

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

概要

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)
- 肖辉 (2020)

1 实验介绍

1.1 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。

1.2 提交内容

每次 lab 实验需要提交实验报告 (PDF/Word), 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。

命名格式:

压缩包: lab[1-3]-学号-姓名.zip

报告: lab[1-3]-学号-姓名.[txt/doc/pdf]

脚本: lab[1-3]-[1-5]-学号-姓名.[c/py/...]

e.g. lab1-17000000001-张三.zip

e.g. lab1-17000000001-张三.pdf

e.g. lab1-2-17000000001-张三.py

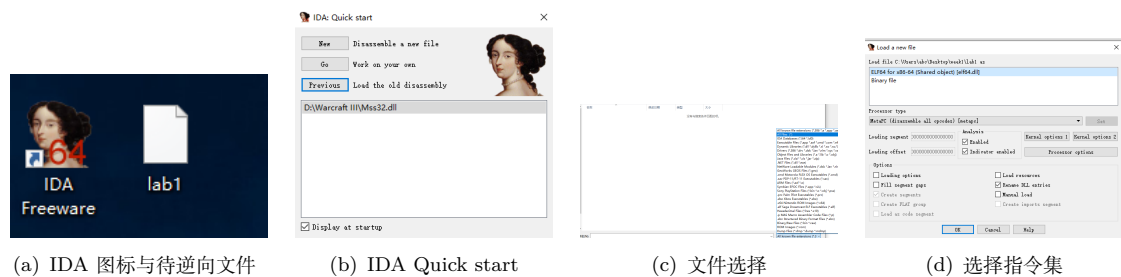


图 1: IDA 打开文件过程

2 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/>)

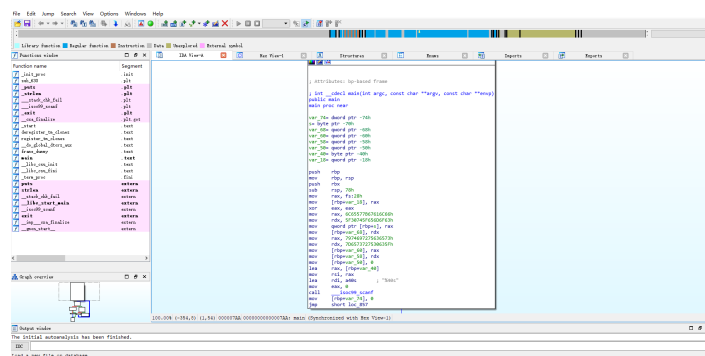


图 2: IDA 界面

3 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 也是个具有移植性的调试器, 经过移植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。

3.1 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/>)

表 1: 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

3.2 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。

3.3 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。

代码 1: 使用 gdb 打开要调试的文件

```
1 $ gdb ./lab1
2 GNU gdb (GDB) 7.12.1
3 Copyright (C) 2017 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-pc-linux-gnu".
9 Type "show configuration" for configuration details.
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>.
12 Find the GDB manual and other documentation resources online at:
13 <http://www.gnu.org/software/gdb/documentation/>.
14 For help, type "help".
15 Type "apropos word" to search for commands related to "word"...
16
17 warning: ~/Pwngdb/pwngdb.py: No such file or directory
18 Reading symbols from ./lab1...(no debugging symbols found)...done.
19 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

代码 2: gdb 下断点

```

1 gdb-peda$ b * main
2 Breakpoint 1 at 0x400657
3 gdb-peda$ r
4 Starting program: /mnt/hgfs/learn/security_course/week1/lab1
5 [-----registers-----]
6 RAX: 0x400657 (<main>: push rbp)
7 RBX: 0x0
8 RCX: 0x400750 (<__libc_csu_init>: push r15)
9 RDX: 0x7fffffff378 --> 0x7fffffff62d ("LC_TERMINAL_VERSION=3.3.6")
10 RSI: 0x7fffffff368 --> 0x7fffffff602 ("/mnt/hgfs/learn/security_course/week1/lab1")
11 RDI: 0x1
12 RBP: 0x400750 (<__libc_csu_init>: push r15)
13 RSP: 0x7fffffff288 --> 0x7ffff7a05b97 (<__libc_start_main+231>: mov edi,eax)
14 RIP: 0x400657 (<main>: push rbp)
15 R8 : 0x7ffff7dd0d80 --> 0x0
16 R9 : 0x7ffff7dd0d80 --> 0x0
17 R10: 0x0
18 R11: 0x0
19 R12: 0x400570 (<_start>: xor ebp,ebp)
20 R13: 0x7fffffff360 --> 0x1
21 R14: 0x0
22 R15: 0x0
23 EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
24 [-----code-----]
25 0x400651 <frame_dummy+1>: mov rbp, rsp
26 0x400654 <frame_dummy+4>: pop rbp
27 0x400655 <frame_dummy+5>: jmp 0x4005e0 <register_tm_clones>
28 => 0x400657 <main>: push rbp
29 0x400658 <main+1>: mov rbp, rsp
30 0x40065b <main+4>: push rbx
31 0x40065c <main+5>: sub rsp, 0x78
32 0x400660 <main+9>: mov rax, QWORD PTR fs:0x28
33 [-----stack-----]
34 0000| 0x7fffffff288 --> 0x7ffff7a05b97 (<__libc_start_main+231>: mov edi,eax)
35 0008| 0x7fffffff290 --> 0x1
36 0016| 0x7fffffff298 --> 0x7fffffff368 --> 0x7fffffff602 ("/mnt/hgfs/learn/
security_course/week1/lab1")

```

```

37 0024| 0x7fffffff2a0 --> 0x100008000
38 0032| 0x7fffffff2a8 --> 0x400657 (<main>: push rbp)
39 0040| 0x7fffffff2b0 --> 0x0
40 0048| 0x7fffffff2b8 --> 0x50d55bbaf0a02160
41 0056| 0x7fffffff2c0 --> 0x400570 (<_start>: xor ebp,ebp)
42 [-----]
43 Legend: code, data, rodata, value
44
45 Breakpoint 1, 0x000000000400657 in main ()
46 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。

代码 3: gdb 打印内存

```

1 gdb-peda$ x/30xg 0x7fffffff210
2 0x7fffffff210: 0x6c65577b67616c66 0x5f30745f656d6f63
3 0x7fffffff220: 0x7974697275636573 0x7d6573727530635f
4 0x7fffffff230: 0x0000000000000000 0x000000000f0b5ff
5 0x7fffffff240: 0x0000000000666473 0x00000000040079d
6 0x7fffffff250: 0x00007fff7de59a0 0x0000000000000000
7 0x7fffffff260: 0x0000000000400750 0x488a3f7dd9bd1f00
8 0x7fffffff270: 0x00007fffffff360 0x0000000000000000
9 0x7fffffff280: 0x0000000000400750 0x00007fff7a05b97
10 0x7fffffff290: 0x0000000000000001 0x00007fffffff368
11 0x7fffffff2a0: 0x0000000100008000 0x000000000400657
12 0x7fffffff2b0: 0x0000000000000000 0x50d55bbaf0a02160
13 0x7fffffff2c0: 0x0000000000400570 0x00007fffffff360
14 0x7fffffff2d0: 0x0000000000000000 0x0000000000000000
15 0x7fffffff2e0: 0xaf2aa4c53b202160 0xaf2ab47a489e2160
16 0x7fffffff2f0: 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。

代码 4: 运行结束

```

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

```

3.4 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

3.5 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。

3.6 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。

表 2: 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

3.7 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：

代码 5: gdbinit 脚本编写

```
1 #FILENAME: .GDBINIT
2 #GDB WILL READ IT WHEN STARTING
3 b main
4 b func1
5 r
```

4 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。

表 3: 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。

表 4: 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:

打开文件 (一个/多个)

```

1 vim file0 :打开文件file0
2 Vim file1 file2 :打开文件file1 file2
3 :ls 列出当前打开的所有文件
4 :bn 显示第n个文件
5 当然也可以分屏显示多个文件:
6 vim -On file1 file2 ... fileN
7 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。
8 Ctrl + w c :关闭当前的分屏
9 Ctrl + w q :关闭当前的分屏, 如果是最后一个分屏将会退出 VIM。
```

5 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”的选项框, 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

代码 6: 32 位环境搭建

```
1 sudo apt-get update
2 sudo apt-get install libc6:i386
```

pwntools 安装 (python3 版本)

代码 7: pwntools 安装

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

peda 插件安装

代码 8: peda 插件安装

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

6 汇编层面的函数调用与返回

在 C/C++ 程序中，函数调用是非常常见的操作。在这里对函数的传参、调用和返回过程进行简单地介绍，在做逆向实验时会对大家有所帮助。

在 x86/x86-64 架构中，x86 通过栈传递参数，x86-64 由于增加了寄存器，函数的前 6 个参数由寄存器传递，分别对应 rdi/rsi/rdx/rcx/r8/r9，其他参数仍通过寄存器传值；函数的返回值通过 eax/rax 保存。下面给一个示例，我们通过他的汇编代码来了解一下一个函数调用的过程。

示例代码

```
1 #include <stdio.h>
2
3 int callee(int a, int b, int c) {
4     int d = a + b + c;
5     return d;
6 }
7
8 int main(int argc, char const *argv[])
9 {
10     int a = callee(1, 2, 3);
11     printf("编译\n");
12     printf("gcc demo.c -o demo\n");
13     printf("反汇编可执行文件查看汇编代码\n");
14     printf("objdump -Mintel -d demo\n");
15     return 0;
16 }
```

在调用 callee 函数时，首先进行参数传递，三个参数分别由 rdi/rsi/rdx 保存。

```
mov     edx,0x3; ARG C = 3
mov     esi,0x2; ARG B = 2
mov     edi,0x1; ARG A = 1
call    callee ; CALL CALLEE FUNCTION
```

进入 callee 函数之后，首先要保存调用函数的栈状态。在这个例子中，在进入 callee 函数之后，首先要做的就是保存 main 函数的当前状态。在 x86 架构中，通过下面两条指令实现。

```
; 标准函数开头
;<CALLEE>:
push    rbp      ;
mov     rbp, rsp ;通过两条指令保存MAIN函数的栈帧
```

然后进入 callee 的逻辑，获取三个参数的值并完成相加计算，将计算结果保存到 eax 作为返回值。

```
mov     DWORD PTR [rbp-0x14],edi ;取第1个参数 A
mov     DWORD PTR [rbp-0x18],esi ;取第2个参数 B
mov     DWORD PTR [rbp-0x1c],edx ;取第2个参数 C
mov     edx,DWORD PTR [rbp-0x14]
mov     eax,DWORD PTR [rbp-0x18]
add     edx,eax
mov     eax,DWORD PTR [rbp-0x1c]
add     eax,edx                ;完成加法
mov     DWORD PTR [rbp-0x4],eax
mov     eax,DWORD PTR [rbp-0x4] ;保存返回值到EAX
```

完成 callee 函数之后，恢复 main 函数的栈帧，返回到 main 函数继续执行。

```
pop     rbp ; 恢复MAIN栈帧
ret     ; 返回MAIN
```

返回 main 函数后，将返回值取出来

```
mov     DWORD PTR [rbp-0x4],eax
```

到这里调用 callee 函数的整个过程就结束了。

7 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。

代码 9：验证逆向结果

```
1 $ ./lab1-1
2 123
3 error flag
4 $ ./lab1-1
5 flag{XXXXXXXXXXXXX}
6 good!
7 $
```