由安全函数整改想到的

秦新良

2015 年 1 月 11 日

目　录

1 问题提出

前段时间做安全函数整改，有部分文件未整改彻底，最近由赵阳同学接手将剩余遗留的非安全函数清理干净。在评审整改后的代码时，我对代码1的修改[1]方案提出了如下检视意见：

- pgwlib.h 是 BE 发布给 FE 的头文件，不应该在该头文件中包含 uscdbcommon.h；
- USCDB_MEMSET 宏对 FE 不可见，不能在这里使用该宏定义，否则 FE 切换版本后会导致 lib 库编译失败。

针对以上两点检视意见，我给出的解决方法是：将类 DataNode 的构造函数挪到 pgwlib.cpp 文件中，这样即解决了安全问题，也不会对 FE 的 lib 库有任何影响。

```
1  #ifndef PGWLIB_H
2  #define PGWLIB_H
3
4  #include <string>
5
6  // Include uscdbcommon.h for USCDB_MEMSET macro's definition
7  #include "uscdbcommon.h"
8
9  class DataNode
10 {
11 public:
12     DataNode()
13     {
14         // Change memset to secure function USCDB_MEMSET
15         USCDB_MEMSET(data, 256, 256, 0);
16     }
17
18     virtual ~DataNode();
19
20 private:
21     char   data[256];
22     string strData;
23 };
24
25 #endif /* PGWLIB_H */
```

代码 1: pgwlib.h

[1]代码1中的注释处即修改点。

这一切看似很完美，但过了一会儿，赵阳发eSpace说，这样改不行，如果lib库中有该类的实例，实例化时会失败，因为找不到类DataNode构造函数的定义。

看了这样的回复觉得很“无语[2]”。这不就是找一个函数地址吗，lib库被PGW加载后，和PGW是在同一进程地址空间内的，为什么会找不到PGW定义的函数？

支撑我这一想法的，除了lib库最终加载后和PGW在同一进程地址空间外，还有就是在该类整改前，DataNode的构造函数会调用memset函数和string的构造函数，这两个函数分别来自libc和libstdc++库。在编译lib库时，lib库看到的也只是标准库的头文件，并不会真正的把这两个标准库链接到动态库中，最终lib库中在有使用到memset或string的相关函数时，使用的也还是PGW进程加载起来的且和PGW在同一地址空间的标准库中的函数。那么lib库能找得到PGW加载起来的标准库中定义的函数，就能找的到PGW自己定义的函数。

但在冷静的分析之后，发现赵阳的说法也有一定道理。因为目前pgwlib.h头文件中定义的类，这些类的所有函数均是申明时就直接实现了，并没有放到任何一个cpp文件里。看了这样的实现后，决定写段代码来验证一下我的想法。

2 问题验证

验证的代码很简单，共三个源文件，见代码2、代码3和代码4。

common.h模拟pgwlib.h。在common.h中定义类DataNode。该类有数据成员strData和除构造、析构函数外的成员函数printData。

```
1  #ifndef COMMON_H
2  #define COMMON_H
3
4  #include <string>
5
6  using namespace std;
7
8  class DataNode
9  {
10 public:
11     DataNode();
12     virtual ~DataNode();
13
```

[2]无语归无语，但不得不说赵阳同学的基本功还是很扎实，能一眼看出问题之所在。

```
14 public:
15     void printData();
16
17 private:
18     string strData;
19 };
20
21 #endif /* COMMON_H */
```

代码 2: common.h

main.cpp 模拟 PGW。在该文件中除了实现 main 函数外，还完成了类 DataNode 相关函数的实现。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <dlfcn.h>
4
5  #include "common.h"
6
7  typedef DataNode* (*PFGETNODE)();
8
9  DataNode::DataNode()
10 {
11     strData = "Hello world!";
12 }
13
14 DataNode::~DataNode()
15 {
16 }
17
18 void DataNode::printData()
19 {
20     printf("%s\n", strData.c_str());
21     return;
22 }
23
24 int main(int argc, char *argv[])
25 {
26     void *handle = dlopen("./libdatanode.so", RTLD_LAZY);
27     if (NULL == handle)
28     {
29         fprintf(stderr, "%s\n", dlerror());
30         return -1;
31     }
32
33     char *error = NULL;
34
35     PFGETNODE pfFunc = (PFGETNODE)dlsym(handle, "_Z7getNodev");
36     if ((error = dlerror()) != NULL)
37     {
```

```
38          fprintf(stderr, "%s\n", error);
39          return -1;
40      }
41
42      DataNode *pNode = pfFunc();
43      if (NULL == pNode)
44      {
45          printf("allocate data node failed.\n");
46          return -1;
47      }
48
49      pNode->printData();
50      delete pNode;
51
52      if (dlclose(handle) < 0)
53      {
54          fprintf(stderr, "%s\n", dlerror());
55          return -1;
56      }
57
58      return 0;
59  }
```

代码 3: main.cpp

datanode.cpp 模拟 FE 的 lib 库，该文件包含了 common.h 头文件，并且定义了函数 getNode，返回类 DataNode 的对象。

```
1   #include <stdio.h>
2
3   #include "common.h"
4
5   DataNode* getNode()
6   {
7       DataNode *pNode = new DataNode;
8       if (NULL == pNode)
9       {
10          printf("new failed.\n");
11          return NULL;
12      }
13
14      return pNode;
15  }
```

代码 4: datanode.cpp

模拟验证所需的源文件都已具备，执行如下两条命令即可生成可执行文件 main 和 libdatanode.so 动态库：

```
g++ -m32 -g -Wall main.cpp -o main -ldl
g++ -m32 -g -Wall -shared -fPIC datanode.cpp -o libdatanode.so
```

生成可执行文件和库文件后，按照之前的设想，运行main程序会打印出字符串”Hello world!”，但实际执行的结果如下：

```
q00148943@Inspiron-3421:~$ ./main
./main: symbol lookup error: ./libdatanode.so: undefined symbol: _ZN8DataNodeC1Ev
```

符号_ZN8DataNodeC1Ev用c++filt demangle后就是DataNode::DataNode(),即类DataNode的构造函数。也就是说，将pgwlib.h中类的构造及成员函数放到pgwlib.cpp文件中实现是行不通的。

虽然验证的结果是不可行的，但仍觉得这个结果太不靠谱了，同一地址空间中的函数地址肯定是有办法相互可见的。想到这里，将libdatanode.so中的getNode函数反汇编后研究了下，看getNode函数中是如何调用libc及libstdc++库中的printf和new的，它和调用DataNode的构造函数有何不同。反汇编getNode的代码如5所示。

```
1  <_Z7getNodev>:
2  55                      push   %ebp
3  89 e5                   mov    %esp,%ebp
4  57                      push   %edi
5  56                      push   %esi
6  53                      push   %ebx
7  83 ec 2c                sub    $0x2c,%esp
8  e8 c7 fe ff ff          call   5a0 <__x86.get_pc_thunk.bx>
9  81 c3 27 19 00 00       add    $0x1927,%ebx
10 c7 04 24 08 00 00 00    movl   $0x8,(%esp)
11 e8 75 fe ff ff          call   560 <_Znwj@plt>
12 89 c6                   mov    %eax,%esi
13 89 34 24                mov    %esi,(%esp)
14 e8 5b fe ff ff          call   550 <_ZN8DataNodeC1Ev@plt>
15 89 75 e4                mov    %esi,-0x1c(%ebp)
```

```
16 83 7d e4 00              cmpl   $0x0,-0x1c(%ebp)
17 75 15                    jne    713 <_Z7getNodev+0x48>
18 8d 83 48 e7 ff ff        lea    -0x18b8(%ebx),%eax
19 89 04 24                 mov    %eax,(%esp)
20 e8 64 fe ff ff           call   570 <puts@plt>
21 b8 00 00 00 00           mov    $0x0,%eax
22 eb 19                    jmp    72c <_Z7getNodev+0x61>
23 8b 45 e4                 mov    -0x1c(%ebp),%eax
24 eb 14                    jmp    72c <_Z7getNodev+0x61>
25 89 c7                    mov    %eax,%edi
26 89 34 24                 mov    %esi,(%esp)
27 e8 1e fe ff ff           call   540 <_ZdlPv@plt>
28 89 f8                    mov    %edi,%eax
29 89 04 24                 mov    %eax,(%esp)
30 e8 54 fe ff ff           call   580 <_Unwind_Resume@plt>
31 83 c4 2c                 add    $0x2c,%esp
32 5b                       pop    %ebx
33 5e                       pop    %esi
34 5f                       pop    %edi
35 5d                       pop    %ebp
36 c3                       ret
```

代码 5: getNode 函数的汇编代码

代码5中的第 11 行是调用 new 函数；14 行是调用 DataNode 的构造函数；第 20 行是调用 puts 函数[3]。先不管这三行 call 指令后面的 @plt 是什么意思，单从这三处函数调用的“模样”来看，它们是一样的。即调用 DataNode 的构造函数和调用 new 及 printf 没有任何差异。而且调用 new 函数在前，DataNode 的构造函数在后。但 main 运行时在打开 libdatanode.so 时，并没有对 new 报 undefined symbol。

前段时间同样是在做安全整改时，切换 VPP 组件出了很多问题。出问题后又将 CSAPP 中的 Linking 章节看了一遍。该章节在对动态库的使用举例的过程中，编译可执行程序

[3]源代码中是调用 printf 函数，编译后是直接调用的 puts 函数。

时用到了 gcc 的 -rdynamic 这个编译选项，对该选项的功能还有一点印象，于是在编译 main.cpp 时，将该选项加上，编译顺利通过，再运行，久违的"Hello world" 出现了，如下所示：

```
q00148943@Inspiron-3421:~$ ./main
Hello world!
```

3 问题分析

通过上一节知道，在编译 main.cpp 时加上 -rdynamic 参数后，程序就可以正常运行了。-rdynamic 表示什么，先看 gcc 的 man 手册对 rdynamic 的说明，如下：

-rdynamic

Pass the flag **-export-dynamic** to the ELF linker, on targets that support it. This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table. This option is needed for some uses of **dlopen** or to allow obtaining backtraces from within a program.

从 man 手册的解释可以看出，rdynamic 选项的作用就是告诉链接器将所有符号[4]都添加到动态符号表中；而且这里提到了两个使用场景：dlopen 和 backtraces。

dlopen 的使用场景就是上一节的测试场景，这里再通过 CSAPP 中对 dlopen 函数的说明来加深对 rdynamic 的理解，如下：

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
```

The dlopen function loads and links the shared library **filename**. The external symbols in **filename** are resolved using libraries previously opened with the **RTLD_GLOBAL** flag. If the current executable was compiled with the **-rdynamic** flag, then its global symbols are also available for symbol resolution. The flag argument must include either RTLD_NOW, which tells the linker

[4] 目标链接程序的所有符号。

to resolve references to external symbols immediately, or the RTLD_LAZY flag, which instructs the linker to defer symbol resolution until code from the library is executed.

上面蓝色字体部分解释了为什么在打开libdatanode.so动态库时，new和printf能正确解析而不报undefined symbol的错误；绿色字体部分解释了为什么开启rdynamic选项之后可以正确解析DataNode的构造函数。

rdynamic选项的另一个使用场景就是在程序运行出错打印错误日志的同时记录当前函数的调用栈，代码6为使用样例代码。

```c
/*
 * http://www.gnu.org/software/libc/manual/html_node/Backtraces.html
 */

#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>

/* Obtain a backtrace and print it to stdout. */
void print_trace(void)
{
    void *array[10];
    size_t size;
    char **strings;
    size_t i;

    size = backtrace(array, 10);
    strings = backtrace_symbols(array, size);

    for (i = 0; i < size; i++)
        printf("%s\n", strings[i]);

    free(strings);
}

/* A dummy function to make the backtrace more interesting. */
void dummy_function(void)
{
    print_trace();
}

int main(int argc, char *argv[])
{
    dummy_function();
    return 0;
}
```

代码 6: back_trace.c

代码6开启与不开启 rdynamic 编译选项编译的二进制的执行结果对比如图1所示。即开启 rdynamic 选项后，不仅可以打出函数调用栈上每个函数的地址，还可以打印各个函数的函数名，有了函数名更利于分析定位问题。

```
q00148943@Inspiron-3421:~$ ./back_trace
./back_trace(print_trace+0x19) [0x8048716]
./back_trace(dummy_function+0xb) [0x8048774]
./back_trace(main+0xb) [0x8048781]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0xf7612a83]
./back_trace() [0x8048621]
--------------------------------------------------------------------------------
q00148943@Inspiron-3421:~$ ./back_trace
./back_trace() [0x80484e6]
./back_trace() [0x8048544]
./back_trace() [0x8048551]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0xf75e1a83]
./back_trace() [0x80483f1]
```

图 1: 函数调用栈

上面对 rdynamic 选项的基本功能及使用场景作了简要的说明。下面阐述为什么编译时加上 rdynamic 选项能达到这样的效果。

```
q00148943@Inspiron-3421:~$ readelf --dyn-sym main

Symbol table '.dynsym' contains 23 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsaSEPKc@GLIBCXX_3.4 (2)
     2: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsC1Ev@GLIBCXX_3.4 (2)
     3: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
     5: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNKSs5c_strEv@GLIBCXX_3.4 (2)
     6: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZdlPv@GLIBCXX_3.4 (2)
     7: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlclose@GLIBC_2.0 (3)
     8: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (4)
     9: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
    10: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsD1Ev@GLIBCXX_3.4 (2)
    11: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    12: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlsym@GLIBC_2.0 (3)
    13: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlopen@GLIBC_2.1 (6)
    14: 00000000     0 FUNC    GLOBAL DEFAULT  UND fprintf@GLIBC_2.0 (4)
    15: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlerror@GLIBC_2.0 (3)
    16: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __pthread_key_create
    17: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (4)
    18: 00000000     0 FUNC    GLOBAL DEFAULT  UND _Unwind_Resume@GCC_3.0 (7)
    19: 0804a060    44 OBJECT  WEAK   DEFAULT   26 _ZTVN10__cxxabiv117__clas@CXXABI_1.3 (5)
    20: 08048b0c     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    21: 0804a08c     4 OBJECT  GLOBAL DEFAULT   26 stderr@GLIBC_2.0 (4)
    22: 08048760     0 FUNC    GLOBAL DEFAULT  UND __gxx_personality_v0@CXXABI_1.3 (5)
```

图 2: 不开启 rdynamic

从 gcc 的 man 手册可以知道，rdynamic 选项的作用就是将目标链接程序的所有符号添加到程序的动态符号表[5]中，这一点可以使用代码3和 readelf 这一命令行工具来做验证。

[5]动态符号表就是动态链接器在解析符号时所要查找的表，如果表中存在某个待解析的符号，程序运行正常，否则上报 undefined symbol。

readelf 工具的 –dyn-sym 选项的功能就是显示 ELF 文件的动态符号表。代码3不开启 rdynamic 选项编译后，使用 readelf 查看编译后二进制的动态符号表结果如图2所示。

图3是开启 rdynamic 选项后的结果。把图2和图3的结果作对比可以看出，开启 rdynmic 选项后，动态符号表中的记录多了 21 个 entries，其中就包括类 DataNode 的成员函数及其它的一些函数，这就是开启 rdynamic 选项后能解析到类 DataNode 的构造函数的最根本原因。

```
q00148943@Inspiron-3421:~$ readelf --dyn-sym main

Symbol table '.dynsym' contains 44 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsaSEPKc@GLIBCXX_3.4 (2)
     2: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsC1Ev@GLIBCXX_3.4 (2)
     3: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    [...] skipping output
    19: 08048df4     0 FUNC    GLOBAL DEFAULT   14 _fini
    20: 08048c45   312 FUNC    GLOBAL DEFAULT   13 main
    21: 08048c26    31 FUNC    GLOBAL DEFAULT   13 _ZN8DataNode9printDataEv
    22: 08048958     0 FUNC    GLOBAL DEFAULT   11 _init
    23: 0804b050     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    24: 08048e0c     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    25: 08048e60    16 OBJECT  WEAK   DEFAULT   15 _ZTV8DataNode
    26: 0804b094     0 NOTYPE  GLOBAL DEFAULT   26 _end
    27: 0804b048     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    28: 0804b060    44 OBJECT  WEAK   DEFAULT   26 _ZTVN10__cxxabiv117__clas@CXXABI_1.3 (5)
    29: 08048e78    10 OBJECT  WEAK   DEFAULT   15 _ZTS8DataNode
    30: 08048a60     0 FUNC    GLOBAL DEFAULT  UND __gxx_personality_v0@CXXABI_1.3 (5)
    31: 0804b08c     4 OBJECT  GLOBAL DEFAULT   26 stderr@GLIBC_2.0 (4)
    32: 08048e70     8 OBJECT  WEAK   DEFAULT   15 _ZTI8DataNode
    33: 0804b050     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    34: 08048df0     2 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
    35: 08048a80     0 FUNC    GLOBAL DEFAULT   13 _start
    36: 08048d80    97 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    37: 0804b048     0 NOTYPE  WEAK   DEFAULT   25 data_start
    38: 08048e08     4 OBJECT  GLOBAL DEFAULT   15 _fp_hw
    39: 08048c08    30 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeD0Ev
    40: 08048bd4    51 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeD1Ev
    41: 08048b7e    86 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeC1Ev
    42: 08048bd4    51 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeD2Ev
    43: 08048b7e    86 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeC2Ev
```

图 3: 开启 rdynamic

如果使用 readelf -s 查看一个二进制，结果会有两个符号表，如图4所示。图中的.dynsym 即是动态符号表，另一个则是符号表.symtab。这两个符号表的不同点如下：

- 动态符号表是符号表的子集；
- loader 加载二进制时会将动态符号表加载到内存，而符号表则不会加载到内存；
- 动态符号表是给 dynamic linker 使用的，而符号表是给 linker 及 debugger 使用的。
- 符号表可通过 objcopy 或 strip 命令从二进制中剥离，动态符号表则不可以，如果强制剥离，会导致二进制无法执行。

图5演示了首先使用strip命令将二进制main的符号表剥离，然后执行，main运行正常；最后再通过strip命令将动态符号表剥离，再执行，则报错。

```
q00148943@Inspiron-3421:~$ readelf -s main

Symbol table '.dynsym' contains 44 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsaSEPKc@GLIBCXX_3.4 (2)
     2: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsC1Ev@GLIBCXX_3.4 (2)
    [...] skipping output
    27: 0804b048     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    28: 0804b060    44 OBJECT  WEAK   DEFAULT   26 _ZTVN10__cxxabiv117__clas@CXXABI_1.3 (5)
    29: 08048e78    10 OBJECT  WEAK   DEFAULT   15 _ZTS8DataNode
    [...] skipping output
    43: 08048b7e    86 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeC2Ev

Symbol table '.symtab' contains 98 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048154     0 SECTION LOCAL  DEFAULT    1
     2: 08048168     0 SECTION LOCAL  DEFAULT    2
    [...] skipping output
    89: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlerror@@GLIBC_2.0
    90: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __pthread_key_create
    91: 0804b094     0 NOTYPE  GLOBAL DEFAULT   26 _end
    92: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.0
    93: 0804b050     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    94: 08048a60     0 FUNC    GLOBAL DEFAULT  UND __gxx_personality_v0@@CXX
    95: 00000000     0 FUNC    GLOBAL DEFAULT  UND _Unwind_Resume@@GCC_3.0
    96: 08048c45   312 FUNC    GLOBAL DEFAULT   13 main
    97: 08048958     0 FUNC    GLOBAL DEFAULT   11 _init
```

图 4: 符号表

```
q00148943@Inspiron-3421:~$ strip --strip-all main
q00148943@Inspiron-3421:~$ readelf -s main

Symbol table '.dynsym' contains 44 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSsaSEPKc@GLIBCXX_3.4 (2)
    [...] skipping output
    28: 0804b060    44 OBJECT  WEAK   DEFAULT   26 _ZTVN10__cxxabiv117__clas@CXXABI_1.3 (5)
    29: 08048e78    10 OBJECT  WEAK   DEFAULT   15 _ZTS8DataNode
    [...] skipping output
    43: 08048b7e    86 FUNC    GLOBAL DEFAULT   13 _ZN8DataNodeC2Ev
q00148943@Inspiron-3421:~$ ./main
Hello world!
q00148943@Inspiron-3421:~$ strip -R .dynsym main
q00148943@Inspiron-3421:~$ readelf -s main
q00148943@Inspiron-3421:~$ ./main
./main: relocation error: ./main: symbol , version GLIBC_2.0 not defined in file libc.so.6 with link time reference
```

图 5: strip

在本章节的最后，对代码5中call指令后面的@plt作一点介绍性的说明。PLT，Procedure Linkage Table的缩写，是为解决动态库代码段不能共享而提出的一个概念。

代码5的第20行即是代码4第10行的汇编代码，该行汇编代码在执行时会直接跳转到Procedure Linkage Table的一个entry，即代码7的第23行[6]，假设是PLT的第n个entry，记为PLT[n]。第23行同样是一条jump指令，这次这条jump指令跳转到了GOT（Global Offset Table的缩写）的一个entry，假设是GOT[n]。第一次调用puts函数时，GOT[n]是指

[6]call后面的地址是570，即代码7的第23行。

向PLT[n]的第二条指令的，即代码7的第24行。第24行压栈了一个立即数，紧接着25行又是一条跳转指令，跳转的地址是520，即第7行，PLT[0]的第一条指令。PLT[0]比较特殊，它的功能就是解析出函数puts的真正地址并将该地址写到GOT[n][7]，经过这样的一番折腾后，call puts@plt就转化成了对puts函数的调用。后续如果再调用puts函数，则不需要再经过dynamic loader来解析地址了，因为第一次解析完成后，puts函数的地址已经写到GOT[n]，直接取出该地址调用即可，如图6所示。

```
1
2  libdatanode.so:     file format elf32-i386
3
4  Disassembly of section .plt:
5
6  00000520 <__gmon_start__@plt-0x10>:
7   520:   ff b3 04 00 00 00       pushl  0x4(%ebx)
8   526:   ff a3 08 00 00 00       jmp    *0x8(%ebx)
9   52c:   00 00                   add    %al,(%eax)
10         ...
11
12 00000550 <_ZN8DataNodeC1Ev@plt>:
13  550:   ff a3 14 00 00 00       jmp    *0x14(%ebx)
14  556:   68 10 00 00 00          push   $0x10
15  55b:   e9 c0 ff ff ff          jmp    520 <_init+0x2c>
16
17 00000560 <_Znwj@plt>:
18  560:   ff a3 18 00 00 00       jmp    *0x18(%ebx)
19  566:   68 18 00 00 00          push   $0x18
20  56b:   e9 b0 ff ff ff          jmp    520 <_init+0x2c>
21
22 00000570 <puts@plt>:
23  570:   ff a3 1c 00 00 00       jmp    *0x1c(%ebx)
24  576:   68 20 00 00 00          push   $0x20
25  57b:   e9 a0 ff ff ff          jmp    520 <_init+0x2c>
```

代码7: PLT

前面说了，PLT是为了解决动态库的代码段不能在进程间共享引入的，那说明在未引入这个概念前，动态库也是可以工作的，只是工作方式略有不同。在编译libdatanode.so时，不指定-fPIC参数编译出来的lib库即是以进程间不能共享代码段这种方式工作的。

PIC，Position Independent Code的缩写，位置无关代码，即带了这个参数编译出来的动态库，被加载到进程的任何地址段都是可以正常工作的。

代码8是代码4不带-fPIC参数编译出来的动态库libdatanode.so中getNode函数的汇

[7]因为是从puts函数的PLT跳转过去的，所以解析的是puts函数的地址，同理如果是从new函数的PLT跳转过去的，那解析出来的就是new的地址。

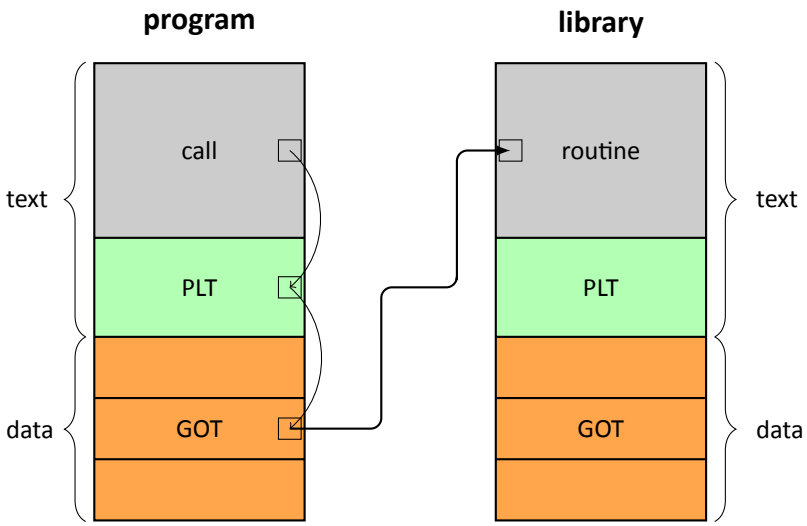

图 6: Loading

编代码，其中第 16 行即是调用 puts 函数[8]。可以看出，这里的调用是直接 call 一个地址 6ba，而不再是 call puts@plt 了。很明显，main 加载 libdatanode.so 后，6ba 不可能是 puts 函数的地址，main 在加载 libdatanode.so 之后，需要对 call 6ba 做重定位，使 call 后面的地址是真正 puts 函数的地址，如图7所示。这就是为什么不加 -fPIC 选项编译出的动态库不能进程间共享的原因：一个进程加载了动态库，需要对动态库中的某些符号做重定位，会修改该动态库的代码段；而每个进程对动态库中的符号重定位的地址是不相同的，导致各个进程间不能共享同一个动态库的代码段。但加上 -fPIC 参数后，就有了 PLT 和 GOT，有了这两个表后，解析函数的过程中修改的是各个进程的数据段[9]，而不是动态库的代码段，所以多个进程是可以共享一份代码的。

```
1  0000068b <_Z7getNodev>:
2   68b:   55                      push   %ebp
3   68c:   89 e5                   mov    %esp,%ebp
4   68e:   56                      push   %esi
5   68f:   53                      push   %ebx
6   690:   83 ec 20                sub    $0x20,%esp
7   693:   c7 04 24 08 00 00 00    movl   $0x8,(%esp)
8   69a:   e8 fc ff ff ff          call   69b <_Z7getNodev+0x10>
9   69f:   89 c3                   mov    %eax,%ebx
10  6a1:   89 1c 24                mov    %ebx,(%esp)
11  6a4:   e8 fc ff ff ff          call   6a5 <_Z7getNodev+0x1a>
12  6a9:   89 5d f4                mov    %ebx,-0xc(%ebp)
13  6ac:   83 7d f4 00             cmpl   $0x0,-0xc(%ebp)
```

[8]从这段代码很难直接的看出来这里就是调用 puts 函数，运行二进制 main 加载 libdatanode.so 再通过 gdb 反汇编 getNode 函数就可以很清晰的看出来。

[9]数据段本来各个进程就是独立的。

```
6b0:   75 13                   jne    6c5 <_Z7getNodev+0x3a>
6b2:   c7 04 24 fc 06 00 00    movl   $0x6fc,(%esp)
6b9:   e8 fc ff ff ff          call   6ba <_Z7getNodev+0x2f>
6be:   b8 00 00 00 00          mov    $0x0,%eax
6c3:   eb 19                   jmp    6de <_Z7getNodev+0x53>
6c5:   8b 45 f4                mov    -0xc(%ebp),%eax
6c8:   eb 14                   jmp    6de <_Z7getNodev+0x53>
6ca:   89 c6                   mov    %eax,%esi
6cc:   89 1c 24                mov    %ebx,(%esp)
6cf:   e8 fc ff ff ff          call   6d0 <_Z7getNodev+0x45>
6d4:   89 f0                   mov    %esi,%eax
6d6:   89 04 24                mov    %eax,(%esp)
6d9:   e8 fc ff ff ff          call   6da <_Z7getNodev+0x4f>
6de:   83 c4 20                add    $0x20,%esp
6e1:   5b                      pop    %ebx
6e2:   5e                      pop    %esi
6e3:   5d                      pop    %ebp
6e4:   c3                      ret
```

代码 8: load time relocation

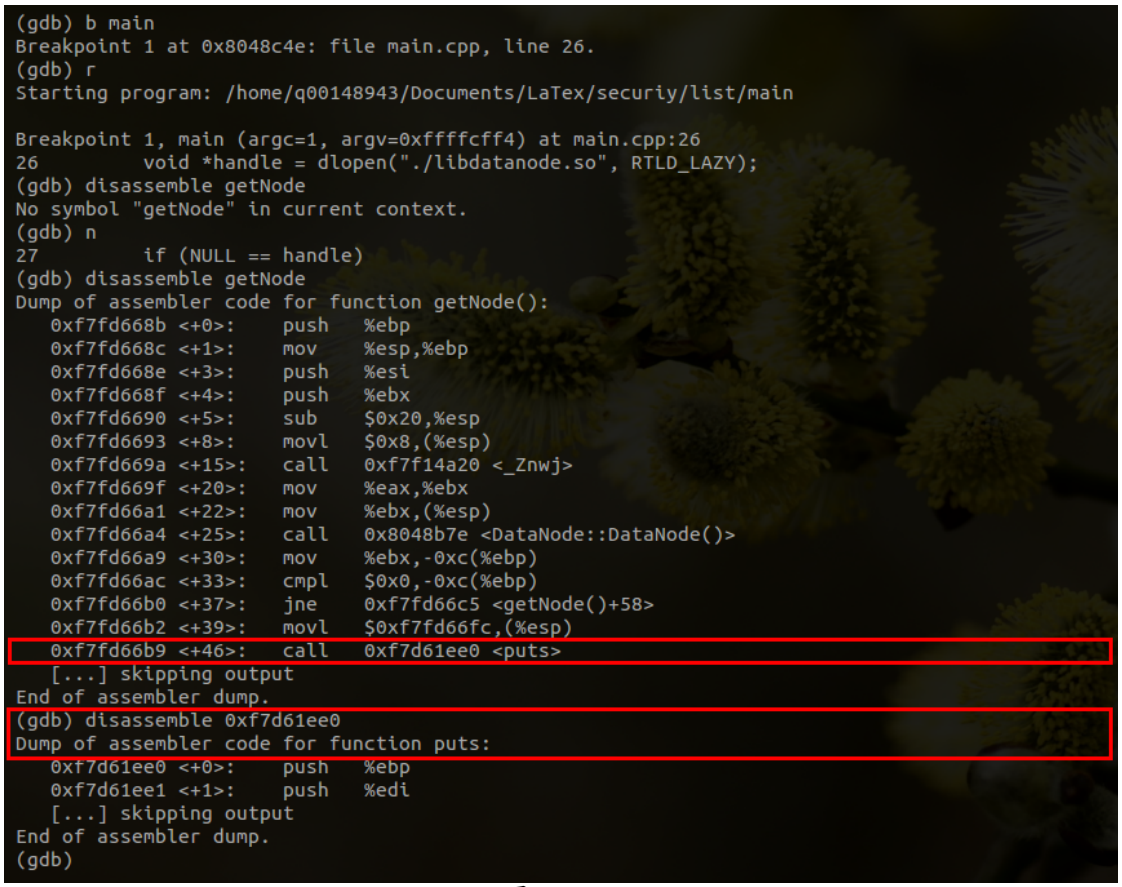

图 7: gdb

图8是使用带 -fPIC 参数编译出的动态库同时被两个进程加载的内存布局示意图。图9是不带 -fPIC 参数编译出的动态库同时被两个进程加载的内存布局示意图。从两图的

对比中可以看出，不带-fPIC参数的动态库，不同的进程加载后，需要映射到不同的物理内存，即进程间不能共享同一物理内存中的该动态库。

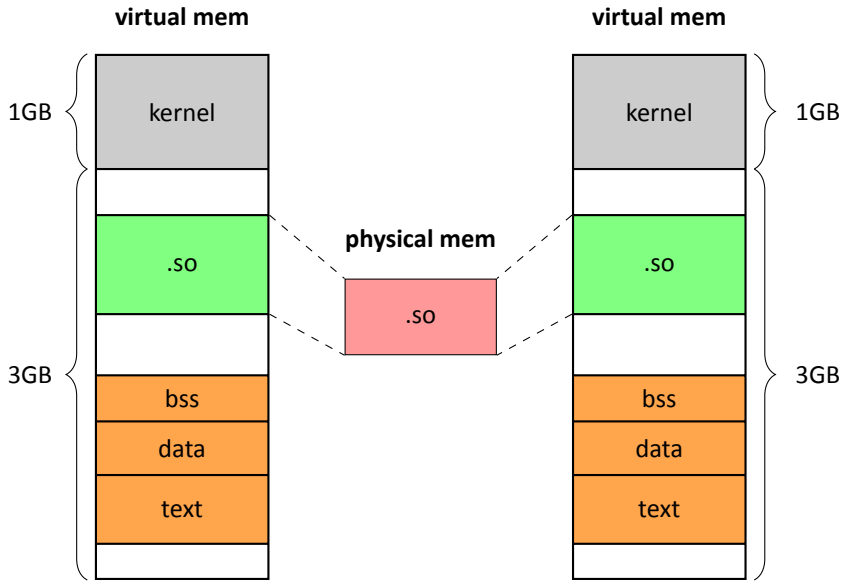

图 8: position independent code

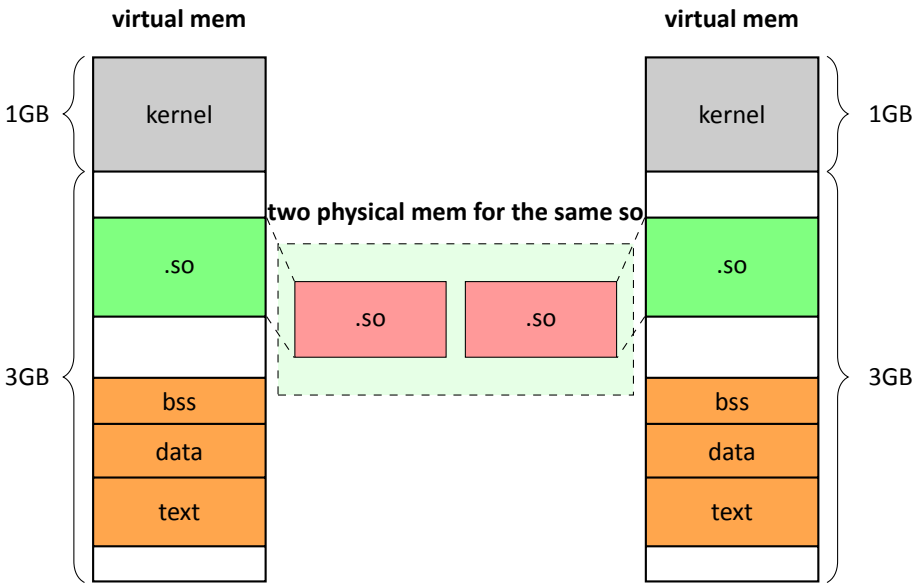

图 9: load time relocation

4 问题改进

前面章节说了，如果代码3在编译时不加rdynamic选项，二进制main在打开动态库时会报undefined reference，导致整个程序运行失败。那问题来了，PGW也给lib库提供

了很多接口，在编译PGW时并没有带rdynamic参数，那么在PGW加载lib库时，这些提供给lib库的接口函数是如何解析成功的。

PGW当前的处理方式是：每新增一个lib库接口，就需要FE的lib库提供一个注册函数来注册这个接口，以这种方式告诉FE的lib库，PGW提供函数A，原型是prototypeA，地址是0xABCDEF。FE的lib库拿到这个地址后，直接强转成protoypeA类型的函数来使用。这种方式的一个最大好处就是效率高，因为FE的lib库拿到这个地址后，直接强转就可以使用，不需要经过多条指令来查找A函数的地址。但也有一个坏处就是，PGW每开放一个lib库接口，就需要FE的lib库对应的开发一个注册接口，同时PGW在初始化FE的lib库时，需要找到FE的该注册接口，并调用该接口将新开放的lib库接口注册给FE的lib库，绕晕没:)。

PGW初始化FE的lib库的代码有1K，其中有0.5K就是在找FE lib库中的注册函数来注册提供给FE的lib库接口的。在160版本引入了虚基类，以虚函数的方式来封装提供给FE的lib库接口，有效的遏制了初始化lib库的代码继续膨胀，同时也减轻了FE使用lib库接口的负担；但这种方式相比第一种，性能略有下降，因为每次函数调用都得先到虚函数表中查找接口函数的地址，然后才能真正调用这些接口，同时对各个版本的兼容性有了更高的要求，因为一旦虚函数表不匹配，很容易导致PGW进程重启。

还有一种方式就是编译PGW时加上rdynamic选项，这样PGW提供给FE的lib库接口，只要把相应的头文件发布给FE就可以了，不需要FE开发注册接口，FE使用这些接口，就和我们使用VPP组件的接口一样，看了函数原型就可以直接拿来用，这样对FE的开发人员来说也更“人性化”一些。这种方式的性能和使用虚函数是相当的，但对兼容性的要求没有使用虚函数高。唯一的不足就是加了rdynamic编译选项后，编译出的PGW二进制会稍大，运行时占用内存稍多，因为要加载动态符号表。在目前PGW二进制编译出来32M的情况下，加了该选项后，编译出的二进制在38M左右。