snprintf 和 sprintf 的区别

2020 年 6 月 14 日

摘要

一些文献中描述，snprintf 比 sprintf 更安全，为了确定其使用的原理，对 snprintf 和 sprintf 进行分析，探索 snprintf 更加安全的奥秘。为此本文做了以下工作：

1. 分析了 snprintf 和 sprintf 实现中用到的数据结构，主要用的数据结构是：FILE, IO_strfile 和 IO_strnfile。

2. 分析了 snprintf 和 sprintf 函数的实现方式，解析 snprintf 的安全性体现在哪里。

3. 分析其中一个字符串查找函数 rawmemchr 的实现方式。

1 introduction

snprintf 和 sprintf 的函数原型及其实现为 listing 1, 可以看到在最外层，只是解析了可变参数 (代码第 4 行和第 14 行)，实际上 ___nldbl_vsnprintf 函数没有做任何实际性的工作，而是调用了 ___vsnprintf_internal，同理 ___nldbl_vsprintf 函数调用了 ___vsprintf_internal，而这两个函数的实现 listing **??**.

这两个函数实现上有三点需要注意的

1._IO_strnfile 和 _IO_strfile 结构体不同。

2._IO_str_init_static_internal 使用的参数不同。

3.___vfprintf_internal 执行实际的格式化操作((C 程序设计语言这本书里有类似的函数，但是实现上会简单)。

接下来对这三点进行分析。

```
1  //sysdeps/ieee754/ldbl-opt/nldbl-
       snprintf.c
2  int snprintf (char *s, size_t maxlen,
       const char *fmt, ...){
3    ...
4    va_start (arg, fmt);
5    done = __nldbl_vsnprintf (s, maxlen,
         fmt, arg);
6    va_end (arg);
7    ...
8  }
9
10 //sysdeps/ieee754/ldbl-opt/nldbl-sprintf
       .c
11 int attribute_hidden sprintf (char *s,
       const char *fmt, ...)
12 {
13   ...
14   va_start (arg, fmt);
15   done = __nldbl_vsprintf (s, fmt, arg);
16   va_end (arg);
17   ...
18 }
```

Listing 1: sprintf snprintf

```
//libio/iovsprintf.c
int __vsprintf_internal (char *string,
    size_t maxlen, const char *format,
    va_list args, unsigned int
    mode_flags)
{
  _IO_strfile sf;
  _IO_no_init (&sf._sbf._f,
      _IO_USER_LOCK, -1, NULL, NULL);
  _IO_str_init_static_internal (&sf,
      string, (maxlen == -1) ? -1 :
      maxlen - 1, string);
  ret = __vfprintf_internal (&sf._sbf._f
      , format, args, mode_flags);
  *sf._sbf._f._IO_write_ptr = '\0';
}

int __vsnprintf_internal (char *string,
    size_t maxlen, const char *format,
    va_list args, unsigned int
    mode_flags)
{
  _IO_strnfile sf;
  if (maxlen == 0)
  {
    string = sf.overflow_buf;
    maxlen = sizeof (sf.overflow_buf);
  }

  _IO_no_init (&sf.f._sbf._f,
      _IO_USER_LOCK, -1, NULL, NULL);
  string[0] = '\0';
  _IO_str_init_static_internal (&sf.f,
      string, maxlen - 1, string);
  ret = __vfprintf_internal (&sf.f._sbf.
      _f, format, args, mode_flags);
  if (sf.f._sbf._f._IO_buf_base != sf.
      overflow_buf)
    *sf.f._sbf._f._IO_write_ptr = '\0';
  }
}
```

Listing 2: internal

2 struct

对比 _IO_strnfile 和 _IO_strfile 两个结构体的结构，两个结构值的关注的地方如图 1

可以看到 ___IO_strnfile 其实是将一个 ___IO_strfile 和一串多余的字符串（缓存区）结合在一起，这里继续往内部看，看到我们最熟悉的 FILE 结构体（在 vim 还会变颜色的），发现其实只是对指针进行了封装，共分为两类指针 wide_data 内的 wchar_t(4 bytes) 指针，char(1 bytes)。

顾名思义，wide_data 一次性传输的数据会更多，减少循环次数。两类指针其实都包含着指向内存的指针，分为读指针和写指针，FILE 只对内存进行操作，也就是是说，在打开文件的时候，会调用系统调用 mmap 将文件内容映射到内存当中。应用程序只需对内存操作。mmap 具体实现还没有细看，但是我的理解是会将磁盘上的一个块映射到内存中，如果文件操作超出一个块则利用缺页中断对内存进行补充（这些以后写操作系统时再细讲）。同时，每个 FILE 其实都有个锁结构（图中没有写出来），也就是说 FILE 相当于在裸内存中外层加了一个读写器，并且保证每个读写器进行操作是原子操作，当然可以两个 FILE 指向同一个内存区（但是没有人会这么用，因为初始化过于复杂）。

本文的主要目的是讨论 ___IO_strnfile 和 ___IO_strfile，发现 ___IO_strnfile 实际上是 ___IO_strfile 加上一个缓存区，再对比 listing 2中

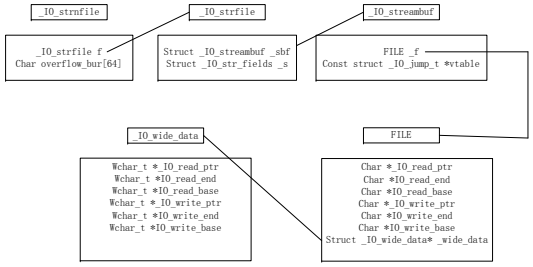

图 1: _IO_strnfile and _IO_strfile

的 14 到 17 行，发现 snprintf 的第一个保护机制是当指定了目标地址的长度为 0 的时候，使用 ___IO_strnfile 中的 overflow_buf 作为目标字符串，作为第一个保护机制，但是直觉上该保护机制并没有提供很强的保护，将长度设置为 1，写入大与长度 1 的字符串则依然是不安全的函数。

3 IO__str__init__static__internal

该函数的主要作用是初始化 FILE 结构体中的指针，完成初始化即可通过对内存的操作完成对 FILE 的操作，函数具体内容为如 listing 3所示，初始化是对几个重要指针进行赋值，值的注意的是第 6 到第 11 行，对 end 的值进行计算，这里又有一个保护机制，情况也是处理 size 为 0 的情况，第 7 行调用 rawmemchr 找到字符串结尾函数，就是不超过现有的字符串长度，rawmemchr 是一个不可靠的查找函数，使用者必须保证当前字符串有特定的字符，否着会一直查找下去。

```
1  void _IO_str_init_static_internal (
       _IO_strfile *sf, char *ptr, size_t
       size, char *pstart)
2  {
3    FILE *fp = &sf->_sbf._f;
4    char *end;
5
6    if (size == 0)
7      end = __rawmemchr (ptr, '\0');
8    else if ((size_t) ptr + size > (size_t
         ) ptr)
9      end = ptr + size;
10   else
11     end = (char *) -1;
12   _IO_setb (fp, ptr, end, 0);
13
14   fp->_IO_write_base = ptr;
15   fp->_IO_read_base = ptr;
16   fp->_IO_read_ptr = ptr;
17   if (pstart)
18   {
19     fp->_IO_write_ptr = pstart;
20     fp->_IO_write_end = end;
21     fp->_IO_read_end = pstart;
22   }
23   else
24   {
25     fp->_IO_write_ptr = ptr;
26     fp->_IO_write_end = ptr;
27     fp->_IO_read_end = end;
28   }
29   /* A null _allocate_buffer function
        flags the strfile as being static.
         */
30   sf->_s._allocate_buffer_unused = (
       _IO_alloc_type) 0;
31 }
```

Listing 3: _IO__str__init__static__internal

4 vsnprintf

vsnprintf 格式化字符串，格式化字符串的部分对安全没有特别大的影响，这里主要查看写入函数，写入函数最后调用 PUT 函数往 FILE 结构体写入数据，PUT 实际是 _IO_default_xsputn 函数，其实现如 listing 4所示。在第 10 行中，对剩余空间进行检查，保证安全性。

这个函数有个有意思的地方，对剩余空间进行检查，发现大于 20 个字节调用 memcpy，否则一个一个复制，保证效率。

```
1  size_t _IO_default_xsputn (FILE *f,
       const void *data, size_t n)
2  {
3    const char *s = (char *) data;
4    size_t more = n;
5    if (more <= 0)
6      return 0;
7    for (;;)
8    {
9    /* Space available. */
10     if (f->_IO_write_ptr < f->
         _IO_write_end)
11     {
12       size_t count = f->_IO_write_end -
           f->_IO_write_ptr;
13       if (count > more)
14         count = more;
15       if (count > 20)
16       {
17         f->_IO_write_ptr = __mempcpy (f
             ->_IO_write_ptr, s, count);
18         s += count;
19       }
20       else if (count)
21       {
22         char *p = f->_IO_write_ptr;
23         ssize_t i;
24         for (i = count; --i >= 0; )
25           *p++ = *s++;
26         f->_IO_write_ptr = p;
27       }
28       more -= count;
29     }
30     if (more == 0 || _IO_OVERFLOW (f, (
         unsigned char) *s++) == EOF)
31       break;
32     more--;
33   }
34   return n - more;
35 }
```

Listing 4: _IO_default_xsputn

5 rawmemchr

上述分析中遇到了两个有意思的函数 rawmemchr 和 memcpy，字符串操作函数在编写程序时是最常见的函数，所以往往要求会很高，选一个分析，分析字符串查找函数 ___rawmemchr，其实现是用汇编代码写成的 (这里分析 X86 的 32 体系里的，长模式不懂，不过原理都一样的)，其实现如 listing 5所示。

正常思路是一个一个查找，但是为了提高效率，glic 选择了把 4byte 一起查找，充分利用寄存器长度，其主要思路是通过将有个寄存器复制为 c|c|c|c(c 为目标字符)，然后与字符串进行异或，然后查找为 0 的字符串，所以 1. 将 ecx 赋值成 c|c|c|c(7 到 13 行). 2. 对不对齐的部分进行单一字符查找，代码 (15-31)。3. 利用相等的值异或后为 0 查找是否包含字符串，(由第 33 开始，同样的也是一个循环处理 4 次，一次处理 64 个 byte)，利用异或之后的结果与 0xfefefeff 想加，其中 0x00 那一 byte 则肯定是 0xff，如果最高位则不会进位 (40 行)，否则其高位等与原值减 2，比如

第 1 个 byte 为 0，第 2byte 为 a，想加后第 2byte 为 a-2, 再跟 a 异或，则第 2byte 的最低位肯定为 0，则整个 edi 加 1 不会进位，从而发现目标。

```
    .text
ENTRY (__rawmemchr)
pushl %edi
cfi_adjust_cfa_offset (4)
cfi_rel_offset (edi, 0)

movl STR(%esp), %eax
movl CHR(%esp), %edx

movb %dl, %dh
movl %edx, %ecx
shll 16, %edx
movw %cx, %dx

testb 3, %al
je L(1)
cmpb %dl, (%eax)
je L(9)
incl %eax

testb 3, %al
je L(1)
cmpb %dl, (%eax)
je L(9)
incl %eax

testb 3, %al
je L(1)
cmpb %dl, (%eax)
je L(9)
incl %eax

ALIGN (4)

L(1):   movl (%eax), %ecx
movl 0xfefefeff, %edi
xorl %edx, %ecx
addl %ecx, %edi

jnc L(8)

xorl %ecx, %edi

orl 0xfefefeff, %edi
incl %edi

jnz L(8)

movl 4(%eax), %ecx
movl 0xfefefeff, %edi
xorl %edx, %ecx
addl %ecx, %edi
jnc L(7)
xorl %ecx, %edi
orl 0xfefefeff, %edi
incl %edi
jnz L(7)

movl 8(%eax), %ecx
movl 0xfefefeff, %edi
xorl %edx, %ecx
addl %ecx, %edi
jnc L(6)
xorl %ecx, %edi
orl 0xfefefeff, %edi
incl %edi
jnz L(6)

movl 12(%eax), %ecx
movl 0xfefefeff, %edi
xorl %edx, %ecx
addl %ecx, %edi
jnc L(5)
```

```
74  xorl %ecx, %edi
75  orl 0xfefefeff, %edi
76  incl %edi
77  jnz L(5)
78
79  L(5):   addl 4, %eax
80  L(6):   addl 4, %eax
81  L(7):   addl 4, %eax
82  L(8):   testb %cl, %cl
83  jz L(9)
84  incl %eax
85
86  testb %ch, %ch
87  jz L(9)
88  incl %eax
89
90  testl 0xff0000, %ecx
91  jz L(9)
92  incl %eax
93
94  L(9):
95  popl %edi
96  cfi_adjust_cfa_offset (-4)
97  cfi_restore (edi)
98
99  ret
100 END (__rawmemchr)
```

Listing 5: ___rawmemstr

6 summary

1. snprintf 其实只是对长度进行检查，并没有提供十分强的安全保护，编写代码程序员还是需要谨慎使用，比如目的字符串的长度为 2，size 为 4，一样能完成编译，出现漏洞。

2. 用汇编写代码能提高效率，但是为了减少循环，要一次性处理多个字节，但是在此之前要先对齐，最好使用位运算提高计算效率。