a62a8] (0000012d)
0x001a62a8	0x001a62a8 -> [0x00204008 | 0x001c1878] (00000116)
0x001c1878	0x001c1878 -> [0x001a62a8 | 0x001d07c0] (0000009c)
0x001d07c0	0x001d07c0 -> [0x001c1878 | 0x00150178] (00000081)
0x00150180	[001] 0x00150180 -> [0x00150180 | 0x00150180]
0x00150188	[002] 0x00150188 -> [0x001becf8 | 0x001ab500]
0x001ab500	0x001ab500 -> [0x00150188 | 0x00188d40] (00000002)
0x00188d40	0x00188d40 -> [0x001ab500 | 0x001c4100] (00000002)
0x001c4100	0x001c4100 -> [0x00188d40 | 0x0018e868] (00000002)
0x0018e868	0x0018e868 -> [0x001c4100 | 0x001a5b48] (00000002)
0x001a5b48	0x001a5b48 -> [0x0018e868 | 0x0019b9a0] (00000002)
0x0019b9a0	0x0019b9a0 -> [0x001a5b48 | 0x00197da8] (00000002)
0x00197da8	0x00197da8 -> [0x0019b9a0 | 0x001d50b8] (00000002)
0x001d50b8	0x001d50b8 -> [0x00197da8 | 0x00201660] (00000002)
0x00201660	0x00201660 -> [0x001d50b8 | 0x001a9fd8] (00000002)
0x001a9fd8	0x001a9fd8 -> [0x00201660 | 0x001f70c8] (00000002)
0x001f70c8	0x001f70c8 -> [0x001a9fd8 | 0x001edde8] (00000002)
0x001edde8	0x001edde8 -> [0x001f70c8 | 0x001dcca0] (00000002)
0x001dcca0	0x001dcca0 -> [0x001edde8 | 0x001e0ff8] (00000002)
0x001e0ff8	0x001e0ff8 -> [0x001dcca0 | 0x001ffb58] (00000002)
0x001ffb58	0x001ffb58 -> [0x001e0ff8 | 0x001cb9b0] (00000002)
0x001cb9b0	0x001cb9b0 -> [0x001ffb58 | 0x001becf8] (00000002)
0x001becf8	0x001becf8 -> [0x001cb9b0 | 0x00150188] (00000002)
0x00150190	[003] 0x00150190 -> [0x00150190 | 0x00150190]
0x00150198	[004] 0x00150198 -> [0x00150198 | 0x00150198]
0x001501a0	[005] 0x001501a0 -> [0x001501a0 | 0x001501a0]
0x001501a8	[006] 0x001501a8 -> [0x001501a8 | 0x001501a8]

DISCLAIMER

The events and characters in this presentation are fictitious. Any similarity to actual persons, living or dead, is purely coincidental

FreeListInUse: Analogy

1

2

3

At least,
One chunk

Vulnerability in NNTP Could Allow Remote Code Execution

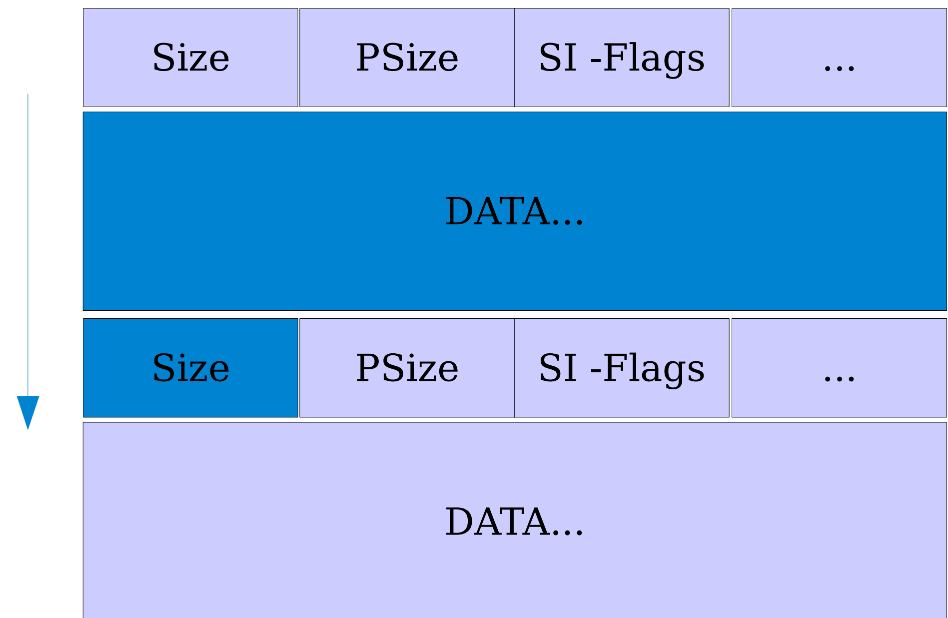
- Found in October 2004 by two Argentinian researchers: Lucas "Rompedor" Lavarello y Juliano Rizzo
- Various parsing errors of the XPAT command could lead into part of the heap being overwritten.
- The challenge was to exploit an **off-by-two**

Moral: "Everything good start with a bug"

Off By Two

Based on a 2000 bytes chunks, we were able to overwrite the next chunk with only two bytes.

We are only able to control one of the two characters:



\x41\x41\x41\x41\x41\x00

How would you Exploit it?

- Lookaside: Nothing interesting
- Busy Chunk: Forward coalescation
- Free Chunk: Make the size bigger.
- Last Chunk: Make the size bigger.
- New Technique. (The whole point of this talk)

Size	PSize	SI -Flags	...
DATA...			

“A technique should take less requirements than a strawberry pudding recipe”

Strawberry pudding law
– Sinan Eren

Interaction with the FreeListInUse

Set bit: When a FreeList slot of some size is **empty** and you want to free a chunk there.

Interaction with the FreeListInUse

0 BL FL
1 BL FL
2 BL FL
3 BL FL
4 BL FL

1600 bytes
BL FL

2000 bytes
BL FL

0 0 0 0 0 0 0
0 1 2 3 4 5 6 ...

Free

24 bytes

Interaction with the FreeListInUse

<u>Unset bit</u>: When you allocate a chunk which is the **only member** of a FreeList Slot.

Interaction with the FreeListInUse

	BL	FL
0	BL	FL
1	BL	FL
2	BL	FL
3	BL	FL
4	BL	FL

1600 bytes

BL FL

2000 bytes

BL FL

24 bytes

BL FL

Alloc(16)

0	0	0	1	0	0	0	
0	1	2	3	4	5	6	...

In Code...

CPU - thread 00000CE4, module ntdll

```
7C9111FE   0FB70E            MOVZX ECX,WORD PTR DS:[ESI]           ecx = Chunk->Size
7C911201   8BC1              MOV EAX,ECX
7C911203   C1E8 03           SHR EAX,3                             ByteInFreeList = Size / 8
7C911206   8985 28FFFFFF     MOV DWORD PTR SS:[EBP-D8],EAX
7C91120C   83E1 07           AND ECX,7                             EntryInByte = Size & 7
7C91120F   33D2              XOR EDX,EDX
7C911211   42                INC EDX
7C911212   D3E2              SHL EDX,CL                            ByteToSet = 1 << EntryInByte
7C911214   8995 04FFFFFF     MOV DWORD PTR SS:[EBP-FC],EDX
7C91121A   8D8418 58010000   LEA EAX,DWORD PTR DS:[EAX+EBX+158]
7C911221   33C9              XOR ECX,ECX
7C911223   8A08              MOV CL,BYTE PTR DS:[EAX]              tmpbyte = FreeListInUse[ ByteInFreeList ]
7C911225   33CA              XOR ECX,EDX                           tmpbyte = xor( tmpbyte, ByteToSet)
7C911227   8808              MOV BYTE PTR DS:[EAX],CL              FreeListInUse[ ByteInFreeList] = tmpbyte
7C911229   E9 18020000       JMP ntdll.7C911446
7C91122E   8D93 78010000     LEA EDX,DWORD PTR DS:[EBX+178]
7C911234  ^E9 5AFFFFFF       JMP ntdll.7C911193
7C911239   81E1 FF000000     AND ECX,0FF
```

Tabla XOR

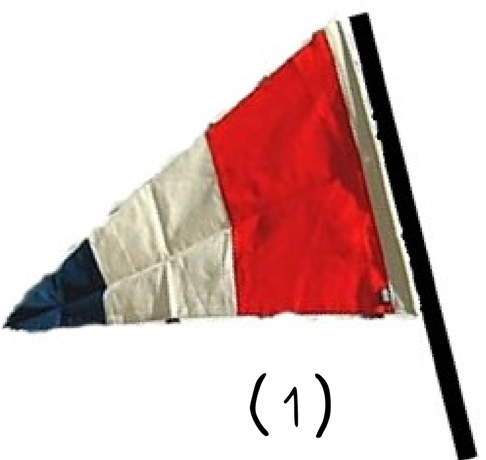
1 xor 1 = 0

1 xor 0 = 1

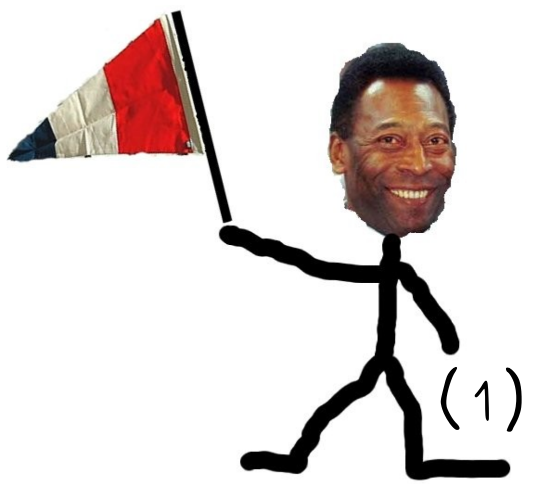
0 xor 1 = 1

0 xor 0 = 0

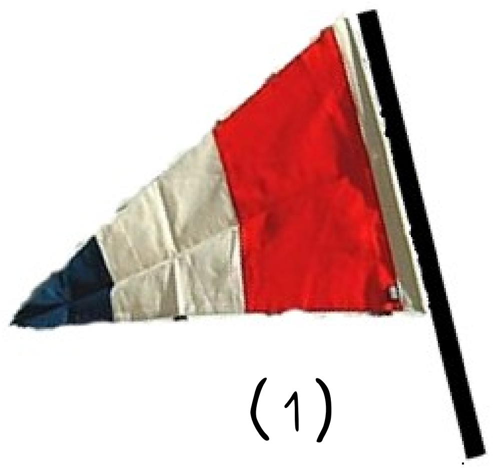
In Pseudo-Code...



XOR



=

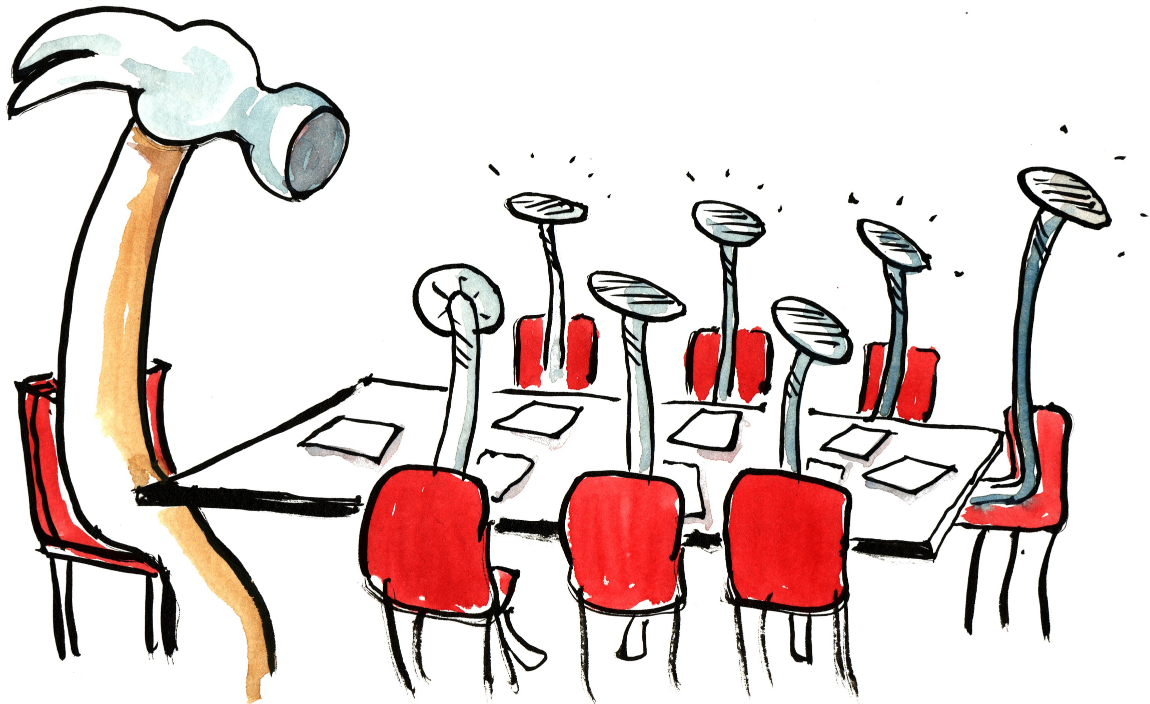


XOR

=

Did you get it?

Exploiting... the Bitmask



When the only tool you have is a hammer...

Exploiting the Bitmask

```
Heap dump 0x00150000
Address     Chunks
0x00150220  [015]  0x00150220 -> [ 0x00150220 | 0x00150220 ]
0x00150228  [016]  0x00150228 -> [ 0x0017dc60 | 0x001be040 ]
0x001be040         0x001be040 -> [ 0x00150228 | 0x001a9770 ] (00000016)
0x001a9770         0x001a9770 -> [ 0x001be040 | 0x0017dc60 ] (00000016)
0x0017dc60         0x0017dc60 -> [ 0x001a9770 | 0x00150228 ] (00000016)
0x00150230  [017]  0x00150230 -> [ 0x001caa28 | 0x001caa28 ]
0x001caa28         0x001caa28 -> [ 0x00150230 | 0x00150230 ] (00000017)
0x00150238  [018]  0x00150238 -> [ 0x00150238 | 0x00150238 ]
0x00150240
0x001c5fa0
0x001c0668
0x00150248
0x00150250
0x0015fd88
0x00150258
0x001bf960
0x001c2990
```

```
Heap dump 0x00150000
Address     Chunks
0x001ca9b8              heap:  *0x00150000*        flags:    0x00000001 (B)
0x001ca9c8  0x001ca9c8> size:   0x00000058  (000b) prevsize: 0x00000010 (0002)
0x001ca9c8              heap:  *0x00150000*        flags:    0x00000001 (B)
0x001caa20  0x001caa20> size:   0x000000b8  (0017) prevsize: 0x00000058 (000b)
0x001caa20              heap:  *0x00150000*        flags:    0x00000000 (F)
0x001caa20              next:   0x00150230         prev:     0x00150230
```

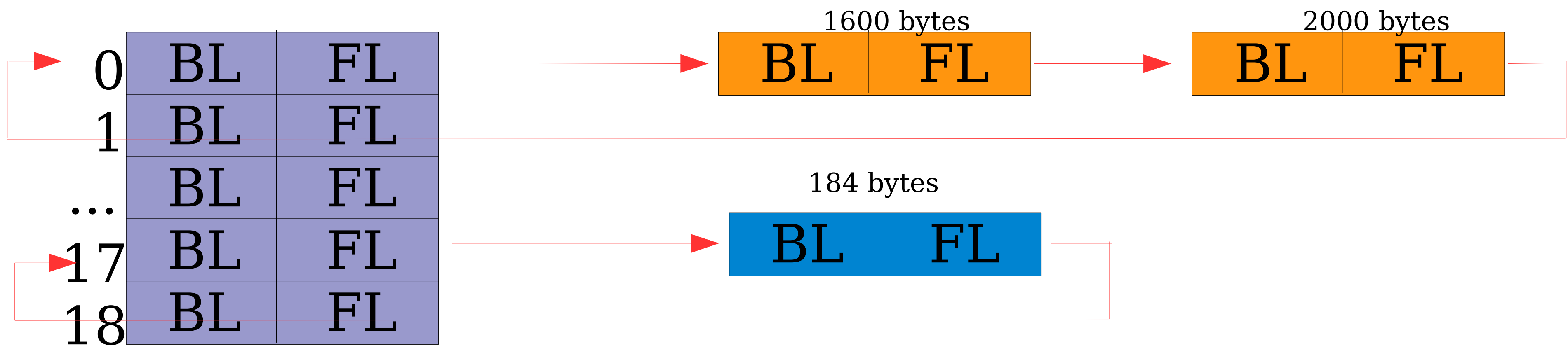
Forcing the overflowed chunk to be:

- Free
- Size < 1024
- Only chunk on a FreeList Slot

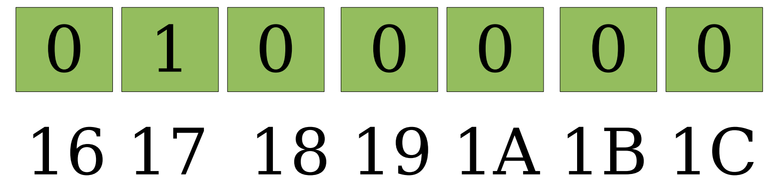
Exploiting the bitmask

- Overflow ONLY the chunk's size with a size that correspond to an empty FreeList slot.

Exploiting the bitmask



Overflow the chunk of size 184 for 992. Chunk continue being connected to slot 17.



Exploiting the bitmask

Exploiting the bitmask

Exploiting the bitmask

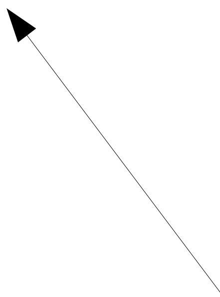
- Force an Allocation of the chunk of the original size.

Exploiting the bitmask

16

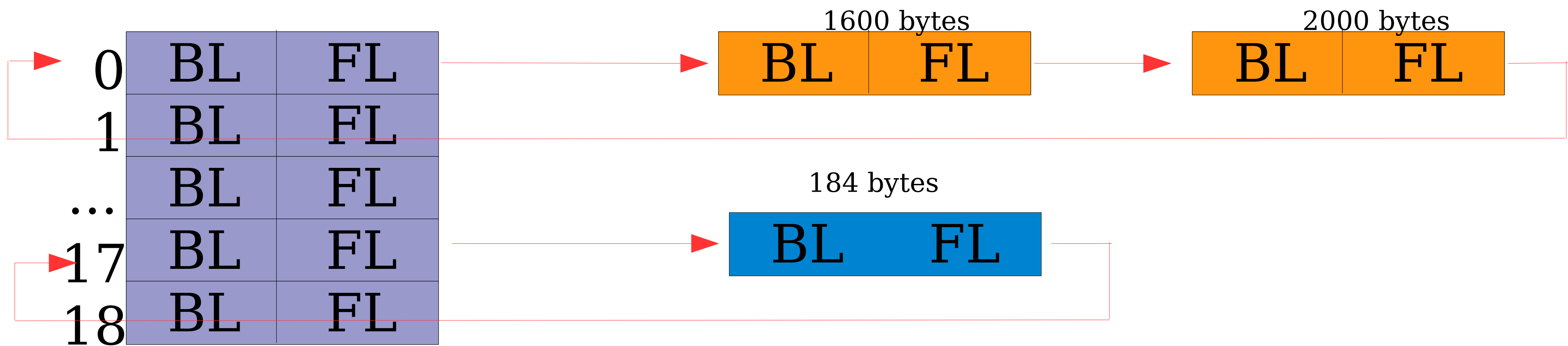
17

18



Allocate(184)

Exploiting the bitmask

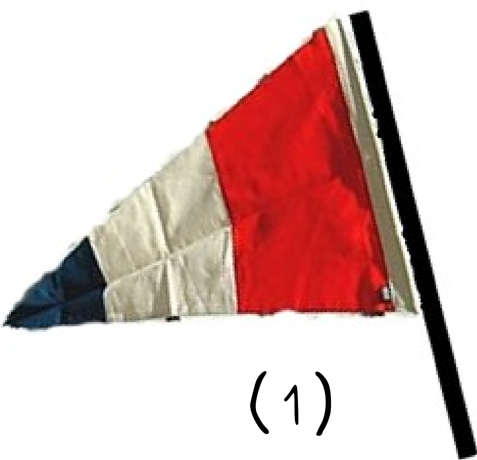


79	7A	7B	7C	7D	7E	7F
0	0	0	1	0	0	0

An allocation of 184 bytes will try to take that chunk out of the list. But since it's the only (and last) chunk it need to **Unset** the corresponding FreeListInUse, but the opposite happens...

Unset pseudo code: `XOR[7c]=1`

Exploiting the bitmask



XOR

=

Exploiting the bitmask

7b

7c

As a result, we had a FreeListInUse slot looking like it has chunks, but it doesn't

7d

Fallum ergo Sum

Exploiting the bitmask

- Next time **allocation** of fake size (992), the algorithm will find out that 0x7C seems available (but it's empty).
- An empty slot points to itself (This means, its point to the FreeList)

Exploiting the bitmask

FreeList[7c] points to itself.

An allocation will return a pointer to itself.

What to write?

HEAP + 0x57C = RtlCommitRutine

Address	Hex dump	ASCII
00150558	58 05 15 00 58 05 15 00	X■■.X■■.
00150560	60 05 15 00 60 05 15 00	`■■.`■■.
00150568	C8 47 18 00 C8 47 18 00	ÈG■.ÈG■.
00150570	E0 81 15 00 10 FA 17 00	à■■.■ú■.
00150578	08 06 15 00 00 00 00 00	■■■.....
00150580	F0 9F 1E 00 00 00 02 05	ð■■...■■
00150588	00 00 00 00 00 50 D9 02	.....PÙ■

0222E4EC	001
0222E4F0	000
0222E4F4	000
0222E4F8	000
0222E4FC	000
0222E500	000
0222E504	023
0222E508	024

Heap dump 0x00150000

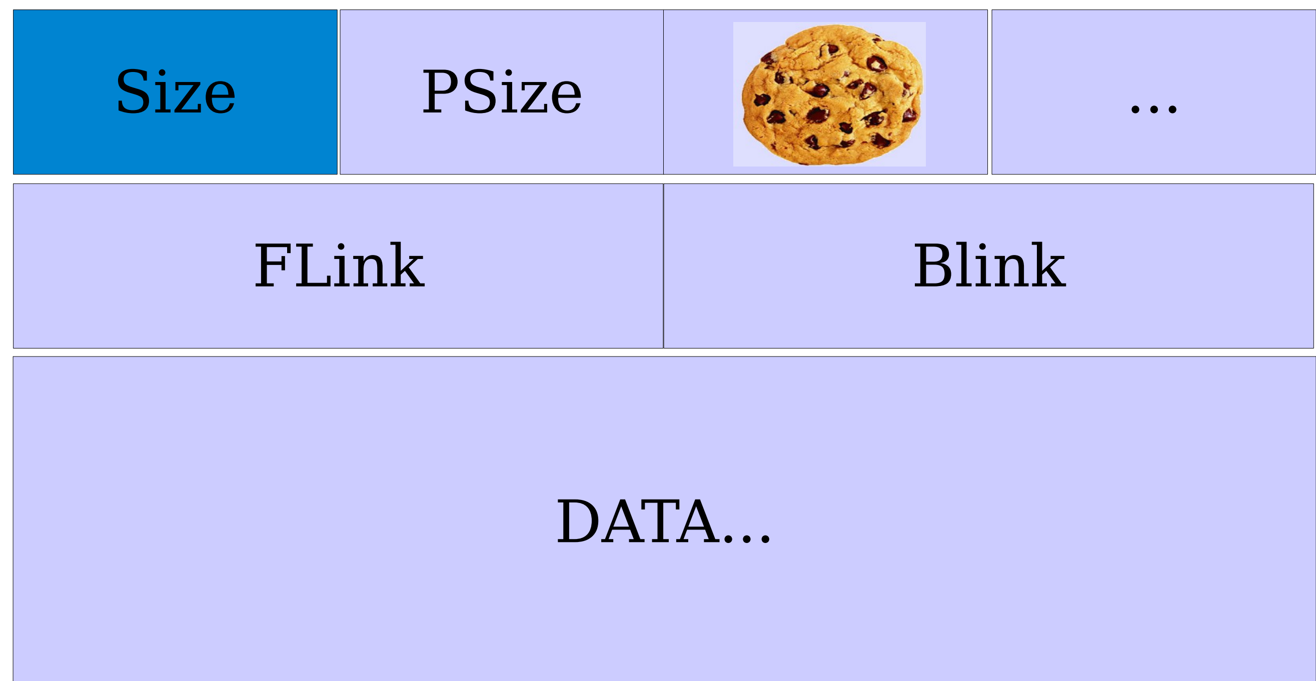
Address	Chunks
0x00150540	[079] 0x00150540 -> [0x001d6168 | 0x0015d660]
0x0015d660	0x0015d660 -> [0x00150540 | 0x001b40a0] (00000079)
0x001b40a0	0x001b40a0 -> [0x0015d660 | 0x001d6168] (00000079)
0x001d6168	0x001d6168 -> [0x001b40a0 | 0x00150540] (00000079)
0x00150548	[07a] 0x00150548 -> [0x00150548 | 0x00150548]
0x00150550	[07b] 0x00150550 -> [0x001bdad8 | 0x001b3638]
0x001b3638	0x001b3638 -> [0x00150550 | 0x001bdad8] (0000007b)
0x001bdad8	0x001bdad8 -> [0x001b3638 | 0x00150550] (0000007b)
0x00150558	[07c] 0x00150558 -> [0x00150558 | 0x00150558]
0x00150560	[07d] 0x00150560 -> [0x00150560 | 0x00150560]
0x00150568	[07e] 0x00150568 -> [0x001847c8 | 0x001847c8]

Neither the Heap Cookie or ASLR Affect this technique

Cookie Diet...

At the moment of Allocation, the algorithm doesn't check FreeList chunk's for cookies!

(FTW!)



Two bytes overwrite

+

Two controlled allocations

=

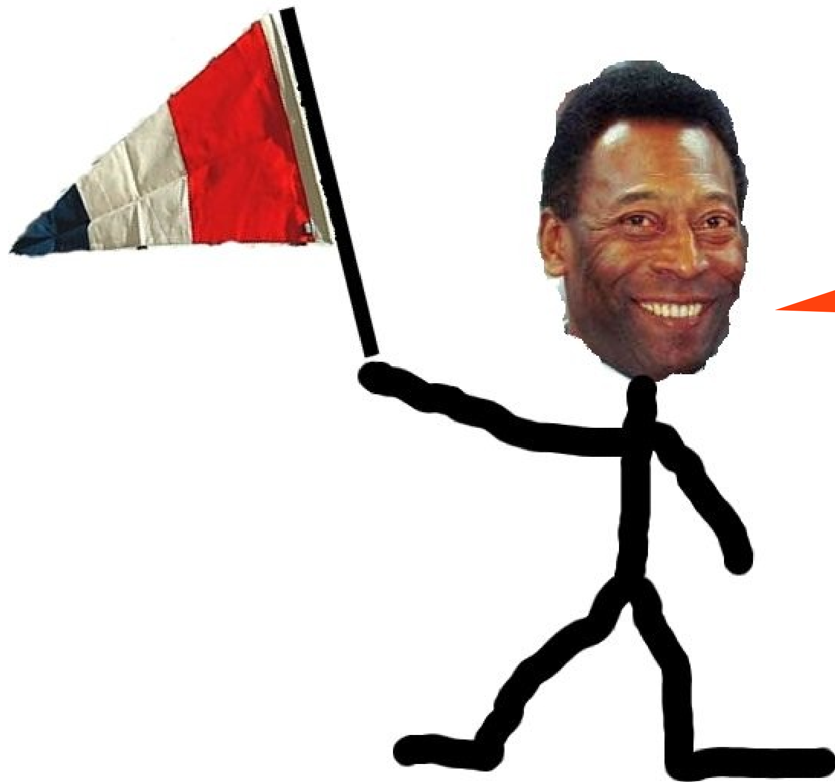
One reliable exploit

Bitmask Alternatives: XOR another bit

What if we make the fake size be bigger than 1018 (FreeList size)?

John McDonald y Chris Valasek technique

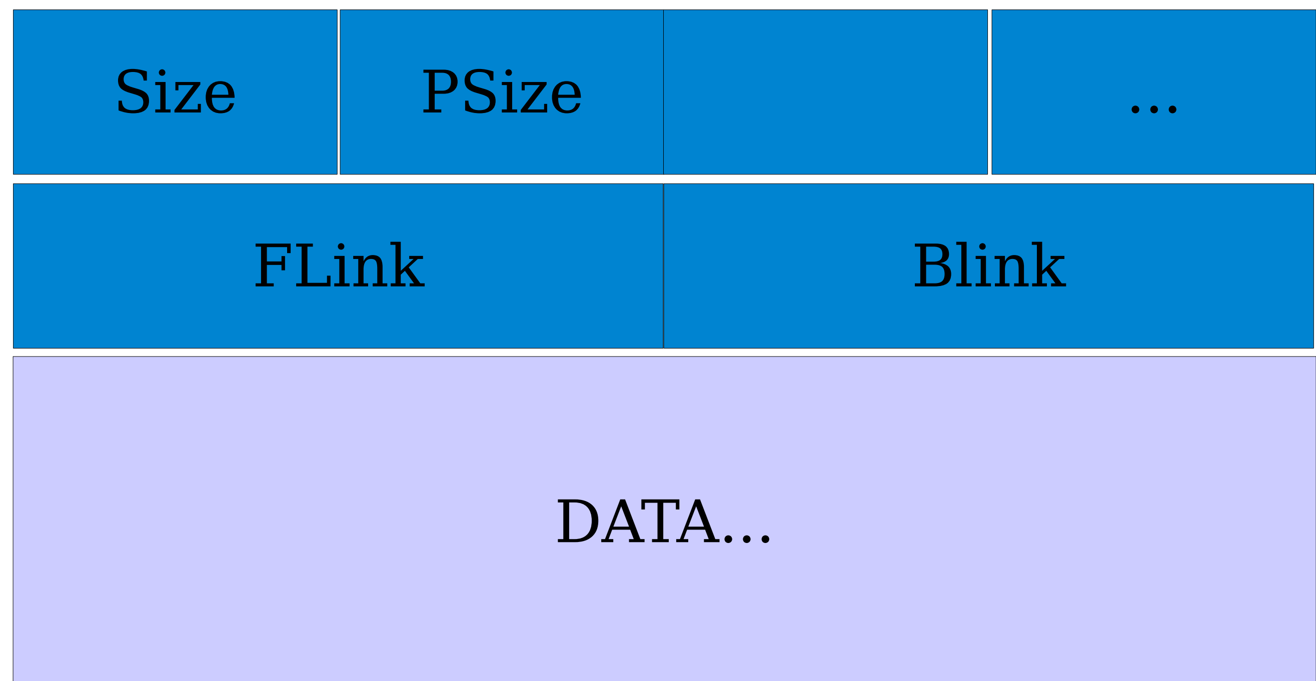
We will XOR 1 bit after the FreeList.



I recommend the RtlCommitRutine

Bitmask Alternatives: 16 bytes overwrite

What if instead of two, you can overwrite 16 bytes?



Make:

Forward link = Backward link

Trick:

Bitmask to think you are the only chunk.

Exploiting...

Demo

Considerations

- Vista and Windows7 protect the chunk's header (per-heap key encryption and checksum).
- As it was explained the technique won't work on Vista/Win7, but its highly possible that one the variations work.

Homework

- Read "*Practical Windows XP/2003 Heap Exploitation*" by John McDonald y Chris Valasek for more techniques and fun.

Questions?