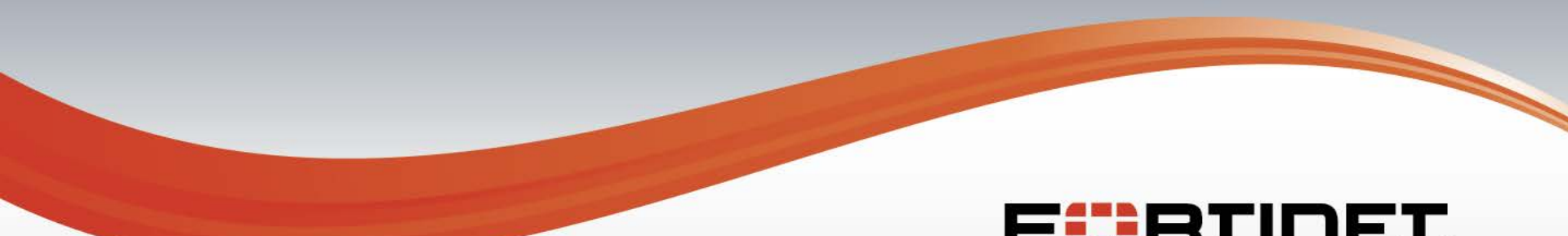




Advanced Heap Manipulation in Windows 8



FORTINET®

NSC #1

Who Am I

Zhenhua(Eric) Liu
Senior Security Researcher
Fortinet, Inc.

Previous:

Dissecting Adobe ReaderX's Sandbox:
Breeding Sandworms@BlackHat EU 2012

Agenda

0x01:

Why start this research

0x02:

Quick View of The Idea

0x03:

*Implementations
(Kernel Pool / User heap)*

Intro

Why start this research. (Motivation)

*Exploiting Memory corruption vulnerability
are more difficult today*

Windows 8: Exploit mitigation improvements.

Possible ways for Sandbox bypassing

Kernel Vulnerability

3rd-party plug-ins Vulnerability

Sandbox flaws

Windows 8 Kernel

-- The patched Win 7 Kernel

A: NULL Dereference protection

B: Kernel pool integrity checks

C: Non-paged pool NX

D: Enhanced ASLR

E: SMEP/PXN

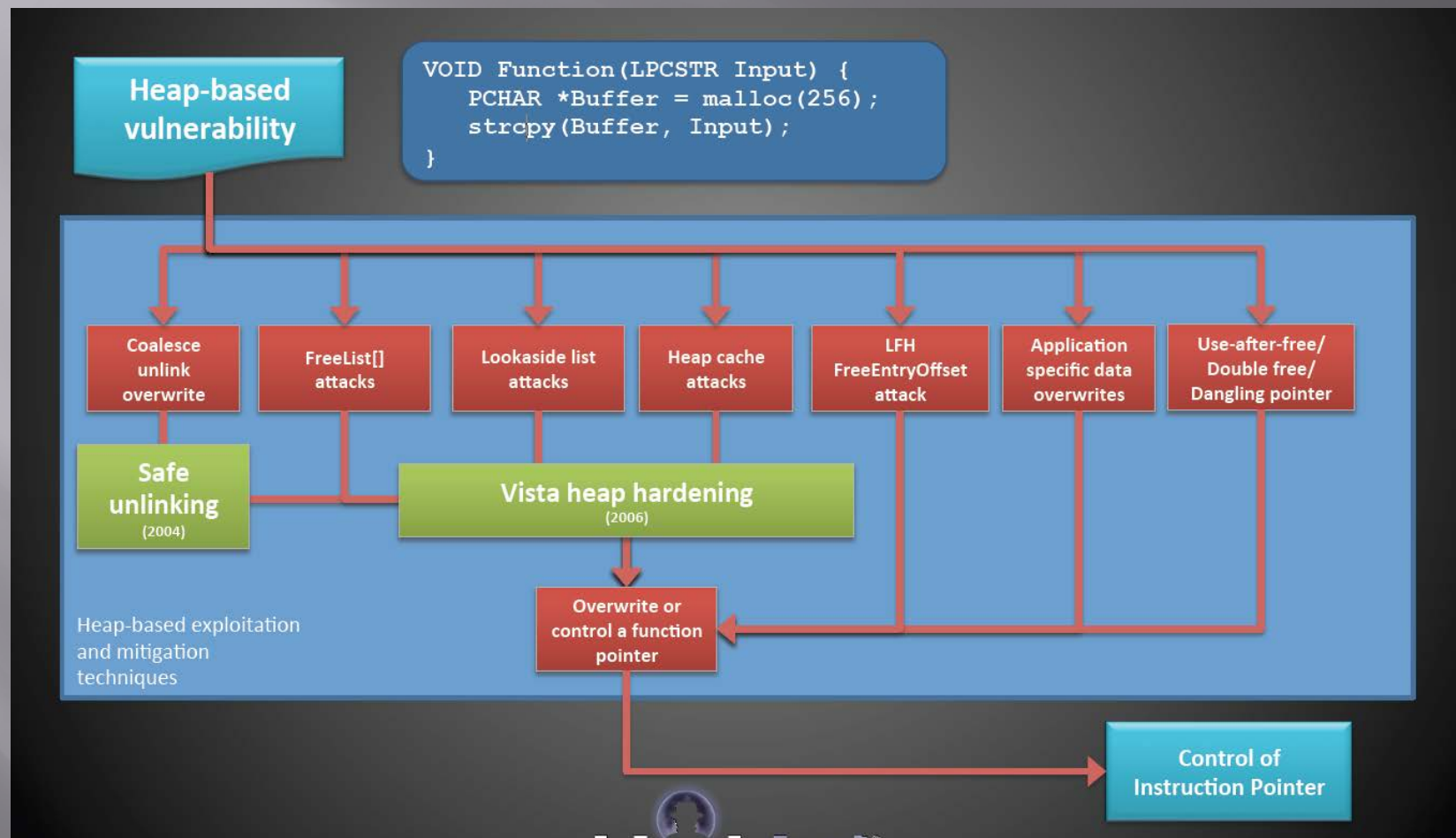
Windows 8 User Heap

-- determinism is at a all time low

A: High entropy Randomized LFH allocator

B: Guard pages

What's left



Matt Miller

http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

Why Application Specific Data attacking?

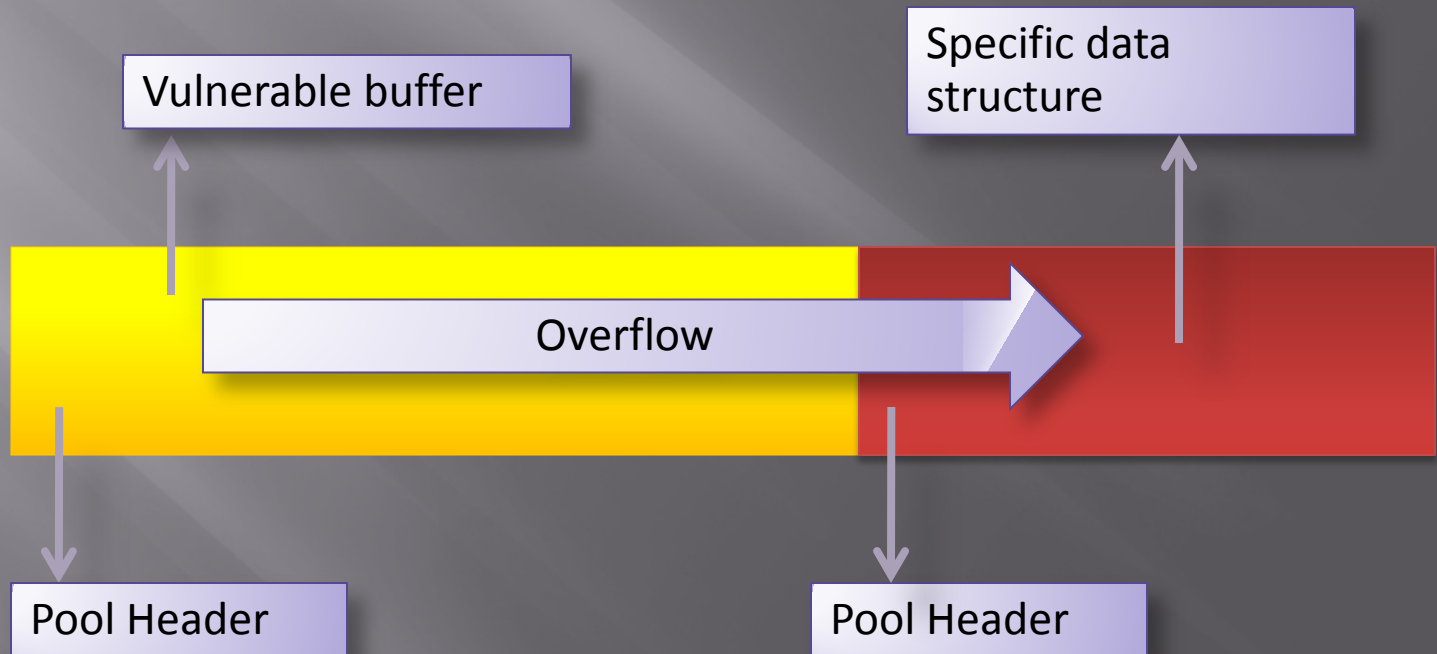
*Application Specific data attacking are the future.
- Ben Hawkes*

*Compromising Application Specific data are
facilitated by **heap manipulation***

What is ..

Overflow the target application's data stored on the heap.

Adjacent is the key!



风水 feng shui



吴成槐

Taken

Noise

Free

Vul buffer

Defragment



Taken

Noise

Free

Vul buffer

Make Holes



Taken

Noise

Free

Vul buffer

Allocate vulnerable buffer



vulnerable buffer will fall
into this place

The limitations 1:

Arbitrary size of vulnerable buffer?

We can not always find kernel object which size is the same as the vulnerable buffer, and it also contains the data structure for exploitation.

The limitations 2:

Randomized LFH makes it fail

*Defragment will trigger Randomized LFH,
vulnerable buffer will not fall into the hole we made.*

Target of This Research

let the arbitrary **vulnerable** buffer **adjacent** with **arbitrary** data structure.

without triggering the **LFH** in user heap.



0x01:
Quick View of The Idea

Windows Objects in Kernel Vulnerability Exploitation

How to place a desired object just behind the vulnerable buffer?

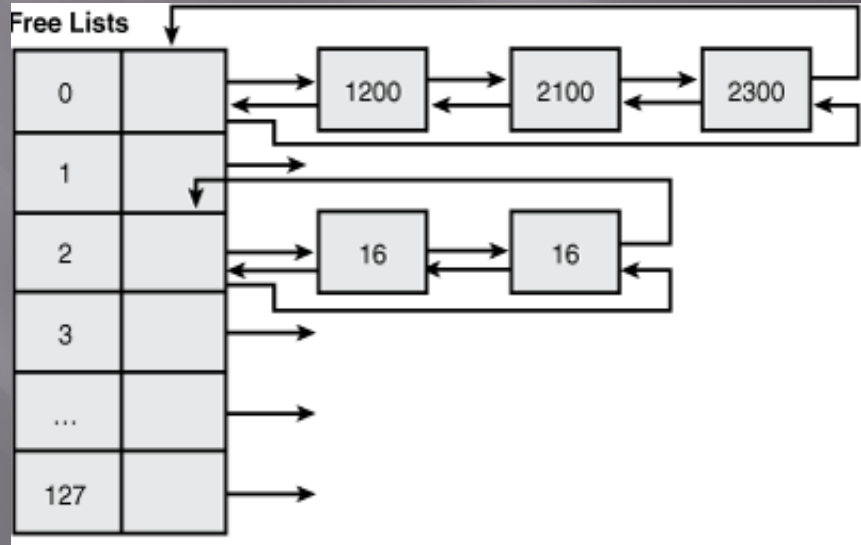
Can we place something else other than object?

FreeLists

A: Doubly linked lists

B: For fast allocation and free

C:LIFO manner



Drawbacks

A: Metadata attacking

*B: **0** allocation entropy*

FreeLists are still been used in both kernel pool and user heap as of Windows 8.

Control the FreeLists



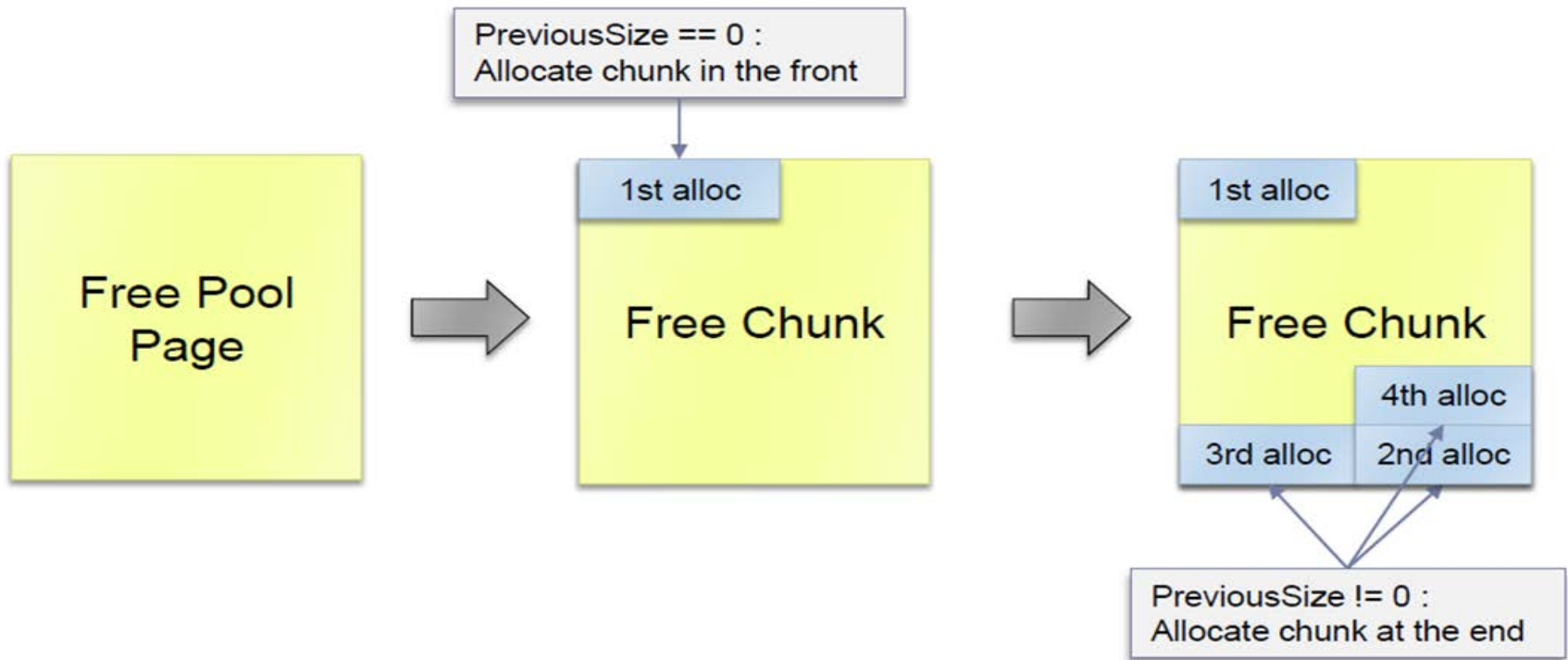
3 ways to write into the FreeLists

1: Direct free.
(Fixed FreeLists)

2: Split big chunk when allocating.
(Calculated FreeLists)

3: Coalescence when freeing.
(Calculated FreeLists)

Splitting Pool Chunks process

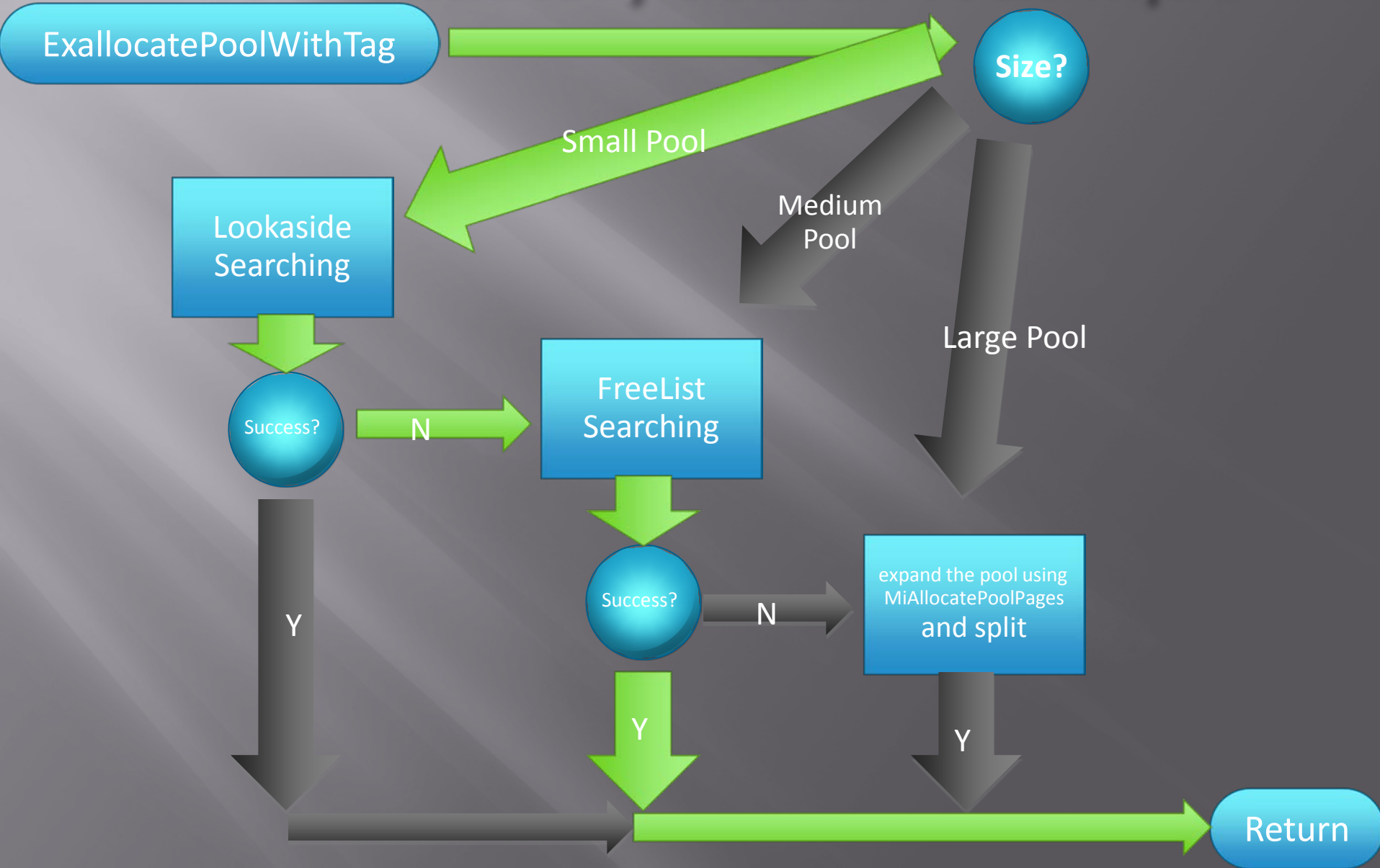


The Mandatory Search Technique

Force the FreeLists searching process to take place.
Force the searching result greater than requested.

--To control the Freelists dynamically when allocating

The Mandatory Search Technique



The Mandatory Search Technique

ExallocatePoolWithTag

Size?

Small Pool

Medium
P

Lookaside
Searching

Success?

N

FreeList
Searching

Success?

Y

Y

Return

```
RtlpFindEntry();  
RtlpHeapRemoveListEntry();  
// FreeListEntry is controlled  
  
if (CommitSize < FreeListEntry ->Size){  
// Force the CommitSize smaller than  
// the FreeListEntry ->Size  
    RtlpCreateSplitBlock();  
}  
return Chunk
```

Taken

Noise

Free

Vul buffer

The target

0x200

C0

The size 0x200 vulnerable buffer

Directory Object (size 0xC0)

Taken

Noise

0x01: Initial status

Free



0x1000

0x1000

0x1000

0x1000

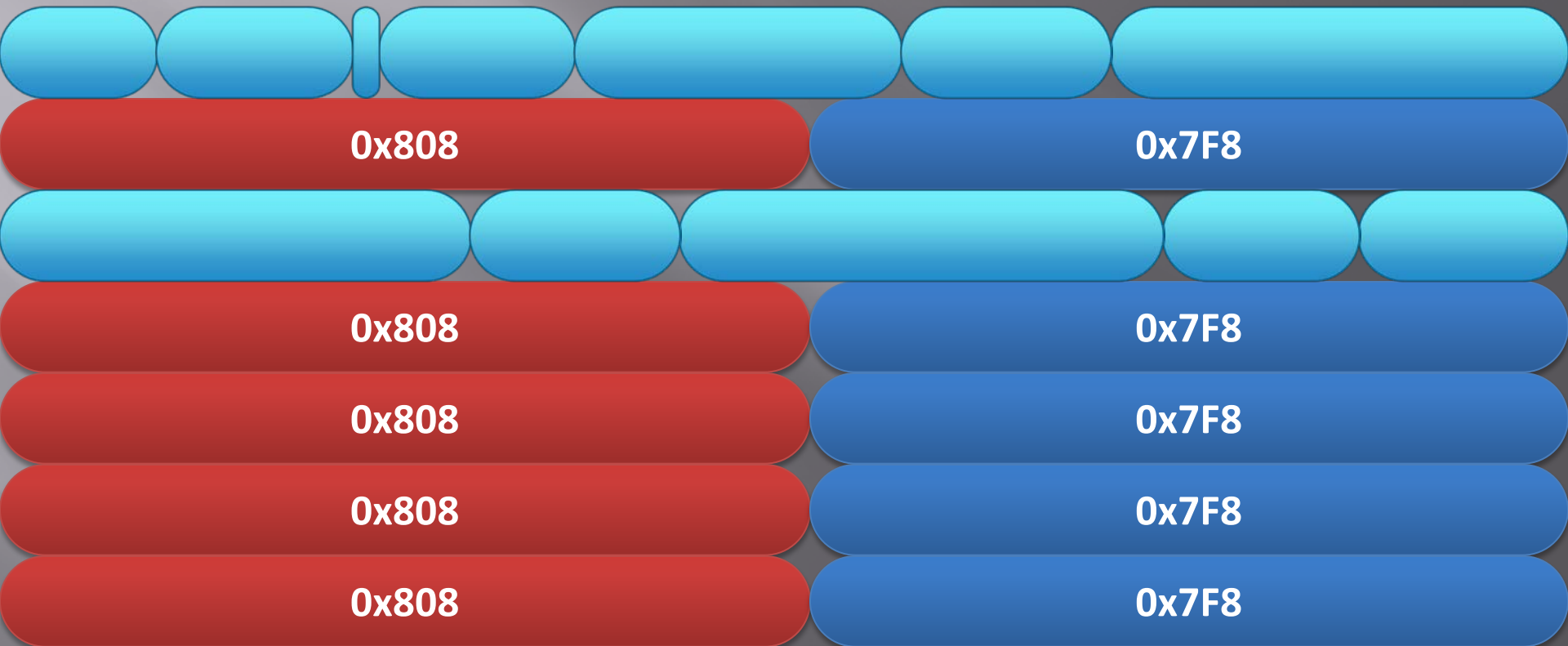
0x1000

Taken

Noise

Free

0x02: Alloc 0x808 block



Taken

Noise

Free

0x03: Alloc 0x5F8 block
and make 0x200 hole

The same size as of
vulnerable buffer



Taken

Noise

Free

0x05: Free 0x5F8 block



Taken

Noise

0x06: Alloc 0x538 block
and make 0xC0 hole

Free



Taken

Noise

0x07: Alloc 0xC0 block

Free

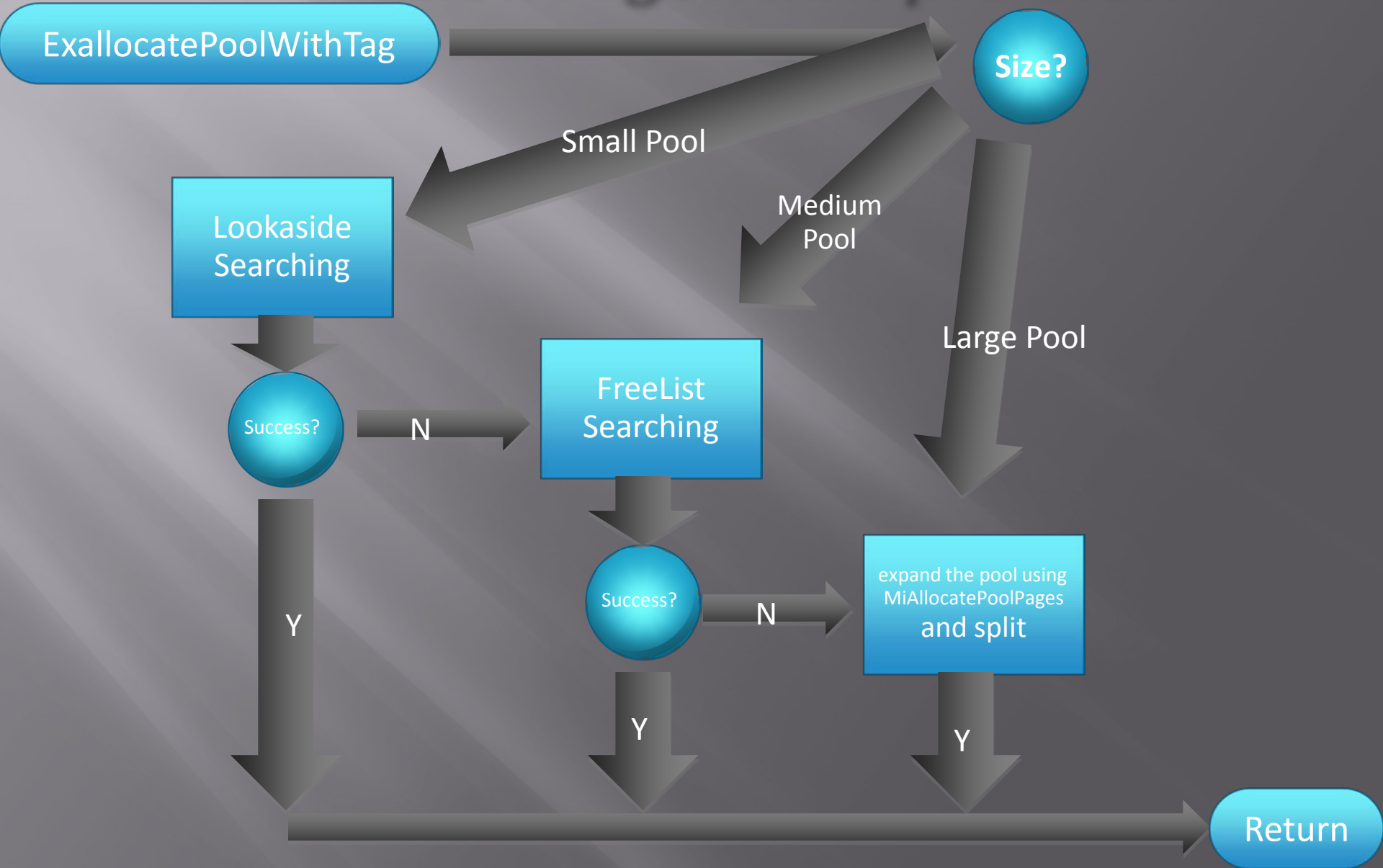
Data structure we want
corruption to



Demo of this section

0x02:
Implementation in Kernel Pool

Allocation Algorithm pre-view



Prerequisites

- Allocate Buffer of Arbitrary Size
- Free Buffer of Arbitrary Size
- Control Allocations and Frees using user code.

Example Alloc Proxy

Alloc (paged)

```
HANDLE UserAlloc(int size){
    HANDLE LinkHandle;
    std::wstring s((size - 2) / 2, 'a');
    UNICODE_STRING TargetName;
    MyRtlInitUnicodeString (&TargetName, s.c_str());
    OBJECT_ATTRIBUTES Test1;
    InitializeObjectAttributes(&Test1, NULL, 0, NULL, NULL);

    int Status = MyCreateSymbolicLinkObject(&LinkHandle,
                                           1,
                                           &Test1,
                                           &TargetName);

    return LinkHandle;
}
```

Example Free Proxy

Free

```
void UserFree(HANDLE Handle){  
    if (Handle){  
        CloseHandle(Handle);  
    }  
}
```

Massage the Kernel Pool

ExAllocatePoolWithTag()

When FreeList search failed, allocation will come from a new page.

```
82928443 bf00100000  mov  edi,1000h
82928448 57          push edi
82928449 ff742424    push dword ptr [esp+24h]
8292844d e8b3ebffff  call nt!MiAllocatePoolPages (82927005)
```

As 1000h is hard coded which leads to allocation aligned by 0x1000
(Paged , NonPaged, NonPagedNX,)

Kernel Virtual Address Space Allocation

nt!MiAllocatePoolPages

-- RtlFindClearBitsAndSet

-- MiObtainSystemVa

▣ kd> dt ntkrpamp!_RTL_BITMAP 827a1194

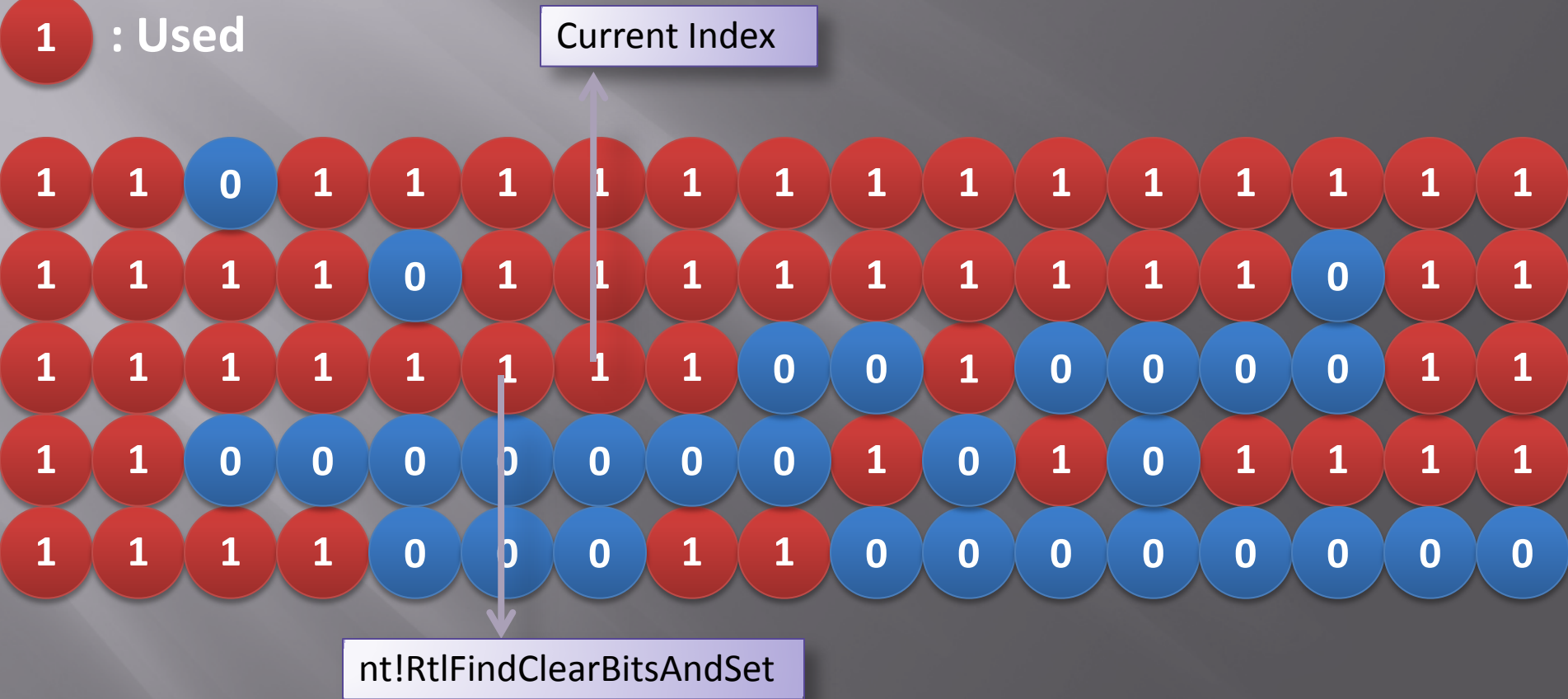
+0x000 SizeOfBitMap : 0x7fc00

+0x004 Buffer : 0x80731000 -> 0xffffffff

Request For 1 block

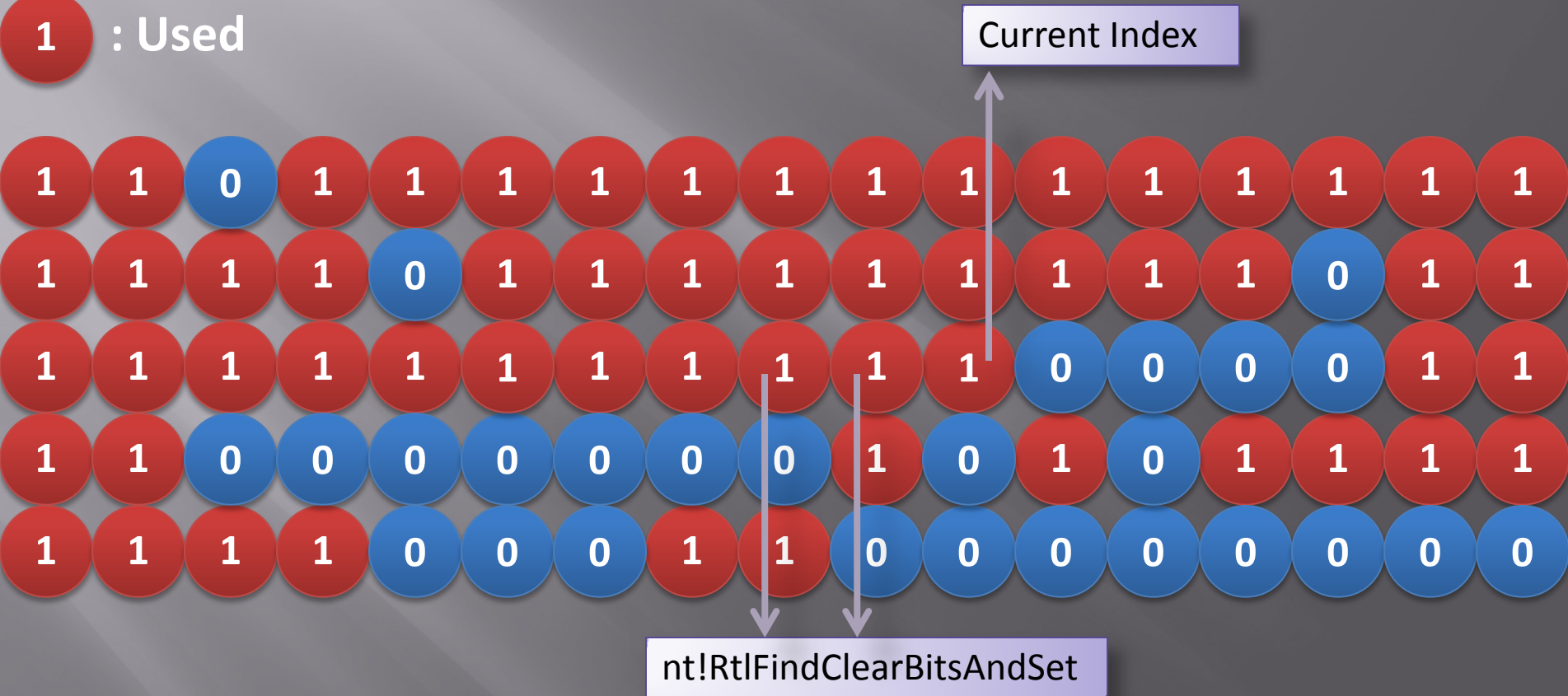
0 : Free

1 : Used



Request For 2 blocks

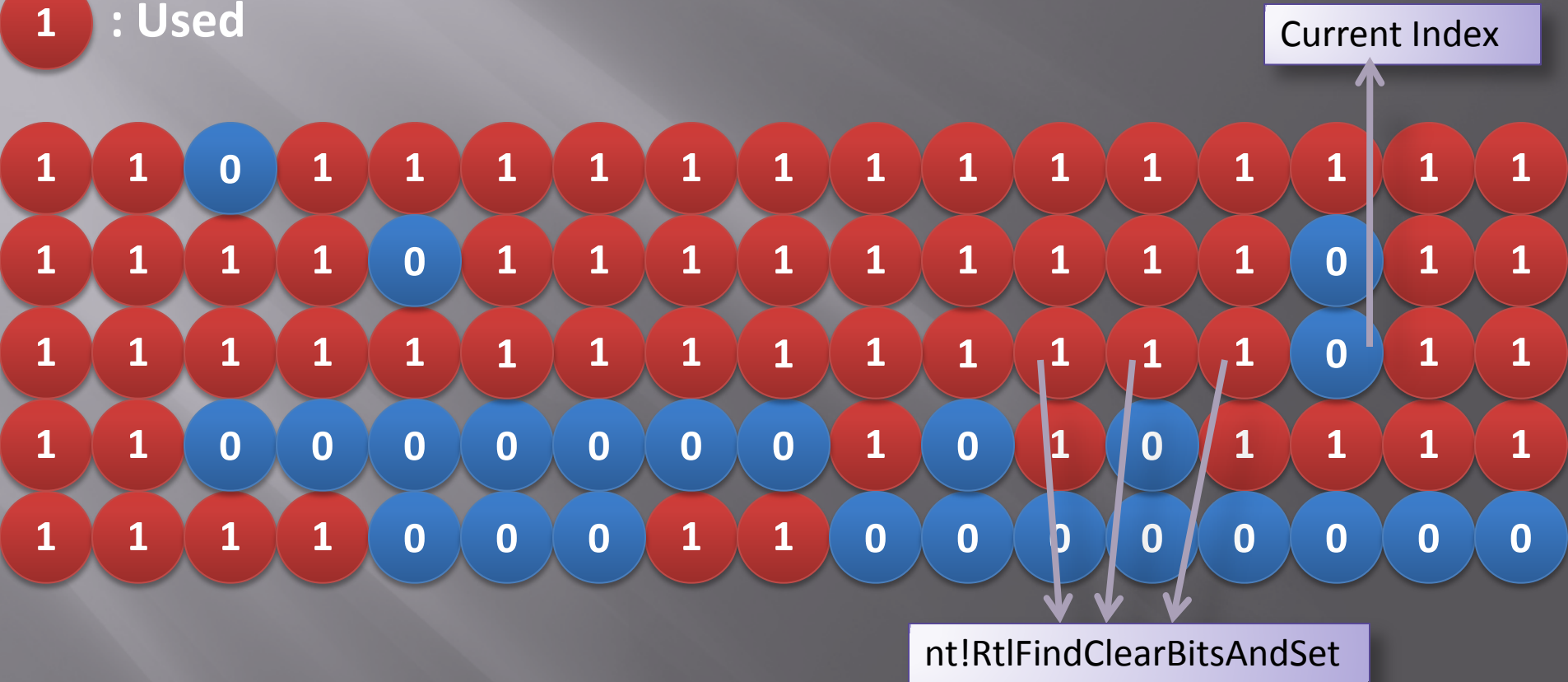
0 : Free
1 : Used



Request For 3 blocks

0 : Free

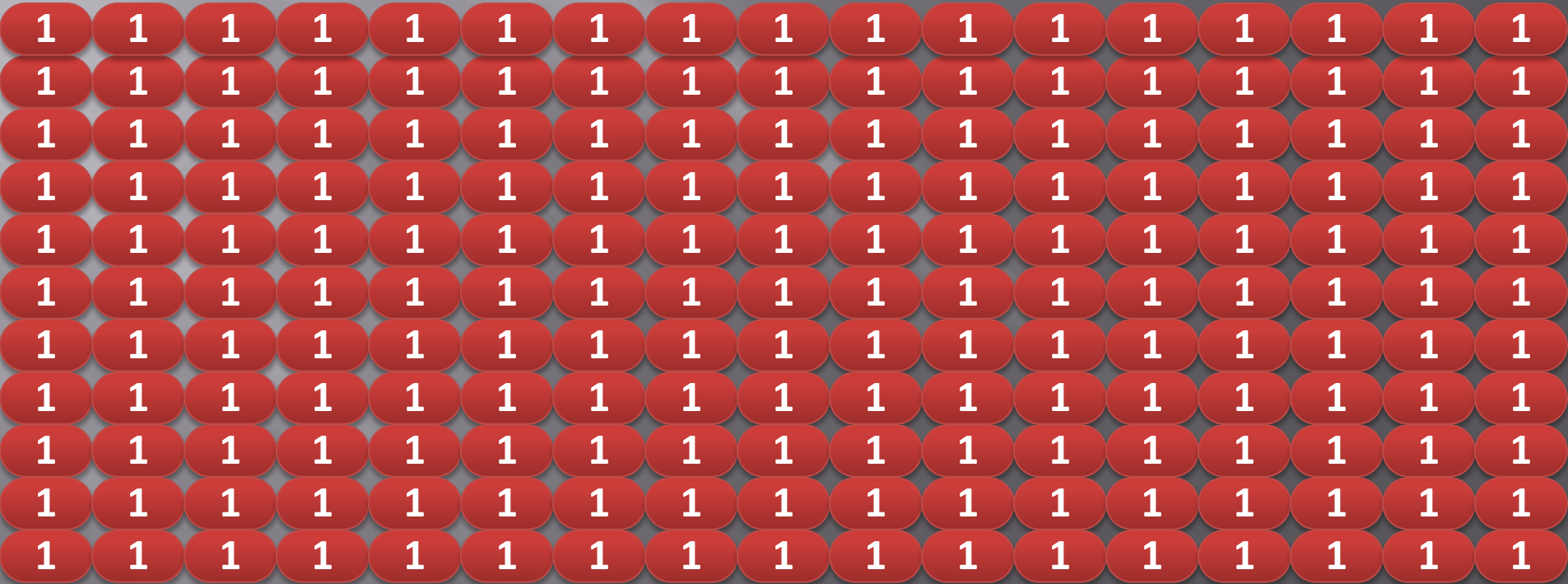
1 : Used



If all searches failed

0 : Free

1 : Used

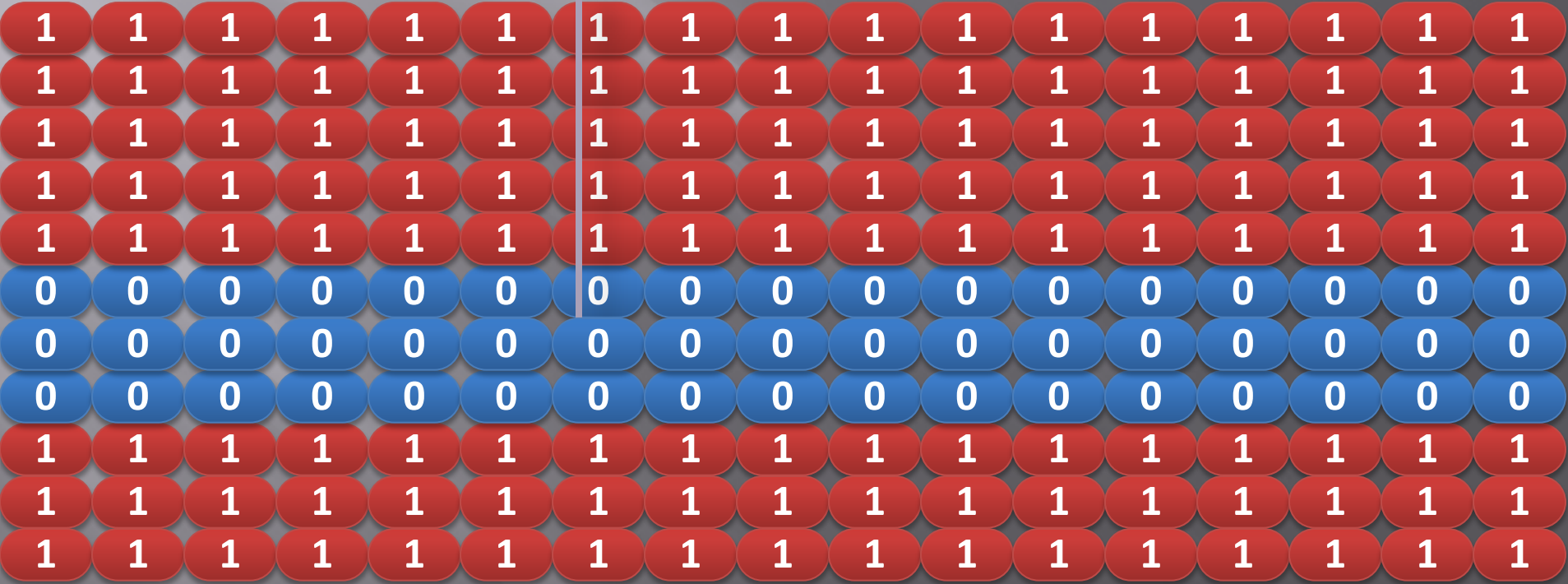


Kernel VA dynamic allocate will taken (32bit)

0 : Free

1 : Used

MiObtainSystemVa is used to dynamically allocate VA range



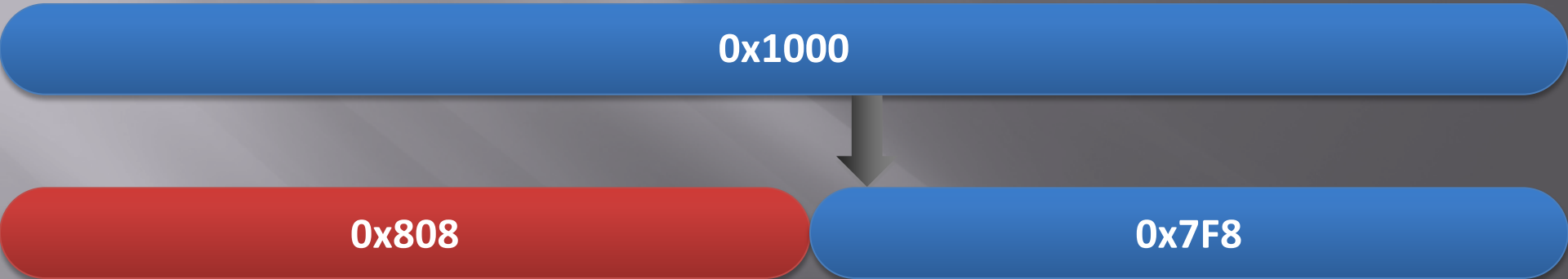
Interesting picking sequence

An empty page:

0x1000

Interesting picking sequence

1st allocation picked from front:

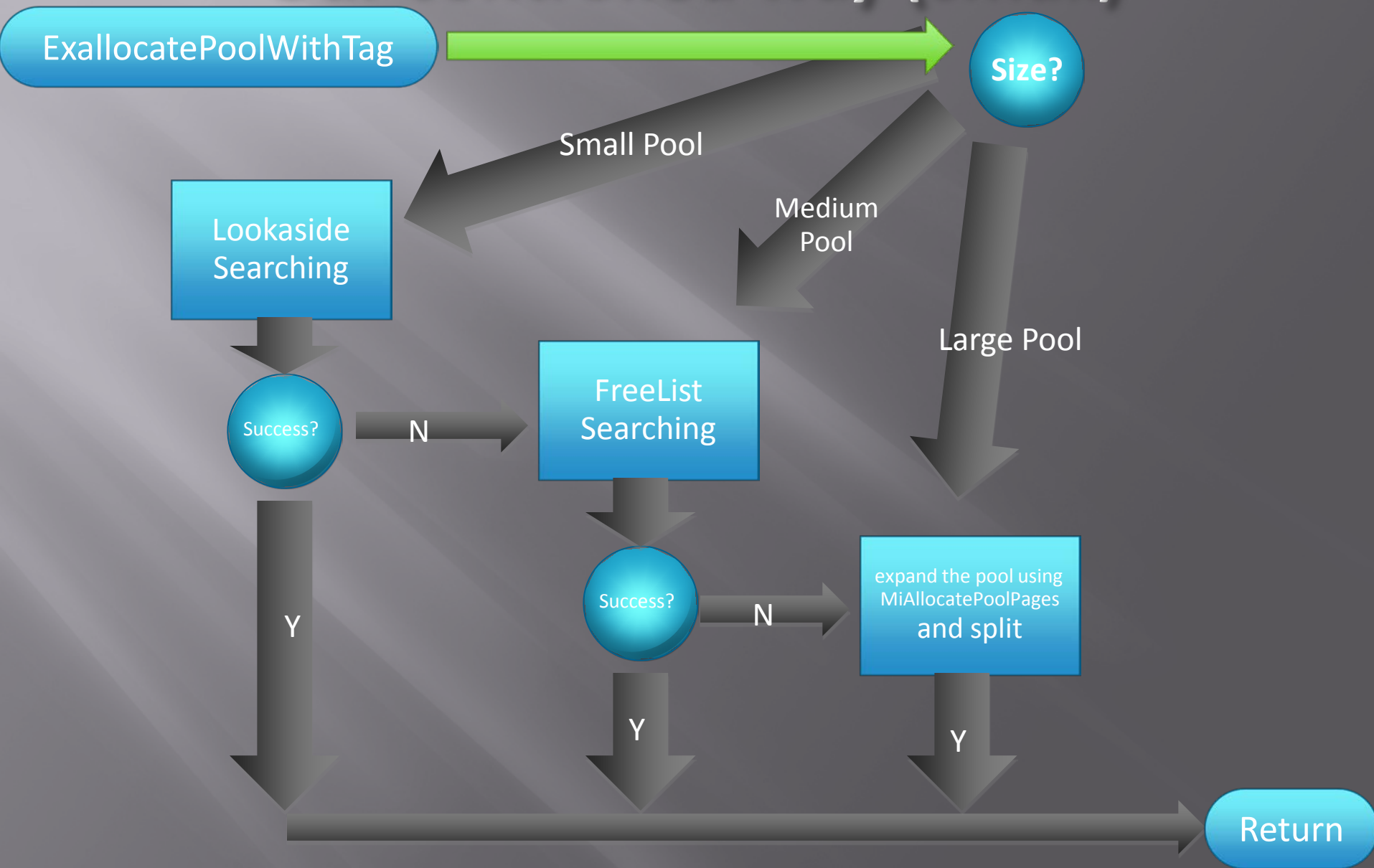


Interesting picking sequence

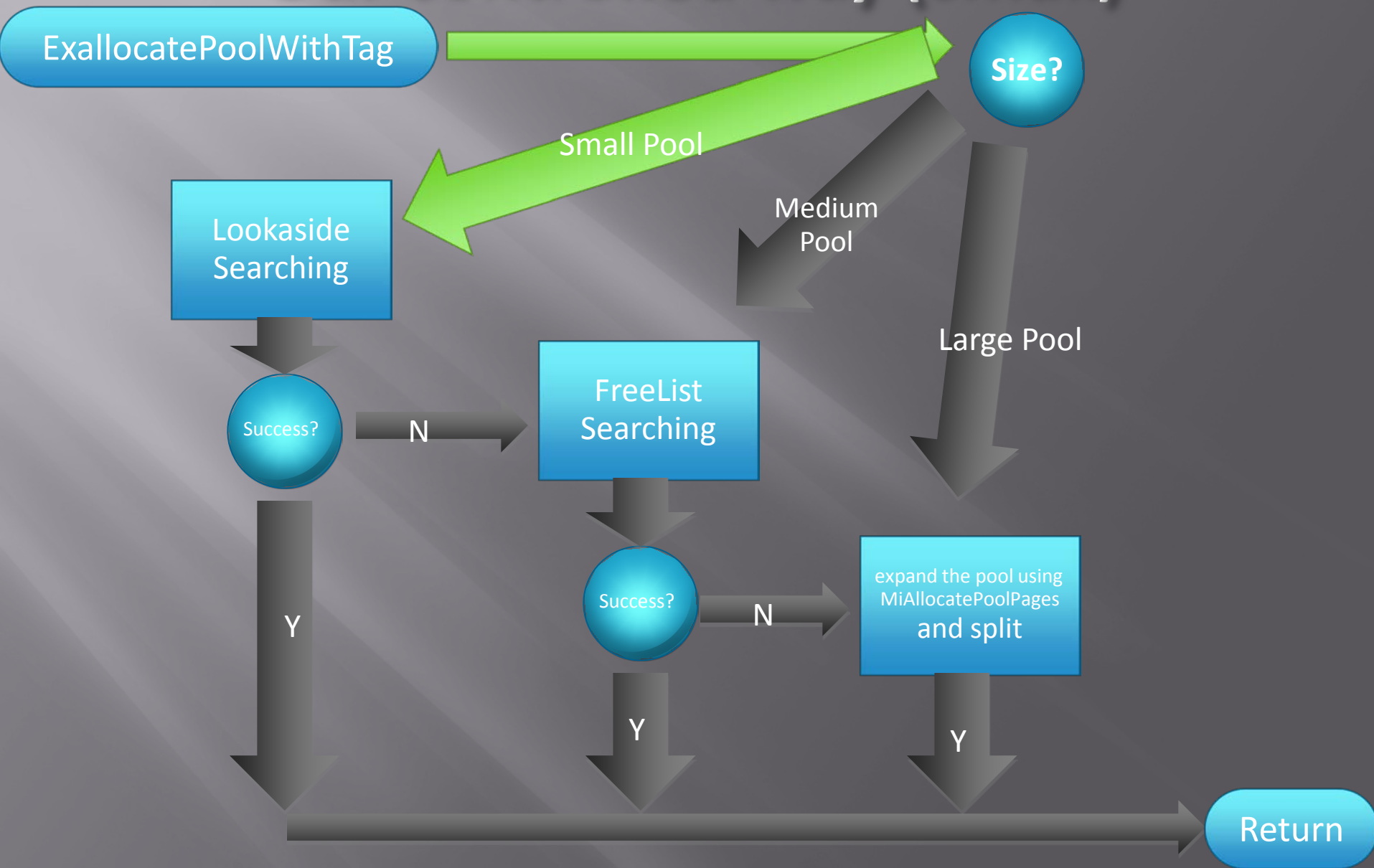
2nd allocation picked from end:



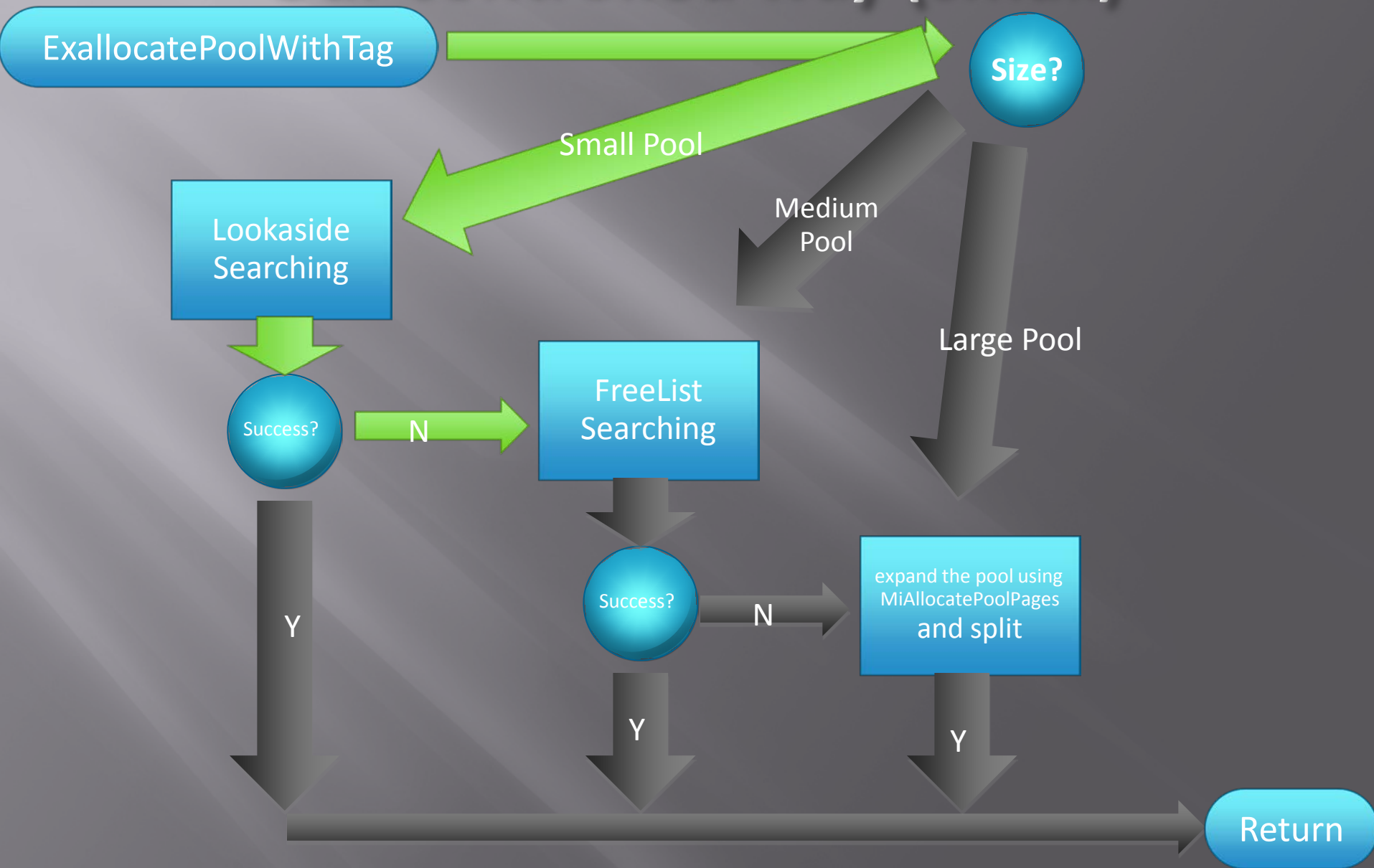
Our controlled way (small)



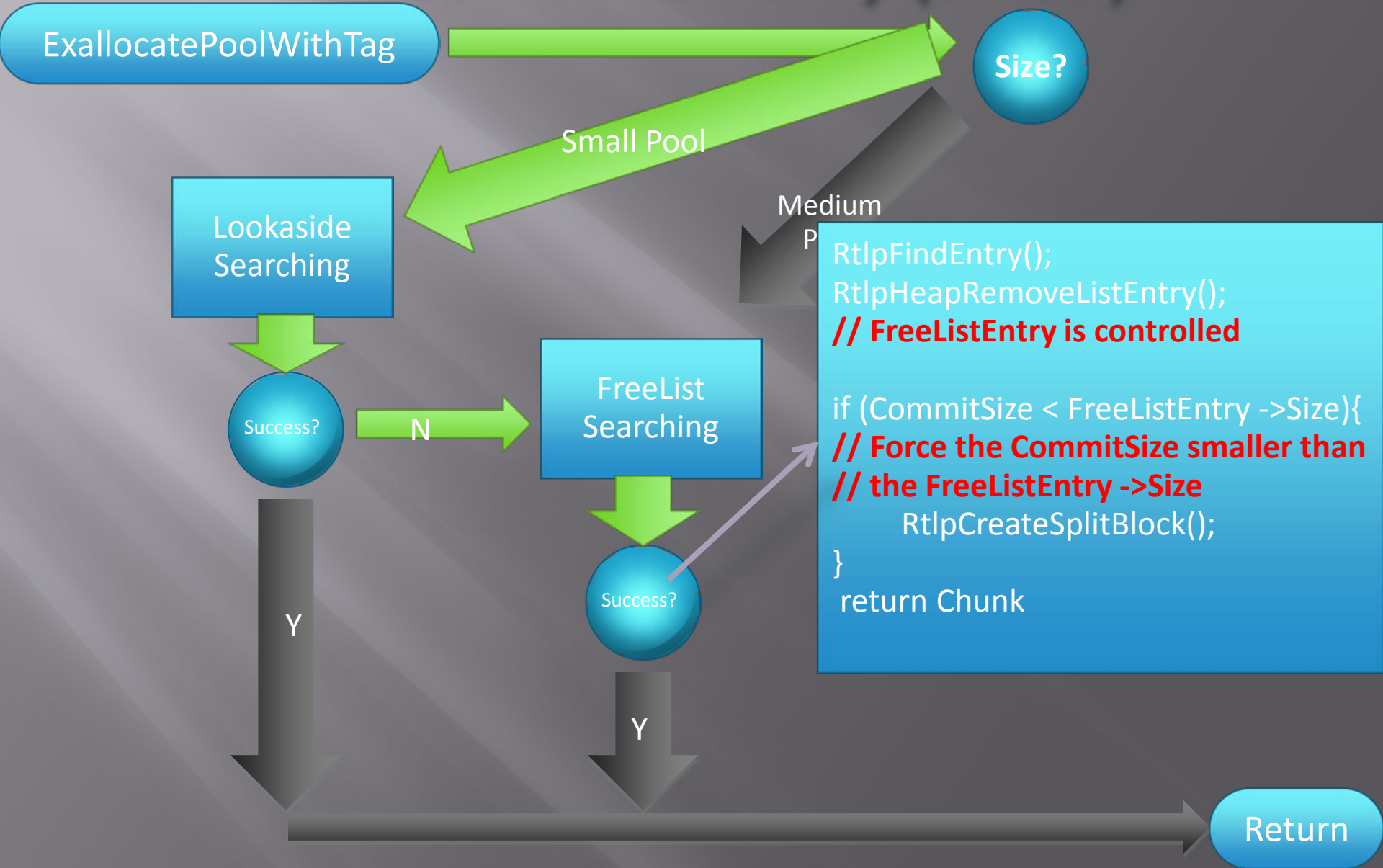
Our controlled way (small)



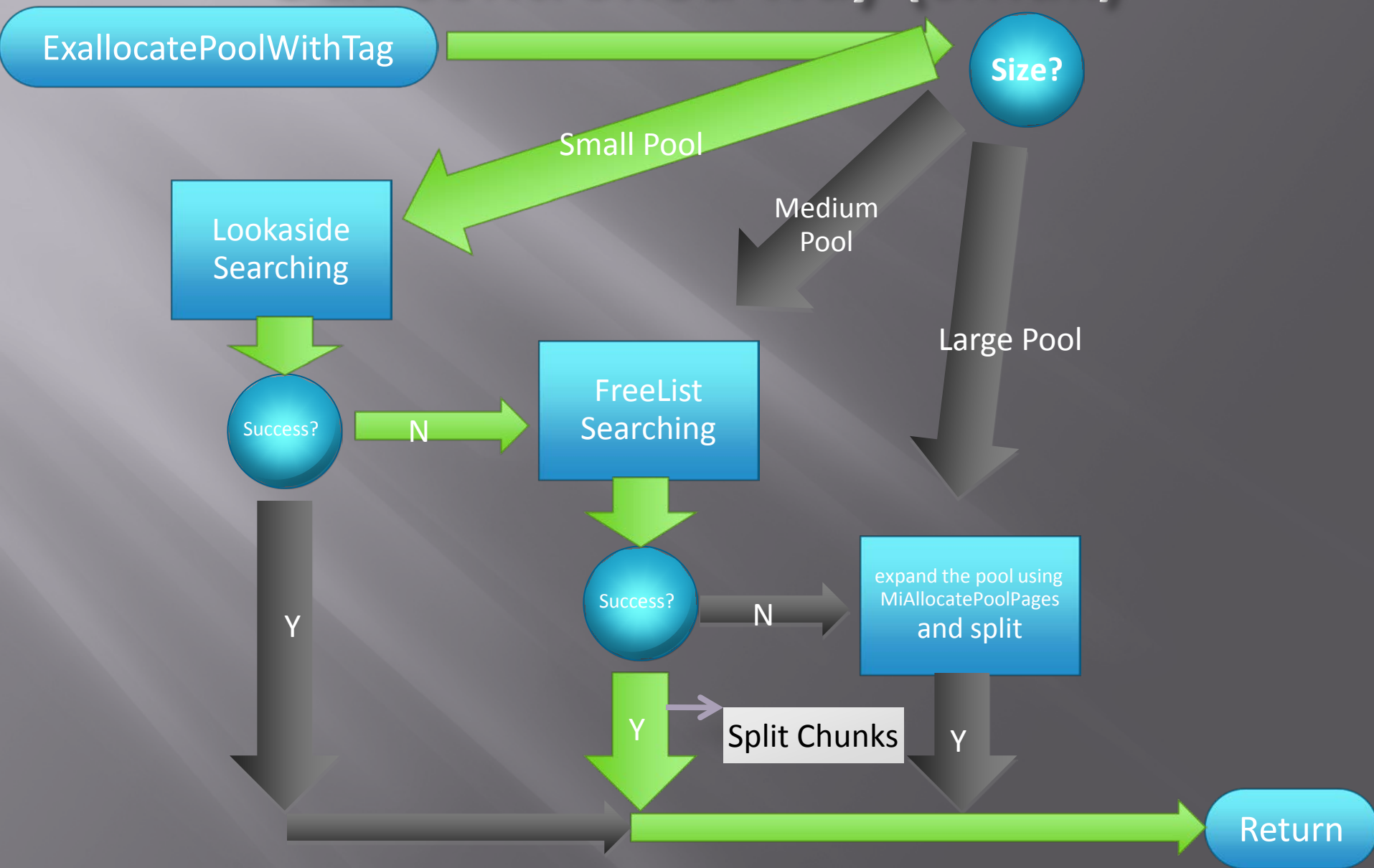
Our controlled way (small)



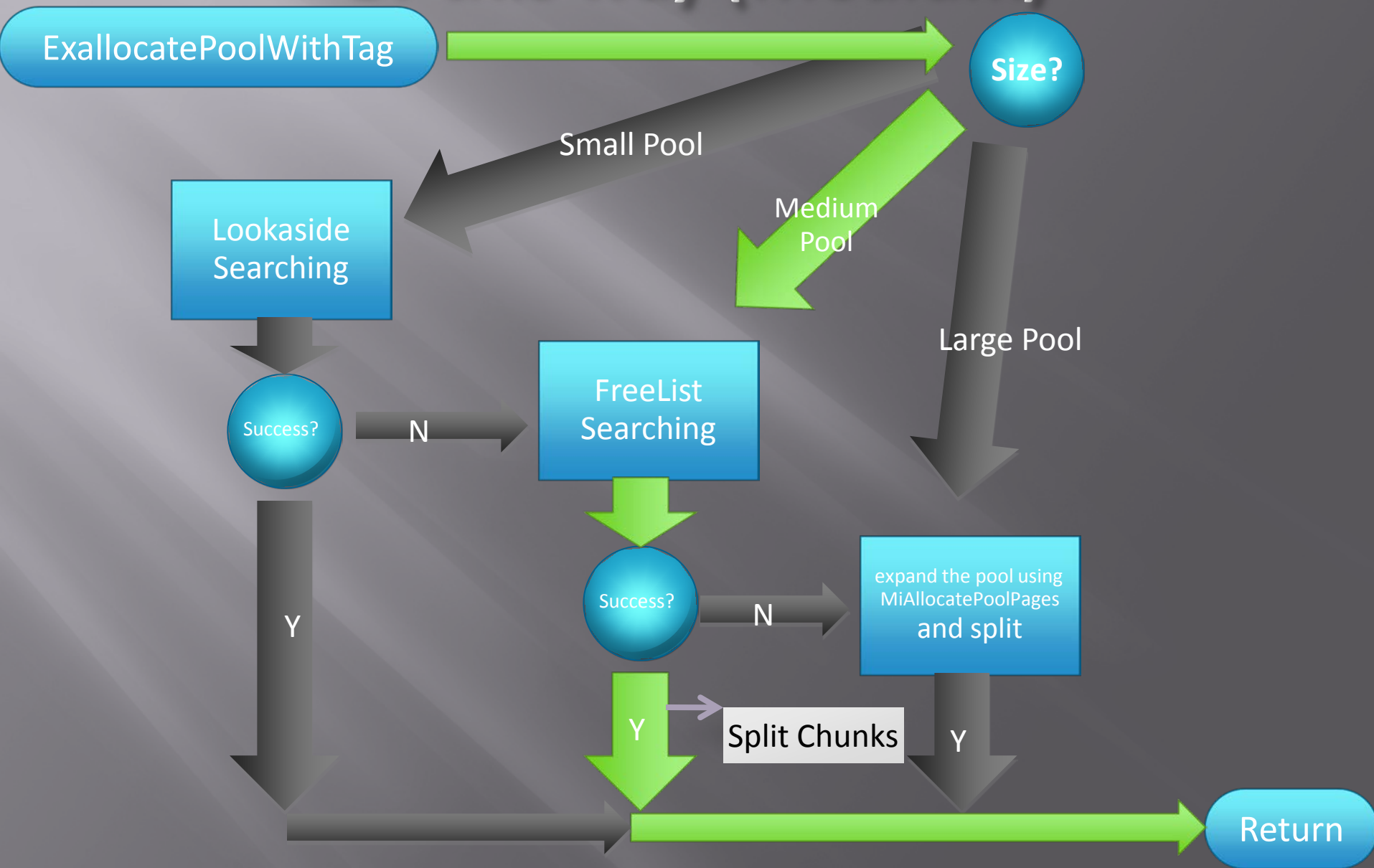
Our controlled way (small)



Our controlled way (small)



Or this way (Medium)



What about size > 0xFF0?

Daniel: Yes it will. There's always a way out...
-Quotes from Stargate SG-1 "Abyss"

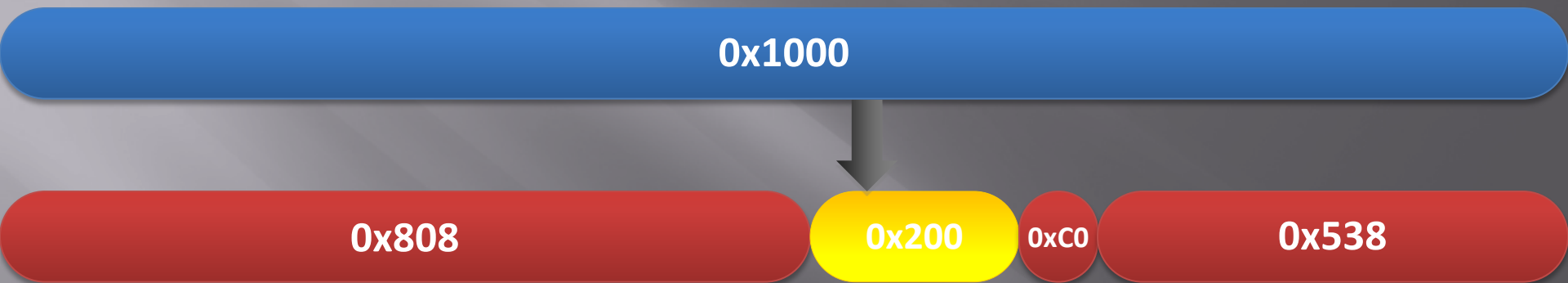
A: if (size_t < 0x400)

B: if ((size_t >= 0x400) & (size_t < 0x800))

C: if ((size_t >= 0x800) & (size_t < 0xFF0))

D: if (size_t >= 0xFF0)

A: if (size_t < 0x400)
Make holes on size 0x1000 chopping board



*B: if ((size_t < 0x400) & (size_t < 0x800))
Make holes on size 0x2000 chopping board*

0x1000

0x1000



0x1010

0x9F8

0xC0

0x538

C: if ((size_t > 0x800) & (size_t < 0xFF0))
Make holes on size 0x3000 chopping board



D: if (size_t > 0xFF0)

Vulnerable buf will be allocated by MiAllocatePoolPages directly

0x1000

0x1000



0x1010

0xC0

0xF30

2.01:
Windows Objects in
Kernel Vulnerability Exploitation

Exploitation in Windows 7

(Bonus)

```
kd> dt nt!_OBJECT_HEADER
+0x000 PointerCount   : Int4B
+0x004 HandleCount    : Int4B
+0x004 NextToFree     : Ptr32 Void
+0x008 Lock           : _EX_PUSH_LOCK
+0x00c TypeIndex      : Uchar           // used to be a Ptr in XP
+0x00d TraceFlags      : UChar
+0x00e InfoMask        : UChar
+0x00f Flags           : UChar
+0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
+0x010 QuotaBlockCharged : Ptr32 Void
+0x014 SecurityDescriptor : Ptr32 Void
+0x018 Body           : _QUAD
```

Exploitation in Windows 7 (Bonus)

0x01: InitTrampoline:

Mapping VA 0x0 through NtAllocateVirtualMemory

*0x02: Modify **TypeIndex***

Then..

0x03: *Jump into shellcode when CloseHandle()*

The diagram illustrates the execution flow of the `CloseHandle` function, highlighting a jump to shellcode. It consists of several assembly snippets and a callout box.

Main Function Snippet:

```
; Attributes: bp-based frame
; int __stdcall ObpCloseHandleTableEntry(int, int, int, ULONG_PTR BugCheckParameter1, int, char)
_ObpCloseHandleTableEntry024 proc near

var_25= byte ptr -25h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h
BugCheckParameter1= dword ptr 14h
arg_10= dword ptr 18h
arg_14= byte ptr 1Ch

mov     edi, edi
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
sub     esp, 2Ch
mov     eax, [ebp+arg_4]
push    ebx
push    esi
mov     esi, [eax]
and     esi, 0FFFFFFF8h
movzx   ecx, byte ptr [esi+0Ch]
mov     ebx, _ObTypeIndexTable[ecx*4]
push    edi
mov     edi, large fs:124h
cmp     dword ptr [ebx+74h], 0
lea     ecx, [esi+18h]
mov     [esp+38h+var_24], ecx
mov     [esp+38h+var_25], 0
jz      loc_82881C30
```

Flow Snippets:

- Snippet 1:**

```
mov     ecx, large fs:124h
mov     eax, [ebp+arg_8]
cmp     [ecx+50h], eax
jz      short loc_82881B88
```
- Snippet 2:**

```
lea     ecx, [esp+38h+var_18]
push    ecx
push    eax
; BugCheckParameter1
; KeStackAttachProcess00 ; KeStackAttachProcess(x,x)
call    _KeStackAttachProcess00
mov     [esp+38h+var_25], 1
```
- Snippet 3 (loc_82881B88):**

```
loc_82881B88:
push    [ebp+arg_10]
push    [ebp+BugCheckParameter1]
push    [esp+40h+var_24]
push    [ebp+arg_8]
call    dword ptr [ebx+74h]
test    al, al
jnz     short loc_82881C20
```

Callout Box (Shellcode):

```
mov     ebx, _ObTypeIndexTable[ecx*4]
// ecx is TypeIndex
...
call    dword ptr [ebx+74h]
```

Arrows indicate the flow from the main function to the snippets and the callout box.

Exploitation in Windows 8

(Mateusz 'j00ru' Jurczyk way)

```
kd> dt nt!_KTIMER 84247538
+0x000 Header          : _DISPATCHER_HEADER
+0x010 DueTime         : _ULARGE_INTEGER 0x4`9b8e6360
+0x018 TimerListEntry  : _LIST_ENTRY [ 0x85360160 -
                                0x82765ce4 ]
+0x020 Dpc             : 0x84247590 _KDPC
+0x024 Period          : 0x7d0
```

Exploitation in Windows 8 (Mateusz 'j00ru' Jurczyk way)

▣ kd> dt nt!_KDPC

+0x000 Type : UChar

+0x001 Importance : UChar

+0x002 Number : Uint2B

+0x004 DpcListEntry : _LIST_ENTRY

+0x00c DeferredRoutine : Ptr32 void

+0x010 DeferredContext : Ptr32 Void

+0x014 SystemArgument1 : Ptr32 Void

+0x018 SystemArgument2 : Ptr32 Void

+0x01c DpcData : Ptr32 Void

Demo of this section

0x03:
Implementation in User Heap

Allocation Algorithm pre-view

HeapAlloc(x, x, size)

Size?

size < 0x4000

0x4000 – 0x7FFFF

size > 0x7FFFF

FrontEnd
(LFH)

Activated?

N

Backend

Success?

N

VirtualAlloc

Y

Y

Y

Return

3.01:
Practical Attacking
_HEAP_USERDATA_HEADER

`_HEAP_USERDATA_HEADER`

Idea brought by Chris Valasek

$\text{Chunk} = \text{UserBlocks} + \text{RandIndex} * \text{BlockStride} + \text{FirstAllocationOffset}$

Two Challenges

18 times of allocations will trigger LFH

400 times of allocations will trigger guard pages.

GP

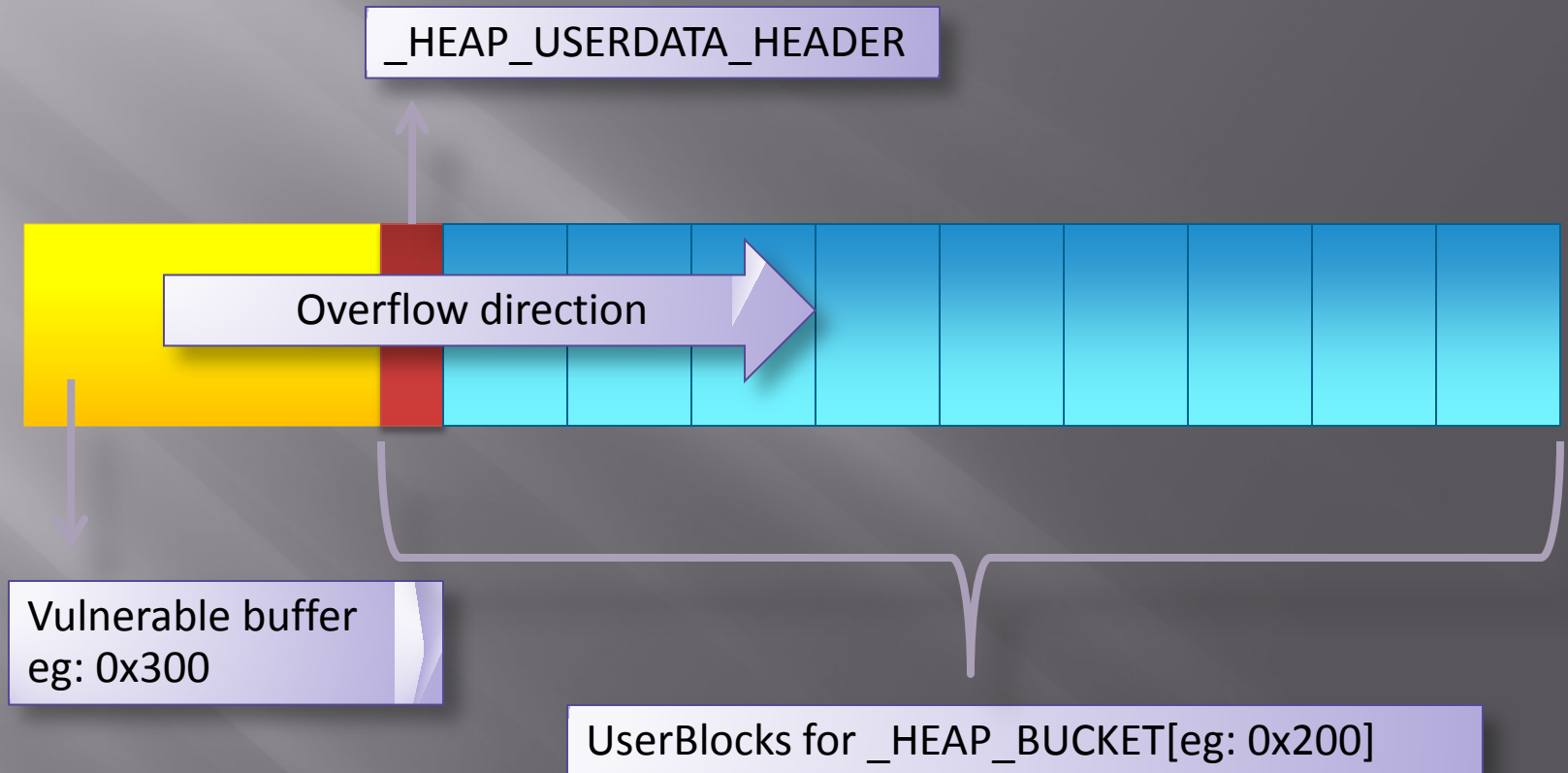
LFH

Vul buffer

LFH & Guard Pages



The target



to position the vulnerable buffer just BEFORE an important structure.

Like: `_HEAP_USERDATA_HEADER` structure

Mandatory Search in Action

Defragment using chunk 0x4000 - 0x7FFFFF.

Freeing (0x70100) --> Allocating (0x70000)
Could make 0x100 hole.

Hey, get out of my way -- LFH

The size of UserBlocks (total size) is fixed.

Taken

Noise

0x01: Defragment

Free



Taken

Noise

Free

0x02: Freeing

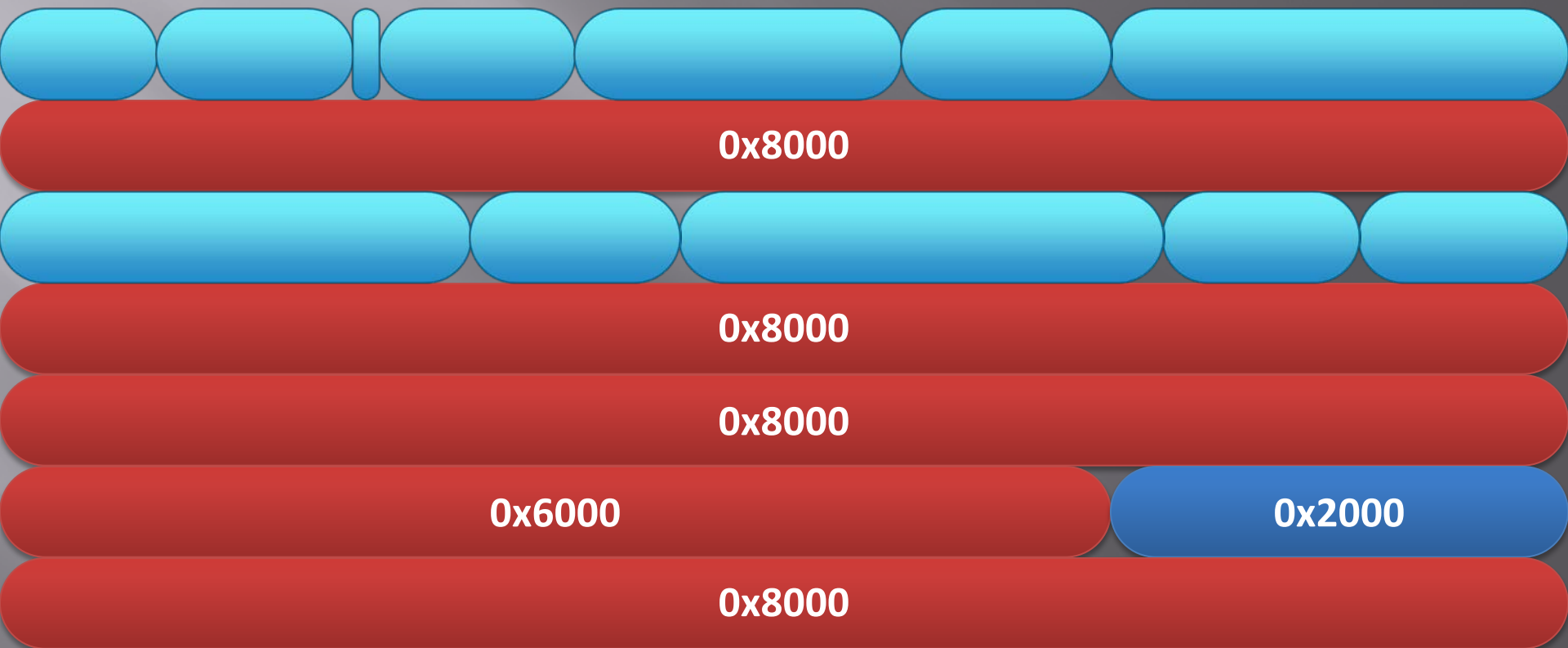


Taken

Noise

0x03: Alloc 0x6000 block
and make 0x2000 hole

Free

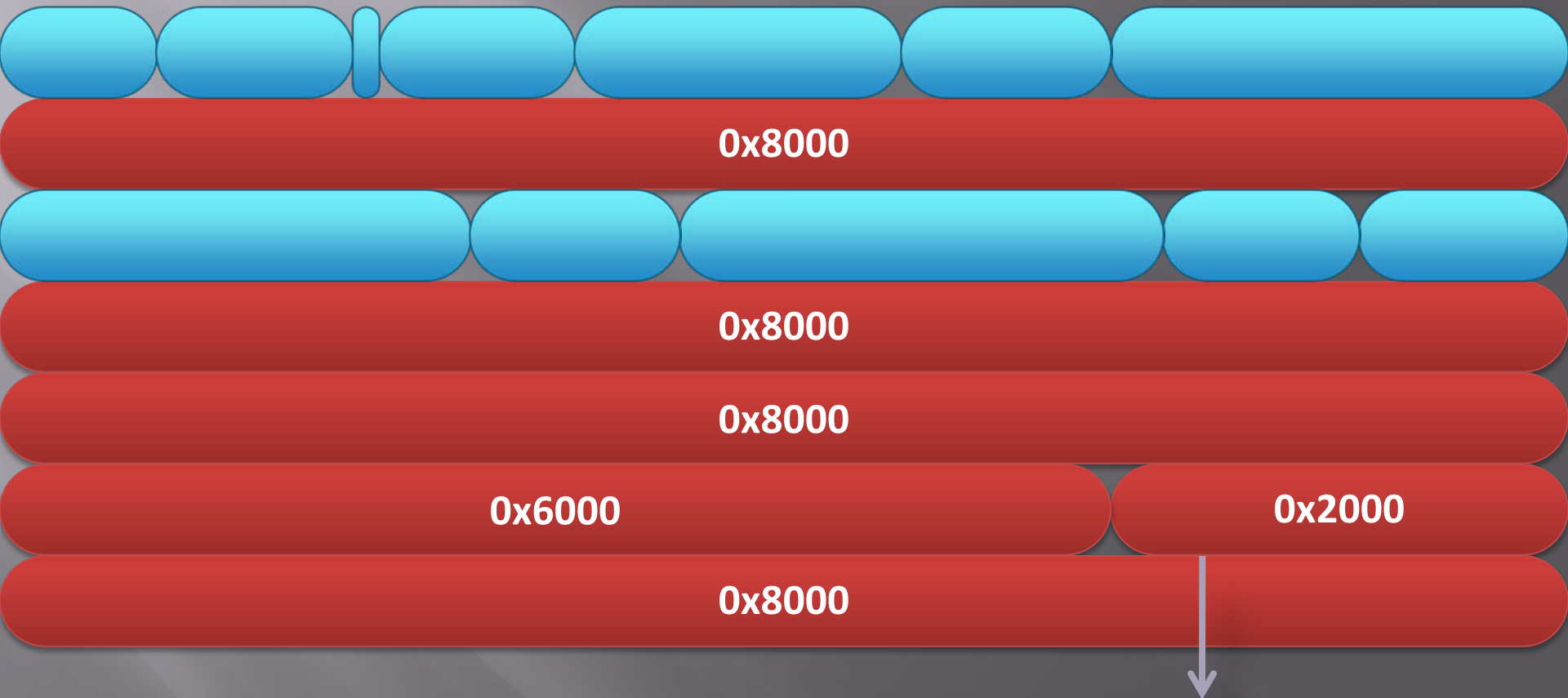


Taken

Noise

Free

0x04: Trigger LFH (0x200)



UserBlocks for _HEAP_BUCKET[0x200]

Taken

LFH

Take a closer look at

0x6000

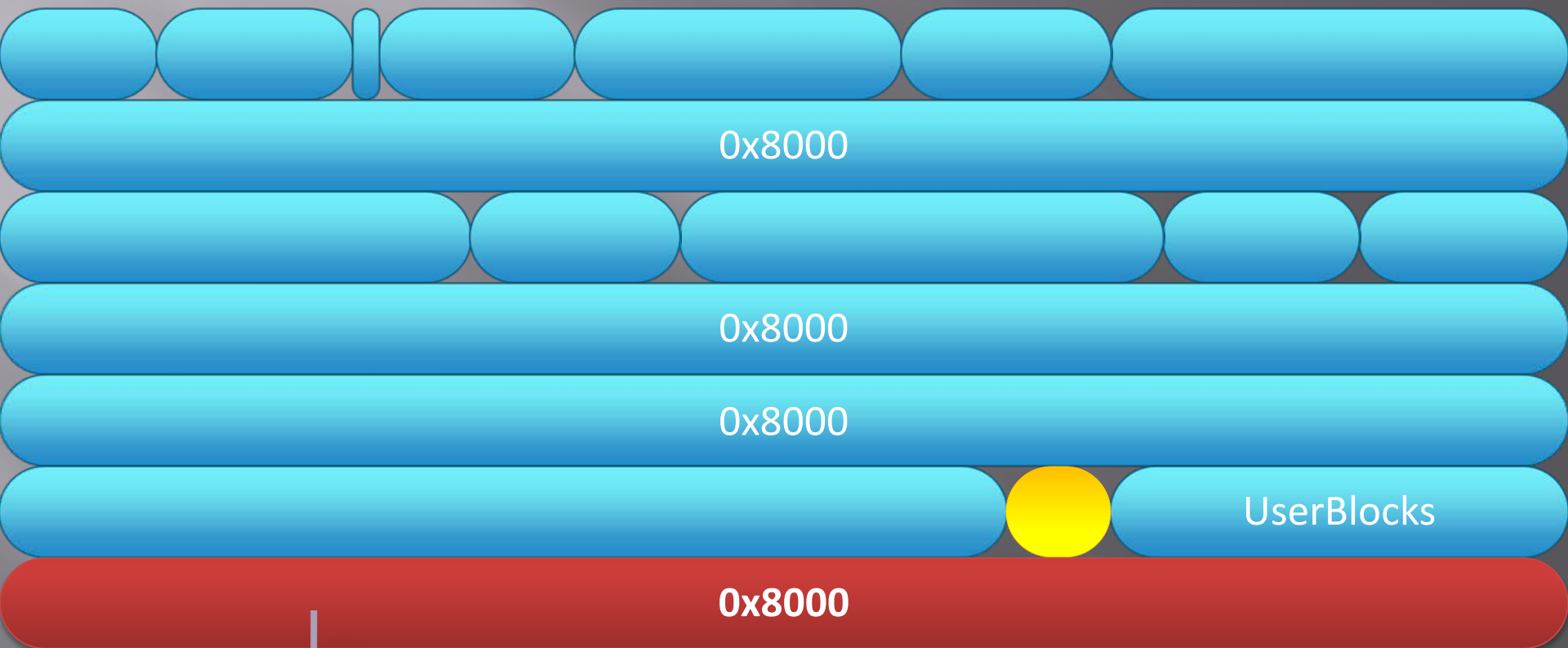
_HEAP_USERDATA_HEADER							
_HEAP_ENTRY							
_HEAP_ENTRY							
_HEAP_ENTRY							
_HEAP_ENTRY							
_HEAP_ENTRY							
_HEAP_ENTRY							
_HEAP_ENTRY							

UserBlocks for _HEAP_BUCKET[0x200]

Vul buffer

Controlled

Future allocation will get controlled after overflow



Future allocation s will fall into this controlled area.

Applicable circumstance (Prerequisites)

The LFH of the certain bin size has not been activated by the time of allocation.

(no 16 consecutive allocations of the vulnerable buffer's size)

Allocate Buffer of Arbitrary Size w/ Arbitrary Content

Free Buffer of Arbitrary Size

Programmatic Control of Allocations and Frees

The exploitation process:

Step 0: Figure out the vulnerability

Step 1: Heap Feng Shui.

Step 2: Trigger the overflow, modify "FirstAllocationOffset"

Step 3: Allocate new objects with proper size.

Step 4: Modify new object's content.

Step 5: Control the EIP.

Step 6: ...

Conclusion

Questions?