Introduction to GDB & Debugging

相关工具更加完整的使用方法请自行查询官方文档，这里只是介绍在本次拆弹实验中可能使用到的一些基本方法。

Objdump

objdump 是一个命令行程序（区别于GUI程序），用于在类 Unix 操作系统上显示有关**目标文件**的各种信息。例如，它可以用作**反汇编程序以查看汇编形式的可执行文件**。我们可以通过Objdump获得，可执行目标文件（二进制文件）的汇编代码文件。

Objdump的典型使用方法：

```bash
linux> objdump -d executable_file > filename
```

可以得到可执行文件的汇编文件。

生成反汇编

所以在拆弹实验中我们可以使用工具：Objdump来生成bomb（可执行目标文件）的汇编代码。

```SHELL
linux> objdump -d bomb > bomb.s
```

此时将会在对应目录下生成一个test.s文件

```SHELL
linux> ls
... bomb.s ....
```

这个文件内容包含了完整的所有汇编代码。同理我们也可以使用objdump去生成其他的可执行文件。

汇编介绍

典型由objdump生成的汇编代码如下所示：

```
0000000000001187 <main>:
    1187: f3 0f 1e fa          endbr64
    118b: 55                   push   %rbp
    118c: 48 89 e5             mov    %rsp,%rbp
    118f: 48 83 ec 10          sub    $0×10,%rsp
    1193: c7 45 fc 02 00 00 00 movl   $0×2,-0×4(%rbp)
    119a: 48 8d 05 6f 2e 00 00 lea    0×2e6f(%rip),%rax        # 4010 <x>
    11a1: 48 89 c7             mov    %rax,%rdi
    11a4: e8 c0 ff ff ff       call   1169 <add>
    11a9: 8b 05 61 2e 00 00    mov    0×2e61(%rip),%eax        # 4010 <x>
    11af: 39 45 fc             cmp    %eax,-0×4(%rbp)
    11b2: 74 1b                je     11cf <main+0×48>
    11b4: 48 8d 05 49 0e 00 00 lea    0xe49(%rip),%rax        # 2004 <_IO_stdin_used+0×4>
    11bb: 48 89 c7             mov    %rax,%rdi
    11be: b8 00 00 00 00       mov    $0×0,%eax
    11c3: e8 a8 fe ff ff       call   1070 <printf@plt>
    11c8: b8 00 00 00 00       mov    $0×0,%eax
    11cd: eb 14                jmp    11e3 <main+0×5c>
    11cf: 48 8d 05 3e 0e 00 00 lea    0xe3e(%rip),%rax        # 2014 <_IO_stdin_used+0×14>
    11d6: 48 89 c7             mov    %rax,%rdi
    11d9: e8 82 fe ff ff       call   1060 <puts@plt>
    11de: b8 00 00 00 00       mov    $0×0,%eax
    11e3: c9                   leave
    11e4: c3                   ret
```

一个典型的汇编代码行如下：

```
11b4: 48 8d 05 49 0e 00 00  lea    0xe49(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
```

从左到右分别是：

地址：11b4

最左边的标号为objdump反汇编出来的地址，**注意，在拆弹实验中，objdump生成的地址并非最后的内存虚拟地址，而是一个相对地址。**

objdump生成的地址在有些类型的可执行文件反汇编结果中，就是最后用于CPU寻址的虚拟地址；但是在拆弹实验中，我们的bomb得到的汇编文件的地址标号并不是最后运行的虚拟地址，这是由我们的可执行文件类型决定的。具体相关知识将会在第七章做主要解释。

字节码：48 8d 05 49 0e 00 00

对应内存中的指令内容，即最后被CPU读取执行的指令字节内容。

指令：lea 0xe49(%rip),%rax

字节码代表的指令内容，包含指令类型、操作数...

注释： # 2004 <IOstdin_used+0x4>

注释通常会解释一些地址的含义，比如：

```
lea    0xe49(%rip),%rax
```

这里将一个地址保存到了rax寄存器中，注释说明这个地址

```
# 2004 <_IO_stdin_used+0x4>
```

表明这个地址在_IO_stdin_used标号所处地址 + 0x4的位置上。

类似的还有注释还有：

```
0000000000001187 <main>:
```

这个注释表示这个地址标号`0000000000001187`小节主要是函数main的小节。

```
11a9: 8b 05 61 2e 00 00     mov    0x2e61(%rip),%eax  # 4010 <x>
```

这里的注释表明，`0x2e61(%rip)`地址代表的是 全局变量x。

GDB

GNU Debugger。

GDB，GNU 项目调试器，允许您查看另一个程序在执行时“内部”发生了什么——或者另一个程序在崩溃时正在做什么。

GDB 可以帮助你在运行中捕捉错误：

- 启动程序
- 让程序在特定条件下停止
- 可以在程序停下时，检查程序到底发生了什么。

运行程序

我们使用gdb装载可执行文件：（这里不方便以bomb直接显示，采用与拆弹实验中的例子`test`）

```shell
linux> gdb test
```

之后会进入gdb的运行空间

```shell
linux> gdb test
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...
(No debugging symbols found in test)
(gdb)
```

之后我们就可以在gdb的CLI中输入参数进行debug。

程序执行

在gdb中键入`run`即可运行程序：

```shell
(gdb) run
Starting program: /home/slyang/xjtu-ics/lab2/test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0×0000555555555193 in main ()
```

若存在输入参数或者输入文件，则在run后添加即可。比如实验二拆弹实验中，需要传入输入文件：

```shell
(gdb) run < solution.txt
```

设置断点

断点，就是在程序执行时，你所希望暂停的点，你可以通过断点的方式，让整个程序在执行的到这个部分的时候暂停。

在执行用户程序的入口`main`函数之前，程序会执行一些libc的初始化函数（包含`init`与`_start`）但是我们不用做过多的了解，为了可以快速的从入口开始观察程序的执行，我们通常会在main函数入口打下断点。

```shell
(gdb) b main
```

除了使用b + 函数标号之外，我们还可以使用

```shell
(gdb)break main
```

含义都是类似的，都是在main函数地址处添加断点（breakpoint）。

或如果想在其他的函数入口上（比如add函数）添加断点，则：

```
(gdb)b add
```

设下端点后，执行run之后会停在对应的第一个断点：

```SHELL
(gdb) run
Starting program: /home/slyang/xjtu-ics/lab2/test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0×000055555555518f in main ()
```

之后希望程序继续运行，则键入continue：

```SHELL
(gdb) continue
Continuing.

Breakpoint 2, 0×000055555555519a in main ()
```

相对地址：

如上都是我们将断点设置在函数入口位置时候的方法，但是如果我想在一些普通指令的位置添加断点呢?

```
(gdb)b *(指令地址)
```

之前我们提到了，objdump生成的汇编代码的指令地址并不准确，但是指令之间的相对位置确实固定的不变的，所以objdump中的指令他可以作为一个相对地址来做使用。所以这里简单介绍一下如何得到绝对指令虚拟地址：

```
0000000000001187 <main>:
    1187: f3 0f 1e fa          endbr64
    118b: 55                   push   %rbp
    118c: 48 89 e5             mov    %rsp,%rbp
    118f: 48 83 ec 10          sub    $0×10,%rsp
    1193: c7 45 fc 02 00 00 00  movl   $0×2,-0×4(%rbp)
    119a: 48 8d 05 6f 2e 00 00  lea    0×2e6f(%rip),%rax        # 4010 <x>
    11a1: 48 89 c7             mov    %rax,%rdi
    11a4: e8 c0 ff ff ff       call   1169 <add>
    11a9: 8b 05 61 2e 00 00     mov    0×2e61(%rip),%eax        # 4010 <x>
    11af: 39 45 fc             cmp    %eax,-0×4(%rbp)
    11b2: 74 1b                je     11cf <main+0×48>
    11b4: 48 8d 05 49 0e 00 00  lea    0xe49(%rip),%rax        # 2004 <_IO_stdin_used+0×4>
    11bb: 48 89 c7             mov    %rax,%rdi
    11be: b8 00 00 00 00       mov    $0×0,%eax
    11c3: e8 a8 fe ff ff       call   1070 <printf@plt>
    11c8: b8 00 00 00 00       mov    $0×0,%eax
    11cd: eb 14                jmp    11e3 <main+0×5c>
    11cf: 48 8d 05 3e 0e 00 00  lea    0xe3e(%rip),%rax        # 2014 <_IO_stdin_used+0×14>
    11d6: 48 89 c7             mov    %rax,%rdi
    11d9: e8 82 fe ff ff       call   1060 <puts@plt>
    11de: b8 00 00 00 00       mov    $0×0,%eax
    11e3: c9                   leave
    11e4: c3                   ret
```

获取得到反汇编后，我们假设需要在这一条指令处获得断点。

```
11b4: 48 8d 05 49 0e 00 00  lea    0xe49(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
```

为了获取这条指令的位置，首先在main函数处添加断点：

```SHELL
(gdb) b main
Breakpoint 1 at 0x1185
```

然后运行程序：

```SHELL
(gdb) run
Starting program: /home/slyang/xjtu-ics/lab2/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000555555555185 in main ()
```

这是你的程序暂停在main的入口处，所以，此时我们尝试读取rip（程序计数器）的值，并阅读得对应这个位置的指令：

```SHELL
(gdb) p /x $rip
$1 = 0x55555555518f
(gdb) display /i $rip
2: x/i $rip
=> 0x55555555518f <main+8>:  sub    $0x10,%rsp
```

得到结果，此时rip的指针处于**0x55555555518f**中。对应指令。

```SHELL
0x55555555518f <main+8>:  sub    $0x10,%rsp
```

（这里的两条指令的含义分别是显示寄存器rip的值（**p /x $rip**），以及显示内存中的地址（**x/i $rip**），这里并不用深究用法，只要知道大致意思即可，后续分别针对两类指令做简要的解释）

此时我们有两种方法可以获得你所需要指令对应的地址：

- 回去看objdump生成的汇编指令文件中的两条指令的相对位置，然后加减得到指令地址
- 直接在gdb中键入disas

```
(gdb) disas
Dump of assembler code for function main:
   0x0000555555555187 <+0>:  endbr64
   0x000055555555518b <+4>:  push   %rbp
   0x000055555555518c <+5>:  mov    %rsp,%rbp
=> 0x000055555555518f <+8>:  sub    $0x10,%rsp
   0x0000555555555193 <+12>:  movl   $0x2,-0x4(%rbp)
   0x000055555555519a <+19>:  lea    0x2e6f(%rip),%rax        # 0x555555558010 <x>
   0x00005555555551a1 <+26>:  mov    %rax,%rdi
   0x00005555555551a4 <+29>:  call   0x555555555169 <add>
   0x00005555555551a9 <+34>:  mov    0x2e61(%rip),%eax        # 0x555555558010 <x>
   0x00005555555551af <+40>:  cmp    %eax,-0x4(%rbp)
   0x00005555555551b2 <+43>:  je     0x5555555551cf <main+72>
   0x00005555555551b4 <+45>:  lea    0xe49(%rip),%rax        # 0x555555556004
   0x00005555555551bb <+52>:  mov    %rax,%rdi
   0x00005555555551be <+55>:  mov    $0x0,%eax
   0x00005555555551c3 <+60>:  call   0x555555555070 <printf@plt>
   0x00005555555551c8 <+65>:  mov    $0x0,%eax
   0x00005555555551cd <+70>:  jmp    0x5555555551e3 <main+92>
   0x00005555555551cf <+72>:  lea    0xe3e(%rip),%rax        # 0x555555556014
   0x00005555555551d6 <+79>:  mov    %rax,%rdi
   0x00005555555551d9 <+82>:  call   0x555555555060 <puts@plt>
   0x00005555555551de <+87>:  mov    $0x0,%eax
```

GDB也会输出当前的汇编指令，通过**Enter**键翻页，这里的汇编地址就是最后的运行地址，但是查询比较麻烦。

获取得到一个指令的相对地址后，我们可以通过之前提过的方法去在任意指令位置设置断点：

```shell
# 将会在0x00005555555551cf <+72>:  lea    0xe3e(%rip),%rax        # 0x555555556014设下端点
(gdb)b *0x00005555555551cf
```

注意地址前的这个 * 。

观察寄存器

观察寄存器的方式主要通过：

```shell
(gdb) p /x $rax
$1 = 0x555555555187
```

等类似的方式进行。

其中这里的gdb指令主要分成三部分：

```shell
#  (print)(输出方式 x 16进制 d 10进制...)     (寄存器号)
(gdb) p     /x                          $rax
```

我们也可以通过eax/ax之类查看低32位/16位的结果。

```shell
(gdb) p /x $eax
$2 = 0x55555187
(gdb) p /x $ax
$3 = 0x5187
```

或通过`/d`的方式来进行10进制的输出

```shell
(gdb) p /d $ax
$4 = 20871
```

观察内存

除了可以观察寄存器，内存在程序运行中也是很重要的一员。gdb可以通过以下方式访问所有可以被进程访问虚拟地址：

```shell
(gdb) x /1× 0×555555558010
0×555555558010 <x>: 0×00000001
```

这里的gdb指令也可以被分成三部分：

```shell
# (examine)（输出长度：1一个字（64位是8字节） 格式：x十六进制输出）  （内存虚拟地址）
(gdb) x    /1x                                0×555555558010
```

也可以通过寄存器来进行寻址：

```shell
(gdb) x /x $rax
0×555555558010 <x>: 0×00000001
# 0×555555558010地址上保存着 <x>：x的值0x00000001
```

也可以在地址位置输入表达式，gdb会帮助你完成运算。

```shell
(gdb) x /x $rip + 0×2e61
0×555555557ff0: 0×00000000
# 0×555555557ff0 = $rip + 0×2e61
```

并查找到对应的内存做输出。

单步调试

很多时候，为了详细观察程序行为，可能你需要观察每一步的汇编指令，这时候在每一步打一个断点就太过于麻烦了。gdb为我们提供了单步调试的方法，具体分为以下几种：

nexti

- run next line of program and does NOT step into functions

nexti：next命令（可简写为n）用于在程序断住后，继续执行下一条语句，假设已经启动调试，并在第12行停住，如果要继续执行，则使用n执行下一条语句，如果后面跟上数字num，则表示执行该命令num次，就达到继续执行n行的效果了.

nexti

```shell
# 执行到下一条
(gdb)nexti

# 执行到下面5条指令
(gdb)nexti 5

# ni 是nexti的缩写，作用相同，执行下一条指令
(gdb)ni
```

注意nexti会跳过函数调用，他的含义其实是运行到当前函数的下移指令，所以比如你的函数在某一步执行了：

```shell
0×00005555555551a4 <+29>:  call   0×555555555169 <add>
```

nexti不会进入add执行，而是直接跳过函数具体内容。nexti只关心我们的当前过程内的指令内容，并不关心其他过程。

那如果我们想进入其他（过程）内部去观察运行呢?

stepi

- run next line of program AND step into functions

对于上面的情况，如果我们想跟踪add函数内部的情况，可以使用step命令（可简写为s），它可以单步跟踪到函数内部，但前提是该函数有调试信息并且有源码信息。

```shell
# 运行停止在main调用add函数的地址
# 0x00005555555551a4 <+29>:  call   0×555555555169 <add>
Breakpoint 1, 0×00005555555551a4 in main ()
# si之后进入add过程内部执行
(gdb) si
0×0000555555555169 in add ()
# 继续si都处在add过程内部
(gdb) si
0×000055555555516d in add ()
```

其他工具

拆弹实验，主要集中于二进制层面的代码执行与debug，所以以上主要简单介绍二进制的Debug工具（Ogjdump & GDB）。