פרק 6

חולשות והמשך

Heap ועוד

חולשות Heap

- Stack Overflow הוא בעיה מוכרת.
- כיצד ניתן להתמודד?
  - Cookies
  - DEP
  - ASLR
  - להעביר את כל ה-buffer-ים ל-Heap
- ניצול חולשת Heap Overflow הוא קשה יותר, אבל אפשרי:

```
void func(void* arg, int len) {
        char* ptr = malloc(100);
        memcpy(ptr, arg, len); //buffer overflow if len>100
        ...
}
```

וכבר ראיתם דוגמא בתרגיל...

Heap overflow

- חולשות Heap הרבה פחות סטנדרטיות:
  - תלויות בארכיטקטורה
  - המבנים יותר סבוכים, ותלויים במערכת ההפעלה ולעיתים גם בקומפיילר
- המטרה הבסיסית: להגיע מכתיבה לא מורשית (Data Overflow) להרצת קוד עוין.

מה הפרימיטיב המעניין?

- לא כל החולשות מחייבות מצב של Buffer overflow
- ישנן חולשות המאפשרות כתיבת מידע למקומות בזיכרון, שתכליתן להשיג מה שמכונה **Write What Where**.
  - לעיתים זה עדיף מ-Buffer Overflow, בעיקר בהיבטים של עקיפת הגנות.
- בדרך כלל מדובר במספר קטן של בתים (4 בתים – למה?), אבל זה בהחלט מספיק כדי שהחולשה תאפשר השתלטות מלאה.
  - ניתן לשכתב Function pointer,RET ועוד...

Integer overflow

- מה יודפס?

```c
int _tmain(int argc, _TCHAR* argv[])
{
    int a=-5;
    unsigned int b=80;

if ( (unsigned int )a < b )
    printf(" %d < %d\n",a,b);
else
    printf(" %d >= %d\n",a,b);

    return 0;
}
```

Stagefright (2015)

CVE-ID	
CVE-2015-3864	Learn more at National Vulnerability Database (NVD) • Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

Integer underflow in the MPEG4Extractor::parseChunk function in MPEG4Extractor.cpp in libstagefright in mediaserver in Android before 5.1.1 LMY48M allows remote attackers to execute arbitrary code via crafted MPEG-4 data, aka internal bug 23034759. NOTE: this vulnerability exists because of an incomplete fix for CVE-2015-3824.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- MLIST:[android-security-updates] 20150909 Nexus Security Bulletin (September 2015)
- URL:https://groups.google.com/forum/message/raw?msg=android-security-updates/1M7qbSvACjo/Y7jewiW1AwAJ
- MISC:https://blog.zimperium.com/cve-2015-3864-metasploit-module-now-available-for-testing/
- MISC:https://blog.zimperium.com/reflecting-on-stagefright-patches/
- CONFIRM:https://android.googlesource.com/platform/frameworks/av/+/6fe85f7e15203e48df2cc3e8e1c4bc6ad49dc968
- BID:76682

Stagefright (bug)

From Wikipedia, the free encyclopedia

Stagefright is the group of software bugs that affect versions 2.2 ("Froyo") and newer of the Android operating system, allowing an attacker to perform arbitrary operations on the victim's device through remote code execution and privilege escalation.[1] Security researchers demonstrate the bugs with a proof of concept that sends specially crafted MMS messages to the victim device and in most cases requires no end-user actions upon message reception to succeed - the user doesn't have to do anything to 'accept' the bug – it happens in the background. The phone number is the only target information.[2][3][4][5]

Logo of the Stagefright library

The underlying attack vector exploits certain integer overflow vulnerabilities in the Android core component called "Stagefright",[6][7][a] which is a complex software library implemented primarily in C++ as part of the Android Open Source Project (AOSP) and used as a backend engine for playing various multimedia formats such as MP4 files.[5][9]

The discovered bugs have been provided with multiple Common Vulnerabilities and Exposures (CVE) identifiers, CVE-2015-1538, CVE-2015-1539, CVE-2015-3824, CVE-2015-3826, CVE-2015-3827, CVE-2015-3828, CVE-2015-3829 and CVE-2015-3864 (the latter one has been assigned separately from the others), which are collectively referred to as the Stagefright bug.[10][11][12]

Network card – דוגמא מהחיים driver remote vulnerability

- מצאו את החולשה ב-vuln.c

```c
/* Snippet code from RT73 USB Network card Driver (sanity.c - version < V1.0.5.0 ) . */

#define MAX_LEN_OF_SSID 32

typedef struct PACKED _FRAME_802_11 {
    HEADER_802_11 Hdr;
    CHAR Octet[1];
    } FRAME_802_11, *PFRAME_802_11;

BOOLEAN PeerProbeReqSanity( IN PRTMP_ADAPTER pAd, IN VOID *Msg,
     IN ULONG MsgLen, OUT PUCHAR pAddr2, OUT CHAR Ssid[], OUT UCHAR *pSsidLen)
    {
    UCHAR Idx;
    UCHAR RateLen;
    CHAR IeType;
    PFRAME_802_11 pFrame = (PFRAME_802_11)Msg;

    if ((pFrame->Octet[0] != IE_SSID) || (pFrame->Octet[1] > MAX_LEN_OF_SSID))
    {
    DBGPRINT(RT_DEBUG_TRACE, "PeerProbeReqSanity fail - wrong SSID IE(Type=%d,Len=%d)\n",
                                                pFrame->Octet[0],pFrame->Octet[1]);
    return FALSE;
    }

    *pSsidLen = pFrame->Octet[1];
    memcpy(Ssid, &pFrame->Octet[2], *pSsidLen);
    .
    .
    }
```

Use after free

- החולשה: נעשה שימוש במצביע לאחר שהוא שוחרר.
- בדרך-כלל אלו מקרי קצה או צירוף של תנאים שלא היה אמור לקרות, שכיח מאוד בהקשרים של multi-threaded.
- על-ידי מניפולציות על התוכנה, התוקף ינסה לגרום למצביע ששוחרר להצביע על מידע שהוא יכול לשלוט בו.

Use-After-Free in the Real World

[ThreatPost, September 17, 2013]

The attacks are targeting IE 8 and 9 and there's no patch for the vulnerability right now… The vulnerability exists in the way that Internet Explorer accesses an object in memory that has been deleted or has not been properly allocated. The vulnerability may corrupt memory in a way that could allow an attacker to execute arbitrary code…

The exploit was attacking a **Use After Free vulnerability** in IE's HTML rendering engine (mshtml.dll) and was implemented entirely in Javascript (no dependencies on Java, Flash etc), but did depend on a Microsoft Office DLL which was not compiled with ASLR (Address Space Layout Randomization) enabled.

The purpose of this DLL in the context of this exploit is to bypass ASLR by providing executable code at known addresses in memory, so that a hardcoded ROP (Return Oriented Programming) chain can be used to mark the pages containing shellcode (in the form of Javascript strings) as executable…

The most likely attack scenarios for this vulnerability are the typical link in an email or drive-by download.

MICROSOFT WARNS OF NEW IE ZERO DAY, EXPLOIT IN THE WILD

Dangling pointer דוגמא:

- מקרים שבהם משתנה האמור להכיל ערכים לשימוש פנימי נשלט על-ידי התוקף.
- נובע בדרך-כלל מטעות של המתכנת.

```
void function()
{
    list_node current_node;
    current_node.id = 1234;
    list_insert(list_head,current_node);

    .
    .
    .

    if ( problem )
    {
        printf( "rare error... ");
        return;
    }

    .
    .

    list_remove(list_head,current_node);

}
```

```
typedef struct list_node{
    char *data;
    int id;
    struct list_node *next;
};
```

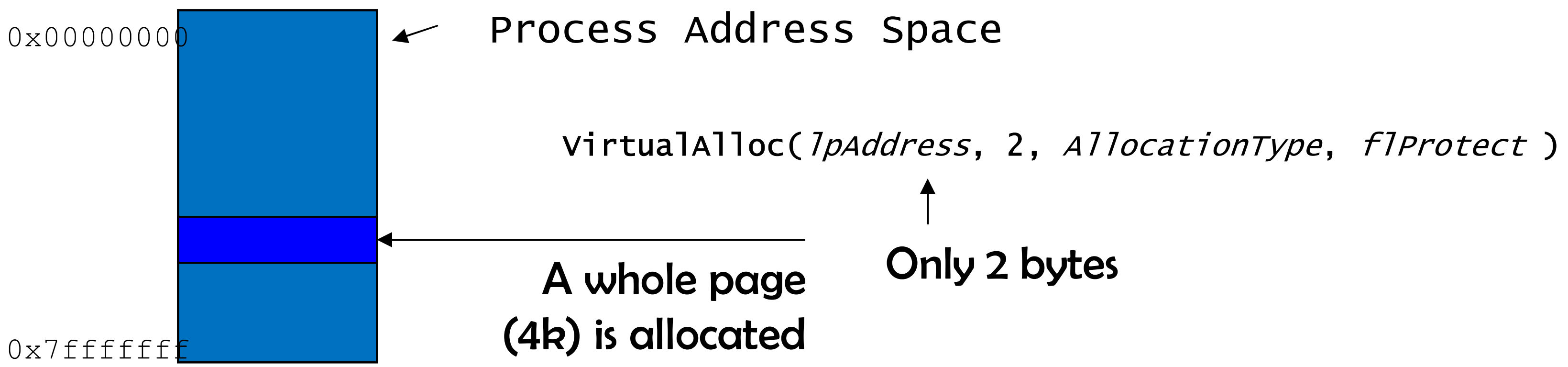
After function() returns , **current_node** is dangling on the stack, if the stack will grow enough - **current_node** will get overwritten.

Linux futex() local vulnerability.

Digging Deeper…

The Heap API

The Virtual Memory Manager allocates and deallocates with the granularity of one page (4k)



Even the stack region grows and shrink in pages (done automatically by the OS)

The Heap API

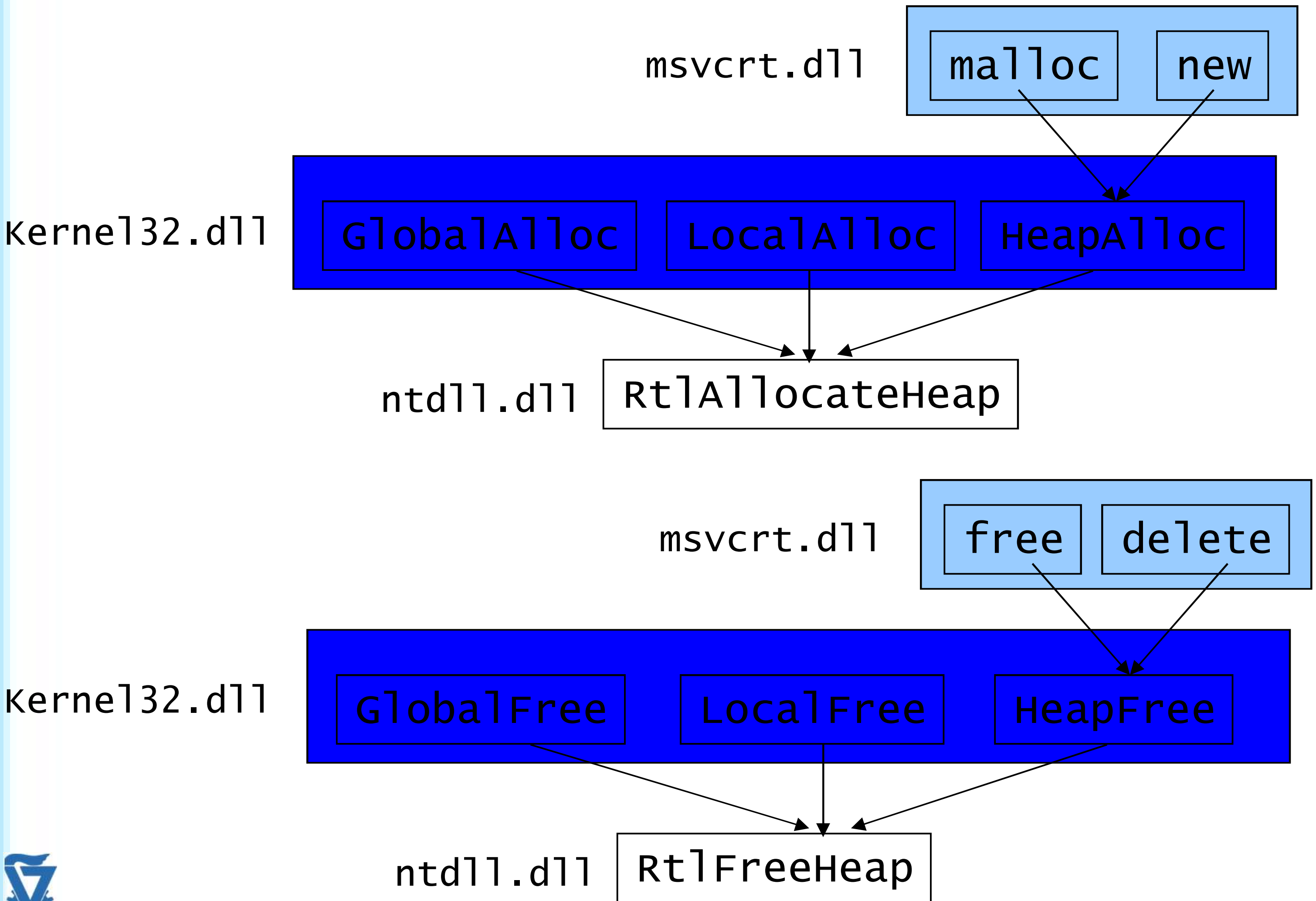
- We need a more flexible structure!! We want not to waste space and have allocations of few bytes
```
char* buff = malloc(2 * sizeof(char));
```
and of MBs
```
void* buff = malloc(1048576);
```

- This is done using the structure called *Heap*
- API for the heap is provided by the C runtime library (malloc, free, delete, new)
- In Win32, Heap API is provided by the OS too!

Heap API

- In Windows mainly following APIs are used to operate on heap.
  - HeapCreate Create a heap
    - _HEAP structure resides at the beginning of a heap and the address is returned as a HANDLE.
    - _HEAP structure contains information to manage the heap.
  - HeapAlloc Obtain specific size of memory from heap region
    - Applications can save data into the returned memory region.
    - To manage heap there is management information (Chunk header) just before allocated memory.
    - Regions not allocated yet are managed as free chunks
  - HeapFree Release obtained memory region

The Heap API


msvcrt.dll
malloc
new
Kernel32.dll
GlobalAlloc
LocalAlloc
HeapAlloc
ntdll.dll
RtlAllocateHeap
msvcrt.dll
free
delete
Kernel32.dll
GlobalFree
LocalFree
HeapFree
ntdll.dll
RtlFreeHeap

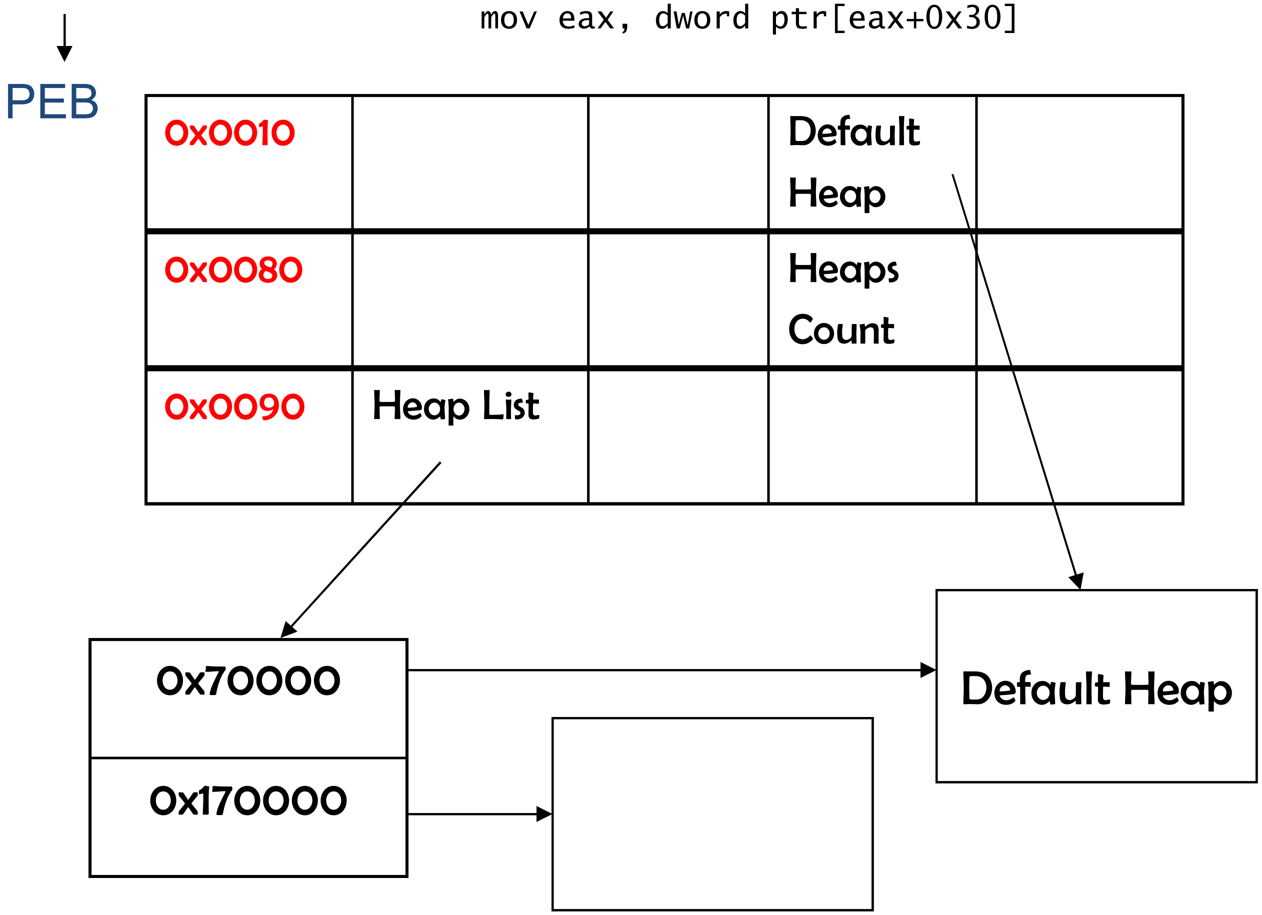
The Heap structures

- In order to handle different size, and cope well with problems (fragmentation) the Heap has a complex structure
- It uses several algorithms too:
  - Allocation algorithm (different strategies for different size)
  - Free algorithm
  - Coalesce (fusion of 2 adiacent free segments)

The Heap structures

This is the structure that holds Process properties. Its location is fixed at 7ffdf000 (or go via TEB -> PEB)

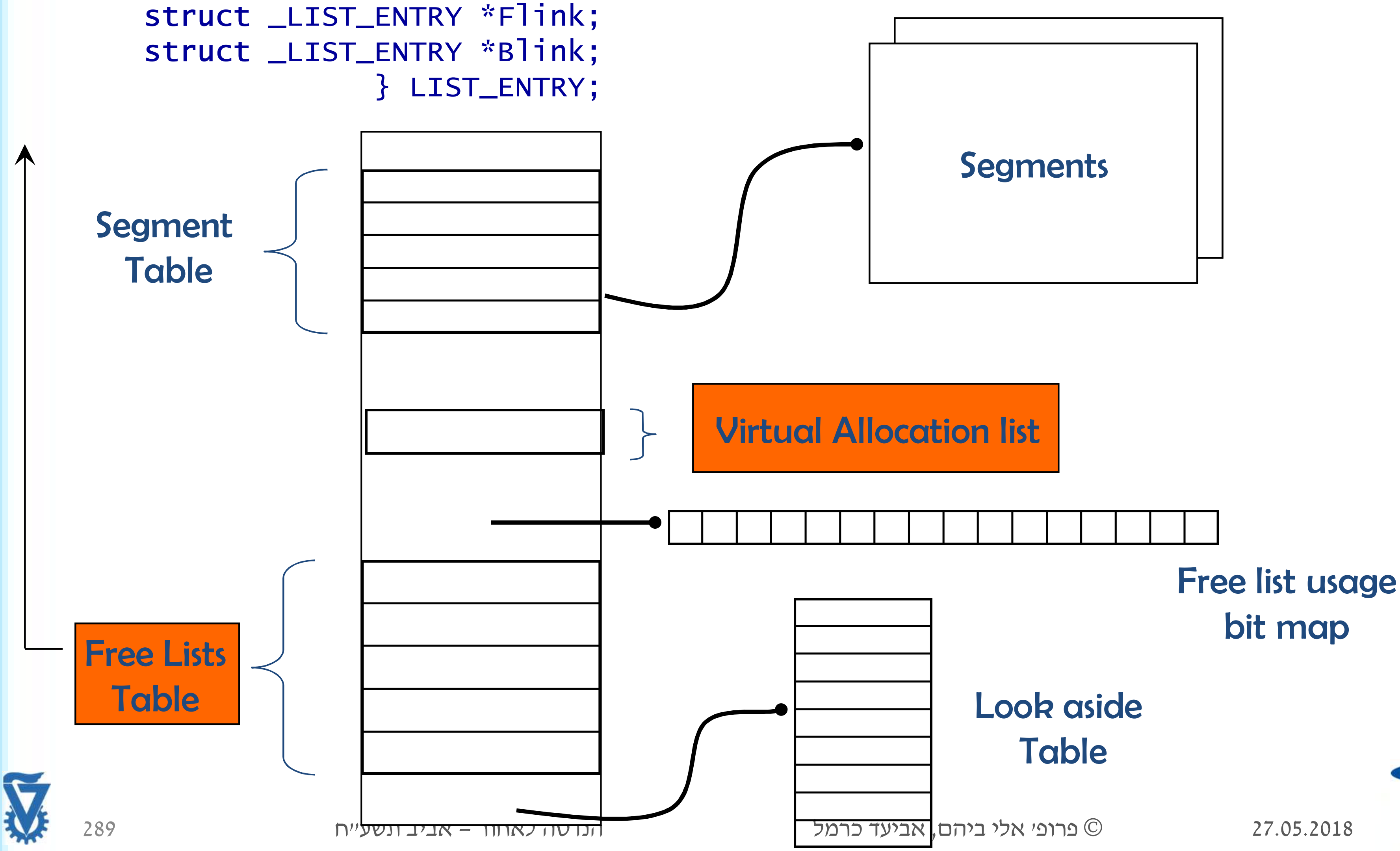
```
mov eax, dword ptr fs:[18]
mov eax, dword ptr[eax+0x30]
```



So we can easily access to the Heap address. This will be useful later!

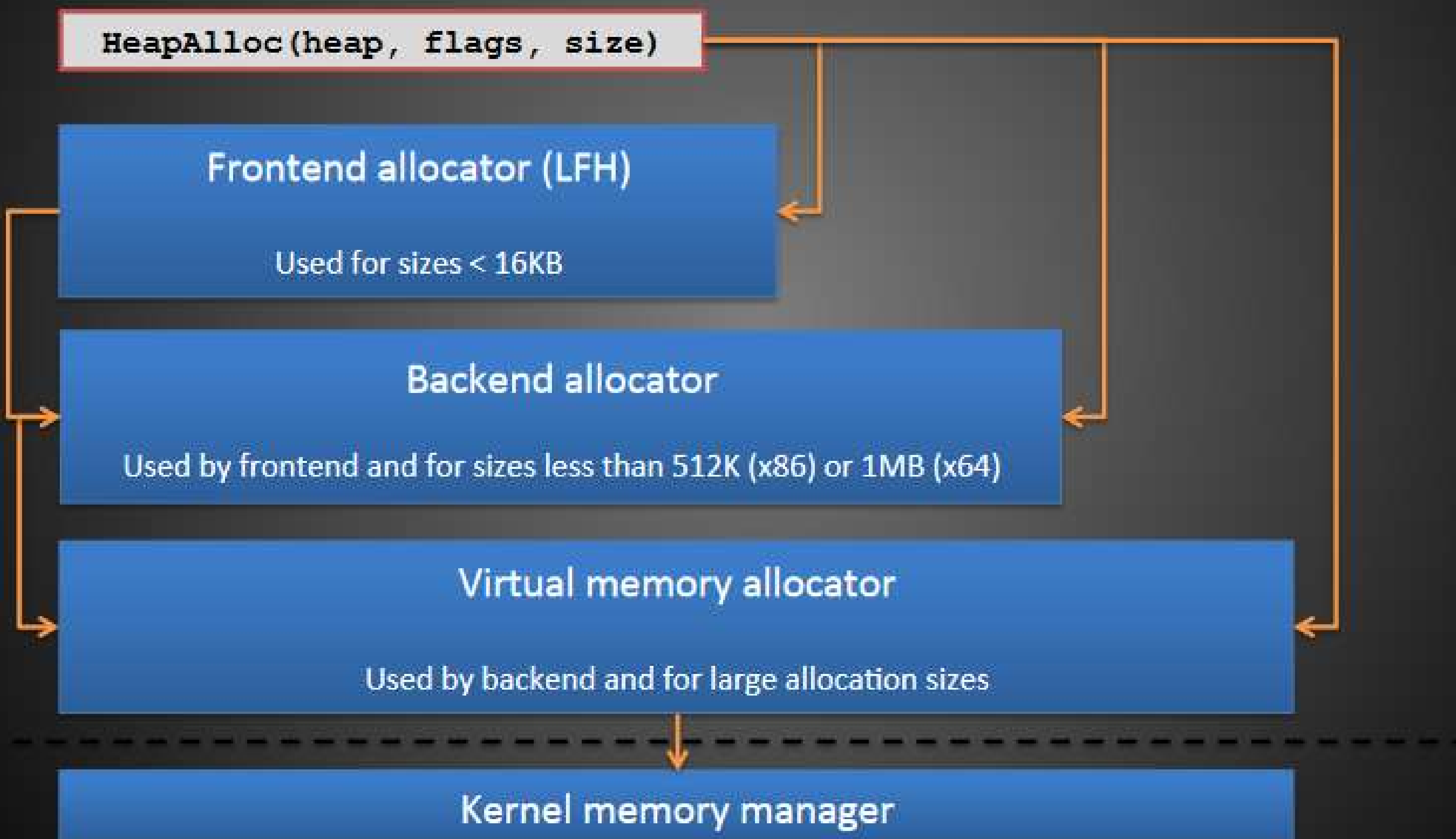
The Heap structures

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY *Flink;
  struct _LIST_ENTRY *Blink;
               } LIST_ENTRY;
```



Windows 8 heap architecture

The general design of the Windows heap is unchanged in Windows 8

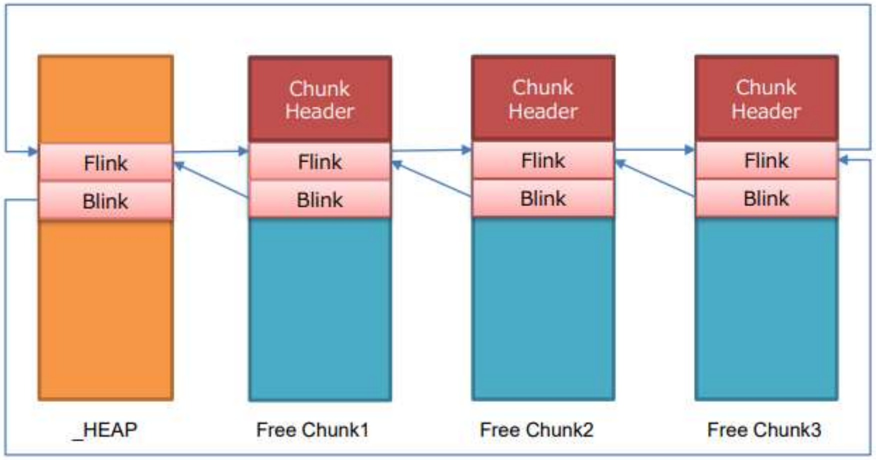


Heap Structure

- Windows heap manager consist of mainly following 2 components.
  - Frontend
  - Backend
- Frontend is an interface to an application
  - Optimizes allocating small memory blocks
  - If it is able to respond to a request, it returns a memory block
  - If not, pass the request to the backend.
  - There are 2 frontend implementations.
    - Lookaside List (LAL) on Windows XP
    - Low Fragmentation Heap on Vista or later.

Managing Free Chunks (Backend)

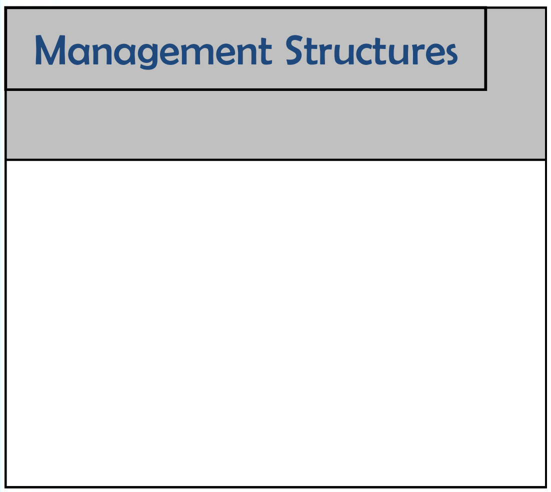
- As an application calling HeapAlloc or HeapFree in variety of order allocated chunks and free chunks are fragmented.
- To manage free chunks Windows heap manager uses doubly cyclic linked list.
- Free chunks also have a chunk header



※ In Windows XP actual free chunks are not managed as a single linked list but as a multiple linked lists. It is simplified here for the purpose of explanation of the exploit.

The Heap structures

The heap management structures reside in the heap!



When a heap is first created there are two pointers that point to the first free block set in FreeList[0]. Assuming the heap base address is 0x00350000 then first available block can be found at 0x00350688.

```
0x00350178 (FreeList[0].Flink) = 0x00350688 (First Free Block)
0x0035017C (FreeList[0].Blink) = 0x00350688 (First Free Block)

    0x00350688 (First Free Block)   = 0x00350178 (FreeList[0])
    0x0035068C (First Free Block+4) = 0x00350178 (FreeList[0])
```

The Heap structures

Access violation / Memory overwrite

Observe the following code:

```
mov dword ptr [ecx],eax
```

If we own both EAX and ECX we have an *arbitrary DWORD overwrite*. We can overwrite the data at any 32bit address with a 32bit value of our choosing.

had

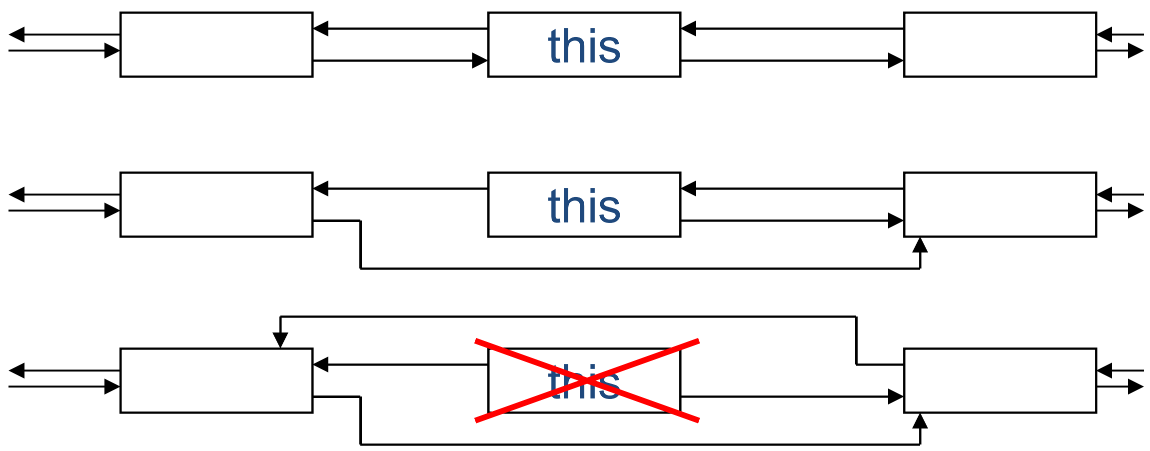
In `RtlHeapFree` we ~~have~~ such a line of code!!

Heap Structures

The instructions to remove an entry from a double linked list are:

```
prev_chunk->FLink = next_chunk
next_chunk->BLink = prev_chunk
```

Where `next_chunk` is `this->FLink`
and `prev_chunk` is `this->BLink`



Heap smashing

If we assume that prev_chunk->FLink is loaded in ECX (and since prev_chunk->FLink == &Prev_chunk + 0 in ECX is prev_chunk)
And that next_chunk is in EAX…

If we build a fake header with prev_chunk and next_chunk of our choiche we have (from the previous code)

```
mov dword ptr [ecx],eax
```

```
EAX (Address A) written in the address
pointed by ECX (Address B)
```

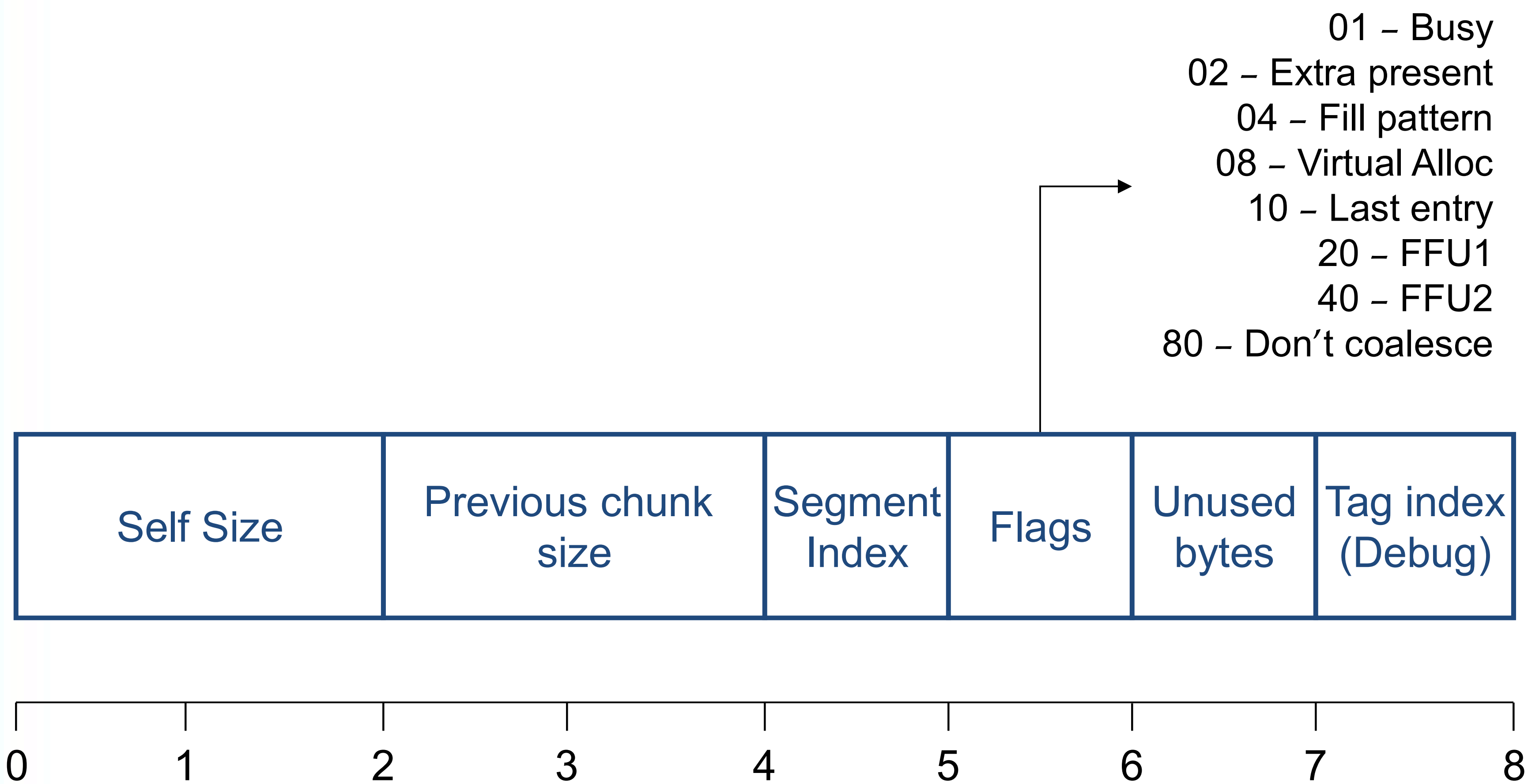
```
Arbitrary memory overwrite will happen on
free of our faked chunk!
```

Heap smashing

In fact, two heap algorithms (Free of very large (I.e. virtually allocated) blocks and coalesce) adjusted the list in this way!!

```
ntdll!RtlpCoalesceFreeBlocks+0x2fb:
784ac8d4 call     ntdll!RtlpUpdateIndexRemoveBlock (784624a3)
784ac8d9 mov      ecx,[edi+0xc]
784ac8dc mov      eax,[edi+0x8]
784ac8df cmp      eax,ecx
784ac8e1 mov      [ecx],eax          ds:0023:f1f1f1f1=????????
784ac8e3 mov      [eax+0x4],ecx
```

The Heap structures



The Heap structures

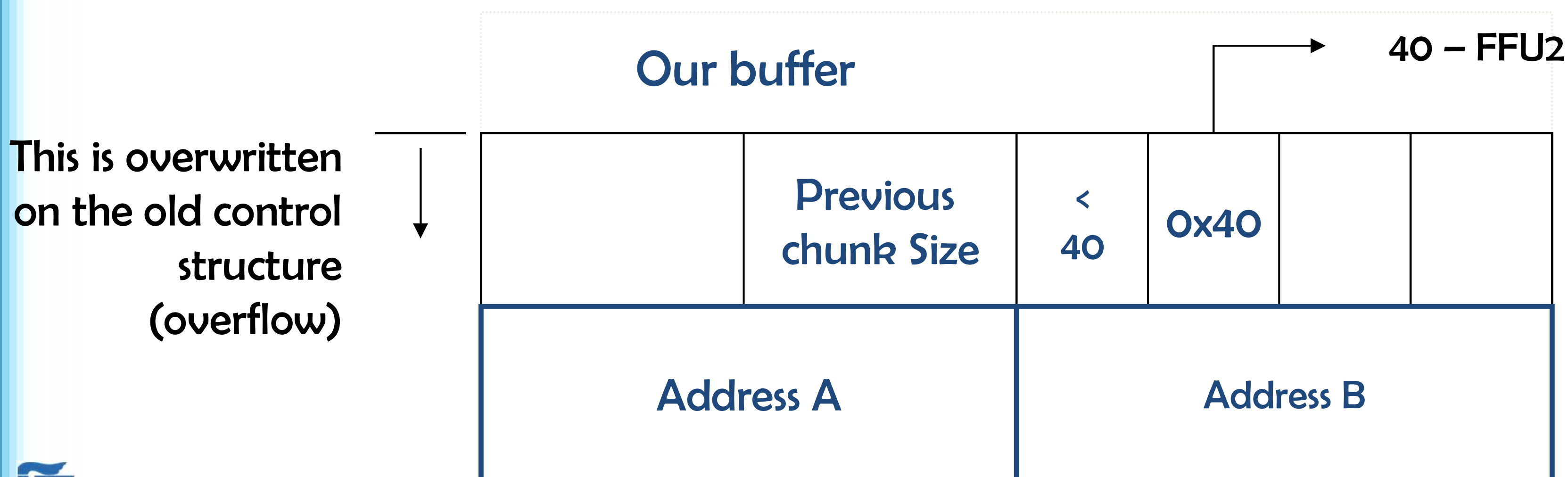
Free chunk structure – 16 Bytes

<table>
<tr><td colspan="2">Self Size</td><td colspan="2">Previous chunk size</td><td>Segment Index</td><td>Flags</td><td>Unused bytes</td><td>Tag index (Debug)</td></tr>
<tr><td colspan="4">Next chunk</td><td colspan="4">Previous chunk</td></tr>
</table>

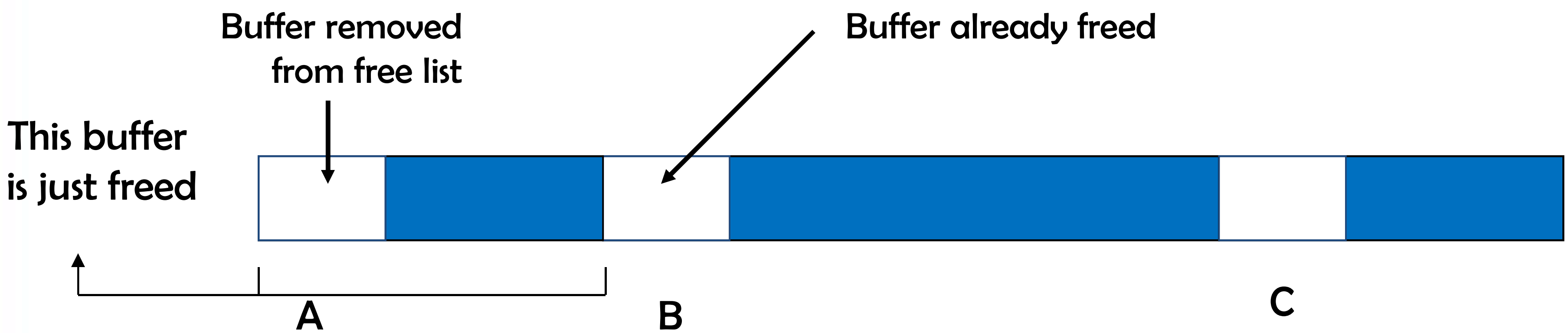
0 1 2 3 4 5 6 7 8

Heap smashing

- Exploit: fake a freed buffer
- Utilize coalescing algorithms of the heap
- Arbitrary overwrite happens when either the overflowed buffer gets freed (usually guaranteed) or when the buffer AFTER the faked buffer gets freed



Heap smashing: coalesce algo




```
B.BLink.FLink (A.FLink) = B.Flink (C)
B.FLink.Blink (C.BLink) = B.Blink (A)
```

```
   *(B.Blink) = B.Flink
=> *AddressB = AddressA
```

Arbitrary memory overwrite!

Heap smashing: repairing the heap

- Many of the Windows API calls use the default process heap.
- After the overflow the heap is corrupt, so there will be surely an access violation.
- We then repair the heap following Litchfield's method: we reset the heap making it "appear" as if it is a fresh new heap.

Heap smashing: repairing the heap

- Get a pointer to the Thread Information Block at fs:[18]
- Get a pointer to the Process Environment Block from the TEB.
- Get a pointer to the default process heap from the PEB
- We now have a pointer to the heap. Read the TotalFreeSize dword of the heap structure (at offset 0x28)
- Write this to our heap control structure.
- In the heap control structure, also set the flags to 0x14 (first segment) and and the next 2 bytes to 0
- At heap_base+0x178 we have FreeLists[0]. Set FreeLists[0].Flink and write edx into FreeLists[0].Blink to this
- Finally set the pointers at the end of our block to point to FreeLists[0]

Arbitrary memory write

- So, as in the data pointer overwrite, we have an arbitrary DWORD memory overwrite
- This can be used to overwrite particular function pointers (Exception handlers VEH + SEH, atexit(), stack cookie exception handler, Lock and Win32k function pointers in PEB)

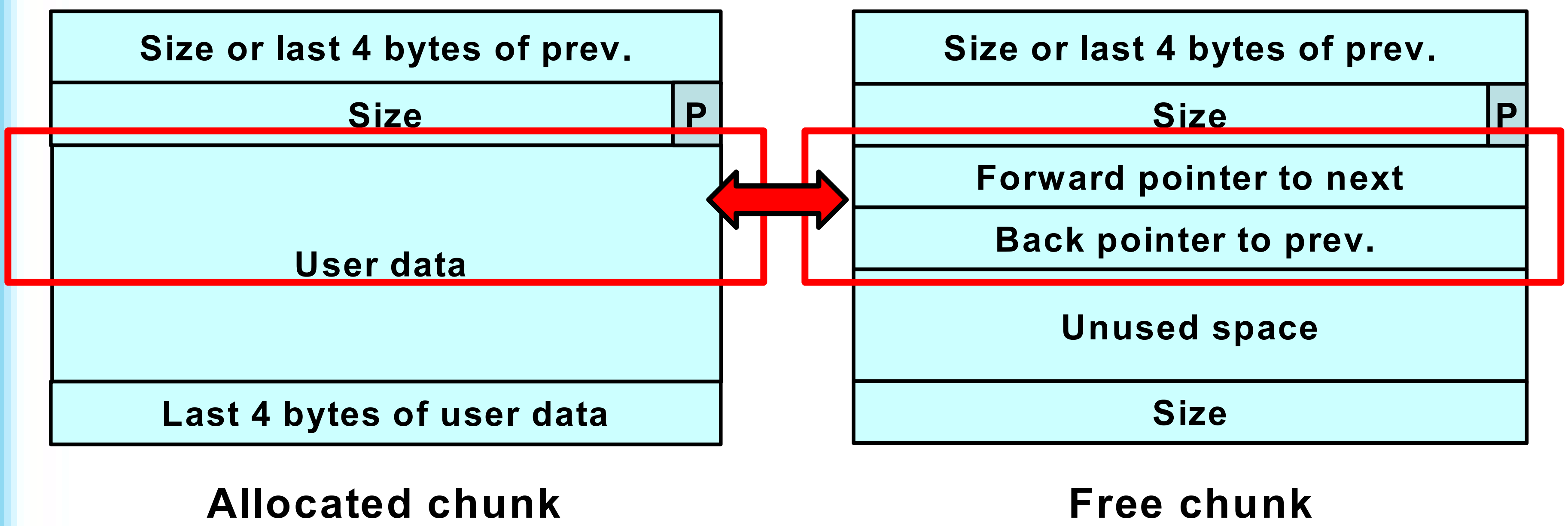
בלינוקס זה לא
היה קורה!

Dynamic Memory Management in C

- Memory allocation: malloc(size_t n)
  - Allocates n bytes and returns a pointer to the allocated memory; memory not cleared
  - Also calloc(), realloc()
- Memory deallocation: free(void * p)
  - Frees the memory space pointed to by p, which must have been returned by a previous call to malloc(), calloc(), or realloc()
  - If free(p) has already been called before, undefined behavior occurs
  - If p is NULL, no operation is performed

Doug Lea's Memory Allocator

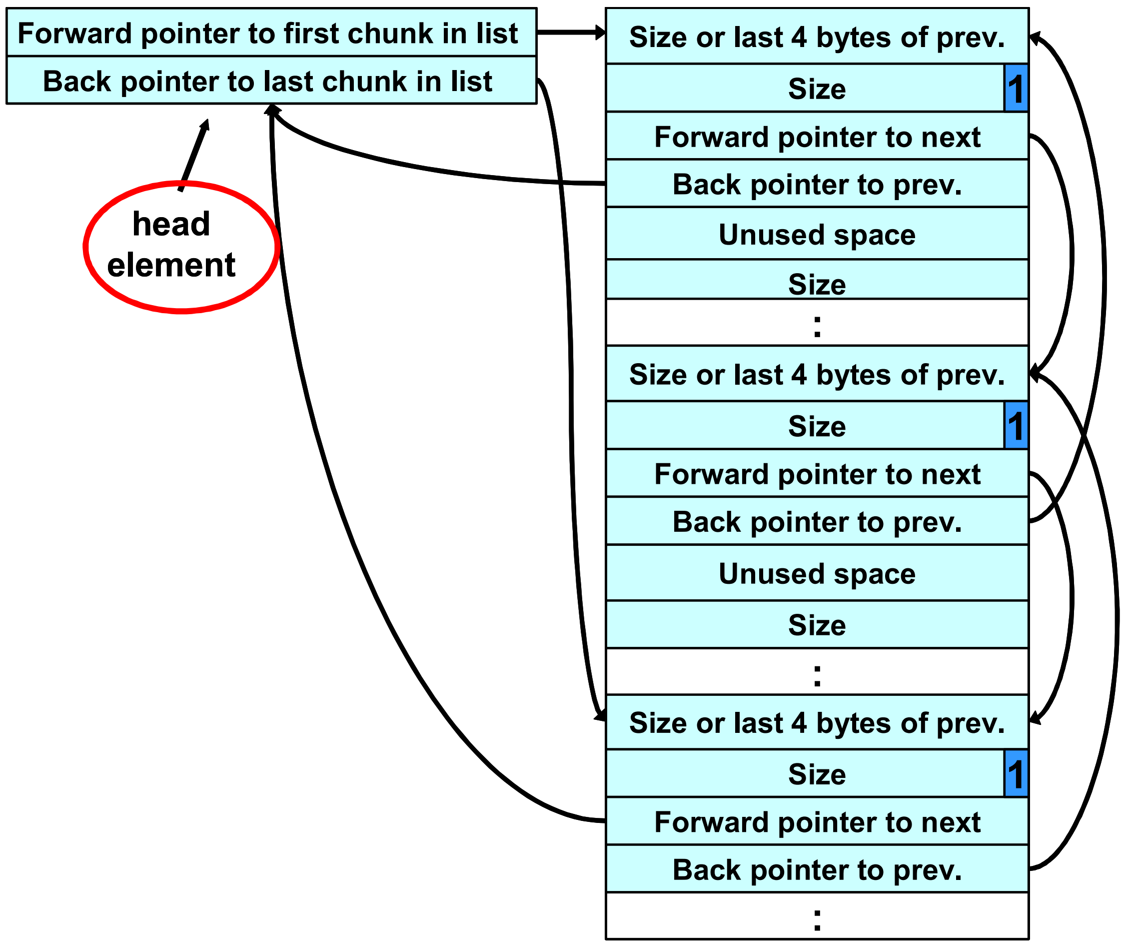
- The GNU C library and most versions of Linux are based on Doug Lea's malloc (dlmalloc) as the default native version of malloc



Free Chunks in dlmalloc

- Organized into circular double-linked lists (bins)
- Each chunk on a free list contains forward and back pointers to the next and previous chunks in the list
  - These pointers in a free chunk occupy the same eight bytes of memory as user data in an allocated chunk
- Chunk size is stored in the last four bytes of the free chunk
  - Enables adjacent free chunks to be consolidated to avoid fragmentation of memory

A List of Free Chunks in dlmalloc



Responding to Malloc

- Best-fit method
  - An area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n (requested allocation)
- First-fit method
  - Returns the first chunk encountered containing n or more bytes
- Prevention of fragmentation
  - Memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful

The Unlink Macro

```
#define unlink(P, BK, FD) {
   FD = P->fd;
   BK = P->bk;
   FD->bk = BK;
   BK->fd = FD;
}
```

Removes a chunk from a free list - when?

The Unlink Macro

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

Hmm… memory copy…

Address of destination read from the free chunk

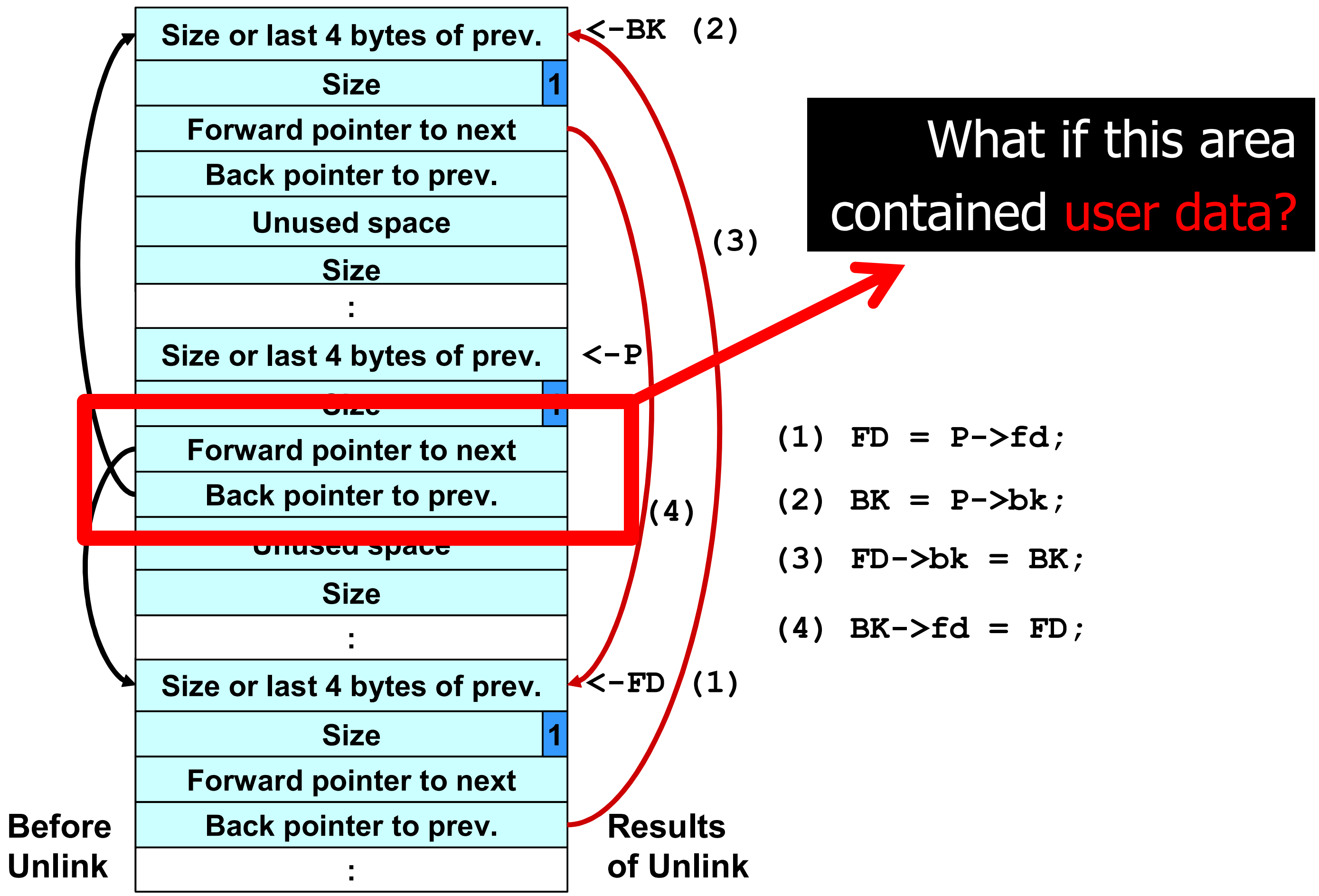
The value to write there also read from the free chunk

What if the allocator is confused and this chunk has actually been allocated…

… and user data written into it?

Rem t -when?

Example of Unlink

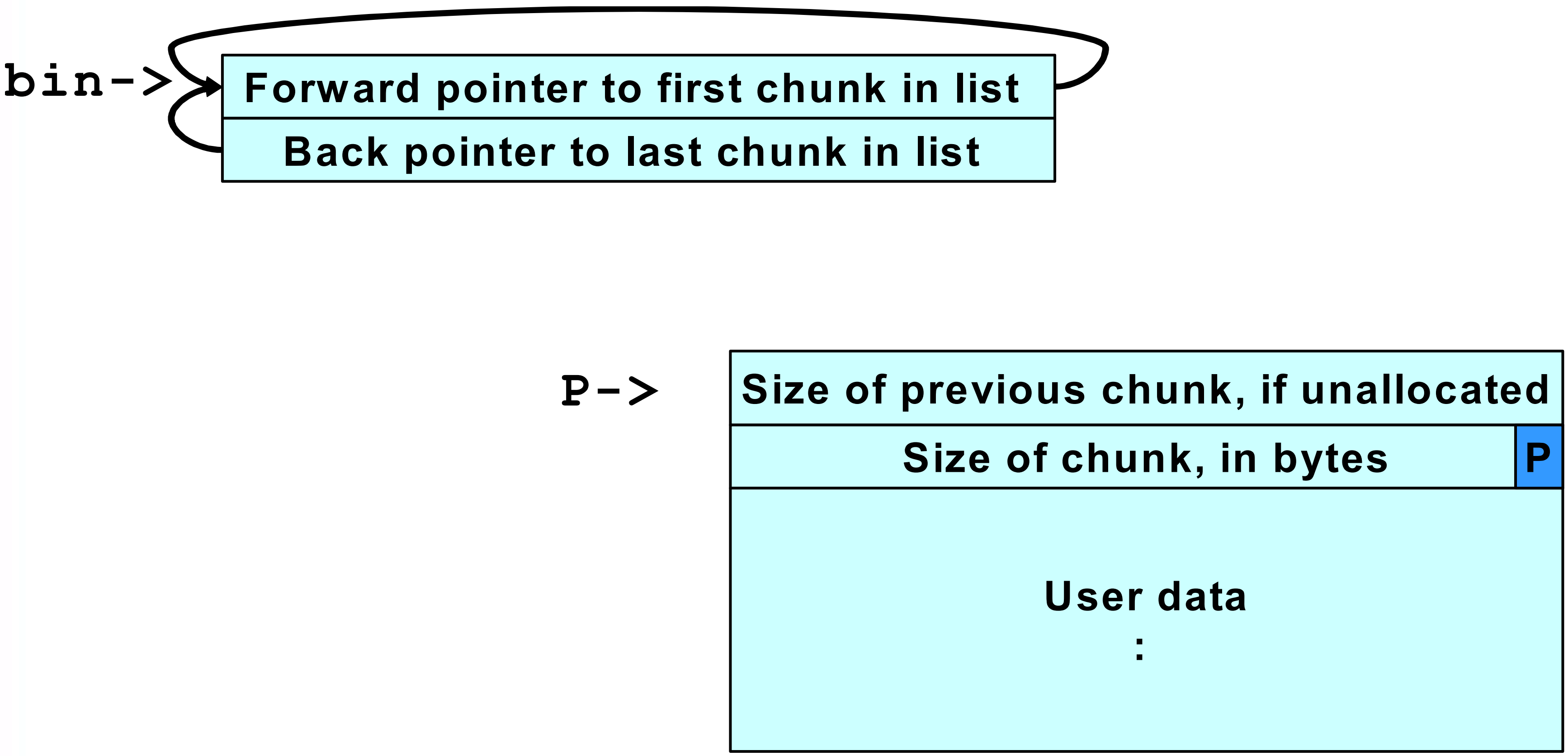



```
(1) FD = P->fd;
(2) BK = P->bk;
(3) FD->bk = BK;
(4) BK->fd = FD;
```

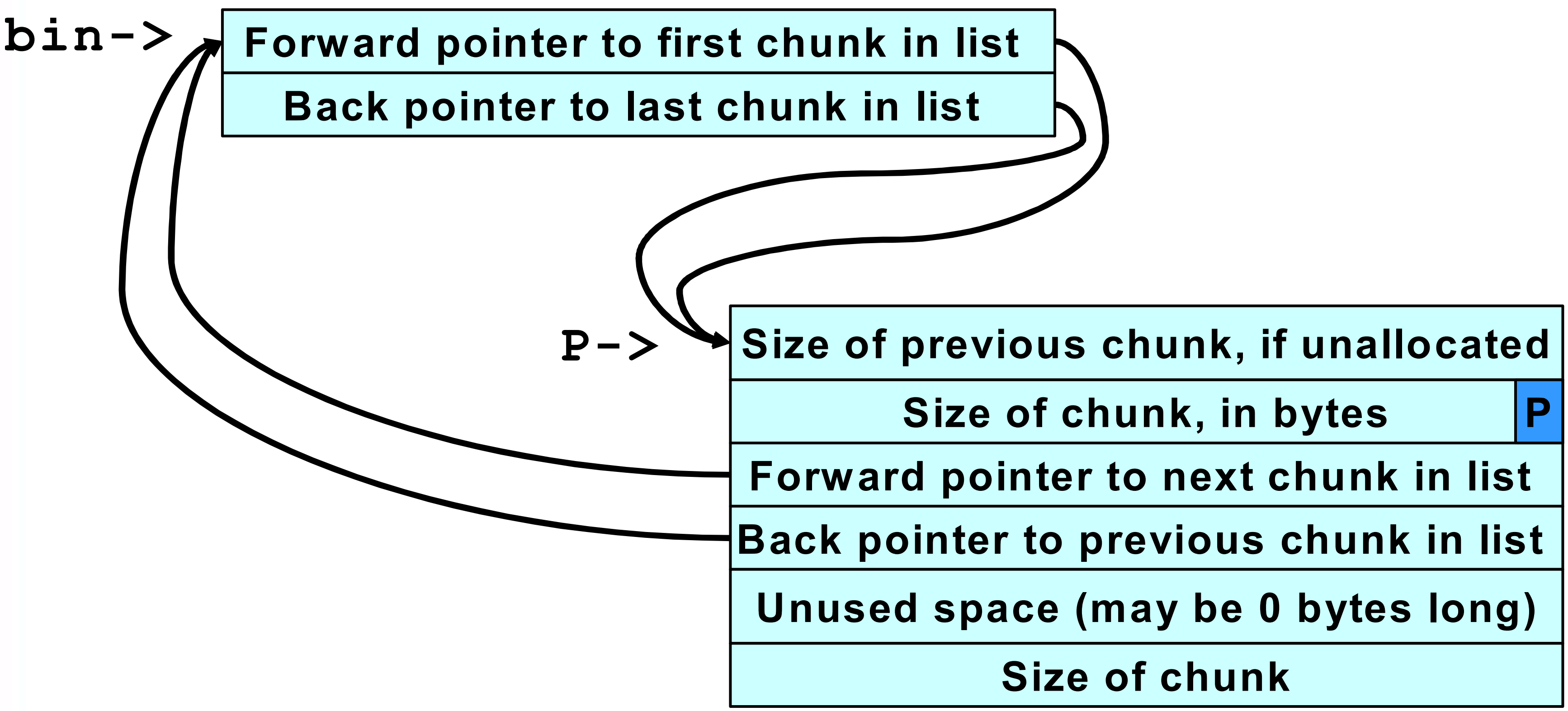
Double-Free Vulnerabilities

- Freeing the same chunk of memory twice, without it being reallocated in between
- The simple case:
  - The chunk to be freed is isolated in memory
  - The bin (double-linked list) into which the chunk will be placed is empty
- Other cases are possible…

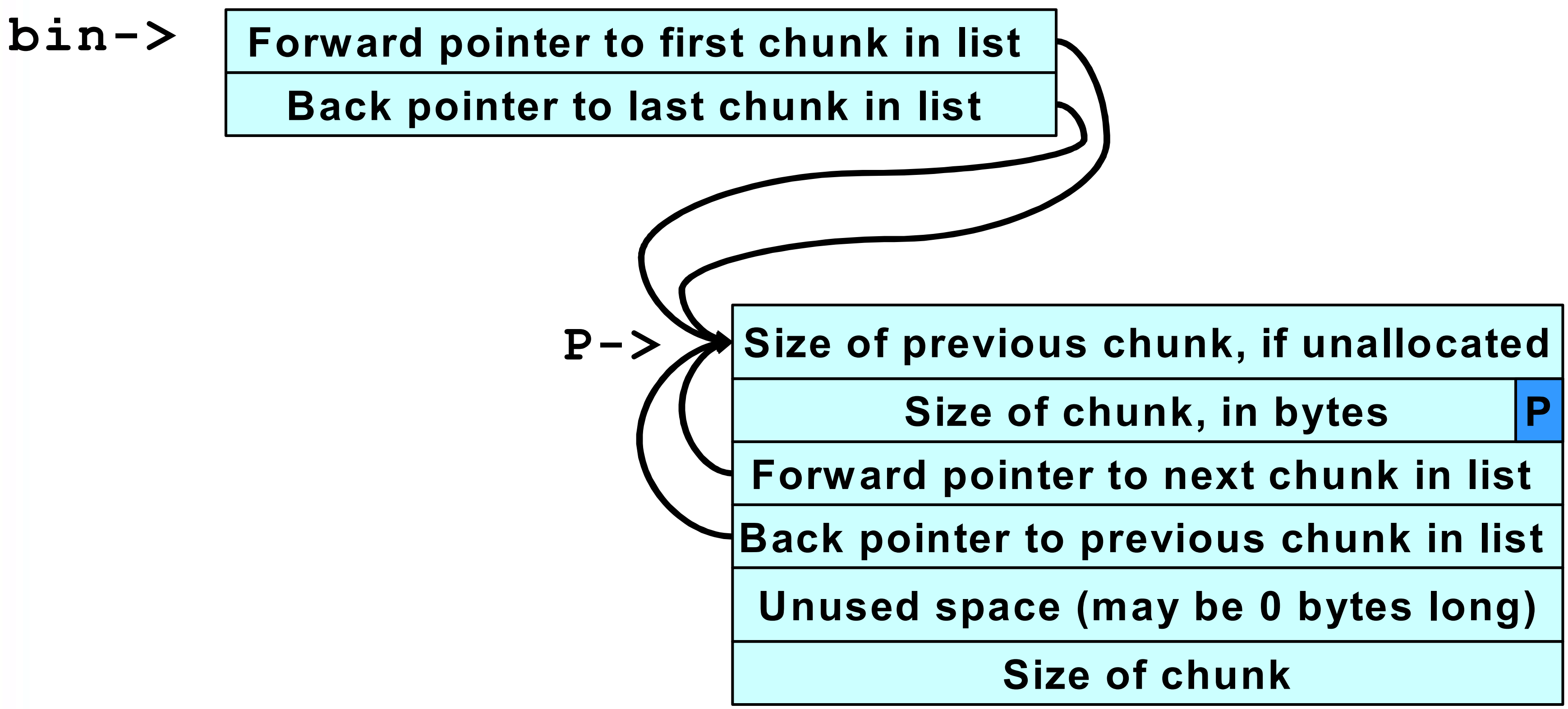
Empty Bin and Allocated Chunk



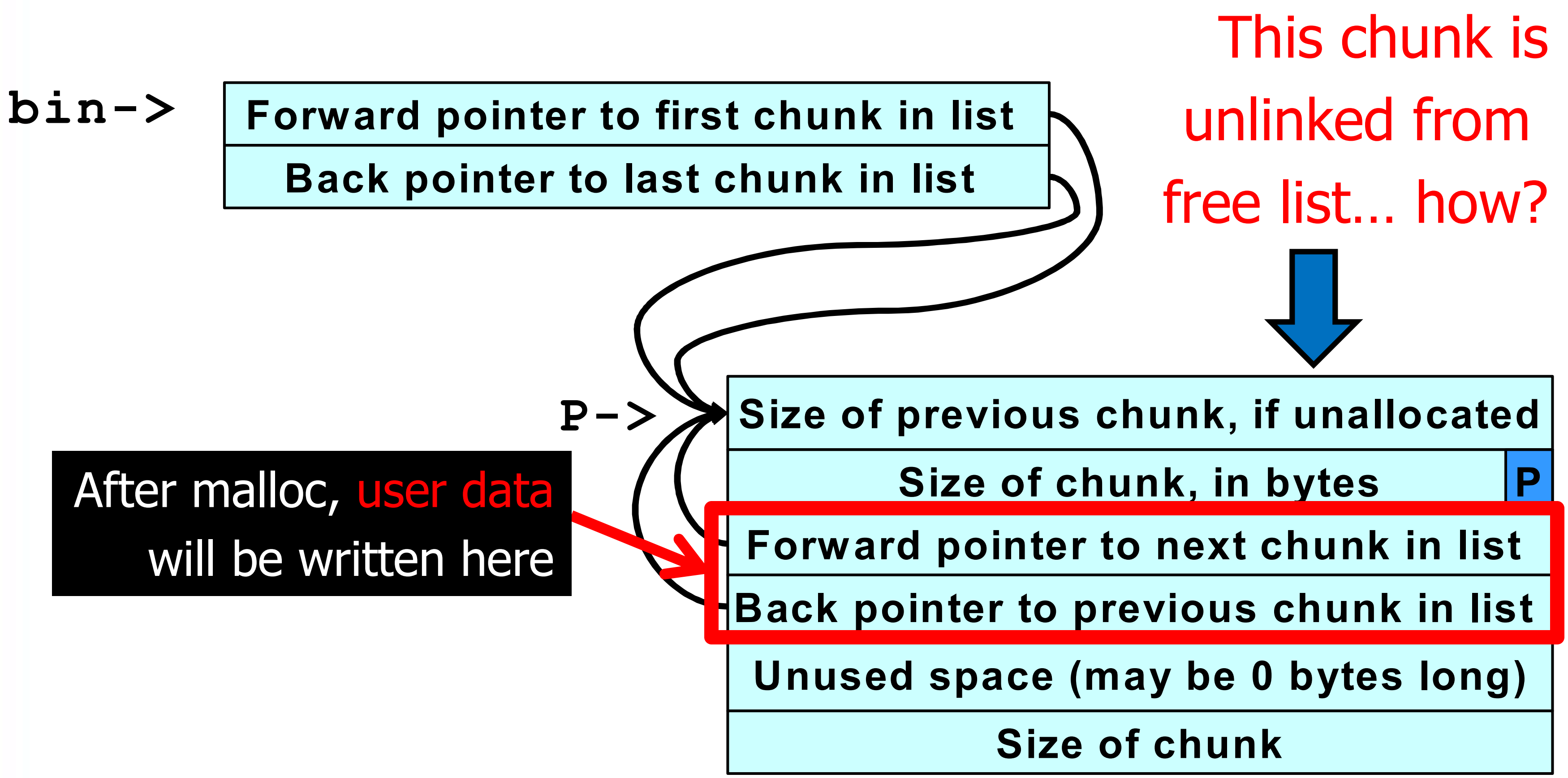
After First Call to free()



After Second Call to free()

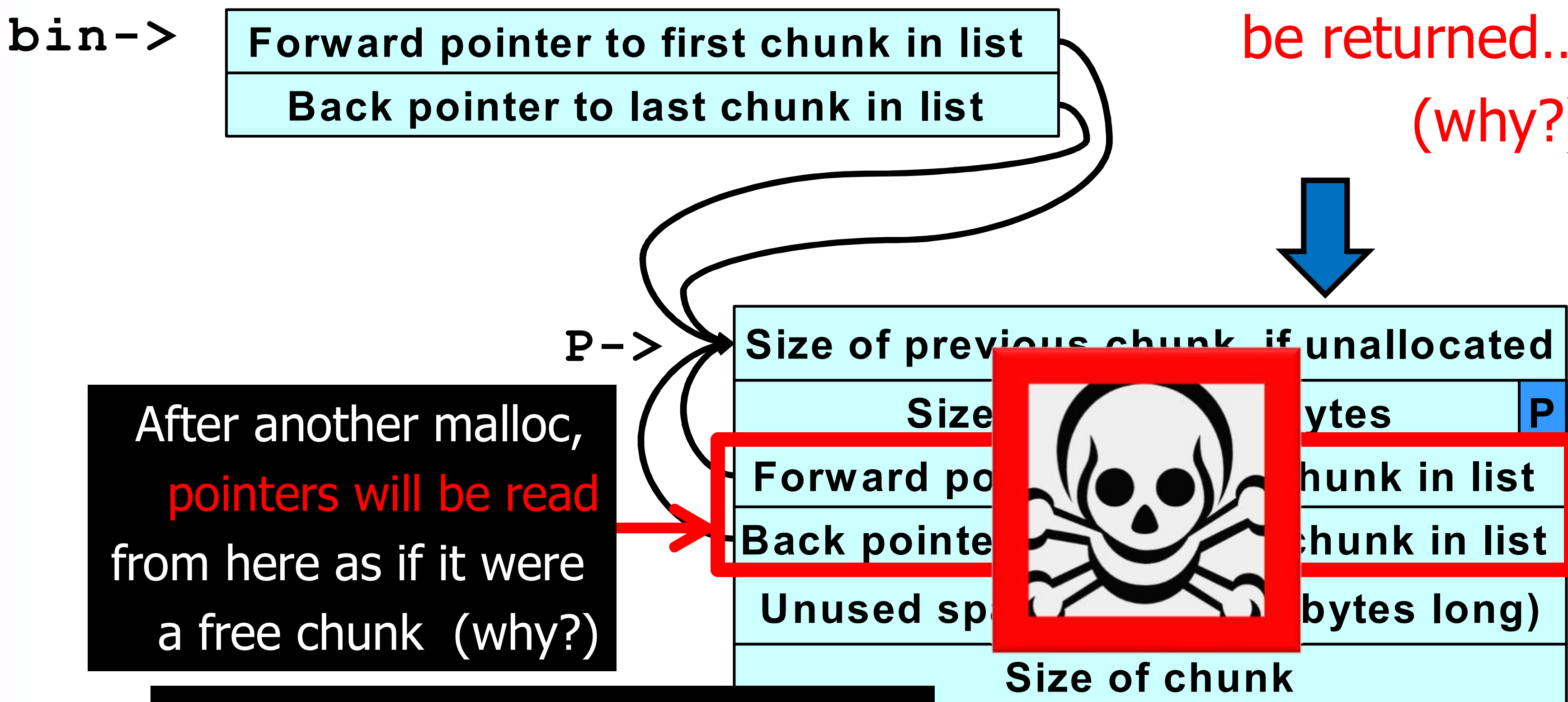


After malloc() Has Been Called



After Another malloc()

Same chunk will be returned… (why?)



After another malloc, pointers will be read from here as if it were a free chunk (why?)

One will be interpreted as address, the other as value (why?)

Mitigations in Windows XP SP2

- Cookie in chunk header
  - 8bit checksum(cookie) is introduced in chunk header.
  - By validating its value it can detect overwrite of a chunk header.
  - The value of a cookie is based on the address of the chunk.
- Safe unlinking
  - Before removing an element from doubly linked list it checks if following condition is met:

    [Chunk]->Flink->Blink == [Chunk]->Blink->Flink == [Chunk]

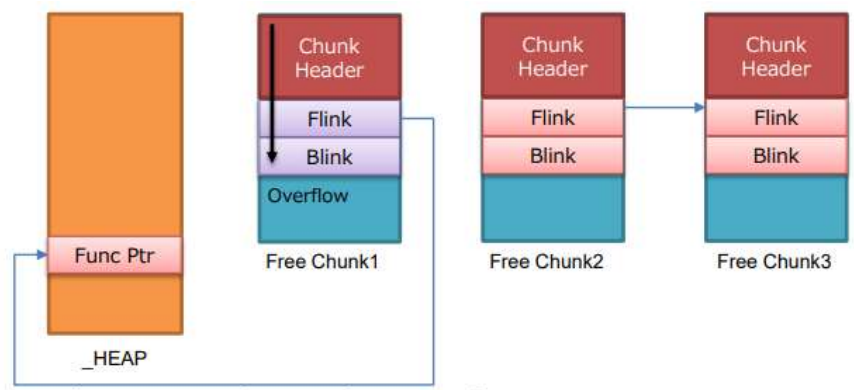
  (Confirming if next/prev elements also point to the element)
- PEB Randomization
  - Randomize the address of PEB(Process Environment Block).
  - PEB contains values and addresses used by attackers not only in heap exploit.
  - This randomization makes the success rate of attacks low.

Bypassing WinXP SP2 mitigations

- Using lookaside list
  - Freeing a chunk via HeapFree is handled by lookaside list first
  - Lookaside lists are singly linked lists of free chunks in each sizes.
  - If memory chunk is requested via HeapAlloc, it first checks if there is a free chunk in lookaside list and returns it if one exists.
- Important 2 mitigations does not work on lookaside list
  - Allocating a chunk from lookaside list does not make use of cookie
  - Safe Unlinking is not done (because it is not doubly linked list)

Bypassing WinXP SP2 mitigations

- Using lookaside list
  - Overflow to make Flink have an address of a region containing a function pointer
  - If data written in the region allocated from subsequent second HeapAlloc can be controlled, the pointer can be rewritten by an arbitrary address.
  - In the figure below, Flink uses a function pointer in _HEAP structure (This function pointer is used and called in heap management process)



Windows Vista

- Low Fragmentation Heap
  - Lookaside list is replaced with Low Fragmentation Heap
  - Impossible to attack using lookaside list
- Randomizing Block Metadata
  - Chunk header is xored with _HEAP->Encoding
  - Overwriting chunk header with predicted cookie still results in unexpected state of the chunk header.
- Enhanced entry header cookie
  - Cookie value also checks values in chunk header
  - Cookie had been calculated based on the chunk address but now it is calculated/validated with values in a chunk header.
- Heap base randomization
  - The base address of _HEAP structure is randomized – Makes it difficult to overwrite data in _HEAP structure
- Heap function pointer encoding
  - A function pointer in _HEAP structure is xored with a value
  - Mitigations for rewriting a function pointer in _HEAP structure

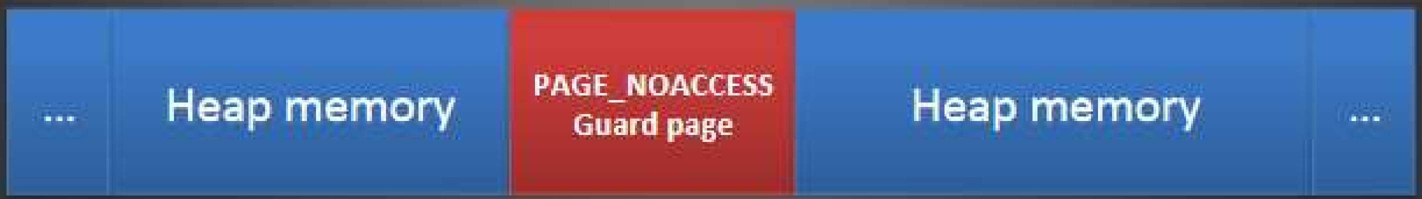
LFH design changes & integrity checks

Change in Windows 8	Impact
LFH is now a bitmap-based allocator	LinkOffset corruption no longer possible [8]
Multiple catch-all EH blocks removed	Exceptions are no longer swallowed
HEAP handle can no longer be freed	Prevents attacks that try to corrupt HEAP handle state [7]
HEAP CommitRoutine encoded with global key	Prevents attacks that enable reliable control of the CommitRoutine pointer [7]
Validation of extended block header	Prevents unintended free of in-use heap blocks [7]
Busy blocks cannot be allocated	Prevents various attacks that reallocate an in-use block [8,11]
Heap encoding is now enabled in kernel mode	Better protection of heap entry headers [19]

Outcome: attacking metadata used by the heap is now even more difficult

Guard pages

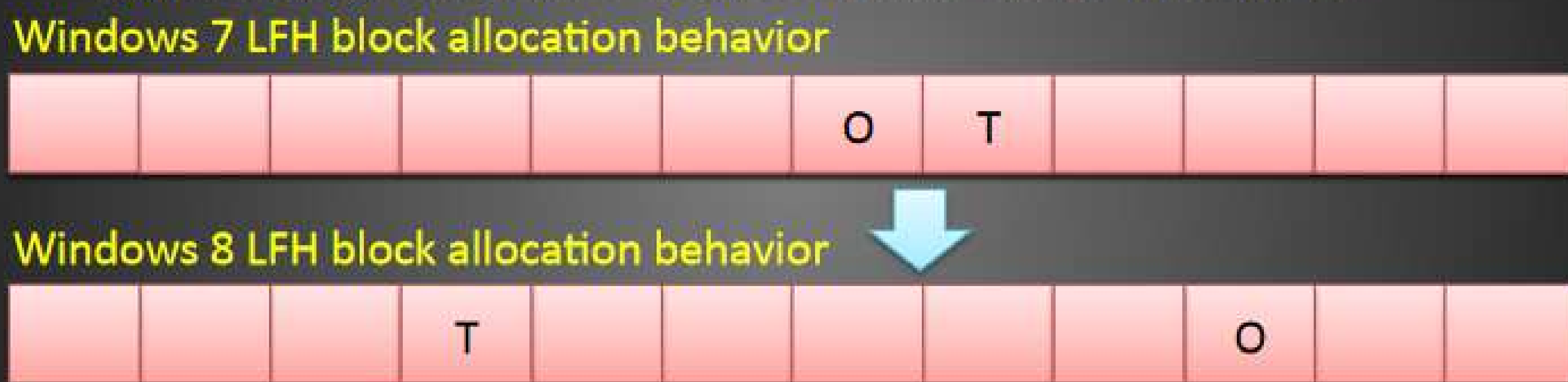
- Guard pages are now used to partition the heap
  - Designed to prevent & localize corruption in some cases
  - Touching a guard page results in an exception



- Insertion points for guard pages are constrained
  - Large allocations
  - Heap segments
  - Max-sized LFH subsegments (probabilistic on 32-bit)

Allocation order randomization

- Allocation order is now nondeterministic (LFH only)
  - Exploits often rely on surgical heap layout manipulation [10]
  - Randomization makes heap normalization unreliable



- Maximizing reliability is more challenging
  - Application-specific and vulnerability-specific
  - May require corrupting more data (increasing instability)
  - May require allocating more data (triggering guard pages)

לסיכום..

- ניצול חולשות Heap קשה ומסובך הרבה יותר מחולשות Stack
  - וגם תלוי מימוש וארכיטקטורה – דורש RE
- התקדמות משמעותית גם במנגנוני ההגנה ב-Heap
  - בעיקר רנדומיזציה ו-Guard Pages
- **אבל** ב-Heap קיימים מנגנונים מסובכים הרבה יותר
  - **שלפיכך פגיעים יותר לשגיאות**