文章编号:1001 - 9081(2003)06Z - 0292 - 02

Linux 下动态内存分配安全性分析

刘圣卓,谢余强,魏 强
(解放军信息工程大学 信息工程学院,河南 郑州 450002)

摘 要:介绍了在 Linux 系统下由 malloc、free 等一系列函数实现的动态内存分配算法,分析了这种算法实现可能带来的安全问题及其产生的原因,并提出了相应的解决办法。

关键词:动态内存分配;Chunk;GNU;安全;缓冲区溢出
中图分类号: TP333.3 **文献标识码**:A

1 引言

C 程序运行过程中主要涉及三种内存分配方式——静态分配、自动分配和动态分配。C 语言自身支持两种重要的内存分配方式——静态分配和自动分配。程序中定义的静态或全局变量占用一块固定大小的内存,这些内存只在程序开始运行时被分配一次,并且不会被释放,这种分配方式被称为静态分配。程序定义的自动变量的分配和释放情况就不同了。自动变量的内存空间是在进入包含这些变量定义的声明部分时分配的,并在退出时释放,这种分配方式被称为自动分配。除了这两种内存分配方法外,GNU C 函数库还支持另外一种重要的内存分配方式——动态分配。

GNU C 函数库,即 glibc,是 Linux 上最重要的函数库,它定义了 ISO C 标准指定的所有的库函数,以及由 POSIX 或其他 Unix 操作系统变种指定的附加函数,还包括有与 GNU 系统相关的扩展。目前,流行的 Linux 系统使用 glibc 2.0 以上的版本。

GNU C 库函数中有一些函数可以动态地分配与回收可变大小的内存,这些函数包括 malloc(3)/calloc(3)/free(3)/realloc(3)。它们提供了一个应用程序和系统调用之间的内存分配接口。通过使用这些函数,应用程序在运行过程中可以将一块大的内存块分割成小的块(chunk),并分配给应用程序使用;在不使用时,可以释放这些分配的 chunk 并回收碎片。这种在程序运行过程分配和回收内存的分配方式被称为动态分配。

这个动态内存分配实现的设计目标包括:稳定、高效、避免碎片和低空间开销。

2 GNU C 库动态内存分配的实现

GNU C 用来管理动态内存的数据结构被称为 chunk(注:本文在不产生歧义的情况下,把"chunk"称为"块"或"内存块"),其具体定义为:

```
struct malloc-chunk
{
  INTERNAL-SIZE-T prev-size;
                        /* 前一个 chunk 的大小(如果未分配) */
  INTERNAL-SIZE-T size;          /* 本 chunk 的大小 */
  struct malloc-chunk * fd;      /* 双向链指针(未分配时) */
  struct malloc-chunk * bk;
};
```

此结构的各字段并不都是一直有效的,在不同的情况下某些字段可能没有意义,具体说明如下:内存块被分配后的情形如图 1 所示:

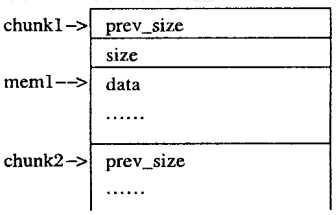

图 1 chunk 被分配的情况

如果使用 mem1 = malloc(20);申请内存,mem1 为 malloc 函数返回的指针,它将如图 1 所示指向申请到的内存块的起始位置,程序的数据从这里开始存放。在 mem1 之前有两个字段:prev-size 和 size,它们存放着这个块的管理信息。

prev-size 和 size 均占用 4 字节空间。对 prev-size 而言,如果当前块的前一个块未被使用,它存放着前一个块的大小;当前块的前一个块被使用,它则作为前一个块的数据区的一部分。如图 1 所示 chunk2 的 prev-size 就是 chunk1 的数据区的一部分。

Size 字段也有其特殊的定义:它除了包含当前块的大小信息外,还包含有其它的管理信息。size 的低三位存放其它信息。size 的计算方法是将欲申请的空间大小加 4(size 自身占用的空间)然后与下一个 4 字边界对齐。也就是说,malloc(7)时,size 为 16;malloc(20)时,size 为 24 等。这样做更多的是性能方面的考虑。size 的最低位称为 PREV-INUSE 位,表示前一个块是否被使用(1 表示被使用,0 表示未被使用)。第二低位称为 IS-MMAPPED 位,表示此块是不是由 mmap()分配的(1 表示此块由 mmap()分配)。如果是由 mmap()分配的,则释放时由 munmap-chunk()去释放;否则,释放时由 chunk-free()完成。第三低位没有被使用。由此,检查当前的块是否被使用,只需查看下一个块的 size 字段的最低位是否为 1。

内存块被释放后的情形如图 2 所示:

收稿日期:2002 - 08 - 21

作者简介:刘圣卓(1975 -),男,山东即墨人,助理工程师,硕士研究生,主要研究方向:计算机网络安全; 谢余强(1963 -),男,湖北武汉人,副教授,硕士,主要研究方向:计算机网络安全; 魏强(1979 -),男,江西南昌人,硕士研究生,主要研究方向:计算机网络安全.

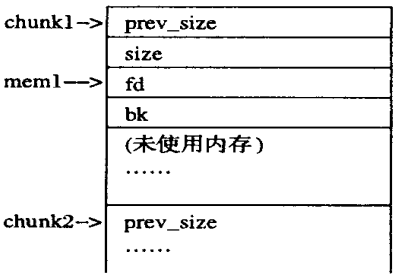

图 2　chunk 被释放的情况

在块被释放时,除了 prev-size 和 size 字段外,fd 和 bk 也变为有意义的字段。另外,chunk2 的 prev-size 字段将表示 chunk1 的大小。

内存的分配和释放可能多次进行,如果每次分配都只是在未分配的空间分配一块新的空间,将导致空间的浪费,因而必须要回收被释放的空间;同样,还应当考虑分配后的碎片的回收。在 GNU C 实现中使用一个双向链表来管理被释放的空间,所有被释放的块都会被放到这个链中。fd 和 bk 就是用来构成双向链表的两个指针,fd 指向下一个块,bk 指向前一个块。在内存块被释放时,要将与其相邻的空闲的块组成较大的块以减少碎片。

当调用 free() 释放内存时,首先对释放的内存做一些相关的检查,如当前的块是否是由 mmap() 分配的等。然后调用 chunk-free() 释放当前的块。

chunk-free() 首先要检查相邻的两个块的使用情况。要检查前一个块是否是被分配的,只需检查当前块的 size 最低位是否为 1;检查后一个块的使用情况则复杂些,要检查当前块之后的第二个块的 size 字段的最低位。

如果有空闲的块,就将它从空闲块的双向链表中摘下(调用 unlink 完成),然后把这些相邻的空闲的块合并,并将合并后的块加到空闲链表中。如果都不是空闲的,就只修改下一块的 prev-size 和 size 字段的值,然后将当前块加入到空闲链表中。

将一个空闲块从空闲链表中摘下的具体操作如图 3、图 4 所示。图 3 表示了 chunk0、chunk1 和 chunk2 为空闲链表中的 3 个相邻的块。其中 chunk1 为需要摘下的块。

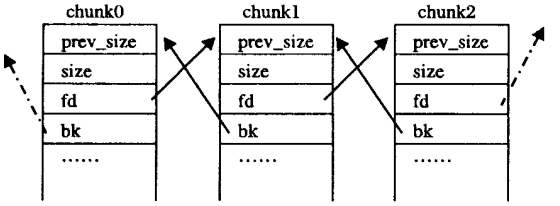

图 3　空闲链表的链接情况

将 chunk1 从链表中摘下时,链表指针需要修改,如图 4 所示,加粗的两个指针为被修改的指针。

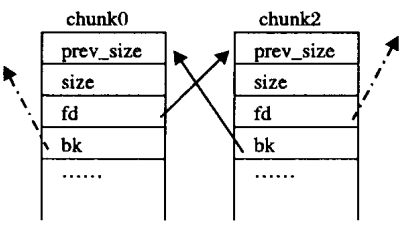

图 4　从空闲链表中摘下 chunk1

在具体实现中,这一步的操作使用了一个名为 unlink 的宏,其定义如下:

```
# define unlink(P, BK, FD)        \
{                                 \
  BK = P->bk;                     \
  FD = P->fd;                     \
  FD->bk = BK;                    \
  BK->fd = FD;                    \
}
```

其操作的结果就是将链上前一个块的 fd 指针改写成后一个块的地址;将后一个块的 bk 指针改写为前一个块的地址。从而将当前块从空闲链表中摘下。

将一个块加入到空闲链表中的操作与从空闲链表中摘下块的操作是类似的,在此就不再赘述了。

通过以上的分析,我们可以看到 GNU C 库函数对内存的动态分配与回收的算法达到了它的设计目标——稳定、高效、避免碎片和低空间开销。尤其是在高效、避免碎片和低空间开销方面效果明显,但是这个算法却会给系统带来严重的安全问题,这也是过多的追求效率和性能的原因。

3　安全问题

大多数的系统服务程序都要对客户端发来的请求信息进行保存,保存的方式一般先为这段数据动态申请一块空间,然后将数据拷贝到这块内存中。由于出于性能的考虑或编程人员的粗心,可能忽略对其进行严格的边界检查,有可能允许在一块内存块中写入过多的数据,从而导致内存中的其它块结构被改写;这时如果做释放内存操作,将可能导致某处内存空间被意外改写。

这个问题的主要原因是出在释放内存时,如果释放时需要回收碎片,则首先需要调用 unlink() 将碎片从空闲链上摘下。unlink 所做的工作是要修改两个指针的值,正如原理介绍部分中所述,unlink 的具体操作是:

- BK = P->bk
- FD = P->fd
- FD->bk = BK
- BK->fd = FD

其中包括两个对内存的修改:

- P->fd->bk = P->bk
- P->bk->fd = P->fd

根据块的结构上面两式可改写为:

- *((P->fd) + 12) = P->bk
- *((P->bk) + 8) = P->fd

就此可以看到 unlink 做的实际工作是:如果传给它的参数为 P,它将 (P->bk) 的内容写到 ((P->fd) + 12) 处,并将 (P->fd) 的内容写到 ((P->bk) + 8) 处。

如果可以通过在请求服务时,在请求数据中携带过长的字符串或其它数据,而达到覆盖部分内存的目的。就有可能在一块将要释放的块处放置一个或多个伪造的块,当程序调用 free() 释放内存时,由于伪造的块的作用,可使得程序调用 unlink() 释放一个伪造的块。根据前面的分析,可以在此块的 fd 和 bk 字段写入特定值,当 unlink 执行时,就可在指定的内存处写入指定的值。

改写内存往往是某些攻击手段的重要步骤。可以被用于改变程序的执行流程,如改写. dtors段,使得程序退出时运行

(下转第 296 页)

4.1 计算图标“获得机器码”

用于获取用户计算机的机器码,在计算图标里写入以下代码:

```
MemUnit := AllocMem(1)
                                        //分配内存单元
GetVolumeInformation("c:\ \ ","",30,MemUnit,256,"","",30)
                                        //读取硬盘分区 C 序列号
HardSerialNumber := MemUnit
                                        //序列号赋给自定义变量 HardSerialNumber
FreeMem(MemUnit)
                                        //释放分配单元
```

其中 AllocMem、FreeMem 这两个函数是封装于 memtools.u32 的内存分配函数,memtools.u32 也是 Authorware6.0 的一个附带扩展函数库。本代码中的关键函数是 GetVolumeInformation,它是 Winapi.u32 的函数一员,用于实现对硬盘某分区序列号的获取,并返回一个 10 进制的序列号。

4.2 显示机器码

在前面已建立了一个自定义变量“HardSerialNumber”来记录读取出来的序列号,即所谓的机器码 在“显示机器码”中用文本框显示 HardSerialNumber,其显示属性选中“Update Displayed Variables”选项,这样变量才会动态改变。

4.3 设计“发送机器码”交互

添加一个交互图标到流程上,然后建立“发送机器码”和“退出”按钮响应分支。发送机器码响应分支的计算图标代码如下:

```
ShellExecuteA(WindowHandle,"open","mailto:lfy1234 @21cn.com?Subject=请求注册 &body=机器码:
"^HardSerialNumber,"","",5)
```

其中 ShellExecuteA 是 Windows(或 WINNT)系统目录下 shell32.dll 的一个封装函数,导入方法和导入 U32 函数方法类似。在这里设定作者的电子邮箱地址为 lfy1234 @21cn.com。这样作为 Authorware 作品的作者一方就可以根据用户的机器码通过一定的算法产生一个有效的注册码并返回给用户,需要注册的 Authorware 程序的注册部分设计方法类似于上文的“注册登录法”。

5 总结

以上是 Authorware 作品的加密保护的四种常用方法,在实际中往往将它们结合起来使用,以此提高作品的加密能力。但是没有一种加密方法是绝对安全的,但可以不断开发与运用更新的技术如反跟踪技术,在不影响软件的正常使用的基础上合理使用多种加密设计,可以大大提高 Authorware 作品的安全性。

参考文献

[1] 赵永吉. Authorware6 多媒体制作技术与实例[M]. 北京:中国水利水电出版社,2002.

[2] 宋一兵. Authorware5 多媒体制作实例详解[M]. 北京:人民邮电出版社,2000.

[3] 冯凯锋. 一种基于公钥密码算法的序列号软件保护方案[J]. 计算机应用,2002,22(4):94-95.

[4] 丁思捷,张普朝. 应用硬盘序列号生成计算机指纹[J]. 计算机应用,2002,22(5):106-108.

(上接第 293 页)

特定的代码;改写某些函数指针,当被改写的指针函数被调用时,执行的将不是原来的函数;改写函数的返回地址,函数返回时程序流程改变,等等。这意味着攻击者可以借此执行恶意代码。

4 安全问题的解决

问题产生的主要原因有两个:一是将内存块的管理信息和数据连续存放在同一连续空间内;二是对内存块的分配和释放操作过程中没有进行必要的有效性检查。管理信息和数据的连续存放使得可以通过过长的数据改写并伪造管理信息,缺少有效性检查使伪造的管理信息能达到其攻击目的——改写内存。

4.1 阻止管理信息被数据覆盖

管理信息被覆盖通常首先是由于缓冲区溢出引起的,也就是在将数据拷贝到缓冲区时没有作边界条件检查。防止缓冲区溢出成为最基本的解决办法,但是由于 C 语言等高级语言的容易出错倾向,缓冲区溢出在实际编程中是很难完全避免的。管理信息和数据连续存放也是管理信息被覆盖的一个重要原因,为此将管理信息和数据分开存放也是一个好方法。这样做的确可以防止管理信息被覆盖,但无疑必将增加空间和性能开销。

4.2 阻止对内存的非法修改

对内存的非法修改是通过伪造管理信息实现的,而更直接的原因是在释放内存时缺少必要的合法性检查。因而可着重对如下条件进行检查:

- 释放的内存块指针的范围应当在.BSS 区内;
- 块的 size 字段应大于 0,即下一个块不应位于本块之前;
- 双向链指针的范围应有所限定,如一般情况下应指向.BSS 区。

通过这些条件检查可以阻止大部分的攻击。

以上两种解决方法虽然可以在一定程度上阻止利用动态内存分配机制对内存的改写,但是同时也带来了空间的浪费和性能的下降。可见,安全性和系统开销是一对矛盾,增加安全性必将导致系统开销增大,同样过分追求降低系统开销将可能带来安全问题。

参考文献

[1] GNU C 库手册[EB/OL]. http://www.gnu.org/manual/glibc-2.2.3/html-node/libc-toc.html,2001.

[2] Anonymous. Once upon a free() ...[J/OL]. http://www.phrack.org/phrack/57/p57-0x09,Phrack,2001.

[3] Matt Conover. w00w00 on Heap Overflows [EB/OL]. http://www.w00w00.org/files/articles/heaptut.txt,1999.