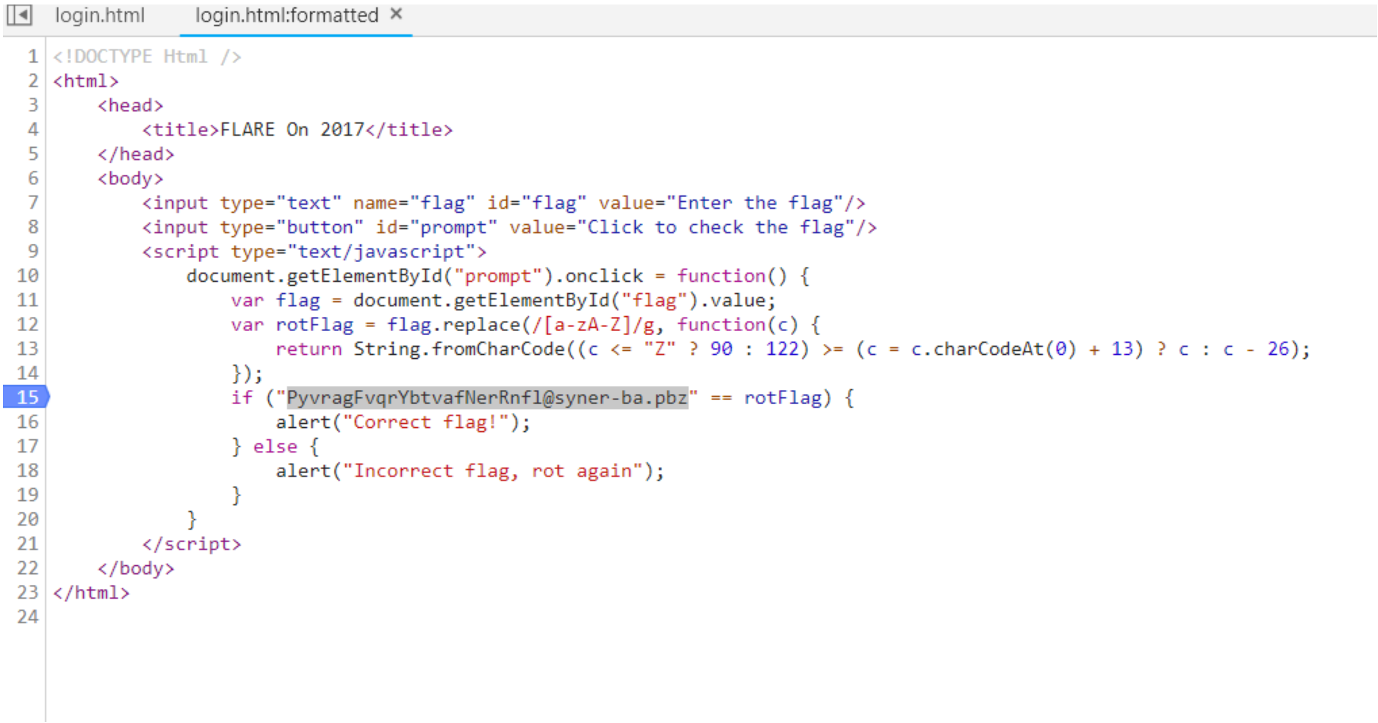
Flare-On4 解题复现

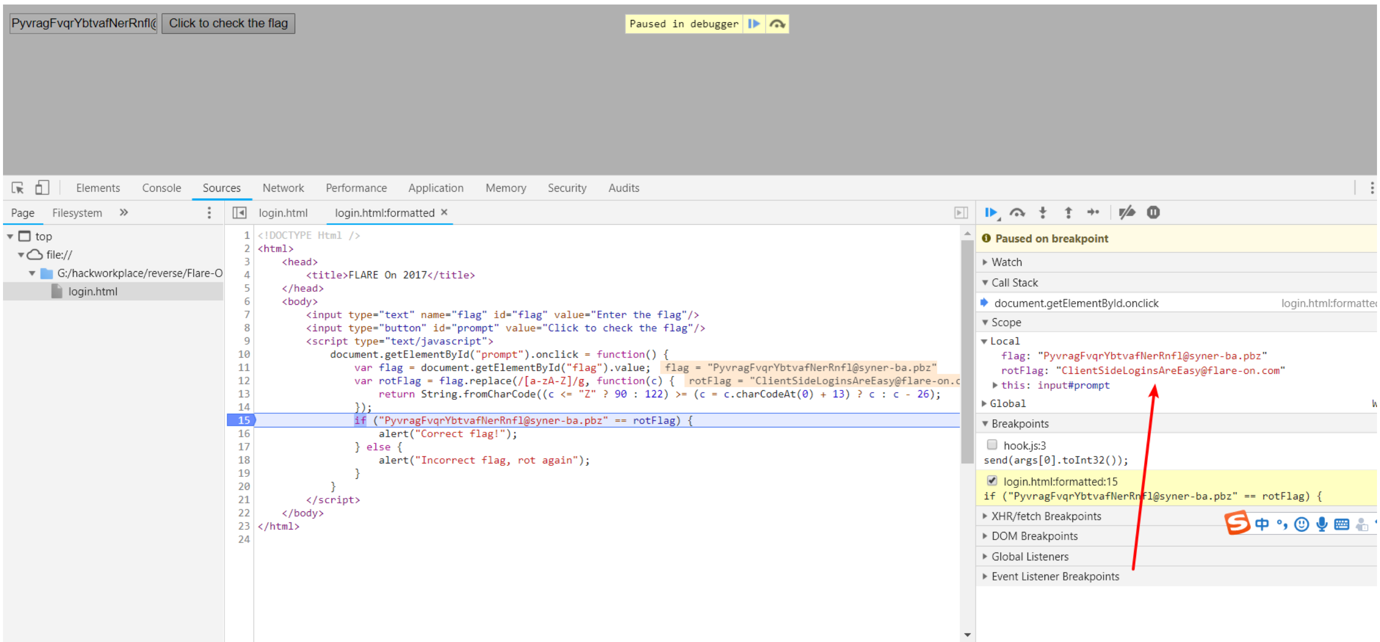
01

是一个 html 页面， 用开发者工具看看，发现是简单的 js 加密。



```
<!DOCTYPE Html />
<html>
    <head>
        <title>FLARE On 2017</title>
    </head>
    <body>
        <input type="text" name="flag" id="flag" value="Enter the flag"/>
        <input type="button" id="prompt" value="Click to check the flag"/>
        <script type="text/javascript">
            document.getElementById("prompt").onclick = function() {
                var flag = document.getElementById("flag").value;
                var rotFlag = flag.replace(/[a-zA-Z]/g, function(c) {
                    return String.fromCharCode((c <= "Z" ? 90 : 122) >= (c = c.charCodeAt(0) + 13) ? c : c - 26);
                });
                if ("PyvragFvqrYbtvafNerRnfl@syner-ba.pbz" == rotFlag) {
                    alert("Correct flag!");
                } else {
                    alert("Incorrect flag, rot again");
                }
            }
        </script>
    </body>
</html>

```

猜测加密算法可逆，试着用 PyvragFvqrYbtvafNerRnfl@syner-ba.pbz 作为输入，然后调试 ，得到 flag 为 ClientSideLoginsAreEasy@flare-on.com



```
<!DOCTYPE Html />
<html>
    <head>
        <title>FLARE On 2017</title>
    </head>
    <body>
        <input type="text" name="flag" id="flag" value="Enter the flag"/>
        <input type="button" id="prompt" value="Click to check the flag"/>
        <script type="text/javascript">
            document.getElementById("prompt").onclick = function() {
                var flag = document.getElementById("flag").value;  flag = "PyvragFvqrYbtvafNerRnfl@syner-ba.pbz"
                var rotFlag = flag.replace(/[a-zA-Z]/g, function(c) {  rotFlag = "ClientSideLoginsAreEasy@flare-on.c
                    return String.fromCharCode((c <= "Z" ? 90 : 122) >= (c = c.charCodeAt(0) + 13) ? c : c - 26);
                });
                if ("PyvragFvqrYbtvafNerRnfl@syner-ba.pbz" == rotFlag) {
                    alert("Correct flag!");
                } else {
                    alert("Incorrect flag, rot again");
                }
            }
        </script>
    </body>
</html>

```

Paused on breakpoint

▸ Watch

▾ Call Stack

document.getElementById.onclick   login.html:formatte

▾ Scope

```
▾ Local
    flag: "PyvragFvqrYbtvafNerRnfl@syner-ba.pbz"
    rotFlag: "ClientSideLoginsAreEasy@flare-on.com"
  ▸ this: input#prompt
▸ Global
```

▾ Breakpoints

□ hook.js:3
`send(args[0].toInt32());`

☑ login.html:formatted:15
`if ("PyvragFvqrYbtvafNerRnfl@syner-ba.pbz" == rotFlag) {`

▸ XHR/fetch Breakpoints

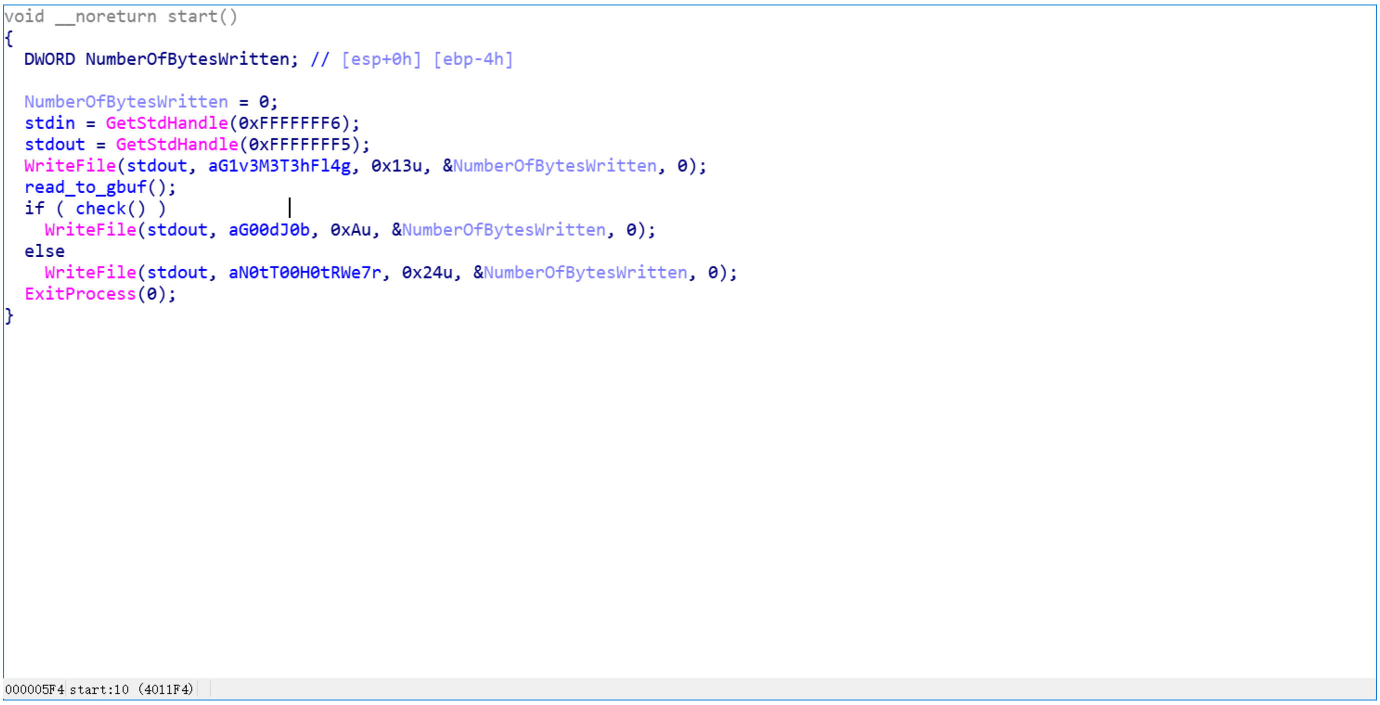
▸ DOM Breakpoints

▸ Global Listeners

▸ Event Listener Breakpoints

02

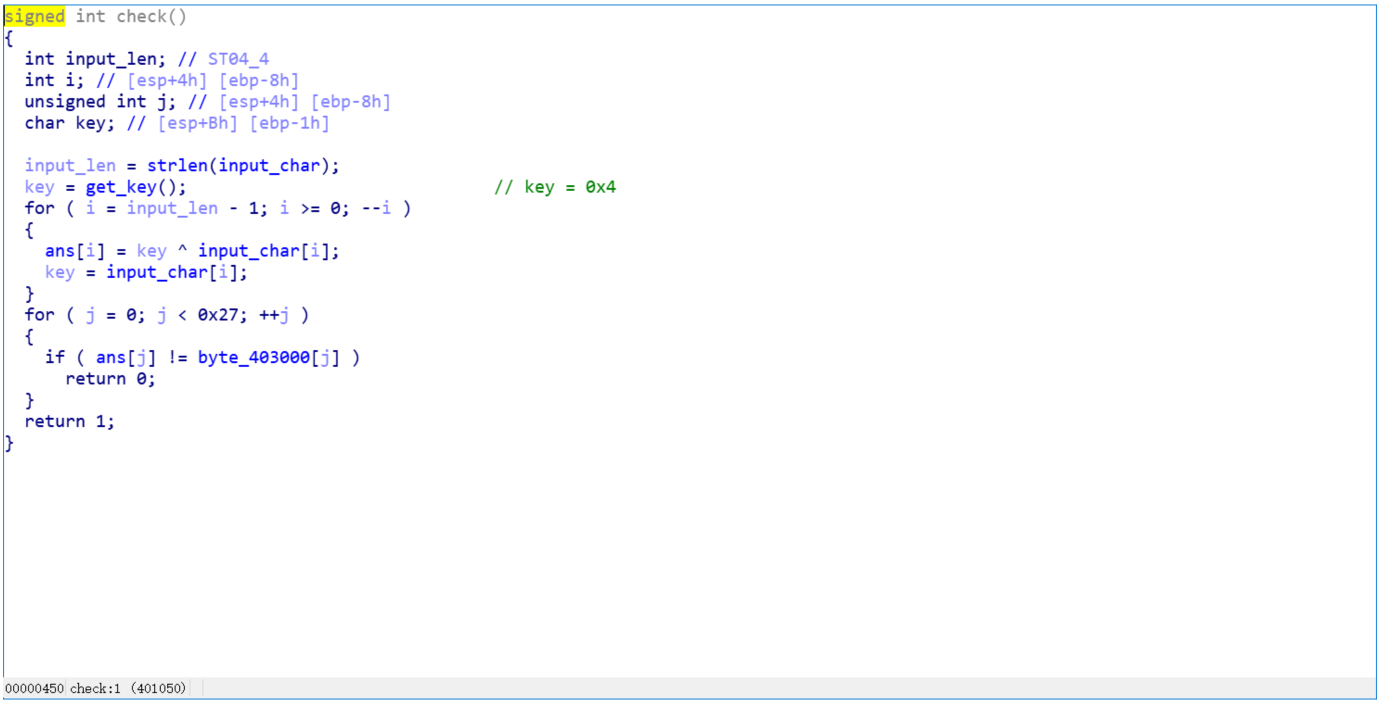
程序逻辑如下



```
void __noreturn start()
{
  DWORD NumberOfBytesWritten; // [esp+0h] [ebp-4h]

  NumberOfBytesWritten = 0;
  stdin = GetStdHandle(0xFFFFFFF6);
  stdout = GetStdHandle(0xFFFFFFF5);
  WriteFile(stdout, aG1v3M3T3hFl4g, 0x13u, &NumberOfBytesWritten, 0);
  read_to_gbuf();
  if ( check() )
    WriteFile(stdout, aG00dJ0b, 0xAu, &NumberOfBytesWritten, 0);
  else
    WriteFile(stdout, aN0tT00H0tRWe7r, 0x24u, &NumberOfBytesWritten, 0);
  ExitProcess(0);
}
```

```
000005F4 start:10 (4011F4)
```

首先 获取输入， 然后 调用 check 进行判断， 下面分析 check 函数



```
signed int check()
{
  int input_len; // ST04_4
  int i; // [esp+4h] [ebp-8h]
  unsigned int j; // [esp+4h] [ebp-8h]
  char key; // [esp+Bh] [ebp-1h]

  input_len = strlen(input_char);
  key = get_key();                              // key = 0x4
  for ( i = input_len - 1; i >= 0; --i )
  {
    ans[i] = key ^ input_char[i];
    key = input_char[i];
  }
  for ( j = 0; j < 0x27; ++j )
  {
    if ( ans[j] != byte_403000[j] )
      return 0;
  }
  return 1;
}
```

```
00000450 check:1 (401050)
```

通过异或操作加密我们的输入， 首先获取一个固定的初始 key , 后面每一步 key 从输入中取，获取到密文后就和程序中已有的密文做对比。

那么 flag 应该就是程序里面那段密文解密后的字符串， 对加密算法求反，写出解密的 idapython 脚本

```
import idc


encoded_data = get_bytes(0x403000, 0x27)


key = 0x4
flag = ""
```

```
i = 0x26

while i >= 0:
    key = ord(encoded_data[i])^key
    flag = chr(key) + flag
    i = i - 1

print flag
```

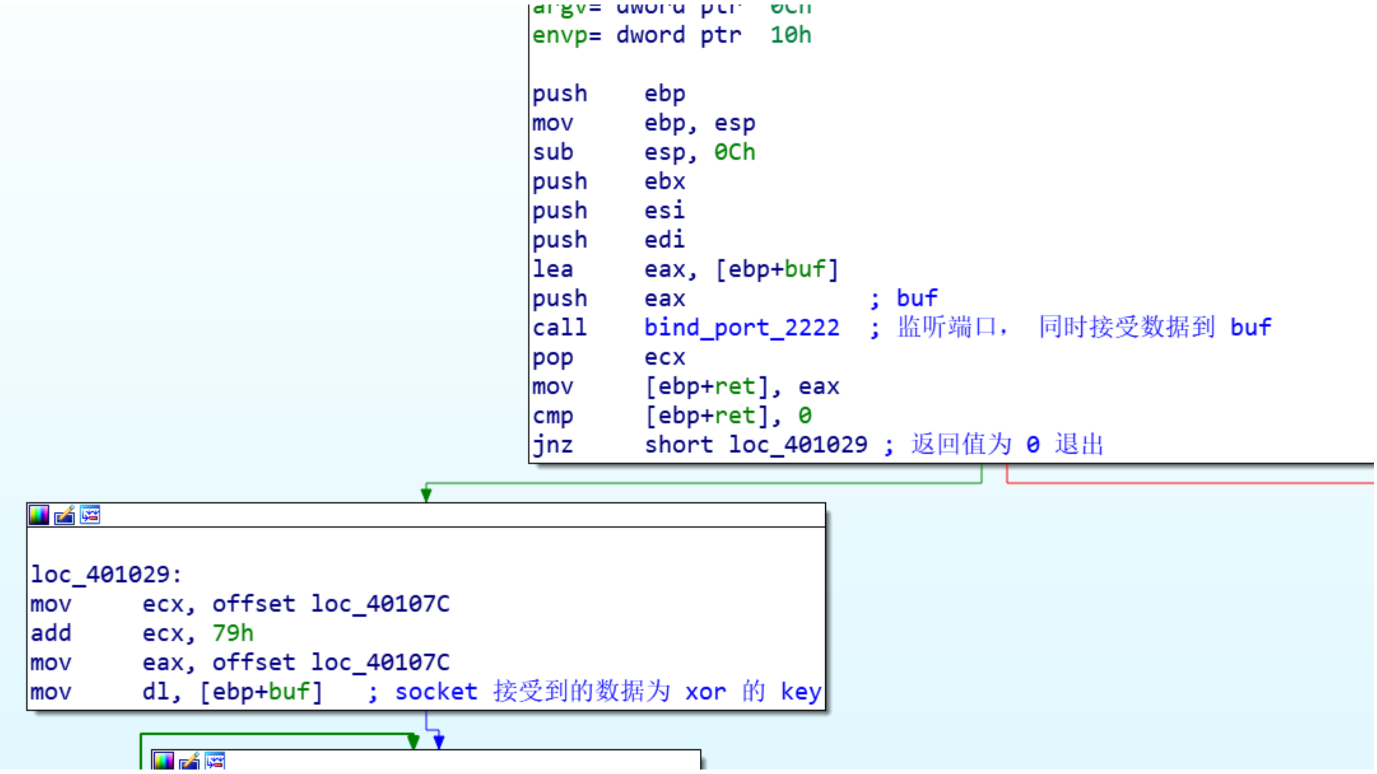
由于 key 是输入中来的，所以这里的 key 应该是解密后的数据。得出 flag 为

R_y0u_H0t_3n0ugH_t0_1gn1t3@flare-on.com

03

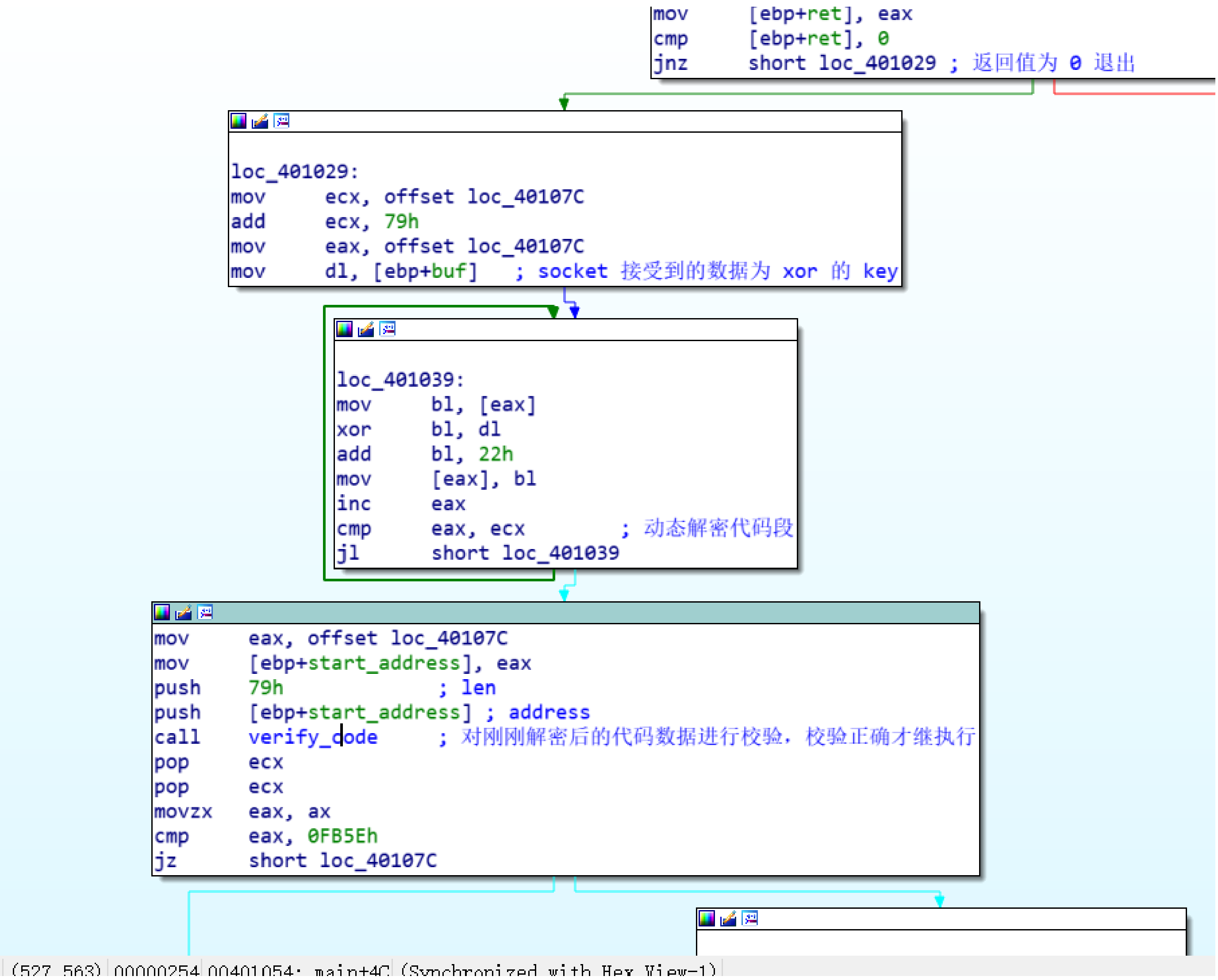
分析

程序首先监听 2222 端口，然后接收 4 个字节



```
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    ebx
push    esi
push    edi
lea     eax, [ebp+buf]
push    eax             ; buf
call    bind_port_2222  ; 监听端口， 同时接受数据到 buf
pop     ecx
mov     [ebp+ret], eax
cmp     [ebp+ret], 0
jnz     short loc_401029 ; 返回值为 0 退出

loc_401029:
mov     ecx, offset loc_40107C
add     ecx, 79h
mov     eax, offset loc_40107C
mov     dl, [ebp+buf]   ; socket 接受到的数据为 xor 的 key
```

然后用刚接收的 4 个字节的其中一个字节作为 key， 对 0x40107C 开始的 0x79字节的代码进行解密，然后校验解密后的数据，校验成功继续执行，如果不成功则退出。



```
mov     [ebp+ret], eax
cmp     [ebp+ret], 0
jnz     short loc_401029 ; 返回值为 0 退出
```

```
loc_401029:
mov     ecx, offset loc_40107C
add     ecx, 79h
mov     eax, offset loc_40107C
mov     dl, [ebp+buf]   ; socket 接受到的数据为 xor 的 key
```

```
loc_401039:
mov     bl, [eax]
xor     bl, dl
add     bl, 22h
mov     [eax], bl
inc     eax
cmp     eax, ecx           ; 动态解密代码段
jl      short loc_401039
```

```
mov     eax, offset loc_40107C
mov     [ebp+start_address], eax
push    79h                ; len
push    [ebp+start_address] ; address
call    verify_code        ; 对刚刚解密后的代码数据进行校验，校验正确才继执行
pop     ecx
pop     ecx
movzx   eax, ax
cmp     eax, 0FB5Eh
jz      short loc_40107C
```

```
(527,563) 00000254 00401054: main+4C (Synchronized with Hex View-1)
```

所以想要继续分析，首先得解出解密代码的 `key` , `key` 的大小为 1 个字节，255 中可能。爆破之即可。

解密

借助 unicorn

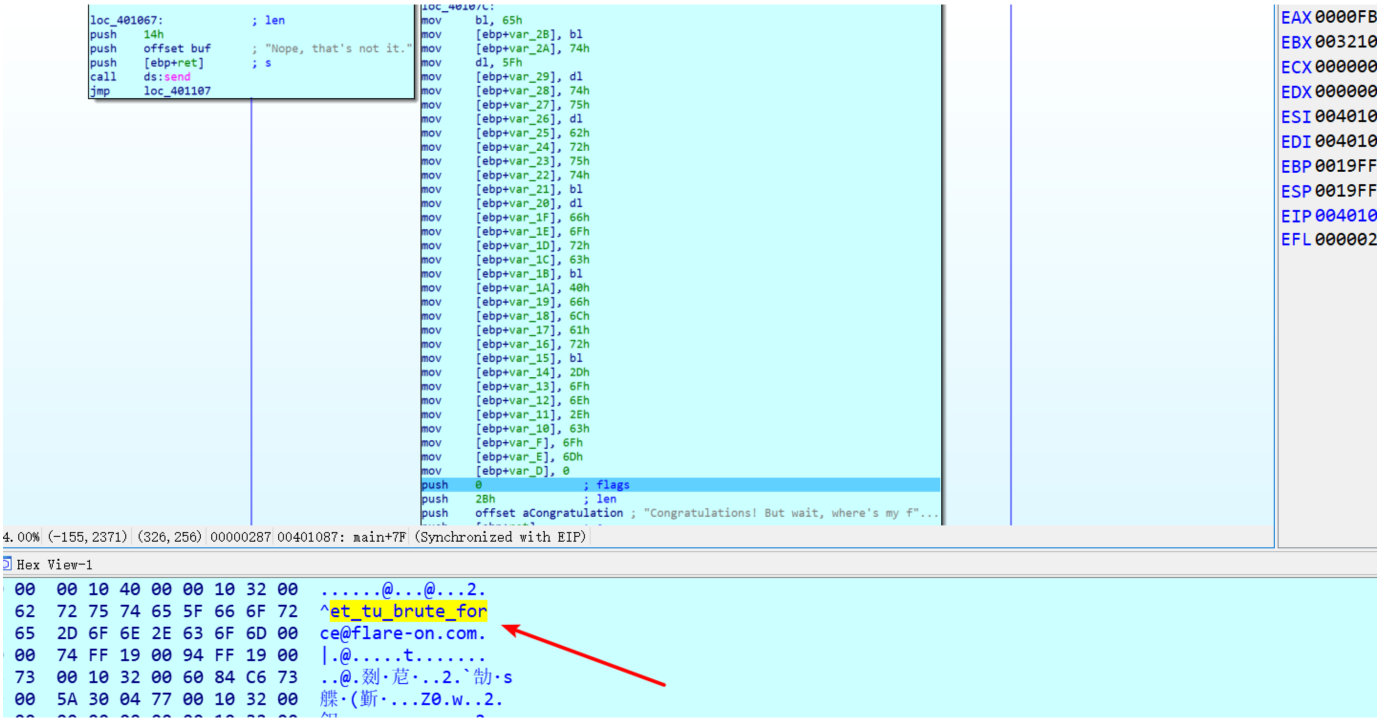
把解密逻辑用 `python` 实现， 然后把 校验解密结果的代码用 `unicorn` 模拟运行，然后整合一下爆破出正确的解密 `key`

```python
import binascii
import struct
from unicorn import *
from unicorn.x86_const import *



def list_to_str(arr):
    res = ""
    for i in arr:
        res += chr(i)
    return res
```

```
verify_code = list_to_str([
    0x55, 0x8B, 0xEC, 0x51, 0x8B, 0x55, 0x0C, 0xB9, 0xFF, 0x00, 0x00, 0x00, 0x89, 0x4D, 0xFC, 0x85,
    0xD2, 0x74, 0x51, 0x53, 0x8B, 0x5D, 0x08, 0x56, 0x57, 0x6A, 0x14, 0x58, 0x66, 0x8B, 0x7D, 0xFC,
    0x3B, 0xD0, 0x8B, 0xF2, 0x0F, 0x47, 0xF0, 0x2B, 0xD6, 0x0F, 0xB6, 0x03, 0x66, 0x03, 0xF8, 0x66,
    0x89, 0x7D, 0xFC, 0x03, 0x4D, 0xFC, 0x43, 0x83, 0xEE, 0x01, 0x75, 0xED, 0x0F, 0xB6, 0x45, 0xFC,
    0x66, 0xC1, 0xEF, 0x08, 0x66, 0x03, 0xC7, 0x0F, 0xB7, 0xC0, 0x89, 0x45, 0xFC, 0x0F, 0xB6, 0xC1,
    0x66, 0xC1, 0xE9, 0x08, 0x66, 0x03, 0xC1, 0x0F, 0xB7, 0xC8, 0x6A, 0x14, 0x58, 0x85, 0xD2, 0x75,
    0xBB, 0x5F, 0x5E, 0x5B, 0x0F, 0xB6, 0x55, 0xFC, 0x8B, 0xC1, 0xC1, 0xE1, 0x08, 0x25, 0x00, 0xFF,
    0x00, 0x00, 0x03, 0xC1, 0x66, 0x8B, 0x4D, 0xFC, 0x66, 0xC1, 0xE9, 0x08, 0x66, 0x03, 0xD1, 0x66,
    0x0B, 0xC2, 0x8B, 0xE5, 0x5D
])

encoded_code = list_to_str(
    [0x33, 0xE1, 0xC4, 0x99, 0x11, 0x06, 0x81, 0x16, 0xF0, 0x32, 0x9F, 0xC4, 0x91, 0x17, 0x06, 0x81,
     0x14, 0xF0, 0x06, 0x81, 0x15, 0xF1, 0xC4, 0x91, 0x1A, 0x06, 0x81, 0x1B, 0xE2, 0x06, 0x81, 0x18,
     0xF2, 0x06, 0x81, 0x19, 0xF1, 0x06, 0x81, 0x1E, 0xF0, 0xC4, 0x99, 0x1F, 0xC4, 0x91, 0x1C, 0x06,
     0x81, 0x1D, 0xE6, 0x06, 0x81, 0x62, 0xEF, 0x06, 0x81, 0x63, 0xF2, 0x06, 0x81, 0x60, 0xE3, 0xC4,
     0x99, 0x61, 0x06, 0x81, 0x66, 0xBC, 0x06, 0x81, 0x67, 0xE6, 0x06, 0x81, 0x64, 0xE8, 0x06, 0x81,
     0x65, 0x9D, 0x06, 0x81, 0x6A, 0xF2, 0xC4, 0x99, 0x6B, 0x06, 0x81, 0x68, 0xA9, 0x06, 0x81, 0x69,
     0xEF, 0x06, 0x81, 0x6E, 0xEE, 0x06, 0x81, 0x6F, 0xAE, 0x06, 0x81, 0x6C, 0xE3, 0x06, 0x81, 0x6D,
     0xEF, 0x06, 0x81, 0x72, 0xE9, 0x06, 0x81, 0x73, 0x7C])


def decode_bytes(i):
    decoded_bytes = ””
    for byte in encoded_code:
        decoded_bytes += chr(((ord(byte) ^ i) + 0x22) & 0xFF)
    return decoded_bytes


def emulate_checksum(decoded_bytes):
    # establish memory addresses for checksum code, stack, and decoded bytes
    address = 0
    stack_addr = 0x10000
    dec_bytes_addr = 0x20000

    # write checksum code and decoded bytes into memory
    mu = Uc(UC_ARCH_X86, UC_MODE_32)
    mu.mem_map(address, 2 * 1024 * 1024)
    mu.mem_write(address, verify_code)
    mu.mem_write(dec_bytes_addr, decoded_bytes)
    # place the address of decoded bytes and size on the stack
    mu.reg_write(UC_X86_REG_ESP, stack_addr)
    mu.mem_write(stack_addr + 4, struct.pack(’<I’, dec_bytes_addr))   # arg1 , address
    mu.mem_write(stack_addr + 8, struct.pack('<I', 0x79))  # arg2 , len

    # emulate and read result in AX
    mu.emu_start(address, address + len(verify_code))
    checksum = mu.reg_read(UC_X86_REG_AX)
    return checksum


for i in range(256):
    checksum = emulate_checksum(decode_bytes(i))
    if checksum & 0xffff == 0xFB5E:
        print(hex(i))
        break`
```

其中 `verify_code` 不需要 `ret` 指令，因为我们只需要函数的返回值。

最后得到的 `key` 为 `0xa2`, 然后在调试的时候，设置正常的 `key`，解密代码后发现是一段复制语句，调试 得到 flag`



```
loc_401067:             ; len
push    14h
push    offset buf      ; "Nope, that's not it."
push    [ebp+ret]       ; s
call    ds:send
jmp     loc_401107
```

```
loc_40107C:
mov     bl, 65h
mov     [ebp+var_2B], bl
mov     [ebp+var_2A], 74h
mov     dl, 5Fh
mov     [ebp+var_29], dl
mov     [ebp+var_28], 74h
mov     [ebp+var_27], 75h
mov     [ebp+var_26], dl
mov     [ebp+var_25], 62h
mov     [ebp+var_24], 72h
mov     [ebp+var_23], 75h
mov     [ebp+var_22], 74h
mov     [ebp+var_21], bl
mov     [ebp+var_20], dl
mov     [ebp+var_1F], 66h
mov     [ebp+var_1E], 6Fh
mov     [ebp+var_1D], 72h
mov     [ebp+var_1C], 63h
mov     [ebp+var_1B], bl
mov     [ebp+var_1A], 40h
mov     [ebp+var_19], 66h
mov     [ebp+var_18], 6Ch
mov     [ebp+var_17], 61h
mov     [ebp+var_16], 72h
mov     [ebp+var_15], bl
mov     [ebp+var_14], 2Dh
mov     [ebp+var_13], 6Fh
mov     [ebp+var_12], 6Eh
mov     [ebp+var_11], 2Eh
mov     [ebp+var_10], 63h
mov     [ebp+var_F], 6Fh
mov     [ebp+var_E], 6Dh
mov     [ebp+var_D], 0
push    0               ; flags
push    2Bh             ; len
push    offset aCongratulation ; "Congratulations! But wait, where's my f"...
```

```
EAX 0000FB
EBX 003210
ECX 000000
EDX 000000
ESI 004010
EDI 004010
EBP 0019FF
ESP 0019FF
EIP 004010
EFL 000002
```

```
4.00% (-155,2371) (326,256) 00000287 00401087: main+7F (Synchronized with EIP)
```

```
Hex View-1
00  00 10 40 00 00 10 32 00  ......@...@...2.
62  72 75 74 65 5F 66 6F 72  ^et_tu_brute_for
65  2D 6F 6E 2E 63 6F 6D 00  ce@flare-on.com.
00  74 FF 19 00 94 FF 19 00  |.@.....t.......
73  00 10 32 00 60 84 C6 73  ..@.剟·苊·..2.`劼·s
00  5A 30 04 77 00 10 32 00  牒·(蘄·...Z0.w..2.
```

flag

et_tu_brute_force@flare-on.com

借助 frida

```
# -*- coding:utf-8 -*-
from __future__ import print_function
import frida
from time import sleep
retval = 0
is_ret = 0
```

```
def list_to_str(arr):
    res = ””
    for i in arr:
        res += chr(i)
    return res

def str_to_list(string):
    res = []
    for i in string:
        res.append(ord(i))
    return res



encoded_code_array = [0x33, 0xE1, 0xC4, 0x99, 0x11, 0x06, 0x81, 0x16, 0xF0, 0x32, 0x9F, 0xC4, 0x91, 0x17, 0x06, 0x81,
                      0x14, 0xF0, 0x06, 0x81, 0x15, 0xF1, 0xC4, 0x91, 0x1A, 0x06, 0x81, 0x1B, 0xE2, 0x06, 0x81, 0x18,
                      0xF2, 0x06, 0x81, 0x19, 0xF1, 0x06, 0x81, 0x1E, 0xF0, 0xC4, 0x99, 0x1F, 0xC4, 0x91, 0x1C, 0x06,
                      0x81, 0x1D, 0xE6, 0x06, 0x81, 0x62, 0xEF, 0x06, 0x81, 0x63, 0xF2, 0x06, 0x81, 0x60, 0xE3, 0xC4,
                      0x99, 0x61, 0x06, 0x81, 0x66, 0xBC, 0x06, 0x81, 0x67, 0xE6, 0x06, 0x81, 0x64, 0xE8, 0x06, 0x81,
                      0x65, 0x9D, 0x06, 0x81, 0x6A, 0xF2, 0xC4, 0x99, 0x6B, 0x06, 0x81, 0x68, 0xA9, 0x06, 0x81, 0x69,
                      0xEF, 0x06, 0x81, 0x6E, 0xEE, 0x06, 0x81, 0x6F, 0xAE, 0x06, 0x81, 0x6C, 0xE3, 0x06, 0x81, 0x6D,
                      0xEF, 0x06, 0x81, 0x72, 0xE9, 0x06, 0x81, 0x73, 0x7C]

encoded_code = list_to_str(encoded_code_array)

def on_message(message, data):
    global retval,is_ret
    retval = message[’payload’]
    is_ret = 1



def decode_bytes(i):
    decoded_bytes = ””
    for byte in encoded_code:
        decoded_bytes += chr(((ord(byte) ^ i) + 0x22) & 0xFF)
    return decoded_bytes



def main():
    global retval, is_ret
    session = frida.attach(”greek_to_me.exe”)
    for i in range(256):
        script = session.create_script(”””
            var verify_code = ptr(’0x4011E6’);
            var f = new NativeFunction(verify_code, ’int’, [’pointer’, ’int’]);
            var save_address = ptr(’0x40107C’);
            Memory.writeByteArray(save_address, {})
            send(f(save_address, 121));
        ”””.format(str_to_list(decode_bytes(i))))
        script.on(’message’, on_message)
        script.load()

        while is_ret != 1:  # 等待远程函数执行完
            sleep(0.2)
        is_ret = 0

        if retval & 0xffff == 0xFB5E:
            print(hex(i))
            break

    session.detach()

if __name__ == ’__main__’:
    main()
```

每次解密`code`后，直接用 `frida` 调用进程里面的校验函数，通过这样可以爆破出 `key`

最后附一个导出光标所在函数的二进制代码的 `idapython` 脚本

```
import idaapi

def list_to_str(arr):
    res = ””
    for i in arr:
        res += chr(i)
    return res

def str_to_list(string):
    res = []
    for i in string:
        res.append(ord(i))
    return res

compiled_functions = {}
def ida_run_python_function(func_name):
    if func_name not in compiled_functions:
        ida_func_name = ”py_%s” % func_name
        idaapi.CompileLine(’static %s() { RunPythonStatement(”%s()”); }’
            % (ida_func_name, func_name))
        compiled_functions[func_name] = ida_func_name
    return ida_func_name
```

```
def GetFunctionCode():
    func_start = get_func_attr(here(), FUNCATTR_START)
    func_end = get_func_attr(here(), FUNCATTR_END)
    func_name = GetFunctionName(func_start)
    data = get_bytes(func_start, func_end - func_start)
    with open(func_name, "wb") as fp:
        fp.write(data)

    with open(func_name + ".list", "w") as fp:
        fp.write(str(str_to_list(data)))


    Message("Write code of %s done!!!\n" %(func_name))


AddHotkey("Ctrl+Shift+A", ida_run_python_function("GetFunctionCode"));
```

参考

http://blog.nsfocus.net/flare-onchallenge4th/

来源： https://www.cnblogs.com/hac425/p/9752840.html