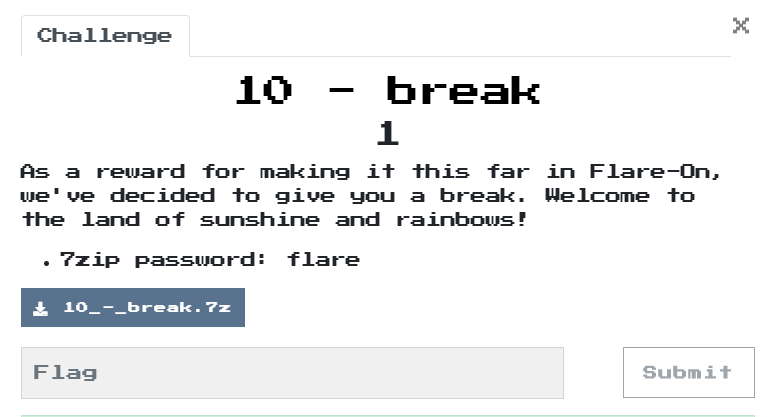
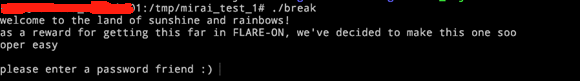
## 0x00 初出茅庐

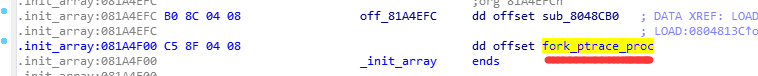


解压后发现是i386的elf文件，执行后需要输入正确密码：既此题的flag。

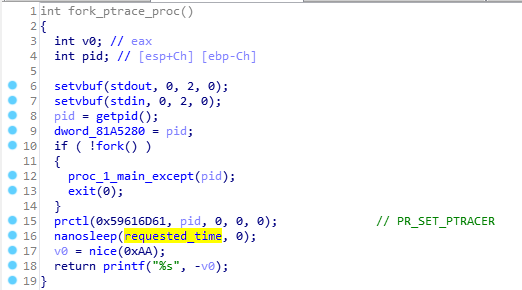


## 0x01 牛刀小试

查看程序入口\_start -> \_\_libc\_start\_main 启动过程，发现进入main之前会执行.init\_array节的如下函数。



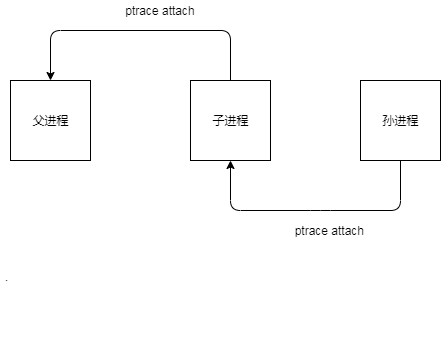
此函数会fork一个子进程，在子进程中又会fork另一个子进程。



fork出来的子进程都会通过dlopen动态加载ptrace函数，attach到父进程。



最终形成如下父子孙三进程结构



静态分析会发现有如下陷阱

### SYSCALLHOOK

分析子进程主逻辑可知主进程的syscall已经被子进程PTRACE\_SYSEMU接管。

PTRACE\_SYSEMU典型使用例子

**for**(;;) {

    ptrace(PTRACE\_SYSEMU, pid, 0, 0);

    waitpid(pid, 0, 0);

**struct** user\_regs\_struct regs;

    ptrace(PTRACE\_GETREGS, pid, 0, &regs);

**switch** (regs.orig\_rax) {

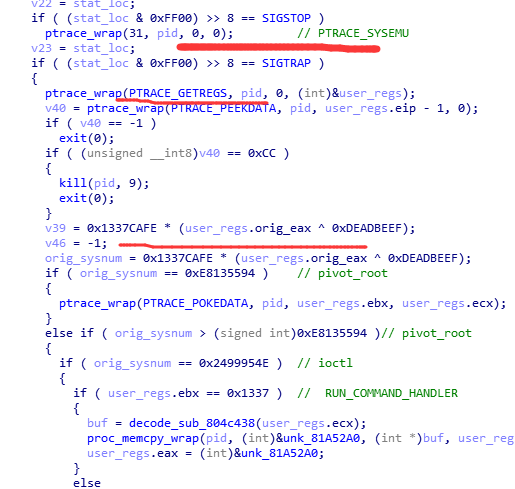
...

   }

}

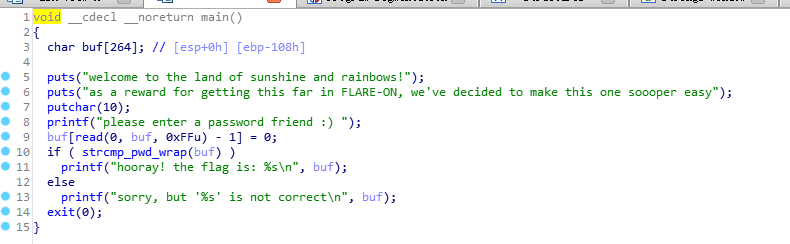
同时原始系统调用号所在的orig\_eax被子进程通过公式 0x1337CAFE \* (原始系统调用号^0xDEADBEEF)算出新标号然后在做对应的处理。反向结算得到原始系统调用号，然后查对应linux x86系统调用表即可确定真正的系统调用名称。

|  |  |  |
| --- | --- | --- |
| 0xe8135594 | 217 | pivot\_root |
| 0x2499954e | 54 | ioctl |
| 0x4a51739a | 92 | truncate |
| 0x7e85db2a | 4 | write |
| 0x3dfc1166 | 34 | nice |
| 0xf7ff4e38 | 11 | execve |
| 0x9c7a9d6 | 122 | uname |
| 0x9678e7e2 | 96 | getpriority |
| 0xb82d3c24 | 1 | exit |
| 0xc93de012 | 152 | mlockall |
| 0xab202240 | 15 | chmod |
| 0x83411ce4 | 97 | setpriority |
| 0x91bda628 | 3 | read |

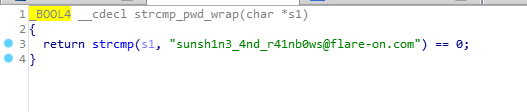


### 改写.text段制造SIGILL异常

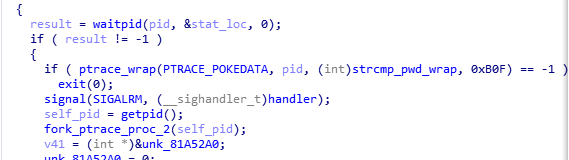
下面回到主进程main函数



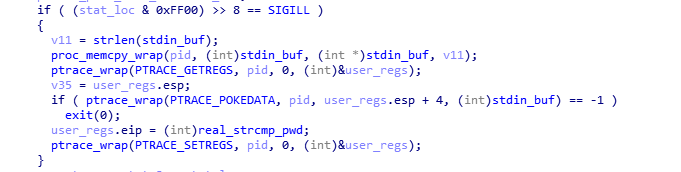
会发现其逻辑异常简单，甚至有个显而易见的密码字符串比较函数，然而输入此密码会提示密码错误。



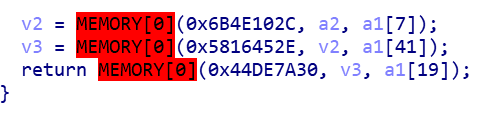
观察子进程逻辑就会发现，其通过ptrace朝父进程虚假的密码比较函数地址写入0xB0F，从而使主进程执行比较函数的时候就会触发SIGILL异常信号。



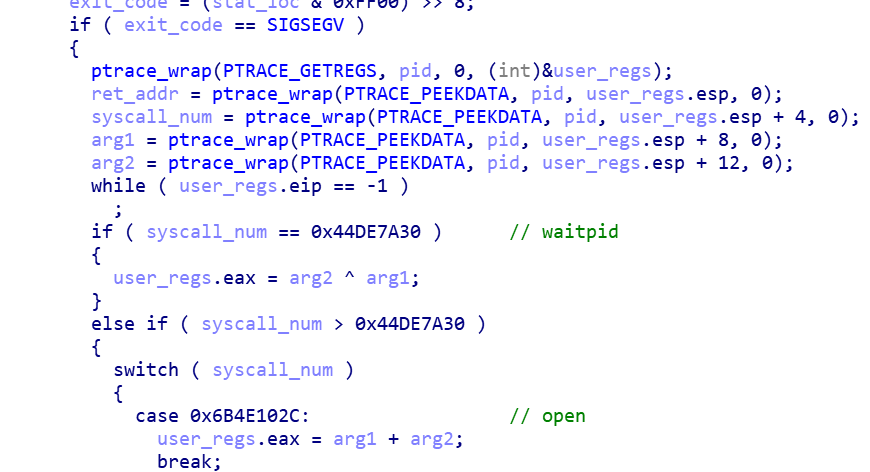
子进程handle SIGILL信号，修改父进程eip跳转到真正的密码比较执行流程



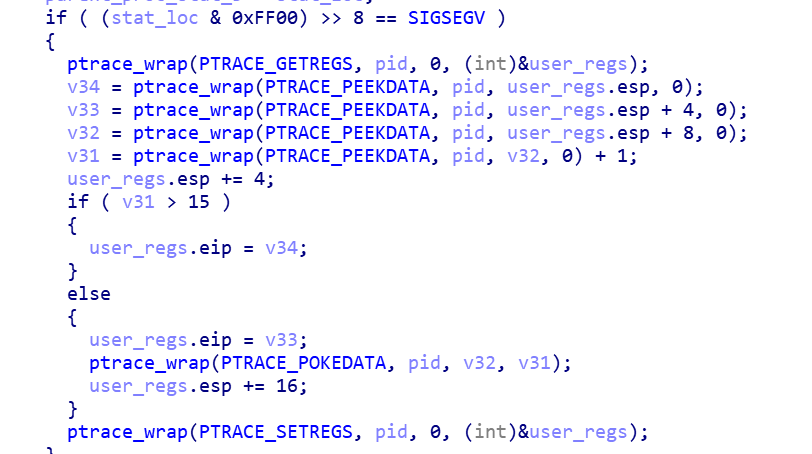
### CALL0制造SIGSEGV异常



程序通过故意call0触发段错误，当前进程的子进程通过ptracehandle后从中取出返回地址和参数，然后在做对应逻辑处理。



孙进程对子进程的段错误handle，注意这里只是用了在子进程syscallhook中相同的数值，其实和syscall并没有任何关系。



子进程对父进程的段错误handle

总结一下此题使用ptrace达到了如下目的：

* 读写当前进程的父进程寄存器
* 父子进程间的内存拷贝
* handle当前进程父进程SIGSEGV SIGILL等信号，通过人为制造异常，子进程捕获后改变原有逻辑。
* hook父进程syscall 改变原有逻辑
* 反调试

## 0x02 明修栈道

由于此题有很多的加解密操作，为了省事考虑尝试使用qiling/unicorn模拟框架来模拟执行

Qiling 简单地说就是一个基于unicorncpu 模拟，在其之上增加了二进制文件加载、地址分配、syscall模拟等功能。

项目地址：

<https://github.com/qilingframework/qiling>

主要目的如下：

1. 避开反调试：ptrace一个萝卜一个坑，调试起来麻烦。
2. 避开完整的算法还原：算法太多，浪费精力。
3. 顺便练习下qiling/unicorn框架。

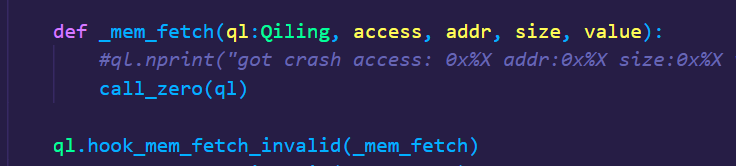
由于qiling的不完善，对于此题来说依然有很多问题，导致并不能完整模拟。

1. Syscall实现不全，比如此题大量使用的ptrace调用在qiling是直接返回0，并不真正的做操作。
2. 对call0异常会抛出unicorn的mem fetch invalid异常，无法实现此题中的异常接管处理。
3. 需要模拟父进程进入syscall hook后转入子进程处理逻辑
4. 将父子孙三进程结构归并到一个模拟进程单元处理

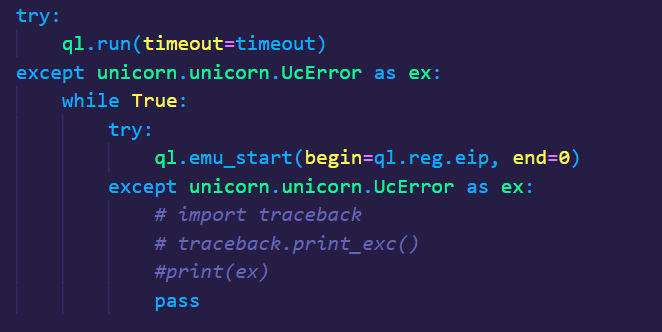
下面我们通过一系列模式来处理当前模拟存在的问题：

### 模拟call 0异常处理

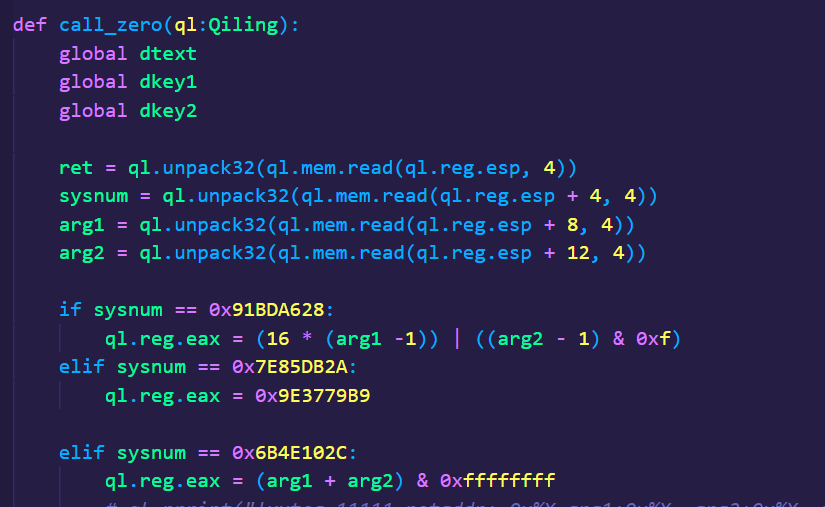
首先增加hook call 0导致的mem fetch invalid事件，在callback函数中转向我们自己的call\_zero异常处理函数。注意，这里就算增加eventhook callback，处理完以后依然会抛出异常。



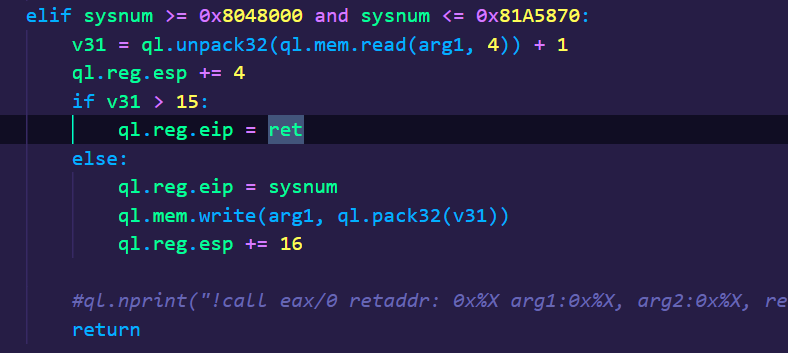
我们通过捕获UcError后继续执行模拟来解决这个问题。



Call\_zero函数中由于其真正有用的部分核心逻辑简单，这里就不跳到真正的地址模拟执行，而且直接在python里实现。这里处理掉了子孙进程的call0异常代码。



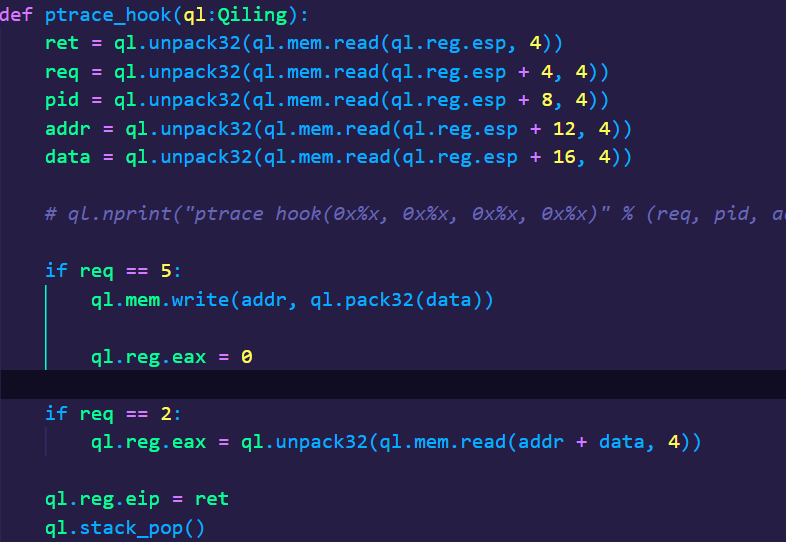
子进程对父进程 call0段错误的跳转逻辑



### 模拟ptrace调用

由于程序使用的自定义函数dlopen动态加载ptrace，这里直接hook地址

        ql.hook\_address(ptrace\_hook, 0x804BAE6)



我们将跨进程的内存拷贝全部转向本进程内存读写，这里偷懒只模拟内存读写操作。寄存器读写后面单独处理就行了。

### 模拟子进程对父进程的syscallhook

模拟子进程的堆栈，后面会用到这些偏移。进入syscallhook之前模拟ptraceGETREG保存父进程寄存器到子进程栈上

child\_proc\_regs = None

def load\_shadow\_stack(ql:Qiling):

    global child\_proc\_regs

    child\_proc\_regs = PtRegs(ql)

    ql.stack\_push(ql.reg.ebp)

    ql.reg.ebp = ql.reg.esp

    ql.stack\_push(ql.reg.edi)

    ql.stack\_push(ql.reg.esi)

    ql.stack\_push(ql.reg.ebx)

    ql.reg.esp = ql.reg.esp - 0x404c

    ql.mem.write(ql.reg.ebp - 0x28, ql.pack32(0))

    ql.mem.write(ql.reg.ebp - 0x20, ql.pack32(0xFFFFFFFF))

    child\_proc\_regs.write(ql, ql.reg.ebp - 0xcc)

退出syscall的时候恢复

def reset\_shadow\_stack(ql:Qiling):

    child\_proc\_regs.read(ql, ql.reg.ebp - 0xcc)

    ql.reg.esp = ql.reg.ebp - 12

    ql.reg.ebx = ql.stack\_pop()

    ql.reg.esi = ql.stack\_pop()

    ql.reg.edi = ql.stack\_pop()

    ql.reg.ebp = ql.stack\_pop()

    child\_proc\_regs.set\_reg(ql)

这里只需要hooksyscall的回调函数中跳转到子进程对应的case就可以模拟了子进程对父进程syscallhook了。

 ql.set\_syscall("chmod", my\_syscall\_chmod)

 ql.set\_syscall(0x98, my\_syscall\_mlockall)

 ql.set\_syscall(0xd9, my\_syscall\_pivot\_root)

 ql.set\_syscall("uname", my\_syscall\_uname)

 ql.set\_syscall("truncate", my\_syscall\_truncate)

例子如下：

def my\_syscall\_mlockall(ql, flags, \*args, \*\*kw):

    load\_shadow\_stack(ql)

    ql.reg.eip = 0x80496E5

现在我们有了这个基础的模拟框架，就可以对主密码验证逻辑做模拟了

比如nice函数，我们就可以通过传参解码不同的字符串。

*#fake\_flag@no-flare.com*

*There are many lessons to be learned by studying the Bee Movie.*

*sm0l\_bin\_b1g\_h34rt@no-flare.com*

*This challenge is so easy it exploits itself.*

*okay\_1\_sw34r\_th1s\_1s\_th3\_r34l\_0ne@no-flare.com*

*moc.eralf-on@gn1hs1n1f\_n0\_st4rgn0c*

*not\_a\_fake\_flag@no-flare.com elf\_0n\_a\_sh3lf@on-flare.com*

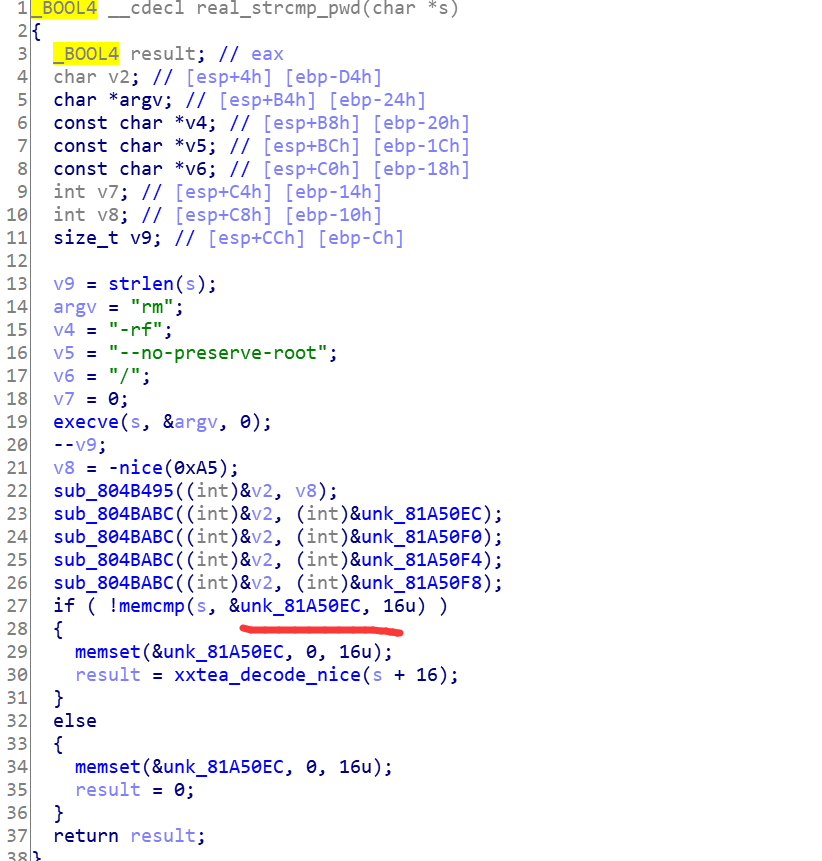
*sorry i stole your input :)*

## 0x03 暗度陈仓

### Flag第一部分

现在进入SIGILL跳转进入的真正密码验证主逻辑。我们的模拟框架加载elf完成后，进入入口点直接跳过来模拟执行，跳过前面不重要的部分。

Flag的第一部分是16个字节。注意execve和nice都会被劫持到子进程syscallhook逻辑。



hook26行之后的地址，直接打印unk\_81A50EC即可得到



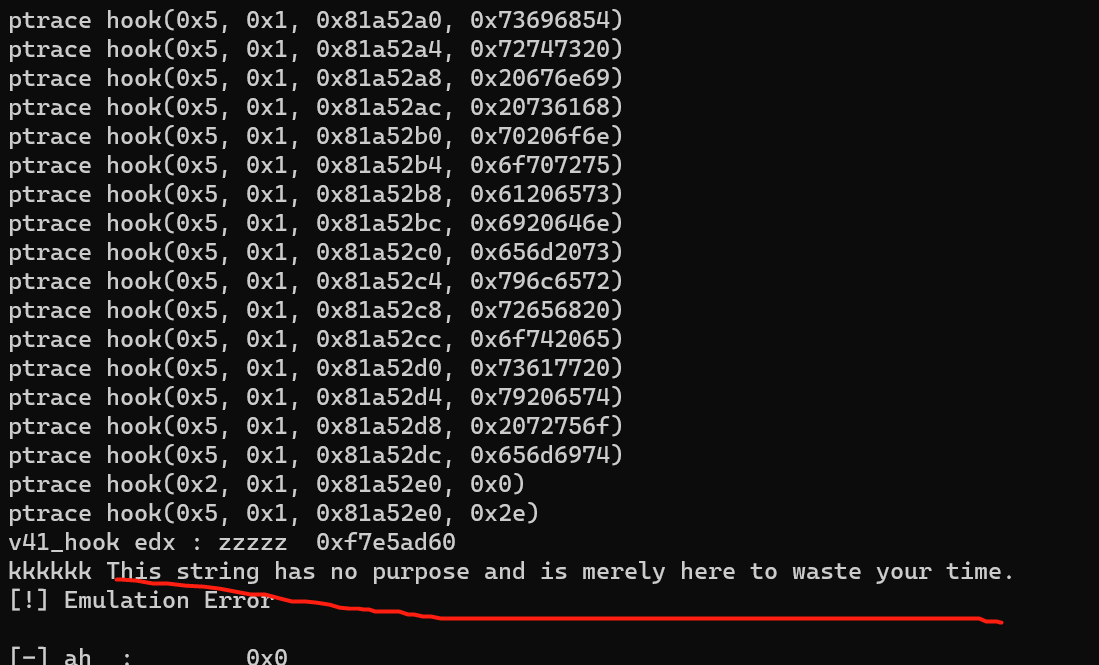
既第一部分flag

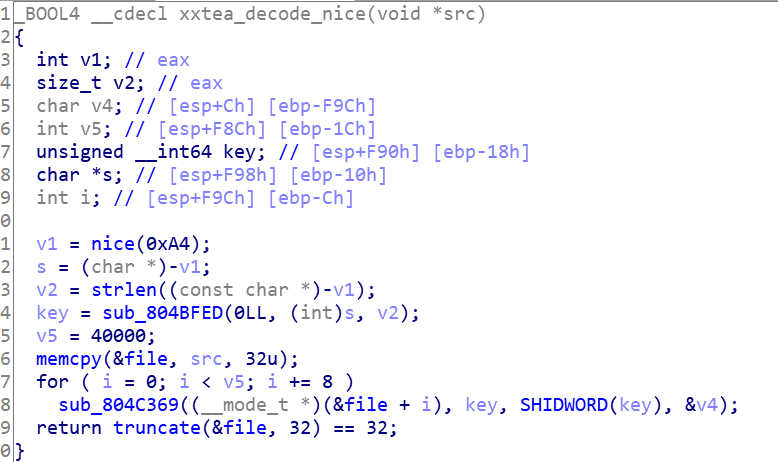
w31c0mE\_t0\_Th3\_l

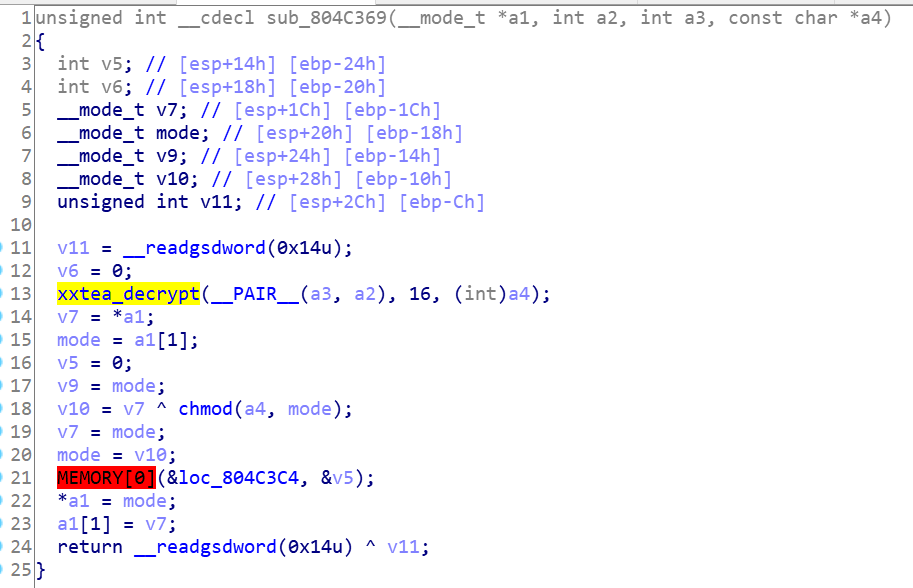
### Flag 第二部分

继续深入，里面有call0操作和一堆syscall劫持，简单分析下算法其实并不复杂。不过我们依然不打算完整还原算法，只通过模拟执行观察关键部分。

这里nice(0xA4)其作为key的种子。虽然告诉你是浪费时间，其实这个字符串是正确的。

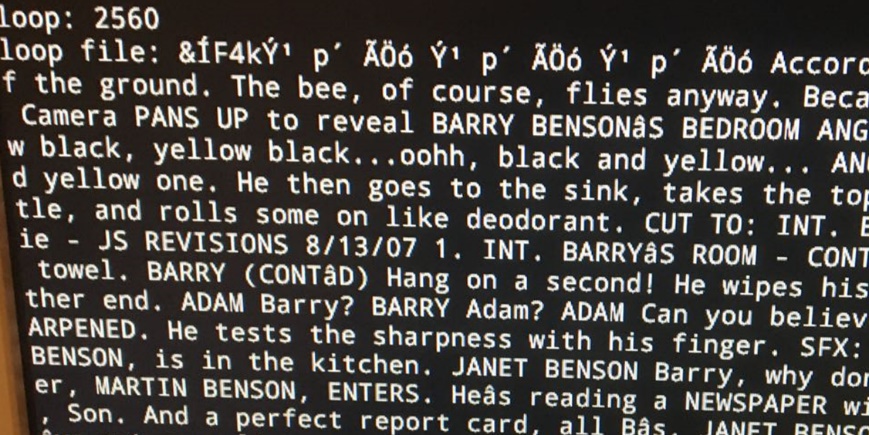






这里循环解码4w字节的.text空间

Hook sub\_804C369返回，打印解码file地址会发现里面是蜜蜂总动员的台词



观察解码输出，我们轻易锁定伪chmodsyscall为算法关键部分。

其chmodsyscall可以概况为

def simple\_decrypt(arg1, key1, key2, key3):

    sum = (arg1 + key1) & 0xffffffff

    sum = (sum >> (key2 & 0x1F)) | (sum << (-(key2 & 0x1F) & 0x1F)) & 0xffffffff

    return (sum ^ key3) & 0xffffffff

chmod会进入call0处理异常。打印参数2即为每轮的密码。得到如下码表，然后构建一个反向表。

elif sysnum == 0x6B4E102C:

        ql.reg.eax = (arg1 + arg2) & 0xffffffff

*#ql.nprint("!xxtea 11111 retaddr: 0x%X arg1:0x%X, arg2:0x%X, retval:0x%X" % (ret, arg1, arg2, ql.reg.eax))*

    elif sysnum == 0x5816452E:

        ql.reg.eax = ((arg1 >> (arg2 & 0x1F)) | (arg1 << (-(arg2 & 0x1F) & 0x1F))) & 0xffffffff

*# ql.nprint("!xxtea 222222 retaddr: 0x%X arg1:0x%X, arg2:0x%X, retval:0x%X" % (ret, arg1, arg2, ql.reg.eax))*

    elif sysnum == 0x44DE7A30:

        ql.reg.eax = (arg1 ^ arg2) & 0xffffffff

*# ql.nprint("!xxtea 3333333 retaddr: 0x%X arg1:0x%X, arg2:0x%X, retval:0x%X" % (ret, arg1, arg2, ql.reg.eax))*

!xxtea 11111 retaddr: 0x804C1C9 arg1:0xF4DC92AA, arg2:0x4B695809, retval:0x4045EAB3

!xxtea 222222 retaddr: 0x804C1EC arg1:0x4045EAB3, arg2:0xF, retval:0xD566808B

!xxtea 3333333 retaddr: 0x804C20C arg1:0xD566808B, arg2:0x674A1DEA, retval:0xB22C9D61

!xxtea 11111 retaddr: 0x804C1C9 arg1:0xA246A2B7, arg2:0xE35B9B24, retval:0x85A23DDB

!xxtea 222222 retaddr: 0x804C1EC arg1:0x85A23DDB, arg2:0x11, retval:0x1EEDC2D1

!xxtea 3333333 retaddr: 0x804C20C arg1:0x1EEDC2D1, arg2:0xAD92774C, retval:0xB37FB59D

。。。。。。

KEYS\_TABLES = {

    0x78F7B625: [0x4B695809, 0xF, 0x674A1DEA],

    0x7C31020B: [0xE35B9B24, 0x11, 0xAD92774C],

    0xF8620416: [0x71ADCD92, 0x11, 0x56C93BA6],

    0x7D1A666D: [0x38D6E6C9, 0x11, 0x2B649DD3],

    0xFA34CCDA: [0x5A844444, 0xC, 0x8B853750],

    0x79B7F7F5: [0x2D422222, 0xC, 0x45C29BA8],

    0xF36FEFEA: [0x16A11111, 0xC, 0x22E14DD4],

    0xE6DFDFD4: [0xCDBFBFA8, 0x15, 0x8F47DF53],

    0xCDBFBFA8: [0xE6DFDFD4, 0x15, 0x47A3EFA9],

    0x16A11111: [0xF36FEFEA, 0x15, 0x23D1F7D4],

    0x2D422222: [0x79B7F7F5, 0x15, 0x11E8FBEA],

    0x5A844444: [0xFA34CCDA, 0xF, 0x96C3044C],

    0x38D6E6C9: [0x7D1A666D, 0xF, 0x4B618226],

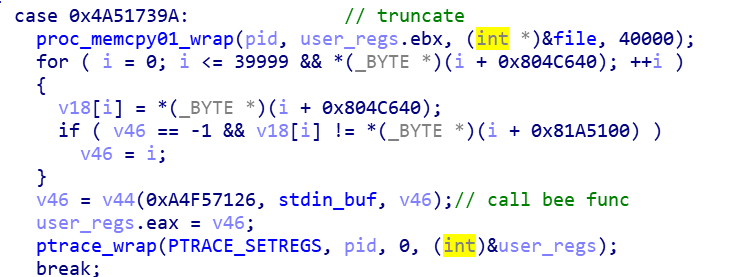
    0x71ADCD92: [0xF8620416, 0xF, 0xBB87B8AA],

    0xE35B9B24: [0x7C31020B, 0xF, 0x5DC3DC55],

    0x4B695809: [0x78F7B625, 0x12, 0xB0D69793],}

修改callzero逻辑，根据参数2反查码表即是反向解码操作。

继续跟进，尾部会进入伪truncate syscall，其会比较0x81A5100的32个字节。

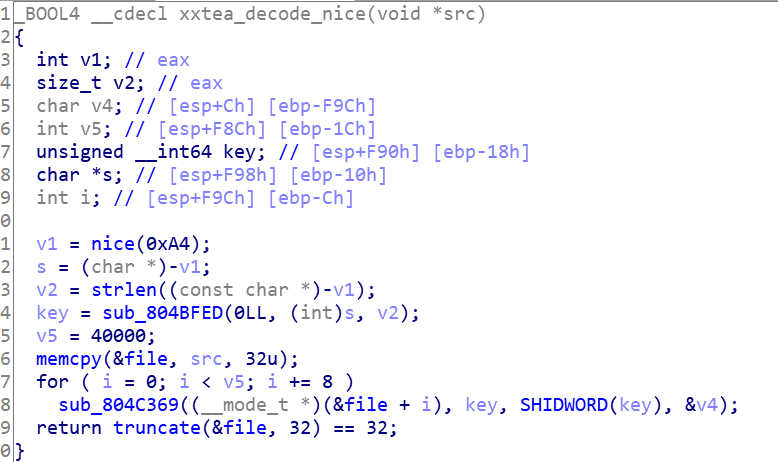


我们直接将0x81A5100地址作为参数传递，跳转这个函数模拟执行，同时修改我们的call\_zero异常处理函数将码表反转。

    ql.stack\_push(0x81A5100)

    ql.stack\_push(0x0)

    ql.reg.eip = 0x8048F05 *#jmp to xxtea\_decode*

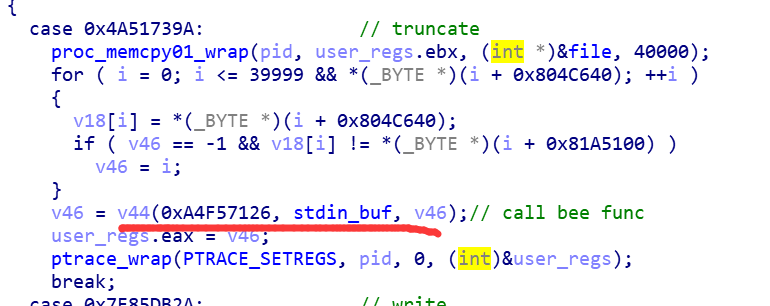


进入truncate打印file即可得到第二部分flag

4nD\_0f\_De4th\_4nd\_d3strUct1oN\_4nd

### Flag第三部分

进入最后的truncate syscall。这个v44栈变量被初始化为0，看起来会进入孙进程的段错误异常处理逻辑





你以为苦难要结束了，兴冲冲的输入flag：

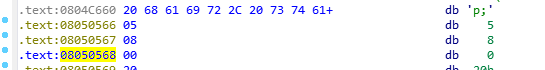
w3lc0mE\_t0\_Th3\_l4nD\_0f\_De4th\_4nd\_d3strUct1oN\_4nd@no-flare.com痛苦的发现依然不对，还得继续努力。

通过我们的模拟执行框架揭示了这个秘密，v44的调用进入了另一个地址，而没有进入你以为的call\_zero处理，显然这里又是一个陷阱。

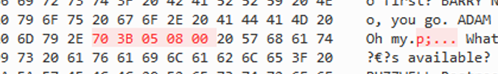
将file地址解码后的内容patch回原二进制，蜜蜂总动员剧本后面会发现出现的新的代码段。



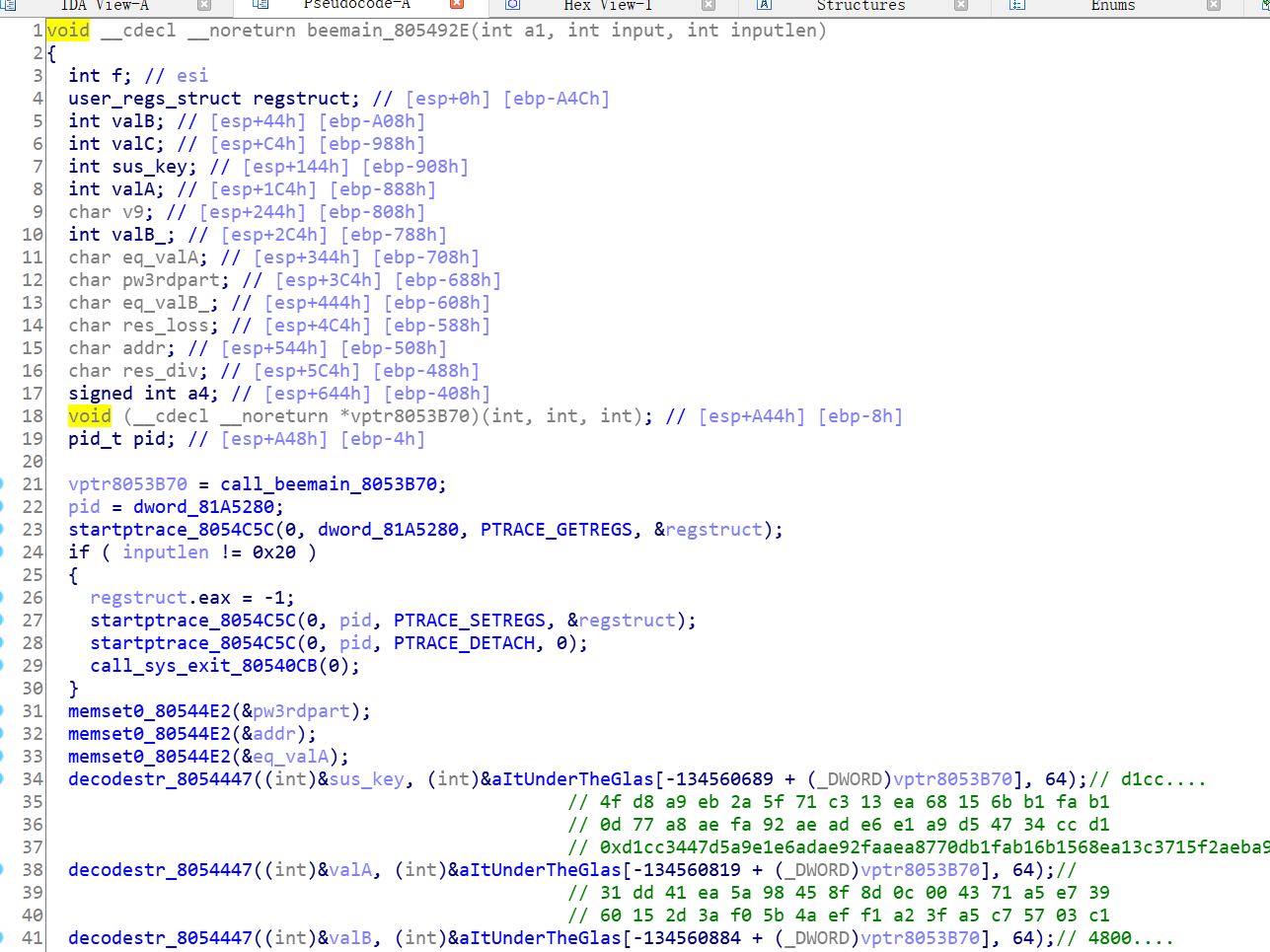
回到truncatesyscall，分析发现v18栈空间分配空间是16000字节，循环拷贝的时候发生了栈溢出，计算偏移后刚好覆盖了栈变量v44的值。从patch后的二进制找蜜蜂总动员剧本找第一个0字节，地址是0x8050568



可以看到存储的正是真正的第三部分比较函数入口地址



Flag第三部分验证主逻辑



分析后得出32字节大数的混合运算

得出方程

0xd1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f  \* res\_div + 0xd036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb = 0xc10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31 \* key

key约束

[根据语法盲猜以\_开头和@flare-on.com](mailto:根据语法盲猜_开头和@flare-on.com)总0x18个字节

\_0000000000@flare-on.com~\_zzzzzzzzzz@flare-on.com

0x6d6f632e6e6f2d6572616c6640303030303030303030305f < key < 0x6d6f632e6e6f2d6572616c66407a7a7a7a7a7a7a7a7a7a5f

算出第三部分flag为

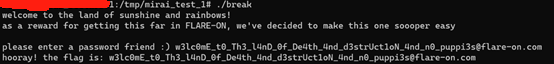
6d6f632e6e6f2d6572616c6640733369707075705f306e5f

即\_n0\_puppi3s@flare-on.com

## 0x04柳暗花明

统合3部分得到最后的flag为：

w3lc0mE\_t0\_Th3\_l4nD\_0f\_De4th\_4nd\_d3strUct1oN\_4nd\_n0\_puppi3s@flare-on.com

ss