但是，如果程序员懒惰地将上述代码段写为：

```
char *s = "Hello World";
printf(s);
```

这样调用printf是合法的，因为printf具有可变个数的参数，其中第一个参数必须是格式化字符串（Format String），当然不包括任何格式信息（如“%s”）的字符串也是允许的，所以尽管第二种编程风格并不推荐，但在这里并不会出问题，而且它还使得程序员少敲了五个按键，似乎是不错的改进。

6个月以后，其他程序员要求对这段代码进行修改，首先询问用户的姓名，在对该用户发出特定的欢迎信息。在草率阅读完原先的代码后，该程序员只做了一点改变：

```
char s[100], g[100] = "Hello";          /* 声明数组s和g，并初始化g */
gets(s);                                /*从键盘读取字符串，存放到数组s中*/
strcat(g, s);                           /* 把s连接到g的末尾 */
printf(g);                              /*输出g */
```

这段代码首先将用户输入的字符串存入*s*，然后将*s*连接到已经被初始化的字符串*g*之后，以在*g*中形成最终的输出信息。到现在这种方式依然能够正确地显示结果（gets函数很容易遭受缓冲区溢出攻击，然而由于其便于书写，因此到现在依然流行）。

然而，如果一个对C语言有所了解的用户看到了这段代码，他会立刻意识到程序从键盘输入的并不只是一个简单的字符串，而是一个格式化字符串（Format String），因此任何格式化标识符都会起作用。尽管绝大多数格式化标识符都规范了输出（例如，“%s”：打印一个字符串；“%d”：打印一个十进制整数），有一些却比较特殊。特别是“%n”，它不打印出任何信息，而是计算直到“%n”出现之前，总共打印了多少字符，并且将这个数字保存到printf下一个将要使用的参数中去。下面给出一个使用“%n”的例子：

```
int main(int argc, char *argv[])
{
    int i = 0;
    printf("Hello %nworld\n", &i);          /* 把%n 前出现的字符个数保存到变量i中*/
    printf("i = %d\n", i);                  /* i 现在的值为6 */
}
```

当这段代码被编译运行后，输出为：

```
Hello World
i = 6
```

注意到*i*的值在函数printf中以一种很不显眼的方式被修改了。这种特性只在极少数情况下有用，它意味着打印一个格式化字符串可能导致一个单词（或者很多单词）被存储在内存中。很显然让printf具有这样的特性并不是一个好主意，然而这个功能在当时看来是非常方便的。绝大多数软件的弱点都是因此而存在。

就像我们刚刚看到的一样，由于程序员对程序不严谨的修改，可能导致用户有了输入格式化字符串的机会。而打印一个格式化字符串可能导致内存被重写（overwrite），这就为覆盖栈中printf函数的返回地址提供了一种方法，通过重写这个返回地址，可以使得函数在printf函数返回时跳到任何位置，例如跳到刚刚输入的格式化字符串。这种攻击方式叫做*格式化字符串攻击*（format string attack）。

一旦用户可以修改内存并强制程序跳转到一段新注入的代码段，这段代码就具有了被攻击程序所拥有的所有权限。如果该程序是SETUID root，那么攻击者就可以创建一个具有root权限的shell。实现这种攻击的具体细节过于复杂，本书不再赘述。这里只想让读者知道，格式化字符串攻击是一个严重的问题。如果读者在Google搜索栏中输入“format string attack”（格式化字符串攻击），会找到很多的相关信息。

另外，值得一提的是，在本节的例子中，采用定长字符数组也很容易遭受缓冲区溢出攻击。

9.6.3 返回libc攻击

缓冲区溢出攻击和格式化字符串攻击都要求向栈中加入必要的数据，并将函数返回的地址指向这些