C语言：防止缓冲区溢出

C中大多数缓冲区溢出问题可以直接追溯到标准 C 库。最有害的罪魁祸首是不进行自变量检查的、有问题的字符串操作`strcpy`、`strcat`、`sprintf` 和 `gets`。一般来讲，象“避免使用 `strcpy()`和永远不使用`gets()`这样严格的规则接近于这个要求。

今天，编写的程序仍然利用这些调用，因为从来没有人教开发人员避免使用它们。某些人从各处获得某个提示，但即使是优秀的开发人员也会被这弄糟。他们也许在危险函数的自变量上使用自己总结编写的检查，或者错误地推论出使用潜在危险的函数在某些特殊情况下是安全的。

第一位公共敌人是 `gets()`。永远不要使用 `gets()`。该函数从标准输入读入用户输入的一行文本，它在遇到 `EOF`字符或换行字符之前，不会停止读入文本。也就是：`gets()` 根本不执行边界检查。因此，使用 `gets()`总是有可能使任何缓冲区溢出。作为一个替代方法，可以使用方法 `fgets()`。它可以做与 `gets()`所做的同样的事情，但它接受用来限制读入字符数目的大小参数，因此，提供了一种防止缓冲区溢出的方法。例如，不要使用以下代码：

```
void main()
{
    char buf[1024];
    gets(buf);
}
```

而使用以下代码：

```
#define BUFSIZE 1024
void main()
{
    char buf[BUFSIZE];
    fgets(buf, BUFSIZE, stdin);
}
```

C 编程中的主要陷阱

C语言中一些标准函数很有可能使您陷入困境。但不是所有函数使用都不好。通常，利用这些函数之一需要任意输入传递给该函数。这个列表包括：

- strcpy()
- strcat()
- sprintf()
- scanf()
- sscanf()
- fscanf()
- vfscanf()
- vsprintf
- vscanf()
- vsscanf()
- streadd()
- strecpy()

- strtrns()

坏消息是我们推荐，如果有任何可能，避免使用这些函数。好消息是，在大多数情况下，都有合理的替代方法。我们将仔细检查它们中的每一个，所以可以看到什么构成了它们的误用，以及如何避免它。

strcpy()函数将源字符串复制到缓冲区。没有指定要复制字符的具体数目。复制字符的数目直接取决于源字符串中的数目。如果源字符串碰巧来自用户输入，且没有专门限制其大小，则有可能会陷入大的麻烦中！

如果知道目的地缓冲区的大小，则可以添加明确的检查：

```
if(strlen(src) >= dst_size)
{
  /* Do something appropriate, such as throw an error. */
}
else
{
  strcpy(dst, src);
}
```

完成同样目的的更容易方式是使用 strncpy() 库例程：

```
strncpy(dst, src, dst_size-1);
dst[dst_size-1] = '\0'; /* Always do this to be safe! */
```

如果 src 比 dst 大，则该函数不会抛出一个错误；当达到最大尺寸时，它只是停止复制字符。注意上面调用 strncpy()中的 -1。如果 src 比 dst 长，则那给我们留有空间，将一个空字符放在 dst 数组的末尾。

当然，可能使用strcpy()不会带来任何潜在的安全性问题，正如在以下示例中所见：

```
strcpy(buf, "Hello!");
```

即使这个操作造成 buf 的溢出，但它只是对几个字符这样而已。由于我们静态地知道那些字符是什么，并且很明显，由于没有危害，所以这里无须担心 — 当然，除非可以用其它方式覆盖字符串Hello所在的静态存储器。

确保strcpy()不会溢出的另一种方式是，在需要它时就分配空间，确保通过在源字符串上调用 strlen() 来分配足够的空间。例如：

```
dst = (char *)malloc(strlen(src));
strcpy(dst, src);
```

strcat()函数非常类似于strcpy()，除了它可以将一个字符串合并到缓冲区末尾。它也有一个类似的、更安全的替代方法strncat()。如果可能，使用strncat()而不要使用strcat()。

函数sprintf()和vsprintf()是用来格式化文本和将其存入缓冲区的通用函数。它们可以用直接的方式模仿strcpy()行为。换句话说，使用sprintf()和vsprintf()与使用strcpy()一样，都很容易对程序造成缓冲区溢出。例如，考虑以下代码：

```
void main(int argc, char **argv)
{
    char usage[1024];
    sprintf(usage, "USAGE: %s -f flag [arg1]\n", argv[0]);
}
```

我们经常会看到类似上面的代码。它看起来没有什么危害。它创建一个知道如何调用该程序字符串。那样，可以更改二进制的名称，该程序的输出将自动反映这个更改。虽然如此，该代码有严重的问题。文件系统倾向于将任何文件的名称限制于特定数目的字符。那么，您应该认为如果您的缓冲区足够大，可以处理可能的最长名称，您的程序会安全，对吗？只要将1024改为对我们的操作系统适合的任何数目，就好了吗？但是，不是这样的。通过编写我们自己的小程序来推翻上面所说的，可能容易地推翻这个限制：

```
void main()
{
  execl("/path/to/above/program",
        <<insert really long string here>>,
        NULL);
}
```

函数execl()启动第一个参数中命名的程序。第二个参数作为argv[0]传递给被调用的程序。我们可以使那个字符串要多长有多长！

那么如何解决sprintf()带来得问题呢？遗憾的是，没有完全可移植的方法。某些体系结构提供了snprintf()方法，即允许程序员指定将多少字符从每个源复制到缓冲区中。例如，如果我们的系统上有snprintf，则可以修正一个示例成为：

```
void main(int argc, char **argv)
{
    char usage[1024];
    char format_string = "USAGE: %s -f flag [arg1]\n";
    snprintf(usage, format_string, argv[0],1024-strlen(format_string) + 1);
}
```

注意，在第四个变量之前，snprintf()与sprintf()是一样的。第四个变量指定了从第三个变量中应被复制到缓冲区的字符最大数目。注意，1024 是错误的数目！我们必须确保要复制到缓冲区使用的字符串总长不超过缓冲区的大小。所以，必须考虑一个空字符，加上所有格式字符串中的这些字符，再减去格式说明符 %s。该数字结果为 1000,但上面的代码是更具有可维护性，因为如果格式字符串偶然发生变化，它不会出错。

sprintf()的许多（但不是全部）版本带有使用这两个函数的更安全的方法。可以指定格式字符串本身每个自变量的精度。例如，另一种修正上面有问题的sprintf()的方法是：

```
void main(int argc, char **argv)
{
    char usage[1024];
    sprintf(usage, "USAGE: %.1000s -f flag [arg1]\n", argv[0]);
}
```

注意，百分号后与 s 前的 .1000。该语法表明，从相关变量（本例中是 argv[0]）复制的字符不超过 1000 个。

如果任一解决方案在您的程序必须运行的系统上行不通，则最佳的解决方案是将snprintf()的工作版本与您的代码放置在一个包中。可以找到以sh归档格式的、自由使用的版本；请参阅 参考资料。

继续，scanf系列的函数也设计得很差。在这种情况下，目的地缓冲区会发生溢出。考虑以下代码：

```
void main(int argc, char **argv)
{
  char buf[256];
  sscanf(argv[0], "%s", &buf);
}
```

如果输入的字大于 buf 的大小，则有溢出的情况。幸运的是，有一种简便的方法可以解决这个问题。考虑以下代码，它没有安全性方面的薄弱环节：

```
void main(int argc, char **argv)
{
  char buf[256];
  sscanf(argv[0], "%255s", &buf);
}
```

百分号和 s 之间的 255 指定了实际存储在变量 buf 中来自 argv[0] 的字符不会超过 255 个。其余匹配的字符将不会被复制。

接下来，我们讨论streadd()和strecpy()。由于，不是每台机器开始就有这些调用，那些有这些函数的程序员，在使用它们时，应该小心。这些函数可以将那些含有不可读字符的字符串转换成可打印的表示。例如，考虑以下程序：

```
#include <libgen.h>

void main(int argc, char **argv)
{
    char buf[20];
    streadd(buf, "\t\n", "");
    printf(%s\n", buf);
}
```

该程序打印：

```
\t\n
```

而不是打印所有空白。如果程序员没有预料到需要多大的输出缓冲区来处理输入缓冲区（不发生缓冲区溢出），则 streadd() 和 strecpy()函数可能有问题。如果输入缓冲区包含单一字符 — 假设是 ASCII 001（control-A）–则它将打印成四个字符\001。这是字符串增长的最坏情况。如果没有分配足够的空间，以至于输出缓冲区的大小总是输入缓冲区大小的四倍，则可能发生缓冲区溢出。

另一个较少使用的函数是strtrns()，因为许多机器上没有该函数。函数strtrns()取三个字符串和结果字符串应该放在其内的一个缓冲区，作为其自变量。第一个字符串必须复制到该缓冲区。一个字符被从第一个字符串中复制到缓冲区，除非那个字符出现在第二个字符串中。如果出现的话，那么会替换掉第三个字符串中同一索引中的字符。这听上去有点令人迷惑。让我们看一下，将所有小写字符转换成大写字符的示例：

```
#include <libgen.h>
void main(int argc, char **argv)
{
    char lower[] = "abcdefghijklmnopqrstuvwxyz";
    char upper[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    char *buf;
    if(argc < 2) {
      printf("USAGE: %s arg\n", argv[0]);
      exit(0);
    }
    buf = (char *)malloc(strlen(argv[1]));
    strtrns(argv[1], lower, upper, buf);
    printf("%s\n", buf);
}
```

以上代码实际上不包含缓冲区溢出。但如果我们使用了固定大小的静态缓冲区，而不是用malloc()分配足够空间来复制argv[1]，则可能会引起缓冲区溢出情况。

避免内部缓冲区溢出

realpath()函数接受可能包含相对路径的字符串，并将它转换成指同一文件的字符串，但是通过绝对路径。在做这件事时，它展开了所有符号链接。

该函数取两个自变量，第一个作为要规范化的字符串，第二个作为将存储结果的缓冲区。当然，需要确保结果缓冲区足够大，以处理任何大小的路径。分配的MAXPATHLEN缓冲区应该足够大。然而，使用realpath()有另一个问题。如果传递给它的、要规范化的路径大小大于MAXPATHLEN，则realpath()实现内部的静态缓冲区会溢出！虽然实际上没有访问溢出的缓冲区，但无论如何它会伤害您的。结果是，应该明确不使用realpath()，除非确保检查您试图规范化的路径长度不超过MAXPATHLEN。

其它广泛可用的调用也有类似的问题。经常使用的syslog()调用也有类似的问题，直到不久前，才注意到这个问题并修正了它。大多数机器上已经纠正了这个问题，但您不应该依赖正确的行为。最好总是假定代码正运行在可能最不友好的环境中，只是万一在哪天它真的这样。getopt()系列调用的各种实现，以及getpass()函数，都可能产生内部静态缓冲区溢出问题。如果您不得不使用这些函数，最佳解决方案是设置传递给这些函数的输入长度的阈值。

自己模拟gets()的安全性问题以及所有问题是非常容易的。例如，下面这段代码：

```
char buf[1024];
int i = 0;
char ch;
while((ch = getchar()) != '\n')
{
    if(ch == -1) break;
    buf[i++] = ch;
}
```

哎呀！可以用来读入字符的任何函数都存在这个问题，包括getchar()、fgetc()、getc() 和 read()。

缓冲区溢出问题的准则是：总是确保做边界检查。

C 和 C++ 不能够自动地做边界检查，这实在不好，但确实有很好的原因，来解释不这样做的理由。边界检查的代价是效率。一般来讲，C 在大多数情况下注重效率。然而，获得效率的代价是，C 程序员必须十分警觉，并且有极强的安全意识，才能防止他们的程序出现问题，而且即使这些，使代码不出问题也不容易。

在现在，变量检查不会严重影响程序的效率。大多数应用程序不会注意到这点差异。所以，应该总是进行边界检查。在将数据复制到您自己的缓冲区之前，检查数据长度。同样，检查以确保不要将过大的数据传递给另一个库，因为您也不能相信其他人的代码！（回忆一下前面所讨论的内部缓冲区溢出。）

其它危险是什么?

遗憾的是，即使是系统调用的“安全”版本 — 譬如，相对于strcpy()的strncpy()也不完全安全。也有可能把事情搞糟。即使安全的调用有时会留下未终止的字符串，或者会发生微妙的相差一位错误。当然，如果您偶然使用比源缓

冲区小的结果缓冲区，则您可能发现自己处于非常困难的境地。

与我们目前所讨论的相比，往往很难犯这些错误，但您应该仍然意识到它们。当使用这类调用时，要仔细考虑。如果不仔细留意缓冲区大小，包括`bcopy()`、`fgets()`、`memcpy()`、`snprintf()`、`strccpy()`、`strcadd()`、`strncpy()` 和 `vsnprintf()`，许多函数会行为失常。

另一个要避免的系统调用是 `getenv()`。使用`getenv()` 的最大问题是您从来不能假定特殊环境变量是任何特定长度的。我们将在后续的专栏文章中讨论环境变量带来的种种问题。

到目前为止，我们已经给出了一大堆常见 C 函数，这些函数容易引起缓冲区溢出问题。当然，还有许多函数有相同的问题。特别是，注意第三方 COTS 软件。不要设想关于其他人软件行为的任何事情。还要意识到我们没有仔细检查每个平台上的每个常见库（我们不想做那一工作），并且还可能存在其它有问题的调用。

即使我们检查了每个常见库的各个地方，如果我们试图声称已经列出了将在任何时候遇到的所有问题，则您应该持非常非常怀疑的态度。我们只是想给您起一个头。其余全靠您了。

静态和动态测试工具

我们将在以后的专栏文章中更加详细地介绍一些脆弱性检测的工具，但现在值得一提的是两种已被证明能有效帮助找到和去除缓冲区溢出问题的扫描工具。这两个主要类别的分析工具是静态工具（考虑代码但永不运行）和动态工具（执行代码以确定行为）。

可以使用一些静态工具来查找潜在的缓冲区溢出问题。很糟糕的是，没有一个工具对一般公众是可用的！许多工具做得一点也不比自动化 grep 命令多，可以运行它以找到源代码中每个有问题函数的实例。由于存在更好的技术，这仍然是高效的方式将几万行或几十万行的大程序缩减到只有数百个“潜在的问题”。（在以后的专栏文章中，将演示一个基于这种方法的、草草了事的扫描工具，并告诉您有关如何构建它的想法。）

较好的静态工具利用以某些方式表示的数据流信息来断定哪个变量会影响到其它哪个变量。用这种方法，可以丢弃来自基于 grep 的分析的某些“假肯定”。David Wagner 在他的工作中已经实现了这样的方法（在“Learning the basics of buffer overflows”中描述；请参阅 参考资料），在 Reliable Software Technologies 的研究人员也已实现。当前，数据流相关方法的问题是它当前引入了假否定（即，它没有标志可能是真正问题的某些调用）。

第二类方法涉及动态分析的使用。动态工具通常把注意力放在代码运行时的情况，查找潜在的问题。一种已在实验室使用的方法是故障注入。这个想法是以这样一种方式来检测程序：对它进行实验，运行“假设”游戏，看它会发生什么。有一种故障注入工具 — FIST（请参阅 参考资料）已被用来查找可能的缓冲区溢出脆弱性。

最终，动态和静态方法的某些组合将会给您的投资带来回报。但在确定最佳组合方面，仍然有许多工作要做。

Java 和堆栈保护可以提供帮助

堆栈捣毁是最恶劣的一种缓冲区溢出攻击，特别是，当在特权模式下捣毁了堆栈。这种问题的优秀解决方案是非可执行堆栈。通常，利用代码是在程序堆栈上编写，并在那里执行的。（我们将在下一篇专栏文章中解释这是如何做到的。）获取许多操作系统（包括 Linux 和 Solaris）的非可执行堆栈补丁是可能的。（某些操作系统甚至不需要这样的补丁；它们本身就带有。）

非可执行堆栈涉及到一些性能问题。（没有免费的午餐。）此外，在既有堆栈溢出又有堆溢出的程序中，它们易出问题。可以利用堆栈溢出使程序跳转至利用代码，该代码被放置在堆上。没有实际执行堆栈中的代码，只有堆中的代码。

当然，另一种选项是使用类型安全的语言，譬如 Java。较温和的措施是获取对 C 程序中进行数组边界检查的编译器。对于 gcc 存在这样的工具。这种技术可以防止所有缓冲区溢出，堆和堆栈。不利的一面是，对于那些大量使用指针、速度是至关重要的程序，这种技术可能会影响性能。但是在大多数情况下，该技术运行得非常好。

Stackguard 工具实现了比一般性边界检查更为有效的技术。它将一些数据放在已分配数据堆栈的末尾，并且以后会在缓冲区溢出可能发生前，查看这些数据是否仍然在那里。这种模式被称之为“金丝雀”。（威尔士的矿工将 金丝雀放在矿井内来显示危险的状况。当空气开始变得有毒时，金丝雀会昏倒，使矿工有足够时间注意到并逃离。）

Stackguard 方法不如一般性边界检查安全，但仍然相当有用。Stackguard 的主要缺点是，与一般性边界检查相比，它不能防止堆溢出攻击。一般来讲，最好用这样一个工具来保护整个操作系统，否则，由程序调用的不受保护库（譬如，标准库）可以仍然为基于堆栈的利用代码攻击打开了大门。

类似于 Stackguard 的工具是内存完整性检查软件包，譬如，Rational 的 Purify。这类工具甚至可以保护程序防止堆溢出，但由于性能开销，这些工具一般不在产品代码中使用。

结束语

在本专栏的上两篇文章中，我们已经介绍了缓冲区溢出，并指导您如何编写代码来避免这些问题。我们还讨论了可帮助使您的程序安全远离可怕的缓冲区溢出的几个工具。表 1 总结了一些编程构造，我们建议您小心使用或避免一起使用它们。如果有任何认为我们应该将其它函数加入该列表，请则通知我们，我们将更新该列表。

函数	严重性	解决方案
gets	最危险	使用 fgets（buf, size, stdin）。这几乎总是一个大问题！
strcpy	很危险	改为使用 strncpy。
strcat	很危险	改为使用 strncat。
sprintf	很危险	改为使用 snprintf，或者使用精度说明符。
scanf	很危险	使用精度说明符，或自己进行解析。
sscanf	很危险	使用精度说明符，或自己进行解析。
fscanf	很危险	使用精度说明符，或自己进行解析。
vfscanf	很危险	使用精度说明符，或自己进行解析。
vsprintf	很危险	改为使用 vsnprintf，或者使用精度说明符。
vscanf	很危险	使用精度说明符，或自己进行解析。
vsscanf	很危险	使用精度说明符，或自己进行解析。
streadd	很危险	确保分配的目的地参数大小是源参数大小的四倍。
strecpy	很危险	确保分配的目的地参数大小是源参数大小的四倍。
strtrns	危险	手工检查来查看目的地大小是否至少与源字符串相等。
realpath	很危险（或稍小，取决于实现）	分配缓冲区大小为 MAXPATHLEN。同样，手工检查参数以确保输入参数不超过 MAXPATHLEN。
syslog	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getopt	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getopt_long	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getpass	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getchar	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
fgetc	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
read	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
bcopy	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
fgets	低危险	确保缓冲区大小与它所说的一样大。
memcpy	低危险	确保缓冲区大小与它所说的一样大。
snprintf	低危险	确保缓冲区大小与它所说的一样大。
strccpy	低危险	确保缓冲区大小与它所说的一样大。

函数	严重性	解决方案
strcadd	低危险	确保缓冲区大小与它所说的一样大。
strncpy	低危险	确保缓冲区大小与它所说的一样大。
getchar	低危险	确保缓冲区大小与它所说的一样大。
vsnprintf	低危险	确保缓冲区大小与它所说的一样大。

-END-

推荐阅读

【01】C语言内存泄露很严重，如何应对?
【02】编译C语言程序，使用 gcc 指令，而C++程序则推荐使用 g++指令！
【03】C语言：优雅的字符串函数库
【04】在C 语言中，请一定记得初始化局部变量！
【05】嵌入式编程是否应该用C++替代C语言