缓冲区溢出基础

watercloud @ xfocus.org
2006-2-14

缓冲区溢出通常是向数组中写数据时写入的数据超出了数组原始定义的大小。比如前面你定义了 int buff[10]，但后来往里面写入数据时出现了 buff[12]=0x10。C语言常用的strcpy、sprintf、strcat等函数都非常容易导致缓冲区溢出问题。查阅C语言编程的书籍时通常会告诉你程序溢出后会发生不可预料的结果。在网络安全领域，缓冲区溢出利用的艺术在于让这个“不可预料的结果”变为我们期望的结果。

看下面这个演示程序：buf.c

```
#include<stdio.h>
void why_here(void)    /*这个函数没有任何地方调用过 */
{
        printf("why u here ?!\n");
        _exit(0);
}
int main(int argc,char * argv[])
{
        int buff[1];
        buff[2]=(int)why_here;
        return 0;
}
```

在命令行用 VC的命令行编译器编译(在 Linux下用 gcc编译并运行也是同样结果）：

```
C:\Temp>cl buf.c
```

运行程序：

```
C:\Temp>buf.exe
why u here ?!
```

仔细分析程序和打印信息，你可以发现程序中我们没有调用过why_here函数，但该函数却在运行的时候被调用了！这里唯一的解释是buff[2]=why_here;操作导致。要解释此现象需要理解一些C语言底层（和计算机体系结构相关）及一些汇编知识，尤其是“栈”和汇编中CALL/RET的知识，如果这方面你尚有所欠缺的话建议参考一下相关书籍，否则后面的内容会很难跟上。

假设你已经有了对栈的基本认识，我们来理解一下程序运行情况：

进入 main函数后的栈内容下：

```
[  eip  ][  ebp  ][ buff[0] ]
高地址    <----      低地址
```

以上 3个存储单元中 eip为main函数的返回地址，buff[0]单元就是buff申明的一个 int空

间。程序中我们定义 int buff[1]，那么只有对 buff[0]的操作才是合理的（我们只申请了一个 int 空间），而我们的 buff[2]=why_here 操作超出了 buff 的空间，这个操作越界了，也就是溢出了。溢出的后果是： buff[2]其实就是操作了栈中的 eip 存放单元，将 main 函数的返回地址改为了 why_here 函数的入口地址。这样 main 函数结束后返回的时候将这个地址作为了返回地址而加以运行。

上面这个演示是缓冲区溢出最简单也是最核心的溢出本质的演示，需要仔细的理解。如果还不太清楚的话可以结合对应的汇编代码理解。
用 VC 的命令行编译器编译的时候指定 FA 参数可以获得对应的汇编代码（Linux 平台可以用 gcc 的-S 参数获得）：

```
C:\Temp>cl /FA tex.c
C:\Temp>type tex.asm
        TITLE   tex.c
        .386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS    ENDS
$$SYMBOLS       SEGMENT BYTE USE32 'DEBSYM'
$$SYMBOLS       ENDS
_TLS    SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS    ENDS
FLAT    GROUP _DATA, CONST, _BSS
        ASSUME  CS: FLAT, DS: FLAT, SS: FLAT
endif

INCLUDELIB LIBC
INCLUDELIB OLDNAMES

_DATA   SEGMENT
$SG775  DB      'why u here ?!', 0aH, 00H
_DATA   ENDS
PUBLIC  _why_here
EXTRN   _printf:NEAR
EXTRN   __exit:NEAR
```

```
_TEXT   SEGMENT
_why_here PROC NEAR
        push    ebp
        mov     ebp, esp
        push    OFFSET FLAT:$SG775
        call    _printf
        add     esp, 4
        push    0
        call    __exit
        add     esp, 4
        pop     ebp
        ret     0
_why_here ENDP
_TEXT   ENDS

PUBLIC  _main
_TEXT   SEGMENT
_buff$ = -4                                             ; size = 4
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC NEAR
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _buff$[ebp+8], OFFSET FLAT:_why_here
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
END
```

实施对缓冲区溢出的利用（即攻击有此问题的程序）需要更多尚未涉及的主题：溢出地址定位、shellcode 存放和地址定位、shellcode功能等。

SHELLCODE 基础

溢出发生后要控制溢出后的行为关键就在于 shellcode 的功能。shellcode 其实就是一段机器码。因为我们平时顶多用汇编写程序，绝对不会直接用机器码编写程序，所以感觉 shellcode 非常神秘。这里让我们来揭开其神秘面纱。
看看程序 shell0.c：

```
#include<stdio.h>
int add(int x,int y) {
```

```
        return x+y;
}
int main(void) {
        result=add(129,127);
        printf("result=%i\n",result);
        return 0;
}
```

这个程序太简单了！那么我们来看看这个程序呢？shell1.c

```
#include<stdio.h>
int add(int x,int y)  {
        return x+y;
}
int main(void)  {
        unsigned char buff[256];
        unsigned char * ps   = (unsigned char *) &add;
        unsigned char * pd   = buff;
        int (* pf) (int,int) = buff; /*函数指针 */
        int result=0;
        printf("shell:");
        while(1)
        {
                *pd = * ps;
                printf("\\x%02x",*ps);
                if(*ps == 0xc3)  /* ret指令对应的机器码值 */
                {
                        break;
                }
                pd++, ps++;
        }
        result=pf(129,127);
        printf("\nresult=%i\n",result);
        return 0;
}
```

编译出来运行，结果如下：

```
shell:\x55\x89\xe5\x8b\x45\x0c\x03\x45\x08\x5d\xc3
result=25
```

shell1和shell0的不同之处在于shell1将add函数对应的机器码从代码空间拷贝到了buff中（拷贝过程中顺便把他们打印出来了），然后通过函数指针运行了buff中的代码！

关键代码解释：

```
unsigned char * ps   = (unsigned char *) &add;
```

&add 为函数在代码空间中开始地址，上面语句让ps指向了 add函数的起始地址。

```
int (* pf) (int,int) = buff;
```

让pf指向 buff，以后调用pf时将会指向buff中的代码。

```
*pd = * ps;
```

把机器码从 add函数开始的地方拷贝到 buff数组中。

```
if(*ps == 0xc3) { break }
```

每个函数翻译为汇编指令后都是以ret指令结束,对应的机器码为 0xc3，这个判断控制拷贝到函数结尾时停止拷贝，退出循环。

```
result=pf(129,127);
```

由于 pf指向 buff，这里调用 pf后将把 buff中的数据作为代码执行。

shell1和 shell0做的事情一样，但机制就差别很大了。值得注意的是 shell1的输出中这一行:

```
shell:\x55\x89\xe5\x8b\x45\x0c\x03\x45\x08\x5d\xc3
```

直接以C语言表示字符串的形式将平时深藏不露的机器码给打印了出来。其对应的C语言代码是:

```
      int add(int x,int y)  {
        return x+y;
    }
```

对应的汇编码(AT&T的表示）为:

```
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    popl    %ebp
    ret
```

接下来理解这个程序应该就很容易了 shell2.c:

```
#include<stdio.h>
int main(void)
{
    unsigned char buff[]="\x55\x89\xe5\x8b\x45\x0c\x03\x45\x08\x5d\xc3";
    int (* pf) (int,int) = buff;
    int result=0;
    result=pf(129,127);
    printf("result=%i\n",result);
    return 0;
}
```

我们直接把 add函数对应的机器码写到 buff数组中，然后直接从 buff中运行 add功能。编译运行结果为:

```
result=256
```

本质上来看上面的"\x55\x89\xe5\x8b\x45\x0c\x03\x45\x08\x5d\xc3"就是一段shellcode。shellcode的名称来源和 Unix的 Shell有些关系，早期攻击程序中 shellcode的功能是开启一个新的 shell，也就是说溢出攻击里 shellcode的功能远远不像我们演示中这么简单，需要完成更多的功能，这需要解决很多这里没有遇到的问题：函数重定位、系统调用接口、自身优化、不能包含某些特别字符等等。