2023-2024学年春季学期

计算机体系结构安全

Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

[第6次课] 计算机内存架构基础（二）

授课教师：陈李维

授课时间：2024.4.1

[第6次课] 计算机内存架构基础

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

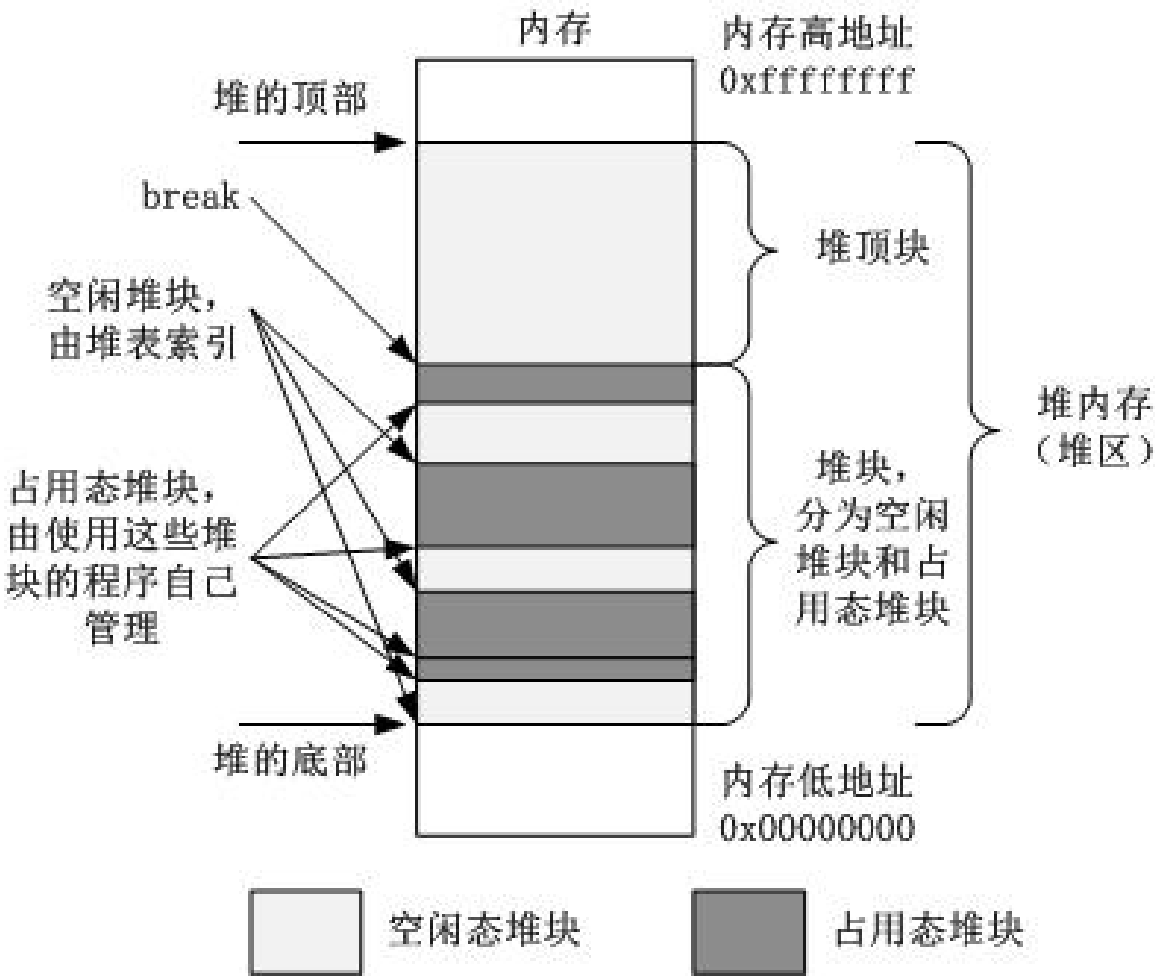
前一次课的内容

- 详细介绍了堆的相关知识，以及堆的正常管理过程（即堆块的分配和释放过程，空闲堆块合并过程等）。
- 本节课将详细介绍堆漏洞的原理和具体实现过程。
  - 堆溢出漏洞（heap overflow）
  - 重复释放漏洞（double free）
  - 释放后使用漏洞（use after free）

堆的基本知识回顾

堆的几个基本概念

- 堆内存，也就是堆区，即整个堆内存空间。
- 堆块，堆内存管理的最小单元，分为空闲态和占用态。
- 堆顶块，处于堆顶的一个特殊空闲堆块，由break指针索引。
- 堆表，索引所有空闲堆块的链表（除了堆顶块）。

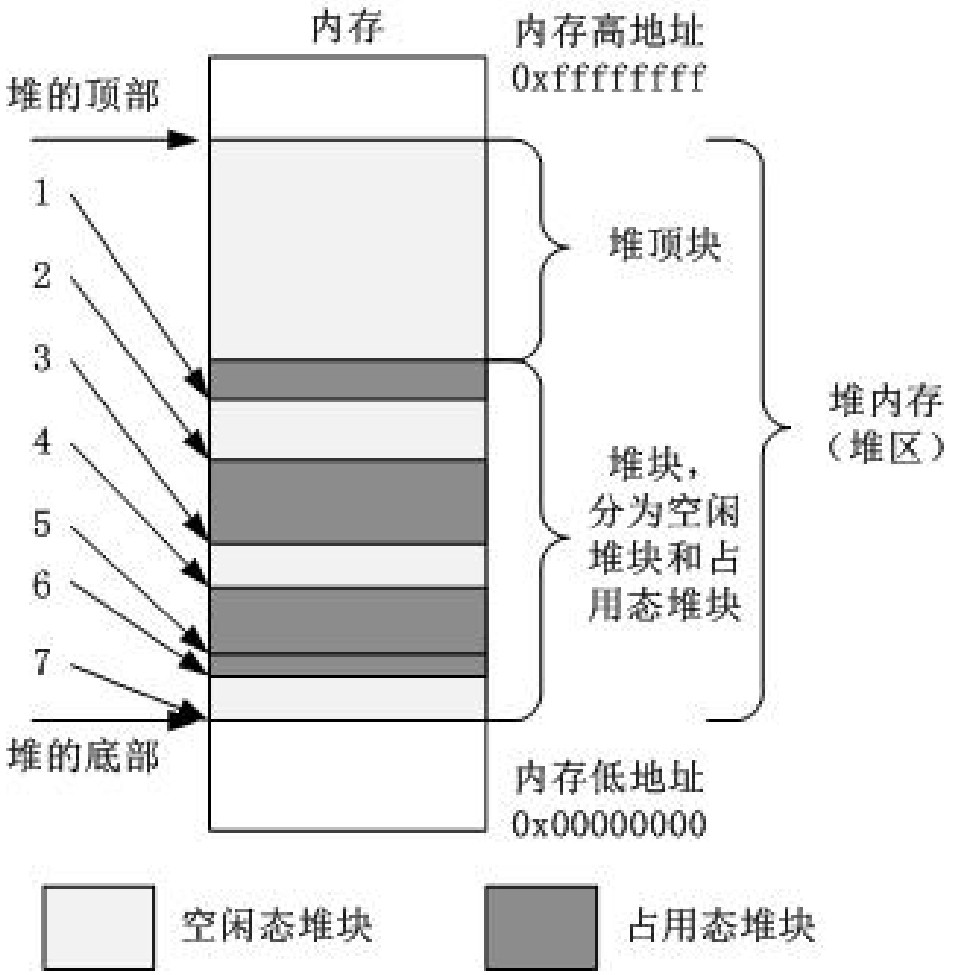


堆的基本知识回顾

相邻空闲堆块合并

- 如果被释放的堆块与空闲堆块相邻，则触发**空闲堆块合并**。
- 空闲堆块合并需要通过**拆链（unlink）**过程将空闲堆块从堆表中取下

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



堆漏洞

- 常见的堆漏洞类型：
  - 堆溢出漏洞（Heap overflow）
  - 重复释放漏洞（Double free）
  - 释放后使用漏洞（Use after free，UAF）

堆溢出漏洞

堆溢出漏洞的基本思路

- 堆块中包含一些特殊的数据，如prev_size、size、fd、bk等。如果能够控制堆块的特殊数据，则能对程序运行造成一定的影响。
- 溢出攻击的思路都是要覆盖一些特殊的数据，通过控制这些特殊的数据来控制程序的行为。
  - 栈溢出通过修改返回地址，来改变函数返回地址。
  - 堆溢出通过修改prev_size、size、fd、bk等特殊数据，来实现任意地址写操作。

堆溢出漏洞

- 堆溢出漏洞，就是指向固定长度的堆块可用空间写入超出预先分配长度的内容，造成数据溢出，从而覆盖了相邻堆块的内存空间，尤其是相邻堆块的元数据等。
- 特点
  - 缓冲区在**堆**中分配
  - 拷贝的数据过长，**超过**预先分配的长度
  - 覆盖了堆块中的prev_size、size、fd、bk等**关键数据**

堆溢出漏洞

堆溢出漏洞示例

- 如图所示，使用malloc为buf1分配了128B的内存。
- 然后向buf1填充了200B的数据，明显产生了数据溢出。

```
int main(int argc, char *argv[])
{
  char *buf1 = malloc(128);
  char *buf2 = malloc(256);

  read(fileno(stdin), buf1, 200);

  free(buf2);
  free(buf1);
}
```

堆溢出漏洞

堆溢出漏洞利用原理

- 首先，利用堆溢出漏洞，篡改相邻堆块的部分数据，欺骗堆管理器，使其认为相邻堆块为**空闲态**。
- 当当前堆块被释放时，由于相邻堆块已经被篡改为空闲堆块，所以**引发空闲堆块合并操作**。
- 然后，利用空闲堆块合并时的**拆链函数（unlink）**，可以实现一个**任意地址写操作**，能够将一个内存任意地址的数据修改为任意的数值。

困难的不是漏洞本身，而是对漏洞的利用！

堆溢出漏洞

任意地址写

- **任意地址写**，即任意写任意地址，可以将一个内存中任意地址的数据改为任意数值。

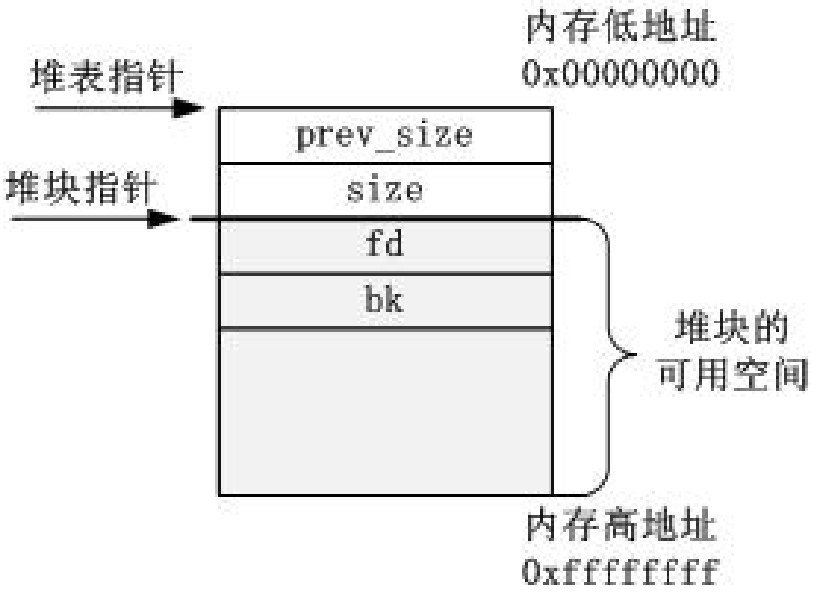
- *A=B;
- 如果能够控制**地址A和数据B**，就认为实现了一个任意地址写操作。

堆溢出漏洞

任意地址写的实现

- 通过堆溢出等手段伪造一个空闲堆块P，所以堆块P中的数据，如fd和bk等就可以被设置成任意的数值，然后就能利用unlink实现任意地址写操作。
- 见“堆块合并的任意地址写.pdf”

```
#define unlink(P) {
    FD = P->fd;  //P->fd是伪造的，FD被控制
    BK = P->bk;  //P->bk是伪造的，BK被控制
    …
    FD->bk = BK;  //任意地址写操作。FD->bk就是FD+12，即向地址FD+12写入BK
    BK->fd = FD;
    … }
```



堆溢出漏洞

- 任意地址写操作是一种很强的能力，可以用于很多种不同的攻击行为：
  - 修改系统关键数据
    - 例如，直接修改系统配置参数。
  - 修改系统关键函数地址
    - 例如，将free函数地址改为恶意shellcode地址，当系统执行free时，就会跳转到恶意shellcode。

堆溢出漏洞

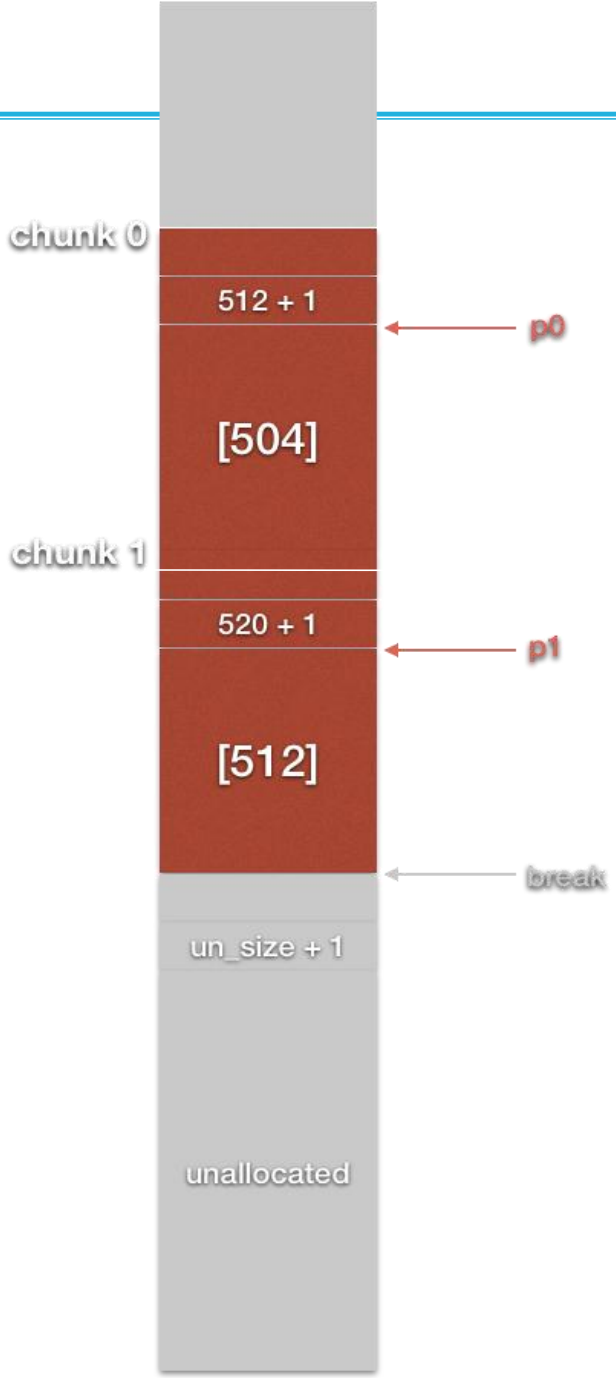
修改系统关键函数地址

- 在可执行文件的动态连接机制中，每一个外部定义的符号在**全局偏移表** (Global Offset Table，**GOT**)中都有相应的条目，用来保存符号所在地址相对于GOT首地址的偏移，以方便对符号的引用。
  - 例如，free是系统库函数，程序每次调用free时，需要根据其在GOT中的条目找到其在系统库中的地址以完成调用。
- 简单的说，就是free的地址保存在GOT表中。如果能够修改GOT表，就能改变free的地址。如果将free地址改成shellcode的地址，那么下一次调用free时，就变成了执行shellcode。

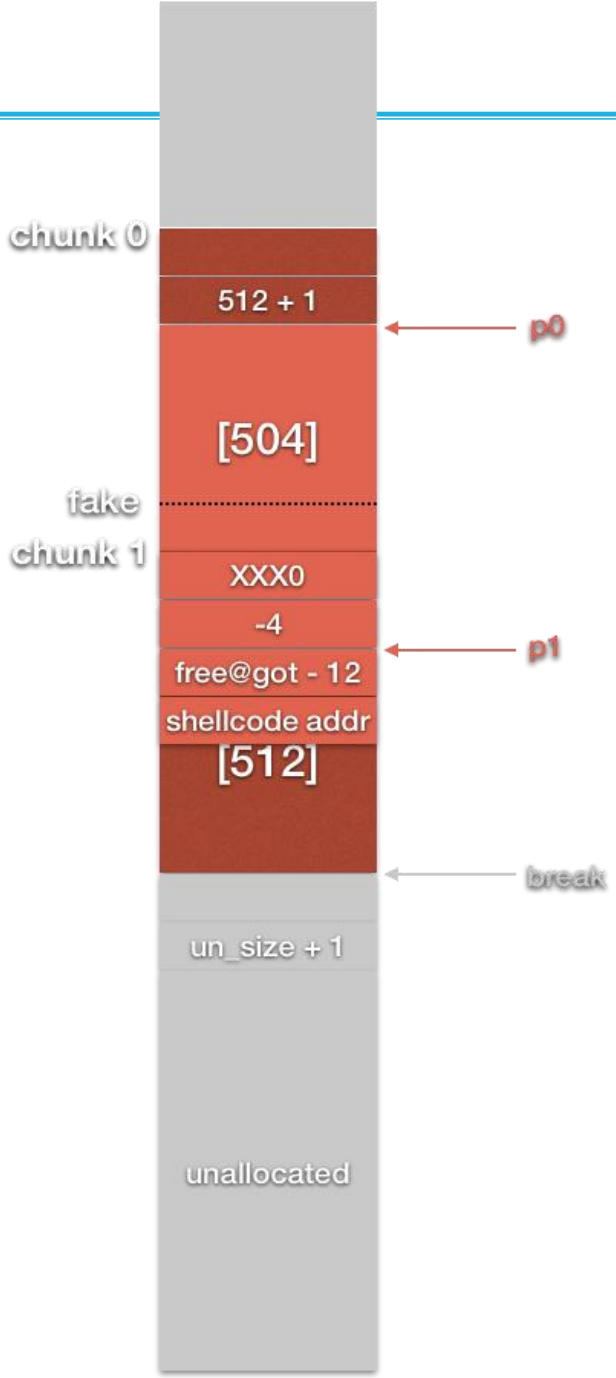
堆溢出漏洞

```
int main( int argc, char * argv[] )
{
        char *p0, *p1;

    p0 = malloc( 504 );
    p1 = malloc( 512 );
       if(argc!=1)
           strcpy( p0, argv[1] );
    free( p0 );
    free( p1 );
    return( 0 );
}
```

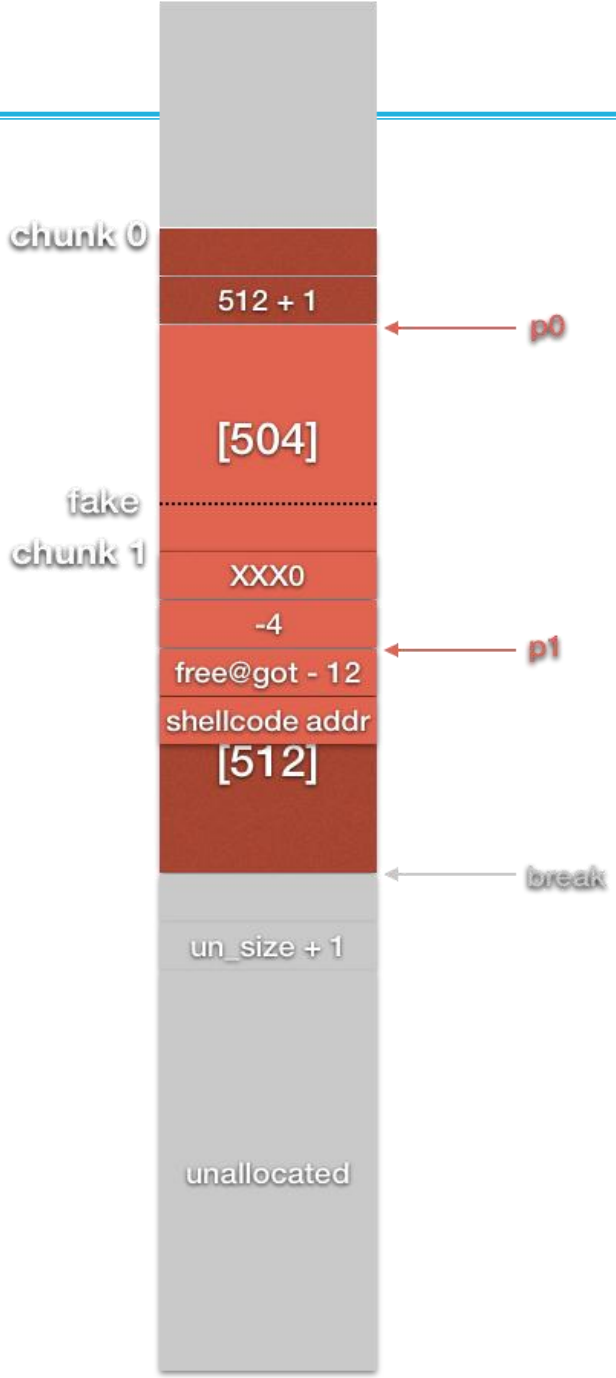
- 在strcpy处有很明显的堆溢出漏洞。只要argv[1]中的内容足够长，就会越界覆盖到p1部分。



- 使用malloc分配两个堆块。
- p0和p1指向堆块的数据区。
- 需要注意的是，可供使用的内存空间长度并不等于堆块实际长度。
- 堆块实际长度=可供使用的内存空间长度 + 8（32位系统）。



- 对chunk0进行溢出，填入shellcode，同时覆盖chunk1的头部。
- 将chunk1的size修改为-4的补码，这样堆管理器会认为chunk1的下一个堆块fake位于chunk1的前4个字节（即p1 - 4）。
- 此时，堆块fake的size字段（即chunk1的prev_size）被覆盖为一个偶数，使堆管理器以为chunk1为空闲态。
- 简单来说，通过堆溢出伪造数据，使得堆管理器认为，堆块0 -> 堆块1 -> 堆块fake，并且堆块1是空闲堆块。
- 然后，free(p0)时触发空闲堆块合并。



- 堆管理器误以为chunk1为空闲，触发unlink(chunk1)过程。

```
#define unlink(p1) {
    FD = p1->fd;    //FD=free@got-12
    BK = p1->bk;   //BK=shellcode
    FD->bk = BK;  //free@got=shellcode
    BK->fd = FD;  //shellcode+8=free@got-12
}
```

- 空闲堆块合并完成后，got中指向free函数的表项被修改为指向shellcode。
- 此后，调用free函数变成执行shellcode。
- 副作用：shellcode+8位置的4字节数据会被替换为free@got–12，所以我们编写的shellcode应该跳过前面的12字节。

重复释放漏洞

- 重复释放漏洞（double free），是指程序在进行一次free之后没有及时把被free的指针p置空，从而可以再次free指针p而产生的漏洞。

重复释放漏洞

重复释放漏洞示例

- 申请了一个堆块p，重复释放p。

```
char *p = malloc(100);
…
free(p);
…
free(p);     // double free
```

重复释放漏洞

重复释放攻击原理

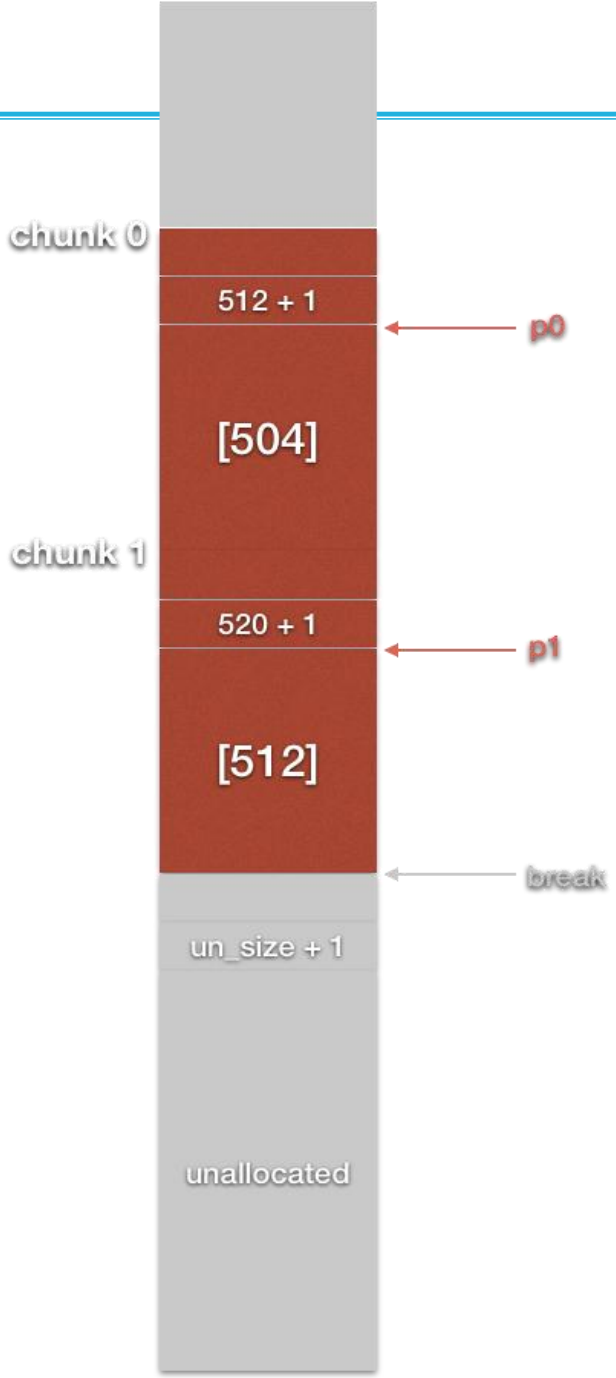
- 首先，利用double free漏洞，伪造堆块数据，欺骗堆管理器，使其认为p仍然指向一个可用的堆块。
- 然后，和堆溢出攻击类似，当再次释放p时，利用空闲堆块合并时的拆链函数（unlink）实现一个**任意地址写操作**。

重复释放漏洞

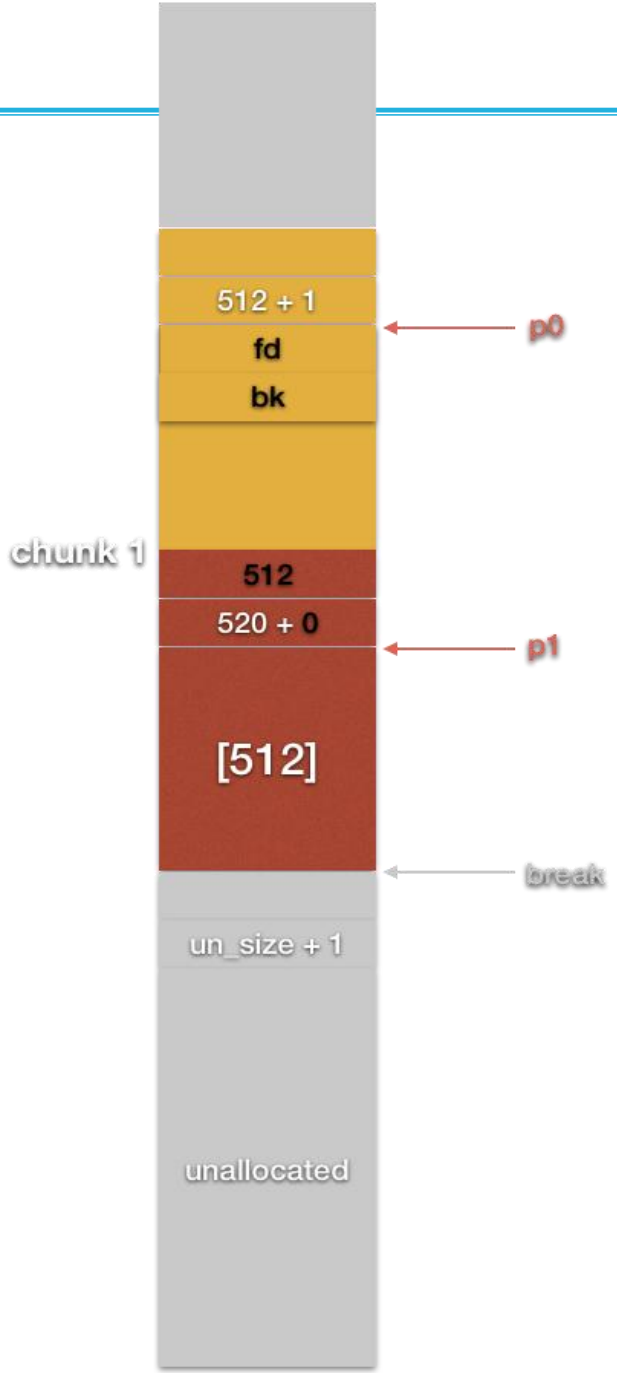
重复释放攻击示例

- 堆块p1被重复释放。

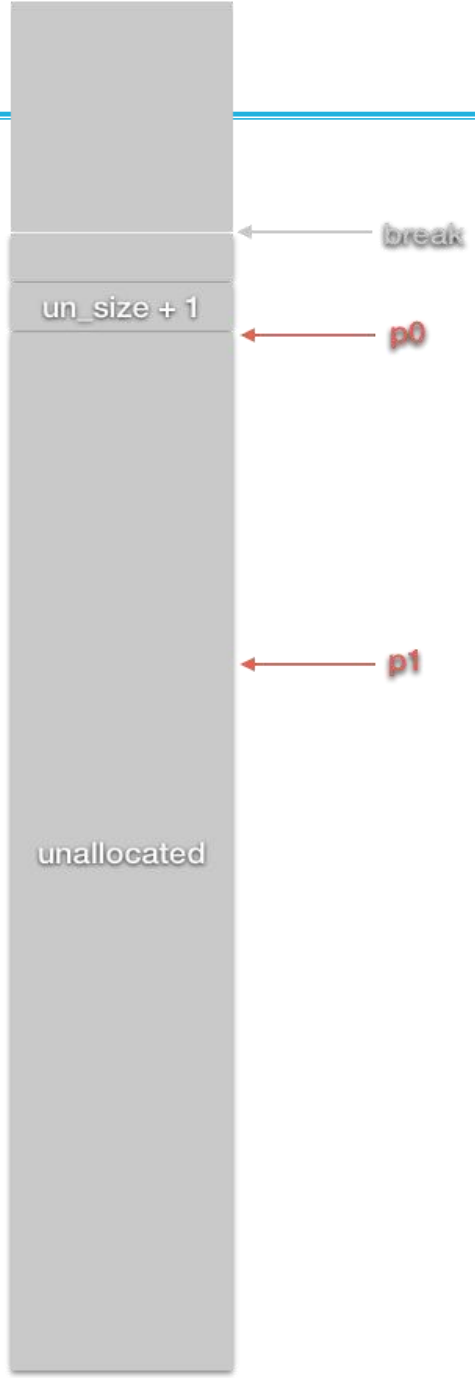
```
char *p0 = malloc(504);
char *p1 = malloc(512);
free(p0);
free(p1);
char *p2 = malloc(768);  // 申请一个大于堆块p0的新的堆块
memcpy(p2, malicious_data, length); //向堆块p2输入设计好的恶意数据，伪造堆块信息
free(p1);                           // double free
```



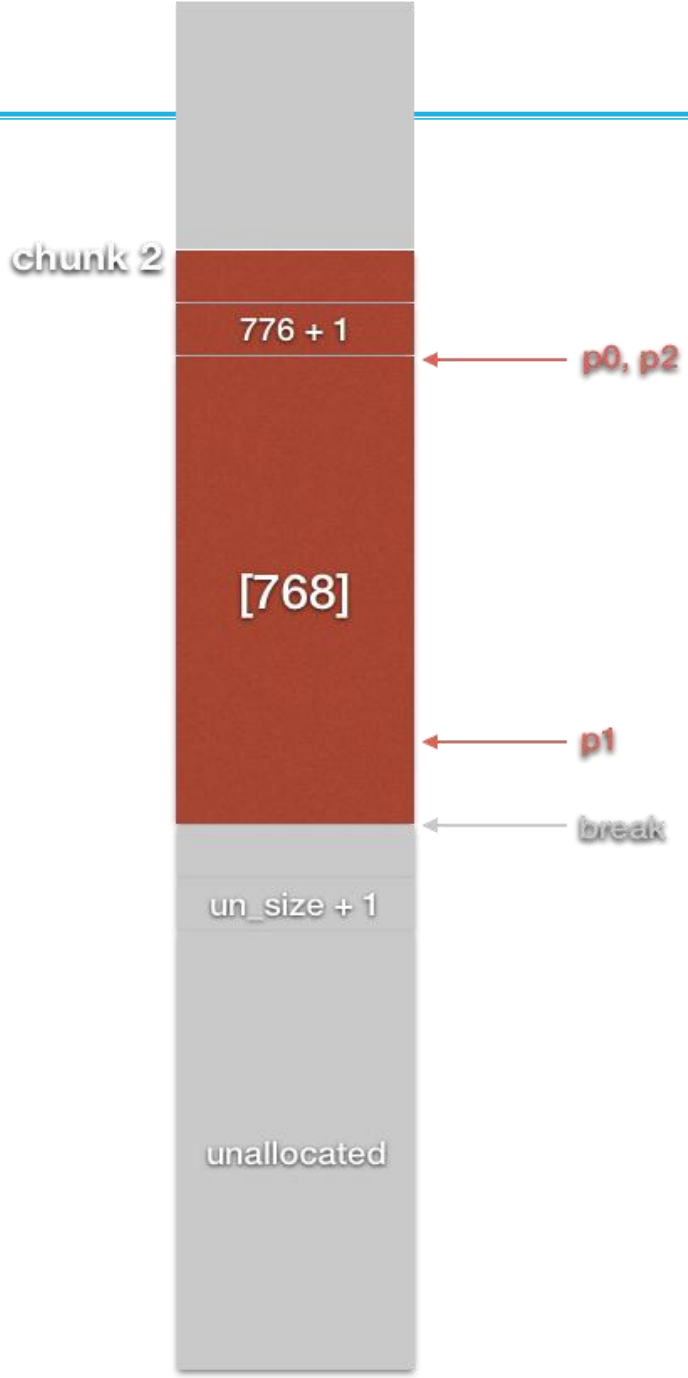
- 使用malloc分配两个堆块。
- 获得两个可控的指针p0和p1。



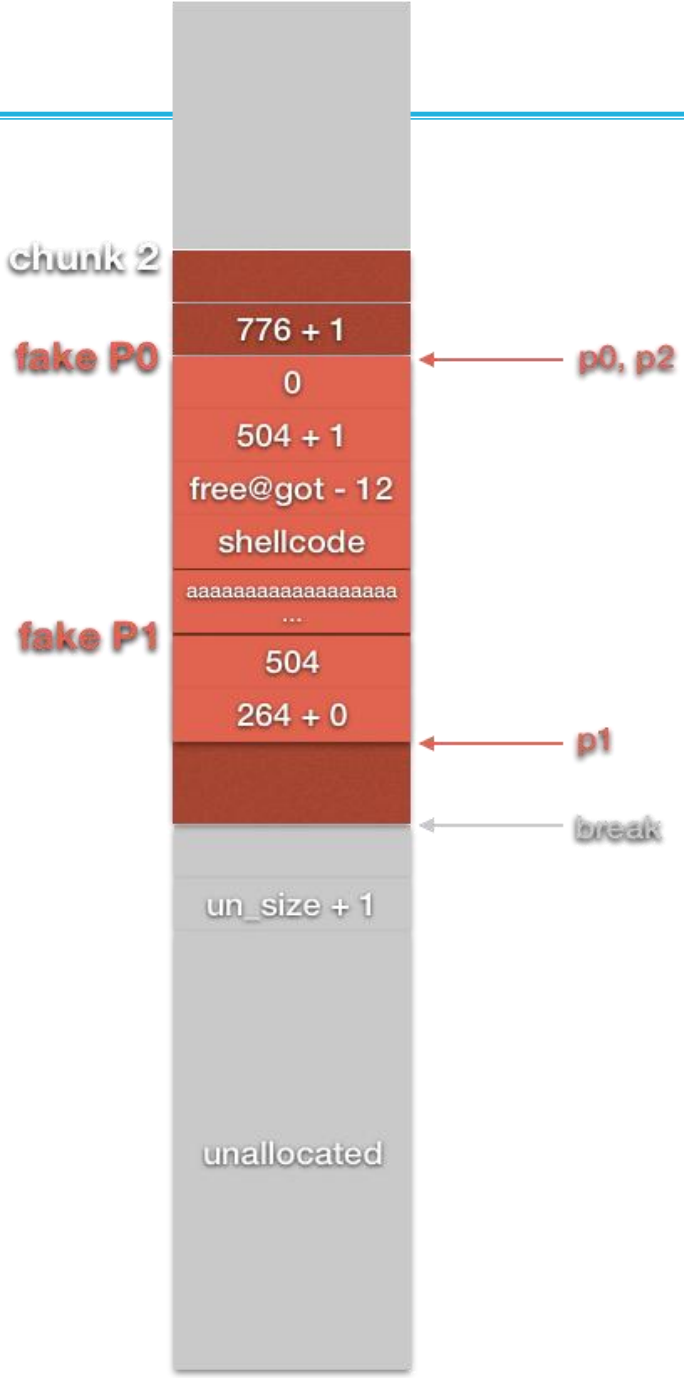
- free(p0)后，chunk0 被释放并链入堆表unsorted bins中，填充fd和bk的值。
- chunk1中的prev_size值和空闲标记位也被正常修改。



- free(p1)后，chunk0和chunk1发生空闲堆块合并后被链入堆表unsorted bins。
- 由于chunk1紧邻堆顶块，合并后的堆块被取出堆区，和堆顶块合并。
- 由于程序未及时将被释放的指针置0，出现了两个悬空指针p0和p1。



- malloc一个chunk2，其大小应该超过chunk0+8，保证能够包含原chunk1的头部。
- 悬空指针p1指向chunk2的数据区域。



- 在chunk2内部伪造堆块FP0和FP1。
- 原来的p1指向伪造堆块FP1。
- 伪造的FP0和FP1相邻。伪造的FP0是空闲堆块，伪造的FP1是占用态。
- 再次free(p1)，堆管理器会对伪造的堆块FP0和FP1进行空闲堆块合并。
- 类似堆溢出攻击中的unlink过程，即unlink(FP0)，实现一个任意地址写操作，最终将对free函数的调用篡改为调用shellcode。

释放后使用漏洞

- 释放后使用漏洞（use after free，UAF），是指程序在进行一次free之后没有及时把被free的指针p置空，从而可以利用该指针p访问堆块内部函数而产生的漏洞。

释放后使用漏洞

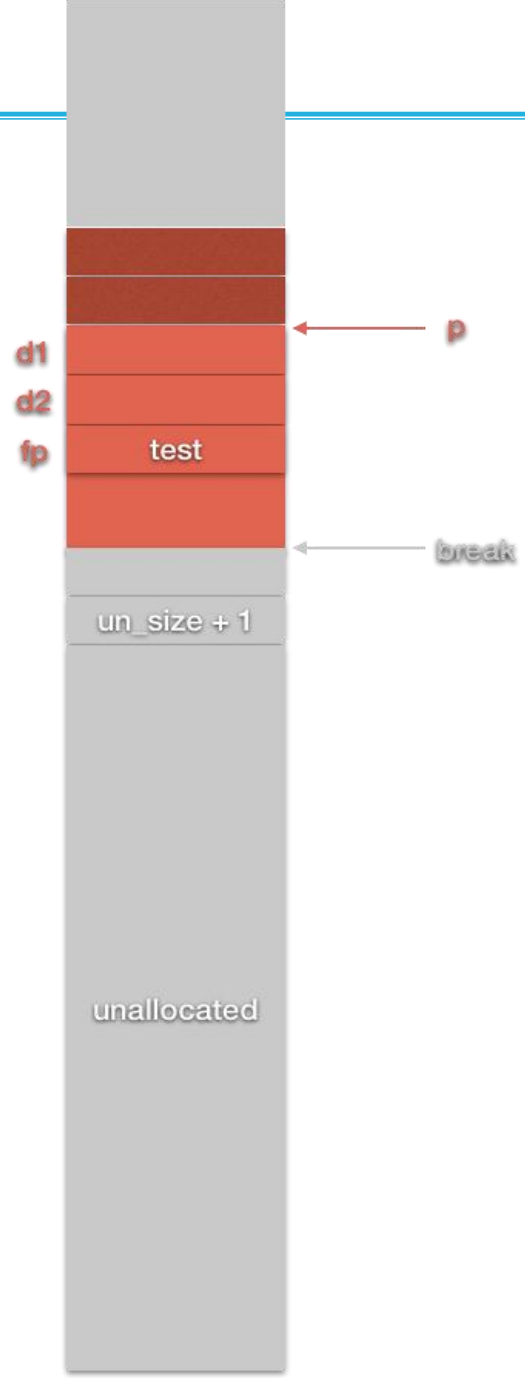
释放后使用漏洞原理

- 首先，利用释放后使用漏洞，修改悬空指针对应的内存数据，将其修改为攻击所需要的数据（如shellcode的地址）。
- 然后，利用悬空指针，跳转并执行恶意shellcode。

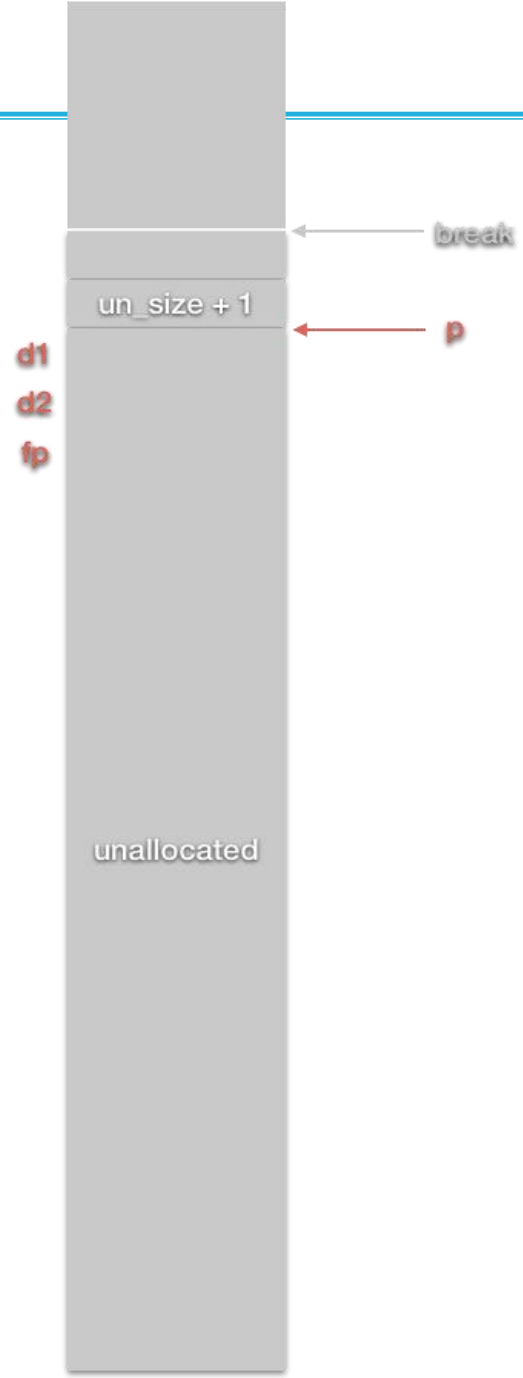
释放后使用漏洞

- 在str起始位置填充8字节长度的数据后，放置shellcode的地址（也可以直接放置shellcode的代码）。
- strcpy之后，p->fp指向了shellcode，执行p->fp()也就变成了运行shellcode。

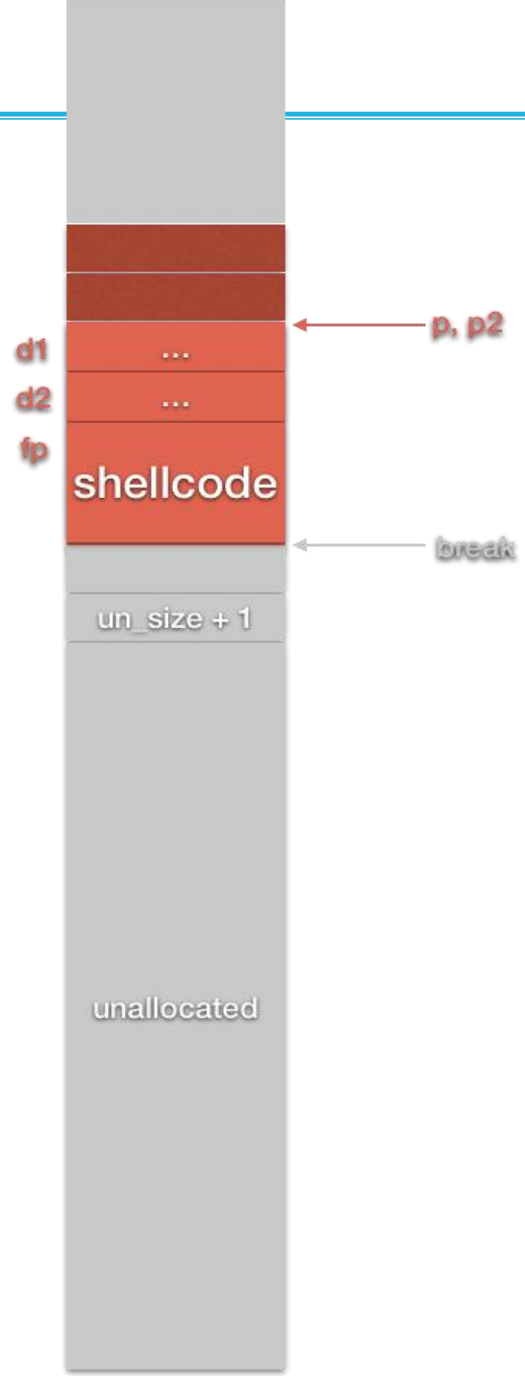
```
struct vul
{
    int d1, d2;
    void (*fp)();
};

void test()
{
    cout<<"hello!"<<endl;
}

void useAfterFree(char * str)
{
    vul * p = (vul *)malloc(sizeof(vul));
    p->fp = test;
    free(p);
    char *p2 = (char *)malloc(100);
    strcpy(p2, str);
    p->fp();
}
```



- malloc一个vul结构的堆块p。
- 让堆块中的指针p->fp指向test函数。



- free(p)，堆块p被合并到堆顶块。
- 但是，悬空指针p被保留，所以p->fp依然可以被使用。



- malloc一个大小合适的新堆块p2。
- 在堆块p2内填充包含shellcode的数据，并让p->fp刚好指向shellcode（也可以将p->fp覆盖为shellcode的地址）。
- 于是，调用p->fp()就变成了执行shellcode。

小结

- 详细介绍了堆漏洞的原理和具体实现过程。
  - 堆溢出漏洞（heap overflow）
  - 重复释放漏洞（double free）
  - 释放后使用漏洞（use after free）
- 关键在于对堆块管理过程以及堆块结构的深入理解，尤其是空闲堆块合并过程，以此利用相应漏洞实施攻击

小结

- 栈漏洞和堆漏洞的异同：
  - 对栈漏洞的利用主要就是**栈溢出攻击**：
    - 利用栈溢出漏洞，覆盖栈中的关键数据，如返回地址等，最后执行恶意代码。
  - 对堆漏洞的利用主要是以下两种方式：
    - 通过修改或伪造堆块中关键数据，误导堆管理器，然后**利用unlink函数实现任意地址写操作**。
    - 利用堆块释放后未及时置空的**悬空指针**，访问非法数据，执行恶意代码。

小结

- 思考：
  - 漏洞和攻击的原理都很简单，难点在细节上，如何利用具体细节实现有意义的行动。
  - 如何对漏洞进行修补，如何对攻击进行防御?

- 新版本的Linux 针对 Unlink 函数容易被攻击的弱点，新增了对前后堆块 fd 和 bk 的判断，提升了攻击难度，同学们可以尝试搜索并学习对应的绕过方法。

小结

- [1] Justin N. Ferguson, Understanding the heap by breaking it. Blackhat USA, 2007.
- [2] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. The Shellcoder's Handbook: Discovering and Exploiting Security Holes. Wiley, 2003.
- [3] Doug Lea, Design of dlmalloc: A Memory Allocator. Personal website.
- [4] Andries Brouwer, Hackers Hut: Exploiting the heap.
- [5] Phantasmal Phantasmagoria, The Malloc Maleficarum. bugtraq mailing list, 2005.
- [6] blackngel, Malloc Des-Maleficarum. Phrack Volume 0x0d, Issue 0×42, 2009.
- [7] MaXX, Vudo malloc tricks. Phrack Volume 0x0b, Issue 0×39, 2001.
- [8]裴中煜, 张超, 段海新，Glibc堆利用的若干方法，信息安全学报，2018，http://jcs.iie.ac.cn/ch/index.aspx

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

内存信息泄露

- 在内存中，无论是代码段还是数据段都是默认**可读的**。
- 所以，攻击者可以通过悬空指针或者利用程序自身的漏洞对内存页面进行读取和扫描，导致内存信息泄露。
  - 一方面攻击者可以直接读取系统中的关键数据。
  - 另一方面可以辅助攻击者获知内存数据的具体排布，让攻击者可以精确的读取或修改内存中的某一个关键数据（如库函数的跳转地址）。

内存信息泄露

- 内存信息泄露可以分为两类：
  - 数据信息泄露
    - 缓冲区过读
    - 悬空指针
  - 代码信息泄露
    - 返回地址、函数指针
    - 跳转指令的目标地址

缓冲区过读漏洞

- 缓冲区过读，是指向从固定长度的缓冲区**读取超出预先分配长度的内容**，造成缓冲区数据过读，而获取了缓冲区相邻内存空间的数据。
- 和缓冲区溢出的原理类似，只不过缓冲区过读是指读取了过多的数据，缓冲区溢出是指写入了过多的数据。

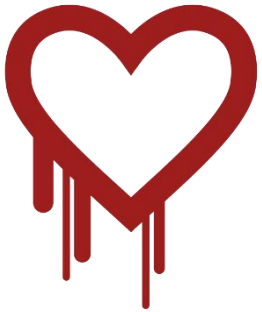
缓冲区过读漏洞

```
void func(char *output, int length)
{
   char buffer[16];
   memcpy(output, buffer, length);
}
```

- 在函数func中，memcpy()直接把buffer中长度为length的内容复制到output中。这样只要length大于16，就会造成buffer的过读，使output获得不应该获得的非法数据。
- 考虑极端情况，如果length足够大，output能够获取**几乎整个内存空间的数据**。

心脏滴血漏洞

- 心脏滴血漏洞是一个非常著名的、非常典型的、现实当中真实存在的缓冲区过读漏洞。
- 心脏滴血漏洞是一个出现在加密程序库OpenSSL的程序错误，首次于2014年4月披露。
- OpenSSL中有一个叫做“心跳”（heartbeat）的机制出现错误，没有对输入进行适当验证（缺少边界检查），从而导致心脏滴血漏洞（heartbleed）的出现。



心脏滴血漏洞

- 心脏滴血漏洞
  - 好的名字是成功的一半。
  - 是一个比较新的真实的漏洞（2014年出现）
  - 原理很简单，危害很严重，属于典型的缓冲区过读漏洞。
  - 广泛存在。OpenSSL被广泛用于实现互联网的传输层安全（TLS）协议。只要使用的是存在缺陷的OpenSSL实例，都可能因此而受到攻击。
  - 是一个非常适合用于讲课的漏洞实例，名气极大，大家都讲它。

心脏滴血漏洞

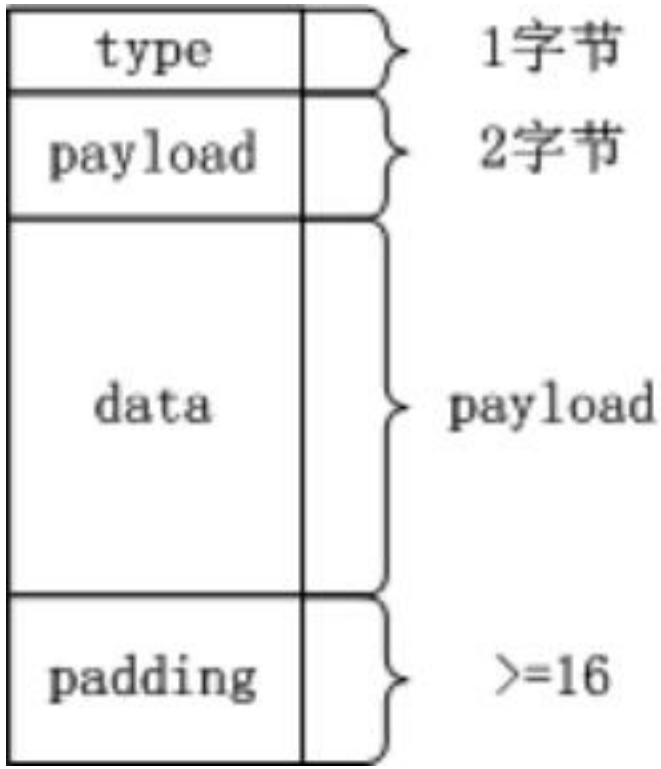
- 心跳机制：在OpenSSL中，为了保证网络连接正常，客户端和服务端通过**发送数据包来保持通信畅通**。这些数据包也被称为**心跳包**。
  - 判断当前连接是否有效、可被使用
  - 特定的客户端和服务端保持连接，防止断线

心脏滴血漏洞

- 心跳机制的基本步骤：
  - 客户端每隔一段时间发送一个探测包给服务器，同时启动一个超时定时器。**探测包包含一段用户自定义的数据。**
  - 服务器接收探测包，回复应答包。**将接收到的探测包中用户自定义的数据拷贝到返回的应答包。**
  - 如果客户端接收到应答包，并匹配两个包的数据正确，则说明服务器正常，然后将超时定时器清零复位。
  - 如果客户端一直没有接收到应答包，直到超时定时器超时，则说明和服务器的连接失败，需要额外的处理。

心脏滴血漏洞

- 用户发送的探测包和服务器返回的应答包是基本一样的，统称为心跳数据包。
  - 心跳包的具体结构：
    - type：心跳包类型，1个字节，分为探测包和应答包两种。
    - payload：实际载荷数据的长度，2个字节。所以，data长度的可能范围是0~2^{16}-1。
    - data：实际载荷数据。正常情况下，是用户自定义的数据，可以是任意的数据。
    - padding：填充数据，至少是16字节。



心脏滴血漏洞

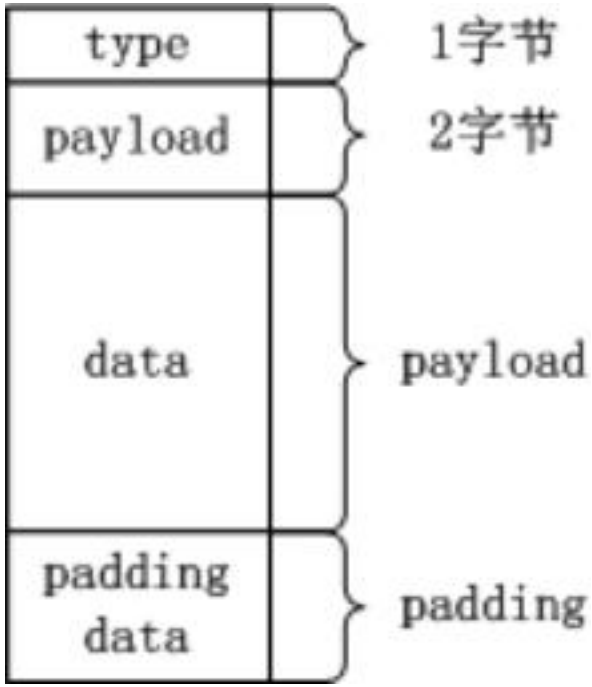
```
unsigned char *p = &s->s3->rrec.data[0], *pl;
unsigned int payload;
unsigned int padding = 16;
n2s(p,payload);
pl = p;
```

- 以上是服务器用于处理客户发来探测包的代码。
- *p = &s->s3->rrec.data[0]就是指探测包的数据。
- 宏n2s(p,payload)从探测包数据中取出对应两个字节赋值给payload。所以，payload是探测包中载荷数据的长度，也就是应答包的载荷数据长度。
- 注意：服务器返回应答包的数据的长度是由用户自行指定的！

心脏滴血漏洞

```
unsigned char *buffer, *bp;
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
```

- 以上是服务器为应答包分配的数据内存空间。
- buffer就是服务器应答包的数据，长度为：1+2+payload+padding
  - 1：消息类型，type
  - 2：payload自身的长度，即2字节
  - payload：应答数据实际长度，**由用户指定**。范围：0~2^{16}-1
  - padding：补位填充数据的长度



心脏滴血漏洞

```
s2n(payload, bp);
memcpy(bp, pl, payload);
```

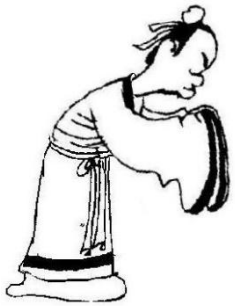
- 以上是服务器制作应答包的代码。
- 宏s2n(payload,bp)将应答包载荷数据的长度payload存入应答包中。
- memcpy将从客户端接收到的探测包数据（pl）完全拷贝到buffer中（即服务器准备发送的应答包），拷贝长度是payload，即用户自定义的数值。

心脏滴血漏洞

- 心脏滴血漏洞的具体内容：
  - 用户发送一个探测包，其中包含一段数据和这段数据的长度payload。
  - 服务器接收到探测包，将探测包数据拷贝到应答包中并返回，拷贝数据的长度等于payload。
  - 当探测包中数据的真实长度小于用户声明的长度payload时，就发生了缓冲区过读，即心脏滴血攻击。
  - 攻击效果：服务器从本地内存中拷贝多于探测包数据的内容发送给了用户，造成信息泄露。

心脏滴血漏洞

在否？请予
鄙人“saygoodbye”(10个字母)

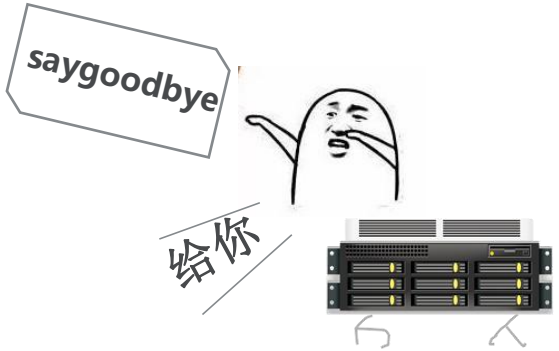


user

server

这. . .

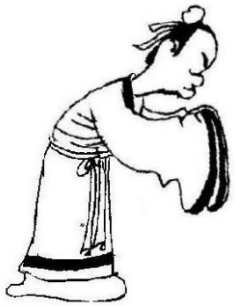
user



server

心脏滴血漏洞

在否？请予
鄙人“HACK”(500个字母)



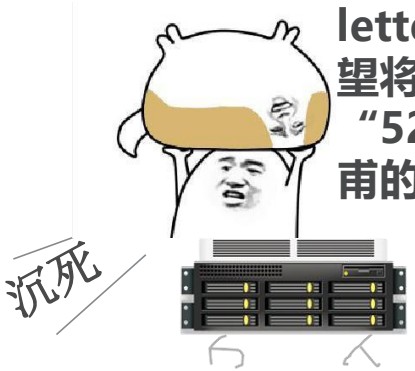
Mr Du wants these 500 letters: “HACK”

user

server

HACK.李白希望将服务器密码设为“521125”.李白请求“杜甫的个人资料”的页面… …

Mr Du wants these 500 letters:“HACK”.李白希望将服务器密码设为“521125”.李白请求“杜甫的个人资料”的页面… …



user

server

心脏滴血漏洞

心脏滴血漏洞的修补：

```
/* read type and payload length first */
if  (1 + 2 + 16 > s->s3->rrec.length)
        return  0;  /* silently discard */
hbtype = *p++;
n2s(p,  payload);
if (1 + 2 + payload +16 > s->s3->rrec.length)
        return  0;  /* silently discard per RFC 6520 sec.  4 */
pl = p;
```

- 如以上代码所示：
  - 确保探测包载荷数据不为空。
  - 确保探测包载荷数据的真实长度不小于用户声明的长度payload。

小结

- 介绍了内存信息泄露漏洞，并且以心脏滴血漏洞为例，详细介绍了缓冲区过读漏洞。
- 内存信息泄露漏洞虽然不能帮助攻击者直接控制系统运行，但是能够帮助攻击者获取系统内部的信息，是辅助攻击者攻击的一种重要手段。

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

其他内存漏洞

- 内存漏洞数量很多，变化多种多样，除了典型的栈溢出和堆漏洞以外，还有许多奇奇怪怪的形式。
- 接下来，以格式化字符串漏洞为例，再介绍一种新的内存漏洞形式。

格式化字符串漏洞

- 格式化字符串漏洞利用了*printf()类函数的参数格式问题（如printf、fprintf、 sprintf等），以此来读取或者修改内存中的数据。

格式化字符串漏洞

- *printf()类函数定义了多种不同的格式符。
- 其中，%n表示向对应参数**输出**该符号之前字符串的长度。

格式符	含义	英	传
%d	十进制数(int)	decimal	值
%u	无符号十进制数(unsigned int)	unsigned decimal	值
%x	十六进制数(unsigned int)	hexadecimal	值
%s	字符串((const)(unsigned) char *)	string	引用(地址)
%n	%n符号之前输入的字符数量(* int)	number of bytes written so far	引用(地址)

格式化字符串漏洞

- 为了打印一个字符串，通常情况如下：

```
printf("%s", str);
```

打印输出的结果是str指向的内容。

- %n的正常用法如下：

```
int pos;
printf("hello world%n", &pos);
```

“hello world”的长度是11，所以pos=11。

格式化字符串漏洞

- printf函数的定义和使用

int printf(const char *format, arg1, arg2, …);

printf(“%s%x”, str, num);

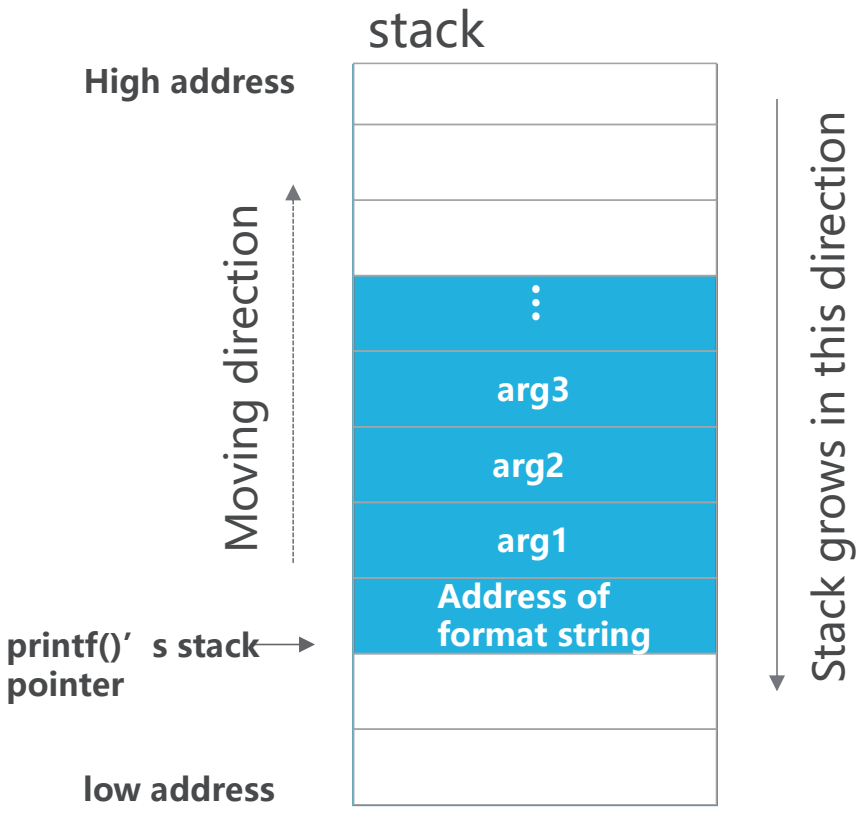
printf(“hello world%n”,&pos);

- format：格式化字符串，内部包含格式符和普通字符。
- arg1、arg2等：输入参数。
- 功能：根据format中的格式符（%s, %x, %n等），将format和输入参数arg1、arg2等按照规定的格式打印输出。

格式化字符串漏洞

int printf(const char *format,arg1,arg2,…);

- printf函数的行为由格式化字符串format控制
- 根据format中的格式符的类型和数量（如%s, %d, %p, %x等），分别从栈中format紧邻的地址（高地址方向）取得参数（arg1、arg2等）



格式化字符串漏洞

int printf(const char *format,arg1,arg2,…);

- *printf()类函数的参数数量是可变的，处理器和编译器都**不会检查**参数数量是否合理。
- 当*printf函数中format的格式符数量**大于**匹配参数的数量时，*printf()就会抓取栈上不属于该函数的参数。
- 利用格式化字符串漏洞，可以读取栈上的数据，也可以修改栈上的数据（%n）。
  - 通过%n格式符，将%n之前所有字符的长度写入相应地址

格式化字符串漏洞

```
int snprintf( char *str,  size_t size,
                const char *format, arg1, arg2, arg3 …);
```

- 四个参数：
  - str：目标字符串
  - size：拷贝数据的长度
  - format：格式化字符串
  - arg1、arg2等：匹配format中格式符的参数
- 用途：
  - 如果format的实际长度小于size，将format拷贝到str，并在最后增加一个结束符。
  - 如果format的实际长度大于或等于size，将format中size-1个字符拷贝到str，并在最后增加一个结束符。

格式化字符串漏洞

- 一个看似非常简单的小程序：formatstring.c

```
int main(int argc, char **argv)
{
   char buf[100];
   int x;

   x = 0xabab;
   snprintf(buf, sizeof (buf), argv[1]);  //格式化字符串漏洞

   printf(“buffer (%d): %s",strlen(buf),buf);
   printf("x is %d/%x (@ %p)",x,x,&x);
   return 0; }
```

格式化字符串漏洞

./formatstring “hello world”

- 执行结果如下：

buffer (11): hello world

x is 43947/0xabab (@ 0xbffff044)

- 结果表明：
  - hello world字符串长度为11个字节
  - x是0xabab，地址是0xbffff044

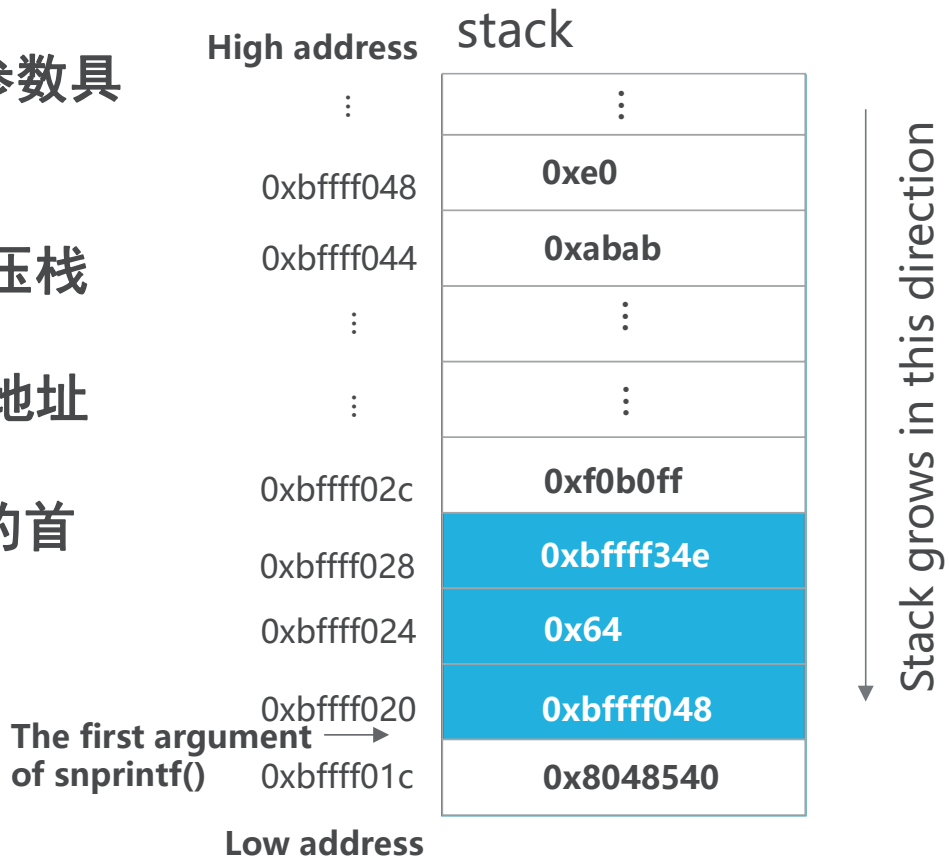
格式化字符串漏洞

```
snprintf(buf, sizeof (buf), argv[1]);
```

右图为程序正常运行时，栈上参数具体的排布情况。

阴影区域为snprintf()函数参数压栈之后的内存布局情况:

- 0xbffff048表示buf的首地址
- 0x64为buf的大小100
- 0xbffff34e表示argv[1]的首地址



格式化字符串漏洞

./formatstring “aaaa %x %x %x %x %x %x %x %x”

- 执行结果如下：

buffer (66): aaaa f0b0ff bffff05e 1 c2 bffff154 bffff05e abab 61616161

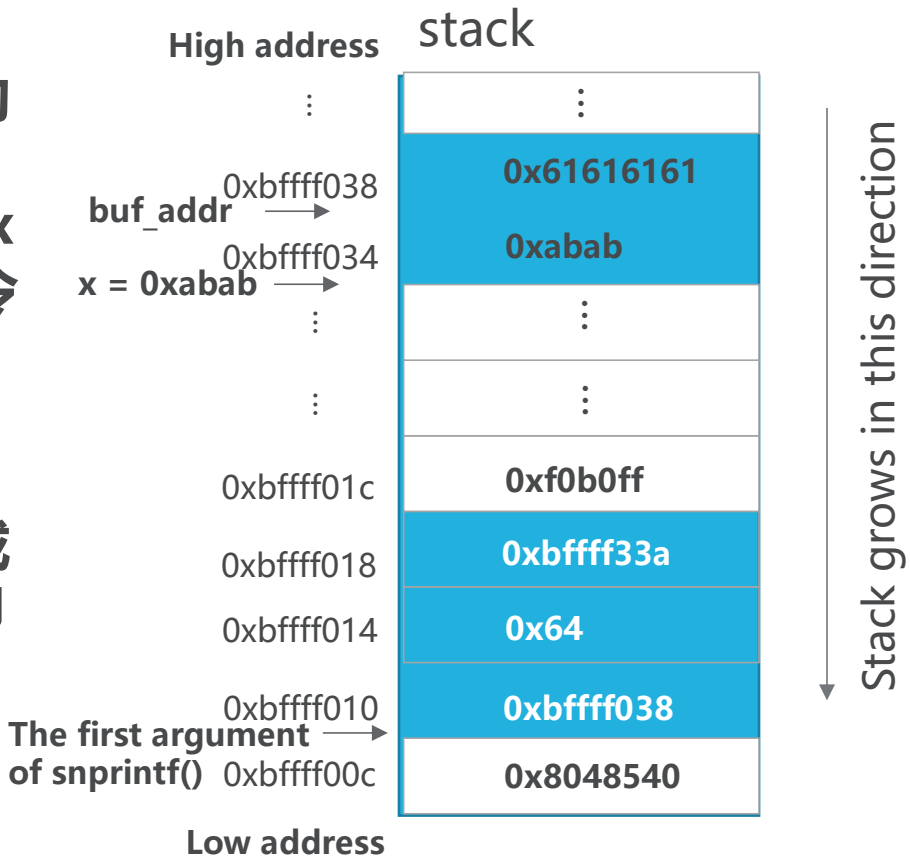
x is 43947/0xabab (@ 0xbffff034)

- 结果表明：
  - 打印输出了x的值，即0xabab
  - 打印输出了用户输入argv的值，即aaaa(0x61616161)
- 所以，如果人为恶意构造特定的输入格式化字符串，就能让snprintf打印输出一些栈上的重要数据。

格式化字符串漏洞

右图为读取栈上数据时，栈上参数具体排布情况

- 由于snprintf()函数将第三个参数后面地址中的内容当作参数，通过%x不断打印出之后栈上的数据。
- 可以看到最终打印出了程序中x变量的值0xabab和我们在命令行输入的字符串“aaaa… …”的内容。
- 如果将输入的aaaa换成内存中的地址，并且将对应的%x换成%s，就可以利用snprintf打印出内存地址中的数据了。



格式化字符串漏洞

perl -e 'system "./ formatstring", "\x74\xf0\xff\xbf %x %x %x %x %x %x %x %n" ‘

- 构造特殊输入，使得最终参数结构如下：

snprintf(buf, sizeof (buf), "\x74\xf0\xff\xbf %x %x %x %x %x %x %x %n");

- 执行结果如下：

buffer (49): t\x74\xf0\xff\xbf f0b0ff bffff09e 1 c2 bffff194 bffff09e abab

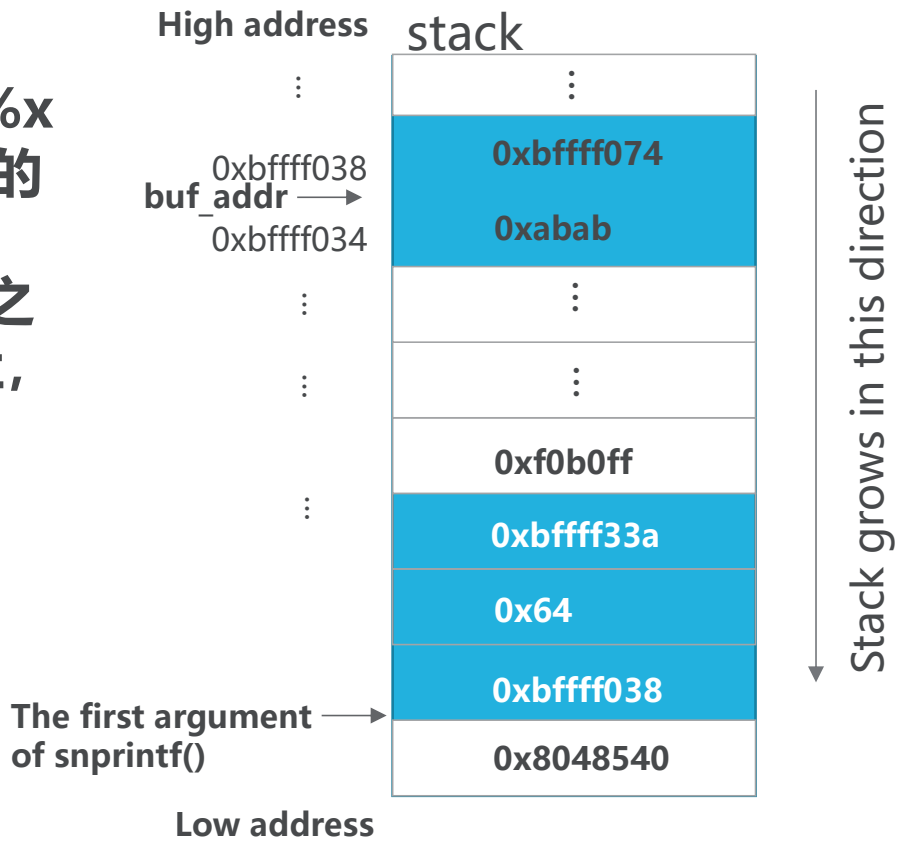
x is 49/0x31 (@ 0xbffff074)

- 结果表明：
  - x的值被修改，初始值为0xabab，修改为0x31

格式化字符串漏洞

右图为修改栈上数据时，栈上参数具体排布情况

- 通过构造输入，使得buf中保存了x的地址0xbffff074。
- 然后，通过输入的多个格式符%x，使得栈指针刚好指向buf中x的地址。
- 最后，通过%n格式符，将%n之前所有字符的长度写入x的地址，这样就修改了x的值。
  - before：x = 0xabab
  - after ： x = 0x31
- x的值变成了49/0x31



小结

- 格式化字符串漏洞的原理很简单，就是*printf()函数中的**格式符数量和参数数量不匹配**。
- 所以，只要在编译器中增加对*printf()的格式符数量和参数数量的匹配检查，就可以很容易的检测到格式化字符串漏洞。
- 参考资料：

[1] Newsham T. Format string attacks[J]. 2000.

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

其他内存漏洞

内存的主要安全问题

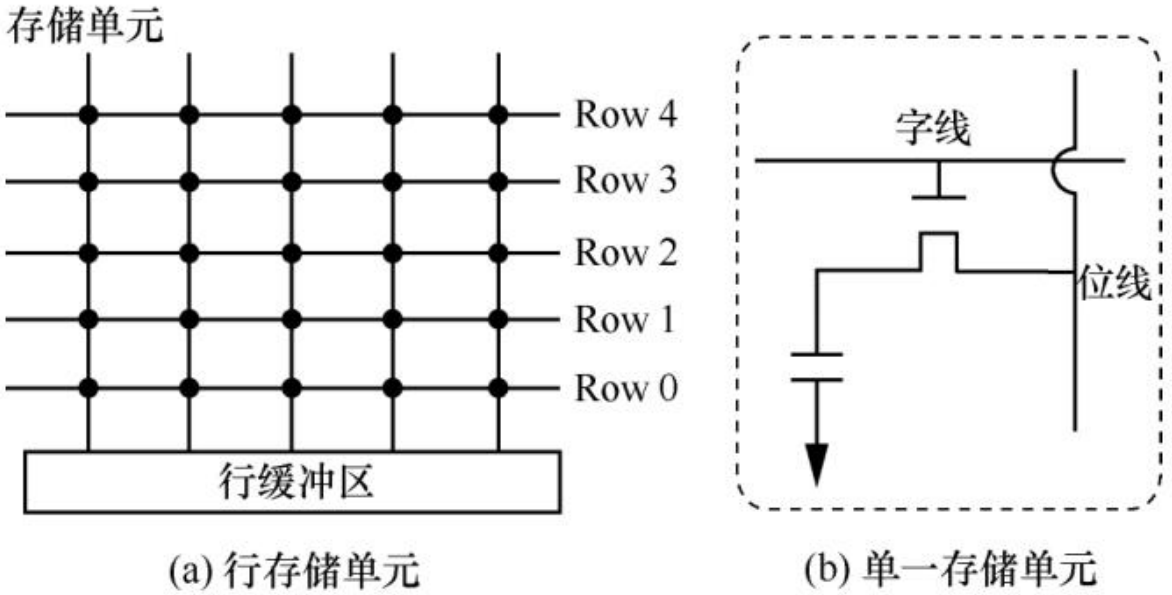
- 内存损坏漏洞（Memory corruption bug，简称**内存漏洞**）
  - 用户或程序员自行管理内存导致的安全漏洞
  - 举例：缓冲区溢出漏洞，use after free漏洞
- 内存管理问题
  - 操作系统、编译器等对内存的**管理机制**存在问题，缺少足够的检查和管控
  - 举例：**堆管理器缺陷，内存资源耗尽，占用内存未释放**
- 内存设计缺陷
  - 内存的**设计和实现**存在安全缺陷
  - 举例：**Rowhammer攻击**

Rowhammer攻击简介

- Rowhammer漏洞是由卡内基梅隆大学和英特尔实验室于2014年提出的，并在2015年由Google Zero进行攻击利用。
- 漏洞原因：动态随机访问存储器(DRAM)内存设计缺陷
- 漏洞能力：利用比特位翻转攻击DRAM，获取目标系统内核特权
- 危害：利用此硬件漏洞，可以劫持计算机系统。

Rowhammer攻击原理

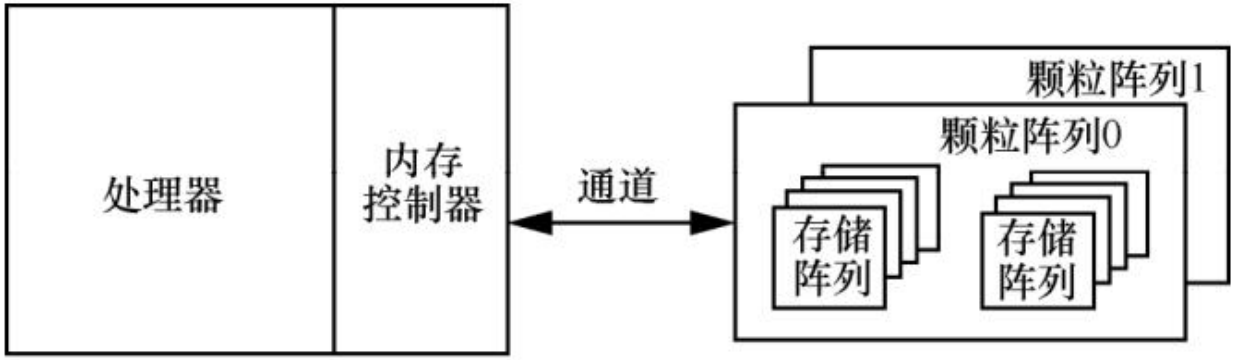
- DRAM由许多重复的存储单元组成，每个存储单元由一个电容和一个访问晶体管组成，通过电容的充放电对数据进行存储。



(a) 行存储单元　　(b) 单一存储单元

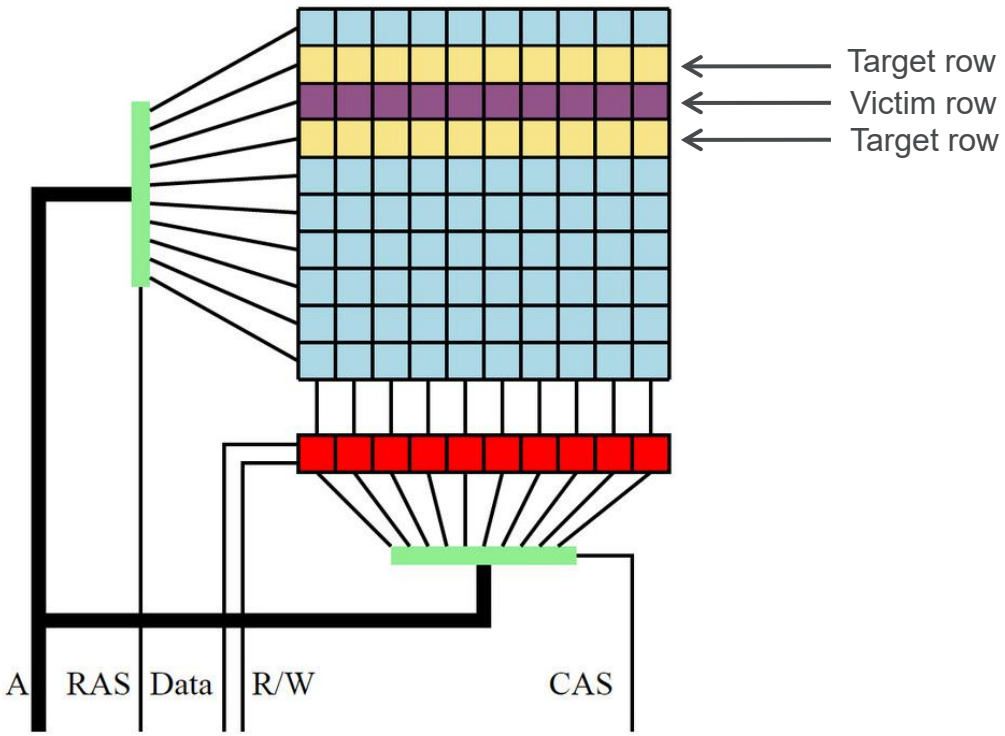
Rowhammer攻击原理

- DRAM架构由通道、颗粒阵列、存储阵列三个部分组成。
- 内存控制器与DRAM之间的物理连接模块称为通道；通道内连接到主板上的物理内存模块称为双列直插存储器模块，由一两个颗粒阵列组成；一个颗粒阵列由多个存储阵列组成；一个存储阵列是有多个单元组成的二维空间集合，通常由2^14 ~2^17 行和一个行缓冲区组成。



Rowhammer攻击原理

- 由于DRAM密度的不断增大，内存单元越来越小，导致两个相互独立的内存单元之间存在电荷干扰。
- RowHammer攻击针对这一缺陷，反复读写DRAM内存单元中的某一行来对相邻行造成干扰，导致相邻行产生位翻转现象。



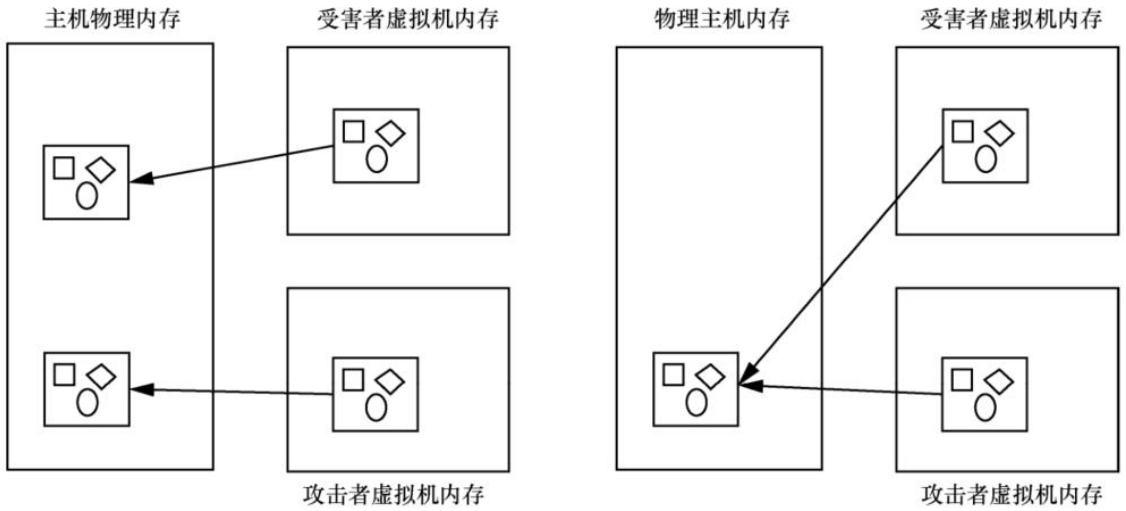
Rowhammer攻击方法

- 缓存刷新技术
  - 基于x86架构中的clflush指令来刷新缓存，让内存访问直接指向DRAM
  - 通过反复读写内存的地址X和地址Y，不断调用clflush函数来清除缓存中读入的数据，迫使处理器每次都读取内存地址中的地址X和地址Y，导致二者之间的存储单元发生翻转。

```
//攻击代码片段
code_rh:
    mov(X),%eax   //读地址X
    mov(Y),%ebx   //读地址Y
    clfush(X)     //清空缓存中从地址X读取的数据
    clfush(Y)     //清空缓存中从地址Y读取的数据
    jmp code_rh
```

Rowhammer攻击方法

- 内存重复数据删除技术
  - 内存重复数据删除通过合并多个内存页中的相同内容，来节省内存空间。但该技术存在技术漏洞，即位于同一主机的攻击者可以获取处于同一主机上相邻受害者虚拟机的内存信息。
  - 攻击者利用此漏洞，伪造一个受害者易受攻击位置的内存页，通过内存重复数据删除的合并机制，将其与受害者内存页映射到与相同的物理内存页上，从而触发RowHammer漏洞修改受害者内存地址。

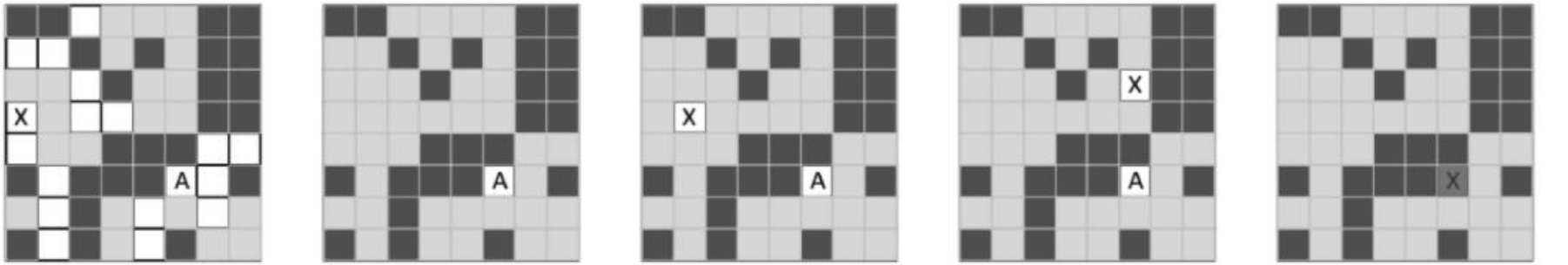


Rowhammer攻击方法

- 页表喷射技术
  - 页表喷射通过在内存中部署**大量**页表，期望至少有一个能够出现在易受攻击的物理内存上，然后使易受攻击的物理内存页发生比特翻转，触发RowHammer漏洞，具有一定概率性。
  - 具体做法为，
    - 在/dev/shm中创建一个文件，在该文件的每4KB内存开始出写入一个标记数。
    - 调用 mmap 从物理内存申请一块空间，避免内存页被分配到连续的物理内存地址。
    - 调用 madvise，从该空间中释放一个内存页，使系统重新申请一个 4 kB 内存页。
    - 最后反复调用 mmap 来映射前面生成的文件，在此过程中，通过调用munmap来释放掉可以进行bit翻转的内存页，这时系统会有很高的可能性紧接着使用这篇物理内存保存页表，反复以上过程可以触发RowHammer漏洞。

Rowhammer攻击方法

- 内存伏击技术
  - 2017年Gruss和Lipp等人首次提出内存伏击技术，该技术能够实现云端的DDoS攻击和本地的提升特权攻击。
  - 示例中展示了内存伏击的基本步骤，首先初始的内存情况包含空闲页与正被使用的页面；下一步对所有空闲页面进行填充，并将目标页X进行驱逐；然后将X重新加载到内存中的另一个位置，这样经过多次访问和驱逐，最终使X被加载到易受攻击的位置A上。



(a) 开始　(b) 驱逐　(c) 访问二进制　(d) 重复驱逐和访问　(e) 到达目标位置后停止

Rowhammer攻击方法

- Rowhammer不同攻击方法对比

RowHammer 攻击技术	攻击技术特点	攻击技术的局限
缓存刷新技术	操作简单，只需调用clflush指令就能使缓存失效	仅适用于 x86 架构，且部分系统已禁止用户级别访问 clflush 指令
内存重复数据删除技术	攻击者可以轻松修改目标的内存地址且不易检测	部分系统已禁用该技术
页表喷射技术	能使目标页表位于指定位置	会耗尽整个内存
内存伏击技术	不会耗尽内存	所耗时间较长

Rowhammer防御方法

- 硬件解决方法
  - ECC技术：即错误更正码，用来检验存储在DRAM中的整体数据。
  - PARA(probabilistic adjacent row activation)技术：原理是当每一个行关闭时，内存控制器都会以一定的概率来刷新该行邻接行的值。
  - 提高刷新速率
  - TRR(target row refresh)技术：TRR应用在DRR4内存中，采用一种特殊模块，可以跟踪记录内存中哪些行被经常激活，并且刷新这些激活行相邻行的值。

Rowhammer防御方法

- 软件解决方法
  - 禁止使用cache相关指令：当使用clfush指令时，可导致直接访问DRAM，从而触发RowHammer漏洞，目前谷歌沙箱已禁止使用clflush指令。
  - 禁止一般用户访问物理地址接口pagemap：如果一般用户访问pagemap，它能够获取所有物理地址和虚拟地址之间的映射关系，有效地反复访问物理内存中的具体行，从而触发RowHammer漏洞。

Rowhammer防御方法

- 软件解决方法
  - 基于虚拟化的内存隔离技术RDXA(RowHammer defense on Xen allocator)
  - 该防御技术分为两个阶段：
    - 通过基于访问时间的旁路分析方法，逆向出不同物理地址和实际内存在芯片上的布局关系，存储在数据库。
    - 虚拟监视器中，将物理内存分为DRAM内存上互不相邻的若干区域，并通过分配算法分给至少隔离一行的不同安全实体，保证不同实体间所对应的物理内存不会存放在芯片的同一存储阵列的相邻行上

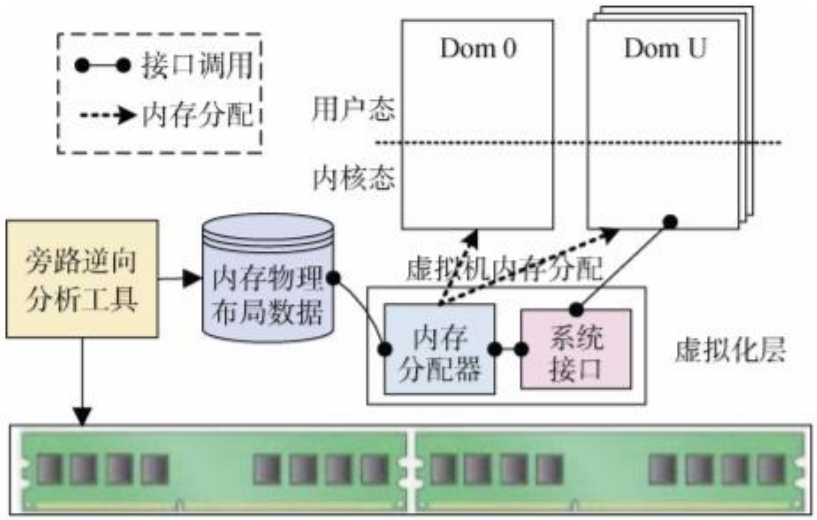


图 3　RDXA 系统架构图

Rowhammer攻击与防御的发展

- Rowhammer攻击方法的研究方向：
  - 提高攻击的稳定性、成功率和降低时间消耗，使得攻击者能够更好的定向地攻击内存中某个特定位置。
- Rowhammer防御技术的发展思路：
  - 第一，短期的缓解措施，能够应对当前的攻击手段，包括增加限制条件、提高攻击的不稳定性和增加时间消耗等
  - 第二，长期的解决方案，旨在消除硬件漏洞，受实际情况和现有技术限制，该方向的研究还有很长的路要走

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

总结

- 介绍了内存漏洞相关基础知识，并且详细讲解了几类典型的内存漏洞，包括：
  - 缓冲区溢出漏洞（栈溢出和堆溢出）
  - 堆漏洞漏洞（堆溢出、double free和use after free）
  - 内存信息泄露漏洞（缓冲区过读、心脏滴血）
  - 格式化字符串漏洞
  - Rowhammer攻击
- 以上内容基本涵盖了所有常见的内存漏洞类型。

总结

- 内存漏洞是最重要最常见的一类软件漏洞，是计算机系统安全研究的基础之一。
  - **构造非法输入，修改或者读取内存数据，控制程序运行**
  - **直接原因：C/C++等语言允许用户直接管理内存，缺少对边界、指针的检查**
    - 空间漏洞：**指针越界**，让指针指向合法范围以外的地址
    - 时间漏洞：**指针悬空**，指针释放后的非法利用

总结

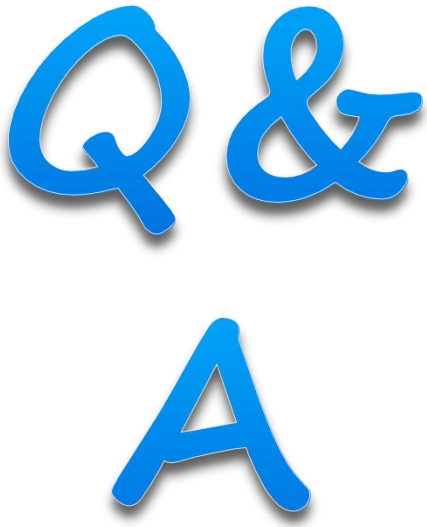
- 对于一些原理简单、特征明显、容易修改的内存漏洞，比如格式化字符串漏洞和心脏滴血漏洞，**可以通过特征检测、打补丁**的方式来弥补。
  - 但是，打补丁不是一个通用的防御方法。对于其他更加复杂、类型多样的内存漏洞，只通过打补丁的方式，防御效果不好。
- 到目前为止，对于内存漏洞，**还没有找到一种通用可行且实用**的防御方法。
  - 比如，栈溢出漏洞是一种七八十年代就被发现的漏洞，原理也很清晰。但是因为种种原因，到目前为止还是最为常见的一种内存漏洞，被广泛用于对计算机系统的各种攻击。

总结

- 研究如何防御内存漏洞不是我们的研究目标。我们的研究目标是：在存在内存漏洞的情况下，如何设计一个安全的计算机系统。
- 如果内存漏洞存在，那么攻击者就能够利用内存漏洞修改或者读取内存中的数据。
- 因此，我们需要进一步研究，在攻击者能够**任意修改或读取内存数据**的前提下，攻击者到底是如何进行攻击的，我们又应该如何针对这些攻击行为进行防御？

经典论文推荐

- SoK：Eternal War in Memory，IEEE S&P 2013
  - 信息安全领域四大国际顶级会议，CCF-A类会议（CCS，S&P，Usenix Security，NDSS）
  - 这是一篇关于内存漏洞的综述论文，对内存漏洞进行了非常全面而深刻的分析，强烈推荐！！！
  - 读懂了这篇论文，就基本掌握了内存漏洞和攻击的主要内容。


Q &
A