

Linux Debugging (一)：使用反彙編理解C++程序函數調用棧

拿到CoreDump後，如果看到的地址都是????，那麼基本上可以確定，程序的棧被破壞掉了。GDB也是使用函數的調用棧去還原「事故現場」的。因此理解函數調用棧，是使用GDB進行現場調試或者事後調試的基礎，如果不理解調用棧，基本上也從GDB得不到什麼有用的信息。當然了，也有可能你非常「幸運」，一個bt就把哪兒越界給標出來了。但是，大多數的時候你不夠幸運，通過log，通過簡單的code walkthrough，得不到哪兒出的問題；或者說只是推測，不能確診。我們需要通過GDB來最終確定CoreDump產生的真正原因。

本文還可以幫助你深入理解C++函數的局部變量。我們學習時知道局部變量是存儲到棧裡的，內存管理對程序員是透明的。通過本文，你將明白這些結論是如何得出的。

棧，是LIFO（Last In First Out）的數據結構。C++的函數調用就是通過棧來傳遞參數，保存函數返回後下一步的執行地址。接下來我們通過一個具體的例子來探究。

[cpp] view plaincopy 

```
1.  int func1(int a)
2.  {
3.      int b = a + 1;
4.      return b;
5.  }
6.  int func0(int a)
7.  {
8.      int b = func1(a);
9.      return b;
10. }
11.
12. int main()
13. {
14.     int a = 1234;
15.     func0(a);
16.     return 0;
```

17. }

可以使用以下命令將上述code編程成彙編代碼：

```
g++ -g -S -O0 -m32 main.cpp -o|c++filt >main.format.s
```

c++filt 是為了Demangle symbols。-m32是為了編譯成x86-32的。因為對於x86-64來說，函數的參數是通過寄存器傳遞的。

main的彙編代碼：

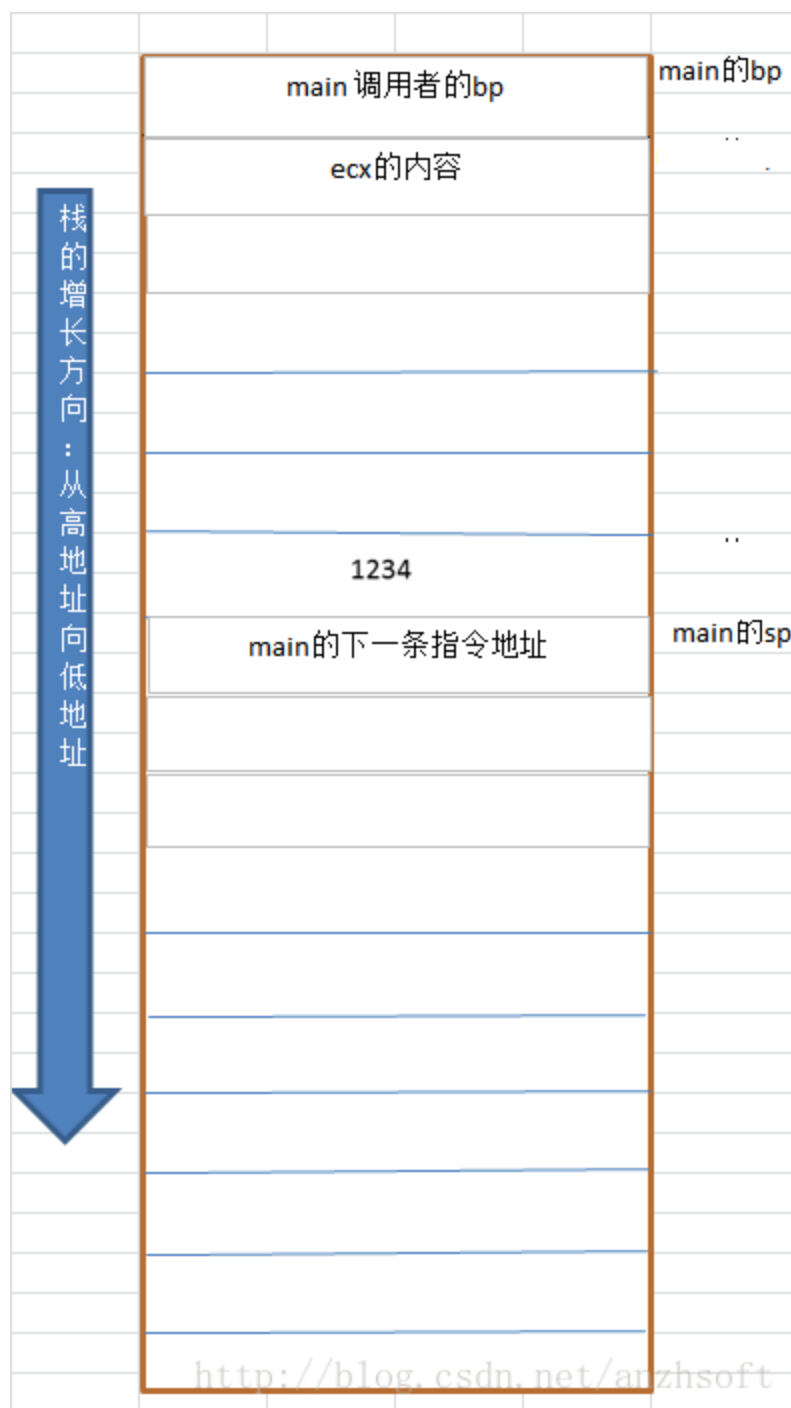
[cpp] view plaincopy 

```
1. main:
2.     leal    4(%esp), %ecx
3.     andl    $-16, %esp
4.     pushl   -4(%ecx)
5.
6.     pushl   %ebp          #1: push %ebp指令把ebp寄存器的值壓棧，同時把esp的值減4
7.     movl    %esp, %ebp    #2 把esp的值傳送給ebp寄存器。
8.                                     #1 + #2 合起來是把原來ebp的值保存在棧上，然後又給ebp賦了新
    值。
9.                                     #2+ ebp指向棧底，而esp指向棧頂，在函數執行過程中esp
10.                                    #2++隨著壓棧和出棧操作隨時變化，而ebp是不動的
11.     pushl   %ecx
12.     subl    $20, %esp     #3 現在esp地址-20/4 = 5，及留出5個地址空間給main的局部變量
13.     movl    $1234, -8(%ebp)#4 局部變量1234 存入ebp - 8 的地址
14.     movl    -8(%ebp), %eax #5 將地址存入eax
15.     movl    %eax, (%esp)  #6 將1234存入esp指向的地址
16.     call    func0(int)    #7 調用func0，注意這是demangle後的函數名，實際是一個地址
17.     movl    $0, %eax
18.     addl    $20, %esp
19.     popl    %ecx
20.     popl    %ebp
21.     leal    -4(%ecx), %esp
22.     ret
```

對於call指令，這個指令有兩個作用：

1. func0函數調用完之後要返回到call的下一條指令繼續執行，所以把call的下一條指令的地址壓棧，同時把esp的值減4。
2. 修改程序計數器eip，跳轉到func0函數的開頭執行。

至此，調用func0的棧就是下面這個樣子：

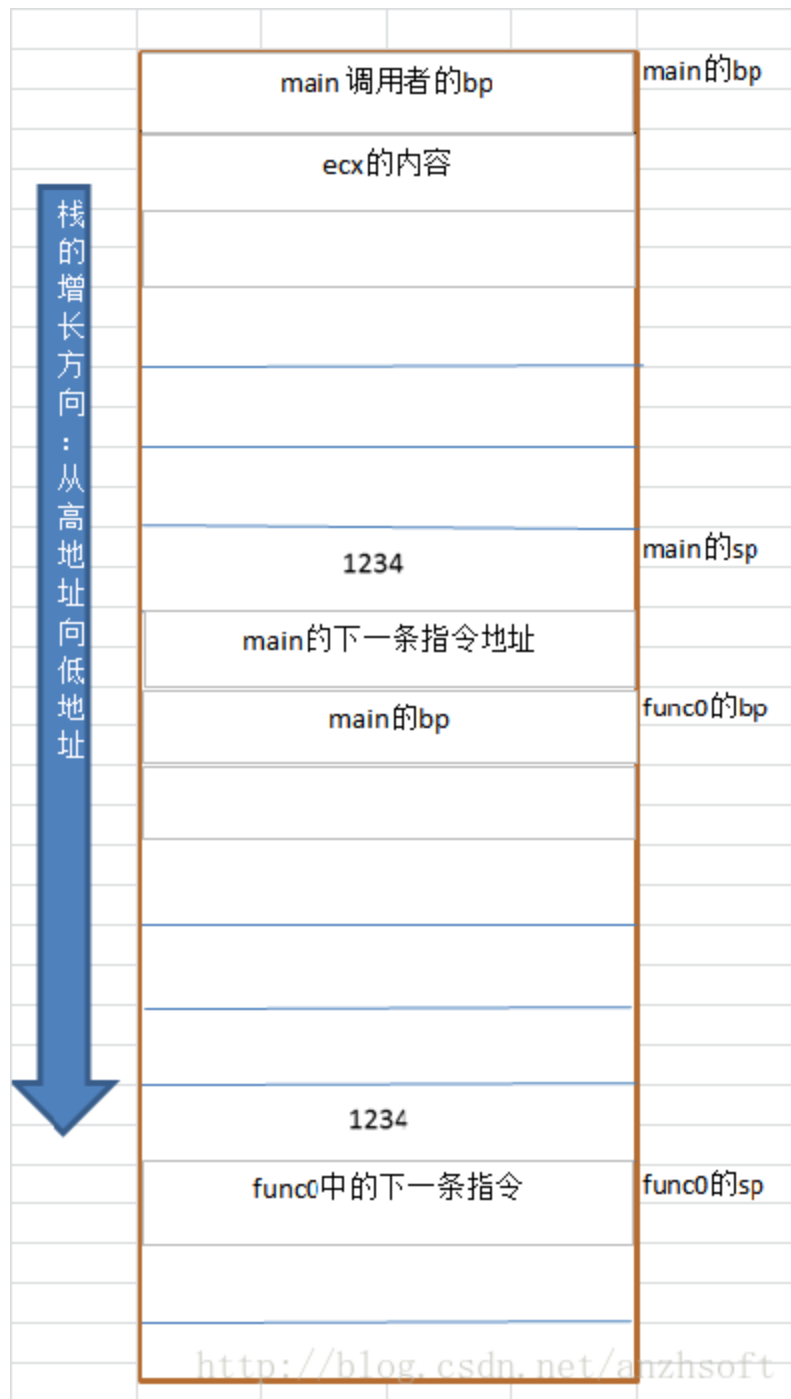


下面看一下func0的彙編代碼：

[plain] view plaincopy C

```
1. func0(int):
2.     pushl   %ebp
3.     movl    %esp, %ebp
4.     subl    $20, %esp
5.     movl    8(%ebp), %eax
6.     movl    %eax, (%esp)
7.     call    func1(int)
8.     movl    %eax, -4(%ebp)
9.     movl    -4(%ebp), %eax
10.    leave
11.    ret
```

需要注意的是esp也是留了5個地址空間給func0使用。並且ebp的下一個地址就是留給局部變量b的，調用棧如圖：



通過調用棧可以看出，8(%ebp)其實就是傳入的參數1234。

func1的代碼：

[plain] view plaincopy

```
1. func1(int):
```

```

2.      pushl   %ebp
3.      movl    %esp, %ebp
4.      subl    $16, %esp
5.      movl    8(%ebp), %eax #去傳入的參數，即1234
6.      addl    $1, %eax # +1 運算
7.      movl    %eax, -4(%ebp)
8.      movl    -4(%ebp), %eax #將計算結果存入eax，這就是返回值
9.      leave
10.     ret

```

leave指令，這個指令是函數開頭的push %ebp和mov %esp,%ebp的逆操作：

1. 把ebp的值賦給esp
2. 現在esp所指向的棧頂保存著foo函數棧幀的ebp，把這個值恢復給ebp，同時esp增加4。
。注意，現在esp指向的是這次調用的返回地址，即上次調用的下一條執行指令。

最後是ret指令，它是call指令的逆操作：

1. 現在esp所指向的棧頂保存著返回地址，把這個值恢復給eip，同時esp增加4，esp指向了當前frame的棧頂。
2. 修改了程序計數器eip，因此跳轉到返回地址繼續執行。

調用棧如下：



至此，func1返回後，控制權交還給func0，當前的棧就退化成func0的棧的情況，因為棧保存了一切信息，因此指令繼續執行。直至func0執行

leave

ret

以同樣的方式將控制權交回給main。

到這裡，你應該知道下面問題的答案了：

1. 局部變量的生命週期，
2. 局部變量是怎麼樣使用內存的；
3. 為什麼傳值不會改變原值（因為編譯器已經幫你做好拷貝了）
4. 為什麼會有棧溢出的錯誤
5. 為什麼有的寫壞棧的程序可以運行，而有的卻會crash（如果棧被破壞的是數據，那麼數據是髒的，不應該繼續運行；如果破壞的是上一層調用的bp，或者返回地址，那麼程序會crash，or unexpected behaviour...）

小節一下：

1. 在32位的機器上，C++的函數調用的參數是存到棧上的。當然gcc可以在函數聲明中添加 `_attribute__((regparm(3)))` 使用eax，edx，ecx傳遞開頭三個參數。
2. 通過bp可以訪問到調用的參數值。
3. 函數的返回地址（函數返回後的執行指令）也是存到棧上的，有目的的修改它可以使程序跳轉到它不應該的地方。。。
4. 如果程序破壞了上一層的bp的地址，或者程序的返回地址，那麼程序就很有可能crash
5. 拿到一個CoreDump，應該首先先看有可能出問題的線程的frame的棧是否完整。
6. 64位的機器上，參數是通過寄存器傳遞的，當然寄存器不夠用就會通過棧來傳遞

Linux Debugging（二）：熟悉AT&T彙編語言

沒想到《[Linux Debugging:使用反彙編理解C++程序函數調用棧](#)》發表了收到了大家的歡迎。但是有網友留言說不熟悉彙編，因此本書列了彙編的基礎語法。這些對於我們平時的調試應該是夠用了。

1 AT&T與Intel彙編語法對比

本科時候大家學的基本上都是Intel的8086彙編語言，微軟採用的就是這種格式的彙編。GCC採用的是AT&T的彙編格式，也叫GAS格式(Gnu ASsembler GNU彙編器)。

1、寄存器命名不同

AT&T	Intel	說明
%eax	eax	Intel的不帶百分號

2、操作數順序不同

AT&T	Intel	說明
movl %eax, %ebx	mov ebx, eax	Intel的目的操作數在前,源操作數在後；AT&T相反

3、常數/立即數的格式不同

AT&T	Intel	說明
movl \$_value,%ebx	mov eax,_value	Intel的立即數前面不帶\$符號

movl \$0xd00d,%ebx	mov ebx,0xd00d	規則同樣適用於16進制的立即數
-----------------------	----------------	-----------------

4、操作數長度標識

AT&T	Intel	說明
movw %ax,%bx	mov bx,ax	<p>Intel的彙編中, 操作數的長度並不通過指令符號來標識。</p> <p>AT&T的格式中, 每個操作都有一個字符後綴, 表明操作數的大小. 例如:mov指令有三種形式:</p> <p>movb 傳送字節</p> <p>movw 傳送字</p> <p>movl 傳送雙字</p> <p>如果沒有指定操作數長度的話, 編譯器將按照目標操作數的長度來設置。比如指令「mov %ax, %bx」, 由於目標操作數bx的長度為word, 那麼編譯器將把此指令等同於「movw %ax, %bx」。</p>

5、尋址方式

AT&T	Intel	說明
imm32(basepointer, r, indexpointer, indexscale)	[basepointer + indexpointer*indexscale + imm32)	兩種尋址的實際結果都應該是 imm32 + basepointer + indexpointer*indexscale

例如: 下面是一些尋址的例子：

AT&T	Intel	說明
mov 4(%ebp), %eax	mov eax, [ebp + 4]	基址尋址（Base Pointer Addressing Mode）,用於訪問結構體成員比較方便，例如一個結構體的基地址保存在eax寄存器中，其中一個成員在結構體內的偏移量是4字節，要把這個成員讀上來就可以用這條指令
data_items(,%edi,4)	[data_items+edi*4]	變址尋址（Indexed Addressing Mode），訪問數組
movl \$addr, %eax	mov eax, addr	直接尋址（Direct Addressing Mode）
movl (%eax), %ebx	mov ebx, [eax]	間接尋址（Indirect Addressing Mode），把eax寄存器的值看作地址，把內存中這個地址處的32位數傳送到ebx寄存器
mov \$12, %eax	mov eax, 12	立即數尋址（Immediate Mode）
mov %eax, %eax	mov eax, eax	寄存器尋址（Register Addressing Mode）

6.跳轉方式不同

AT&T 彙編格式中，絕對轉移和調用指令（jump/call）的操作數前要加上'-'作為前綴，而在Intel 格式中則不需要。

AT&T	Intel	說明
jmp *%eax	jmp %eax	用寄存器%eax中的值作為跳轉目標
jmp *(%eax)	jmp (%eax)	以%eax中的值作為讀入的地址, 從存儲器中讀出跳轉目標

2 求一個數組最大數

通過求一個數組的最大數，來進一步學習AT&T的語法

```

1. #PURPOSE: This program finds the maximum number of a
2. #      set of data items.
3. #
4. #VARIABLES: The registers have the following uses:
5. #
6. # %edi - Holds the index of the data item being examined
7. # %ebx - Largest data item found
8. # %eax - Current data item
9. #
10. # The following memory locations are used:
11. #
12. # data_items - contains the item data. A 0 is used
13. # to terminate the data
14. #
15. .section .data #全局變量
16. data_items:      #These are the data items
17. .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0
18.
19. .section .text
20. .globl _start
21. _start:
22. movl $0, %edi      # move 0 into the index register
23. movl data_items(,%edi,4), %eax # load the first byte of data
24. movl %eax, %ebx     # since this is the first item, %eax is
25.                    # the biggest

```

```

26.
27. start_loop:      # start loop
28.  cml $0, %eax    # check to see if we've hit the end
29.  je loop_exit
30.  incl %edi       # load next value
31.  movl data_items(,%edi,4), %eax
32.  cml %ebx, %eax  # compare values
33.  jle start_loop  # jump to loop beginning if the new
34.                  # one isn't bigger
35.  movl %eax, %ebx  # move the value as the largest
36.  jmp start_loop  # jump to loop beginning
37.
38. loop_exit:
39.  # %ebx is the status code for the _exit system call
40.  # and it already has the maximum number
41.  movl $1, %eax    #1 is the _exit() syscall
42.  int $0x80

```

彙編程序中以.開頭的名稱並不是指令的助記符，不會被翻譯成機器指令，而是給彙編器一些特殊指示，稱為彙編指示（Assembler Directive）或偽操作（Pseudo-operation），由於它不是真正的指令所以加個「偽」字。 .section指示把代碼劃分成若干個段（Section），程序被操作系統加載執行時，每個段被加載到不同的地址，操作系統對不同的頁面設置不同的讀、寫、執行權限。 .data段保存程序的數據，是可讀可寫的，相當於C++程序的全局變量。 .text段保存代碼，是只讀和可執行的，後面那些指令都屬於.text段。

.long指示聲明一組數，每個數佔32；.quad類似，佔64位；.byte是8位；.word 是16位。 .ascii，例如.ascii "Hello world"，聲明11個數，取值為相應字符的ASCII碼。

參考資料：

1. [最簡單的彙編程序](#)
2. [第二個彙編程序](#)
3. <http://blog.chinaunix.net/uid-27717694-id-3942757.html>

最後複習一下lea命令：

mov 4(%ebp) %eax #將%ebp+4地址處所存的值，mov到%eax

leal 4(%ebp) %eax #將%ebp+4的地址值， mov到%eax

leal 可以被mov取代：

addl \$4, %ebp

mov. %ebp, %eax

Linux Debugging (三) : C++函數調用的參數傳遞方法總結 (通過gdb+反彙編)

上一篇文章《[Linux Debugging:使用反彙編理解C++程序函數調用棧](#)》沒想到能得到那麼多人的喜愛，因為那篇文章是以32位的C++普通函數（非類成員函數）為例子寫的，因此只是一個特殊的例子。本文將函數調用時的參數傳遞方法進行一下總結。總結將為C++普通函數、類成員函數；32位和64位進行總結。

建議還是讀一下[Linux Debugging:使用反彙編理解C++程序函數調用棧](#)，這樣本文的結論將非常容易理解，將非常好的為CoreDump分析開一個好頭。而且，它也是32位C++ 普通函數的調用的比較好的例子，畢竟從彙編的角度，將參數如何傳遞的進行了比較好的說明。

1. 32位程序普通函數

普通函數的意思是非class member function

[cpp] view plaincopy 

```
1. void func2(int a, int b)
2. {
3.     a++;
4.     b+ = 2;
5. }
6.
7. int main()
8. {
9.     func2( 1111, 2222);
10.    return 0;
```


```
11. }
```

main函數的彙編：

[cpp] view plaincopy  

```
1. main:
2.     pushl   %ebp
3.     movl    %esp, %ebp
4.     subl    $8, %esp
5.     movl    $2222, 4(%esp)
6.     movl    $1111, (%esp)
7.     call    func2(int, int)
8.     movl    $0, %eax
9.     leave
10.    ret
```

1111是第一個參數，放到了esp指向的地址。2222是第二個參數，放到了高地址。因次我們可以知道，在函數func2中，通過ebp+8可以訪問到第一個參數1111，通過ebp+12可以訪問到第二個參數2222。

[cpp] view plaincopy  

```
1. func2(int, int):
2.     pushl   %ebp
3.     movl    %esp, %ebp
4.     addl    $1, 8(%ebp)
5.     addl    $2, 12(%ebp)
6.     popl    %ebp
7.     ret
```

下面我們使用gdb通過ebp打印一下傳入的參數：

[cpp] view plaincopy  

```
1. anzhsoft@ubuntu:~/linuxDebugging/parameter$ gdb a.out
2. GNU gdb 6.8
3. Copyright (C) 2008 Free Software Foundation, Inc.
4. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5. This is free software: you are free to change and redistribute it.
```

```
6. There is NO WARRANTY, to the extent permitted by law. Type "show copying"
7. and "show warranty" for details.
8. This GDB was configured as "i686-pc-linux-gnu"...
9. (gdb) b func2
10. Breakpoint 1 at 0x8048597: file m32noclass.cpp, line 6.
11. (gdb) r
12. Starting program: /home/anzhsoft/linuxDebugging/parameter/a.out
13.
14. Breakpoint 1, func2 (a=1111, b=2222) at m32noclass.cpp:6
15. warning: Source file is more recent than executable.
16. 6      a++;
17. (gdb) p *(int*)($ebp+8)
18. $1 = 1111
19. (gdb) p *(int*)($ebp+12)
20. $2 = 2222
```

總結：

1. 參數通過棧查傳遞，底地址傳遞從左邊開始的第一個參數
2. 使用gdb可以很方便打印傳入參數

[cpp] view plaincopy 

```
1. (gdb) p *(int*)($ebp+8)
2. $1 = 1111
3. (gdb) p *(int*)($ebp+12)
4. $2 = 2222
```

其實32位的程序也可以使用寄存器傳遞參數。請看下一節。

2. 32位普通函數-通過寄存器傳遞參數

可以使用GCC的擴展功能__attribute__使得參數傳遞可以使用寄存器。

修改第一節的函數：

[cpp] view plaincopy 

```
1. #define STACKCALL __attribute__((regparm(3)))
2. void STACKCALL func4(int a, int b, int c, int d)
3. {
4.     a++;
```



```

5.     b += 2;
6.     c += 3;
7.     d += 4;
8. }
9.
10. int main()
11. {
12.     func4(1111, 2222, 3333, 4444);
13.     return 0;
14. }

```

__attribute__((regparm(3)))意思是使用寄存器

[cpp] view plaincopy

```

1. anzhsoft@ubuntu:~/linuxDebugging/parameter$ gdb a.out
2. GNU gdb 6.8
3. Copyright (C) 2008 Free Software Foundation, Inc.
4. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5. This is free software: you are free to change and redistribute it.
6. There is NO WARRANTY, to the extent permitted by law. Type "show copying"
7. and "show warranty" for details.
8. This GDB was configured as "i686-pc-linux-gnu"...
9. (gdb) b main
10. Breakpoint 1 at 0x80485bb: file m32noclass.cpp, line 14.
11. (gdb) r
12. Starting program: /home/anzhsoft/linuxDebugging/parameter/a.out
13.
14. Breakpoint 1, main () at m32noclass.cpp:14
15. 14      func4(1111, 2222, 3333, 4444);
16. (gdb) s
17. func4 (a=1111, b=2222, c=3333, d=4444) at m32noclass.cpp:6
18. 6      a++;
19. (gdb) i r
20. eax      0x457      1111
21. ecx      0xd05      3333
22. edx      0x8ae      2222
23. ebx      0xb3eff4    11792372
24. esp      0xbf8e9580  0xbf8e9580
25. ebp      0xbf8e958c  0xbf8e958c

```

```

26. esi          0x0      0
27. edi          0x0      0
28. eip          0x80485a3  0x80485a3 <func4(int, int, int, int)+15>
29. eflags       0x282    [ SF IF ]
30. cs           0x73     115
31. ss           0x7b     123
32. ds           0x7b     123
33. es           0x7b     123
34. fs           0x0      0
35. gs           0x33     51
36. (gdb) p *(int*)($ebp+8)
37. $1 = 4444
38. (gdb)

```

可以看到，前三個參數分別通過eax/edx/ecx傳遞，第四個參數通過棧傳遞。這種傳遞方式被稱為fastcall

3. 32位 class member function 參數傳遞方式

我們通過宏USINGSTACK強制使用棧來傳遞參數。

[cpp] view plaincopy 

```

1. #define USINGSTACK __attribute__((regparm(0)))
2. class Test
3. {
4. public:
5.     Test():number(3333){}
6.     void USINGSTACK func2(int a, int b)
7.     {
8.         a++;
9.         b += 2;
10.    }
11. private:
12.     int number;
13. };
14.

```

```

15. int main(int argc, char* argv[])
16. {
17.     Test tInst;
18.     tInst.func2(1111, 2222);
19.     return 0;
20. }

```

通過gdb來打印傳入的參數：

[cpp] view plaincopy 

```

1. anzhsoft@ubuntu:~/linuxDebugging/parameter$ gdb a.out
2. GNU gdb 6.8
3. Copyright (C) 2008 Free Software Foundation, Inc.
4. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5. This is free software: you are free to change and redistribute it.
6. There is NO WARRANTY, to the extent permitted by law. Type "show copying"
7. and "show warranty" for details.
8. This GDB was configured as "i686-pc-linux-gnu"...
9. (gdb) b main
10. Breakpoint 1 at 0x804859d: file m32class.cpp, line 18.
11. (gdb) r
12. Starting program: /home/anzhsoft/linuxDebugging/parameter/a.out
13.
14. Breakpoint 1, main (argc=1, argv=0xbf98a454) at m32class.cpp:18
15. 18     Test tInst;
16. (gdb) n
17. 19     tInst.func2(1111, 2222);
18. (gdb) s
19. Test::func2 (this=0xbf98a39c, a=1111, b=2222) at m32class.cpp:9
20. 9         a++;
21. (gdb) p *(int*)($ebp+12)
22. $1 = 1111
23. (gdb) p *(int*)($ebp+16)
24. $2 = 2222
25. (gdb) p *this
26. $3 = {number = 3333}

```


可以看到，class成員函數的第一個參數是對象的指針。通過這種方式，成員函數可以和非成員函數以類似的方式進行調用。

4. x86-64 class member function 的參數傳遞

在x86-64中，整形和指針型參數的參數從左到右依次保存到rdi，rsi，rdx，rcx，r8，r9中。

浮點型參數會保存到xmm0，xmm1.....。多餘的參數會保持到棧上。

下面這個例子將傳遞九個參數。可以通過它來驗證一下各個寄存器的使用情況：

[cpp] view plaincopy 

```
1. class Test
2. {
3. public:
4.     Test():number(5555){}
5.     void func9(int a, int b, int c, int d, char*str, long e, long f, float h, double i)
6.     {
7.         a++;
8.         b += 2;
9.         c += 3;
10.        d += 4;
11.    }
12. private:
13.     int number;
14. };
15.
16. int main(int argc, char* argv[])
17. {
18.     Test tInst;
19.     tInst.func9(1111, 2222, 3333, 4444, "hello, world!", 6666, 7777, 8.888, 9.999);
20.     return 0;
21. }
```

下面通過gdb驗證各個寄存器的使用情況：

[cpp] view plaincopy 

```

1. khawk-dev-zhanga12:~/study/c++callstack # gdb a.out
2. GNU gdb (GDB) SUSE (7.0-0.4.16)
3. Copyright (C) 2009 Free Software Foundation, Inc.
4. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5. This is free software: you are free to change and redistribute it.
6. There is NO WARRANTY, to the extent permitted by law. Type "show copying"
7. and "show warranty" for details.
8. This GDB was configured as "x86_64-suse-linux".
9. For bug reporting instructions, please see:
10. <http://www.gnu.org/software/gdb/bugs/>...
11. Reading symbols from /root/study/c++callstack/a.out...done.
12. (gdb) b main
13. Breakpoint 1 at 0x40071b: file m64class.cpp, line 19.
14. (gdb) r
15. Starting program: /root/study/c++callstack/a.out
16. Breakpoint 1, main (argc=1, argv=0x7fffffffef188) at m64class.cpp:19
17. 19      Test tInst;
18. (gdb) n
19. 20      tInst.func9(1111, 2222, 3333, 4444, "hello, world!", 6666,7777, 8.888,
      9.999);
20. (gdb) s
21. Test::func9 (this=0x7fffffffef0a0, a=1111, b=2222, c=3333, d=4444, str=0x400918 "hello,
      world!", e=6666, f=7777, h=8.88799953, i=9.9990000000000006)
22.   at m64class.cpp:8
23. 8      a++;
24. (gdb) info reg
25. rax      0x400918 4196632
26. rbx      0x400830 4196400
27. rcx      0xd05    3333
28. rdx      0x8ae    2222
29. rsi      0x457    1111
30. rdi      0x7fffffffef0a0 140737488347296
31. rbp      0x7fffffffef070 0x7fffffffef070
32. rsp      0x7fffffffef070 0x7fffffffef070
33. r8       0x115c    4444
34. r9       0x400918 4196632
35. r10      0xffffffffffffffff -1
36. r11      0x7ffff733d890 140737340758160

```

```

37. r12          0x400620 4195872
38. r13          0x7fffffffef180 140737488347520
39. r14          0x0      0
40. r15          0x0      0
41. rip          0x4007ff 0x4007ff <Test::func9(int, int, int, int, char*, long, long,
float, double)+35>
42. eflags       0x202    [ IF ]
43. cs           0x33     51
44. ss           0x2b     43
45. ds           0x0      0
46. es           0x0      0
47. fs           0x0      0
48. gs           0x0      0
49. fctrl        0x37f    895
50. fstat        0x0      0
51. ftag         0xffff    65535
52. fiseg        0x0      0
53. fioff        0x0      0
54. foseg        0x0      0
55. fooff        0x0      0
56. fop          0x0      0
57. mxcsr        0x1f80    [ IM DM ZM OM UM PM ]
58. (gdb) p *this
59. $1 = {number = 5555}
60. (gdb) p *(char*)$r9@13
61. $2 = "hello, world!"
62. (gdb) p *(int*)($rbp+16)
63. $4 = 6666
64. (gdb) p *(int*)($rbp+24)
65. $5 = 7777
66. (gdb) p *(int*)$rdi
67. $6 = 5555

```

r9存儲的是指針型char *str的字符串。因為寄存器只能存儲6個整形、指針型參數，注意，Test對象的指針佔用了一個。因此參數e和f只能通過棧傳遞。注意每個地址空間佔8個字節。因此rbp+2*8存儲的是e，rbp+3*8存儲的是f。

float h是通過xmm0，double i是通過xmm1傳遞的。這類寄存器的size大小是128bits，當然128個bits可以不填滿。GDB將這些寄存器看成下面這些數據的聯合：

[cpp] view plaincopy 

```
1. union{
2.     float   v4_float[4];
3.     double  v2_double[2];
4.     int8_t  v16_int8[16];
5.     int16_t v8_int16[8];
6.     int32_t v4_int32[4];
7.     int64_t v2_int64[2];
8.     int128_t unit128;
9. }xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6,xmm7;
```

打印方式如下：

[cpp] view plaincopy 

```
1. (gdb) p $xmm0.v4_float[0]
2. $7 = 8.88799953
3. (gdb) p $xmm1.v2_double[0]
4. $8 = 9.99900000000000006
```

總結：

在x86-64中，整形和指針型參數的參數從左到右依次保存到rdi，rsi，rdx，rcx，r8，r9中。

浮點型參數會保存到xmm0，xmm1.....。多餘的參數會保持到棧上。

5. 總結

32位：

- 1) 默認的傳遞方法不使用寄存器，使用ebp+8可以訪問第一個參數，ebp+16可以訪問第二個參數。。。
- 2) 可以使用GCC擴展__attribute__將參數放到寄存器上，依次使用的寄存器是eax/edx/ecx
- 3) 對於class 的member function，調用時第一個參數為this（對象指針）地址。因此函數的第一個參數使用ebp+16才可以訪問到。

4) ebp存儲的是上層的bp的地址。ebp+4存儲的是函數返回時的地址指令。

64位：

1) 默認的傳遞方法是使用寄存器。當然也可以強制使用棧，方式：函數聲明時使用

[cpp] view plaincopy 

1. `__attribute__((regparm(0)))`

2) 整形和指針型參數的參數從左到右依次保存到rdi，rsi，rdx，rcx，r8，r9中。浮點型參數會保存到xmm0，xmm1.....。多餘的參數會保持到棧上。

3) xmm0.....是比較特殊的寄存器，訪問內容時需要注意。

Linux Debugging（四）：使用GDB來理解C++ 對象的內存佈局（多重繼承，虛繼承）

前一段時間再次拜讀《Inside the C++ Object Model》深入探索C++對象模型，有了進一步的理解，因此我也寫了四篇博文算是讀書筆記：

[Program Transformation Semantics（程序轉換語義學）](#)

[The Semantics of Copy Constructors\(拷貝構造函數之編譯背後的行為\)](#)

[The Semantics of Constructors: The Default Constructor（默認構造函數什麼時候會被創建出來）](#)

[The Semantics of Data: Data語義學](#) 深入探索C++對象模型

這些文章都獲得了很大的瀏覽量，雖然類似的博文原來都有，可能不容易被現在仍活躍在CSDN Blog的各位同仁看到吧。因此萌生了接著將這這本書讀完的同時，再接著談一下我的理解，或者說讀書筆記。

關於C++虛函數，很多博文從各個角度來探究虛函數是如何實現的，或者說編譯器是如何實現虛函數的。比較經典的文章有陳皓先生的《[C++虛函數表解析](#)》和《[C++對象內存佈局](#)》。本文通過GDB來從另外一個角度來理解C++ object的內存佈局，一來熟悉語言背後編譯器為了實現語言特性為我們做了什麼；二來熟悉使用GDB來調試程序。

同時，本文也將對如何更好的理解C++語言提供了一個方法：使用GDB，可以很直觀的理解編譯器的實現，從根本上掌握C++！我們不單單只會開車，還應該知道車的內部的構造。

2、帶有虛函數的單一繼承

[cpp] view plaincopy 

```
1. class Parent
2. {
3. public:
4.     Parent():numInParent(1111)
5.     {}
6.     virtual void Foo(){
7.     };
8.     virtual void Boo(){
9.     };
10. private:
11.     int numInParent;
12. };
13.
14. class Child: public Parent
15. {
16. public:
17.     Child():numInChild(2222){}
18.     virtual void Foo(){
19.     }
20.     int numInChild;
21. };
```

編譯時不要忘記-g，使得gdb可以把各個地址映射成函數名。

[cpp] view plaincopy 

```
1. (gdb) set p obj on
2. (gdb) p *this
3. $2 = (Child) {<Parent> = {_vptr.Parent = 0x400a30, numInParent = 1111}, numInChild = 2222}
4. (gdb) set p pretty on
```

```

5. (gdb) p *this
6. $3 = (Child) {
7.   <Parent> = {
8.     _vptr.Parent = 0x400a30,
9.     numInParent = 1111
10.  },
11.  members of Child:
12.  numInChild = 2222
13. }
14. (gdb) p /a (*(void ***)this)[0]@3
15. $4 = {0x4008ec <Child::Foo()>, 0x4008b4 <Parent::Boo()>, 0x6010b0
      <_ZTVN10__cxxabiv120__si_class_type_infoE@@CXXABI_1.3+16>}

```

解釋一下gdb的命令：

set p obj <on/off>: 在C++中，如果一個對象指針指向其派生類，如果打開這個選項，GDB會自動按照虛方法調用的規則顯示輸出，如果關閉這個選項的話，GDB就不管虛函數表了。這個選項默認是off。使用show print object查看對象選項的設置。

set p pretty <on/off>: 按照層次打印結構體。可以從設置前後看到這個區別。on的確更容易閱讀。

[\[cpp\] view plaincopy](#) 

```
1. p /a (*(void ***)this)[0]@3
```

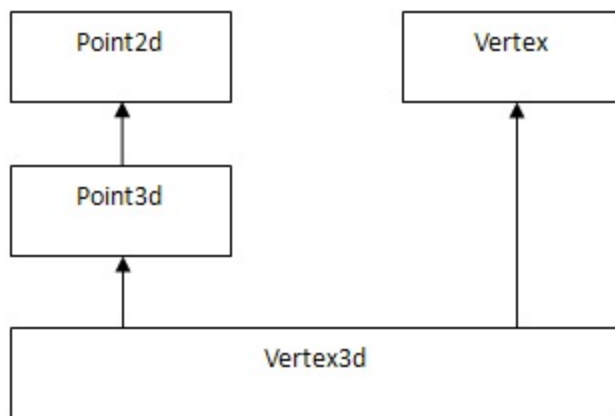
就是打印虛函數表了。因為知道是兩個，可以僅僅打印2個元素。為了知道下一個存儲了什麼信息，我們打印了3個值。實際上後幾個元素存儲了Parent 和Child的typeinfo name和typeinfo。

總結：

對於單一繼承，

1. vptr存儲到了object的開始。
2. 在vptr之後，從Parent開始的data member按照聲明順序依次存儲。

3. 多重繼承，包含有相同的父類



對應的C++codes：

[cpp] view plaincopy

```
1. class Point2d{
2. public:
3.     virtual void Foo(){}
4.     virtual void Boo(){}
5.     virtual void non_overwrite(){}
6. protected:
7.     float _x, _y;
8. };
9.
10. class Vertex: public Point2d{
11. public:
12.     virtual void Foo(){}
13.     virtual void BooVer(){}
14. protected:
15.     Vertex *next;
16. };
17.
18. class Point3d: public Point2d{
19. public:
20.     virtual void Boo3d(){}
```

```

21. protected:
22.     float _z;
23. };
24.
25. class Vertex3d: public Vertex, public Point3d{
26. public:
27.     void test(){}
28. protected:
29.     float mumble;
30. };

```

使用GDB打印的對象內存佈局：

[cpp] view plaincopy 

```

1.  <Vertex> = {
2.      <Point2d> = {
3.          <span style="color:#CC0000;">_vptr.Point2d = 0x400ab0,</span>
4.          _x = 5.88090213e-39,
5.          _y = 0
6.      },
7.      members of Vertex:
8.      next = 0x0
9.  },
10. <Point3d> = {
11.     <Point2d> = {
12.         <span style="color:#990000;">_vptr.Point2d = 0x400ae0,</span>
13.         _x = -nan(0x7fe180),
14.         _y = 4.59163468e-41
15.     },
16.     members of Point3d:
17.     _z = 0
18. },
19. members of Vertex3d:
20. mumble = 0
21. }

```

可見v3d有兩個vptr，指向不同的vtable。首先看一下第一個：

[cpp] view plaincopy 

```
1. (gdb) p /a (*(void ***)this)[0]@5
2. $9 = {0x4008be <Vertex::Foo()>,
3.      0x4008aa <Point2d::Boo()>,
4.      0x4008b4 <Point2d::non_overwrite()>,
5.      0x4008c8 <Vertex::BooVer()>,
6.      0xfffffffffffffffe8}
7. (gdb) p /a (*(void ***)this)[0]@6
8. $10 = {0x4008be <Vertex::Foo()>,
9.        0x4008aa <Point2d::Boo()>,
10.       0x4008b4 <Point2d::non_overwrite()>,
11.       0x4008c8 <Vertex::BooVer()>,
12.       0xfffffffffffffffe8,
13.       0x400b00 <_ZTI8Vertex3d>}
14. (gdb) info addr _ZTI8Vertex3d
15. Symbol "typeinfo for Vertex3d" is at 0x400b00 in a file compiled without debugging.
```

你可以注意到了，vtable打印分行了，可以使用 set p array on將打印的數組分行，以逗號結尾。

注意到該虛函數表以

[cpp] view plaincopy 

```
1. 0xfffffffffffffffe8
```

結尾。在單一繼承中是沒有這個結束標識的。

接著看第二個vtable：

[cpp] view plaincopy 

```
1. (gdb) p /a (*(void ***)this)[1]@5
2. $11 = {0x4008b2 <Point2d::Boo()>,
3.        0x4008bc <Point2d::non_overwrite()>,
4.        0x4008d0 <Vertex::BooVer()>,
5.        0xfffffffffffffffe8,
6.        0x400b00 <_ZTI8Vertex3d>}
7. (gdb) info addr _ZTI8Vertex3d
8. Symbol "typeinfo for Vertex3d" is at 0x400b00 in a file compiled without debugging.
```

當然這個只是為了舉個例子。現實中很少有人這麼幹吧。比如訪問Foo，下面的code將會導致歧義性錯誤：

[cpp] view plaincopy 

```
1. v3d.Boo();
```

error: request for member Boo is ambiguous

multInheritance.cpp:8: error: candidates are: virtual void Point2d::Boo()

只能指定具體的subobject才能進行具體調用：

[cpp] view plaincopy 

```
1. v3d::Vertex::Boo();
```

4. 虛擬繼承

C++ codes:

[cpp] view plaincopy 

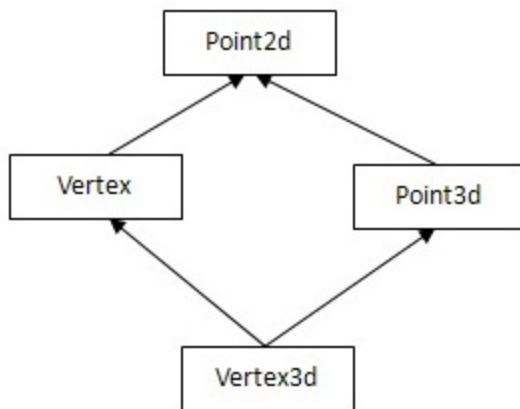
```
1. class Point2d{
2. public:
3.     virtual void Foo(){}
4.     virtual void Boo(){}
5.     virtual void non_overwrite(){}
6. protected:
7.     float _x, _y;
8. };
9.
10. class Vertex: public virtual Point2d{
11. public:
12.     virtual void Foo(){}
13.     virtual void BooVer(){}
14. protected:
15.     Vertex *next;
16. };
```

```

17.
18. class Point3d: public virtual Point2d{
19. public:
20.     virtual void Boo3d(){}
21. protected:
22.     float _z;
23. };
24.
25. class Vertex3d: public Vertex, public Point3d{
26. public:
27.     void test(){}
28. protected:
29.     float mumble;
30. };

```

繼承關係圖：



使用gdb打印object的內存佈局：

[cpp] view plaincopy 

```

1. (gdb) p *this
2. $10 = (Vertex3d) {
3.     <Vertex> = {
4.         <Point2d> = {
5.             <span style="color:#990000;">_vptr.Point2d = 0x400b70,</span>
6.             _x = 0,

```

```

7.      _y = 0
8.    },
9.    members of Vertex:
10.    <span style="color:#660000;">_vptr.Vertex = 0x400b18,</span>
11.    next = 0x4009c0
12.  },
13.  <Point3d> = {
14.    members of Point3d:
15.    <span style="color:#990000;">_vptr.Point3d = 0x400b40,</span>
16.    _z = 5.87993804e-39
17.  },
18.  members of Vertex3d:
19.  mumble = 0
20. }

```

gdb打印的vptr相關：

[cpp] view plaincopy 

```

1. (gdb) p /a (*(void ***)this)[0]@60
2. $25 = {0x400870 <Vertex::Foo()>,
3. 0x40087a <Vertex::BooVer()>,
4. 0x10,
5. 0xfffffffffffffffff0,
6. 0x400c80 <_ZTI8Vertex3d>, #"typeinfo for Vertex3d"
7. 0x400884 <Point3d::Boo3d()>,
8. 0x0,
9. 0x0,
10. 0xfffffffffffffffffe0,
11. 0xfffffffffffffffffe0,
12. 0x400c80 <_ZTI8Vertex3d>, #"typeinfo for Vertex3d"
13. 0x400866 <_ZTV0_n24_N6Vertex3FooEv>, #"virtual thunk to Vertex::Foo()"
14. 0x400852 <Point2d::Boo()>,
15. 0x40085c <Point2d::non_overwrite()>,
16. 0x0,
17. 0x0,
18. 0x0,
19. 0x20,
20. 0x0,

```



```
21. 0x400cc0 <_ZTI6Vertex>, #"typeinfo for Vertex"
22. 0x400870 <Vertex::Foo()>,
23. 0x40087a <Vertex::BooVer()>,
24. 0x0,
25. 0x0,
26. 0xfffffffffffffffe0,
27. 0xfffffffffffffffe0,
28. 0x400cc0 <_ZTI6Vertex>, #"typeinfo for Vertex"
29. 0x400866 <_ZTv0_n24_N6Vertex3FooEv>, #"virtual thunk to Vertex::Foo\(\)"
30. 0x400852 <Point2d::Boo()>,
31. 0x40085c <Point2d::non_overwrite()>,
32. 0x0,
33. 0x0,
34. 0x0,
35. 0x10,
36. 0x0,
37. 0x400d00 <_ZTI7Point3d>, #"typeinfo for Point3d"
38. 0x400884 <Point3d::Boo3d()>,
39. 0x0,
40. 0x0,
41. 0x0,
42. 0xffffffffffffffff0,
43. 0x400d00 <_ZTI7Point3d>, #"typeinfo for Point3d"
44. 0x400848 <Point2d::Foo()>,
45. 0x400852 <Point2d::Boo()>,
46. 0x40085c <Point2d::non_overwrite()>,
47. 0x6020b0 <_ZTVN10__cxxabiv121__vmi_class_type_infoE@@CXXABI_1.3+16>,
48. 0x400d28 <_ZTS8Vertex3d>,
49. 0x200000002,
50. 0x400cc0 <_ZTI6Vertex>, #"typeinfo for Vertex"
51. 0x2,
52. 0x400d00 <_ZTI7Point3d>, #"typeinfo for Point3d"
53. 0x1002,
54. 0x0,
55. 0x6020b0 <_ZTVN10__cxxabiv121__vmi_class_type_infoE@@CXXABI_1.3+16>,
56. 0x400d32 <_ZTS6Vertex>,
57. 0x100000000,
58. 0x400d40 <_ZTI7Point2d>,
59. 0xffffffffffffffe803,
```

```
60. 0x0,  
61. 0x0}
```

有興趣的話可以看一下反彙編的vtable的構成。

參考：

1. <http://stackoverflow.com/questions/6191678/print-c-vtables-using-gdb>
2. <http://stackoverflow.com/questions/18363899/how-to-display-a-vtable-by-name-using-gdb>

Linux Debugging (五) : coredump 分析入門

作為工作幾年的老程序猿，肯定會遇到coredump，log severity設置的比較高，導致可用的log無法分析問題所在。更悲劇的是，這個問題不好復現！所以現在你手頭唯一的線索就是這個程序的屍體：coredump。你不得不通過它，來尋找問題根源。

通過上幾篇文章，我們知道了函數參數是如何傳遞的，和函數調用時棧是如何變化的；當然了還有AT&T的彙編基礎，這些，已經可以使我們具備了一定的調試基礎。其實，很多調試還是需要經驗+感覺的。當然說這句話可能會被打。但是你不得不承認，隨著調試的增多，你的很多推斷在解決問題時顯得很重要，因此，我們需要不斷積累經驗，來面對各種case。

導致coredump的原因很多，比如死鎖，這些還不要操作系統相關的知識，這些問題的分析不在本文的討論範圍之內。大家敬請期待接下來的文章吧！本文從一個非常典型的coredump入手。

請下載本文用到的coredump：[Linux Debugging: coredump 分析入門的材料](#)

首先使用gdb a.out core.25992打開這個core

看一下backtrace是什麼：

[cpp] view plaincopy

1. Program received signal SIGSEGV, Segmentation fault.

2. 0x000000000400703 in __do_global_dtors_aux ()
3. (gdb) bt
4. Cannot access memory at address 0x303938373635343b

出錯的地方很奇怪，而且整個callstack都被破壞了，因此首先看一下寄存器和bp是否正常：

[cpp] view plaincopy 

1. (gdb) i r
2. rax 0x7fffffff040 140737488347200
3. rbx 0x400820 4196384
4. rcx 0x3332312c21646c72 3689065110378409074
5. rdx 0x0 0
6. rsi 0x40091d 4196637
7. rdi 0x7fffffff059 140737488347225
8. rbp 0x3039383736353433 0x3039383736353433
9. rsp 0x7fffffff060 0x7fffffff060
10. r8 0x30393837363534 13573712489362740
11. r9 0x7ffff7dc3680 140737351792256
12. r10 0xffffffffffffff -1
13. r11 0x7ffff7389ae0 140737341070048
14. r12 0x400660 4195936
15. r13 0x7fffffff0180 140737488347520
16. r14 0x0 0
17. r15 0x0 0
18. rip 0x400703 0x400703 <__do_global_dtors_aux+83>
19. eflags 0x10202 [IF RF]
20. cs 0x33 51
21. ss 0x2b 43
22. ds 0x0 0
23. es 0x0 0
24. fs 0x0 0
25. gs 0x0 0
26. fctrl 0x37f 895
27. fstat 0x0 0
28. ftag 0xffff 65535
29. fiseg 0x0 0
30. fioff 0x0 0

```

31. foseg          0x0      0
32. fooff         0x0      0
33. fop           0x0      0
34. mxcsr         0x1f80   [ IM DM ZM OM UM PM ]
35. (gdb) x/i $rip
36. 0x400703 <__do_global_dtors_aux+83>:   fidivl -0x1e(%rdx)
37. (gdb) x/20i $rip-10
38. 0x4006f9 <__do_global_dtors_aux+73>:   add    %cl,-0x75(%rax)
39. 0x4006fc <__do_global_dtors_aux+76>:   adc    $0x200947,%eax
40. 0x400701 <__do_global_dtors_aux+81>:   cmp    %rbx,%rdx
41. 0x400704 <__do_global_dtors_aux+84>:   jnb    0x4006e8 <__do_global_dtors_aux+56>
42. 0x400706 <__do_global_dtors_aux+86>:   movb   $0x1,0x200933(%rip)          # 0x601040
    <completed.6159>
43. 0x40070d <__do_global_dtors_aux+93>:   add    $0x8,%rsp
44. 0x400711 <__do_global_dtors_aux+97>:   pop    %rbx
45. 0x400712 <__do_global_dtors_aux+98>:   leaveq
46. 0x400713 <__do_global_dtors_aux+99>:   retq
47. 0x400714 <__do_global_dtors_aux+100>:  nopw   %cs:0x0(%rax,%rax,1)
48. 0x400720 <frame_dummy>: push    %rbp
49. 0x400721 <frame_dummy+1>:          cmpq   $0x0,0x2006df(%rip)          # 0x600e08
    <__JCR_LIST__>
50. 0x400729 <frame_dummy+9>:          mov    %rsp,%rbp
51. 0x40072c <frame_dummy+12>:         je     0x400748 <frame_dummy+40>
52. 0x40072e <frame_dummy+14>:         mov    $0x0,%eax
53. 0x400733 <frame_dummy+19>:         test   %rax,%rax
54. 0x400736 <frame_dummy+22>:         je     0x400748 <frame_dummy+40>
55. 0x400738 <frame_dummy+24>:         mov    $0x600e08,%edi
56. 0x40073d <frame_dummy+29>:         mov    %rax,%r11
57. 0x400740 <frame_dummy+32>:         leaveq

```

rbp的值很奇怪，基本確定棧被破壞了（bt不正常，也應該看一下棧是否出問題了）。打印一下棧的內容，看是否棧被寫壞了：

[cpp] view plaincopy 

```

1. (gdb) x/30c $rsp-20
2. 0x7fffffffef04c: 33 '!' 44 ',' 49 '1' 50 '2' 51 '3' 52 '4' 53 '5' 54 '6'

```

```

3. 0x7fffffffef054: 55 '7' 56 '8' 57 '9' 48 '0' 0 '\000' 7 '\a' 64 '@' 0
   '\000'
4. 0x7fffffffef05c: 0 '\000' 0 '\000' 0 '\000' 0 '\000' 0
   '\000' 0 '\000' 0 '\000' 0 '\000'
5. 0x7fffffffef064: 0 '\000' 0 '\000' 0 '\000' 0 '\000' -21
   '\353' 5 '\005'

```

我們看到了特殊的字符!,1234567890。當然實際的生產環境可能不會這麼簡單，比如筆者曾經遇到過這個字符串是/local/share/tracker_...這種目錄的字符串，後來發現拼接路徑的時候出現錯誤導致路徑非常長，在路徑拷貝的時候出現了寫壞棧的情況。

多打印一下棧的內容：

[cpp] view plaincopy

```

1. (gdb) x/40c $rsp-40
2. 0x7fffffffef038: -100 '\234' 7 '\a' 64 '@' 0 '\000' 1 '\001' 0
   '\000' 0 '\000' 0 '\000'
3. 0x7fffffffef040: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 44 ',' 32 ' ' 119 'w'
4. 0x7fffffffef048: 111 'o' 114 'r' 108 'l' 100 'd' 33 '!' 44 ',' 49 '1' 50 '2'
5. 0x7fffffffef050: 51 '3' 52 '4' 53 '5' 54 '6' 55 '7' 56 '8' 57 '9' 48 '0'
6. 0x7fffffffef058: 0 '\000' 7 '\a' 64 '@' 0 '\000' 0 '\000' 0
   '\000' 0 '\000' 0 '\000'

```

可以看出，字符串"Hello, World!,1234567890"這個字符串溢出導致棧被破壞。

我們不應該用rbp打印嗎？

還記得在函數返回時，rbp會恢復為上一層調用者的bp嗎？因為字符串溢出/越界，導致已經恢復不了原來的bp了。

這個bug很簡單。實際上，有的coredump就是這種無心的錯誤啊。

Linux Debugging（六）：動態庫注入、ltrace、strace、Valgrind

實際上，Linux的調試方法非常多，針對不同的問題，不同的場景，不同的應用，都有不同的方法。很難去概括。本篇文章主要涉及本專欄還沒有涵蓋，但是的確有很重要的方法。本文主要包括動態庫注入調試；使用ltrace命令處理動態庫的調試；使用strace調試系統調用的問題；Valgrind的簡要介紹。

1. 動態庫注入

如何排除其他library的調用問題？動態庫注入（library injection）有可能會讓你事半功倍。

一個大型的軟件系統，會用到非常多的動態庫。那麼如果該動態庫的一個api調用出了問題，而調用該api的地方非常非常多，不同的調用都分散的記錄在不同的log裡。那麼，如何快速的找到是哪個調用者出的問題？當然我們可以通過動態庫注入的方式去調試。

下面的代碼hook了兩個常見的函數memcpy和socket：

[cpp] view plaincopy 

```
1. void _init(void)
2. {
3.     mtrace();
4.     printf("HOOKing: hello\n");
5. }
6.
7. void _fini(void)
8. {
9.     printf("HOOKing: goodbye\n");
10. }
11.
12. typedef void* (*real_memcpy)(void*, const void*, size_t);
13. void *memcpy( void *dest, const void*src, size_t size)
14. {
15.     real_memcpy real = dlsym((void*)-1, "memcpy");
16.     printf("Coping from %p to %p, size %d\n", src, dest, size);
17.
18.     return real(dest, src, size);
```

```

19. }
20. typedef int (*real_socket)(int socket_family, int socket_type, int protocol);
21.
22. int socket(int socket_family, int socket_type, int protocol)
23. {
24.     printf(" SOCKET family %d, SOCKET type %d, SOECKT protocol %d", socket_family,
        socket_type, protocol);
25.
26.     real_socket sock = dlsym((void*)-1, "socket");
27.
28.     return sock(socket_family, socket_type, protocol );
29. }

```

將上述代碼編譯成動態庫後，需要指定環境變量LD_PRELOAD為上述動態庫。它的作用是強制load指定的動態庫，即使不需要它。你可以在上面的動態庫裡添加你想要的任何函數。

2. ltrace

ltrace能夠跟蹤進程的庫函數調用,它會顯現出哪個庫函數被調用。

還是使用hello，world進行簡單的瞭解吧：

[cpp] view plaincopy 

```

1. #include <stdio.h>
2. int main ()
3. {
4.     printf("Hello world!\n");
5.     return 0;
6. }

```

使用ltrace + 命令可以啟動對任何程序的調試，上述hello world的ltrace為：

[cpp] view plaincopy 

```

1. __libc_start_main(0x8048354, 1, 0xbf869aa4, 0x8048390, 0x8048380 <unfinished ...>
2. puts("Hello world!")Hello world!
3. ) = 13
4. +++ exited (status 0) +++

```

其實ltrace是一個不用去閱讀庫的實現代碼，而去學習庫的整體調用棧的很好的方式。當然了結合代碼你可以得到更加詳細的實現。

3. strace

strace會跟蹤程序系統調用。所以如果是由於程序的系統調用出問題的話，使用strace可以很快的進行問題定位。上述hello world的strace輸出為：

[cpp] view plaincopy 

```
1.  execve("./hello", [ "./hello" ], [ /* 30 vars */ ]) = 0
2.  brk(0)                                     = 0x83d4000
3.  mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f8a000
4.  access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or directory)
5.  open("/etc/ld.so.cache", O_RDONLY)         = 3
6.  fstat64(3, {st_mode=S_IFREG|0644, st_size=80846, ...}) = 0
7.  mmap2(NULL, 80846, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f76000
8.  close(3)                                   = 0
9.  open("/lib/libc.so.6", O_RDONLY)           = 3
10. read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\000?\270"... , 512) = 512
11. fstat64(3, {st_mode=S_IFREG|0755, st_size=1576952, ...}) = 0
12. mmap2(0xb6e000, 1295780, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
    0xb6e000
13. mmap2(0xca5000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
    0x137) = 0xca5000
14. mmap2(0xca8000, 9636, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
    0) = 0xca8000
15. close(3)                                   = 0
16. mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f75000
17. set_thread_area({entry_number:-1 -> 6, base_addr:0xb7f756c0, limit:1048575,
    seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
    useable:1}) = 0
18. mprotect(0xca5000, 8192, PROT_READ)        = 0
19. mprotect(0xb6a000, 4096, PROT_READ)        = 0
20. munmap(0xb7f76000, 80846)                  = 0
21. fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
22. mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f89000
23. write(1, "Hello world!\n", 13Hello world!
24. )                                           = 13
25. exit_group(0)                              = ?
```


26. Process 2874 detached

可以看到strace的輸出非常豐富。

4. Valgrind

Valgrind是一款用於內存調試、內存洩漏檢測以及性能分析的軟件開發工具。

Valgrind包括如下一些工具：

1. Memcheck。這是valgrind應用最廣泛的工具，一個重量級的內存檢查器，能夠發現開發中絕大多數內存錯誤使用情況，比如：使用未初始化的內存，使用已經釋放了內存，內存訪問越界等。這也是本文將重點介紹的部分。
2. Callgrind。它主要用來檢查程序中函數調用過程中出現的問題。
3. Cachegrind。它主要用來檢查程序中緩存使用出現的問題。
4. Helgrind。它主要用來檢查多線程程序中出現的競爭問題。
5. Massif。它主要用來檢查程序中堆棧使用中出現的問題。
6. Extension。可以利用core提供的功能，自己編寫特定的內存調試工具。

IBM Developer有一篇很好的文章：<https://www.ibm.com/developerworks/cn/linux/l-cn-valgrind/>

Linux Debugging（七）：使用反彙編理解動態庫函數調用方式 GOT/PLT

本文主要講解動態庫函數的地址是如何在運行時被定位的。首先介紹一下PIC和Relocatable的動態庫的區別。然後講解一下GOT和PLT的理論知識。GOT是Global Offset Table，是保存庫函數地址的區域。程序運行時，庫函數的地址會設置到GOT中。由於動態庫的函數是在使用時才被加載，因此剛開始GOT表是空的。地址的設置就涉及到了PLT，Procedure Linkage Table，它包含了一些代碼以調用庫函數，它可以被理解成一系

列的小函數，這些小函數的數量其實就是庫函數的被使用到的函數的數量。簡單來說，PLT就是跳轉到GOT中所設置的地址而已。如果這個地址是空，那麼PLT的跳轉會巧妙的調用`_dl_runtime_resolve`去獲取最終地址並設置到GOT中去。由於庫函數的地址在運行時不會變，因此GOT一旦設置以後PLT就可以直接跳轉到庫函數的真實地址了。最後使用反彙編驗證和跳轉流程圖對上述結論加深理解。

1. 背景-PIC VS Relocatable

在 Linux 下製作動態鏈接庫，「標準」的做法是編譯成位置無關代碼（Position Independent Code，PIC），然後鏈接成一個動態鏈接庫。那麼什麼是PIC呢？如果是非PIC的，那麼會有什麼問題？

(1) 可重定位代碼（relocatable code）：Windows DLL 以及不使用 `-fPIC` 的 Linux so。

生成動態庫時假定它被加載在地址 0 處。加載時它會被加載到一個地址（base），這時要進行一次重定位（relocation），把代碼、數據段中所有的地址加上這個 base 的值。這樣代碼運行時就能使用正確的地址了。當要再加載時根據加載到的位置再次重定位的。（因為它裡面的代碼並不是位置無關代碼）。因為so被每個程序加載的位置都不同,顯然這些重定位後的代碼也不同,當然不能共享。如果被多個應用程序共同使用,那麼它們必須每個程序維護一份so的代碼副本了。當然，主流現代操作系統都啟用了分頁內存機制，這使得重定位時可以使用 COW（copy on write）來節省內存（32 位 Windows 就是這樣做的）；然而，頁面的粒度還是比較大的（例如 IA32 上是 4KiB），至少對於代碼段來說能節省的相當有限。不能共享就失去了共享庫的好處,實際上和靜態庫的區別並不大,在運行時佔用的內存是類似的,僅僅是二進制代碼佔的硬盤空間小一些。

(2) 位置無關代碼（position independent code）：使用 -fPIC 的 Linux so。

這樣的代碼本身就能被放到線性地址空間的任意位置，無需修改就能正確執行。通常的方法是獲取指令指針（如 x86 的 EIP 寄存器）的值，加上一個偏移得到全局變量/函數的地址。AMD64 下，必須使用位置無關代碼。x86 下，在創建 so 時會有一個警告。但是這樣的 so 可以完全正常工作。PIC 的缺點主要就是代碼有可能長一些。例如 x86，由於不能直接使用 [EIP+constant] 這樣的尋址方式，甚至不能直接將 EIP 的值交給其他寄存器，要用到 GOT（global offset table）來定位全局變量和函數。這樣導致代碼的效率略低。PIC 的加載速度稍快，因為不需要做重定位。多個進程引用同一個 PIC 動態庫時，可以共用內存。這一個庫在不同進程中的虛擬地址不同，但操作系統顯然會把它們映射到同一塊物理內存上。

因此，除非你的 so 不會被共享，否則還是加上 -fPIC 吧。

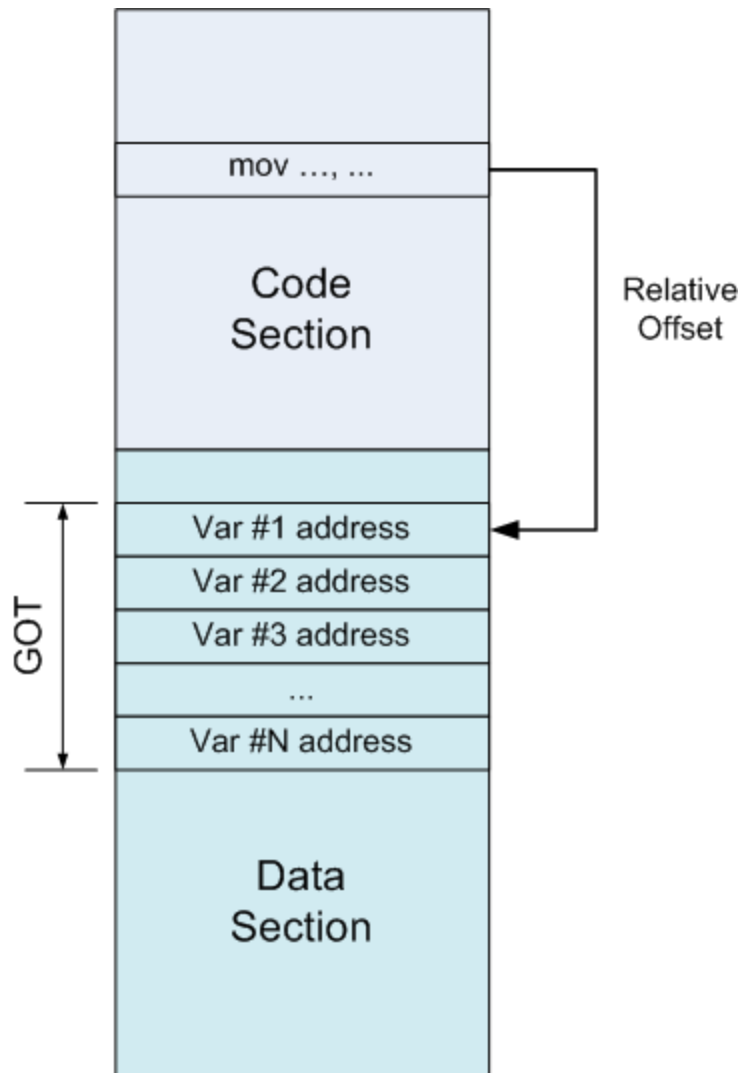
2. GOT和PLT

我們都知道動態庫是在運行時綁定的。那麼編譯器是如何找到動態鏈接庫裡面的函數的地址呢？事實上，直到我們第一次調用這個函數，我們並不知道這個函數的地址，這個功能要做延遲綁定 lazy bind。因為程序的分支很多，並不是所有的分支都能跑到，想想我們的異常處理，異常處理分支的動態鏈接庫裡面的函數也許永遠跑不到，所以，啟動時解析所有出現過的動態庫裡面的函數是個浪費的辦法，降低性能並且沒有必要。

Global Offset Table (GOT)

在位置無關代碼中，一般不能包含絕對虛擬地址（如共享庫）。當在程序中引用某個共享庫中的符號時，編譯鏈接階段並不知道這個符號的具體位置，只有等到動態鏈接器將所需要的共享庫加載時進內存後，也就是在運行階段，符號的地址才會最終確定。因此，需要有一個數據結構來保存符號的絕對地址，這就是GOT表的作用，GOT表中每項保存程序中引用其它符號的絕對地址。這樣，程序就可以通過引用GOT表來獲得某個符號的地址。

在x86結構中，GOT表的前三項保留，用於保存特殊的數據結構地址，其它的各項保存符號的絕對地址。對於符號的動態解析過程，我們只需要瞭解的就是第二項和第三項，即GOT[1]和GOT[2]：GOT[1]保存的是一個地址，指向已經加載的共享庫的鏈表地址；GOT[2]保存的是一個函數的地址，定義如下：GOT[2] = &_dl_runtime_resolve，這個函數的主要作用就是找到某個符號的地址，並把它寫到與此符號相關的GOT項中，然後將控制轉移到目標函數，後面我們會詳細分析。GOT示意如下圖，GOT表slot的數量就是3 + number of functions to be loaded.



Procedure Linkage Table (PLT)

過程鏈接表（PLT）的作用就是將位置無關的函數調用轉移到絕對地址。在編譯鏈接時，鏈接器並不能控制執行從一個可執行文件或者共享文件中轉移到另一個中（如前所說，這時候函數的地址還不能確定），因此，鏈接器將控制轉移到PLT中的某一項。而PLT通過引用GOT表中的函數的絕對地址，來把控制轉移到實際的函數。

在實際的可執行程序或者共享目標文件中，GOT表在名稱為.got.plt的section中，PLT表在名稱為.plt的section中。

3. 反彙編

我們使用的代碼是：

[cpp] view plaincopy 

```
1. #include <iostream>
2. #include <stdlib.h>
3. void fun(int a)
4. {
5.     a++;
6. }
7.
8. int main()
9. {
10.     fun(1);
11.     int x = rand();
12.     return 0;
13. }
```

動態庫裡面需要重定位的函數在.got.plt這個段裡面，通過readelf我們可以看到，它一共有六個地址空間，前三個我們已經解釋了。說明該程序預留了三個所需要重新定位的函數。因此用不到的函數是永遠不會被加載的。

[cpp] view plaincopy 

```
1. [23] .dynamic          DYNAMIC          0000000000600e10 00000e10
2.     00000000000001d0 0000000000000010 WA      8      0      8
3. [24] .got              PROGBITS          0000000000600fe0 00000fe0
4.     0000000000000008 0000000000000008 WA      0      0      8
5. [25] .got.plt          PROGBITS          0000000000600fe8 00000fe8
6.     0000000000000048 0000000000000008 WA      0      0      8
```

反彙編main函數：

[cpp] view plaincopy  

```
1. (gdb) disas main
2. Dump of assembler code for function main:
3. 0x000000000400549 <main+0>:    push    %rbp
4. 0x00000000040054a <main+1>:    mov     %rsp,%rbp
5. 0x00000000040054d <main+4>:    sub     $0x10,%rsp
6. 0x000000000400551 <main+8>:    mov     $0x1,%edi
7. 0x000000000400556 <main+13>:   callq   0x40053c <fun>
8. 0x00000000040055b <main+18>:   callq   0x400440 <rand@plt>
9. 0x000000000400560 <main+23>:   mov     %eax,-0x4(%rbp)
10. 0x000000000400563 <main+26>:   mov     $0x0,%eax
11. 0x000000000400568 <main+31>:   leaveq  0
12. 0x000000000400569 <main+32>:   retq
13. End of assembler dump.
```

可以看到其實調用我們自定義的fun和系統庫函數rand形成的彙編差不多，沒有額外的處理。接著向下看rand：

[cpp] view plaincopy  

```
1. (gdb) disas 0x400440
2. Dump of assembler code for function rand@plt:
3. 0x000000000400440 <rand@plt+0>:    jmpq     *0x200bc2(%rip)          # 0x601008
   <_GLOBAL_OFFSET_TABLE_+32>
4. 0x000000000400446 <rand@plt+6>:    pushq    $0x1
5. 0x00000000040044b <rand@plt+11>:   jmpq     0x400420
6. End of assembler dump.
```

真正有意思的在# 0x601008 <_GLOBAL_OFFSET_TABLE_+32>。也就是rand@plt首先會跳到這裡。我們看一下這裡是什麼：

[cpp] view plaincopy  

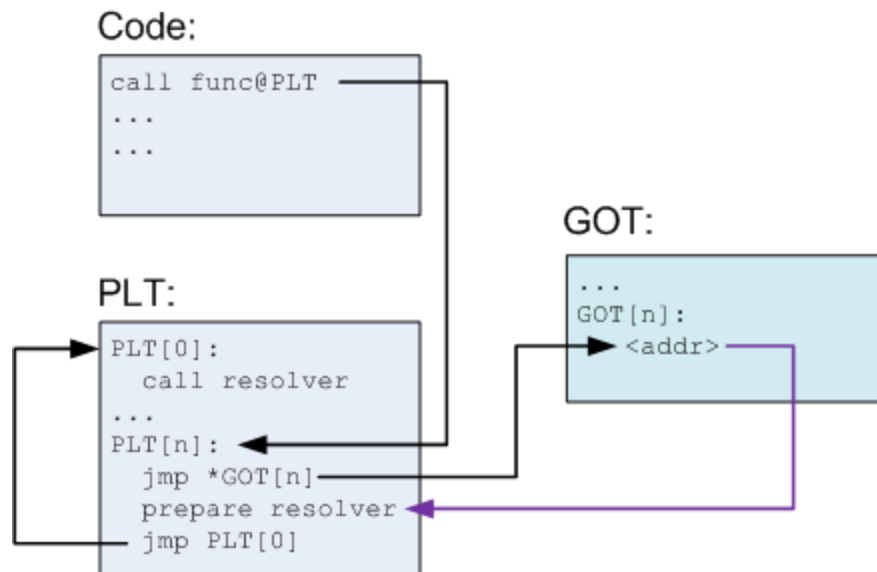
```
1. (gdb) x 0x601008
2. 0x601008 <_GLOBAL_OFFSET_TABLE_+32>: 0x00400446
```

接著看0x00400446是什麼：

[cpp] view plaincopy  

1. (gdb) x/5i 0x00400446
2. 0x400446 <rand@plt+6>: pushq \$0x1
3. 0x40044b <rand@plt+11>: jmpq 0x400420

可能你注意到了，這裡的處理是和剛才的rand@plt的jmpq一樣。都是將0x1入棧，然後jmpq 0x400420。因此這樣就避免了GOT表是否為真實值的檢查：如果是空，那麼去尋址；否則直接調用。



其實接下來處理的就是調用_dlopen_resolve_()函數，該函數最終會尋址到rand的真正地址並且會調用_dlopen_fixup來將rand的實際地址填入GOT表中。

我們將整個程序執行完，然後看一下0x601008

<_GLOBAL_OFFSET_TABLE_+32>是否已經修改成rand的實際地址：

[cpp] view plaincopy

1. (gdb) x 0x601008
2. 0x601008 <_GLOBAL_OFFSET_TABLE_+32>: 0xf7ab6470

可以看到，rand的地址已經修改為0xf7ab6470了。然後可以通過maps確認一下是否libc load在這個地址：

[cpp] view plaincopy 

```
1. (gdb) shell cat /proc/`pgrep a.out`/maps
2. 00400000-00401000 r-xp 00000000 08:02 491638
   /root/study/got/a.out
3. 00600000-00601000 r--p 00000000 08:02 491638
   /root/study/got/a.out
4. 00601000-00602000 rw-p 00001000 08:02 491638
   /root/study/got/a.out
5. 7ffff7a80000-7ffff7bd5000 r-xp 00000000 08:02 327685
   /lib64/libc-2.11.1.so
6. 7ffff7bd5000-7ffff7dd4000 ---p 00155000 08:02 327685
   /lib64/libc-2.11.1.so
7. 7ffff7dd4000-7ffff7dd8000 r--p 00154000 08:02 327685
   /lib64/libc-2.11.1.so
8. 7ffff7dd8000-7ffff7dd9000 rw-p 00158000 08:02 327685
   /lib64/libc-2.11.1.so
9. 7ffff7dd9000-7ffff7dde000 rw-p 00000000 00:00 0
10. 7ffff7dde000-7ffff7dfd000 r-xp 00000000 08:02 327698
    /lib64/ld-2.11.1.so
11. 7ffff7fc4000-7ffff7fc7000 rw-p 00000000 00:00 0
12. 7ffff7ffa000-7ffff7ffb000 rw-p 00000000 00:00 0
13. 7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0                                [vdso]
14. 7ffff7ffc000-7ffff7ffd000 r--p 0001e000 08:02 327698
    /lib64/ld-2.11.1.so
15. 7ffff7ffd000-7ffff7ffe000 rw-p 0001f000 08:02 327698
    /lib64/ld-2.11.1.so
16. 7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
17. ffffffffefa000-fffffffffff000 rw-p 00000000 00:00 0                                [stack]
18. ffffffffff600000-fffffffffff601000 r-xp 00000000 00:00 0                                [vsyscall]
```

沒有問題，如我們所分析的那樣：

[cpp] view plaincopy 

1. 7ffff7a80000-7ffff7bd5000 r-xp 00000000 08:02 327685
/lib64/libc-2.11.1.so

以後的調用就直接調用庫函數了：

Code:

```
call func@PLT
...
...
```

PLT:

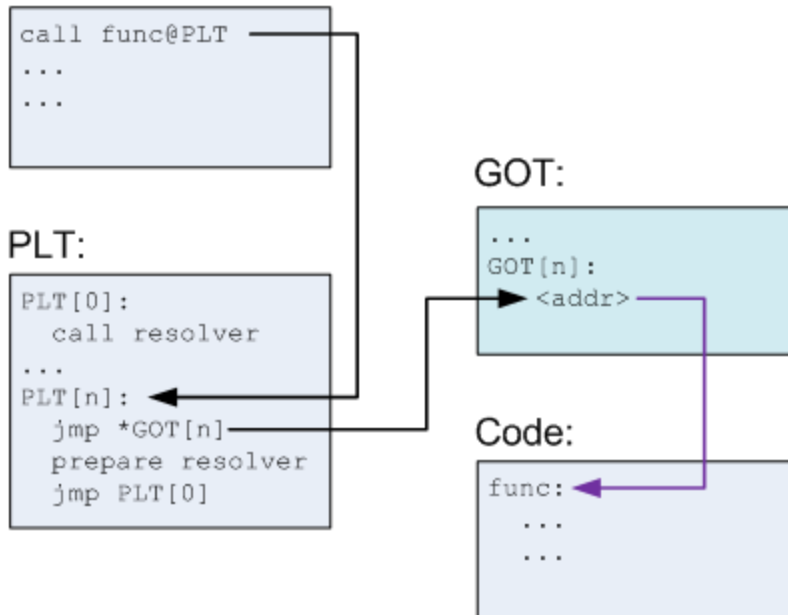
```
PLT[0]:
  call resolver
...
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

GOT:

```
...
GOT[n]:
  <addr>
```

Code:

```
func:
  ...
  ...
```



尊重原創，轉載請註明出處 anzhsoft：

<http://blog.csdn.net/anzhsoft/article/details/18776111>

參考資料：

1. <http://www.linuxidc.com/Linux/2011-06/37268.htm>

2. <http://blog.chinaunix.net/uid-24774106-id-3349549.html>

3. <http://www.linuxidc.com/Linux/2011-06/37268.htm>

4.

<http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/>

