APPENDIX A

Hands-on Analysis of Linker-Loader Coordination

The purpose of this appendix is to try to demonstrate the practical techniques which can be used to analyze the linker-loader coordination effects. Even though this particular topic is outside the domain of practicalities for most programmers, the techniques that will be explained shortly often come in handy in real-world scenarios.

Overall Objectives

The essential problem in dealing with dynamic libraries is the process of resolving the references of the symbols located in the dynamic libraries. For the most part, the "symbols" mean the "functions," especially in Scenario 1A and 1B. Additionally, in Scenario 2, the dynamic library needs to resolve the access to its non-local variables as well.

Prerequisite Skills

Before reading through this section of the book, the reader is strongly advised to reread Chapters 9 and 10, as these chapters contain detailed descriptions of simple drills, and mastering these drills will help you understand this material. In particular, the following skills are important for the proper understanding of the discussion that will follow shortly:

Binary Files Analysis:

- Disassembling the binary file sections (primarily code section, but the other sections as well)

Runtime Analysis:

- Determining the library loading address range
- Disassembling the running process by using the debugger

- Using the debugger to walk/step through the code
- Using the debugger to examine the data variables' values

Overview of Illustrated Cases

The illustrative hands-on examples will focus on the following typical scenarios:

- Scenario 1A: Application calling a dynamic library function

  The implementation of Scenario 1A in applications follows the PIC scenario as the predominant real-world choice.

- Scenario 1B: Dynamic library calling another dynamic library

  Equal attention is paid to both the PIC and the LTR cases.

- Scenario 2: Dynamic library tries to resolve its own symbols

  Again, both the PIC and the LTR cases are analyzed in details.

64-bit OS Scenario

The implementation details specific to the 64-bit architecture are explained for LTR (required `-mcmodel=large` compiler flag) vs. PIC implementation (the impact of the relative instruction pointer mode (RIP) is illustrated).

Scenario 1A: App Calling Dynamic Library Function

In this section, I will analyze in detail the simplest case in which an application references the dynamic library's symbol. The linker and loader are to establish the connection between the code residing at the fixed location in the process memory map (such as the app's `main()` function) and the symbols residing at the "moving target" (the dynamic library whose loading address is not known up front and will be determined by the loader); see Figure A-1.

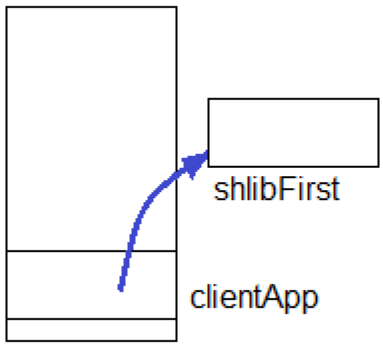

Figure A-1.

Demo Project Source Code

In order to illustrate Scenario 1A (when the application calls the dynamic library function), Listing A-1 contains a simple project.

Listing A-1. Please Add Caption

```
shlibFirst:
file: shlibexports.h
#pragma once

int shlib_function(void);

file: shlib.c
#include "shlibexports.h"

int shlib_function(void)
{
    return 10;
}

file: build.sh
gcc -Wall -g -O0 -fPIC -c shlib.c
gcc -shared shlib.o -Wl,-soname,libfirst.so.1 -o libfirst.so.1.0.0

clientApp:
file: main.c

#include <stdio.h>
#include "shlibexports.h"

int main(int argc, char* argv[])
{
```

```
    int nRetValue = shlib_function();
    // purposefully calling second time
    nRetValue    += shlib_function();
    return nRetValue;
}

file: build.sh
gcc -Wall -g -O0 -I../shlibFirst -c main.c
gcc main.o -L../shlibFirst -lfirst -Wl,-R../shlibFirst -o clientApp
```

In order to make the example extremely simple, the bare minimum of the code is used; even the `printf` statements have been eliminated to limit the linking requirements to the theoretical minimum. Note that in order to demonstrate the properties of the lazy binding mechanism, the application is purposefully designed to call the same dynamic library function twice.

Relocation Information Analysis

It is obvious that in this example the application needs to resolve the call to the function `shlib_function()`, which resides in `shlibfirst.so`. In order to figure out how this problem becomes resolved, take a closer look at the relocation information that the linker embedded into the clientApp binary, shown in Figure A-2.

```
milan@milan$ readelf -r clientApp

Relocation section '.rel.dyn' at offset 0x378 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff0  00000206 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x380 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   shlib_function
0804a004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a008  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main
milan@milan$
```

Figure A-2.

Obviously, the linker found it important to pass the directive to the loader to take care of the `shlib_function` symbol. A very interesting detail, however, is the offset at which the loader is expected to operate in order to help resolve the symbol at runtime. According to the clientApp sections table, the address 0x804a000 belongs to the `.got.plt` section, which resides between the addresses 0x8049ff4 and 0x804a00c, as shown in Figure A-3.

```
milan@milan$ readelf --sections clientApp
There are 36 section headers, starting at offset 0x1420:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
                                  o
                                  o
                                  o
  [23] .got.plt          PROGBITS        08049ff4 000ff4 000018 04  WA  0   0  4
  [24] .data             PROGBITS        0804a00c 00100c 000008 00  WA  0   0  4
  [25] .bss              NOBITS          0804a014 001014 000008 00  WA  0   0  4
                                  o
                                  o
                                  o
```

Figure A-3.

Disassembling the Binary Files

Another useful piece of information can be obtained by disassembling the clientApp binary file, especially the contents of the `main()` function. The disassembled code shows that the actual call to the `shlib_function` is not implemented directly, but instead through the call to `shlib_function@plt()`, as shown in Figure A-4.

```
milan@milan$ objdump -d -S -M intel clientApp | grep -A 25 "<main>:"
080484c4 <main>:
#include <stdio.h>
#include "shlibexports.h"

int main(int argc, char* argv[])
{
 80484c4:       55                      push   ebp
 80484c5:       89 e5                   mov    ebp,esp
 80484c7:       83 e4 f0                and    esp,0xfffffff0
 80484ca:       83 ec 10                sub    esp,0x10
    int nRetValue = shlib_function();
 80484cd:       e8 0e ff ff ff          call   80483e0 <shlib_function@plt>
 80484d2:       89 44 24 0c             mov    DWORD PTR [esp+0xc],eax

    // purposefully making another call
    // to the same function
    nRetValue    += shlib_function();
 80484d6:       e8 05 ff ff ff          call   80483e0 <shlib_function@plt>
 80484db:       01 44 24 0c             add    DWORD PTR [esp+0xc],eax
    return nRetValue;
 80484df:       8b 44 24 0c             mov    eax,DWORD PTR [esp+0xc]
}
 80484e3:       c9                      leave
 80484e4:       c3                      ret
 80484e5:       90                      nop
 80484e6:       90                      nop
```

Figure A-4.

As a rule, functions with the suffix "@plt" are automatically generated by the compiler to aid the implementation of the PIC concept. Examination of the disassembled code and a closer look at the clientApp's section layout shows that several "@plt" functions reside in the dedicated `.plt` section. In fact, direct disassembling of the `.plt` section may provide a nice view of the `shlib_function@plt` implementation details, after which pieces of puzzle start to fall in place (see Figure A-5).

```
milan@milan$ objdump -d -j .plt clientApp

clientApp:     file format elf32-i386


Disassembly of section .plt:

080483d0 <shlib_function@plt-0x10>:
 80483d0:       ff 35 f8 9f 04 08       pushl  0x8049ff8
 80483d6:       ff 25 fc 9f 04 08       jmp    *0x8049ffc
 80483dc:       00 00                   add    %al,(%eax)
        ...

080483e0 <shlib_function@plt>:
 80483e0:       ff 25 00 a0 04 08       jmp    *0x804a000
 80483e6:       68 00 00 00 00          push   $0x0
 80483eb:       e9 e0 ff ff ff          jmp    80483d0 <_init+0x38>

080483f0 <__gmon_start__@plt>:
 80483f0:       ff 25 04 a0 04 08       jmp    *0x804a004
 80483f6:       68 08 00 00 00          push   $0x8
 80483fb:       e9 d0 ff ff ff          jmp    80483d0 <_init+0x38>

08048400 <__libc_start_main@plt>:
 8048400:       ff 25 08 a0 04 08       jmp    *0x804a008
 8048406:       68 10 00 00 00          push   $0x10
 804840b:       e9 c0 ff ff ff          jmp    80483d0 <_init+0x38>
milan@milan$
```

Figure A-5.

Interestingly, the implementation of `shlib_function@plt` provides the connection with the address 0x804a000, which was previously implicated in the relocation section in the linker's directive to the loader. More specifically, you can see that the first instruction to `shlib_function@plt` in fact implements the jump to the address in program memory pointed to by the value of the memory location 0x804a000 residing in the `.got.plt` section. The whole scheme resembles the mechanical model shown in Figure A-6.

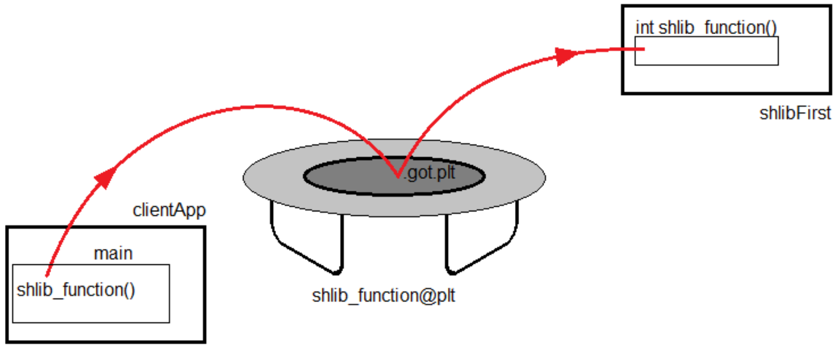

Figure A-6.

Runtime Analysis

The facts established so far are within the range of our expectations. The extra level of indirection featured by the PIC concept appears to be in place. If the loader correctly applies the directives passed by the linker, resolving the ultimate address of the jump should result in landing directly into the **`shlib_function()`** code. In order to provide the ultimate proof that the PIC scheme works in the Scenario 1A circumstances, a detailed runtime analysis must take place. As you will see, the concept of lazy binding adds an extra notch to the complexity of the PIC implementation.

The initial few steps of debugging the clientApp show what you would expect (see Figure A-7).

```
(gdb) break main
Breakpoint 1 at 0x80484cd: file main.c, line 6.
(gdb) run
Starting program: /home/milan/clientApp

Breakpoint 1, main (argc=1, argv=0xbffff304) at main.c:6
6           int nRetValue = shlib_function();
(gdb) set disassembly-flavor intel
(gdb) disassemble /m
Dump of assembler code for function main:
5       {
   0x080484c4 <+0>:     push   ebp
   0x080484c5 <+1>:     mov    ebp,esp
   0x080484c7 <+3>:     and    esp,0xfffffff0
   0x080484ca <+6>:     sub    esp,0x10

6           int nRetValue = shlib_function();
=> 0x080484cd <+9>:     call   0x80483e0 <shlib_function@plt>
   0x080484d2 <+14>:    mov    DWORD PTR [esp+0xc],eax

7
8           // purposefully making another call
9           // to the same function
10          nRetValue    += shlib_function();
   0x080484d6 <+18>:    call   0x80483e0 <shlib_function@plt>
   0x080484db <+23>:    add    DWORD PTR [esp+0xc],eax

11          return nRetValue;
   0x080484df <+27>:    mov    eax,DWORD PTR [esp+0xc]

12      }
   0x080484e3 <+31>:    leave
   0x080484e4 <+32>:    ret

End of assembler dump.
```

Figure A-7.

Stepping into the `shlib_function@plt` provides you the chance to examine the value of the variable at the address 0x804a000 (see Figure A-8).

```
(gdb) stepi
0x080483e0 in shlib_function@plt ()
(gdb) disassemble /m
Dump of assembler code for function shlib_function@plt:
=> 0x080483e0 <+0>:     jmp    DWORD PTR ds:0x804a000
   0x080483e6 <+6>:     push   0x0
   0x080483eb <+11>:    jmp    0x80483d0
End of assembler dump.
(gdb) display /x *0x804a000
1: /x *0x804a000 = 0x80483e6
(gdb) stepi
0x080483e6 in shlib_function@plt ()
1: /x *0x804a000 = 0x80483e6
(gdb) stepi
0x080483eb in shlib_function@plt ()
1: /x *0x804a000 = 0x80483e6
(gdb) stepi
0x080483d0 in ?? ()
1: /x *0x804a000 = 0x80483e6
(gdb) stepi
0x080483d6 in ?? ()
1: /x *0x804a000 = 0x80483e6
(gdb) stepi
0xb7ff26a0 in ?? () from /lib/ld-linux.so.2    we are inside the loader code
1: /x *0x804a000 = 0x80483e6
(gdb) step
Cannot find bounds of current function
(gdb) finish
Run till exit from #0  0xb7ff26a0 in ?? () from /lib/ld-linux.so.2
0x080484d2 in main (argc=1, argv=0xbffff304) at main.c:6
6            int nRetValue = shlib_function();
1: /x *0x804a000 = 0xb7fd842c    <-----------    after which the .got.plt address
(gdb)                                            gets initialized correcly
```

Figure A-8.

Surprisingly, when you arrived for the first time at the `shlib_function@plt` code, the variable at the address 0x804a000 does not point to the expected address. Instead, it points to the one instruction below, at the address 0x80483e6.

By following where that route goes, you find out that on this very first occasion the code actually dives into the depths of the loader code, where the actual initialization of the `.got.plt` variable happens. In fact, in this first run you've just witnessed the mechanism of lazy binding in action. The very first time, the code flow took a bit different route than usual, during which the address of the `.got.plt` variable got updated by the loader according to the linker's directives. Figure A-9 illustrates what actually happened.

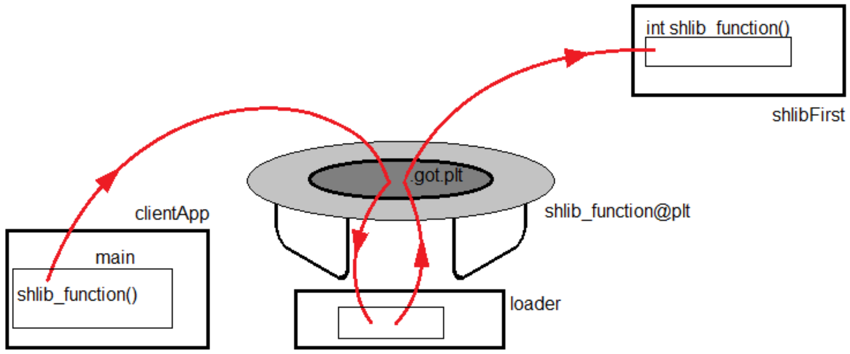

Figure A-9.

In order to verify that the lazy binding happens only the first time and never again thereafter, you can run your example again. This time you will not be diving as deep into the assembly code. Instead, you will watch the execution progress from the C language level, while at the same time printing the value of the variable at the address 0x804a000 (the `.got.plt` trampoline destination address), as shown in Figure A-10.

```
Starting program: /home/milan/clientApp/clientApp

Breakpoint 1, main (argc=1, argv=0xbffff304) at main.c:6
6           int nRetValue = shlib_function();
(gdb) l
1       #include <stdio.h>
2       #include "shlibexports.h"
3
4       int main(int argc, char* argv[])
5       {
6           int nRetValue = shlib_function();
7
8           // purposefully making another call
9           // to the same function
10          nRetValue    += shlib_function();
(gdb) display/x *0x804a000
1: /x *0x804a000 = 0x80483e6  <-------- before lazy binding
(gdb) next
10          nRetValue    += shlib_function();
1: /x *0x804a000 = 0xb7fd842c <-------- after lazy binding...
(gdb) next
11          return nRetValue;
1: /x *0x804a000 = 0xb7fd842c <-------- ... and every time thereafter
(gdb) l
6           int nRetValue = shlib_function();
7
8           // purposefully making another call
9           // to the same function
10          nRetValue    += shlib_function();
11          return nRetValue;
12      }
13
(gdb)
```

Figure A-10.

The .got.plt Variable

Although it's great that the **.got.plt** variable ultimately gets initialized properly, the open question is where its value actually points to. If everything is as expected, the value of the **.got.plt** variable at address 0x804a000 should point to the inside part of the process memory map where the shared library gets mapped. To verify that assumption, you should first try to find out the address range at which the dynamic library gets loaded. Since you are running the clientApp through the debugger, you have a chance to take a look at the clientApp process memory map while the debugger is waiting for the next step, as shown in Figure A-11.

```
milan@milan$ ps -ef | grep clientApp
milan    23945 23707  0 09:55 pts/0    00:00:00 gdb -q clientApp
milan    23947 23945  0 09:55 pts/0    00:00:00 /home/milan/clientApp/clientApp
milan    24045 23950  0 09:59 pts/1    00:00:00 grep --color=auto clientApp
milan@milan$ cat /proc/23947/maps
08048000-08049000 r-xp 00000000 08:01 2885321    /home/milan/clientApp
08049000-0804a000 r--p 00000000 08:01 2885321    /home/milan/clientApp
0804a000-0804b000 rw-p 00001000 08:01 2885321    /home/milan/clientApp
b7e1a000-b7e1c000 rw-p 00000000 00:00 0
b7e1c000-b7fc0000 r-xp 00000000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc0000-b7fc2000 r--p 001a4000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc2000-b7fc3000 rw-p 001a6000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc6000 rw-p 00000000 00:00 0
b7fd8000-b7fd9000 r-xp 00000000 08:01 2885318    /home/milan/shlibFirst/libfirst.so.1.0.0
b7fd9000-b7fda000 r--p 00000000 08:01 2885318    /home/milan/shlibFirst/libfirst.so.1.0.0
b7fda000-b7fdb000 rw-p 00001000 08:01 2885318    /home/milan/shlibFirst/libfirst.so.1.0.0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
milan@milan$
```

Figure A-11.

Obviously, the **shlibFirst** got loaded at the address range starting at the address 0xb7fd8000. By the same token, this is the start address of the **shlibFirst**'s code (**.text**) segment. Nice, but where exactly does the **shlib_function()** code reside? Disassembling the **shlibfirst.so** binary file reveals at which offset in the **.text** section the **shlib_function()** resides, as shown in Figure A-12.

```
milan@milan$ objdump -d -S -M intel libfirst.so | grep -A 7 shlib_function
0000042c <shlib_function>:
 42c:   55                      push   ebp
 42d:   89 e5                   mov    ebp,esp
 42f:   b8 0a 00 00 00          mov    eax,0xa
 434:   5d                      pop    ebp
 435:   c3                      ret
 436:   90                      nop
 437:   90                      nop
milan@milan$
```

Figure A-12.

By adding the start of `.text` segment (0xb7fd8000) to the `shlib_function()` offset (0x42c), you actually get the expected value of 0xb7fd842c. Obviously, the resolved address mechanism works perfectly!

Scenario1B: Dynamic Library Calling Another Dynamic Library's Function

In the previous section, I analyzed in detail the case when the application references the dynamic library's symbol. In that particular scenario you saw how the linker and the loader tried to establish the connection between the code residing at the fixed location in the process memory map (the app's main function) and the symbols residing at the moving target (the dynamic library whose loading address is not known up front and will be determined by the loader).

A more general case of the same problem happens when a dynamic library references another dynamic library's symbol, regardless of where in the chain of loading the dynamic libraries the interaction between the two libraries occurs. This particular scenario may be described as the linker and loader trying to establish a connection between the code in one moving target (a dynamic library) and the symbols residing in another moving target (another dynamic library loaded by the first dynamic library), as shown in Figure A-13.

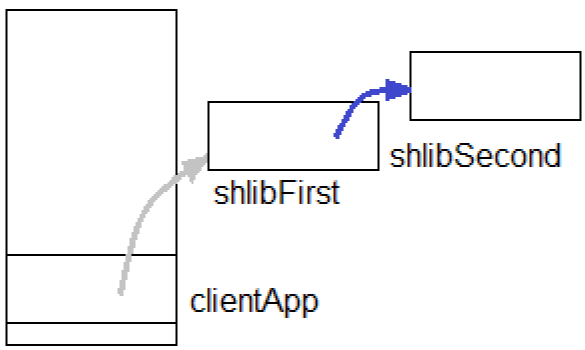

Figure A-13.

As will be demonstrated shortly, the actual implementation of resolving the dynamic symbols varies depending on whether the dynamic library (`shlibFirst` in this example) has been compiled with the `-fPIC` compiler flag or not—in other words, whether the position-independent code (PIC) or the load-time relocation (LTR) approach has been chosen. The major focus in my analyses will be the interior of `shlibFirst`, as we are mostly interested in seeing how it resolves the exact location of the another dynamic library's symbols it needs.

Position-Independent Code (PIC) Case

The way the PIC concept solves Scenario 1B’s problem is almost completely identical to how it was done in the case of application calling the dynamic library's symbol. The differences stemming from the fact that in this case both dynamic libraries are moving targets will impact only certain elements of the design. A special effort in this case will be taken in order to determine the exact location of the starting point (from which address the (call) instruction tries to make the calculated jump to another function).

Demo Project Source Code

In order to illustrate Scenario 1B (the case when the application calls the dynamic library function), Listing A-2 contains the code for a simple project.

Listing A-2. Please Add Caption

```
shlibSecond:
file: secondshlibexports.h
#pragma once

int second_shlib_function(void);

file: shlib.c
#include "secondshlibexports.h"

int second_shlib_function(void)
{
    return 10;
}

file: build.sh
gcc -Wall -g -O0 -fPIC -c shlib.c
gcc -shared shlib.o -Wl,-soname,libsecond.so.1 -o libsecond.so.1.0.0

shlibFirst:
file: shlibexports.h
#pragma once

int shlib_function(void);

file: shlib.c
#include "shlibexports.h"
#include "secondshlibexports.h"

int shlib_function(void)
{
    int nRetValue = second_shlib_function();

    // purposefully calling second time
    nRetValue    += second_shlib_function();
    return nRetValue;
}

file: build.sh
gcc -Wall -g -O0 -fPIC -I../shlibSecond -c shlib.c
gcc -shared shlib.o -L../shlibSecond -lsecond -Wl,-R../shlibSecond \
    -Wl,-soname,libfirst.so.1 -o libfirst.so.1.0.0
```

```
clientApp:
file: main.c

#include <stdio.h>
#include "shlibexports.h"

int main(int argc, char* argv[])
{
    int nRetValue = shlib_function();
    // purposefully calling second time
    nRetValue    += shlib_function();
    return nRetValue;
}

file: build.sh
gcc -Wall -g -O0 -I../shlibFirst -c main.c
gcc main.o -L../shlibFirst -lfirst -Wl,-R../shlibFirst -o clientApp
```

In order to make the example extremely simple, the bare minimum of the code is used; even the `printf` statements have been eliminated to limit the linking requirements to the theoretical minimum. In order to demonstrate the properties of the lazy binding mechanism, `shlibFirst` is purposefully designed to call the same `shlibSecond`'s function twice.

Relocation Information Analysis

The usual first step in the analysis is to examine the dynamic library relocation information. As expected, you see that the linker has indicated to the loader that `libfirst.so` requires the resolved address of `second_shlib_function()`, as shown in Figure A-14.

```
milan@milan$ readelf -r shlibFirst/libfirst.so

Relocation section '.rel.dyn' at offset 0x30c contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000200c  00000008 R_386_RELATIVE
00001fe8  00000106 R_386_GLOB_DAT    00000000   __cxa_finalize
00001fec  00000206 R_386_GLOB_DAT    00000000   __gmon_start__
00001ff0  00000306 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses

Relocation section '.rel.plt' at offset 0x32c contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
00002008  00000407 R_386_JUMP_SLOT   00000000   second_shlib_function
milan@milan$
```

Figure A-14.

The address offset related to the **second_shlib_function()** is indicated to have the value of 0x2008. Based on the analysis of the **libfirst.so** section layout, this particular address belongs to the **.got.plt** section, which covers the address range between 0x1ff4 and 0x200c, as you can see in Figure A-15.

```
milan@milan$ readelf --sections libfirst.so
There are 33 section headers, starting at offset 0x132c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
                                    o
                                    o
                                    o
  [11] .text             PROGBITS        000003c0 0003c0 000128 00  AX  0   0 16
  [12] .fini             PROGBITS        000004e8 0004e8 00001a 00  AX  0   0  4
  [13] .eh_frame_hdr     PROGBITS        00000504 000504 00001c 00   A  0   0  4
  [14] .eh_frame         PROGBITS        00000520 000520 000064 00   A  0   0  4
  [15] .ctors            PROGBITS        00001efc 000efc 000008 00  WA  0   0  4
  [16] .dtors            PROGBITS        00001f04 000f04 000008 00  WA  0   0  4
  [17] .jcr              PROGBITS        00001f0c 000f0c 000004 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         00001f10 000f10 0000d8 08  WA  4   0  4
  [19] .got              PROGBITS        00001fe8 000fe8 00000c 04  WA  0   0  4
  [20] .got.plt          PROGBITS        00001ff4 000ff4 000018 04  WA  0   0  4
  [21] .data             PROGBITS        0000200c 00100c 000004 00  WA  0   0  4
                                    o
                                    o
                                    o
```

Figure A-15.

Disassembling the Binary Files

Another useful piece of information can be obtained by disassembling the **libfirst.so** binary file, especially the contents of the **shlib_function()** function. The disassembled code shows that the actual call to the **second_shlib_function()** is not implemented directly, but instead through the call to **second_shlib_function@plt()**, as shown in Figure A-16.

```
milan@milan$ objdump -d -S -M intel libfirst.so | grep -A 27 "<shlib_function>:"
0000047c <shlib_function>:
#include "shlibexports.h"
#include "secondshlibexports.h"

int shlib_function(void)
{
 47c:   55                      push   ebp
 47d:   89 e5                   mov    ebp,esp
 47f:   53                      push   ebx
 480:   83 ec 14                sub    esp,0x14
 483:   e8 ef ff ff ff          call   477 <__i686.get_pc_thunk.bx>
 488:   81 c3 6c 1b 00 00       add    ebx,0x1b6c
        int nRetValue = second_shlib_function();
 48e:   e8 1d ff ff ff          call   3b0 <second_shlib_function@plt>
 493:   89 45 f4                mov    DWORD PTR [ebp-0xc],eax

        // purposefully calling second time
        nRetValue    += second_shlib_function();
 496:   e8 15 ff ff ff          call   3b0 <second_shlib_function@plt>
 49b:   01 45 f4                add    DWORD PTR [ebp-0xc],eax
    return nRetValue;
 49e:   8b 45 f4                mov    eax,DWORD PTR [ebp-0xc]
}
 4a1:   83 c4 14                add    esp,0x14
 4a4:   5b                      pop    ebx
 4a5:   5d                      pop    ebp
 4a6:   c3                      ret
 4a7:   90                      nop
milan@milan$
```

Figure A-16.

After studying the previous example, this is exactly what you should expect to see. The examination of the disassembled code and a closer look at the `libfirst.so` section layout shows that in fact several `"@plt"` functions reside in the dedicated `.plt` section.

A careful look at the `second_shlib_function@plt()` implementation reveals that it is implemented a bit differently than when the application calls a shared library symbol. This is exactly the place where the moving target-to-moving target paradigm impacts the PIC implementation. The first difference is that the function body features the call to the function `__i686.get_pc_thunk.bx()`, whose true meaning will be explained shortly. The second difference is that the `second_shlib_function@plt()` function makes a jump not based on the memory location address, but instead based on the value carried by the ebx register, as you can see in Figure A-17.

```
milan@milan$ objdump -d -S -M intel -j .plt libfirst.so

libfirst.so:     file format elf32-i386


Disassembly of section .plt:

00000380 <__cxa_finalize@plt-0x10>:
 380:   ff b3 04 00 00 00       push   DWORD PTR [ebx+0x4]
 386:   ff a3 08 00 00 00       jmp    DWORD PTR [ebx+0x8]
 38c:   00 00                   add    BYTE PTR [eax],al
        ...

00000390 <__cxa_finalize@plt>:
 390:   ff a3 0c 00 00 00       jmp    DWORD PTR [ebx+0xc]
 396:   68 00 00 00 00          push   0x0
 39b:   e9 e0 ff ff ff          jmp    380 <_init+0x3c>

000003a0 <__gmon_start__@plt>:
 3a0:   ff a3 10 00 00 00       jmp    DWORD PTR [ebx+0x10]
 3a6:   68 08 00 00 00          push   0x8
 3ab:   e9 d0 ff ff ff          jmp    380 <_init+0x3c>

000003b0 <second_shlib_function@plt>:
 3b0:   ff a3 14 00 00 00       jmp    DWORD PTR [ebx+0x14]
 3b6:   68 10 00 00 00          push   0x10
 3bb:   e9 c0 ff ff ff          jmp    380 <_init+0x3c>
milan@milan$
```

Figure A-17.

The Role of __i686.get_pc_thunk.bx() Function

The examination of the disassembled code of the `__i686.get_pc_thunk.bx()` function shows that the sole purpose of this function is to copy the contents of the stack pointer to the ebx register, as shown in Figure A-18.

```
milan@milan$ objdump -d -S -M intel libfirst.so | grep -A 5 "<__i686.get_pc_thunk.bx>:"
00000477 <__i686.get_pc_thunk.bx>:
 477:   8b 1c 24                mov    ebx,DWORD PTR [esp]
 47a:   c3                      ret
 47b:   90                      nop

0000047c <shlib_function>:
milan@milan$
```

Figure A-18.

This purpose of this maneuver, which may look a bit strange, is to capture the information about where exactly in the process memory map is the starting point from which you will call another function. Once obtained, this information is typically combined with the values of the fixed offsets to the other segments, which are all known to linker. In particular, if you go two figures back, this is exactly what the following snippet of code does:

```
483:   e8 ef ff ff ff          call   477 <__i686.get_pc_thunk.bx>
488:   81 c3 6c 1b 00 00       add    ebx,0x1b6c
```

The reason why the linker generated the code that increases the current program counter value (captured in ebx register) by the constant value 0x1b6c becomes clear in Figure A-19.

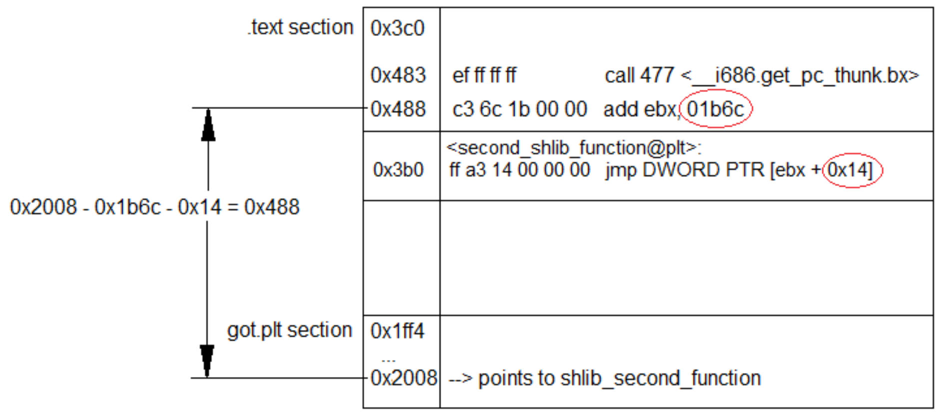

Figure A-19.

Obviously, the whole purpose of this linker trick is to calculate relative offset *from wherever the code ends up being loaded* to exactly the `.got.plt` variable to which the loader will stamp in the ultimate address of the function `second_shlib_function()`. Let's see how it works at runtime.

Runtime Analysis

The runtime analysis brings no particular surprises, as you can see in Figure A-20.

```
milan@milan$ gdb -q clientApp
Reading symbols from /home/milan/PIC/clientApp/clientApp...done.
(gdb) break shlib_function
Breakpoint 1 at 0x80483e0
(gdb) run
Starting program: /home/milan/PIC/clientApp/clientApp

Breakpoint 1, shlib_function () at shlib.c:6
6               int nRetValue = second_shlib_function();
(gdb) set disassembly-flavor intel
(gdb) disassemble /m
Dump of assembler code for function shlib_function:
5       {
   0xb7fd847c <+0>:     push   ebp
   0xb7fd847d <+1>:     mov    ebp,esp
   0xb7fd847f <+3>:     push   ebx
   0xb7fd8480 <+4>:     sub    esp,0x14
   0xb7fd8483 <+7>:     call   0xb7fd8477 <__i686.get_pc_thunk.bx>
   0xb7fd8488 <+12>:    add    ebx,0x1b6c

6               int nRetValue = second_shlib_function();
=> 0xb7fd848e <+18>:    call   0xb7fd83b0 <second_shlib_function@plt>
   0xb7fd8493 <+23>:    mov    DWORD PTR [ebp-0xc],eax

7
8               // purposefully calling second time
9               nRetValue    += second_shlib_function();
   0xb7fd8496 <+26>:    call   0xb7fd83b0 <second_shlib_function@plt>
   0xb7fd849b <+31>:    add    DWORD PTR [ebp-0xc],eax

10          return nRetValue;
   0xb7fd849e <+34>:    mov    eax,DWORD PTR [ebp-0xc]

11      }
   0xb7fd84a1 <+37>:    add    esp,0x14
   0xb7fd84a4 <+40>:    pop    ebx
   0xb7fd84a5 <+41>:    pop    ebp
   0xb7fd84a6 <+42>:    ret

End of assembler dump.
(gdb) stepi
0xb7fd83b0 in second_shlib_function@plt () from ../shlibFirst/libfirst.so.1
(gdb) disassemble /m
Dump of assembler code for function second_shlib_function@plt:
=> 0xb7fd83b0 <+0>:     jmp    DWORD PTR [ebx+0x14]
   0xb7fd83b6 <+6>:     push   0x10
   0xb7fd83bb <+11>:    jmp    0xb7fd8380
End of assembler dump.
```

```
(gdb) info register ebx
ebx            0xb7fd9ff4       -1208115212
(gdb) display /x *0xb7fda008
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) stepi
0xb7fd83b6 in second_shlib_function@plt () from ../shlibFirst/libfirst.so.1
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) stepi
0xb7fd83bb in second_shlib_function@plt () from ../shlibFirst/libfirst.so.1
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) stepi
0xb7fd8380 in ?? () from ../shlibFirst/libfirst.so.1
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) stepi
0xb7fd8386 in ?? () from ../shlibFirst/libfirst.so.1
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) stepi
0xb7ff26a0 in ?? () from /lib/ld-linux.so.2
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) stepi
0xb7ff26a1 in ?? () from /lib/ld-linux.so.2
1: /x *0xb7fda008 = 0xb7fd83b6
(gdb) step
Cannot find bounds of current function
(gdb) finish
Run till exit from #0  0xb7ff26a1 in ?? () from /lib/ld-linux.so.2
0xb7fd8493 in shlib_function () at shlib.c:6
6               int nRetValue = second_shlib_function();
1: /x *0xb7fda008 = 0xb7e1842c
(gdb) disassemble /m 0xb7e1842c
Dump of assembler code for function second_shlib_function:
4       {
   0xb7e1842c <+0>:     push   ebp
   0xb7e1842d <+1>:     mov    ebp,esp

5           return 10;
   0xb7e1842f <+3>:     mov    eax,0xa

6       }
   0xb7e18434 <+8>:     pop    ebp
   0xb7e18435 <+9>:     ret

End of assembler dump.
(gdb)
```

Figure A-20.

Obviously, the jump destination address you are interested in will be carried by the `.got.plt` variable at the address 0xb7fd9ff4 + 0x14 (as you jump to [ebx + 0x14]), which equals 0xb7fda008. Tracking down the contents of that location shows that the loader eventually (lazy binding going on) imprints the address 0xb7e1842c as the ultimate destination of the jump.

The .got.plt Variable

The runtime analysis of the dynamic libraries loading address shows that the jump destination is somewhere inside the `libsecond.so` code (`.text`) section, as you can see in Figure A-21.

```
milan@milan$ ps -ef | grep clientApp
milan     25345 24897  0 14:50 pts/0    00:00:00 gdb -q clientApp
milan     25347 25345  0 14:50 pts/0    00:00:00 /home/milan/PIC/clientApp/clientApp
milan     25408 25350  0 14:52 pts/1    00:00:00 grep --color=auto clientApp
milan@milan$ cat /proc/25347/maps
08048000-08049000 r-xp 00000000 08:01 2885327    /home/milan/PIC/clientApp/clientApp
08049000-0804a000 r--p 00000000 08:01 2885327    /home/milan/PIC/clientApp/clientApp
0804a000-0804b000 rw-p 00001000 08:01 2885327    /home/milan/PIC/clientApp/clientApp
b7e17000-b7e18000 rw-p 00000000 00:00 0
b7e18000-b7e19000 r-xp 00000000 08:01 2885310    /home/milan/PIC/shlibSecond/libsecond.so.1.0.0
b7e19000-b7e1a000 r--p 00000000 08:01 2885310    /home/milan/PIC/shlibSecond/libsecond.so.1.0.0
b7e1a000-b7e1b000 rw-p 00001000 08:01 2885310    /home/milan/PIC/shlibSecond/libsecond.so.1.0.0
b7e1b000-b7e1c000 rw-p 00000000 00:00 0
b7e1c000-b7fc0000 r-xp 00000000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc0000-b7fc2000 r--p 001a4000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc2000-b7fc3000 rw-p 001a6000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc6000 rw-p 00000000 00:00 0
b7fd8000-b7fd9000 r-xp 00000000 08:01 2885325    /home/milan/PIC/shlibFirst/libfirst.so.1.0.0
b7fd9000-b7fda000 r--p 00000000 08:01 2885325    /home/milan/PIC/shlibFirst/libfirst.so.1.0.0
b7fda000-b7fdb000 rw-p 00001000 08:01 2885325    /home/milan/PIC/shlibFirst/libfirst.so.1.0.0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
milan@milan$
```

Figure A-21.

The disassembling of the **libsecond.so** shows that the **second_shlib_function()** resides at the constant offset of 0x42c from the beginning of the **.text** section, as shown in Figure A-22.

```
milan@milan$ objdump -d -S -M intel libsecond.so | grep -A 12 "<second_shlib_function>:"
0000042c <second_shlib_function>:
#include "secondshlibexports.h"

int second_shlib_function(void)
{
 42c:   55                      push   ebp
 42d:   89 e5                   mov    ebp,esp
    return 10;
 42f:   b8 0a 00 00 00          mov    eax,0xa
}
 434:   5d                      pop    ebp
 435:   c3                      ret
 436:   90                      nop
milan@milan$
```

Figure A-22.

The combination of the two values (0xb7e18000 + 0x42c = 0xb7e1842c) exactly matches the jump destination address calculated by the linker. Obviously, the PIC scheme works fine!

The 64-bit Implementation Details

The 64-bit architecture made the implementation of PIC concept far more efficient by adding a new CPU addressing mode called the **relative instruction pointer mode** (**RIP**). In this addressing mode, the relative address passed as an operand to a CPU instruction is incremented by the instruction pointer of the next instruction. This addressing mode greatly simplifies the design by allowing the jump to the `.got` *and* `.got.plt` tables to be performed in a single instruction, which eliminates the need for the `__i686.get_pc_thunk.bx` function.

Figure A-23 illustrates how the RIP addressing mode makes the implementation of the `second_shlib_function@plt()` function more elegant by completely eliminating the need to call the `__i686.get_pc_thunk.bx()` function beforehand.

```
00000000000004f0 <second_shlib_function@plt>:
 4f0:   ff 25 0a 0b 20 00       jmp    QWORD PTR [rip+0x200b0a]
 4f6:   68 00 00 00 00          push   0x0
 4fb:   e9 e0 ff ff ff          jmp    4e0 <_init+0x18>
```

Figure A-23.

Load-Time Relocation (LTR) Case

If the dynamic library is not built with the `-fPIC` compiler flag, its resolutions of the dynamic library's symbol references will be implemented through the load-time relocation (LTR) approach. In this section, I will go through the details of how the LTR approach resolves Scenario 1B's kind of problem. The scenario that I will show here is identical to the previous case in which the application loads one dynamic library which in turn loads another dynamic library. The demo project that I will use to illustrate the Scenario 1B LTR case is almost completely identical to the PIC case in the previous section. The only difference is that the `-fPIC` compiler flag will *not* be used when building the `libfirst.so` dynamic library.

Relocation Information Analysis

The typical first step of taking a look at the relocation information reveals what you probably already expected, but with a special twist. The `second_shlib_function()` is registered by the linker as an item that requires the loader's intervention; however, the linker directive now appears in the `.rel.dyn` instead of the `.rel.plt` section, as was the case when PIC approach was used (see Figure A-24.

```
milan@milan$ readelf -r libfirst.so

Relocation section '.rel.dyn' at offset 0x30c contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002008  00000008 R_386_RELATIVE
00000473  00000402 R_386_PC32        00000000   second_shlib_function
00001fe8  00000106 R_386_GLOB_DAT    00000000   __cxa_finalize
00001fec  00000206 R_386_GLOB_DAT    00000000   __gmon_start__
00001ff0  00000306 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses

Relocation section '.rel.plt' at offset 0x334 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
milan@milan$
```

Figure A-24.

This little detail is not of paramount importance for the overall use case, but it wouldn't hurt to know this little difference, as it is one of the good indicators that your shared library is implementing the LTR approach.

The linker's directive specific to the `second_shlib_function()` mentions the offset of 0x473 at which the loader is supposed to take the corrective action in order for the whole scheme to work. The analysis of the linker section's layouts indicates that this particular offset belongs to the code (.`text`) section, as shown in Figure A-25.

```
milan@milan$ readelf --sections libfirst.so
There are 33 section headers, starting at offset 0x12f8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            00000114 000114 000024 00   A  0   0  4
  [ 2] .gnu.hash         GNU_HASH        00000138 000138 00003c 04   A  3   0  4
  [ 3] .dynsym           DYNSYM          00000174 000174 0000b0 10   A  4   1  4
  [ 4] .dynstr           STRTAB          00000224 000224 0000af 00   A  0   0  1
  [ 5] .gnu.version      VERSYM          000002d4 0002d4 000016 02   A  3   0  2
  [ 6] .gnu.version_r    VERNEED         000002ec 0002ec 000020 00   A  4   1  4
  [ 7] .rel.dyn          REL             0000030c 00030c 000028 08   A  3   0  4
  [ 8] .rel.plt          REL             00000334 000334 000010 08   A  3  10  4
  [ 9] .init             PROGBITS        00000344 000344 00002e 00  AX  0   0  4
  [10] .plt              PROGBITS        00000380 000380 000030 04  AX  0   0 16
  [11] .text             PROGBITS        000003b0 0003b0 000108 00  AX  0   0 16
  [12] .fini             PROGBITS        000004b8 0004b8 00001a 00  AX  0   0  4
```

Figure A-25.

Disassembling the Binary Files

In order to figure out the location the loader needs to fix, the best you can do is to disassemble the binary file. Indeed, the address offset 0x473 resides inside the `shlib_function()`. A closer look at code reveals that at this address you currently have the call instruction calling practically itself (i.e., jumping to its own address).

This nonsense is purposefully inserted by the linker. However, the loader is expected to fix this location after the dynamic library address range has been determined (see Figure A-26).

```
milan@milan$ objdump -d -S -M intel libfirst.so | grep -A 7 "<shlib_function>:"
0000046c <shlib_function>:
 46c:   55                      push   ebp
 46d:   89 e5                   mov    ebp,esp
 46f:   83 ec 08                sub    esp,0x8
 472:   e8 fc ff ff ff          call   473 <shlib_function+0x7>
 477:   c9                      leave
 478:   c3                      ret
 479:   90                      nop
milan@milan$
```

Figure A-26.

Runtime Analysis

The runtime analysis reveals that the LTR mechanism is brutally simple. All you can see is that the address operand of the call instruction has been replaced with the concrete address, which at this time is not meaningless at all. In fact, as you can see, it seems that it points exactly to where it should point—to the `second_shlib_function()` entry point (see Figure A-27).

```
milan@milan$ gdb -q clientApp
Reading symbols from /home/milan/LTR/clientApp/clientApp...done.
(gdb) break shlib_function
Breakpoint 1 at 0x80483e0
(gdb) run
Starting program: /home/milan/LTR/clientApp/clientApp

Breakpoint 1, shlib_function () at shlib.c:6
6           return second_shlib_function();
(gdb) set disassembly-flavor intel
(gdb) disassemble /m
Dump of assembler code for function shlib_function:
5       {
   0xb7fd846c <+0>:     push   ebp
   0xb7fd846d <+1>:     mov    ebp,esp
   0xb7fd846f <+3>:     sub    esp,0x8

6           return second_shlib_function();
=> 0xb7fd8472 <+6>:     call   0xb7e1842c <second_shlib_function>

7       }
   0xb7fd8477 <+11>:    leave
   0xb7fd8478 <+12>:    ret

End of assembler dump.
(gdb) disassemble /m 0xb7e1842c
Dump of assembler code for function second_shlib_function:
4       {
   0xb7e1842c <+0>:     push   ebp
   0xb7e1842d <+1>:     mov    ebp,esp

5           return 10;
   0xb7e1842f <+3>:     mov    eax,0xa

6       }
   0xb7e18434 <+8>:     pop    ebp
   0xb7e18435 <+9>:     ret

End of assembler dump.
(gdb)
```

Figure A-27.

The .got.plt Variable

The additional analysis may reassure you that the loader put the correct value into the address offset specified by the linker. Since you are running the example through the debugger, while the debugger blocks waiting for your next command you may examine in another terminal the layout of the clientApp process memory map. It is obvious that the code (`.text`) section of the `libsecond.so` gets at the address range starting with the address 0xb7e1800, as shown in Figure A-28.

```
milan@milan$ ps -ef | grep clientApp
milan    26834 26319  0 20:28 pts/0    00:00:00 gdb -q clientApp
milan    26836 26834  0 20:28 pts/0    00:00:00 /home/milan/LTR/clientApp/clientApp
milan    26905 26844  0 20:29 pts/1    00:00:00 grep --color=auto clientApp
milan@milan$ cat /proc/26836/maps
08048000-08049000 r-xp 00000000 08:01 2885354    /home/milan/LTR/clientApp/clientApp
08049000-0804a000 r--p 00000000 08:01 2885354    /home/milan/LTR/clientApp/clientApp
0804a000-0804b000 rw-p 00001000 08:01 2885354    /home/milan/LTR/clientApp/clientApp
b7e17000-b7e18000 rw-p 00000000 00:00 0
b7e18000-b7e19000 r-xp 00000000 08:01 2885357    /home/milan/LTR/shlibSecond/libsecond.so.1.0.0
b7e19000-b7e1a000 r--p 00000000 08:01 2885357    /home/milan/LTR/shlibSecond/libsecond.so.1.0.0
b7e1a000-b7e1b000 rw-p 00001000 08:01 2885357    /home/milan/LTR/shlibSecond/libsecond.so.1.0.0
b7e1b000-b7e1c000 rw-p 00000000 00:00 0
b7e1c000-b7fc0000 r-xp 00000000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc0000-b7fc2000 r--p 001a4000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc2000-b7fc3000 rw-p 001a6000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc6000 rw-p 00000000 00:00 0
b7fd8000-b7fd9000 r-xp 00000000 08:01 2885350    /home/milan/LTR/shlibFirst/libfirst.so.1.0.0
b7fd9000-b7fda000 r--p 00000000 08:01 2885350    /home/milan/LTR/shlibFirst/libfirst.so.1.0.0
b7fda000-b7fdb000 rw-p 00001000 08:01 2885350    /home/milan/LTR/shlibFirst/libfirst.so.1.0.0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
milan@milan$
```

Figure A-28.

On the other hand, disassembling the `libsecond.so` shows that the `second_shlib_function()` resides at the offset of 0x42c, as shown in Figure A-29.

```
milan@milan$ objdump -d -S -M intel libsecond.so | grep -A 12 "<second_shlib_function>:"
0000042c <second_shlib_function>:
#include "secondshlibexports.h"

int second_shlib_function(void)
{
 42c:   55                      push   ebp
 42d:   89 e5                   mov    ebp,esp
    return 10;
 42f:   b8 0a 00 00 00          mov    eax,0xa
}
 434:   5d                      pop    ebp
 435:   c3                      ret
 436:   90                      nop
milan@milan$
```

Figure A-29.

The combination of these two addresses (0xb7e1800 + 0x42c = 0xb7e1842c) is in fact exactly the value that loader imprinted into the code, making the whole scheme working just fine.

Scenario 2: Dynamic Library Accessing Its Own Symbols

The implementation mechanism used to resolve the Scenario 2 kind of situation is almost completely identical to what I've already illustrated in the two Scenario 1 cases. Exactly the same techniques (the use of `@plt` functions with the "trampoline" set within the `.got.plt` section plus the lazy binding in the PIC scenario, as well as the LTR direct linker resolution) is what you will find when analyzing the linker-loader coordination techniques in Scenario 2.

It is far more important to thoroughly understand the substantial differences between Scenarios 1A/1B and Scenario 2, which happen a bit above the level of immediate implementation, sections, and assembler code.

The Different Natures of the Two Scenarios

Scenarios 1A/1B and Scenario 2 substantially differ in one particular important detail. Whereas in Scenarios 1A/1B the client binary is the party that suffers the most from the address translation, in Scenario 2 it is the shared library itself that sees an immediate need to fix its own chaos.

When thinking about Scenarios 1A/1B, everything is pretty clear: the shared library exports certain symbols, and whichever client binary wants to use these symbols needs to take care of it. The symbols that need to be fixed are always and only the ABI interface symbols. Speaking in the terms of political science, Scenarios 1A/1B are a matter of the dynamic library's *foreign affairs*; see Figure A-30.

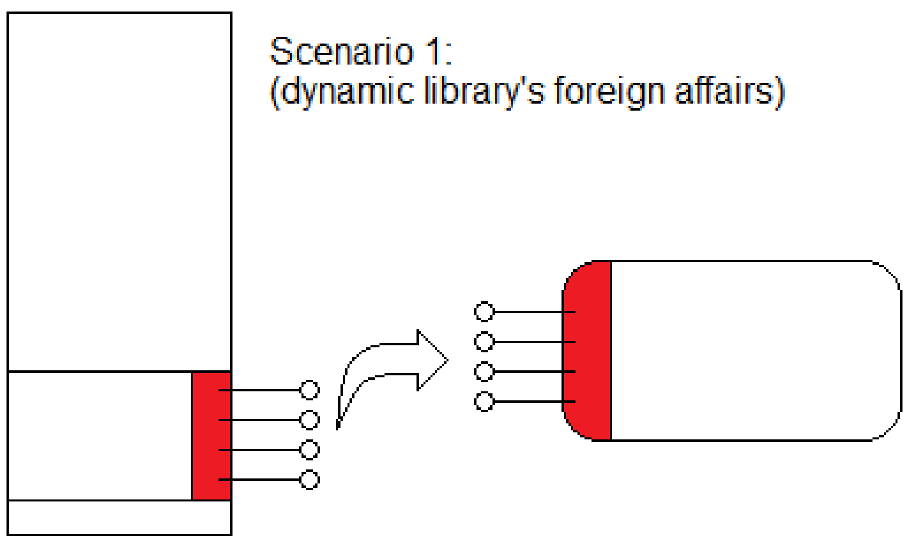

Figure A-30.

In the Scenario 2, however, the dynamic library needs to deal with its own *internal affairs*, as shown in Figure A-31.

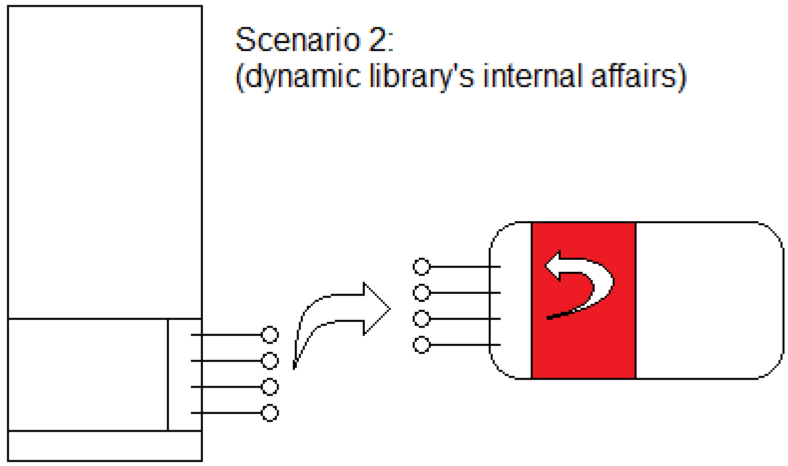

Figure A-31.

More frequently than not, the list of symbols that need to be fixed is in fact a tiny subset of the library's exported ABI symbols list. In fact, the symbols whose references need to be fixed in Scenario 2 typically belong to one of the following categories:

- The library's ABI functions called from the library's other ABI functions.
- The functions and data which are not intended to be part of the dynamic library's ABI, but for some reason are visible to the client.

In the real world, the dynamic library designers for variety of reasons (tardiness, lack of strictness, too many meetings, deadlines too tight) do not strictly follow the recommended design procedures of minimizing the number of exported symbols. Instead, they miss declaring the internal functions as static, and/or do not strictly indicate such symbols as hidden.

These kinds of mistakes are not grave errors, but they are still design imperfections of a kind. Nevertheless, the design of the linker and the loader coordination covers these cases as well.

Data Symbols Resolution

You've probably noticed that in Scenarios 1A/1B, I covered only the resolution of function symbols, whereas the illustrations of fixing the data symbols haven't been provided.

In the Scenario 1 examples, the recommended design rules stipulate using only the functions as the part of ABI interface. If the dynamic library designer decides for whatever reason to export the data variable, the only way the client binary may use it directly is to declare it as extern. During link time, such a variable is placed by the linker into the data (`.data`, `.bss`) sections of the client binary.

In Scenario 2, resolving the data symbols is a far more probable scenario. Sometimes a data symbol simply cannot be declared static, as it may be needed by the functions of the same dynamic library implemented in different source files (i.e., ultimately residing in the different object files) of the same dynamic library. Combined with the lack of strict symbol exporting rules, referencing the data symbol becomes the likely outcome.

Test Model

In order to provide a good test bed for illustrating the typical linker-loading coordination problems and solutions, a simple demo project was created featuring the following items.

Custom Dynamic Library Project

The custom dynamic library project features the following functions:

- Three exported ABI functions, one of them calling the other two
- A non-static function whose symbols are exported outside the dynamic library
- A non-static function whose symbols are not exported outside the dynamic library
- A function declared `static` (in the sense of the C language)

It also features the following variables:

- A global variable, declared "extern" by the client binary code
- A non-static variable
- A variable declared `static` (in the sense of the C language)

The dynamic library's source code is comprised of the files in Listing A-3.

Listing A-3. Please Add Caption

```
file: shlibexports.h

// Variables intended for export
int nShlibExportedVariable  = 0;

// Functions intended for export (ABI functions)
int shlib_abi_initialize(int x, int y);
int shlib_abi_uninitialize(void);
int shlib_abi_reinitialize(int x, int y);
```

```
file: shlib.c
#include "shlibexports.h"

#define DO_NOT_EXPORT __attribute__ ((visibility("hidden")))

int        nShlibNonStaticVariable     = 2;
static int nshlibStaticVariable                   = 3;

static int shlib_static_function(int x, int y)
{
    int retValue = x + y;
    retValue    *= nshlibStaticVariable;
    return retValue;
}

int DO_NOT_EXPORT shlib_nonstatic_hidden_function(int x, int y)
{
    int result = shlib_static_function(x, y);
    return result;
}

int shlib_nonstatic_exported_function(int x, int y)
{
    int result = 2*shlib_static_function(x, y);
    result    *= nShlibNonStaticVariable;
    return result;
}

int shlib_abi_initialize(int x, int y)
{
    int first = shlib_nonstatic_hidden_function(x, y);
    int second = shlib_nonstatic_exported_function(x, y);
    nShlibExportedVariable = first + second;
    return 0;
}

int shlib_abi_uninitialize(void)
{
    return 0;
}

int shlib_abi_reinitialize(int x, int y)
{
    shlib_abi_uninitialize();
    return shlib_abi_initialize(x, y);
}

file: build.sh
```

```
rm -rf *.o lib*
gcc -Wall -g -O0 -c shlib.c # -fPIC compiler flag will be added for the PIC demo case
gcc -shared shlib.o -o libshlib.so.1.0.0 -Wl,-soname,libshlib.so.1
ldconfig -l libshlib.so.1.0.0
ln -s libshlib.so.1 libshlib.so
```

Demo Application Project

The demo application project features the following features:

- Statically aware linking of the custom dynamic library.
- A function call to the dynamic library's ABI functions
- A function call to the dynamic library's mistakenly un-hidden, non-ABI function
- Variable access of the custom library's global “extern” variable

The application's source code is contained in Listing A-4.

Listing A-4.

```
file: main.c
#include <stdio.h>
#include "shlibexports.h"

extern  int shlibAccessedAsExternVariable;

int main(int argc, const char* argv[])
{
    int t;
    int first = nShlibExportedVariable + 1;
    t = shlib_abi_initialize(first, argc);

    int second = nShlibExportedVariable + 2;
    t = shlib_abi_reinitialize(second, argc);

    // compiler warns about implicit declaration
    // of this function, but manages to resolve it
    // at runtime. We did not plan on exporting it
    // but also did not bother to explicitly hide it
    t = shlib_nonstatic_exported_function(first, argc);

#if 0
    // can't access hidden function
    t = shlib_nonstatic_hidden_function(first, argc);

    // calling these two will result with compiler error
    int result = shlib_static_function(first, second);
```

```
    result    *= shlibNonStaticVariable;
#endif
    return t;
}
```

```
file: build.sh
gcc -Wall -g -O0 -c -I../sharedLib main.c
gcc main.o -Wl,-L../sharedLib -lshlib -Wl,-R../sharedLib -o clientApp
```

Detailed Analysis of the Position-Independent Code Approach

The analysis of the dynamic library's relocation information indicates what you expected to encounter in Scenario 2. The list of symbols in need of relocation does not contain all the elements from the list of exported symbols, as shown in Figures A-32 and A-33.

```
milan@milan$ readelf -r libshlib.so

Relocation section '.rel.dyn' at offset 0x3c8 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002014  00000008 R_386_RELATIVE
00001fe0  00000406 R_386_GLOB_DAT    00002018   nShlibNonStaticVariabl
00001fe4  00000106 R_386_GLOB_DAT    00000000   __cxa_finalize
00001fe8  00000206 R_386_GLOB_DAT    00000000   __gmon_start__
00001fec  00000e06 R_386_GLOB_DAT    00002028   nShlibExportedVariable
00001ff0  00000306 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses

Relocation section '.rel.plt' at offset 0x3f8 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000a07 R_386_JUMP_SLOT   000005dd   shlib_nonstatic_export
00002004  00000107 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002008  00000d07 R_386_JUMP_SLOT   00000656   shlib_abi_uninitialize
0000200c  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
00002010  00000607 R_386_JUMP_SLOT   000005ff   shlib_abi_initialize
milan@milan$
```

Figure A-32.

```
milan@milan$ nm -D libshlib.so
         w _Jv_RegisterClasses
00002020 A __bss_start
         w __cxa_finalize
         w __gmon_start__
00002020 A _edata
0000202c A _end
000006d8 T _fini
00000420 T _init
00002028 B nShlibExportedVariable
00002018 D nShlibNonStaticVariable
000005ff T shlib_abi_initialize
00000660 T shlib_abi_reinitialize
00000656 T shlib_abi_uninitialize
000005dd T shlib_nonstatic_exported_function
milan@milan$
```

Figure A-33.

In particular, the `shlib_abi_reinitialize()` function symbol is an exported ABI symbol, yet it is not on the list of symbols you should worry about from the standpoint of Scenario 2.

Handling of Disrupted Function Entry Points

The line describing the `shlib_nonstatic_exported_function` function points to the address 0x2000, which surprisingly does not belong to the code (`.text`) section (!?).

OK....but then...where does it point to?

The best way to start unraveling this mystery is to examine the list of sections carried by the dynamic library, as shown in Figure A-34.

```
milan@milan$ readelf --sections libshlib.so
There are 33 section headers, starting at offset 0x1700:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            00000114 000114 000024 00   A  0   0  4
  [ 2] .gnu.hash         GNU_HASH        00000138 000138 000050 04   A  3   0  4
  [ 3] .dynsym           DYNSYM          00000188 000188 0000f0 10   A  4   1  4
  [ 4] .dynstr           STRTAB          00000278 000278 00010f 00   A  0   0  1
  [ 5] .gnu.version      VERSYM          00000388 000388 00001e 02   A  3   0  2
  [ 6] .gnu.version_r    VERNEED         000003a8 0003a8 000020 00   A  4   1  4
  [ 7] .rel.dyn          REL             000003c8 0003c8 000030 08   A  3   0  4
  [ 8] .rel.plt          REL             000003f8 0003f8 000028 08   A  3  10  4
  [ 9] .init             PROGBITS        00000420 000420 00002e 00  AX  0   0  4
  [10] .plt              PROGBITS        00000450 000450 000060 04  AX  0   0 16
  [11] .text             PROGBITS        000004b0 0004b0 000228 00  AX  0   0 16
  [12] .fini             PROGBITS        000006d8 0006d8 00001a 00  AX  0   0  4
  [13] .eh_frame_hdr     PROGBITS        000006f4 0006f4 00004c 00   A  0   0  4
  [14] .eh_frame         PROGBITS        00000740 000740 000120 00   A  0   0  4
  [15] .ctors            PROGBITS        00001f04 000f04 000008 00  WA  0   0  4
  [16] .dtors            PROGBITS        00001f0c 000f0c 000008 00  WA  0   0  4
  [17] .jcr              PROGBITS        00001f14 000f14 000004 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         00001f18 000f18 0000c8 08  WA  4   0  4
  [19] .got              PROGBITS        00001fe0 000fe0 000014 04  WA  0   0  4
  [20] .got.plt          PROGBITS        00001ff4 000ff4 000020 04  WA  0   0  4
  [21] .data             PROGBITS        00002014 001014 00000c 00  WA  0   0  4
  [22] .bss              NOBITS          00002020 001020 00000c 00  WA  0   0  4
  [23] .comment          PROGBITS        00000000 001020 00002a 01  MS  0   0  1
  [24] .debug_aranges    PROGBITS        00000000 00104a 000020 00      0   0  1
  [25] .debug_info       PROGBITS        00000000 00106a 0001ce 00      0   0  1
  [26] .debug_abbrev     PROGBITS        00000000 001238 000095 00      0   0  1
  [27] .debug_line       PROGBITS        00000000 0012cd 00006b 00      0   0  1
  [28] .debug_str        PROGBITS        00000000 001338 00014e 01  MS  0   0  1
  [29] .debug_loc        PROGBITS        00000000 001486 000150 00      0   0  1
  [30] .shstrtab         STRTAB          00000000 0015d6 000129 00      0   0  1
  [31] .symtab           SYMTAB          00000000 001c28 000430 10     32  53  4
  [32] .strtab           STRTAB          00000000 002058 000262 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
milan@milan$
```

Figure A-34.

Voila!!! The location to be patched by the loader belongs to the `.got.plt` section. Truth be told, in Chapter 9 I didn't say anything about a section called `.got.plt`. When I discussed the principles of the position-independent code implementation, all I mentioned was the `.got` section, not the `.got.plt` section.

The `.got.plt` section is essentially nothing more than the `.got` section specialized to carry information related specifically to the functions (whereas `.got` section pertains to the variables). The acronym "plt" stands for procedure linkage table.

You may think of `.got` and `.got.plt` as of Hansel and Gretel, siblings of different gender but pretty much always together in the grim and scary story of fighting the bad and ugly address translation problems.

Back to the task at hand. Let's watch the places in the code where calls to the function `shlib_nonstatic_exported_function` take place, as it will directly lead us into the details of how the disrupted symbols handling was implemented (see Figure A-35).

```
milan@milan$ objdump -d -S -M intel libshlib.so | grep -A 25 "<shlib_abi_initialize>:"
000005ff <shlib_abi_initialize>:

int shlib_abi_initialize(int x, int y)
{
 5ff:   55                      push   ebp
 600:   89 e5                   mov    ebp,esp
 602:   53                      push   ebx
 603:   83 ec 24                sub    esp,0x24
 606:   e8 5c ff ff ff          call   567 <__i686.get_pc_thunk.bx>
 60b:   81 c3 e9 19 00 00       add    ebx,0x19e9
        int first = shlib_nonstatic_hidden_function(x, y);
 611:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 614:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 618:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 61b:   89 04 24                mov    DWORD PTR [esp],eax
 61e:   e8 79 ff ff ff          call   59c <shlib_nonstatic_hidden_function>
 623:   89 45 f0                mov    DWORD PTR [ebp-0x10],eax
        int second = shlib_nonstatic_exported_function(x, y);
 626:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 629:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 62d:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 630:   89 04 24                mov    DWORD PTR [esp],eax
 633:   e8 28 fe ff ff          call   460 <shlib_nonstatic_exported_function@plt>
 638:   89 45 f4                mov    DWORD PTR [ebp-0xc],eax
        nShlibExportedVariable = first + second;
 63b:   8b 45 f4                mov    eax,DWORD PTR [ebp-0xc]
milan@milan$
```

Figure A-35.

The call to the function is implemented through the function called `shlib_nonstatic_exported_function@plt`, whose code is shown in Figure A-36.

```
00000460 <shlib_nonstatic_exported_function@plt>:
 460:   ff a3 0c 00 00 00       jmp    DWORD PTR [ebx+0xc]
 466:   68 00 00 00 00          push   0x0
 46b:   e9 e0 ff ff ff          jmp    450 <_init+0x30>
```

Figure A-36.

Here we come to familiar ground. The ways the `@plt` functions work (i.e., the "trampoline" mechanism plus the lazy binding concept) have already been discussed within the scope of the Scenario 1 analyses.

There is absolutely no difference between how the whole scheme works under the Scenario 2 circumstances. For that reason, I will stop short of repeating the same story again. The reader who skipped these details is encouraged to go through the previous sections of this chapter.

Handling of Disrupted Variable Addresses

Unlike the `@plt` way of resolving the function symbols references, the mechanism of resolving the data references has not been discussed before (mostly because it is a relatively infrequent case, which mostly collides against solid design guidelines and recommendations).

The line describing the `nShlibNonStaticVariable` points to the address offset 0x1fe0, which surprisingly does not belong to the `.data` section(!?).

OK, but where does it point to?

Another look at the dynamic library sections layout can help solve the mystery (see Figure A-37).

```
milan@milan$ readelf --sections libshlib.so
There are 33 section headers, starting at offset 0x1700:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            00000114 000114 000024 00   A  0   0  4
  [ 2] .gnu.hash         GNU_HASH        00000138 000138 000050 04   A  3   0  4
                                   o
                                   o
                                   o
  [18] .dynamic          DYNAMIC         00001f18 000f18 0000c8 08  WA  4   0  4
  [19] .got              PROGBITS        00001fe0 000fe0 000014 04  WA  0   0  4
  [20] .got.plt          PROGBITS        00001ff4 000ff4 000020 04  WA  0   0  4
  [21] .data             PROGBITS        00002014 001014 00000c 00  WA  0   0  4
                                   o
                                   o
                                   o
```

Figure A-37.

The location to be patched by the loader belongs to the `.got` section. This perfectly fits the previously discussed principles of the position-independent code implementation. Obviously, this is the address of a variable within the `.got` section, which needs to be patched by the loader according to the linker directives. A closer look at the place in the code in which the variable gets referenced reveals the already familiar scheme (see Figure A-38).

```
milan@milan$ objdump -d -S -M intel libshlib.so | grep -A 25 "nonstati
000005bc <shlib_nonstatic_exported_function>:

int shlib_nonstatic_exported_function(int x, int y)
{
 5bc:   55                      push   ebp
 5bd:   89 e5                   mov    ebp,esp
 5bf:   53                      push   ebx
 5c0:   83 ec 18                sub    esp,0x18
 5c3:   e8 9f ff ff ff          call   567 <__i686.get_pc_thunk.bx>
 5c8:   81 c3 2c 1a 00 00       add    ebx,0x1a2c
        int result = 2*shlib_static_function(x, y);
 5ce:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 5d1:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 5d5:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 5d8:   89 04 24                mov    DWORD PTR [esp],eax
 5db:   e8 8c ff ff ff          call   56c <shlib_static_function>
 5e0:   01 c0                   add    eax,eax
 5e2:   89 45 f8                mov    DWORD PTR [ebp-0x8],eax
        result    *= nShlibNonStaticVariable;
 5e5:   8b 83 ec ff ff ff       mov    eax,DWORD PTR [ebx-0x14]
 5eb:   8b 00                   mov    eax,DWORD PTR [eax]
 5ed:   8b 55 f8                mov    edx,DWORD PTR [ebp-0x8]
 5f0:   0f af c2                imul   eax,edx
 5f3:   89 45 f8                mov    DWORD PTR [ebp-0x8],eax
milan@milan$
```

Figure A-38.

From your experience with analyzing the Scenario 1 examples, the presence of the `__i686.get_pc_thunk.bx` function implies that a similar mechanism is in place, as you can see in Figure A-39.

```
00000567 <__i686.get_pc_thunk.bx>:
 567:   8b 1c 24                mov    ebx,DWORD PTR [esp]
 56a:   c3                      ret
 56b:   90                      nop
```

Figure A-39.

The detailed explanation of how this scheme works has been already provided during the analyses of the Scenario 1 examples. Regardless of the fact that in the previous examples the scheme worked on the variables located in the `.got.plt` (instead of the `.got`) section, the principle is completely the same.

The following snippet of code is where the gist of the story resides:

```
5c3:   e8 9f ff ff ff        call   567 <__i686.get_pc_thunk.bx>
5c8:   81 c3 2c 1a 00 00     add    ebx,0x1a2c
…
5e5:   8b 83 ec ff ff ff     mov    eax, DWORD PTR[ebx-0x14]
```

The reason why the linker generated the code that increases the current program counter value (captured in ebx register) by the constant value 0x1a2c becomes clear in Figure A-40.

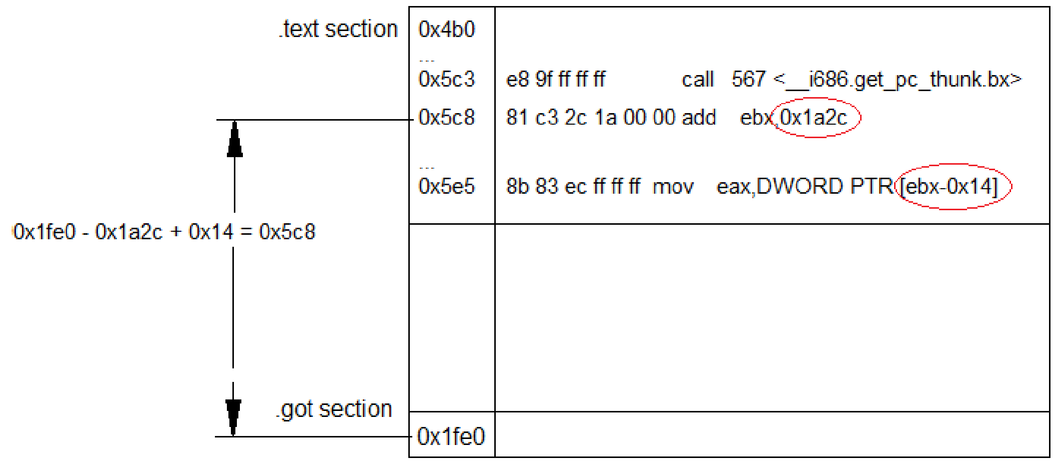

Figure A-40.

Obviously, the whole purpose of this linker trick is to calculate the relative offset *from wherever the code ends up being loaded* to exactly the **`.got`** section variable to which the loader will stamp in the ultimate address of the variable you are trying to access (i.e., **`nShlibNonStaticVariable`**). The linker resorts to the same trick based on the known constant offset between the sections (be it **`.got`** or **`.got.plt`**) to implement the PIC scheme.

Specifics of Position-Independent Code Implementation in a 64-bit OS

As already mentioned in the section on the Scenario 1 examples analyses, the addressing mode called **relative instruction pointer mode (RIP)** is specific to 64-bit architecture, and it improves the performance by eliminating the need for the compiler to implement the **`__i686.get_pc_thunk.bx`** function.

Detailed Analysis of the Load-Time Relocation Approach

The analysis of the linker-loader coordination in Scenario 2 when the dynamic library is built without the **`-fPIC`** compiler flag does not bring substantially new findings. Much as in the PIC case, the list of symbols needing the loader's intervention does not contain all the elements from the list of exported symbols, as you can see in Figures A-41 and A-42.

```
milan@milan$ readelf -r libshlib.so

Relocation section '.rel.dyn' at offset 0x3c8 contains 10 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000055e  00000008 R_386_RELATIVE
00002008  00000008 R_386_RELATIVE
000005ae  00000401 R_386_32          0000200c   nShlibNonStaticVariabl
000005e9  00000a02 R_386_PC32        00000590   shlib_nonstatic_export
000005f9  00000e01 R_386_32          0000201c   nShlibExportedVariable
00000615  00000d02 R_386_PC32        00000604   shlib_abi_uninitialize
00000627  00000602 R_386_PC32        000005c0   shlib_abi_initialize
00001fe8  00000106 R_386_GLOB_DAT    00000000   __cxa_finalize
00001fec  00000206 R_386_GLOB_DAT    00000000   __gmon_start__
00001ff0  00000306 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses

Relocation section '.rel.plt' at offset 0x418 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
milan@milan$
```

Figure A-41.

```
milan@milan$ nm -D libshlib.so
         w _Jv_RegisterClasses
00002014 A __bss_start
         w __cxa_finalize
         w __gmon_start__
00002014 A _edata
00002020 A _end
00000668 T _fini
00000428 T _init
0000201c B nShlibExportedVariable
0000200c D nShlibNonStaticVariable
000005c0 T shlib_abi_initialize
0000060e T shlib_abi_reinitialize
00000604 T shlib_abi_uninitialize
00000590 T shlib_nonstatic_exported_function
milan@milan$
```

Figure A-42.

Again, it is the `shlib_abi_reinitialize()` function symbol that is exported, but yet it's not found in the list of symbols that you should worry about.

The usual analysis approach of disassembling the code to identify the troubled spots and checking at runtime whether the loader fixed the issue reveals nothing unexpected. The analysis techniques you applied when analyzing the Scenario 1 example are completely applicable to the Scenario 2 as well.

The Linker's Handling of Disrupted Function Entry Points

As indicated by the relocation tables, the references to the following functions' symbols require the loader's corrective actions:

- `shlib_abi_uninitialize`, at the address offset 0x615
- `shlib_abi_initialize`, at the address offset 0x627
- `shlib_nonstatic_exported_function`, at address offset 0x5e9

The disassembled dynamic library code shows where exactly and why the loader's corrective action is needed, as shown in Figure A-43.

```
milan@milan$ objdump -d -S -M intel libshlib.so | grep -A 6 abi_re
0000060e <shlib_abi_reinitialize>:

int shlib_abi_reinitialize(int x, int y)
{
 60e:   55                      push   ebp
 60f:   89 e5                   mov    ebp,esp
 611:   83 ec 08                sub    esp,0x8
        shlib_abi_uninitialize();
 614:   e8 fc ff ff ff          call   615 <shlib_abi_reinitialize+0x7>
        return shlib_abi_initialize(x, y);
 619:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 61c:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 620:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 623:   89 04 24                mov    DWORD PTR [esp],eax
 626:   e8 fc ff ff ff          call   627 <shlib_abi_reinitialize+0x19>
}
 62b:   c9                      leave
 62c:   c3                      ret
 62d:   90                      nop
 62e:   90                      nop
 62f:   90                      nop
milan@milan$
```

Figure A-43.

As you can see, the call instruction is set by the linker to jump at itself. Obviously, this instruction needs to be fixed by the loader. A completely similar situation happens with the other two functions, as shown in Figure A-44.

```
milan@milan$ objdump -d -S -M intel libshlib.so | grep -A 30 "<shlib_abi_initialize>:"
000005c0 <shlib_abi_initialize>:

int shlib_abi_initialize(int x, int y)
{
 5c0:   55                      push   ebp
 5c1:   89 e5                   mov    ebp,esp
 5c3:   83 ec 18                sub    esp,0x18
        int first = shlib_nonstatic_hidden_function(x, y);
 5c6:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 5c9:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 5cd:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 5d0:   89 04 24                mov    DWORD PTR [esp],eax
 5d3:   e8 98 ff ff ff          call   570 <shlib_nonstatic_hidden_function>
 5d8:   89 45 f8                mov    DWORD PTR [ebp-0x8],eax
        int second = shlib_nonstatic_exported_function(x, y);
 5db:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 5de:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 5e2:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 5e5:   89 04 24                mov    DWORD PTR [esp],eax
 5e8:   e8 fc ff ff ff          call   5e9 <shlib_abi_initialize+0x29>
 5ed:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
        nShlibExportedVariable = first + second;
 5f0:   8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 5f3:   8b 55 f8                mov    edx,DWORD PTR [ebp-0x8]
 5f6:   01 d0                   add    eax,edx
 5f8:   a3 00 00 00 00          mov    ds:0x0,eax
        return 0;
 5fd:   b8 00 00 00 00          mov    eax,0x0
}
 602:   c9                      leave
 603:   c3                      ret
milan@milan$
```

Figure A-44.

Linker's Handling of Disrupted Variable Addresses

As indicated by the relocation tables, the references to the following data symbols require the loader's corrective actions:

- `nShlibNonStaticVariable`, at the address offset 0x5ae
- `nShlibExportedVariable`, at the address offset 0x5f9

Examining the disassembled code in the neighborhood of these address offsets shows the following, seen in Figure A-45.

```
milan@milan$ objdump -d -S -M intel libshlib.so
00000590 <shlib_nonstatic_exported_function>:

int shlib_nonstatic_exported_function(int x, int y)
{
 590:   55                      push   ebp
 591:   89 e5                   mov    ebp,esp
 593:   83 ec 18                sub    esp,0x18
        int result = 2*shlib_static_function(x, y);
 596:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 599:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 59d:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 5a0:   89 04 24                mov    DWORD PTR [esp],eax
 5a3:   e8 a4 ff ff ff          call   54c <shlib_static_function>
 5a8:   01 c0                   add    eax,eax
 5aa:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
        // causes problem when compiled on 64-bit OS
        // Compiler flag -mcmodel=large fixes the problem
        result    *= nShlibNonStaticVariable;
 5ad:   a1 00 00 00 00          mov    eax,ds:0x0
 5b2:   8b 55 fc                mov    edx,DWORD PTR [ebp-0x4]
 5b5:   0f af c2                imul   eax,edx
 5b8:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
        return result;
 5bb:   8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
}
 5be:   c9                      leave
 5bf:   c3                      ret

milan@milan$
```

```
milan@milan$ objdump -d -S -M intel libshlib.so
000005c0 <shlib_abi_initialize>:

int shlib_abi_initialize(int x, int y)
{
 5c0:   55                      push   ebp
 5c1:   89 e5                   mov    ebp,esp
 5c3:   83 ec 18                sub    esp,0x18
        int first = shlib_nonstatic_hidden_function(x, y);
 5c6:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 5c9:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 5cd:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 5d0:   89 04 24                mov    DWORD PTR [esp],eax
 5d3:   e8 98 ff ff ff          call   570 <shlib_nonstatic_hidden_function>
 5d8:   89 45 f8                mov    DWORD PTR [ebp-0x8],eax
        int second = shlib_nonstatic_exported_function(x, y);
 5db:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 5de:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 5e2:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 5e5:   89 04 24                mov    DWORD PTR [esp],eax
 5e8:   e8 fc ff ff ff          call   5e9 <shlib_abi_initialize+0x29>
 5ed:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
        nShlibExportedVariable = first + second;
 5f0:   8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 5f3:   8b 55 f8                mov    edx,DWORD PTR [ebp-0x8]
 5f6:   01 d0                   add    eax,edx
 5f8:   a3 00 00 00 00          mov    ds:0x0,eax
        return 0;
 5fd:   b8 00 00 00 00          mov    eax,0x0
}
 602:   c9                      leave
 603:   c3                      ret
milan@milan$
```

Figure A-45.

Obviously, trying to access the data from the address 0x0 is the bogus instruction purposefully inserted by the linker, with the intention that it be eventually fixed by the loader. What remains to be seen at runtime is whether the loader truly fixed these troubled locations (see Figure A-46).

Runtime Analysis (How the Loader Fixed the Problems)

The mandatory first step in getting a better orientation is to determine at runtime where exactly the dynamic library got loaded. Since you will be using the gdb debugger to examine the code at runtime, the simplest way to do it is to examine the contents of the `/proc/<PID>/maps` file while the debugger is blocked on the break point (breaking on the `main()` function is guaranteed to work), as shown in Figure A-46.

```
milan@milan$ ps -ef | grep clientApp
milan    30134 28596  0 22:24 pts/1    00:00:00 gdb -q clientApp
milan    30136 30134  0 22:24 pts/1    00:00:00 /home/milan/LTR/clientApp/clientApp
milan    30243 30140  0 22:24 pts/3    00:00:00 grep --color=auto clientApp
milan@milan$ cat /proc/30136/maps
08048000-08049000 r-xp 00000000 08:01 3015424    /home/milan/LTR/clientApp/clientApp
08049000-0804a000 r--p 00000000 08:01 3015424    /home/milan/LTR/clientApp/clientApp
0804a000-0804b000 rw-p 00001000 08:01 3015424    /home/milan/LTR/clientApp/clientApp
b7e1a000-b7e1c000 rw-p 00000000 00:00 0
b7e1c000-b7fc0000 r-xp 00000000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc0000-b7fc2000 r--p 001a4000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc2000-b7fc3000 rw-p 001a6000 08:01 7344063    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc6000 rw-p 00000000 00:00 0
b7fd8000-b7fd9000 r-xp 00000000 08:01 3015401    /home/milan/LTR/sharedLib/libshlib.so.1.0.0
b7fd9000-b7fda000 r--p 00000000 08:01 3015401    /home/milan/LTR/sharedLib/libshlib.so.1.0.0
b7fda000-b7fdb000 rw-p 00001000 08:01 3015401    /home/milan/LTR/sharedLib/libshlib.so.1.0.0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 7344053    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
milan@milan$
```

Figure A-46.

Once you know the start address of the dynamic library's code and data sections at runtime, it will be very easy to verify the effects of the loader's corrective actions on the code. First, calls to the **`shlib_abi_uninitialize()`** and **`shlib_abi_initialize()`** functions seem to be correctly resolved. When combining the start address of dynamic library's code section (0xb7fd8000) with the offsets of these function entry points, you get exactly the values that the loader inserted into the designated call instructions. For example, 0xb7fd800 + 0x5c0 (entry point of **`shlib_abi_initilize()`** function) yields 0xb7fd85c0, which you can see in Figure A-47 is exactly what the loader inserted. Similar reasoning may be applied to verify that the runtime address of **`shlib_abi_uninitialize()`** function has been correctly evaluated by the loader.

```
(gdb) disassemble /m shlib_abi_reinitialize
Dump of assembler code for function shlib_abi_reinitialize:
42      {
   0xb7fd860e <+0>:     push   ebp
   0xb7fd860f <+1>:     mov    ebp,esp
   0xb7fd8611 <+3>:     sub    esp,0x8

43              shlib_abi_uninitialize();
   0xb7fd8614 <+6>:     call   0xb7fd8604 <shlib_abi_uninitialize>

44              return shlib_abi_initialize(x, y);
   0xb7fd8619 <+11>:    mov    eax,DWORD PTR [ebp+0xc]
   0xb7fd861c <+14>:    mov    DWORD PTR [esp+0x4],eax
   0xb7fd8620 <+18>:    mov    eax,DWORD PTR [ebp+0x8]
   0xb7fd8623 <+21>:    mov    DWORD PTR [esp],eax
   0xb7fd8626 <+24>:    call   0xb7fd85c0 <shlib_abi_initialize>

45      }
   0xb7fd862b <+29>:    leave
   0xb7fd862c <+30>:    ret

End of assembler dump.
(gdb)
```

Figure A-47.

The analysis of the data symbols resolution shows similar results. Namely, the access to `nShlibNonStaticVariable` has been resolved correctly (see Figure A-48).

```
   0xb7fd8580 <+16>:    mov    DWORD PTR [esp],eax
   0xb7fd8583 <+19>:    call   0xb7fd854c <shlib_static_function>
   0xb7fd8588 <+24>:    mov    DWORD PTR [ebp-0x4],eax

18              result    *= nShlibNonStaticVariable;
   0xb7fd858b <+27>:    mov    eax,ds:0xb7fda00c
   0xb7fd8590 <+32>:    mov    edx,DWORD PTR [ebp-0x4]
   0xb7fd8593 <+35>:    imul   eax,edx
   0xb7fd8596 <+38>:    mov    DWORD PTR [ebp-0x4],eax

19              return result;
   0xb7fd8599 <+41>:    mov    eax,DWORD PTR [ebp-0x4]

20      }
   0xb7fd859c <+44>:    leave
   0xb7fd859d <+45>:    ret

End of assembler dump.
(gdb)
```

Figure A-48.

This may be easily verified by combining the dynamic library's runtime start address (0xb7fd8000) with the relative address offset of the variable in dynamic library's `.bss` section (see Figure A-49).

```
milan@milan$ objdump -x -j .data libshlib.so | grep Variable
00002010 l     O .data  00000004              nshlibStaticVariable
0000200c g     O .data  00000004              nShlibNonStaticVariable
milan@milan$
```

Figure A-49.

Obviously, 0xb7fd8000 + 0x200c = 0xb7fda00c, which proves that the loader completed the task successfully. The analysis of how the loader fixed the address of the `nShlibExportedVariable` shows something interesting; see Figure A-50.

```
(gdb) disassemble /m shlib_abi_initialize
Dump of assembler code for function shlib_abi_initialize:
29      {
   0xb7fd85c0 <+0>:     push   ebp
   0xb7fd85c1 <+1>:     mov    ebp,esp
   0xb7fd85c3 <+3>:     sub    esp,0x18

30              int first = shlib_nonstatic_hidden_function(x, y);
   0xb7fd85c6 <+6>:     mov    eax,DWORD PTR [ebp+0xc]
   0xb7fd85c9 <+9>:     mov    DWORD PTR [esp+0x4],eax
   0xb7fd85cd <+13>:    mov    eax,DWORD PTR [ebp+0x8]
   0xb7fd85d0 <+16>:    mov    DWORD PTR [esp],eax
   0xb7fd85d3 <+19>:    call   0xb7fd8570 <shlib_nonstatic_hidden_function>
   0xb7fd85d8 <+24>:    mov    DWORD PTR [ebp-0x8],eax

31              int second = shlib_nonstatic_exported_function(x, y);
   0xb7fd85db <+27>:    mov    eax,DWORD PTR [ebp+0xc]
   0xb7fd85de <+30>:    mov    DWORD PTR [esp+0x4],eax
   0xb7fd85e2 <+34>:    mov    eax,DWORD PTR [ebp+0x8]
   0xb7fd85e5 <+37>:    mov    DWORD PTR [esp],eax
   0xb7fd85e8 <+40>:    call   0xb7fd859e <shlib_nonstatic_exported_function>
   0xb7fd85ed <+45>:    mov    DWORD PTR [ebp-0x4],eax

32              nShlibExportedVariable = first + second;
   0xb7fd85f0 <+48>:    mov    eax,DWORD PTR [ebp-0x4]
   0xb7fd85f3 <+51>:    mov    edx,DWORD PTR [ebp-0x8]
   0xb7fd85f6 <+54>:    add    eax,edx
   0xb7fd85f8 <+56>:    mov    ds:0x804a024,eax

33              return 0;
   0xb7fd85fd <+61>:    mov    eax,0x0

34      }
   0xb7fd8602 <+66>:    leave
   0xb7fd8603 <+67>:    ret

End of assembler dump.
(gdb)
```

Figure A-50.

The variable address 0x804a024 suggests that the `nShlibExportedVariable` has been mapped not to the dynamic library address space, but instead to the client applications' address space. This should not be a huge surprise. In order for client binary to access the dynamic library's variable, it must be declared as extern in the client binary's source code. The linker takes that as a suggestion to reserve a place for the variable within the client binary's data section. This fact can be easily checked by disassembling the client application's `.bss` section, as shown in Figure A-51.

```
milan@milan$ objdump -x -j .bss clientApp | grep Exported
0804a024 g     O .bss   00000004              nShlibExportedVariable
milan@milan$
```

Figure A-51.

Specifics of the Load-Time Relocation Implementation in a 64-bit OS

Implementing the load-time relocation concept in 64-bit Linux faces certain challenges. Namely, the mere omission of the `-fPIC` compiler flag causes the linker error shown in Figure A-52.

```
milan@milan$ ./build.sh
/usr/bin/ld: shlib.o: relocation R_X86_64_PC32 against symbol `nShlibNonStaticVariable'
can not be used when making a shared object; recompile with -fPIC
/usr/bin/ld: final link failed: Bad value
collect2: ld returned 1 exit status
/sbin/ldconfig.real: Cannot lstat libshlib.so.1.0.0: No such file or directory
milan@milan$
```

Figure A-52.

The cause of the error is fairly easy to understand: in order to access the variable, the compiler relies on the `mov` instruction which takes a 32-bit argument. Even though the address range is 64-bit wide, for the variables of the local scope the 32-bit storage may be (in fact, mostly is) wide enough to store the address distance between the caller and the referenced symbol.

In case of exported functions (which is the case in this particular example), the possible range of addresses that the relocation will introduce is significantly larger, the order of magnitude being much closer to 64-bit than to 32-bit range. It would be wrong to expect that the 32-bit register would be able to accommodate the address offset. This limitation is recognized by the linker and reported as an error. Fortunately, it is still possible to create the dynamic library featuring load-time relocation on the 64-bit OS.

Passing the `-mcmodel=large` compiler flag causes the compiler to issue the more appropriate `movabs` instruction and the more appropriate `rax` register to handle the symbol access in the 64-bit address range.

In all other aspects, the way the linker resolves the references is pretty much the same as how it is accomplished on a 32-bit OS.

The reader is encouraged to build the code example on the 64-bit OS and apply exactly the same analysis as you did in the 32-bit OS to prove that load time relocation works identically in a 64-bit OS.