漏洞

strcpy所导致的缓冲区溢出

```
int main(){
    int socket = 1;
    char user[100];
    char pass[200];
    char buff[400];
    int c = 0;
    strncpy(buff,"USER:",100); // buff:"USER:00000..."
    send(socket, buff, 7, 0); // 将buff通过socket发送
    recv(socket, buff, 400, 0); //从socket接收400字节存入buff
    strncpy(user, buff, 100); // 将buff的前100个字节送进user
    snprintf(buff, 400, "Hello %s \nPASS", user); // 将Hello 【user】 \nPASS的前400
字节写入buff
    c = strlen(buff) + 1;
    send(socket, buff, c, 0);
    recv(socket, buff, 400, 0);
    strcpy(pass,buff);
    strncpy(buff, "Logged in", 100);
    send(socket, buff, 23, 0);
    return 0;
}
```

- 安全缺陷：缓冲区溢出漏洞，**第9行中的strncpy函数**中，将buff的前100个字节送入user中，但是如果此时user的第100个字节不为"\0"，那么user这个字符串就没有结束符，会将user高地址处的内容溢出泄露出来。同样地，**在第14行执行strcpy函数时**，也有可能导致pass被溢出，从而user以及其上方的内容被覆盖。
- 造成的危害：当用户第一次通过socket所提交的payload长度大于100，则会导致user字符串无法结束，从而user高位的内存空间将随着第10行的sprintf函数写入buff之中，然后通过12行的send发送给用户，用此方法构造payload能实现恶意攻击的作用；同样地，用户第二次通过socket所提交的payload长度大于300时，会将user上方内容覆盖，user之上包括main函数的返回地址，攻击者可以往栈中写入shellcode或者组织ROP链，通过修改main函数的返回地址以实现执行恶意代码的目的。
- 修补方式：将第9行的100改为99，以及将第14行的strcpy改为strncpy，并且限制进行copy的长度

格式化字符串漏洞的利用

```
int main(int argc, char ** argv){
    printf(argv[1]);
    return 0;
}
```

- 安全缺陷：格式化字符串漏洞，在printf函数之中，将main函数的参数直接传入，这样如果所传入的参数是一个格式化字符串，则printf函数就会试图对该字符串进行解析，从而实现对栈中的数据进行读写，以

下是常见的格式化字符串参数：

- %d - 十进制 - 输出十进制整数
- %s - 字符串 - 从内存中读取字符串
- %x - 十六进制 - 输出十六进制数
- %c - 字符 - 输出字符
- %p - 指针 - 指针地址
- %n - 到目前为止所写的字符数
- %hhn - 写1字节
- %hn - 写2字节
- %ln - 写4个字节
- %lln - 写8字节 其中$可以输出格式化字符串上方（这里的上指的是栈底的方向）指定偏移的数据。例如"%2$d"表示输出其上方的第二个参数，而%n则不输出字符，反而把已经成功输入的字符个数写入对应的整型指针参数所指的变量，只要变量对应的地址可写，就可以利用格式化字符串来改变其对应的值，例如"%6$n"表示将当前已经输出的字符数写入栈中的从低往高数第六个位置中。

- 造成的危害：利用格式化字符串漏洞，攻击者可以对栈中的数据进行随意读写，实现修改内存数据，写入shellcode等

- 修补方式：

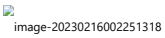
  - 避免使用格式化字符串函数：为了避免格式化字符串漏洞，可以尽量避免使用格式化字符串函数，例如printf()和scanf()等。如果必须使用这些函数，就要确保输入的字符串是受信任的，或者使用更安全的格式化字符串函数，例如snprintf()和sscanf()等。
  - 使用限制参数数量的格式化字符串函数：为了防止攻击者通过添加额外的参数来破坏程序的内存，可以使用限制参数数量的格式化字符串函数，例如printf("%10s", input)。这将确保只有一个参数被处理，避免了攻击者利用额外的参数来访问非法内存。
  - 使用类型安全的格式化字符串函数：某些编程语言和库提供了类型安全的格式化字符串函数，例如C++中的iostream和Java中的String.format()。这些函数使用静态类型检查和编译时检查来防止格式化字符串漏洞。
  - 对用户输入进行严格验证：为了避免格式化字符串漏洞，必须对所有用户输入的数据进行严格的验证和过滤。例如，可以使用正则表达式来验证输入的格式是否符合要求，或者使用白名单来过滤输入的字符集。
  - 使用内存安全的编程技术：内存安全的编程技术可以帮助避免格式化字符串漏洞。例如，可以使用内存安全的语言，例如Rust和Go等，或者使用内存安全的编程模式，例如RAII和智能指针等，来确保程序中没有内存安全问题。 ——ChatGPT

另附一种很难的考法：(115条消息) 格式化字符串漏洞总结_osahha的博客-CSDN博客_格式化字符串漏洞

分析格式化字符串漏洞的输出结果

```
int main(void){
    int i=1, j=2, k=3;
    char buf[] = "test";
    printf("%s %d %d %d\n",buf, i, j);
}
```

分析这段代码，不难发现因为printf函数的格式化字符串中的参数个数与在之前压栈的参数个数不符，会导致printf函数以%d的格式输出栈中不该输出的内容，首先画出栈结构如下：



当执行到printf时，首先从参数列表中读出栈中的格式化字符串，解析发现有四个格式化字符参数，其中前三个分别正常输出了"test"字符串，整数i和j的值，当读取到第四个字符参数时，printf取出了栈中参数j上方的4个字节，并试图将其以%d的方式进行输出，这四个字节恰好是main函数栈中的buf缓冲区的低4个字节，以下分两种情况对第四个输出结果进行分析：

因为"test"字符串实际上由5个字节，为了在内存中对齐，对于buf，实际上开辟了8字节的空间，那么"test\0"这5个字节在8字节空间内就有左右两种对齐方式：

- 左对齐：高 |00 "t" "s" "e" "t" 00 00 00| 低，此时将按照%d的格式输出"t" 00 00 00，即74 00 00 00 = 1946157056
- 右对齐：高 |00 00 00 00 "t" "s" "e" "t"| 低，此时将按照%d的格式输出"t" "s" "e" "t"，即74 73 65 74 = 1953719668

动态内存分配

```
nresp = packet_get_int();
if(nresp>0){
    response = xmalloc(nresp*sizeof(char*));
    for(i=0;i<nresp;i++)
        response[i] = packet_get_string(NULL);
}
```

- 安全缺陷：包含整数溢出漏洞和堆溢出漏洞，对于unsigned int，其最大值为0xffffffff，而因为`sizeof(char\*)`=4，所以如果设计`nresp=(0xffffffff/4)+1`并向上取整，即1073741825，那么`nresp*sizeof(char*)`便会产生整数溢出，最终得到的结果为4，这样malloc只会在堆中开辟4*4=16字节的空间，但是在接下来的for循环中，对response数组进行依次赋值，产生了溢出，覆盖了malloc分配的4字节以及高地址处的程序代码、函数指针等内容
- 修补方式：限定nresp的上限不能超过0xffffffff/4或是在对malloc开辟的缓冲区进行读写时，限制偏移不能超过缓冲区的大小

链表

某程序采用双向链表维护数据，空节点表示链首，数据结构定义如下：

```
struct node{
    struct node *flink;
    struct node *blink;
    char content[256];
}
```

编辑节点：

```
int edit(struct node * p_node){
    char buf[400];
    read(0, &buf, 400);
    if(p_node){
        strcpy(p_node->content, buf);
    }
}
```

移除节点：

```
int remove(struct node* p_node){
    if(p_node && pnode !=HEAD){
        p_node->blink->flink = p_node->flink;
        p_node->flink->blink = p_node->blink;
    }
}
```

- 安全缺陷：在编辑节点函数中，首先在read函数中，直接将读取到的400个字节写入长度为400的缓冲区中，此时如果末尾的截断符"\0"被覆盖，则会导致buf产生泄露，在接下来的strcpy函数中，就会将溢出的buf移动到content之中，导致content也产生了溢出，从而破坏内存布局甚至破坏链表结构。
- 修补方式：在read函数中限定长度为400，防止buf末尾的"\0"被覆盖，或是在strcpy中限制copy的长度。

傅建明猜数字题

2. 下表列出了一个猜数字游戏的部分代码，该游戏在连续猜对 10 次后才可获得奖励，试对代码进行分析并回答下列问题：

（1） 结合表中代码，给出 main 局部变量的位置并画出栈布局(不考虑传入参数)；（4 分）

（2） 结合栈布局分析代码存在的缺陷；（4 分）

（3） 给出利用该缺陷获取奖励的思路。（4 分）

部分源代码 & 汇编代码

```
1.  int main(int argc, char **argv)
2.  {
3.    int guess_num;
4.    int i;
5.    int random_num;
6.    unsigned int seed;
7.    char name[10];
8.
9.    guess_num = 0;
10.   random_num = 0;
11.
12.   seed = get_seed();
13.   puts("This is a guess number game!");
14.   puts("Please input your name!");
15.   printf("Your name:");
16.   gets(name);
17.   srand(seed);
18.   for ( i = 0; i <= 9; ++i )
19.   {
20.     random_num = rand() % 9 + 1;
21.     printf("---------Turn:%d---------\n", i + 1);
22.     printf("Please input your guess number:");
23.     scanf("%d", &guess_num);
24.     if ( guess_num != random_num )
25.     {
26.       puts("The number is Wrong and Game Over!");
27.       exit(1);
28.     }
29.     puts("Right!");
30.   }
31.   get_award();
32.   return 0;
33. }
```

```
text:004010C0 <main>:
10C0    push    ebp
10C1    mov     ebp, esp
10C3    sub     esp, 5Ch
10C6    push    ebx
10C7    push    esi
10C8    push    edi
10C9    lea     edi, [ebp-5Ch]
10CC    mov     ecx, 17h
10D1    mov     eax, 0CCCCCCCCh
10D6    rep stosd
10D8    mov     dword ptr [ebp-4], 0
10DF    mov     dword ptr [ebp-0Ch], 0
10E6    call    j__get_seed
10EB    mov     [ebp-10h], eax
10EE    push    4270CCh
10F3    call    _puts
10F8    add     esp, 4
10FB    push    4281C4h
1100    call    _puts
1105    add     esp, 4
1108    push    4270C0h
110D    call    _printf
1112    add     esp, 4
1115    lea     eax, [ebp-1Ch]
1118    push    eax             ; buffer
1119    call    _gets
111E    add     esp, 4
1121    mov     ecx, [ebp-10h]
1124    push    ecx             ; seed
1125    call    _srand
112A    add     esp, 4
112D    mov     dword ptr [ebp-8], 0
……
```

答案很简单：就是通过栈溢出去覆盖seed，知道了seed便可以知道每次rand取出来数的具体值了，汇编代码是吓唬人的。