文 Linux 堆溢出漏洞利用之 unlink

linux 阿里聚安全 阿里聚安全 1 天前发布

0 前言

之前我们深入了解了glibc malloc的运行机制（文章链接请看文末▼），下面就让我们开始真正的堆溢出漏洞利用学习吧。说实话，写这类文章，我是比较怂的，因为我当前从事的工作跟漏洞挖掘完全无关，学习这部分知识也纯粹是个人爱好，于周末无聊时打发下时间，甚至我最初的目标也仅仅是能快速看懂、复现各种漏洞利用POC而已…鉴于此，后续的文章大致会由两种内容构成：1)各种相关文章的总结，再提炼；2)某些好文章的翻译及拓展。本文两者皆有，主要参考文献见这里。

1 背景介绍

首先，存在漏洞的程序如下：

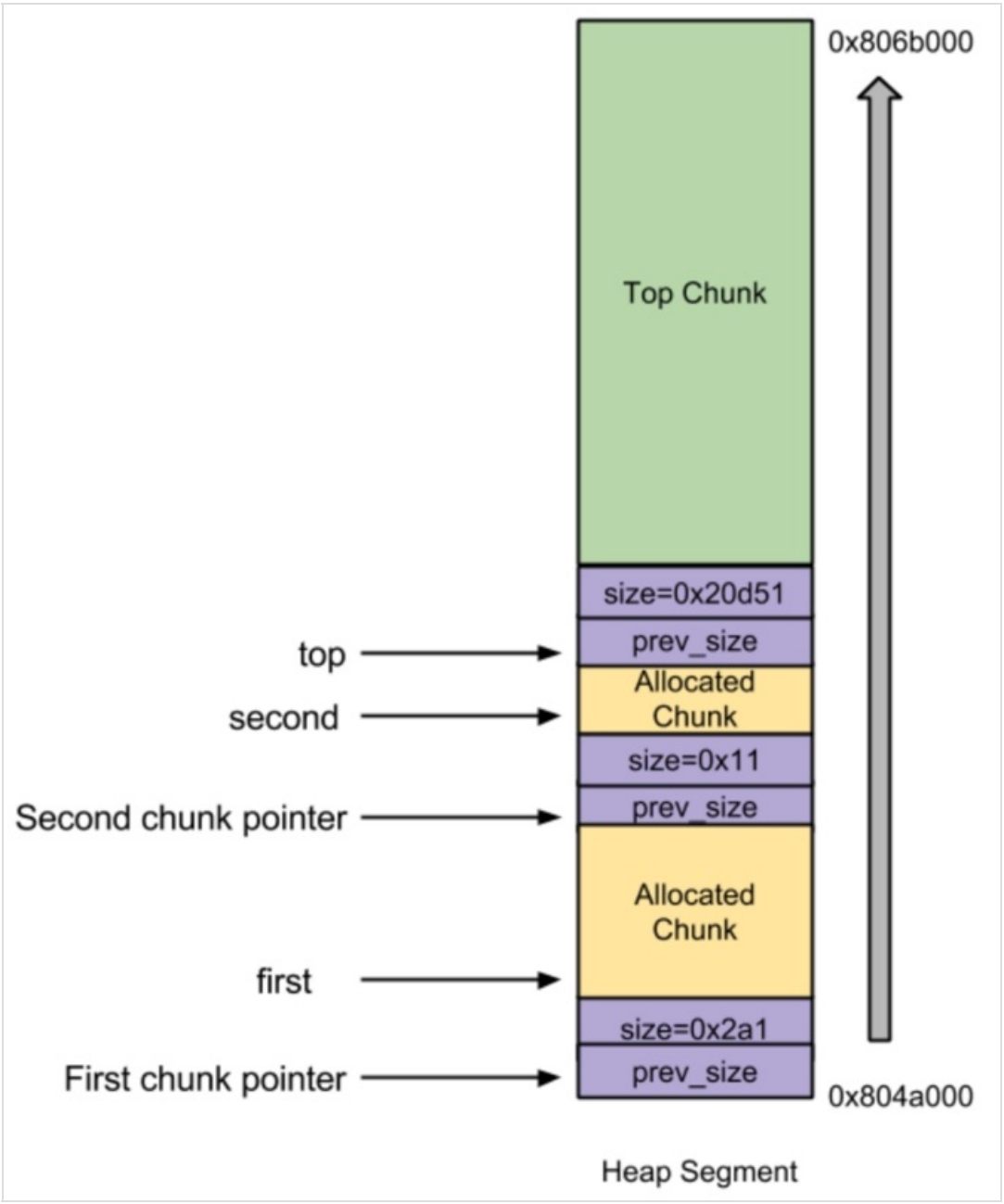
```
/*
 Heap overflow vulnerable program.
 */
#include<stdlib.h>
#include<string.h>

int main( intargc, char * argv[] )
{
        char * first, * second;

/*[1]*/ first= malloc( 666 );
/*[2]*/ second= malloc( 12 );
        if(argc!=1)
/*[3]*/      strcpy( first, argv[1] );
/*[4]*/ free(first );
/*[5]*/ free(second );
/*[6]*/ return(0 );
}
```

在代码[3]中存在一个堆溢出漏洞：如果用户输入的argv[1]的大小比first变量的666字节更大的话，那么输入的数据就有可能覆盖掉下一个chunk的chunk header——这可以导致任意代码执行。而攻击的核心思路就是利用glibc malloc的unlink机制。

上述程序的内存图如下所示：



2 unlink技术原理

2.1 基本知识介绍

unlink攻击技术就是利用”glibc malloc”的内存回收机制，将上图中的second chunk给unlink掉，并且，在unlink的过程中使用shellcode地址覆盖掉free函数(或其他函数也行)的GOT表项。这样当程序后续调用free函数的时候(如上面代码[5])，就转而执行我们的shellcode了。显然，核心就是理解glibc malloc的free机制。

在正常情况下，free的执行流程如下文所述：

PS：鉴于篇幅，这里主要介绍非mmaped的chunks的回收机制，回想一下在哪些情况下使用mmap分配新的chunk，哪些情况下不用mmap？

一旦涉及到free内存，那么就意味着有新的chunk由allocated状态变成了free状态，此时glibc malloc就需要进行合并操作——向前以及(或)向后合并。这里所谓向前向后的概念如下：将previous free chunk合并到当前free chunk，叫做向后合并；将后面的free chunk合并到当前free chunk，叫做向前合并。

一、向后合并

相关代码如下：

```
    /*malloc.c  int_free函数中*/
/*这里p指向当前malloc_chunk结构体，bck和fwd分别为当前chunk的向后和向前一个fre
e chunk*/
/* consolidate backward */
   if (!prev_inuse(p)) {
     prevsize = p->prev_size;
size += prevsize;
//修改指向当前chunk的指针，指向前一个chunk。
     p = chunk_at_offset(p, -((long) prevsize));
     unlink(p, bck, fwd);
}
```

```
//相关函数说明：
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s)))

/*unlink操作的实质就是：将P所指向的chunk从双向链表中移除，这里BK与FD用作临时
变量*/
#define unlink(P, BK, FD) {                                          \
   FD = P->fd;                                                       \
   BK = P->bk;                                                       \
   FD->bk = BK;                                                      \
   BK->fd = FD;                                                      \
   ...
}
```

首先检测前一个chunk是否为free，这可以通过检测当前free chunk的PREV_INUSE(P)比特位知晓。在本例中，当前chunk（first chunk）的前一个chunk是allocated的，因为在默认情况下，堆内存中的第一个chunk总是被设置为allocated的，即使它根本就不存在。

如果为free的话，那么就进行向后合并：

1. 将前一个chunk占用的内存合并到当前chunk;
2. 修改指向当前chunk的指针，改为指向前一个chunk。
3. 使用unlink宏，将前一个free chunk从双向循环链表中移除(这里最好自己画图理解，学过数据结构的应该都没问题)。

在本例中由于前一个chunk是allocated的，所以并不会进行向后合并操作。

二、向前合并操作

首先检测next chunk是否为free。那么如何检测呢？很简单，查询next chunk之后的chunk的PREV_INUSE (P)即可。相关代码如下：

```
.....
/*这里p指向当前chunk*/
nextchunk = chunk_at_offset(p, size);
.....
nextsize = chunksize(nextchunk);
.....
if (nextchunk != av->top) {
    /* get and clear inuse bit */
    nextinuse =inuse_bit_at_offset(nextchunk, nextsize);//判断nextchunk是否为free ch
unk
    /* consolidate forward */
    if (!nextinuse) { //next chunk为free chunk
        unlink(nextchunk, bck, fwd);//将nextchunk从链表中移除
      size += nextsize; //p还是指向当前chunk只是当前chunk的size扩大了，这就是向
前合并!
    } else
          clear_inuse_bit_at_offset(nextchunk, 0);
|
    .....
  }
```

整个操作与”向后合并“操作类似，再通过上述代码结合注释应该很容易理解free chunk的向前结合操作。在本例中当前chunk为first，它的下一个chunk为second，再下一个chunk为top chunk，此时top chunk的 PREV_INUSE位是设置为1的(表示top chunk的前一个chunk，即second chunk,已经使用)，因此first的下一个chunk不会被“向前合并“掉。

介绍完向前、向后合并操作，下面就需要了解合并后(或因为不满足合并条件而没合并)的chunk该如何进一步处理了。在glibc malloc中，会将合并后的chunk放到unsorted bin中(还记得unsorted bin的含义么？)。相关代码如下：

```
/*
 Place the chunk in unsorted chunk list. Chunks are not placed into regular bins until aft
er they have been given one chance to be used in malloc.
*/

bck = unsorted_chunks(av); //获取unsorted bin的第一个chunk
/*
 /* The otherwise unindexable 1-bin is used to hold unsorted chunks. */
  #define unsorted_chunks(M)          (bin_at (M, 1))
*/
    fwd = bck->fd;
    ......
    p->fd = fwd;
    p->bk = bck;
    if (!in_smallbin_range(size))
      {
        p->fd_nextsize = NULL;
        p->bk_nextsize = NULL;
      }
```

```
    bck->fd = p;
    fwd->bk = p;

    set_head(p, size | PREV_INUSE);//设置当前chunk的size,并将前一个chunk标记为已
使用
set_foot(p, size);//将后一个chunk的prev_size设置为当前chunk的size
/*
  /* Set size/use field */
  #define set_head(p, s)       ((p)->size = (s))
  /* Set size at footer (only when chunk is not in use) */
  #define set_foot(p, s)       (((mchunkptr) ((char *) (p) + (s)))->prev_size = (s))
*/
```

上述代码完成的整个过程简要概括如下：将当前chunk插入到unsorted bin的第一个chunk(第一个chunk是链表的头结点，为空)与第二个chunk之间(真正意义上的第一个可用chunk)；然后通过设置自己的size字段将前一个chunk标记为已使用；再更改后一个chunk的prev_size字段，将其设置为当前chunk的size。

注意：上一段中描述的”前一个“与”后一个“chunk，是指的由chunk的prev_size与size字段隐式连接的chunk，即它们在内存中是连续、相邻的！而不是通过chunk中的fd与bk字段组成的bin(双向链表)中的前一个与后一个chunk，切记！

在本例中，只是将first chunk添加到unsorted bin中。

2.2 开始攻击

现在我们再来分析如果一个攻击者在代码[3]中精心构造输入数据并通过strcpy覆盖了second chunk的chunk header后会发生什么情况。

假设被覆盖后的chunk header相关数据如下：

1. prev_size = 一个偶数，这样其PREV_INUSE位就是0了，即表示前一个chunk为free。
2. size = -4
3. fd = free函数的got表地址address – 12；(后文统一简称为“free addr – 12”)
4. bk = shellcode的地址

那么当程序在[4]处调用free(first)后会发生什么呢？我们一步一步分析。

一、向后合并

鉴于first的前一个chunk非free的，所以不会发生向后合并操作。

二、向前合并

先判断后一个chunk是否为free，前文已经介绍过，glibc malloc通过如下代码判断：

```
nextinuse =inuse_bit_at_offset(nextchunk, nextsize);
这里inuse_bit_at_offset宏定义如下：
/* check/set/clear inuse bits in known places */
#define inuse_bit_at_offset(p, s)                                 \
 (((mchunkptr) (((char *) (p)) + (s)))->size& PREV_INUSE)
```

PS：在本例中next chunk即second chunk，为了便于理解后文统一用next chunk。

从上面代码可以知道，它是通过将nextchunk + nextsize计算得到指向下下一个chunk的指针，然后判断下下个chunk的size的PREV_INUSE标记位。在本例中，此时nextsize被我们设置为了-4，这样glibc malloc就会将next chunk的prev_size字段看做是next-next chunk的size字段，而我们已经将next chunk的prev_size字段设置为了一个偶数，因此此时通过inuse_bit_at_offset宏获取到的nextinuse为0，即next chunk为free！既然next chunk为free，那么就需要进行向前合并，所以就会调用unlink(nextchunk, bck, fwd);函数。真正的重点就是这个unlink函数！

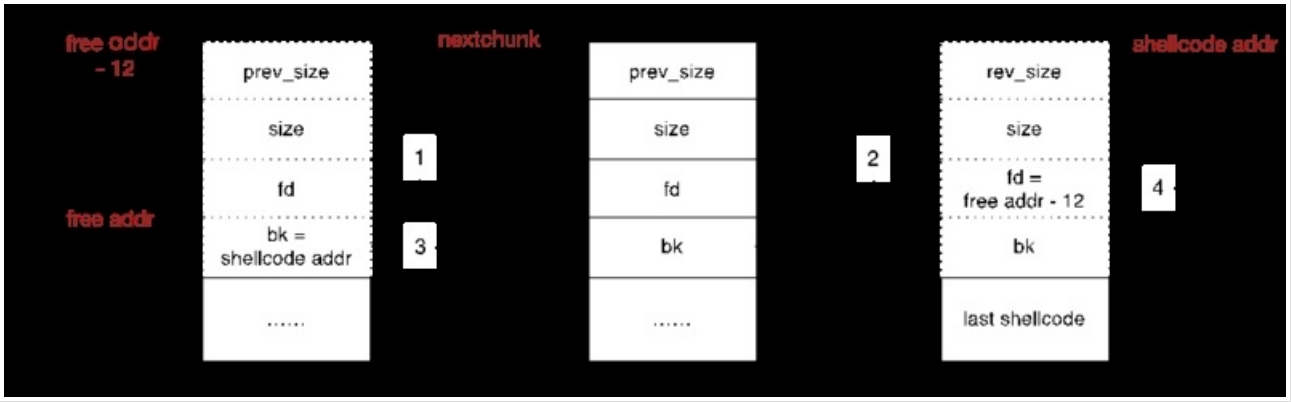
在前文2.1节中已经介绍过unlink函数的实现，这里为了便于说明攻击思路和过程，再详细分析一遍，unlink代码如下：

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

此时 P = nextchunk, BK = bck, FD = fwd。

1. 首先FD = nextchunk->fd = free地址– 12;
2. 然后BK = nextchunk->bk = shellcode起始地址；
3. 再将BK赋值给FD->bk，即（free地址– 12）->bk = shellcode起始地址；
4. 最后将FD赋值给BK->fd，即(shellcode起始地址)->fd = free地址– 12。

前面两步还好理解，主要是后面2步比较迷惑。我们作图理解：



结合上图就很好理解第3，4步了。细心的朋友已经注意到，free addr -12和shellcode addr对应的prev_size等字段是用虚线标记的，为什么呢？因为事实上它们对应的内存并不是chunk header，只是在我们的攻击中需要让glibc malloc在进行unlink操作的时候将它们强制看作malloc_chunk结构体。这样就很好理解为什么要用free addr – 12替换next chunk的fd了，因为(free addr -12)->bk刚好就是free addr，也就是说第3步操作的结果就是将free addr处的数据替换为shellcode的起始地址。

由于已经将free addr处的数据替换为了shellcode的起始地址，所以当程序在代码[5]处再次执行free的时候，就会转而执行shellcode了。

至此，整个unlink攻击的原理已经介绍完毕，剩下的工作就是根据上述原理，编写shellcode了。只不过这里需要注意一点，glibc malloc在unlink的过程中会将shellcode + 8位置的4字节数据替换为free addr – 12，所以我们编写的shellcode应该跳过前面的12字节。

3 对抗技术

当前，上述unlink技术已经过时了(但不代表所有的unlink技术都失效，详情见后文)，因为glibc malloc对相应的安全机制进行了加强，具体而言，就是添加了如下几条安全检测机制。

3.1 Double Free检测

该机制不允许释放一个已经处于free状态的chunk。因此，当攻击者将second chunk的size设置为-4的时候，就意味着该size的PREV_INUSE位为0，也就是说second chunk之前的first chunk(我们需要free的chunk)已经处于free状态，那么这时候再free(first)的话，就会报出double free错误。相关代码如下：

```
/* Or whether the block is actually not marked used. */
  if (__glibc_unlikely(!prev_inuse(nextchunk)))
   {
        errstr = "double free or corruption(!prev)";
        goto errout;
   }
```

3.2 next size非法检测

该机制检测next size是否在8到当前arena的整个系统内存大小之间。因此当检测到next size为-4的时候，就会报出invalid next size错误。相关代码如下：

```
  nextsize =chunksize(nextchunk);
  if (__builtin_expect(nextchunk->size <= 2 * SIZE_SZ, 0)
              || __builtin_expect (nextsize>= av->system_mem, 0)){
        errstr = "free(): invalid next size(normal)";
        goto errout;
  }
```

3.3 双链表冲突检测

该机制会在执行unlink操作的时候检测链表中前一个chunk的fd与后一个chunk的bk是否都指向当前需要unlink的chunk。这样攻击者就无法替换second chunk的fd与fd了。相关代码如下：

```
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))              \
    malloc_printerr (check_action,"corrupted double-linked list", P);    \
```

4 另一种unlink攻击技术

经过上述3层安全检测，是否意味着所有unlink技术都失效了呢？答案是否定的，因为进行漏洞攻击的人脑洞永远比天大！之前刚好看到一篇好文][15，主讲在Android4.4上利用unlink机制实现堆溢出攻击。众所周知，Android内核基于linux，且其堆内存管理也是使用的glibc malloc，虽然在一些细节上有些许不同，但核心原理类似。该文介绍的攻击方式就成功绕过了上述三层检测。

5 总结

本文详细介绍了unlink攻击技术的核心原理，虽然上述介绍的unlink漏洞利用技术已经失效，而其他的unlink技术难度也越来越大，但是我们还是有必要认真学习，因为它一方面可以进一步加深我们对glibc malloc的堆栈管理机制的理解，另一方面也为后续的各种堆溢出攻击技术提供了思路。

从上文的分析可以看出，unlink主要还是利用的glibc malloc中隐式链表机制，通过覆盖相邻chunk的数据实现攻击，那么我们能不能在显示链表中也找到攻击点呢？请关注下一篇文章：基于fastbin的堆溢出漏洞利用介绍。

附：Linux技术分析系列文章

- Linux堆内存管理深入分析(上)
- Linux堆内存管理深入分析（下）

> 作者：走位@阿里聚安全，更多安全技术文章，请访问阿里聚安全博客

本文隶属于专栏

阿里聚安全

阿里聚安全（http://jaq.alibaba.com）由阿里巴巴移动安全部出品，面向企业和开发者提供企业安全解决方案，全面覆盖移动安全、数据风控、内容安全、实人认证等维度，并在业界率先提出“以业务为中心的安全”，赋能生态，与行业共享阿里巴巴集团多年沉淀的专业安全能力。

阿里聚安全
作者

关注专栏

相关收藏夹

换一组

常用命令
26 个条目 | 0 人关注

文章
5 个条目 | 0 人关注

技巧
5 个条目 | 0 人关注

分享扩散：