CTF 实验 2 - 格式化字符串漏洞

学号：2354100 姓名：郝哲逸

---

1. 实验概述

本实验主要学习 C 语言中格式化输出函数（printf）的使用，以及如何利用格式化字符串漏洞进行攻击。通过实验，我们深入理解了格式化字符串漏洞的产生原理、利用方法以及如何防御此类漏洞。

---

2. 实验内容

2.1 格式化输出函数基础

格式化输出函数是 C 语言中用于格式化输出的标准函数，其基本语法为：

```
%[parameter][flags][width][.precision][length]type
```

参数说明：

- `parameter`：指定参数位置（POSIX 扩展），如 `%2$d` 表示第 2 个参数
- `flags`：调整输出格式，如 `-` 左对齐、`+` 显示符号、`#` 显示前缀等
- `width`：指定输出字符的最小个数
- `.precision`：指定精度，如小数位数、有效数字个数
- `length`：指定参数的大小，如 `h`（短整型）、`l`（长整型）
- `type`：转换类型，如 `d`（十进制）、`x`（十六进制）、`s`（字符串）等

实验目的： 掌握 printf 函数的各种格式化功能，理解参数传递机制。

---

3. 格式化字符串漏洞原理

3.1 基本原理

在 x86 结构下，格式字符串的参数是通过栈传递的。根据函数调用约定，程序会将参数从右到左依次压栈。如果实际提供的参数少于格式化字符串预期的个数，多余的控

制字符就会把栈数据当做参数打印出来。

漏洞发生条件： 格式字符串要求的参数和实际提供的参数不匹配。

3.2 攻击方式

1. **使程序崩溃**：通过 `%s` 转换指示符访问无效地址导致 SIGSEGV
2. **栈数据泄露**：使用 `%x` 或 `%p` 转换指示符读取栈上的数据
3. **任意地址内存泄露**：使用 `%s` 转换指示符读取指定地址的内存内容
4. **栈数据覆盖**：使用 `%n` 转换指示符向指定地址写入数据

---

4. 详细实验过程与分析

4.1 实验一：格式化输出函数基础

实验代码：

```
#include <stdio.h>
#include <string.h>

int main(){
    printf("Hello %%\n");
    printf("Hello World!\n");
    printf("Number: %d\n", 123);
    printf("%s %s\n", "Format", "Strings");
    printf("%12c\n", 'A');
    printf("%16s\n", "Hello");
    int n;
    printf("%12c%n\n", 'A', &n);
    printf("%16s%n\n", "Hello!", &n);
    printf("%2$s %1$s\n", "Format", "Strings");
    return 0;
}
```

编译与运行：

```
gcc -m32 printf.c -o printf -g
./printf
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 printf.c -o printf -g
printf.c: In function ‘main’:
printf.c:15:16: warning: format ‘%c’ expects argument of type ‘int’, but argument 2 has type ‘int *’ [-Wformat=]
   15 |     printf("%42c%1$n\n", &n);
      |                |      |
      |                int    int *
      |              %42n
printf.c:15:5: warning: ‘$’ operand number used after format without operand number [-Wformat=]
   15 |     printf("%42c%1$n\n", &n);
      |
soldier@SOLDIER:~/InformationSecurity/hw2$ ./printf
Hello %
Hello World!
Number: 123
Format Strings
           A
           Hello
           A
          Hello!
Strings Format
```

实验分析：

通过运行结果可以看到 printf 函数的各种格式化功能：

- `%%` 输出字面量 `%` 字符
- `%d` 输出十进制整数
- `%s` 输出字符串
- `%12c` 输出字符，宽度为 12，右对齐
- `%16s` 输出字符串，宽度为 16，右对齐
- `%n` 将已写入的字符数存储到指定地址

关键发现： `%n` 转换指示符可以将格式化字符串中已成功写入的字符个数写入到内存地址中，这是实现栈数据覆盖的关键机制。

4.2 实验二：格式化字符串漏洞演示

4.2.1 基本漏洞演示

实验代码：

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

编译与运行：

```
gcc -m32 fmtdemo.c -o fmtdemo -g
./fmtdemo
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 fmtdemo.c -o fmtdemo -g
mosoldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
Hello World! 233
```

实验分析：
程序正常输出三个参数的值，格式化字符串与参数数量匹配，没有漏洞。

4.2.2 修改程序触发漏洞

修改后的代码：

```
#include <stdio.h>
int main(){
    printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
}
```

编译与运行：

```
gcc -m32 fmtdemo.c -o fmtdemo -g
./fmtdemo
```

```
soldier@SOLDIER: /InformationSecurity/hw2$ gdb -q fmtdemo
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 15.1 in 0.00ms using Python engine 3.12
Reading symbols from fmtdemo...
gef➤
```

实验分析：
程序打印了七个值（包括换行），而参数只有三个。后面三个 `%d` 打印的是栈后续的非空字符，而最后一个参数 `%3$s` 则是对第三个参数 `\n` 的重用。

关键发现： 当格式化字符串要求的参数多于实际提供的参数时，多余的控制字符会从栈中读取数据，导致格式化字符串漏洞。这验证了漏洞发生条件：格式字符串要求的参数和实际提供的参数不匹配。

4.2.3 使用 GEF 反汇编分析

操作命令：

```
(gdb) disassemble main
```

```
soldier@SOLDIER: ~/InformationSecurity/hw2
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000118d <+0>:     lea    ecx,[esp+0x4]
   0x00001191 <+4>:     and    esp,0xfffffff0
   0x00001194 <+7>:     push   DWORD PTR [ecx-0x4]
   0x00001197 <+10>:    push   ebp
   0x00001198 <+11>:    mov    ebp,esp
   0x0000119a <+13>:    push   ebx
   0x0000119b <+14>:    push   ecx
   0x0000119c <+15>:    call   0x11d9 <__x86.get_pc_thunk.ax>
   0x000011a1 <+20>:    add    eax,0x2e37
   0x000011a6 <+25>:    lea    edx,[eax-0x1fd0]
   0x000011ac <+31>:    push   edx
   0x000011ad <+32>:    push   0xe9
   0x000011b2 <+37>:    lea    edx,[eax-0x1fce]
   0x000011b8 <+43>:    push   edx
   0x000011b9 <+44>:    lea    edx,[eax-0x1fc1]
   0x000011bf <+50>:    push   edx
   0x000011c0 <+51>:    mov    ebx,eax
   0x000011c2 <+53>:    call   0x1040 <printf@plt>
   0x000011c7 <+58>:    add    esp,0x10
   0x000011ca <+61>:    mov    eax,0x0
   0x000011cf <+66>:    lea    esp,[ebp-0x8]
   0x000011d2 <+69>:    pop    ecx
   0x000011d3 <+70>:    pop    ebx
   0x000011d4 <+71>:    pop    ebp
   0x000011d5 <+72>:    lea    esp,[ecx-0x4]
   0x000011d8 <+75>:    ret
End of assembler dump.
gef➤
```

实验分析：

通过反汇编代码可以看到 main 函数的汇编指令，包括参数压栈、函数调用等过程。从反汇编代码来看没有任何区别，所以需要重点关注其中的参数传递过程。

关键发现： printf 函数的参数是通过栈传递的，根据 cdecl 调用约定，参数从右到左依次压栈。进入 printf 函数后，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 `%` 则直接复制到输出，否则读取下一个非空字符，获取相应参数并解析输出。

4.3 实验三：外部输入触发漏洞

实验代码：

```
#include <stdio.h>
void main(){
    char buf[50];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
```

```
        printf(buf);
    }
```

编译与运行：

```
gcc -m32 fmtdemo.c -o fmtdemo -g
./fmtdemo
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 fmtdemo.c -o fmtdemo -g
fmtdemo.c: In function ‘main’:
fmtdemo.c:3:23: warning: format ‘%x’ expects a matching ‘unsigned int’ argument [-Wformat=]
    3 |     printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
      |                       ~~
      |                        |
      |                        unsigned int
fmtdemo.c:3:26: warning: format ‘%x’ expects a matching ‘unsigned int’ argument [-Wformat=]
    3 |     printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
      |                          ~~
      |                           |
      |                           unsigned int
fmtdemo.c:3:29: warning: format ‘%x’ expects a matching ‘unsigned int’ argument [-Wformat=]
    3 |     printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
      |                             ~~
      |                              |
      |                              unsigned int
fmtdemo.c:3:5: warning: ‘$’ operand number used after format without operand number [-Wformat=]
    3 |     printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
      |     ~~~~~~
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
Hello World! 233
 ffc9bb00 f7effe34 0
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 fmtdemo.c -o fmtdemo -g
mofmtdemo.c: In function ‘main’:
fmtdemo.c:6:12: warning: format not a string literal and no format arguments [-Wformat-security]
    6 |     printf(buf);
      |            ~~~
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
Hello %x %x %x !
Hello 32 f7ee85c0 565f41c4 !
```

实验分析：
在 buf 中输入转换指示符，printf 会把它当成格式字符串进行解析，漏洞由此发生。程序打印出栈上的十六进制值，泄露了栈数据。

关键发现： 当格式字符串由外部输入提供时，很容易导致漏洞产生。例如输入 "Hello %x %x %x !\n"（其中 `\n` 是 fgets() 自动添加的），程序就把栈数据泄露出来。由此可以总结出，格式字符串漏洞发生条件就是格式字符串要求的参数和实际提供的参数不匹配。

4.4 实验四：使程序崩溃

实验代码：

```
#include <stdio.h>
void main(){
    char buf[50];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

编译与运行：

```
gcc -m32 fmtdemo.c -o fmtdemo -g
./fmtdemo
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 -fno-stack-protector -no-pie fmtdemo.c -o fmtdemo
fmtdemo.c: In function ‘main’:
fmtdemo.c:6:12: warning: format not a string literal and no format arguments [-Wformat-security]
    6 |     printf(buf);
      |            ~~~
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
%x.%x.%x.%x.%x
32.f7eed5c0.804918d.f7f3dfe8.18
```

4.5 实验五：栈数据泄露

实验代码：

```
#include <stdio.h>
int main(){
    char format[128];
    int v1 = 1, v2 = 0x88888888, v3 = -1;
    char v4[10] = "ABCD";
    scanf("%s", format);
    printf(format, v1, v2, v3, v4);
    printf("\n");
}
```

编译与运行：

```
gcc -m32 -fno-stack-protector -no-pie fmtdemo.c -o fmtdemo
./fmtdemo
```

输入：

```
%08x.%08x.%08x.%08x.%08x
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 -fno-stack-protector -no-pie fmtdemo.c -o fmtdemo
fmtdemo.c: In function ‘main’:
fmtdemo.c:6:12: warning: format not a string literal and no format arguments [-Wformat-security]
    6 |     printf(buf);
      |            ^~~
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
%x.%x.%x.%x.%x
32.f7eed5c0.804918d.f7f3dfe8.18
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 -fno-stack-protector -no-pie fmtdemo.c -o fmtdemo
demosoldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
%08x.%08x.%08x.%08x.%08x
00000001.88888888.ffffffff.fff7987a.fff79884
```

实验分析：
格式化输出函数使用一个内部变量来标志下一个参数的位置。开始时，参数指针指向第一个参数 arg1，随着每一个参数被相应的格式规范使用，参数指针也根据参数的长度不断递增。在打印完当前函数的剩余参数之后，printf 就会打印当前函数的栈帧。

关键发现： 通过使用 `%x` 或 `%p` 转换指示符，可以读取栈上的数据。由于 x86 的栈由高地址向低地址增长，同时 printf 函数的参数是以逆序被压入栈的，所以参数在内存中出现的顺序与在 printf 调用时出现的顺序是一致的。

4.6 实验六：指定位置读取

继续使用 fmtdemo.c，输入：

```
%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p
ffffffff.00000001.0x88888888.0x88888888.0xfffad06a.0xfffad074.(nil)
soldier@SOLDIER:~/InformationSecurity/hw2$
```

实验分析：
这类格式字符串可以直接泄露指定的某个数据。比如 `%3$x` 表示位于格式字符串后的第 3 个数据。

关键发现： 通过使用参数位置指定（如 `%1$`、`%2$`、`%3$` 等），可以精确控制读取哪个参数的值。这对于精确攻击非常有用，可以避免读取不需要的数据。

4.7 实验七：任意地址读取

继续使用 fmtdemo.c，输入：

```
%4$s
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
%4$s
ABCD
soldier@SOLDIER:~/InformationSecurity/hw2$
```

实验分析：
此时输出的 arg4 就变成了字符串 "ABCD" 而不再是地址。`%s` 可以泄露参数所指向的内存数据，程序会将它作为一个 ASCII 字符串处理，直到遇到一个空字符。所以操控该指针就可以泄露任意地址内容。

关键发现： 通过控制参数的值，可以读取任意地址的内存内容。例如，如果将某个参数设置为指向目标地址的指针，就可以使用 `%s` 读取该地址的内容。

4.8 实验八：任意地址内存泄露

继续使用 fmtdemo.c，输入：

```
AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ ./fmtdemo
AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
AAAA.0x1.0x88888888.0xffffffff.0xffb4c7ca.0xffb4c7d4.(nil).0x80491a0.(nil).(nil).0x42411000.0x4443.(nil).0x41414141.
0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e
soldier@SOLDIER:~/InformationSecurity/hw2$
```

实验分析：
对于打印的第 13 个字符，只要使用 `%13$s` 就可以读出该地址所指向的内存。通过观察输出，可以发现第 13 个字符是 'A'，这表明栈上某个位置存储了指向 "AAAA" 的指针。

关键发现： 通过读取多个参数的值，可以定位到目标地址。一旦知道了目标地址在栈上的位置，就可以使用 `%N$s` 读取该地址的内容。这种方法在 CTF 中常用于泄露 libc 地址或其他敏感信息。

4.9 实验九：栈数据覆盖 - %n 基础

实验代码：

```
#include <stdio.h>
int main(){
    int i;
    char str[] = "hello";
    printf("%s %n\n", str, &i);
    printf("%d\n", i);
}
```

编译与运行：

```
gcc -m32 a.c -o a
./a
```

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 a.c -o a
soldier@SOLDIER:~/InformationSecurity/hw2$ ./a
hello
6
```

实验分析：
在上述程序中 i 被赋值为 6，因为在遇到转换指示符之前一共写入了 6 个字符（"hello" 加上一个空格）。在没有长度修饰符时，默认写入一个 int 类型的值。

关键发现： `%n` 转换指示符将当前已经成功写入流或缓冲区中的字符个数存储到由参数指定的整数中。这是实现栈数据覆盖的关键机制。

4.10 实验十：控制写入字符数

实验代码：

```
#include <stdio.h>
int main(){
    int i;
    printf("%10u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%.50u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%0100u%n\n", 1, &i);
    printf("%d\n", i);
}
```

编译与运行：

```
gcc -m32 a.c -o a
./a
```

[redacted]

```
soldier@SOLDIER:~/InformationSecurity/hw2$ gcc -m32 a.c -o a
soldier@SOLDIER:~/InformationSecurity/hw2$ ./a
         1

10
0000000000000000000000000000000000000000000000001

50
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001

100
```

实验分析：

- `%10u%n`：至少输出 10 位数字，所以写入 10 个字符
- `%.50u%n`：输出 50 位数字，所以写入 50 个字符
- `%0100u%n`：输出 100 位数字，所以写入 100 个字符

关键发现： 在漏洞利用时，需要覆写的值是一个 shellcode 的地址，而这个地址往往是一个很大的数字。这时就需要使用具体的宽度或精度转换规范来控制写入的字符数，即在格式字符串中加上一个十进制整数来表示输出的最小位数，如果实际位数大于定义的宽度，则按实际位数输出，反之则以空格或 0 补齐。

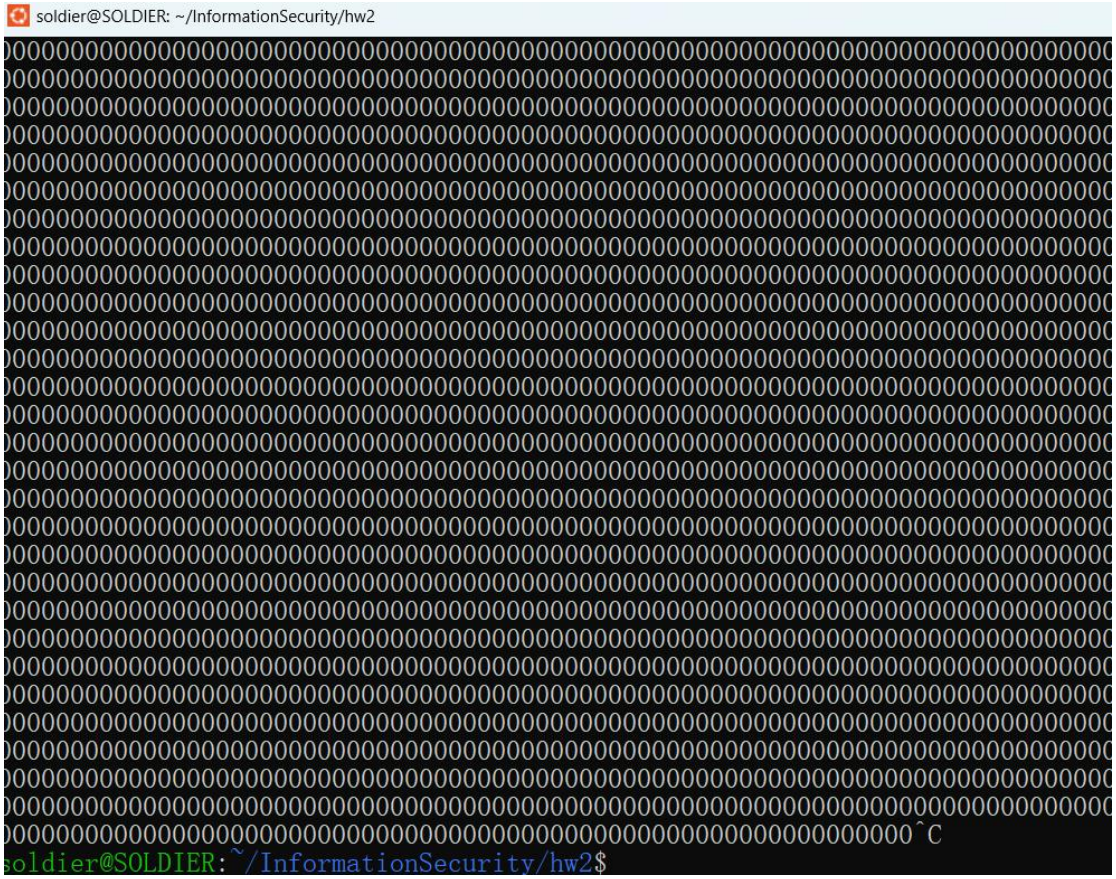
4.11 实验十一：大数值写入

实验代码：

```
#include <stdio.h>
int main(){
    int i;
    printf("%0134512640d%n\n",1, &i);
    printf("0x%x\n", i);
}
```

编译与运行：

```
gcc -m32 a.c -o a
./a
```



实验分析：
输出 `0x134512640`。在漏洞利用时，这个数字通常是 shellcode 的地址，而 shellcode 的地址往往是一个很大的数字。

关键发现： 通过控制写入的字符数，可以向指定地址写入任意数值。这是实现栈数据覆盖、劫持程序执行流的关键技术。在实际的 CTF 攻击中，通常会使用这种方法将 shellcode 的地址写入函数返回地址，从而劫持程序执行流。

---

5. 实验总结与心得

5.1 关键知识点总结

通过本次实验，我深入理解了：

1. **C 语言格式化输出函数的工作原理**
    - printf 函数是变参函数，由固定数量的强制参数和数量可变的可选参数组成
    - 格式化字符串由普通字符和转换规则构成
    - 参数通过栈传递，根据 cdecl 调用约定从右到左依次压栈

2. **格式化字符串漏洞的产生条件和利用方法**
    - 漏洞发生条件：格式字符串要求的参数和实际提供的参数不匹配
    - 攻击方式：使程序崩溃、栈数据泄露、任意地址内存泄露、栈数据覆盖
    - %n 转换指示符是实现栈数据覆盖的关键
3. **栈内存的布局和参数传递机制**
    - x86 栈从高地址向低地址增长
    - printf 参数是逆序压栈的，因此在内存中出现的顺序与调用时的顺序一致
    - 参数指针机制：printf 使用内部变量标志下一个参数的位置
4. **如何利用格式化函数进行内存操作**
    - 使用 %x/%p 读取栈上的十六进制值
    - 使用 %Nx/p 读取指定位置的参数
    - 使用 %N$s 读取指定地址的内存内容
    - 使用 %N$n 向指定地址写入数据

5.2 实验心得

通过本次实验，我深刻理解了格式化字符串漏洞的危害性和利用方法。在 CTF 竞赛中，格式化字符串漏洞是一种常见的漏洞类型，掌握其利用方法对于解决相关题目非常重要。

实验过程中，我学会了如何：

- 使用 GEF 工具进行反汇编和调试
- 通过格式化字符串泄露栈上的敏感数据
- 利用 %n 指示符实现栈数据覆盖
- 控制写入的字符数来实现精确的数据覆盖

同时，我也认识到格式化字符串漏洞的危害性。在实际开发中，应该：

- 避免使用用户输入作为格式字符串
- 使用 snprintf 或 fprintf 替代 printf
- 对格式字符串进行严格验证
- 使用编译器保护机制
- 避免使用 -fno-stack-protector 编译选项

5.3 实验不足与改进

本次实验主要关注漏洞的利用方法，对于漏洞的防御措施涉及较少。在未来的学习中，我应该进一步学习如何编写安全的 C 代码，以及如何使用静态分析和动态分析工具来

检测格式化字符串漏洞。