堆溢出漏洞简介

原创

zh_explorer 于 2018-05-14 11:04:00 发布 6563 收藏 18

分类专栏： pwn

本文链接：https://blog.csdn.net/zh_explorer/article/details/80306976

简介

这次来介绍一下堆溢出漏洞。不过这次的堆溢出漏洞比较复杂，不像栈溢出一样容易理解。所以这一次的内容会比较多。我尽量详细的介绍堆溢出漏洞，以及相关的知识。

首先，关于神马是堆溢出。简而言之就是在堆上产生的溢出。一般我们使用malloc等函数申请的内存都会储存在堆段上。并且由操作系统来管理已经使用和剩余内存，完成内存分配以及回收等等的操作。而堆溢出便是因为程序员的粗心大意，造成了读入的数据超过了malloc申请的内存大小而产生的漏洞。

前提知识

要了解堆溢出到底是如何造成危害的，就需要对linux下的内存管理有一定的了解。因为内存管理比较复杂，所以这里另开一坑对漏洞利用有关的部分进行浅析。如果是了解内存管理的大神可以跳过。

传送门

然后需要了解的是漏洞利用中一种技巧，Dword shoot的使用方法，这个也开一个坑另行介绍。同样如果是了解Dword shoot或者说是双向链表溢出利用的大神可以跳过

传送门

漏洞利用介绍

如果已经基本了解了linux操作系统中内存管理的方式，并且清楚Dword shoot的利用方法。那么接下来我们就要具体的介绍堆溢出漏洞的产生原因以及利用的方法了。

我们已经知道，内存中空闲的堆快会通过一个双向链表连接在一起。而我们能够篡改内存的机会便是在双向链表的节点在删除的时候因为溢出而产生的Dword shoot。而我们需要的便是构造堆内存，篡改指针，并且让系统进行链表的删除，也就是unlink的工作。

1.触发unlink

首先，我们需要能够让内存块进行unlink。为了让系统如我们所愿的进行unlink，要满足几个条件。第一个是分配的内存不能太小，在linux中，小于一定大小的内存会用fastbins的形式进行分配，这样的内存在free后不会用双向链表管理。第二个就是需要有相邻的两块内存先后被free。在一块内存被free时，系统会检查与之相邻的前后2块内存块是否在使用。如果处于空闲状态的话，就会将先前free的内存从双向链表中删除，然后将2块内存合并成一块大的空闲内存再重新链接入双向链表中。而我们利用的就是这个解绑定的过程，如果我们把构成双向链表的两个指针改变的话，就可以在unlink时篡改任意内存了。

2.绕过系统保护

虽然Dword shoot篡改任意内存的理想很丰满，但是现实却是骨感的。操作系统不会对堆溢出攻击没有任何防护，所以我们还需要想办法绕过操作系统的保护。(unlink中增加保护好像是从2.3.6版本的libc开始的，没有求证过。在这之前确实没有任何保护)
操作系统对堆溢出的保护主要是在unlink的时候有如下的检测代码

```
FD = P->fd;
BK = P->bk;
if (FD->bk != P || BK->fd != P)
  malloc_printerr (check_action, "corrupted double-linked list", P, AV);
```

需要说明的是，这里的p指向的是整个chunk，包括chunk头的数据都在内，而不是通过malloc返回的指向chunk中数据段的指针。这里的检测目的便是确保双向链表的指针正确。可以看到，这个检测就是当前堆块的上一个堆块的下一个堆块和下一个堆块的上一个堆块都是指向当前堆块的。2333333双向链表的问题慢慢理解吧。

上张图比较好理解一些。

手绘轻喷....

那么，应该怎么绕过呢，这里需要一个指针，一个指向当前chunk的指针。只要有这个指针就可以绕过防御。简单起见，假设我们正好有一个指向chunk头的指针p，而且我们知道它的地址ptr。那么我们就可以这样构造两个双向链表指针。

> p->fd=ptr-0xc
> p->bk=ptr-0x8

这里的0xc和0x8便代表了chunk中bk和fd指针的位置。这样的话p -> fd -> bk 也就是FD -> bk 就是FD + 0xc正好就是指向chunk的指针p。如此就可以绕过保护。

最后，在系统执行unlink的时候，就会执行这样的代码

```
FD=p->fd(实际是ptr-0xc)
BK=p->bk(实际是ptr-0x8)
FD->bk=BK(p=ptr-0x8)
BK->fd=FD(p=ptr-0xc)
```

可以看到，这样子的结果就是本来指向p的指针变成了指向ptr-0xc，指向了比自身稍微低一点的地址，如果程序中有修改堆段的代码，那么只要构造请求，就可以改写p指针为我们需要的东西了。

当然，有可能程序中不会正好存在一个指向chunk头的指针，不过，只要要有指向堆段的指针也就够了。我们就拿一个最常见的，malloc返回的指向chunk中数据开头的指针为例子来说明吧。
假设在程序将malloc返回的指针放在了.data段中，那么我们可以轻易获得它的地址。然后我们需要的是伪造chunk来骗过操作系统。

可以看到在chunk0中我们布置了一些内容，让它看上去是一个已经被free的内存块，首先是prev_size我们可以填0伪造前一个堆块正在使用。然后是szie&flag这里我们填上伪造的chunk大小，也就是图中的new size。注意这个大小是算上chunk头的。然后在加1表示前一个chunk正在使用。然后就是最重要的2个双向链表指针，分别是ptr-0xc和ptr-0x8。最后在随便填点东西填满chunk。这样子我们就伪造出了一个被free的chunk。
当然，光光长得像还不够，我们需要骗过操作系统，让操作系统以为我们伪造的chunk已经被free而且是从p这里开始的。这里我们就需要覆盖下一个chunk的chunk头部分。第一是要把prev_size的大小改成我们的new size。操作系统就是靠这个来确认上一个堆快的是从哪里开始的。然后就是把size中的最低位至0欺骗操作系统上一个堆块已经free。
这样，所有的伪造就完成了。当我们free掉chunk1的时候就可以看到指针p已经被改写了。接下来就可以用一些别的技巧来完成getshell的任务。

实践

学习的最好方法就是实践。所以我自己写了一个简单的带堆溢出漏洞的程序，并且通过它来获取一台linux主机的shell。我将利用的poc和writeup另外再开一坑介绍。
传送门