假设过程A的任务是得到完整的路径（可能是把当前目录路径和文件名串联起来），然后打开文件实施一些操作。A拥有固定长度的缓冲区（如数组）B，它存放着文件名，如图9-24b所示。使用定长缓冲区存放文件名比起先检测实际大小再动态分配空间要容易得多。如果缓冲区只有1024个字节，那么能够放得下所有的文件名吗？特别是当操作系统把文件名的长度限制（或者更好的是对全路径名的长度限制）在不超过255（或其他固定的长度）个字符时。

然而，上述推论有致命的错误。假设用户提供了一个长达2000个字符的文件名，在使用时就会出错，但攻击者却不予理会。当过程A把文件名复制到缓冲区时，文件名溢出并覆盖了图9-24c的灰色部分。更糟的是，如果文件名足够长，它还会覆盖返回地址，这样当过程A返回时，返回地址是从文件名的中间截取的。如果这一地址是随机数，系统将跳到该随机地址，并可能引起一系列的误操作。

但是如果文件名没有包含某些随机地址会怎么样呢？如果它包含的是有效的二进制地址并且设计得十分吻合某个过程的起始地址，那又会怎么样呢？例如吻合过程B的起始地址。如果真是这样，那么当过程A运行结束后，过程B就开始运行。实际上，攻击者会用他的恶意代码来覆盖内存中的原有代码，并且让这些代码被执行。

同样的技巧还运用于文件名之外的其他场合。如用在对较长的环境变量串、用户输入或任何程序员创建了定长缓冲区并需要用户输入变量的场合。通过手工输入一个含有运行程序的串，就有可能将这段程序装入到栈并让它运行。C语言函数库的gets函数可以把（未知大小的）串变量读入定长的缓冲区里，但并不校验是否溢出，这样就很容易遭受攻击。有些编译器甚至通过检查gets的使用来发出警告。

现在我们来讨论最坏的部分。假设被攻击的UNIX程序的SETUID为root（或在Windows里拥有管理员权限的程序），被插入的代码可以进行两次系统调用，把攻击者磁盘里的shell文件的权限改为SETUID root的权限，这样当程序运行时攻击者就拥有了超级用户的权限。或者，攻击者可以映射进一个特定的共享文件库，从而实施各种各样的破坏。还可以十分容易地通过exec系统调用来覆盖当前shell中运行的程序，并利用超级用户的权限建立新的shell。

更糟的是，恶意代码可以通过互联网下载程序或脚本，并将其存储在本地磁盘上。此后该恶意代码就可以创建一个进程直接从本地运行恶意程序或是脚本。该进程可以一直监听IP端口，从而等待攻击者的命令，这将目标机器变为僵尸。恶意代码必须保证每次机器启动后，恶意程序或脚本可以被启动，然而不论在Windows或所有版本的UNIX系统下，这都是很容易实现的。

绝大多数系统安全问题都与缓冲区溢出漏洞相关，而这类漏洞很难被修复，因为已有的大量C代码都没有对缓冲区溢出进行检查。

检测程序是否有缓冲区溢出问题较为简单：只要输入一个10 000字符长度的文件名，或100位数字的薪水金额，或一般不太会遇到的数字，观察主程序是否停止。接着分析代码找到长字符串存放的位置。从这个位置得到覆盖返回地址的字符就不难了。如有源代码，则对大多数UNIX程序来说就很容易实施攻击，因为栈的布局事先是知道的。对付这类攻击的办法是修改代码，显式地检查用户输入的所有变量的长度，从而避免把长字符串放入定长缓冲里。但是，实际上有些程序在一次攻击得手以后就变得更易遭受攻击。

9.6.2 格式化字符串攻击

尽管很多程序员都是很好的打字员，但事实上他们都不愿意打字。将变量名reference_count缩写为rc表达了相同的意思，却可以在每次使用该变量的时候减少了13个字符的输入，对程序员来说何乐而不为呢？然而这种偷懒行为在下面描述的情况中，却可能导致系统灾难性地崩溃。

考虑下面的C程序代码片段，该段代码打印了一段欢迎信息：

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

在这段代码声明了一个字符指针类型的变量s，该变量被初始化指向一个字符串“Hello World”，注意在这个字符串的末尾有一个额外的字符‘\0’用以标记该字符串的结束。函数printf被传入两个参数，其中格式化字符串‘%s’指定系统接下来打印的是一个字符串，第二个参数s则告诉printf该字符串的起始地址。当被执行的时候，这段代码会在屏幕上打印出“Hello World”（在任何标准输出中，都可以成功执行）。