**Linux下汇编调试器GDB的使用**

2011-12-20 16:30:13

标签：[GDB](http://blog.51cto.com/tag-GDB.html) [linux](http://blog.51cto.com/tag-linux.html) [linux系统](http://blog.51cto.com/tag-linux%E7%B3%BB%E7%BB%9F.html)

版权声明：原创作品，谢绝转载！否则将追究法律责任。

**Linux下汇编调试器GDB的使用**

      GDB 是GNU开源组织发布的一个强大的Linux/Unix下的程序调试工具。大家是否早已习惯了Windows下图形界面方式像VC、BCB等IDE的调试器，但如果你是在Linux平台下做软件调试，你会发现GDB这个调试工具有比VC、BCB的图形化调试器更强大的功能。  
先来看个实例：

1. reader@cg:~/source $ gdb -q
2. (gdb) set dis intel
3. (gdb) quit
4. reader@cg:~/source $ echo "set dis intel" **>** ~/.gdbinit
5. reader@cg:~/source $ cat ~/.gdbinit
6. set dis intel
7. reader@cg:~/source $

现在将GDB配置成为了使用Intel语法，让我们首先来认识Intel语法。在Intel语语法中，汇编指令一般遵循下面这种形式：  
operation <destination>, <source>  
目的操作数和源操作数可以是寄存器、内存地址或数值。操作通常是直观的助记符：mov操作会将源操作数中的值移动到目的操作数中，sub操作会减去，inc指令会增加等。例如，下面的指令将会扣ESP中的值移动到EBP中，然后从ESP中减去8（结果存储在ESP中）。  
8048375:        89 e5                 mov    ebp,esp  
8048377:        83 ec 08              sub    esp,0x8  
还有用于控制执行流程的操作。cmp操作用于对数值进行比较，并且基本上所有以j为首字母的操作都用于转移到代码的不同部分（转移到哪一部分取决于比较的结果）。下面的例子中，首先将位于EBP中的一个4字节的值减去4与数值9进行比较。下一条指令是如果小于等于则转移的简写，它参考的是前一个比较的结果。如果那个数小于或等于9，那么程序就会转移到Ox8048393处的指令执行。否则，就转向下一条无条件转移指令执行。如果那个数不小于或等于9，那么程序执行就会转移到Ox80483a6处。  
804838b:        83 7d fc 09           cmp    DWORD PTR [ebp-4],0x9  
804838f:        7e 02                 jle    8048393 <main+0x1f>  
8048391:        eb 13                 jmp    80483a6 <main+0x32>  
这些例子来自于我们先前的反汇编，并且我们已经将调试工具配置为使用Intel语法，所以让我们使用调试工具在汇编指令级别上单步调试第一个程序吧。  
GCC编译程序可以使用-g标记来包含附加的调试信息，这些调试信息会使得GDB能够访问源代码。

1. reader@cg:~/source $ gcc -g firstprog.c
2. reader@cg:~/source $ ls -l a.out
3. -rwxr-xr-x 1 matrix users 11977 Jul 4 17:29 a.out
4. reader@cg:~/source $ gdb -q ./a.out
5. Using host libthread\_db library "/lib/libthread\_db.so.1".
6. (gdb) list
7. 1       #include **<stdio.h>**
8. 2
9. 3       int main()
10. 4       {
11. 5               int i;
12. 6               for(i=0; i **<** **10**; i++)
13. 7               {
14. 8                       printf("Hello, world!\n");
15. 9               }
16. 10      }
17. (gdb) disassemble main
18. Dump of assembler code for function main():
19. 0x08048384 **<main**+0**>**:    push   ebp
20. 0x08048385 **<main**+1**>**:    mov    ebp,esp
21. 0x08048387 **<main**+3**>**:    sub    esp,0x8
22. 0x0804838a **<main**+6**>**:    and    esp,0xfffffff0
23. 0x0804838d **<main**+9**>**:    mov    eax,0x0
24. 0x08048392 **<main**+14**>**:   sub    esp,eax
25. 0x08048394 **<main**+16**>**:   mov    DWORD PTR [ebp-4],0x0
26. 0x0804839b **<main**+23**>**:   cmp    DWORD PTR [ebp-4],0x9
27. 0x0804839f **<main**+27**>**:   jle    0x80483a3 **<main**+31**>**
28. 0x080483a1 **<main**+29**>**:   jmp    0x80483b6 **<main**+50**>**
29. 0x080483a3 **<main**+31**>**:   mov    DWORD PTR [esp],0x80484d4
30. 0x080483aa **<main**+38**>**:   call   0x80482a8 **<\_init**+56**>**
31. 0x080483af **<main**+43**>**:   lea    eax,[ebp-4]
32. 0x080483b2 **<main**+46**>**:   inc    DWORD PTR [eax]
33. 0x080483b4 **<main**+48**>**:   jmp    0x804839b **<main**+23**>**
34. 0x080483b6 **<main**+50**>**:   leave
35. 0x080483b7 **<main**+51**>**:   ret
36. End of assembler dump.
37. (gdb) break main
38. Breakpoint 1 at 0x8048394: file firstprog.c, line 6.
39. (gdb) run
40. Starting program: /cg/a.out
42. Breakpoint 1, main() at firstprog.c:6
43. 6               for(i=0; i **<** **10**; i++)
44. (gdb) info register eip
45. eip            0x8048394        0x8048394
46. (gdb)

先列出了源代码，并且显示了main()函数的反汇编。然后，在main()函数开始的地方设置了一个断点，并且开始运行程序。这个断点只是告诉调试工具当程序运行到该点时暂停程序的执行。由于断点设置在了main()函数开始的地方，实际上在执行main()中的任何指令之前，程序会到达该断点并且暂停。然后，显示了EIP（指令指针）的值。  
EIP包含一个内存地址，该地址指向main()函数的反汇编指令中的一条指令（以粗体显示）。我们把在这之前的指令（以斜体显示）一起称为函数序言，它们由编译程序生成，用于为main()函数的局部变量设置内存空间。在C语言中需要声明变量的部分原因就是辅助建立这部分代码。调试工具知道这部分代码是自动产生的，并且聪明地将其跳过。我们随后会详细讨论函数序言，但现在我们可以从GDB中获得它的一些信息并暂时将其跳过。  
GDB调试工具提供了一个直接检查内存的方法，即使用命令x，它是检查(examine)的简写。对于任何Hacker来说，检查内存都是一项很关键的技术。大多数Hacker的漏洞发掘很像魔术——它们似乎令人惊讶并且不可思议，除非您知道这些戏法和误导。但是使用像GDB这样的调试工具，程序执行的每个方面都可以被确定地检查、暂停、单步跟踪并且可以随心所欲地重复。因为一个正在运行的程序的主体是处理器和若干内存段，所以检查内存是查看到底正在干什么的首要方法。