lab1实验手册

网络与系统安全综合实验

实验截止: 2020.09.01 星期二 23:59

欢迎来到网络与系统安全综合实验，在本课程中，我们将使用工业界主流的逆向工具IDA对Linux下的二进制程序（ELF格式）进行逆向分析。同时根据逆向的结果找出程序中的漏洞，并且利用漏洞让程序到达非预期的运行结果。在本门课中，所有程序都基于x86架构在Linux下运行。我们强烈推荐使用虚拟机完成实验内容。

本指南为曲海鹏老师的《网络与系统安全综合实验》课程实验手册，有很多助教参与了实验的设计，编写等工作：

* 房建（2020）
* 吕文杰（2020）
* 张政（2020）
* 宋晓琪（2020）
* 李晓慧（2020）

# 实验介绍

## 实验概述

本次课程一共有三个lab：

* lab1：逆向实验
* lab2：shellcode&栈溢出实验
* lab3：格式化字符串&ROP实验

每周布置一个lab，每次都会有4-5个题目需要完成，每个lab实验时间为一周。实验以个人为单位。每次lab都会在截至日期后的下一节课检查。

## 提交内容

每次lab实验需要提交pdf格式实验报告，实验报告需要完整反映题目解答过程和最后的答案。如果编写了对应的解题脚本，连同脚本和实验报告一同压缩为.zip格式上交。

# IDA使用方法

IDA是一个在Windows，Linux，MacOS上的交互式，可编程，可拓展，多处理器的反汇编程序，用于将机器码转换成可读的汇编语言。本实验将使用IDA进行逆向分析，你可以从 <https://www.hex-rays.com/products/ida/support/download_freeware/> 获取免费版本的IDA。

软件安装成功后，桌面上可以看到IDA的图标，如图1（a）所示，双击点击IDA图标（也可直接将文件拖动到图标上）。

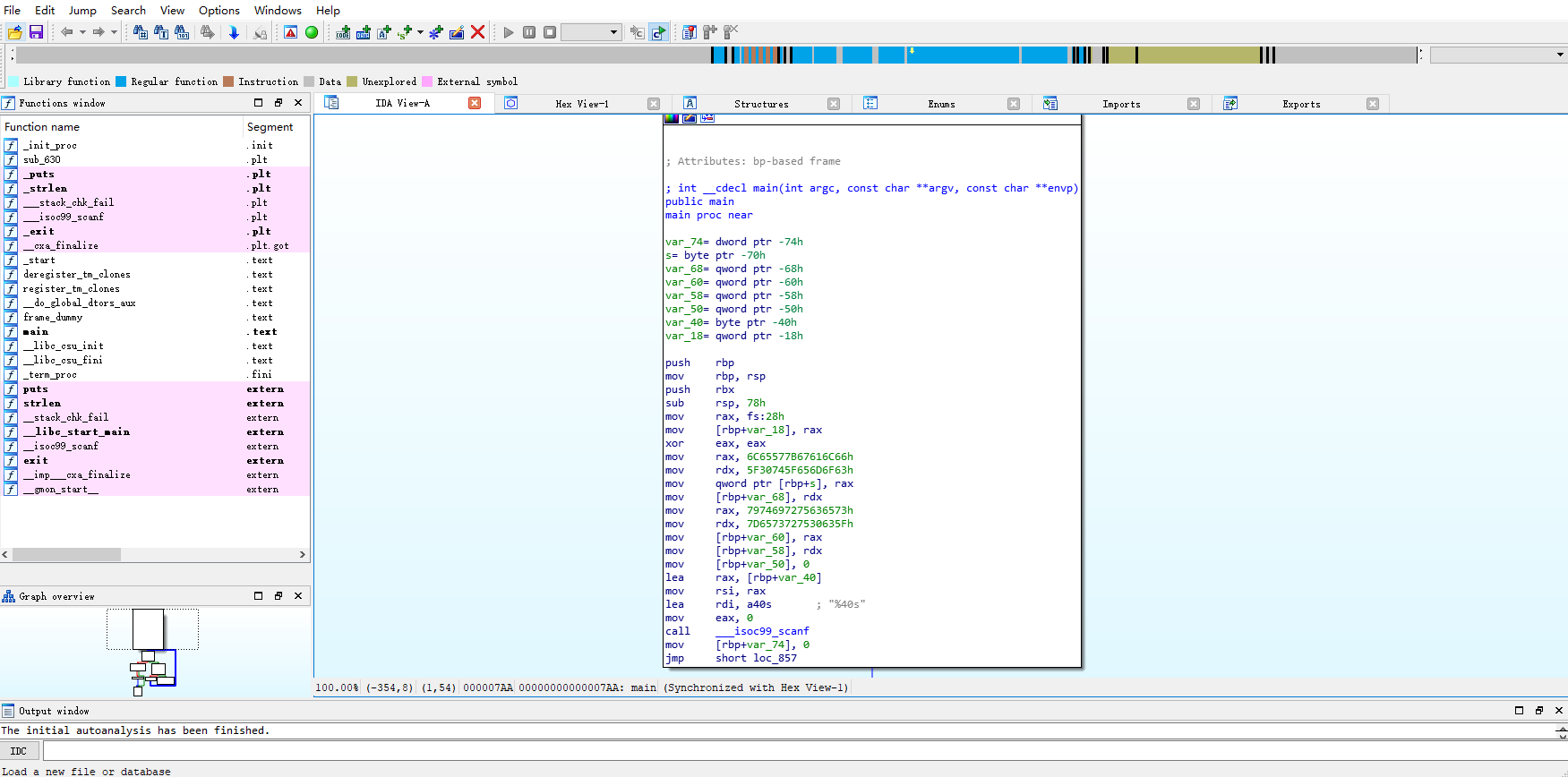
点击new，选择文件如图1（b）所示。

点击左下角选择All Files，如图1（c）所示。之后便可以看见所要逆向的文件

之后选择要逆向的文件点击打开。

选择文件的指令集如图1（d）所示，一般情况下IDA已默认选择好对应的架构，选择好后点击OK后即可进入IDA的主界面，如图2所示。

在IDA中，右侧为IDA反汇编二进制文件产生的汇编代码。左侧为IDA扫描文件后识别出的函数，点击不同的函数名，右侧可显示出不同函数对应的汇编代码。左下角为当前函数对应的控制流图（CFG）。关于IDA其他更详细的操作，可查询IDA文档（<https://www.hex-rays.com/products/ida/support/>）



IDA界面

# gdb使用方法

GNU调试器（英语：GNU Debugger，缩写：GDB），是GNU软件系统中的标准调试器，此外GDB也是个具有移携性的调试器，经过移携需求的调修与重新编译，如今许多的类UNIX操作系统上都可以使用GDB。

## gdb常用命令

在本门课中，我们将使用gdb（建议使用7.12.1及其以上版本）来动态调试二进制程序。一些gdb的插件会让这个过程更加方便，如peda（<https://github.com/longld/peda>）等插件。常用的gdb命令如表1所示。其他gdb命令可查询gdb文档（<https://sourceware.org/gdb/onlinedocs/gdb/>）

gdb常用命令

|  |  |  |
| --- | --- | --- |
| 命令 | 参数 | 含义 |
| run |  | 运行当前程序 |
| break | 地址 | 在参数所表示的地址处添加断点 |
| continue |  | 从当前位置继续执行，直到遇到断点停止 |
| quit |  | 退出gdb |
| attach | 进程的pid | 使用gdb调试当前运行的程序 |
| n |  | set over，遇到函数不进入函数内部 |
| s |  | set into，遇到函数进入函数内部 |
| finish |  | step out,执行完并且退出当前函数 |
| info | registers | 显示所有寄存器 |
| info | b | 当前设置的断点 |
| del | 断点的序号 | 删除某个断点 |
| x/nfu | 地址 | 以f格式打印从地址处开始的n个长度单元为u的内存值。 |
|  |  | f：是输出格式。x:16进制，o:8进制。u：标明一个单元的长度。 |
|  |  | b:一个byte，h:两个byte（halfword） |
|  |  | w:四个byte（word），g:八个byte（giant word） |
| vmmap(需要peda插件) |  | 打印调试程序的虚拟地址映射 |

## gdb原理技术

gdb主要功能的实现依赖于一个系统函数ptrace，ptrace主要用于执行断点的设置和跟踪子进程，查看内存，单步执行等。Ptrace调试需要两个进程，父进程可通过调用 fork方法创建子进程 ，要实现对子进程的跟踪，指定要调试程序的 PTRACE\_TRACEME 行为，最后初始化对一个进程进行跟踪操作。

父进程在跟踪子进程时，子进程在每次接收到信号后都会停止，等待父进程的响应。同时父进程会调用wait方法接受到消息。 函数原型：

#include <sys/ptrace.h> long ptrace(enum \_\_ptrace\_request request, pid\_t pid, void \*addr, void \*data)

request：被调试程序执行的行为

pid：被调试程序的标识。

addr：被调试程序执行操作的目标地址。

data：存放待写入的数据或者要读的地址

request的参数不同，ptrace会执行不同的操作。如ptrace (PTRACE\_TRACEME, 0 ,0 ,0) 执行父进程跟踪子进程，ptrace(PTRACE\_CONT, pid, 0, signal)继续执行程序直到遇到断点等。通过ptrace系统调用，gdb可以实现不同的调试功能。

## 实例

现在通过演示一个实验的例子来展示调试二进制程序的大体流程。

$ gdb ./lab1  
GNU gdb (GDB) 7.12.1  
Copyright (C) 2017 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-pc-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://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"...  
  
warning: ~/Pwngdb/pwngdb.py: No such file or directory  
Reading symbols from ./lab1...(no debugging symbols found)...done.  
gdb-peda$

在打开二进制程序之后，在main函数处下断点，之后运行文件

gdb-peda$ b \* main  
Breakpoint 1 at 0x400657  
gdb-peda$ r  
Starting program: /mnt/hgfs/learn/security\_course/week1/lab1  
[----------------------------------registers-----------------------------------]  
RAX: 0x400657 (<main>: push rbp)  
RBX: 0x0  
RCX: 0x400750 (<\_\_libc\_csu\_init>: push r15)  
RDX: 0x7fffffffe378 --> 0x7fffffffe62d ("LC\_TERMINAL\_VERSION=3.3.6")  
RSI: 0x7fffffffe368 --> 0x7fffffffe602 ("/mnt/hgfs/learn/security\_course/week1/lab1")  
RDI: 0x1  
RBP: 0x400750 (<\_\_libc\_csu\_init>: push r15)  
RSP: 0x7fffffffe288 --> 0x7ffff7a05b97 (<\_\_libc\_start\_main+231>: mov edi,eax)  
RIP: 0x400657 (<main>: push rbp)  
R8 : 0x7ffff7dd0d80 --> 0x0  
R9 : 0x7ffff7dd0d80 --> 0x0  
R10: 0x0  
R11: 0x0  
R12: 0x400570 (<\_start>: xor ebp,ebp)  
R13: 0x7fffffffe360 --> 0x1  
R14: 0x0  
R15: 0x0  
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)  
[-------------------------------------code-------------------------------------]  
 0x400651 <frame\_dummy+1>: mov rbp,rsp  
 0x400654 <frame\_dummy+4>: pop rbp  
 0x400655 <frame\_dummy+5>: jmp 0x4005e0 <register\_tm\_clones>  
=> 0x400657 <main>: push rbp  
 0x400658 <main+1>: mov rbp,rsp  
 0x40065b <main+4>: push rbx  
 0x40065c <main+5>: sub rsp,0x78  
 0x400660 <main+9>: mov rax,QWORD PTR fs:0x28  
[------------------------------------stack-------------------------------------]  
0000| 0x7fffffffe288 --> 0x7ffff7a05b97 (<\_\_libc\_start\_main+231>: mov edi,eax)  
0008| 0x7fffffffe290 --> 0x1  
0016| 0x7fffffffe298 --> 0x7fffffffe368 --> 0x7fffffffe602 ("/mnt/hgfs/learn/security\_course/week1/lab1")  
0024| 0x7fffffffe2a0 --> 0x100008000  
0032| 0x7fffffffe2a8 --> 0x400657 (<main>: push rbp)  
0040| 0x7fffffffe2b0 --> 0x0  
0048| 0x7fffffffe2b8 --> 0x50d55bbaf0a02160  
0056| 0x7fffffffe2c0 --> 0x400570 (<\_start>: xor ebp,ebp)  
[------------------------------------------------------------------------------]  
Legend: code, data, rodata, value  
  
Breakpoint 1, 0x0000000000400657 in main ()  
gdb-peda$

使用n命令运行每一条汇编指令，可以看见每一条指令运行后寄存器和栈的变化。可以通过内存打印指令打印出不同内存的内容。这里使用x/30xg打印出内存中对应地址中的值。

gdb-peda$ x/30xg 0x7fffffffe210  
0x7fffffffe210: 0x6c65577b67616c66 0x5f30745f656d6f63  
0x7fffffffe220: 0x7974697275636573 0x7d6573727530635f  
0x7fffffffe230: 0x0000000000000000 0x0000000000f0b5ff  
0x7fffffffe240: 0x0000000000666473 0x000000000040079d  
0x7fffffffe250: 0x00007ffff7de59a0 0x0000000000000000  
0x7fffffffe260: 0x0000000000400750 0x488a3f7dd9bd1f00  
0x7fffffffe270: 0x00007fffffffe360 0x0000000000000000  
0x7fffffffe280: 0x0000000000400750 0x00007ffff7a05b97  
0x7fffffffe290: 0x0000000000000001 0x00007fffffffe368  
0x7fffffffe2a0: 0x0000000100008000 0x0000000000400657  
0x7fffffffe2b0: 0x0000000000000000 0x50d55bbaf0a02160  
0x7fffffffe2c0: 0x0000000000400570 0x00007fffffffe360  
0x7fffffffe2d0: 0x0000000000000000 0x0000000000000000  
0x7fffffffe2e0: 0xaf2aa4c53b202160 0xaf2ab47a489e2160  
0x7fffffffe2f0: 0x00007fff00000000 0x0000000000000000

使用continue命令，由于后续没有断点，程序将会一直运行直到结束。

gdb-peda$ c  
Continuing.  
error flag  
[Inferior 1 (process 14182) exited normally]  
Warning: not running   
gdb-peda$

## 远程调试与交叉编译

本地调试是指在Linux系统中写代码，在Linux中编译，然后在Linux中调试运行。而远程调试则是在Windows或其他系统编写代码，然后将代码上传Linux服务器编译链接和运行。调试运行在Linux，而调试的动作发生在Windows或其他系统中。Linux系统里的调试需要一个调试工具来将调试信息传递出来，也就是gdb-gdbserver，而客户端将调试动作和结果显示在本机，就好像是在本机调试一样。

在一种计算机环境中运行的编译程序，能编译出在另外一种环境下运行的代码，我们就称这种编译器支持交叉编译。这个编译过程就叫交叉编译。

当我们使用gdb的远程调试模式时，在目标板上通过 gdbserver 运行待调试的程序，在宿主机上运行 gdb 并通过 ’target remote [ip]:[port]’ 来连接到目标板上的 gdbserver，从而启动远程调试。各种调试命令在宿主机上输入，程序执行效果(包括打印)在目标板上展示

## 符号表

在整个编译过程中，编译器把c语言的源文件（.c）编译成汇编语言文件（.s），汇编器把汇编文件翻译成目标文件（.o），最后链接器将所有目标文件和有关的库连接成一个可执行文件（.out）。

调试信息存储的是程序在运行时方方面面的信息，如我们设置的断点信息，调试信息就会判断当进程运行到断点处暂停。我们可以通过gcc -g命令来获得调试信息，在编译的时候，变量名、函数名、函数参数、函数地址等信息作为调试信息会被加载到调试信息里面，经过汇编和链接，最终得到可执行文件。调试信息通常用“.stab”表示，这种调试信息格式叫“stab”，及符号表（symbol table）。stabs调试信息在汇编代码级主要表现为四种伪指令格式.stabs、.stabn、.stabd和.stabx。.stabs格式主要包括源文件的文件名、函数变量等信息，还包括所使用的编程语言等，.stabn则反映程序有关结构的信息，如程序块的结构信息等等。如.stabs格式.stabs ”string”，type，other，desc，value，type字段的值是一个整数值，这个值表示该指令是否为一个stab信息。

## peda

PEDA是为GDB设计的一个强大的插件，全称是Python Exploit Development Assistance for GDB。它提供了很多人性化的功能，比如高亮显示反汇编代码、寄存器、内存信息，提高了debug的效率。同时，PEDA还为GDB添加了一些实用新的命令，如表2所示。

peda常用命令

|  |  |
| --- | --- |
| 命令 | 含义 |
| aslr | 显示/设置GDB的ASLR设置 |
| checksec | 检查二进制文件的各种安全选项 |
| dumpargs | 在呼叫指令处停止时显示传递给函数的参数 |
| dumprop | 将特定内存范围内的所有ROP小工具转储 |
| elfheader | 从被调试的ELF文件中获取标题信息 |
| elfsymbol | 从ELF文件获取非调试符号信息 |
| readelf | 获取elf头信息 |
| shellcode | 生成shellcode |
| vmmap | 可以用来查看栈、bss段是否可以执行 |

## gdbinit

当我们在使用gdb进行调试时，会有一些重复的指令需要进行，每次都重复输入这些指令比较繁琐，我们可以将这些操作写成一个脚本，这个脚本就是我们要说的gdbinit。

gdb在启动的时候，会在当前目录下查找".gdbinit"这个文件，并把它的内容作为gdb命令进行解释，所以如果我们把脚本命名为".gdbinit"，这样在启动的时候就会处理这些命令。 如我们要写一个脚本来设置一些断点：

#filename: .gdbinit  
#gdb will read it when starting  
b main  
b func1  
r

# VIM使用

Vim是从vi发展出来的一个文本编辑器，具有代码补完、编译、错误跳转等方面编程的功能。新用户可以通过阅读帮助文档熟悉vim：终端输入vim，回车，然后 :help进入文档。

Vim具有多种模式，基本上可分为：普通模式/插入模式/命令行模式/可视模式

普通模式：Vim启动后的默认模式。该状态下键盘输入会被识别为命令，普通模式命令往往需要一个操作符结尾。

普通模式下常见命令

|  |  |
| --- | --- |
| 命令 | 含义 |
| h,j,k,l | 分别用于光标左移、下移、上移、右移一个字符 |
| Ctrl+b | 向前翻页 |
| Ctrl+f | 向后翻页 |
| H | 将光标移到当前屏幕首行的行首（即左上角） |
| nH | 将光标移到当前屏幕第n行的行首 |
| O | 移动至光标所在行的行首 |
| $ | 移动至光标所在行的行尾 |
| rc | 用字符c替换光标所指向的第一个字符（非空格） |
| nrc | 用字符c替换光标所指向的前n个字符（非空格） |
| x | 删除光标处的字符 |
| nx | 删除光标位置开始向右的n个字符 |
| dw | 删除一个word |
| dnw | 删除n个word |
| dfa | 删除当前光标处到下一个字符a处（fa定位光标到a位置） |
| dd | 删除光标所在行 |
| ndd | 删除光标所在行及其后的n-1行 |
| yy | 复制当前行到缓冲区 |
| nyy | 复制当前行开始的n行到缓冲区 |
| yw | 复制一个word，还有ynw |
| yfa | 复制光标处到下一个 a 的字符处,还有ynfa |
| p | 将缓冲区内容写入到光标处 |
| U | 撤销操作 |
| CTRL+r | 恢复撤销操作 |
| ZZ | 保存退出 |
| ZQ | 不保存退出 |

由普通模式进入插入模式可以按“a”(append)或“i”(insert)键。 插入模式：该模式下，键盘按键用于向缓冲区插入文本。 由插入模式返回普通模式可以按ESC键。 命令行模式：在命令行模式中可以输入会被解释成并执行的文本。 如执行命令（":"键），搜索（"/"和"?"键）或者过滤命令（"!"键）。在命令执行之后，Vim返回到命令行模式之前的模式，通常是普通模式。

命令行模式下常见命令

|  |  |
| --- | --- |
| 命令 | 含义 |
| :q | quit |
| :wq | 保存退出 |
| :q! | 不保存退出 |
| :w | 保存当前更改 |
| :w! | 强制保存 |
| :w file | 将当前内容写入file文件 |
| :n1 n2wfile | 将自n1至n2行的内容写入file文件 |
| :r file | 打开另一个文件file |
| :e file | 新建file文件 |
| :f file | 把当前文件命名为file |
| :set nu | 显示行号 |
| :set nonu | 隐藏行号 |
| :n | 定位到第n行 |
| :n1,n2d | 删除多行文本 |
| /string | 关键字查找 |
| :s/aaaa/bbbb/g | 把当前行的aaaa替换为bbbb |
| :%s/aaaa/bbbb/g | 把全文的aaaa替换为bbbb |
| :n1,n2s/aaaa/bbbb/g | 把n1到n2行的aaaa替换为bbbb |

可视模式 这个模式与普通模式比较相似。但是移动命令会扩大高亮的文本区域。高亮区域可以是字符、行或者是一块文本。当执行一个非移动命令(d删除/y复制/p粘贴……)时，命令会被执行到选中区域上。 v字符可视化： 按下键盘上的v以后，屏幕底部有VISUAl 的提示，操作 h,j,k,l就选中文本，继续按 v 退出可视化模式。 V 行可视化： 按下键盘上的 V 以后，屏幕底部有VISUAL LINE 的提示，操作 j,k 可以向上或者向下以行为单位选中文本，继续按下 V 退出可视化模式。 Ctrl+v 块状可视化： 按下键盘上的 Ctrl+v 以后，屏幕底部有提示 VISUALBLOCK ，可以通过 h,j,k,l 块状的操作选择区域，继续按下 Ctrl+v 会退出可视化模式。 vim打开文件举例：

vim file0 :打开文件file0  
Vim file1 file2 :打开文件file1 file2  
:ls 列出当前打开的所有文件  
:bn 显示第n个文件  
当然也可以分屏显示多个文件:  
vim -On file1 file2 ... filen  
这里的 n（n 是要打开的具体文件的数目：1,2,3 ...）是代表有几个文件需要分屏，从左至右依次显示 n 个文件。大写O表示左右分屏；小写o表示上下分屏。  
Ctrl + w c :关闭当前的分屏  
Ctrl + w q :关闭当前的分屏，如果是最后一个分屏将会退出 VIM 。

# 实验环境搭建

本门课程配备基础Linux虚拟机（需要按照以下方式进行环境搭建） <https://pan.baidu.com/s/1eu5PFNjvWlhUFy_a41BknQ> 提取码： ndnj

下载后解压.zip文件后，在虚拟机软件中点击左上方文件–>打开–>选择解压后的文件即可。 在打开虚拟机后，会提示“我已复制/移动该虚拟机”的选项狂，选择“我已移动该虚拟机”即可。

本实验中会有32位和64位的程序，如果你使用的虚拟机是64位需要额外配置32位运行环境

sudo apt-get update  
sudo apt-get install libc6:i386

pwntools安装（python3版本）

sudo apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential  
sudo pip3 install pwntools

peda插件安装

git clone https://github.com/longld/peda.git ~/peda  
echo "source ~/peda/peda.py" >> ~/.gdbinit

# 实验内容

lab1包含4个题目，需要对4个程序进行逆向。逆向结果为一个flag开头的可见字符串，正则形式为：flag{[0-9a-zA-Z]+}。

其中lab1-1是简单的逆向入门用于熟悉软件和环境，lab1-2和lab1-3涉及到数学计算，需要通过基本的数学运算性质进行逆计算得出flag，lab1-4需要对原有的二进制进行一些修改，之后才能进行正常的输入输出。

每个程序在输入正确的flag后会显示good，如果flag不正确将会输出error flag。

$ ./lab1-1  
123  
error flag  
$ ./lab1-1  
flag{XXXXXXXXXXXXX}  
good!  
$