编号：_____

实验	一	二	三	四	五	六	七	八	总评	教师签名
成绩										

武汉大学国家网络安全学院

课程实验（设计）报告

题　　目: 格式化字符串漏洞分析

专业 (班) : 2022 级信安 5 班

学　　号: 2022302181145

姓　　名: 周 业 营

课程名称: 软 件 安 全 实 验

任课教师: 赵　磊

2024 年 12 月 25 日

目 录

1 实验名称

格式化字符串漏洞

2 实验环境

- Windows Subsystem for Linux 2 (WSL 2)
- Ubuntu 20.04
- Visual Studio Code (VSCode)
- GNU gdb (Ubuntu 9.2-0ubuntu1 20.04.2) 9.2
- Pwndbg

3 实验关键过程、数据及其分析

3.1 格式化字符串漏洞分析

格式化字符串漏洞是一种内存安全漏洞，可以被攻击者利用实现任意地址读和任意地址写，从而泄露敏感信息或执行任意代码。该漏洞源于 C 语言的格式化输出函数的灵活性——这些函数的参数数量和类型是可变的，具体由格式化字符串中的占位符（如%d, %x, %n 等）决定。然而，由于这些函数不会主动检查格式化字符串与提供参数之间的一致性，当攻击者精心构造的输入不符合规范时，就会引发漏洞。

在正常情况下，printf 函数会按照以下逻辑执行：

1. 解析格式化字符串：扫描格式化字符串并识别其中的占位符（如%d, %x）。
2. 从栈中取参数：根据占位符的数量，从栈上依次读取参数，用以匹配占位符。

这里举个具体的例子来看一下 printf 的实现机制。

```c
#include<stdio.h>
void main() {
    printf("%s %d %s", "Hello World!", 233, "\n");
}
```

dump 来查看一下编译成汇编语言之后的实现机制

```
Dump of assembler code for function main:
   0x0000053d <+0>:     lea    ecx,[esp+0x4]
   0x00000541 <+4>:     and    esp,0xfffffff0
   0x00000544 <+7>:     push   DWORD PTR [ecx-0x4]
   0x00000547 <+10>:    push   ebp
   0x00000548 <+11>:    mov    ebp,esp
   0x0000054a <+13>:    push   ebx
   0x0000054b <+14>:    push   ecx
   0x0000054c <+15>:    call   0x585 <__x86.get_pc_thunk.ax>
   0x00000551 <+20>:    add    eax,0x1aaf
   0x00000556 <+25>:    lea    edx,[eax-0x19f0]
```

```
   0x0000055c <+31>:    push   edx
   0x0000055d <+32>:    push   0xe9
   0x00000562 <+37>:    lea    edx,[eax-0x19ee]
   0x00000568 <+43>:    push   edx
   0x00000569 <+44>:    lea    edx,[eax-0x19e1]
   0x0000056f <+50>:    push   edx
   0x00000570 <+51>:    mov    ebx,eax
   0x00000572 <+53>:    call   0x3d0 <printf@plt>
   0x00000577 <+58>:    add    esp,0x10
   0x0000057a <+61>:    nop
   0x0000057b <+62>:    lea    esp,[ebp-0x8]
   0x0000057e <+65>:    pop    ecx
   0x0000057f <+66>:    pop    ebx
   0x00000580 <+67>:    pop    ebp
   0x00000581 <+68>:    lea    esp,[ecx-0x4]
   0x00000584 <+71>:    ret
End of assembler dump.
```

根据 cdecl 的调用约定，在进入 printf() 函数之前，将参数从右到左依次压栈。进入 printf() 之后，函数首先获取第一个参数，一次读取一个字符。如果字符不是%，字符直接复制到输出中。否则，读取下一个非空字符，获取相应的参数并解析输出。

但是，这种逻辑并没有验证调用者是否提供了足够的参数。当攻击者提供的格式化字符串包含多余的占位符，而实际传递的参数不足时，函数依然会从栈中继续读取数据。这样，攻击者可以通过精心设计的格式化字符串，访问甚至操纵函数调用栈上的数据。

3.1.1 任意地址读

格式化字符串漏洞最直接的危害是任意地址读，即通过格式化字符串泄露程序栈上的数据。

正常情况下的行为：当格式化字符串为”Hello %s” 且提供一个参数（如”World”）时，printf 会从栈上正确读取参数并输出。漏洞触发：当调用 printf(buf)，而 buf 的值为”%x %x” 时，程序没有提供相应的参数，但 printf 会从栈上继续读取数据以填充%x，结果输出了调用栈上相邻地址的数据。这种行为可以泄露栈中的敏感信息。

除此之外 C 语言支持格式化字符串的参数偏移指定（如%n$x），这使攻击者可以精确控制从栈上的哪个位置读取数据。具体来说，%n$x 表示读取相对于格式化字符串的第 n 个参数，从而实现任意位置的内存数据泄露。

这样通过多次试探，攻击者可以逐渐掌握栈中的内容，最终掌握程序的运行状态。

3.1.2 任意地址写

除了任意地址读，格式化字符串漏洞还可以通过%n 占位符实现任意地址写。%n 转换指示符的操作是将当前已经成功写入流或缓冲区中的字符个数存储到地址由参数指定的整数中

printf(”Hello%n”, &x); 会将”Hello” 的长度（5 个字符）写入变量 x 指向的地址。

如果攻击者精心构造格式化字符串并提供一个可控的指针时，可以将任意值写入任意内存地址。这种任意地址写能力可以被用于修改程序的控制流，从而覆盖返回地址或函数指针，实现更进一步的攻击。

3.2 栈桢布局和关键变量布局

根据 vul.c 程序分析栈帧布局。

```
#include <stdio.h>

void main(int argc, char ** argv)
{
   char buff[36] = {0};
   int a = 0;
   int b = 0;
   FILE *fp;
   char ch;

   if( (fp = fopen(argv[1], "r")) != NULL )
    fread(buff, 32, 1, fp);

   printf("the addresses of a b buf are %x,%x,%x\n", &a, &b, buff);

   a = 0;
   printf(buff);
   printf("a= %d\n", a);
   if(fp)
     fclose(fp);
}
```

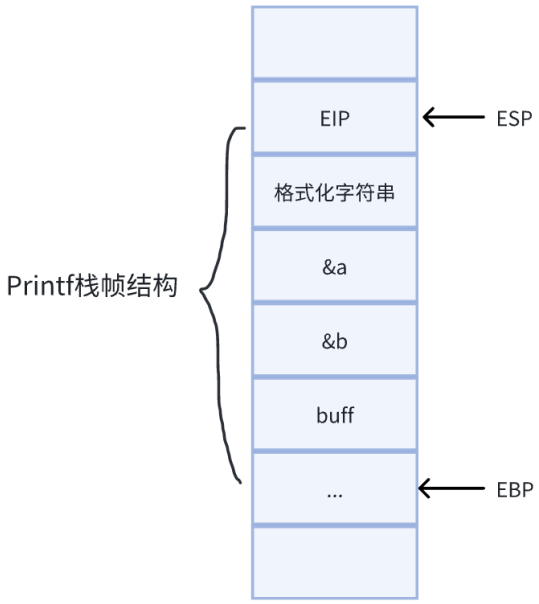
使用 pwndbg 查看变量的内存地址。

```
   0x40122c <main+150>    mov    rsi, rax              RSI => 0x7fffffffd68c ◂— 0
   0x40122f <main+153>    lea    rdi, [rip + 0xdda]    RDI => 0x402010 ◂— 'the addresses of a b buf are %x,%x,%x\n'
   0x401236 <main+160>    mov    eax, 0                EAX => 0
   0x40123b <main+165>    call   printf@plt                  <printf@plt>

   0x401240 <main+170>    mov    dword ptr [rbp - 0x34], 0
   0x401247 <main+177>    lea    rax, [rbp - 0x30]
   0x40124b <main+181>    mov    rdi, rax
   0x40124e <main+184>    mov    eax, 0                      EAX => 0
[ SOURCE (CODE) ]
In file: /home/elephant/code/printf/vul.c:14
    9    char ch;
   10
   11    if( (fp = fopen(argv[1], "r")) != NULL )
   12        fread(buff, 32, 1, fp);
   13
 ► 14    printf("the addresses of a b buf are %x,%x,%x\n", &a, &b, buff);
   15
   16    a = 0;
   17    printf(buff);
   18    printf("a= %d\n", a);
   19    if(fp)
[ STACK ]
00:0000| rsp 0x7fffffffd670 —▸ 0x7fffffffd7b8 —▸ 0x7fffffffda99 ◂— '/home/elephant/code/printf/vul.o'
01:0008|-048 0x7fffffffd678 ◂— 0x1ffffda89
02:0010|-040 0x7fffffffd680 —▸ 0x7ffff7fd15e0 (dl_main) ◂— endbr64
03:0018|-038 0x7fffffffd688 ◂— 0
... ↓        4 skipped
[ BACKTRACE ]
 ► 0         0x401220 main+138
   1   0x7ffff7ded083 __libc_start_main+243

pwndbg> stack 20
00:0000| rsp 0x7fffffffd670 —▸ 0x7fffffffd7b8 —▸ 0x7fffffffda99 ◂— '/home/elephant/code/printf/vul.o'
01:0008|-048 0x7fffffffd678 ◂— 0x1ffffda89
02:0010|-040 0x7fffffffd680 —▸ 0x7ffff7fd15e0 (dl_main) ◂— endbr64
03:0018|-038 0x7fffffffd688 ◂— 0
... ↓        4 skipped
08:0040|-010 0x7fffffffd6b0 ◂— 0x7fff00000000
09:0048|-008 0x7fffffffd6b8 ◂— 0
0a:0050| rbp 0x7fffffffd6c0 ◂— 0
0b:0058|+008 0x7fffffffd6c8 —▸ 0x7ffff7ded083 (__libc_start_main+243) ◂— mov edi, eax
0c:0060|+010 0x7fffffffd6d0 ◂— 0x100000044 /* 'D' */
0d:0068|+018 0x7fffffffd6d8 —▸ 0x7fffffffd7b8 —▸ 0x7fffffffda99 ◂— '/home/elephant/code/printf/vul.o'
0e:0070|+020 0x7fffffffd6e0 ◂— 0x1f7fb17a0
0f:0078|+028 0x7fffffffd6e8 —▸ 0x401196 (main) ◂— endbr64
10:0080|+030 0x7fffffffd6f0 —▸ 0x401290 (__libc_csu_init) ◂— endbr64
11:0088|+038 0x7fffffffd6f8 ◂— 0xb09fb7599f84ccc1
12:0090|+040 0x7fffffffd700 —▸ 0x4010b0 (_start) ◂— endbr64
13:0098|+048 0x7fffffffd708 —▸ 0x7fffffffd7b0 ◂— 1
pwndbg> print &a
$2 = (int *) 0x7fffffffd68c
pwndbg> print &b
$3 = (int *) 0x7fffffffd688
pwndbg> print &buf
$4 = (char **) 0x7ffff7fb67c0 <buf>
pwndbg>
```

结合 pwndbg 中看到的栈帧结构，可以绘制出 printf 函数的栈帧结构示意图。



3.3 构造的文件内容

构造的文件内容为：

%2024x%1$n

。

文件的内容包括两个部分，第一部分是**%2024x**。%x 表示十六进制整数输出格式，%x 前再添加 2024 的参数意味着制定输出宽度为 2024 的十六进制值字符，如果十六进制值的字符数宽度小于 2024，则会在输出字符的前面用空格填充以达到 2024 的宽度。总的来说，%2024x 会尝试读取一个栈上的数据，然后以 2024 的宽度输出出来。

第二部分是 **%1$n** 的格式说明符号。在格式化字符串中，%n 是一个特殊的格式说明符，它不会直接输出内容，而是将到目前为止已经输出的字符数写入对应的参数所指向的内存地址中。而在其前面加上 1$ 的参数则是指定写入的偏移地址 1 指向的地址。

3.4 漏洞触发后的布局

使用 pwndbg 查看 printf 传参情况以及参数的值。

3.5 变量 a 的值重写过程

定义变量 buff，给 buff 开辟内存空间。

$$char \quad buff[36] = 0;$$

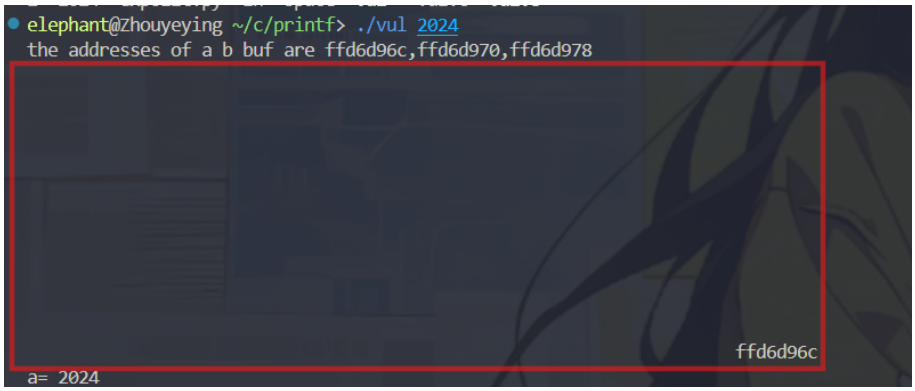
将文件内容 $\%2024x\%1\$n$ 读入 buff 中。

$$fread(buff,\ 32,\ 1,\ fp);$$

然后使用 printf 输出 buff 的内容。

$$printf(buff);$$

首先将输出的格式化字符串之后的第一个参数，将其作为地址，以%x 的格式将这个地址的数据读取出来，也就是输出中的一大串空格之后跟着一个变量 a 的地址，如果说仔细一点的话，这个空格加上 a 的地址 ffd6d96c，一共占据了 2024 个字符的长度。



然后执行 %1\$$n$ 的内容，这里 %1 指定了将格式化字符串之后的第一个参数作为操作的变量地址，结合上面输出的 %2024x，可以发现这个地址就是变量 a 的地址。在这之后，执行 \$$n$ 的占位符运行逻辑，将到目前为止已经输出的字符数写入对应的参数所指向的内存地址中，也就是将 2024 写入指定的变量 a 的地址 ffd6d96c 中。这样就完成了利用格式化字符串漏洞对变量 a 的值进行修改的操作。

4 实验体会和扩展思考

4.1 实验心得与体会

这次的实验主要围绕格式化字符串漏洞的利用展开，通过解读赵磊老师给出的格式化字符串漏洞利用 demo 让我深刻理解了该漏洞的原理、成因及其危害。在实验中，程序中的漏洞点位于 printf(buff)，即用户输入的字符串直接作为格式化字符串传入 printf 函数，而没有进行必要的验证，从而导致了潜在的漏洞利用可能性。这种漏洞的核心原因在于，printf 函数依赖格式化字符串中的占位符决定参数的读取，而如果用户输入了恶意格式化字符串，程序可能访问栈上意料之外的内容，甚至导致代码执行的改变。

实验中可以通过在输入中使用%x 占位符，读取程序栈上的内容。输入%x %x %x，可以连续读取栈上的多个地址值。这种信息泄漏为进一步的攻击提供了关键数据，最经典的就是找到函数的返回地址，然后结合任意地址写来修改函数的返回地址，从而达到截取程序控制流，跳转到 shellcode 执行的目的。

实验还尝试通过%n 占位符将字符数写入指定地址。由于变量 a 在内存中的地址是已知的，因此通过构造恶意输入，可以将任意值写入 a 的地址。这种任意地址写入功能使攻击者可以修改程序的关键变量或控制流。

实验中发现，格式化字符串漏洞的利用依赖于精确的地址信息。如果攻击者能够泄露程序的内存布局信息（如变量地址），便可通过该漏洞进一步实现任意代码执行。此外，利用的效果也与程序的编译环境密切相关，也不是编译器的安全防护机制，包括栈布局和内存保护机制等。

通过本次实验，我不仅深入理解了格式化字符串漏洞的原理，还掌握了其常见的利用方法。尤其是在实验中观察到，通过构造恶意输入实现对栈数据的读取和写入，直观地感受到漏洞的危害性。这种漏洞的成因在于开发者对输入字符串的缺乏验证和不当使用格式化函数。在实际的应用中，应该要严格检查和验证用户的输入，来避免直接将用户数据传入格式化函数。

这次实验还让我认识到，现代安全防护机制对漏洞利用有显著的影响，比如说我在使用本地的环境去编译和执行脚本程序时，格式化字符串漏洞的利用就会被编译器的安全机制给保护，无论怎么控制输入的格式化字符串都无法修改变量 a 的值。如果没有这些安全防护措施，格式化字符串漏洞的利用将变得十分容易。实验中对漏洞的手动利用让我进一步体会到，漏洞挖掘与利用需要攻防视角的结合，这为我在未来的安全学习中提供了新的学习方向和视角。

5 教师评语评分

评语:

评分：_____

评阅人：_____

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）