The Flare-On Challenge 2016

Reno Robert
www.voidsecurity.in
@renorobertr

FLARE-ON CHALLENGE #1

Looking into the strings of challeng1.exe, will reveal the below details:

```
.rdata:0040D160 encoded         db
'x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q',0
.rdata:0040D160
.rdata:0040D195                 align 4
.rdata:0040D198 aEnterPassword  db 'Enter password:',0Dh,0Ah,0
.rdata:0040D1AA                 align 4
.rdata:0040D1AC aCorrect        db 'Correct!',0Dh,0Ah,0
.rdata:0040D1B7                 align 4
.rdata:0040D1B8 aWrongPassword  db 'Wrong password',0Dh,0Ah,0
```

Cross references to these strings points to function @00401420

```
flag = "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q";

WriteFile(stdout, "Enter password:\r\n", 0x12u, &NumberOfBytesWritten, 0);
ReadFile(stdin, input, 0x80u, &NumberOfBytesWritten, 0);
encoded = encode((int)input, NumberOfBytesWritten - 2);

if ( validate(encoded, flag) )
    WriteFile(stdout, "Wrong password\r\n", 0x11u, &NumberOfBytesWritten, 0);
else
    WriteFile(stdout, "Correct!\r\n", 0xBu, &NumberOfBytesWritten, 0);
```

Encode function@00401260 does base64 encode of user input using a different charset.

```
.data:00413000 ; char base64[]
.data:00413000 base64           db
'ZYXABCDEFGHIJKLMNOPQRSTUVWzyxabcdefghijklmnopqrstuvw0123456789+/',0
```

The result is then compared with a hardcoded string. To get the flag, the string **"x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q"** needs to be decoded using the custom charset. Below is the solution:

```
import base64
import string

cus_b64 = "ZYXABCDEFGHIJKLMNOPQRSTUVWzyxabcdefghijklmnopqrstuvw0123456789+/"
std_b64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
encflag = "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q"

encflag = encflag.translate(string.maketrans(cus_b64, std_b64))
print base64.b64decode(encflag)
```

Flag: sh00ting_phish_in_a_barrel@flare-on.com

FLARE-ON CHALLENGE #2

DudeLocker.exe is designed like a ransomware. Along with the binary, an encrypted file BusinessPapers.doc is provided. Challenge is to decrypt the file and read the contents

The binary first gets path to Desktop folder as below:

```
SHGetFolderPathW(NULL, CSIDL_DESKTOPDIRECTORY, NULL, 0, pszPath);
```

Then a handle to Briefcase directory in Desktop is obtained using CreateFileW API by setting FILE_FLAG_BACKUP_SEMANTICS [0x2000000] flag.

Key Generation for Encryption

Key for encryption of data is done in multiple steps.

Step I

Function@00401040 fetches the volume serial number using

```
GetVolumeInformationA(0, 0, 0, &VolumeSerialNumber, 0, 0, 0, 0);
```

Then checks if the volume serial number equals 0x7DAB1D35. This function can be patched to always return 0x7DAB1D35

Step II

Function@00401940 uses the VolumeSerialNumber as key to decrypt another array defined @00403000

```
key = [0x35, 0x1D, 0xAB, 0x7D]

ciphertext = [65, 117, 196, 14, 80, 123, 194, 17, 80, 110, 217, 24, 84, 113,
199, 4, 65, 116, 206, 25, 65, 117, 206, 27, 90, 113, 207, 24, 71, 105, 196,
26, 80, 105, 195, 24, 71]

plaintext = ''
for i in range(len(ciphertext)):
    plaintext += chr(ciphertext[i] ^ key[i % len(key)])

print plaintext
```

Decrypting the array yields **thosefilesreallytiedthefoldertogether**

Step III

Function@00401080 acquires a handle to AES Cryptographic Service Provider

```
CryptAcquireContextW(phProv, 0, 0, PROV_RSA_AES, CRYPT_NEWKEYSET);
```

Function@00401180 derives AES key using the key found from step II

```
CryptCreateHash(hProv, CALG_SHA, 0, 0, &phHash);
CryptHashData(phHash, stage2_key, dwDataLen, 0);
CryptDeriveKey(hProv, CALG_AES_256, phHash, CRYPT_EXPORTABLE, phKey);
```

Then the derived key is set as

```
CryptSetKeyParam(*phKey, KP_MODE, CRYPT_MODE_CBC, 0);
```

Encryption

Function@00401300 fetches files from Briefcase folder and encrypts each of them. It calls function@00401770 to set IV for AES algorithm. The IV is nothing but MD5 hash of filename. Block size of AES == Digest size of MD5

```
CryptGetKeyParam(*phKey, KP_BLOCKLEN, pbData, &pdwDataLen, 0);
CryptCreateHash(*hProv, CALG_MD5, 0, 0, &phHash);
CryptHashData(phHash, filename, size, 0);
CryptGetHashParam(phHash, HP_HASHVAL, *raw_hash, &size, 0);
CryptSetKeyParam(*phKey, KP_IV, *raw_hash, 0);
```

Encryption of file content is done by function@00401500 using ReadFile, CryptEncrypt, WriteFile sequence.

Now the encrypted file can be decrypted since the algorithm is known. Details of implementing key generation can be found here. Below is the solution:

```
#!/usr/bin/env python

from Crypto.Cipher import AES
from hashlib import md5, sha1

filename = 'BusinessPapers.doc'
aeskey = 'thosefilesreallytiedthefoldertogether'

def prehash(key, byte):
    return sha1(''.join([chr(ord(i)^byte) for i in
sha1(key).digest()]+[chr(byte)]*44)).digest()

def cryptderivekey(key):
    return (prehash(key,0x36)+prehash(key,0x5c))[:32]

passphrase = cryptderivekey(aeskey)
IV = md5(filename.lower()).digest()

aes = AES.new(passphrase, AES.MODE_CBC, IV)

encrypted = open(filename).read()
decrypted = aes.decrypt(encrypted)

decrypted_file = open(filename + '_decrypted','wb')
decrypted_file.write(decrypted)
decrypted_file.close()
```

Flag: cl0se_t3h_f1le_On_th1s_One@flare-on.com

FLARE-ON CHALLENGE #3

For this challenge, an executable named “unknown” is provided. Some interesting strings can be found as below:

```
.rdata:00415A24 aYouMakeGoodArg db 'yOU MAKE GOOD Arguhments!',0
.rdata:00415A3E                 align 10h
.rdata:00415A40 aNoRiteArguhmen db 'No rite arguhments!',0
.rdata:00415A40
```

Analyzing the code just above the string references, the binary loops over 27 times:

```
for ( counter = 0; counter < 27; ++counter )
{
    LOWORD(hash_checks[0]) = *(_WORD *)flag;
    flag += 2;
    ++HIWORD(hash_checks[0]);
    calc_hash = hash((_WORD *)hash_checks);
    target_hash = *decrypted_hashes;
    ++decrypted_hashes;
    if ( calc_hash != target_hash )  ret = 0xB019815A;
}
```

Where hash_checks is an array of integers holding 2-byte string **"__FLARE On!"**

```
hash_checks[0] = 0x5F005F;
hash_checks[1] = 0x4C0046;
hash_checks[2] = 0x520041;
hash_checks[3] = 0x200045;
hash_checks[4] = 0x6E004F;
hash_checks[5] = 0x21;
```

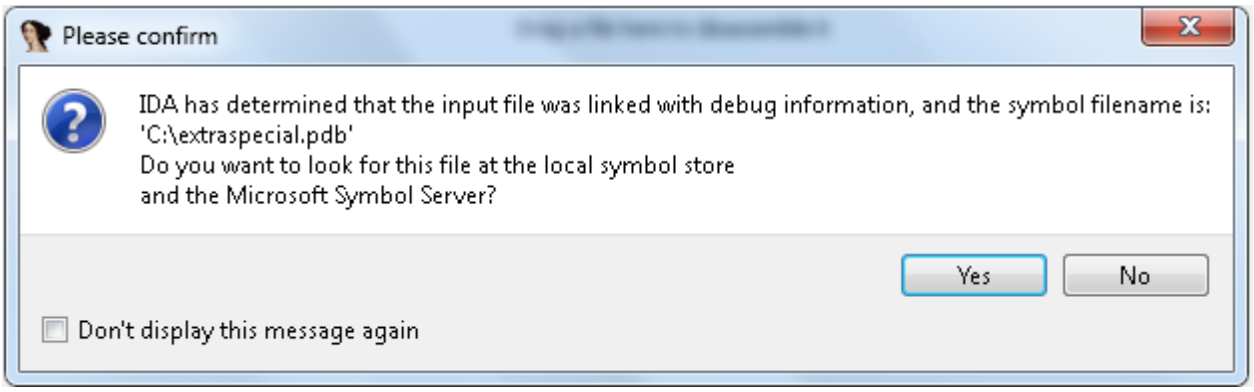
flag variable is user supplied argument which is 27 bytes long and decrypted_hashes points to a hardcoded array @004181C0. This array is not used directly, but is decrypted by function@00401230 before performing hash checks.

Key Generation for Decryption:

The key generation happens over multiple steps with part of filename being used as seed.

```
substring = wcsrchr(*argv, 'r');
seed = hash(substring + 1);
```

At this point something flashed on to mind, IDA message box to look for symbol file when loading the binary:



Since the symbol file name is extraspecial.pdb, the binary should be extraspecial.exe

The decryption function@ 00401230 takes in 3 parameters, 3rd one being 104 ie. 26 DWORDS. So probably the flag is 26 bytes long followed by NUL

```
decrypt(table, &encrypted_hashes, 104);
```

By renaming the binary to extraspecial.exe and by providing a 26 byte dummy string as input, hardcoded target hashes can be decrypted and dumped.

```
>gdb -q extraspecial.exe

(gdb) break *0x00402B7E
Breakpoint 1 at 0x402b7e

(gdb) run AAAAAAAAAAAAA@flare-on.com

[New Thread 8284.0x2130]

Breakpoint 1, 0x00402b7e in ?? ()
(gdb) x/27wx 0x004181C0
0x4181c0:       0xee613e2f      0xde79eb45      0xaf1b2f3d      0x8747bbd7
0x4181d0:       0x739ac49c      0xc9a4f5ae      0x4632c5c1      0xa0029b24
0x4181e0:       0xd6165059      0xa6b79451      0xe79d23ba      0x8aae92ce
0x4181f0:       0x85991a18      0xfee05899      0x430c7994      0x1ab9f36f
0x418200:       0x70c42481      0x05bd27cf      0xc4ff6e6f      0x5a77847c
0x418210:       0xdd9277b3      0x25843cff      0x5fdca944      0x8ee42896
0x418220:       0x2ae961c7      0xa77731da      0x00000000
(gdb)
```

Solver:

```
import string

def calc_hash(str):
    hash = 0
    for i in range(len(str)):
        hash = (str[i] + 0x25 * hash) & 0xffffffff
    return hash

hashes = [0xee613e2f, 0xde79eb45, 0xaf1b2f3d, 0x8747bbd7,
          0x739ac49c, 0xc9a4f5ae, 0x4632c5c1, 0xa0029b24,
          0xd6165059, 0xa6b79451, 0xe79d23ba, 0x8aae92ce,
          0x85991a18, 0xfee05899, 0x430c7994, 0x1ab9f36f,
          0x70c42481, 0x05bd27cf, 0xc4ff6e6f, 0x5a77847c,
          0xdd9277b3, 0x25843cff, 0x5fdca944, 0x8ee42896,
          0x2ae961c7, 0xa77731da, 0x00000000]

flare = [0x5f, 0x5f, 0x46, 0x4c, 0x41, 0x52, 0x45, 0x20, 0x4f, 0x6e, 0x21]
flag = ''

for i in range(len(hashes)):
    flare[1] = flare[1] + 1
    for c in string.printable:
        flare[0] = ord(c)
        if calc_hash(flare) == hashes[i]:
            flag += c
            break
print flag
```

Flag: Ohs0pec1alpwd@flare-on.com

FLARE-ON CHALLENGE #4

The given flareon2016challenge.dll exports 51 functions which can be invoked using ordinal numbers. Functions from 1-48 looks similar. They modify a 16 byte array@10007014 and return some value.

Analyzing the functions 49-51, the function 50 is the only function taking user input. It expects inputs for Beep function and if 18 inputs are provided, the function further invokes function ordinal 49.

Function ordinal 49 invokes couple of functions 0x10001530 and 0x10001000, before calling printf

```
printf("%s\n", &buffer);
```

Assuming this as flag buffer, the code can be renamed as below:

```
generate_key(&key, flag_key, 16);
decrypt(&key, &flag, &flag, 32);
printf("%s\n", &flag);
```

32 bytes passed to decrypt function should be the size of flag. The function ordinal 51 also looks similar:

```
generate_key(&key, music_key, 16);
decrypt(&key, &music, &music, 6672);
printf(aPlayMeASongHav)
```

The 6672 bytes music buffer should have the details needed for the decrypting the flag. All the functions 1-48 modifies music_key buffer. So the challenge is to find the right sequence to invoke the functions.

Finding the sequence:

All the functions have the below sequence of instructions for calculating the return value

```
.text:10002570                  push    ebp
.text:10002571                  mov     ebp, esp
.text:10002573                  sub     esp, 8
.text:10002576                  push    esi
.text:10002577                  mov     [ebp+var_4], 0FDh
.text:1000257E                  mov     [ebp+var_8], 0CEh

.text:100025C3                  mov     eax, [ebp+var_4]
.text:100025C6                  xor     eax, [ebp+var_8] ; 0xFD ^ 0xCE
.text:100025C9                  pop     esi
.text:100025CA                  mov     esp, ebp
.text:100025CC                  pop     ebp
.text:100025CD                  retn
```

IDA python can be used to find the return values of all the functions:

```
EAT = 0x100064A8
count = 48
ordinal_list = []

for address in range(EAT, EAT + count*4, 4):
    func = 0x10000000 + Dword(address)
    a = DecodeInstruction(func + 0x7).Op2.value
    b = DecodeInstruction(func + 0xe).Op2.value
    ordinal = a ^ b
    index = ((address - EAT)/4) + 1
    ordinal_list.append(ordinal)
    print "FUNC[%02d] -> %d" % (index, ordinal)

ordinal_list.sort()
print ordinal_list
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 51]
```

From the above ordinal list 30, 49 and 50 are missing. Functions 49 and 50 does not modify music_key. So probably function ordinal 30 should be invoked first and the return value should be used as next ordinal number.

FUNC[30] -> 15

Modified version of previous IDA python script can be used to validate this assumption:

```
EAT = 0x100064A8
count = 48
ordinal_list = {}

for address in range(EAT, EAT + count*4, 4):
    func = 0x10000000 + Dword(address)
    a = DecodeInstruction(func + 0x7).Op2.value
    b = DecodeInstruction(func + 0xe).Op2.value
    ordinal = a ^ b
    index = ((address - EAT)/4) + 1
    ordinal_list[index] = ordinal

ordinal = 30
sequence = [ordinal]
for _ in range(len(ordinal_list)):
    ordinal = ordinal_list[ordinal]
    sequence.append(ordinal)

print sequence
```

```
[30, 15, 42, 18, 12, 9, 4, 17, 33, 31, 44, 34, 19, 38, 40, 48, 37, 10, 25, 3,
26, 32, 22, 2, 8, 13, 28, 45, 5, 47, 20, 36, 7, 16, 29, 39, 24, 41, 43, 46,
14, 11, 21, 27, 23, 6, 35, 1, 51]
```

Starting with function ordinal 30, the sequence ends at 51, which does the decryption. Now let's invoke the functions in sequence and dump the music buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

#define LIBRARY "flareon2016challenge.dll"
#define START 30
#define END 51

typedef HRESULT(WINAPI *FUNC_Beep)(DWORD, DWORD);

FARPROC fn[52] = { 0 };

int main(int argc, char **argv)
{
    HINSTANCE hDLL = LoadLibraryA(LIBRARY);
    FUNC_Beep func_beep;

    for (int ordinal = 1; ordinal < 52; ordinal++) {
        if (ordinal == 50) {
            func_beep = (FUNC_Beep)GetProcAddress(hDLL,
(LPCSTR)MAKEINTRESOURCE(ordinal));
        }
        else {
            fn[ordinal] = GetProcAddress(hDLL,
(LPCSTR)MAKEINTRESOURCE(ordinal));
        }
    }

    char *music =  (char *)func_beep + 0x4110;
    int music_sz = 6672;
    DWORD no_music_write = 0;

    HANDLE source = CreateFileA("dump", GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    int ret = START;
    while (true) {
        ret = fn[ret]();
        if (ret == END){
            fn[END]();
            break;
        }
    }

    WriteFile(source, (LPCVOID)music, music_sz, &no_music_write, NULL);

    return 0;
}
```

Dumping the decrypted memory gives another exe file.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  Beep(0x1B8u, 0x1F4u);
  Beep(0x1B8u, 0x1F4u);
  Beep(0x1B8u, 0x1F4u);
  Beep(0x15Du, 0x15Eu);
  Beep(0x20Bu, 0x96u);
  Beep(0x1B8u, 0x1F4u);
  Beep(0x15Du, 0x15Eu);
  Beep(0x20Bu, 0x96u);
  Beep(0x1B8u, 0x3E8u);
  Beep(0x293u, 0x1F4u);
  Beep(0x293u, 0x1F4u);
  Beep(0x293u, 0x1F4u);
  Beep(0x2BAu, 0x15Eu);
  Beep(0x20Bu, 0x96u);
  Beep(0x19Fu, 0x1F4u);
  Beep(0x15Du, 0x15Eu);
  Beep(0x20Bu, 0x96u);
  Beep(0x1B8u, 0x3E8u);
  return 0;
}
```

Flag can be obtained by using the Beep sequence in the dumped code as below:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

#define LIBRARY "flareon2016challenge.dll"

typedef VOID(__cdecl *FUNC_Beep)(DWORD, DWORD);

int main(int argc, char **argv)
{

    HINSTANCE hDLL = LoadLibraryA(LIBRARY);
    FUNC_Beep func_beep;
    func_beep = (FUNC_Beep)GetProcAddress(hDLL, (LPCSTR)MAKEINTRESOURCE(50));

    func_beep(0x1B8u, 0x1F4u);
    func_beep(0x1B8u, 0x1F4u);
    func_beep(0x1B8u, 0x1F4u);
    func_beep(0x15Du, 0x15Eu);
    func_beep(0x20Bu, 0x96u);
    func_beep(0x1B8u, 0x1F4u);
    func_beep(0x15Du, 0x15Eu);
    func_beep(0x20Bu, 0x96u);
    func_beep(0x1B8u, 0x3E8u);
    func_beep(0x293u, 0x1F4u);
    func_beep(0x293u, 0x1F4u);
    func_beep(0x293u, 0x1F4u);
    func_beep(0x2BAu, 0x15Eu);
    func_beep(0x20Bu, 0x96u);
```

```
    func_beep(0x19Fu, 0x1F4u);
    func_beep(0x15Du, 0x15Eu);
    func_beep(0x20Bu, 0x96u);
    func_beep(0x1B8u, 0x3E8u);

    return 0;
}
```

```
>flag.exe
f0ll0w_t3h_3xp0rts@flare-on.com
```

FLARE-ON CHALLENGE #5

The given binary implements a virtual machine. The main function copies user supplied argument into a global buffer@ 0040DF20, which I named as VM_STACK. The expected input is 10 bytes long.

```
for ( i = 0; i < 10; ++i ) VM_STACK[i] = argv[1][i];
```

After this, the function @00401610 and 0x00401540 is executed to perform the VM operations.

```
VM_RegA = 0;
VM_RegB = 0;
VM_SP = 9;
VM_IP = 0;
while ( VM_IP < VM_MAX ) function_ptrs[vm_codes[VM_IP]];
```

The function@ 00401570 initializes an array of function pointers used for handling VM instructions.

```
function_ptrs[0] = push_imm;
function_ptrs[1] = pop_imm;
function_ptrs[2] = add;
function_ptrs[3] = sub;
function_ptrs[4] = rotate_r;
function_ptrs[5] = rotate_l;
function_ptrs[6] = xor;
function_ptrs[7] = neg;
function_ptrs[8] = cmp;
function_ptrs[9] = cjmp;
function_ptrs[10] = ret;
function_ptrs[11] = push_reg;
function_ptrs[12] = pop_reg;
function_ptrs[13] = nop;
```

Function@00401080 does POP operation:

```
return VM_STACK[VM_SP--];
```

Function@00401000 does PUSH operation:

```
VM_SP++;
VM_STACK[VM_SP] = arg;
```

All the instructions use the VM_STACK, for performing read and write operations. Debugger can used to trace execution of VM, to figure out the program logic. Below is the gdb python code to perform the trace:

```
import gdb

class HitBreakpoint(gdb.Breakpoint):
    def __init__(self, loc, callback):
        super(HitBreakpoint, self).__init__(
            loc, gdb.BP_BREAKPOINT, internal=False
        )
        self.callback = callback

    def stop(self):
        self.callback()
        return False

function = ''
def add_trace(str):
    global function
    function += str+chr(0xa)

#def func00(): add_trace("push_imm")
#def func01(): add_trace("pop_imm")
def func02(): add_trace("add")
def func03(): add_trace("sub")
def func04(): add_trace("rotate_r")
def func05(): add_trace("rotate_l")
def func06(): add_trace("xor")
def func07(): add_trace("neg")
def func08(): add_trace("cmp")
def func09(): add_trace("cjmp")
def func10(): add_trace("ret")
#def func11(): add_trace("push_reg")
#def func12(): add_trace("pop_reg")
def func13():
    global function
    add_trace("nop")
    f = open("trace",'w')
    f.write(function)
    f.close()

def push():
    dx = gdb.parse_and_eval("$edx") & 0xffff
    eax = gdb.parse_and_eval("$eax") & 0xffff
    # track write into input buffer
    if eax <= 9: add_trace("[stack : %d]" %(eax))
    add_trace("push %s  " %(dx))


def pop():
    cx = gdb.parse_and_eval("$ecx") & 0xffff
    eax = gdb.parse_and_eval("$eax") & 0xffff
    # track read from input buffer
    if eax <= 9: add_trace("[stack : %d]" %(eax))
    add_trace("pop %s   " %(cx))


#HitBreakpoint("*0x00401030", func00)
#HitBreakpoint("*0x004010C0", func01)
HitBreakpoint("*0x004010E0", func02)
```

```
HitBreakpoint("*0x00401130", func03)
HitBreakpoint("*0x00401180", func04)
HitBreakpoint("*0x004011F0", func05)
HitBreakpoint("*0x00401260", func06)
HitBreakpoint("*0x004012B0", func07)
HitBreakpoint("*0x00401300", func08)
HitBreakpoint("*0x00401360", func09)
HitBreakpoint("*0x004013C0", func10)
#HitBreakpoint("*0x004013D0", func11)
#HitBreakpoint("*0x00401480", func12)
HitBreakpoint("*0x00401520", func13)
HitBreakpoint("*0x0040101E", push)
HitBreakpoint("*0x00401093", pop)
```

```
E:\>gdb -q smokestack.exe
Reading symbols from smokestack.exe...(no debugging symbols found)...done.
(gdb) source solver.py
Breakpoint 1 at 0x4010e0
Breakpoint 2 at 0x401130
Breakpoint 3 at 0x401180
Breakpoint 4 at 0x4011f0
Breakpoint 5 at 0x401260
Breakpoint 6 at 0x4012b0
Breakpoint 7 at 0x401300
Breakpoint 8 at 0x401360
Breakpoint 9 at 0x4013c0
Breakpoint 10 at 0x401520
Breakpoint 11 at 0x40101e
Breakpoint 12 at 0x401093
(gdb) run ABCDEFGHIJ
[New Thread 6128.0x22d4]
```

The debugger script traces all instructions executed by VM. It also logs any access to VM_STACK[0-9], which is the buffer for user input. This is similar to taint analysis.

BYTE 9 Algorithm:

```
push 33
add
pop 33
[stack : 9]        <- Access on user input
pop 74
[stack : 9]
push 107
push 145
cmp                <- add(33, userinput[9]) == 145
```

The first access to any user buffer character should be analyzed to find the logic. In the above case, it is first trace of [stack : 9]

```
Key: 145 - 33 = p
```

BYTE 8 Algorithm:

Tracing with ABCDEFGHIp:

```
xor
[stack : 9]
pop 24
[stack : 8]
pop 73
[stack : 8]
push 81
[stack : 9]
push 84
cmp                <- xor(24, userinput[8]) == 84
```

```
Key: 24 ^ 84 = L
```

BYTE 7 Algorithm:

```
sub
[stack : 8]
pop 18
[stack : 7]
pop 72
[stack : 7]
push 54
[stack : 8]
push 93
cmp                <- sub(18, userinput[7]) == 93
```

```
Key: 93 + 18 = o
```

BYTE 6 Algorithm:

```
xor
[stack : 7]
pop 179
[stack : 6]
pop 71
[stack : 6]
push 244
[stack : 7]
push 249
cmp                <- xor(179, userinput[6]) == 249
```

```
Key: 179 ^ 249 = J
```

BYTE 5 Algorithm:

There is a repeating sequence of add and sub instruction, like an unrolled loop. The VM does not have a multiplication instruction, but uses addition to achieve the same. Analyzing the loop operation, the algorithm can be derived as:

```
~(userinput[5] + (userinput[5] * 3)) & 0xffff == 65143
```

```
Key: (~65143 & 0xffff)/4 = b
```

BYTE 4 Algorithm:

```
rotate_r
[stack : 5]
pop 3
[stack : 4]
pop 69
[stack : 4]
push 40968
[stack : 5]
push 140
add
[stack : 5]
pop 140
[stack : 4]
pop 40968
[stack : 4]
push 41108
[stack : 5]
push 24724
cmp                    -> rotate_r(3, userinput[4]) + 140 == 24724
```

```
>>> num = 24724 - 140
>>> shift = 3
Key: ((num >> (16 - shift)) | (num << shift)) & 0xffff = C
```

BYTE 3 Algorithm:

```
xor
[stack : 4]
pop 12
[stack : 3]
pop 68
[stack : 3]
push 72
[stack : 4]
push 116
cmp                    -> xor(12, userinput[3]) == 116
```

```
Key: 116 ^ 12 = x
```

BYTE 2 Algorithm:

```
rotate_l
[stack : 3]
pop 6
[stack : 2]
pop 67
[stack : 2]
push 4288
[stack : 3]
push 7616
cmp                    -> rotate_l(6, userinput[2]) == 7616

>>> num = 7616
>>> shift = 6
Key: ((num << (16 - shift)) | (num >> shift)) & 0xffff = w
```

BYTE 1 Algorithm:

```
add
[stack : 2]
pop 22
[stack : 1]
pop 66
[stack : 1]
push 88
[stack : 2]
push 14
sub
[stack : 2]
pop 14
[stack : 1]
pop 88
[stack : 1]
push 74
[stack : 2]
push 97
cmp                    -> 22 + userinput[1] - 14 == 97


Key: 97 + 14 - 22 = Y
```

BYTE 0 Algorithm:

```
(8492 - (7 * userinput[0])) ^ 458 == 8181
8492 - (7 * userinput[0])  = 8181 ^ 458
8492 - (7 * userinput[0])  = 7743
7 * userinput[0] = 749

Key: 749/7 = k
```

The key for the VM : **kYwxCbJoLp**

Flag: A_p0p_pu$H_&_a_Jmp@flare-on.com

FLARE-ON CHALLENGE #6

Running strings on the challenge binary reveals it's a python script compiled to exe.

```
$ strings -a ./khaki.exe | grep -i python
PYTHONSCRIPT
PYTHON27.DLL
PYTHON27.DLL
Could not load python dll
```

The pyc can be extracted from exe using unpy2exe

```
$ python unpy2exe.py -o khaki  khaki.exe

$ ls
C:\Python27\lib\site-packages\py2exe\boot_common.py.pyc  poc.py.pyc
```

The pyc file can be decompiled to python source using pycdc

```
$ pycdc poc.py.pyc > poc.py
```

The below solver will give the flag:

```
import hashlib
import sys
import string

tmp = '312a232f272e27313162322e372548'
win_msg = 'Wahoo, you guessed it with %d guesses\n'
stuffs = [67, 139, 119, 165, 232, 86, 207, 61, 79, 67, 45, 58, 230, 190, 181,
74, 65, 148, 71, 243, 246, 67, 142, 60, 61, 92, 58, 115, 240, 226, 171]

def isprintable(str):
    return all(c in string.printable for c in str)

for count in range(1, 101):
    msg = win_msg % count
    stuffer = hashlib.md5(msg + tmp).digest()

    flag = ''
    for x in range(len(stuffs)):
        flag += chr(stuffs[x] ^ ord(stuffer[x % len(stuffer)]))

    if isprintable(flag): sys.exit(flag)
```

Flag: 1mp0rt3d_pygu3ss3r@flare-on.com

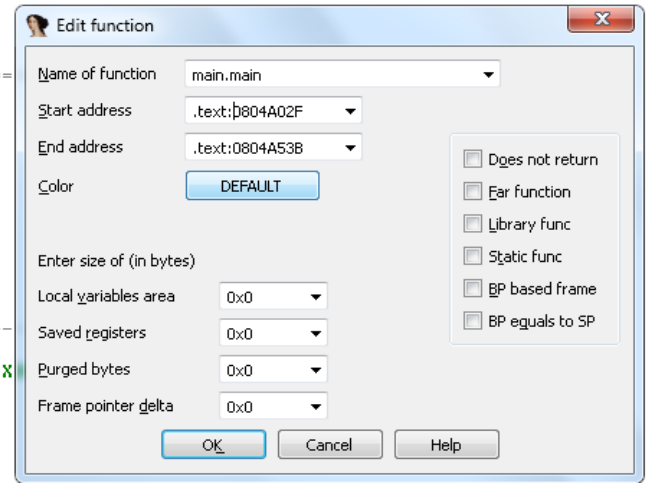
FLARE-ON CHALLENGE #7

This challenge involves reversing a Go ELF binary. This is how main_main function looks like:

```
.text:0804A019                  public main_main
.text:0804A019 main_main        proc near
.text:0804A019                  stc
.text:0804A01A                  lea     esi, [esi+0]
.text:0804A020                  jnb     short loc_804A02F
.text:0804A022                  push    0
.text:0804A024                  push    0ECh
.text:0804A029                  call    sub_804A810
.text:0804A02E                  retn
.text:0804A02F ; ---------------------------------------------------------------
.text:0804A02F
.text:0804A02F loc_804A02F:                             ; CODE XREF: main_main
.text:0804A02F                  push    ebp
.text:0804A030                  mov     ebp, esp
.text:0804A032                  push    edi
.text:0804A033                  push    esi
.text:0804A034                  push    ebx
.text:0804A035                  sub     esp, 0DCh
.text:0804A03B                  mov     edx, 1
.text:0804A040                  mov     ebx, ds:os_Args
```

Decompilation of main_main in IDA fails. To fix this issue, lets edit the function by changing the start address from 0x 0804A019 to 0x0804A02F

```
text:0804A018 sub_8049F6D       endp
text:0804A018
text:0804A019
text:0804A019 ; =============== S U B R O U T I N E =======
text:0804A019
text:0804A019
text:0804A019                   public main_main
text:0804A019 main_main         proc near
text:0804A019                   stc
text:0804A01A                   lea     esi, [esi+0]
text:0804A020                   jnb     short loc_804A02F
text:0804A022                   push    0
text:0804A024                   push    0ECh
text:0804A029                   call    sub_804A810
text:0804A02E                   retn
text:0804A02F ; ---------------------------------------------
text:0804A02F
text:0804A02F loc_804A02F:                              ; CODE X
text:0804A02F                   push    ebp
text:0804A030                   mov     ebp, esp
text:0804A032                   push    edi
text:0804A033                   push    esi
text:0804A034                   push    ebx
```



Now main_main can be decompiled to understandable code. The first check that looked interesting is

```
if ( *(arg - 40) == 30 )
    ret = sub_8049F6D(*(arg - 184), *(arg - 180));
if ( !ret ) goto FAIL;
```

```
.text:0804A0EB                  cmp     dword ptr [ebp-28h], 30
```

Setting up breakpoint in debugger shows it's a length check.

```
gdb-peda$ break *0x0804A0EB
Breakpoint 1 at 0x804a0eb

gdb-peda$ run ABCD
Breakpoint 1, 0x0804a0eb in main.main ()
gdb-peda$ x/d $ebp-0x28
0xb6008fd0: 4

gdb-peda$ run ABCDEFGH
Breakpoint 1, 0x0804a0eb in main.main ()
gdb-peda$ x/d $ebp-0x28
0xb6008fd0: 8
```

So the length of flag is 30 bytes. Function call to 0x8049F6D can be edited as mentioned above, resulting in decompilation of function@08049F80, which does the below check:

```
if ( strings_ContainsAny("abcdefghijklmnopqrstuvwxyz@-._1234", 34, *(a3 -
40), *(a3 - 36)) ^ 1 )
{
      *(a3 - 29) = 0;
      return *(a3 - 29);
}
```

This is the code transformation for Go's strings.ContainsAny, to perform charset validation on flag AAAAAAAAAAAAAAAA@flare-on.com. Following this validation, there is a loop:

```
for (counter = 0; input_size / 6 > counter; ++counter ) {
    __go_string_slice(&v58, user_input);
    compute_sha1(&sha1);
    __go_append(&v60, hash_array, length_of_hash_array, v39, sha1, v43, 1);
```

The inputs are sliced into 6 bytes, hashed with sha1 and appended to an array. The sha1 computation looks like below:

```
for ( i = 0; i <= 2; ++i ) {
    crypto_sha1_New(&v8, v6, v7);
```

It looks like hash operation is done thrice on the string. Next interesting loop in main.main function is

```
for ( counter = 0; length_of_hash_array > counter; ++counter ) {
    v8 = v79;
    if ( v79 ) {
      __go_receive(&::channel, channel, &out);
      target = get_target_byte(out);
      if ( length_of_hash_array <= counter || counter < 0 )
        __go_runtime_error(v10, v9);
      v8 = hash_array[counter] == target;
    }
    v79 = v8;
}
```

The comparison hash_array[counter] == target, needs further analysis. get_target_byte function takes a byte as input and returns a byte, which gets compared to the hash array. Since there is a go_receive, I looked for a go_send, leading to the below code:

```
sha1hashbyte = *hash_array;
v6 = __go_new(&unk_804CEC0, 12);
v45 = v6;
*v6 = channel;
*(v6 + 4) = sha1hashbyte;
*(v6 + 8) = length_of_hash_array;
__go_go(calc_index, v45);
```

calc_index leads to function@08049EFA

```
hash_array_byte = hash_array_arg;
for ( counter = 0; ; ++counter ) {
    if ( counter >= array_size ) break;
    v8 = (hash_array_byte + const_1CD) % const_1000;
    hash_array_byte = v8;
    __go_send_small(&channel, hash_byte, v8, v8 >> 32);
}
```

The first byte of hash is used to compute index values into an array@0804BB80 using get_target_byte

ie. some_array[calc_index(hash_array[0])] == hash_array[0]

IDA python script can be used to bruteforce this value:

```
import hashlib

address = 0x0804BB80
sz = 1024
hash_bytes = []

for va in range(address, address+sz):
    hash_bytes.append(Byte(va))

for byte in range(256):
    index = (byte + 0x1CD) % 0x1000
    target_byte = hash_bytes[index]
    if target_byte == byte: print hex(byte)
```

There are 3 possible values: [0x3c, 0x7a, 0xf9]. Since the suffix of flag is known, this hash value can be used find the right first byte, and hence retrieve the entire hash array

```
import hashlib

address = 0x0804BB80
sz = 4096
hash_bytes = []

for va in range(address, address+sz):
    hash_bytes.append(Byte(va))

flag = 'aaaaaaaaaaaaaaaaa@flare-on.com'
hash = []

for i in range(0, len(flag), 6):
    digest_string = flag[i:i+6]
    for _ in range(3):
        digest_string = hashlib.sha1(digest_string).digest()

    print flag[i:i+6], digest_string.encode('hex')

for pos in [0x3c, 0x7a, 0xf9]:
    digest = ''
    for _ in range(100):
        index = (pos + 0x1CD) % 0x1000
        hash_byte = hash_bytes[index]
        pos = index
        digest += chr(hash_byte)

    print digest.encode('hex')
```

aaaaaa c57770fcb819a543952f318ada9be72307902098
aaaaaa c57770fcb819a543952f318ada9be72307902098
aaaaa@ 57644bbb5c689dc183d0169bff05fd99a6eb3ef6
flare- 06046b721e18e20b841f497e257753b2314b866c
on.com cc720842d0884da08e26d9fccb24bc9c27bd254e
3cab2465e955b78e1dc84ab2aad1773641ef6c294a1bf8bd1e91f3593a6ccc9cc9b2d5682e62244f9e6061
a36250e1c47e69f0312db4e561528a1fb5**06046b721e18e20b841f497e257753b2314b866ccc720842d08**
84da08e26d9fccb24bc9c27bd254e
7afc01ff7c2ae6768ad7281b1025c7d64e9a905fef16ec2a43f5d840efdae1aaaabd7dc3b7670810a4f6f803
89125aad77e918db77a466e5ab7db10ffe140ae073bca6e0071d7d1c29a5fa1b73a99a064714505cf92f2fb
aeaac1059a5613a3928285b88
f91727f8892e34f2be1786fa115bc4ad621dd4ac92e4de8810744a70338e854adc7803e1eab7094138772f
47a05e778af70a1f1d5c8674b6fa63f4127cb25b5598ea410086a995d0c41770b46414599bee613d1a1a64
e064c31b9222f70566b9d6939c52

```
// gcc -std=c99 -O3 -o sol sol.c -lcrypto

#include <stdio.h>
#include <string.h>
#include <openssl/sha.h>
#include <stdlib.h>

#define chunk_size 6

const char *charset = "@_1234etaoinshrdlcumwfgypbvkjxqz";

const char *hashes[] = {
"\x3c\xab\x24\x65\xe9\x55\xb7\x8e\x1d\xc8\x4a\xb2\xaa\xd1\x77\x36\x41\xef\x6c\x29",
"\x4a\x1b\xf8\xbd\x1e\x91\xf3\x59\x3a\x6c\xcc\x9c\xc9\xb2\xd5\x68\x2e\x62\x24\x4f",
"\x9e\x60\x61\xa3\x62\x50\xe1\xc4\x7e\x69\xf0\x31\x2d\xb4\xe5\x61\x52\x8a\x1f\xb5"};

int main(int argc, char **argv)
{
    int len = strlen(charset);
    char computehash[chunk_size+1] = {0};
    char digest0[SHA_DIGEST_LENGTH];
    char digest1[SHA_DIGEST_LENGTH];
    char digest2[SHA_DIGEST_LENGTH];
    int hits = 0;

    for (int a = 0; a < len; a++) {
     for (int b = 0; b < len; b++) {
      for (int c = 0; c < len; c++) {
       for (int d = 0; d < len; d++) {
        for (int e = 0; e < len; e++) {
         for (int f = 0; f < len; f++) {
          computehash[0] = charset[a];
          computehash[1] = charset[b];
          computehash[2] = charset[c];
          computehash[3] = charset[d];
          computehash[4] = charset[e];
          computehash[5] = charset[f];
          SHA1(computehash, chunk_size, digest0);
          SHA1(digest0, SHA_DIGEST_LENGTH, digest1);
          SHA1(digest1, SHA_DIGEST_LENGTH, digest2);

          for (int i = 0; i < 3; i++) {
           if (memcmp(digest2, hashes[i], SHA_DIGEST_LENGTH) == 0) {
            printf("%s\n", computehash);
            hits++;
           }
           if (hits == 3) exit(0);
    }}}}}}}
    return 0;
}
```

Flag: h4sh3d_th3_h4sh3s@flare-on.com

FLARE-ON CHALLENGE #8

The binary provided for this challenge shows only a single function in IDA. Analyzing the XOR operation performed, this function seems to be of no use.

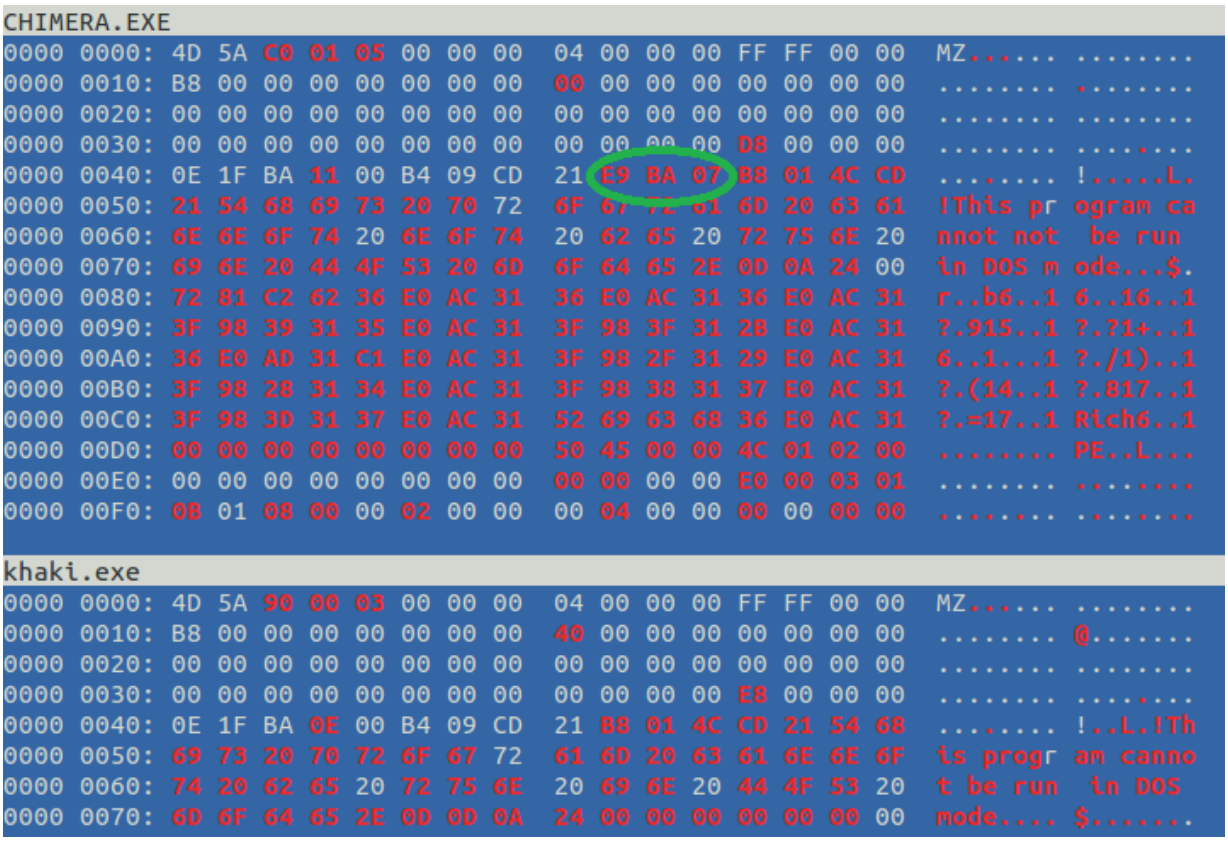
```
import hashlib

address = 0x004021D2
sz = 0x1A
hash_bytes = []

key = ''
for va in range(address, address+sz):
    key += chr((0x7A ^ Byte(va)) - (va - address))

print key
```

Wrong key: this is the wrong password

Text view of IDA reveals more code:

```
.text:004010AB                 db 42h
.text:004010AC                 dd 8663C046h, 0BF08AB2Ah, 19254C8Ch, 0ADB09231h, 67B6A214h
.text:004010AC                 dd 5FD839DDh, 0C25C7B3Fh, 752EF6B2h, 0CF94619Bh, 50986ACEh
.text:004010AC                 dd 45F05BF2h, 0EB380E30h, 7F666C3Bh, 88DF3D24h, 0F1B3B997h
.text:004010AC                 dd 1A9983CBh, 3B1EF0Dh, 7A9A9E55h, 0E836E010h, 0C132E4D3h
.text:004010AC                 dd 6BB70778h, 2CC970C7h, 6D3591A0h, 0F45E73FEh, 43DBD9A4h
.text:004010AC                 dd 0EE8DF569h, 0B5487D44h, 0A1024BDCh, 21A6D2E3h, 0D7A32F3Eh
.text:004010AC                 dd 0FB5A84BBh, 411C128Fh, 5976C528h, 633F79Ch, 0AF0B0A27h
.text:004010AC                 dd 0E94A1671h, 0E26F4F9Fh, 0E72BBE0Fh, 7953D556h, 9517642Dh
.text:004010AC                 dd 1D7CBDA7h, 65A59358h, 0EA1318F8h, 37F3E5BCh, 1EA89604h
.text:004010AC                 dd 51822901h, 8E1F683Ch, 22058ADAh, 87FA4972h, 0C66254A9h
.text:004010AC                 dd 0FDB409AAh, 85ACD1D6h, 9D3A4711h, 0CC1B4DE6h, 0FC238052h
.text:004010AC                 dd 607E8BEDh, 0BA576ECDh, 0C4CAAEDEh, 0D44E0C77h, 0B8E1C8D0h
.text:004010AC                 dd 819026F9h, 34h, 17h dup(0)
.text:00401200                 dd 380h dup(?)
.text:00401200 _text           ends
```

After many trials, I decided to analyze the PE header. Some additional bytes seems to be added to the PE header of CHIMERA.EXE. The diff of chimera.exe and khaki.exe shows bytes 0xE9, 0xBA, and 0x7 in the PE header. These changes happen at an offset 0x49, which is part of DOS stub.



Loading the binary as MS-DOS executable in IDA shows the below details:

```
seg000:0000                 public start
seg000:0000 start:
seg000:0000                 push    cs
seg000:0001                 pop     ds
seg000:0002                 assume ds:seg000
seg000:0002                 mov     dx, 11h
seg000:0005                 mov     ah, 9
seg000:0007                 int     21h             ; DOS - PRINT STRING
seg000:0007                                         ; DS:DX -> string
terminated by "$"
seg000:0009                 jmp     loc_107C6

seg000:07C6 loc_107C6:                              ; CODE XREF: seg000:0009
seg000:07C6                 mov     cx, 70h ; 'p'
seg000:07C9
seg000:07C9 loc_107C9:                              ; CODE XREF: seg000:07D2
seg000:07C9                 mov     bx, cx
seg000:07CB                 dec     bx
seg000:07CC                 add     bx, bx
seg000:07CE                 add     [bx+7D4h], cx
seg000:07D2                 loop    loc_107C9
```

This routine is used for decrypting the code for further execution

```
import math

address = 0x107D4

for cx in range(223, 0, -2):
    bx = cx - 1
    ea = address + bx
    patch = (Word(ea) + int(math.ceil(float(cx)/2))) & 0xffff
    PatchWord(ea, patch)
```

Decrypting and patching with above IDA python script, provides readable idb file for analysis. The validation algorithm looks like below:

```
#include <stdlib.h>
#include <stdint.h>
#include <err.h>
#include <bsd/string.h>

uint8_t table[] = {255, 21, 116, 32, 64, 0, 137, 236, 93, 195, 66, 70, 192,
99, 134, 42, 171, 8, 191, 140, 76, 37, 25, 49, 146, 176, 173, 20, 162, 182,
103, 221, 57, 216, 95, 63, 123, 92, 194, 178, 246, 46, 117, 155, 97, 148,
207, 206, 106, 152, 80, 242, 91, 240, 69, 48, 14, 56, 235, 59, 108, 102, 127,
36, 61, 223, 136, 151, 185, 179, 241, 203, 131, 153, 26, 13, 239, 177, 3, 85,
158, 154, 122, 16, 224, 54, 232, 211, 228, 50, 193, 120, 7, 183, 107, 199,
112, 201, 44, 160, 145, 53, 109, 254, 115, 94, 244, 164, 217, 219, 67, 105,
245, 141, 238, 68, 125, 72, 181, 220, 75, 2, 161, 227, 210, 166, 33, 62, 47,
163, 215, 187, 132, 90, 251, 143, 18, 28, 65, 40, 197, 118, 89, 156, 247, 51,
6, 39, 10, 11, 175, 113, 22, 74, 233, 159, 79, 111, 226, 15, 190, 43, 231,
86, 213, 83, 121, 45, 100, 23, 149, 167, 189, 124, 29, 88, 147, 165, 101,
248, 24, 19, 234, 188, 229, 243, 55, 4, 150, 168, 30, 1, 41, 130, 81, 60,
104, 31, 142, 218, 138, 5, 34, 114, 73, 250, 135, 169, 84, 98, 198, 170, 9,
180, 253, 214, 209, 172, 133, 17, 71, 58, 157, 230, 77, 27, 204, 82, 128, 35,
252, 237, 139, 126, 96, 205, 110, 87, 186, 222, 174, 202, 196, 119, 12, 78,
212, 208, 200, 225, 184, 249, 38, 144, 129, 52};

uint8_t target[] = {0x38, 0xE1, 0x4A, 0x1B, 0x0C, 0x1A, 0x46, 0x46, 0x0A,
0x96, 0x29, 0x73, 0x73, 0xA4, 0x69, 0x03, 0x00, 0x1B, 0xA8, 0xF8, 0xB8, 0x24,
0x16, 0xD6, 0x09, 0xCB};

uint8_t rol8(uint8_t x, uint8_t shift)
{
    return (x << shift) | (x >> (8 - shift));
}

int main(int argc, char **argv)
{
    uint8_t key[64];
    uint8_t key_len;
    uint16_t sum;

    if (argc != 2)
        errx(EXIT_FAILURE, "[!] Error! Provide <KEY> as argument...");

    strlcpy((char *)key, argv[1], sizeof(key));
    key_len = strlen((char *)key);
```

```
    for (int i = 0; i < key_len; i++) {

        uint8_t bx, index;
        index = key_len - i - 1;

        if (i == 0) bx = rol8(0x97, 3);
        else bx = rol8(key[index + 1], 3);

        key[index] ^= table[table[bx]];

    }

    for (int j = 0; j < key_len; j++) {

        uint8_t dl;

        if (j == 0) dl = 0xC5;
        else dl = key[j - 1];

        key[j] ^= dl;

    }

    sum = 0;
    for (int k = 0; k < sizeof(target); k++) {
        sum += (key[k] ^ target[k]);
    }

    if (sum == 0) warnx("[+] You have succeed.");
    else warnx("[!] You have fail. Please try harder.");

    return 0;
}
```

Below is the solver to generate the expected key:

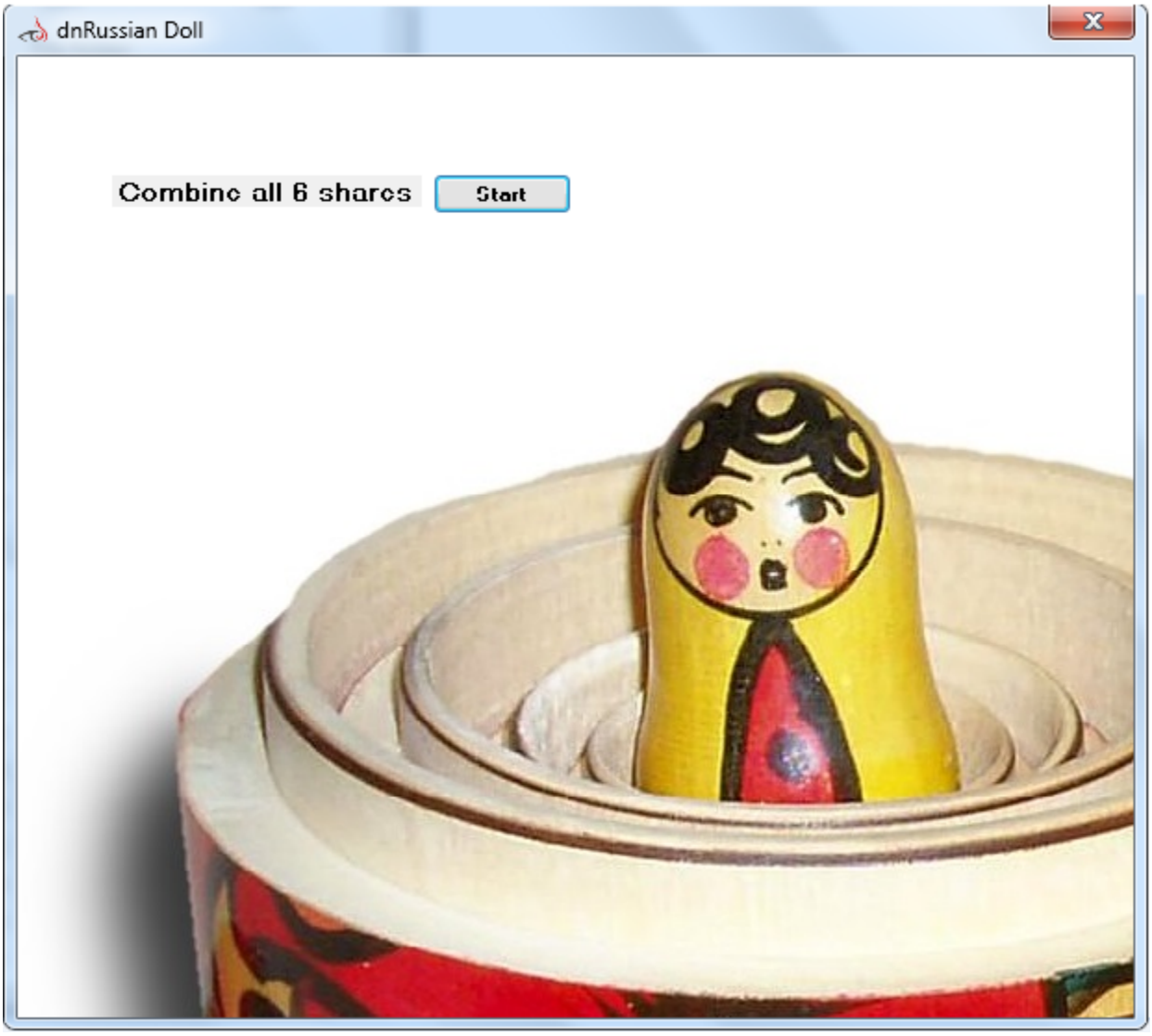
```
#include <stdlib.h>
#include <stdint.h>
#include <err.h>

uint8_t table[] = {255, 21, 116, 32, 64, 0, 137, 236, 93, 195, 66, 70, 192,
99, 134, 42, 171, 8, 191, 140, 76, 37, 25, 49, 146, 176, 173, 20, 162, 182,
103, 221, 57, 216, 95, 63, 123, 92, 194, 178, 246, 46, 117, 155, 97, 148,
207, 206, 106, 152, 80, 242, 91, 240, 69, 48, 14, 56, 235, 59, 108, 102, 127,
36, 61, 223, 136, 151, 185, 179, 241, 203, 131, 153, 26, 13, 239, 177, 3, 85,
158, 154, 122, 16, 224, 54, 232, 211, 228, 50, 193, 120, 7, 183, 107, 199,
112, 201, 44, 160, 145, 53, 109, 254, 115, 94, 244, 164, 217, 219, 67, 105,
245, 141, 238, 68, 125, 72, 181, 220, 75, 2, 161, 227, 210, 166, 33, 62, 47,
163, 215, 187, 132, 90, 251, 143, 18, 28, 65, 40, 197, 118, 89, 156, 247, 51,
6, 39, 10, 11, 175, 113, 22, 74, 233, 159, 79, 111, 226, 15, 190, 43, 231,
86, 213, 83, 121, 45, 100, 23, 149, 167, 189, 124, 29, 88, 147, 165, 101,
248, 24, 19, 234, 188, 229, 243, 55, 4, 150, 168, 30, 1, 41, 130, 81, 60,
104, 31, 142, 218, 138, 5, 34, 114, 73, 250, 135, 169, 84, 98, 198, 170, 9,
180, 253, 214, 209, 172, 133, 17, 71, 58, 157, 230, 77, 27, 204, 82, 128, 35,
```

```
252, 237, 139, 126, 96, 205, 110, 87, 186, 222, 174, 202, 196, 119, 12, 78,
212, 208, 200, 225, 184, 249, 38, 144, 129, 52};

uint8_t target[] = {0x38, 0xE1, 0x4A, 0x1B, 0x0C, 0x1A, 0x46, 0x46, 0x0A,
0x96, 0x29, 0x73, 0x73, 0xA4, 0x69, 0x03, 0x00, 0x1B, 0xA8, 0xF8, 0xB8, 0x24,
0x16, 0xD6, 0x09, 0xCB};

uint8_t temp_key[sizeof(target)];
uint8_t key[sizeof(target) + 1];

uint8_t rol8(uint8_t x, uint8_t shift)
{
    return (x << shift) | (x >> (8 - shift));
}

int main(int argc, char **argv)
{
    for (int i = 0; i < sizeof(target); i++) {

        if (i == 0)
            temp_key[i] = target[i] ^ 0xC5;
        else
            temp_key[i] = target[i] ^ target[i-1];
    }

    for (int j = sizeof(target) - 1; j >= 0; j--) {

        uint8_t index;

        if (j == 25)
            index = rol8(0x97, 3);
        else
            index = rol8(temp_key[j+1], 3);

        key[j] = temp_key[j] ^ table[table[index]];
    }

    warnx("KEY => %s", key);
    return 0;
}
```

Flag: retr0_hack1ng@flare-on.com

FLARE-ON CHALLENGE #9

A .NET executable GUI.exe is provided for this challenge. Running the binary pops the below GUI

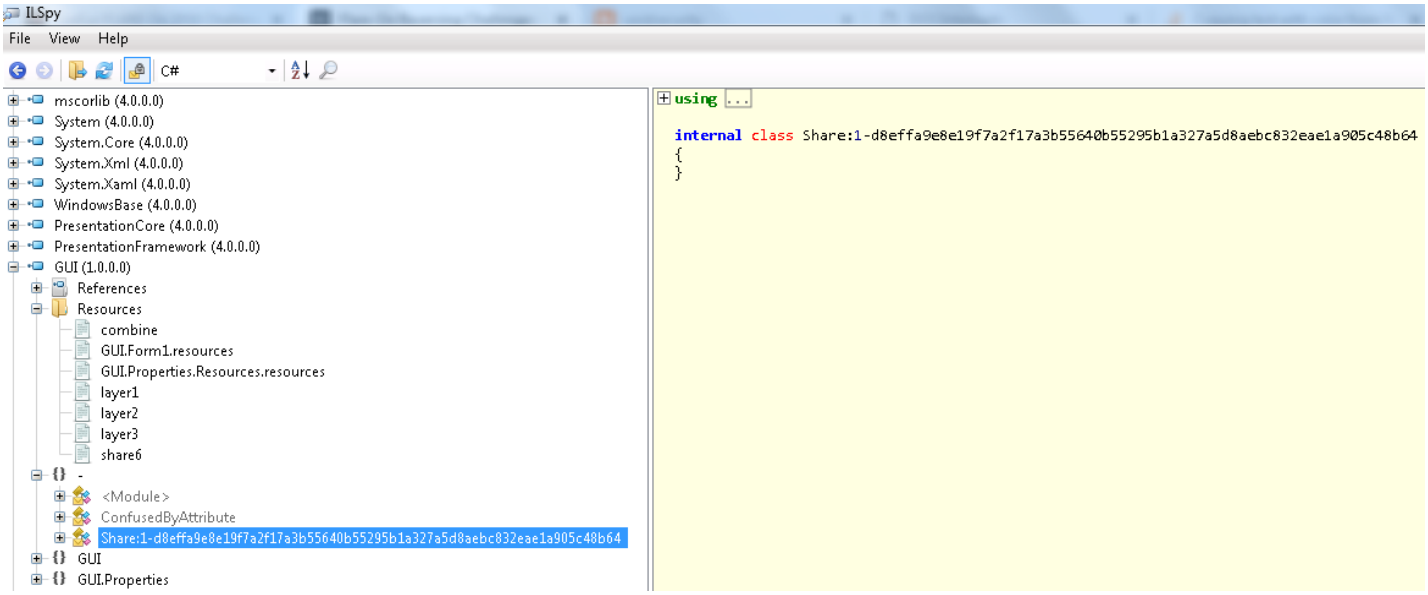


The GUI shows “Combine all 6 shares”. To analyze the binary further I used ILSpy.

```
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyCompany("FireEye")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCopyright("Copyright © FireEye 2016")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: AssemblyProduct("GUI")]
[assembly: AssemblyTitle("GUI")]
```

```
[assembly: AssemblyTrademark("")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: ComVisible(false)]
[assembly: Guid("90be6438-82f0-4236-9536-875456549adc")]
[module: ConfusedBy("ConfuserEx v1.0.0")]
```

It can be noticed that the code is obfuscated using ConfuserEx v1.0.0. Also the share-1 is visible in the obfuscated code:



Next let's deobfuscate the executable using NoFuserEx

E: \NoFuserEx-master\NoFuserEx\NoFuserEx\bin\Debug>NoFuserEx.exe -vv GUI.exe

. . .

[~] Saved successfully in {outputFile}

[~] Elapsed time:
{stopWatch.Elapsed.Minutes}:{stopWatch.Elapsed.Seconds}:{stopWatch.Elapsed.Milliseconds}

Loading the deobfuscated binary in ILSpy gives readable code

```
namespace GUI
{
    public class Form1 : Form
    {
        private string KEY_For_layer1 = "flareOnStartKey";
...
        private void button1_Click(object sender, EventArgs e)
        {
            byte[] buf = Form1.ReadResource("layer1");
```

```
            byte[] buffer = this.decryptBuffer(buf);
            byte[] rawAssembly = util.DecompressBuffer(buffer);
            Assembly assembly = Assembly.Load(rawAssembly);
            Type type = assembly.GetType("Layer1.Layer1");
            MethodInfo method = type.GetMethod("Start");
            bool flag = (bool)method.Invoke(null, new object[]
            {
                "no/-|-\\no/-|-\\no/-|-\\2/-|-\\shareShare:2-
f81ae6f5710cb1340f90cd80d9c33107a1469615bf299e6057dea7f4337f67a3"
            });
            if (flag)
            {
                MessageBox.Show("Thanks for playing!");
                return;
            }
            MessageBox.Show("Try again...");
        }

        private byte[] decryptBuffer(byte[] buf)
        {
            byte[] bytes = Encoding.UTF8.GetBytes(this.KEY_For_layer1);
            byte[] array = new byte[buf.Length];
            for (int i = 0; i < buf.Length; i++)
            {
                array[i] = (buf[i] ^ bytes[i % bytes.Length]);
            }
            return array;
        }
```

There are few things to notice here. First the share-2 can be found. Then the resource “combine” is not encrypted and it’s an executable. Extracting and running it gives the below output:

```
Shamir Secret Sharing Scheme - $Id$
Copyright 2005 B. Poettering, Win32 port by Alex.Popov@leggettwood.com

Combine shares using Shamir's Secret Sharing Scheme.

ssss-combine -t threshold [-x] [-q] [-Q] [-D]
```

So the challenge has something to do with Shamir’s Secret Sharing. “ssss-combine” source and binary is also available for download as “SSSS: Shamir's Secret Sharing Scheme (Windows port)”

Then layer1 resource is decrypted and loaded. To further analyze layer1, let’s extract the content using windbg:

```
ntdll!DbgBreakPoint:
76f2000c cc              int     3
0:004> .load
"C:\\WINDOWS\\Microsoft.NET\\Framework\\v4.0.30319\\sosex.dll"
0:004> !mbm *Assembly.Load*
Breakpoint set at System.Reflection.Assembly.LoadFrom(System.String) in
AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.LoadFrom(System.String,
System.Security.Policy.Evidence) in AppDomain 00552200.
```

```
Breakpoint set at System.Reflection.Assembly.LoadFrom(System.String,
System.Security.Policy.Evidence, Byte[],
System.Configuration.Assemblies.AssemblyHashAlgorithm) in AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.Load(System.String) in AppDomain
00552200.
Breakpoint set at System.Reflection.Assembly.Load(System.String,
System.Security.Policy.Evidence) in AppDomain 00552200.
Breakpoint set at
System.Reflection.Assembly.Load(System.Reflection.AssemblyName) in AppDomain
00552200.
Breakpoint set at
System.Reflection.Assembly.Load(System.Reflection.AssemblyName,
System.Security.Policy.Evidence) in AppDomain 00552200.
Breakpoint set at
System.Reflection.Assembly.LoadWithPartialNameHack(System.String, Boolean) in
AppDomain 00552200.
Breakpoint set at
System.Reflection.Assembly.LoadWithPartialName(System.String) in AppDomain
00552200.
Breakpoint set at
System.Reflection.Assembly.LoadWithPartialName(System.String,
System.Security.Policy.Evidence) in AppDomain 00552200.
Breakpoint set at
System.Reflection.Assembly.LoadWithPartialNameInternal(System.String,
System.Security.Policy.Evidence, System.Threading.StackCrawlMark ByRef) in
AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.Load(Byte[]) in AppDomain
00552200.
Breakpoint set at System.Reflection.Assembly.Load(Byte[], Byte[]) in
AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.Load(Byte[], Byte[],
System.Security.Policy.Evidence) in AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.LoadFile(System.String) in
AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.LoadFile(System.String,
System.Security.Policy.Evidence) in AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.LoadModule(System.String,
Byte[]) in AppDomain 00552200.
Breakpoint set at System.Reflection.Assembly.LoadModule(System.String,
Byte[], Byte[]) in AppDomain 00552200.
0:004> g
*** WARNING: Unable to verify checksum for
C:\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib\9f895c66454577eff9c77
442d0c84f71\mscorlib.ni.dll
Breakpoint 12 hit
eax=00000000 ebx=029dc838 ecx=03916e68 edx=00000001 esi=0287c464 edi=02880930
eip=50bf3e3d esp=0043ea70 ebp=0043ea74 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
mscorlib_ni+0x6b3e3d:
50bf3e3d c745fc01000000  mov     dword ptr [ebp-4],1
ss:002b:0043ea70=00000000

0:000> dd ecx
03916e68  507b398c 0001d200 00905a4d 00000003
03916e78  00000004 0000ffff 000000b8 00000000
03916e88  00000040 00000000 00000000 00000000
03916e98  00000000 00000000 00000000 00000000
```

```
03916ea8  00000000 00000080 0eba1f0e cd09b400
03916eb8  4c01b821 685421cd 70207369 72676f72
03916ec8  63206d61 6f6e6e61 65622074 6e757220
03916ed8  206e6920 20534f44 65646f6d 0a0d0d2e

0:000> .writemem C:\\Layer1.dll @ecx+8 L?poi(@ecx+4)
Writing 1d200
bytes..........................................................
```

This gives a DLL file. Then deobfuscate the DLL using NoFuserEx and decompile using ILSpy. The below code can be observed:

```
namespace Layer1
{
    public static class Layer1
    {
        [DllImport("kernel32.dll")]
        private static extern bool IsDebuggerPresent();

        public static bool Start(string config)
        {
            Config config2 = Config.InitConfig(config);
            if (config2 == null)
            {
                return false;
            }
            if (!CPUDetection.CheckCPUCount(config2.CPUCount))
            {
                return false;
            }
            if (config2.DebugDetection && Layer1.IsDebuggerPresent())
            {
                return false;
            }
            bool result;
            try
            {
                string key = Layer1.getKey();
                byte[] bytesToBeDecrypted = util.ReadResource("layer2");
                byte[] buffer = util.AES_Decrypt(bytesToBeDecrypted,
Encoding.UTF8.GetBytes(key));
                byte[] rawAssembly = util.DecompressBuffer(buffer);
                Assembly assembly = Assembly.Load(rawAssembly);
                Type type = assembly.GetType("Layer2.Layer2");
                MethodInfo method = type.GetMethod("Start");
                bool flag = (bool)method.Invoke(null, new object[]
                {
                    "Nothing useful to see here!"
                });
                result = flag;
            }
            catch (Exception)
            {
                result = false;
            }
```

```
            return result;
        }

        public static string getKey()
        {
            string[] directories = Directory.GetDirectories(".", "*",
SearchOption.TopDirectoryOnly);
            for (int i = 0; i < directories.Length; i++)
            {
                string text = directories[i];
                string text2 = text.Substring(2);
                MD5 mD = MD5.Create();
                byte[] bytes = Encoding.UTF8.GetBytes(text2);
                byte[] inArray = mD.ComputeHash(bytes);
                string text3 = Convert.ToBase64String(inArray);
                if (text3.CompareTo(StringUtils.a1224) == 0)
                {
                    return "flare-" + text2;
                }
                Console.WriteLine(text3);
            }
            return "flare-layer1-key";
        }
    }
}


// Layer1.StringUtils
public static string a1224 = "UtYSc3XYLz4wCCfrR5ssZQ==";
```

The Layer1.dll uses a few anti-debugging techniques, then decrypts and loads layer2. The key is derived using getKey(). getKey() searches the current directory for a directory named ‘sharing’

```
>>> "UtYSc3XYLz4wCCfrR5ssZQ==".decode('base64').encode('hex')
'52d6127375d82f3e300827eb479b2c65'
```

52d6127375d82f3e300827eb479b2c65 => “sharing”

Create the folder and continue the execution in windbg.

```
0:000> g
(2b58.1b78): CLR exception - code e0434f4d (first chance)
ModLoad: 62f20000 62f37000   C:\Windows\SysWOW64\bcrypt.dll
ModLoad: 71310000 71327000   C:\Windows\SysWOW64\CRYPTSP.dll
ModLoad: 712d0000 7130b000   C:\Windows\SysWOW64\rsaenh.dll
Breakpoint 12 hit
eax=00000000 ebx=0043e54c ecx=08263108 edx=00000001 esi=081a3040 edi=00000002
eip=50bf3e3d esp=0043e4ec ebp=0043e4f0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
mscorlib_ni+0x6b3e3d:
50bf3e3d c745fc01000000  mov     dword ptr [ebp-4],1
ss:002b:0043e4ec=00000000
0:000> dd ecx
08263108  507b398c 00026a00 00905a4d 00000003
08263118  00000004 0000ffff 000000b8 00000000
```

```
08263128  00000040 00000000 00000000 00000000
08263138  00000000 00000000 00000000 00000000
08263148  00000000 00000080 0eba1f0e cd09b400
08263158  4c01b821 685421cd 70207369 72676f72
08263168  63206d61 6f6e6e61 65622074 6e757220
08263178  206e6920 20534f44 65646f6d 0a0d0d2e

0:000> .writemem C:\\Layer2.dll @ecx+8 L?poi(@ecx+4)
Writing 26a00
bytes.....................................................
```

Again deobfuscate the DLL using NoFuserEx and decompile using ILSpy. Share-3 is immediately visible in the code:

```
            Block_1:
            return result;
            IL_13D:
            return "Share:3-
523cb5c21996113beae6550ea06f5a71983efcac186e36b23c030c86363ad294";
```

Share-4 is found as part of defined strings:

```
                    case 39u:
                        StringUtils.a704 = "Share:4-
04b58fbd216f71a31c9ff79b22f258831e3e12512c2ae7d8287c8fe64aed54cd";
                        StringUtils.a705 = "Jo+zGnhcDL/kTPlPnj/6MA==";
                        StringUtils.a706 = "r5VHoDfG0STsFiY05JBx2w==";
                        arg_15_0 = (num * 4089424959u ^ 3475654726u);
                        continue;
```

Layer2 also has anti-VM techniques checked using IsVideoCardFromEmulator(). It further gets the key to decrypt layer3 using Layer2.getKey()

```
// Layer2.Layer2
public static bool Start(string config)
{
    if (Layer2.IsVideoCardFromEmulator())
    {
        return false;
    }
    bool result;
    try
    {
        string key = Layer2.getKey();
        byte[] bytesToBeDecrypted = util.ReadResource("layer3");
```

getKey() looks like below:

```
// Layer2.Layer2
public static string getKey()
{
    string[] subKeyNames = Registry.CurrentUser.GetSubKeyNames();
    string result;
```

```
    while (true)
    {
. . .
            case 2u:
            {
                string text;
                arg_15_0 = (((text.CompareTo(StringUtils.a650) == 0) ?
2785585759u : 2681437764u) ^ num * 2156941825u);
                continue;
            }
. . .
```

StringUtils.a650 is defined as:

```
namespace Layer2
{
    internal class StringUtils
    {
. . .
                    case 486u:
                        StringUtils.a650 = "Xr4ilOzQ4PCOq3aQ0qbuaQ==";
                        arg_15_0 = (num * 3638050265u ^ 3200741439u);
                        continue;
```

```
>>> "Xr4ilOzQ4PCOq3aQ0qbuaQ==".decode('base64').encode('hex')
'5ebe2294ecd0e0f08eab7690d2a6ee69'
```

5ebe2294ecd0e0f08eab7690d2a6ee69 => secret

Hence create a registry key “secret” in HKEY_CURRENT_USER and continue the execution to extract Layer3.dll

```
0:000> g
(2b58.1b78): CLR exception - code e0434f4d (first chance)
ModLoad: 632d0000 633d6000
C:\Windows\assembly\NativeImages_v2.0.50727_32\System.Management\00c2b464e52d
4e82c04d61592a12a89d\System.Management.ni.dll
ModLoad: 70670000 7067e000   C:\Windows\SysWOW64\RpcRtRemote.dll
ModLoad: 75af0000 75b73000   C:\Windows\syswow64\CLBCatQ.DLL
ModLoad: 67d40000 67d57000   C:\Windows\SysWOW64\wbem\wmiutils.dll
ModLoad: 5ea70000 5eacc000   C:\Windows\SysWOW64\wbemcomn.dll
ModLoad: 74b50000 74b85000   C:\Windows\syswow64\WS2_32.dll
ModLoad: 75ae0000 75ae6000   C:\Windows\syswow64\NSI.dll
ModLoad: 6fa80000 6fa8a000   C:\Windows\SysWOW64\wbem\wbemprox.dll
ModLoad: 66dd0000 66dee000
C:\Windows\Microsoft.NET\Framework\v2.0.50727\wminet_utils.dll
(2b58.2114): Unknown exception - code 800706b5 (first chance)
ModLoad: 602a0000 602af000   C:\Windows\SysWOW64\wbem\wbemsvc.dll
ModLoad: 5e9d0000 5ea66000   C:\Windows\SysWOW64\wbem\fastprox.dll
ModLoad: 5fbb0000 5fbc8000   C:\Windows\SysWOW64\NTDSAPI.dll
Breakpoint 12 hit
eax=00000000 ebx=00000000 ecx=07b410e8 edx=00000004 esi=32fd681b edi=00000000
eip=50bf3e3d esp=0043df58 ebp=0043df5c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
mscorlib_ni+0x6b3e3d:
```

```
50bf3e3d c745fc01000000  mov     dword ptr [ebp-4],1
ss:002b:0043df58=00000000
0:000> dd ecx
07b410e8  507b398c 00028000 00905a4d 00000003
07b410f8  00000004 0000ffff 000000b8 00000000
07b41108  00000040 00000000 00000000 00000000
07b41118  00000000 00000000 00000000 00000000
07b41128  00000000 00000080 0eba1f0e cd09b400
07b41138  4c01b821 685421cd 70207369 72676f72
07b41148  63206d61 6f6e6e61 65622074 6e757220
07b41158  206e6920 20534f44 65646f6d 0a0d0d2e

0:000> .writemem C:\\Layer3.dll @ecx+8 L?poi(@ecx+4)
Writing 28000
bytes.....................................................................
........
```

NoFuserEx failed to deobfuscate the Layer3.dll, throwing the below error:
Unhandled Exception: System.BadImageFormatException: .NET MetaData RVA is 0

Next I used UnConfuserEx v1.0, which successfully unpack the binary. However the strings were still not recognized.

```
case 2u:
{
    byte[] bytesToBeDecrypted = util.ReadResource(<Module>.<string>(2921636399u));
    arg_10_0 = (num * 3538037274u ^ 2949823500u);
    continue;
}
case 4u:
{
    byte[] bytes2;
    File.WriteAllBytes(<Module>.<string>(2663114732u), bytes2);
    ProcessStartInfo processStartInfo = new ProcessStartInfo(<Module>.<string>(1143424475u));
    arg_10_0 = (num * 1327553900u ^ 1386100579u);
    continue;
}
```

To further clean up the binary for analysis let's use NoFuserEx

E:\NoFuserEx-master\NoFuserEx\NoFuserEx\bin\Debug>NoFuserEx.exe -vv Layer3-unpacked.dll

. . .

Detecting ConfuserEx version...

[!] ConfuserEx doesn't detected. Use de4dot.

In order to clean the binary force the deobfuscation using "--force-deob for force the deobfuscation" option.

```
E:\NoFuserEx-master\NoFuserEx\NoFuserEx\bin\Debug>NoFuserEx.exe --force-deob
Layer3-unpacked.dll
. . .
[~] Forced deobfuscation.
```

```
        [+] Decrypted:      {decryptedConstants}/{detectedConstants}
constant(s).

        [+] Removed:        Anti-dumper protection.
        [+] Removed:        Anti-debugger protection.

[~] Saved successfully in {outputFile}
```

Now we get a readable code to work with

```
case 2u:
{
      byte[] bytesToBeDecrypted = util.ReadResource("share6");
      arg_10_0 = (num * 3538037274u ^ 2949823500u);
      continue;
}
```

Layer3 decrypts the resource share6 using the key obtained from getKey()

```
      public static string getKey()
        {
            SelectQuery query = new SelectQuery("Win32_UserAccount");
            ManagementObjectSearcher managementObjectSearcher = new
ManagementObjectSearcher(query);
            ManagementObjectCollection.ManagementObjectEnumerator enumerator
= managementObjectSearcher.Get().GetEnumerator();
. . .
                        case 0u:
                        {
                            string text;
                            arg_2F_0 = (((text.CompareTo(StringUtils.a66) ==
0) ? 206694793u : 1391732593u) ^ num * 2790437578u);
                            continue;
                        }
. . .
                        case 8u:
                        {
                            ManagementObject managementObject =
(ManagementObject)enumerator.Current;
                            string text2 = (string)managementObject["Name"];
                            arg_2F_0 = 3508415771u;
                            continue;
                        }
```

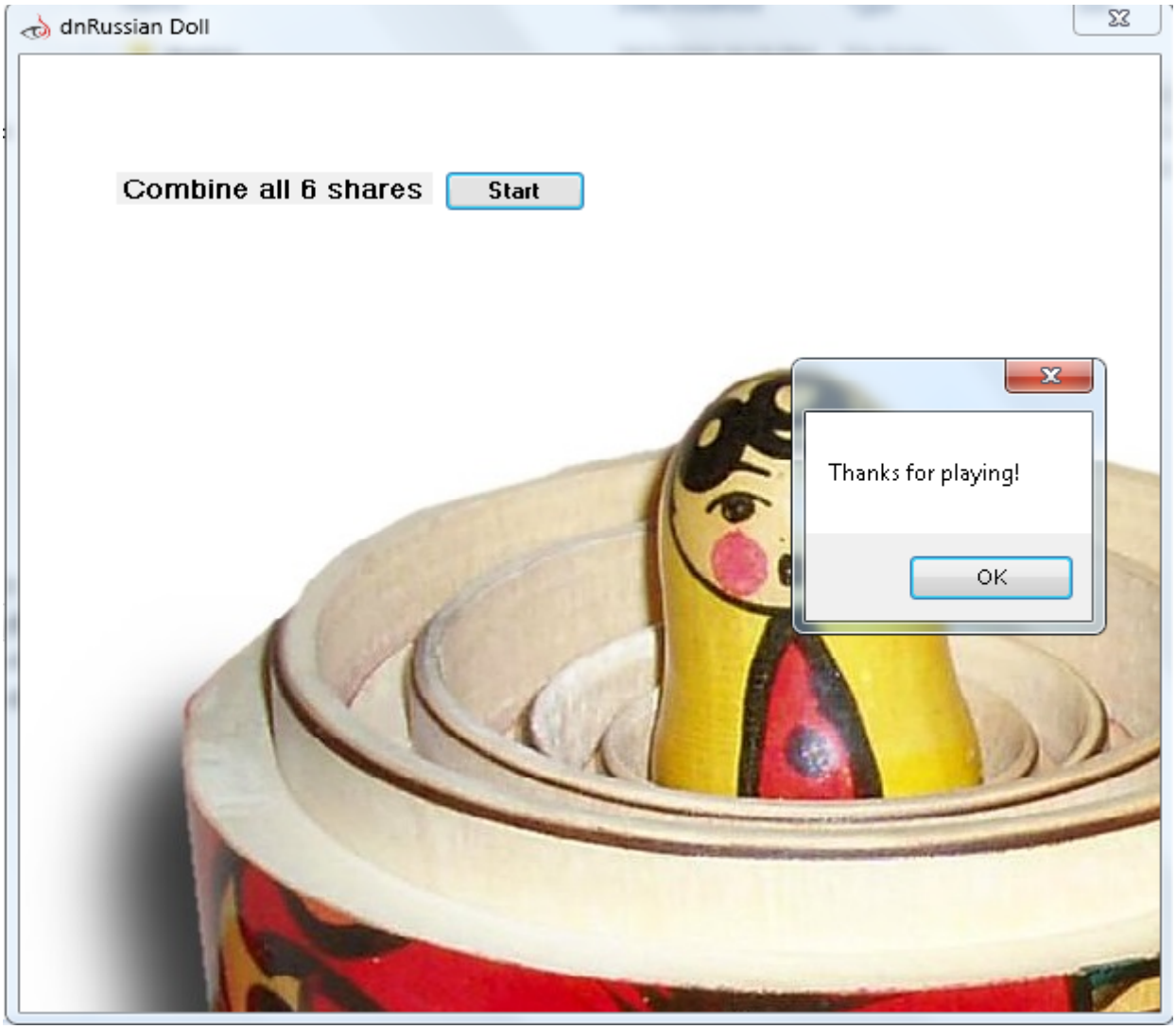
The string compared a66 is defined as below:

```
                    case 762u:
                        StringUtils.a66 = "KTUxM5ElLBtBBAdJXNCW/g==";
                        arg_15_0 = (num * 3331003351u ^ 3414829663u);
                        continue;
```

```
>>> "KTUxM5ElLBtBBAdJXNCW/g==".decode('base64').encode('hex')
'2935313391252c1b410407495cd096fe'
```

2935313391252c1b410407495cd096fe => shamir

Layer3 code checks if a user named shamir exists. So create the user and continue the execution. This gives a message:



Further, 2 new files are written to the directory – ‘share6-decoded.png’ and ‘ssss-combine.exe’ which we extracted earlier. share6-decoded.png has one of the shares

Share
6-a003fcf2955ced997c8741a6473d7e3f3540a8235b5bac16d3913a3892215f0a

Share-5 is found as part of defined strings:

```
                    case 288u:
                        StringUtils.a801 = "tXVnS3QHCdFw65DGlgAXXg==";
                        StringUtils.a802 = "Share:5-
5888733744329f95467930d20d701781f26b4c3605fe74eefa6ca152b450a5d3";
                        StringUtils.a803 = "CT4A6il843cvjOy8VkdY4g==";
                        arg_15_0 = (num * 2858195988u ^ 1606397890u);
                        continue;
```

Combine all the 6 shares to get the flag:

E:\>ssss-combine.exe -t 6
Shamir Secret Sharing Scheme - Id
Copyright 2005 B. Poettering, Win32 port by Alex.Popov@leggettwood.com

Enter 6 shares separated by newlines:
Share [1/6]: 1-d8effa9e8e19f7a2f17a3b55640b55295b1a327a5d8aebc832eae1a905c48b64
Share [2/6]: 2-f81ae6f5710cb1340f90cd80d9c33107a1469615bf299e6057dea7f4337f67a3
Share [3/6]: 3-523cb5c21996113beae6550ea06f5a71983efcac186e36b23c030c86363ad294
Share [4/6]: 4-04b58fbd216f71a31c9ff79b22f258831e3e12512c2ae7d8287c8fe64aed54cd
Share [5/6]: 5-5888733744329f95467930d20d701781f26b4c3605fe74eefa6ca152b450a5d3
Share [6/6]: 6-a003fcf2955ced997c8741a6473d7e3f3540a8235b5bac16d3913a3892215f0a
Resulting secret: **Shamir_1s_C0nfused@flare-on.com**

FLARE-ON CHALLENGE #10

A pcap file flava.pcap is provided for the challenge. Opening it in wireshark some interesting traffic can be noticed as below:

9233	10.11.106.81:18089	text/html	499 bytes	flareon_found_in_milpitas.html
9469	10.11.106.81:18089	img/png	207 kB	flareon.png
9592	10.14.56.20:18089	text/html	68 kB	\
9596	10.14.56.20:18089	application/json	160 bytes	i_knew_you_were_trouble
9601	10.14.56.20:18089	text/plain	2288 bytes	i_knew_you_were_trouble
16163	10.14.56.20:18089	application/x-shockwave-flash	6871 kB	will_tom_hiddleston_be_taylor_swifts_last_song_of_her_career.meh

During the access to 10.14.56.20:18089, an obfuscated JavaScript is executed, followed by access to i_know_you_were_trouble and will_tom_hiddleston_be_taylor_swifts_last_song_of_her_career.meh flash file. The flash file requires a key.

First let's deobfuscate the JavaScript using jsnice.org to get a readable code for analysis. After deobfuscation, I used chrome's developer console to debug the JavaScript. All debugging was carried out in a windows host. Console.log could be added to the obfuscated code to print required information. The interesting part of the code is the below piece:

```
/** @type {(boolean|function (number, number): undefined)} */
AYKbsHqKVlYHn = !!HLgTS ? true : function(near, far) {
    . . .
    text = v5Z16(data["replace"](/\s/g, relationOptions), PNfNaU);
    console.log(text);
    . . .
    try {
        if (FiAwn == 1) {
            /** @type {Function} */
            var expression = new Function(text);
            expression();
```

The large base64 string in the page is decoded and decrypted into a JavaScript to further execute it. The essential part of v5Z16() is below:

```
  var codeSegments = ogv(opt_e);
  var l = clx(v);        // get decryption key

  p = 0;                 // set p = 0 for right decryption
  /** @type {number} */
  var i = 0;
  for (;i < codeSegments.length;i++) {
    var n = codeSegments[i];
    . . .
    if (l.length > 0) {
      n ^= l[p];
    }
    . . .
    args[i] = n;
  }
```

The 'p' value used as index into the decryption key is set to a value based on certain validations. However, p = 0 turned out to be the right value. The function call UIgvkFSsu() to change the value of 'p' can simply be commented out. After this the layer 2 JavaScript is decrypted for analysis.

```
function k() {
. . .
 try {
 m();
 var a = l();
 console.log(a);                // add logging
 var b = Function(a);
 b();
 } catch(zzzzz) {}
}

try {
 k();
} catch (z) {}
```

The function l() generates the next level JavaScript. It simply performs a base64 decode. The value window['UoqK1Yl'] should be set to base64 index table

```
function l() {
 j();
 var a = "dmFy...";          // truncated
 var b,g,f,e,c,d=0;
 var h='';
 var x=window['UoqK1Yl'];
 x='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=';

 do
 b = x.indexOf(a.charAt(d++)),
 g = d < a.length ? x.indexOf(a.charAt(d++)) : 64,
 e = d < a.length ? x.indexOf(a.charAt(d++)) : 64,
 c = d < a.length ? x.indexOf(a.charAt(d++)) : 64,
 f = b << 18 | g << 12 | e << 6 | c,

 b = f >> 16 & 255,
 g = f >> 8 & 255,
 f &= 255,

 h = 64 == e ? h + String.fromCharCode(b) : 64 == c ? h +
String.fromCharCode(b, g) : h + String.fromCharCode(b, g, f);
 while (d < a.length);

 return h;
}
```

The last level JavaScript is heavily obfuscated. The essential part of the code is the Diffie-Hellman Key Exchange handshake.

```
function Il1Iya() {
  var L = Il1Illl1I1l.Il1Iu;

  this.L = new L(Il1Ixa(), Il1Ibbso + Il1Ibbsn);      // g
  console.log(this.L.toString());
  this.ic = new L(Il1Ixa(), Il1Ibbso * 3 - Il1Ibbsi); // secret
  console.log(this.ic.toString());
  this.ha = new L(Il1Ita(), Il1Ibbso * Il1Ibbsi + Il1Ibbsi * Il1Ibbsi); // p
  console.log(this.ha.toString());
```

```
  this.Jb = this.L.Il1IX(this.ic, this.ha);           // A = g ^ secret mod p
  console.log(this.Jb.toString());
}

Il1Iza = function() {
    . . .
  //var u = Il1Iqa();   // anti-analysis
  var u = 0;

  try {

    /* Diffie Hellman parameters */
    var e = {
      g : pair.L[Il1Io](Il1Ibbsi * 8),                // generator g
      A : pair.Jb[Il1Io](" "[Il1Ip](0) / Il1Ibbsi),   // A = g ^ secret mod p
      p : pair.ha[Il1Io](("2"[Il1Ip](0) - 2) / 3),    // modulus p
    };

    e = base64encode(rc4("flareon_is_so_cute", JSON.stringify(e)));

    var listener = new XMLHttpRequest;
    listener.open("POST", "http://10.14.56.20:18089/i knew you were trouble",
true);
    listener.setRequestHeader("Content-type", "application/json; charset=utf-
8");
    listener.send(e);
    . . .
}
```

Once the DH parameters are generated, they are converted to JSON string in the below format:

```
{"g":"91a812d65f3fea132099f2825312edbb","A":"16f2c65920ebeae43aabb5c9af923953
","p":"3a3d4c3d7e2a5d4c5c4d5b7e1c5a3c3e"}
```

Then encrypted using RC4 and encoded using base64 for transmission to 10.14.56.20:18089. The original value transmitted can be found in the pcap file. The response text is used as below:

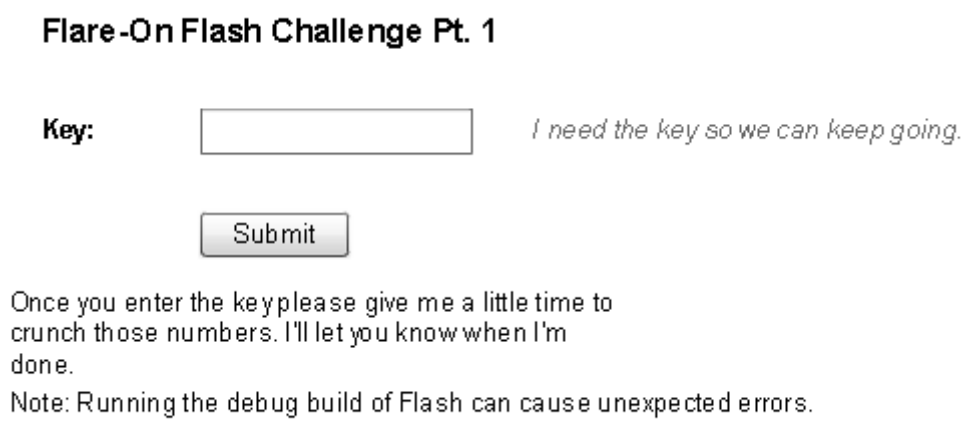
```
        var data = JSON.parse(rc4("how_can_you_not_like_flareon",
base64decode(unescape(response))));
        console.log(data);
        var result = new Il1Illl1I1l.Il1Iu(data.B, 16);
        var key = result.Il1IX(pair.ic, pair.ha); // (secret, p)
        var payload = rc4(key.toString(16), data.fffff);

        if (u < 1) {
          eval(payload);
        }
```

The response text can also be found in the pcap file which has base64 encoded data. After base64 decoding and decrypting using RC4, data.B has the DH parameter and data.fffff has the encrypted data. Then key for decrypting data.fffff is derived using DH parameter. Note that the DH parameters are not checked for primality. Solving the DLP reveals the shared secret value

24c9de545f04e923ac5ec3bcfe82711f. Use this to decrypt data.fffff, which returns the below code:

```
var txt = '<object type="application/x-shockwave-flash"
data="http://10.14.56.20:18089/will_tom_hiddleston_be_taylor_swifts_last_song
 of_her_career.meh" allowScriptAccess=always width="1" height="1">';
 txt = txt + '</object>';
 try {
 document.getElementById("T6Fe3dfg").innerHTML = txt;
 } catch (e) {};
 . . .
 alert("Cool, seems like you're ready to roll!\n\n\n\nThe key for part2 of
the challenge is:\n'HEAPISMYHOME'\n\n\n\nYou will need this key to proceed
with the flash challenge, which is also included in this pcap.\n\nGood luck!
May the force be with you!");
```

HEAPISMYHOME is the key for the flash challenge



The flash file can be analyzed using JPEXS flash decompiler. Decompiling the binary reveals the below code in FlareonFlashLoader.as:

```
protected function submitButton_clickHandler(param1:MouseEvent) : void
{
   this.a = new Run();
   this.a.d3cryp7AndL0ad(this.key.text);
   if(this.a.loaded)
   {
      Alert.show("Now what?");
      Alert.show("That was underwhelming....");
      Alert.show("....");
      Alert.show("Unlocked!");
   }
   else
   {
      Alert.show("Wrong!");
   }
}
```

The provided key HEAPISMYHOME is passed to d3cryp7AndL0ad() function. It further calls pr0udB3lly() which generates the decryption key and performs RC4 decryption. Here binaryData Run_challenge1 is decrypted and loaded using loadBytes()

```
     public function d3cryp7AndL0ad(param1:String) : void
      {
         var _loc2_:Loader = new Loader();
         var _loc3_:LoaderContext = new
LoaderContext(false,ApplicationDomain.currentDomain,null);
         var _loc4_:Class = Run_challenge1;
         var _loc5_:ByteArray = pr0udB3lly(ByteArray(new _loc4_()),param1);
         var _loc6_:String = "1172ca0ede560b08d97b270400347ede";
         if(MD5.hashBytes(_loc5_) == _loc6_)
         {
            this.loaded = true;
            _loc2_.loadBytes(_loc5_);
         }
```

pr0udB3lly() concats HEAPISMYHOME with first 16 bytes of binary data, then performs MD5 to generate the key “97d2c8220a2c2bf4c5bb16b045625222”. The first 16 bytes of binary data is skipped from decryption as it is key material.

```
      public static function pr0udB3lly(param1:ByteArray, param2:String) :
ByteArray
      {
         var _loc5_:* = undefined;
         var _loc3_:ByteArray = new ByteArray();
         var _loc4_:ByteArray = new ByteArray();
         _loc3_.writeBytes(param1,16,param1.length - 16);
         _loc4_.writeUTFBytes(param2);
         _loc4_.writeBytes(param1,0,16);
         _loc5_ = MD5.hashBytes(_loc4_);
```

The second stage flash file is protect using secureSWF

The file can be deobfuscated using JPEXS P-code deobfuscation option to get a readable code. Using “Rename invalid identifiers” also improves the readability. The essential part of the code is below:

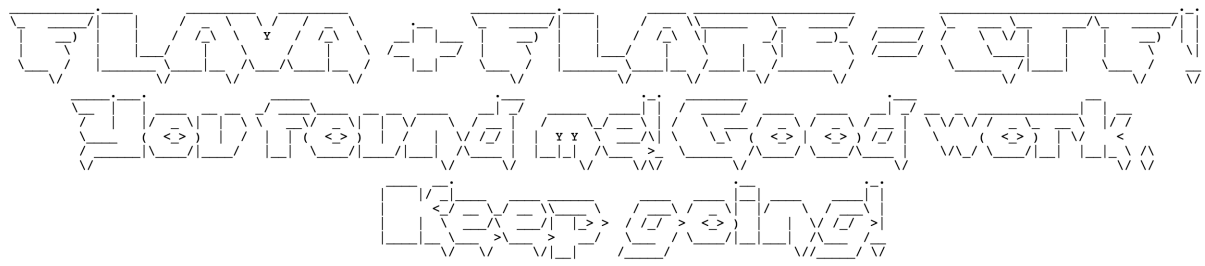
```
         var _loc1_:Object = this.root.loaderInfo.parameters;
         if(_loc1_.flare == "On")
         {
            _loc2_ = new Loader();
```

```
            _loc3_ = new
LoaderContext(false,this.root.loaderInfo.applicationDomain,null);
            this.var_1 = new Array();
            this.var_4 = new ByteArray();
            this.var_39 = _loc1_.x;
            this.var_5 = _loc1_.y;
            _loc4_ = new class_3();
            _loc5_ = "600aa47f484cbd43ecd37de5cf111b10";
```

The flash file expects three parameters flare, x and y. flare has to be set to “On”, but x and y are unknown. There are 52 binary blobs in the file. The parameter x is used for decrypting them except for 2 of them including blob Int3lIns1de_t3stImgurvnUziJP

```
            counter = 0;
            while(counter < this.payload_array.length)
            {
               this.payload_array[counter] = keygen_and_rc4
(this.payload_array[counter],this.param_x);
               counter++;
            }
```

Int3lIns1de_t3stImgurvnUziJP actually refers to imgur.com to fetch the file vnUziJP. This turned out to be an image file:



The file doesn’t look interesting, however, the image size of the PNG file and the binary blob Int3lIns1de_t3stImgurvnUziJP are the same. So the binary blog is probably encrypted data of the image. XOR the image and the blob to get the key. There is another blob which was not pushed into the array for decryption using keygen_and_rc4(). Decrypting 31_--__-.-__--__.bin with the recovered key gives the x and y parameters. X is “1BR0K3NCRYPT0FTW” and y is a list of number pairs e.g. 47:14546, 46:1617 . . . This key “x” can be used to decrypt other 50 blobs. The parameter y is used to build the next stage flash file from the decrypted blobs as below:

```
      public final function build_flash_from_y() : void
      {
         var _loc3_:Array = null;
         var _loc1_:Array = this.param_y.split(",");
         var _loc2_:uint = 0;
         while(_loc2_ < _loc1_.length)
         {
```

```
            _loc3_ = _loc1_[_loc2_].split(":");
            this.flash[_loc2_] = this.payload_array[_loc3_[0]][_loc3_[1]];
            _loc2_++;
        }
    }
```

The order of payload_array i.e. blobs can be found in class_3.as

```
    public function class_3()
    {
        this.var_44 = class_36;
        this.var_38 = class_37;
        this.var_35 = class_38;
        this.var_10 = class_39;
        this.var_46 = class_45;
        this.var_8 = class_47;
        this.var_33 = class_49;
        this.var_50 = class_51;
        this.var_21 = class_53;
        this.var_24 = class_54;
        this.var_47 = class_40;
        this.var_45 = class_44;
        this.var_51 = class_43;
        this.var_22 = class_42;
        this.var_7 = class_41;
        this.var_16 = class_52;
        this.var_53 = class_50;
        this.var_12 = class_48;
        this.var_41 = class_46;
        this.var_26 = class_55;
        this.var_31 = class_6;
        this.var_15 = class_5;
        this.var_58 = class_10;
        this.var_37 = class_9;
        this.var_54 = class_8;
        this.var_48 = class_7;
        this.var_36 = class_14;
        this.var_30 = class_13;
        this.var_28 = class_12;
        this.var_14 = class_11;
        this.var_11 = class_17;
        this.var_18 = class_16;
        this.var_55 = class_15;
        this.var_43 = class_21;
        this.var_52 = class_20;
        this.var_17 = class_19;
        this.var_42 = class_18;
        this.var_49 = class_24;
        this.var_40 = class_23;
        this.var_29 = class_22;
        this.var_32 = class_29;
        this.var_13 = class_28;
        this.var_27 = class_27;
        this.var_19 = class_26;
        this.var_23 = class_33;
        this.var_20 = class_32;
```

```
            this.var_6 = class_31;
            this.var_57 = class_30;
            this.var_56 = class_35;
            this.var_9 = class_34;
            this.var_34 = class_25;
            this.var_25 = Int3lIns1de_t3stImgurvnUziJP;
            super();
        }
```

The 3rd stage flash script can be unpacked using a simple python script:

```
from Crypto.Cipher import ARC4
import hashlib

x = "1BR0K3NCRYPT0FTW"
y = open('flashvars').read().strip()
indexes = y.split(",")

flash_codes = []

flash_codes.append(open("44_package_2.class_36.bin").read())
...
flash_codes.append(open("33_package_2.class_34.bin").read())

for i in range(len(flash_codes)):
    key_bytes = flash_codes[i][:16]
    swf_bytes = flash_codes[i][16:]

    cipher = ARC4.new(hashlib.md5(x + key_bytes).hexdigest())
    flash_codes[i] = cipher.decrypt(swf_bytes)

swf = ''
for index in indexes:
    a, b = index.split(':')
    swf += flash_codes[int(a)][int(b)]

print hashlib.md5(swf).hexdigest()
```

The stage-3 flash file can be deobfuscated using JPEXS P-code deobfuscation. After deobfuscation, a long series of if conditions with calls to writeByte() can be seen. The source code was around ~1500 lines. Most of the if conditions are dead code.

```
package
{
   import flash.display.Sprite;
   import flash.utils.ByteArray;

   public final class ยง52142312314123423632234ยง extends Sprite
   {


      public function ยง52142312314123423632234ยง()
      {
```

```
            super();
            var _loc1_:int = 0;
            var _loc2_:ByteArray = new ByteArray();
            _loc1_ = 1;
            if(!_loc1_)
            {
               _loc2_.writeByte(120);
            }
...
```

I rewrote part of the decompiled action script to C to get the flag:

```
#include <stdio.h>

struct write {
    int (*writeByte)(int);
};

int main(int argc, char **argv)
{
    int _loc1_  = 0;
    struct write _loc2_;
    _loc2_.writeByte = &putchar;

    _loc1_ = 1;
    if(!_loc1_)
    {
        _loc2_.writeByte(120);
    }
...
```

renorobert@ubuntu:~$ gcc -Wall -O3 -o solver solver.c
renorobert@ubuntu:~$ ./solver
angl3rcan7ev3nprim3@flare-on.com