The Flare-On Challenge 2015

Reno Robert
v0ids3curity.blogspot.in
@renorobertr

FLARE-ON CHALLENGE #1

Flare-On_start_2015.exe is a 64 bit executable. Running the binary, extracted a 32 bit executable i_am_happy_you_are_to_playing_the_flareon_challenge.exe

The binary does XOR operation on the user input using a single byte key, and compares it with a hardcoded array.

```
.text:0040104D loop:
.text:0040104D                mov     al, user_input[ecx]
.text:00401053                xor     al, 7Dh         ; key
.text:00401055                cmp     al, encrypted_flag[ecx]
.text:0040105B                jnz     short fail
.text:0040105D                inc     ecx
.text:0040105E                cmp     ecx, 24         ; length
.text:00401061                jl      short loop

.data:00402140 encrypted_flag  db 1Fh, 8, 13h, 13h, 4, 22h, 0Eh, 11h, 4Dh,
0Dh, 18h, 3Dh
.data:00402140                 db 1Bh, 11h, 1Ch, 0Fh, 18h, 50h, 12h, 13h,
53h, 1Eh, 12h, 10h
```

The length of the hardcoded array is 24 bytes, and the key is 0x7D. Below is the solution:

```
encrypted_flag = [0x1F, 0x08, 0x13, 0x13, 0x04, 0x22, 0x0E, 0x11,
                  0x4D, 0x0D, 0x18, 0x3D, 0x1B, 0x11, 0x1C, 0x0F,
                  0x18, 0x50, 0x12, 0x13, 0x53, 0x1E, 0x12, 0x10]

flag = ''
for c in encrypted_flag:
    flag += chr(c ^ 0x7D)

print flag
```

Flag: bunny_sl0pe@flare-on.com

FLARE-ON CHALLENGE #2

Get Your FLARE On Now webinar has a good level of information regarding the validation algorithm in challenge #2. Commented disassembly of the algorithm is given below:

```
.text:004010DF          public start
.text:004010DF start    proc near
.text:004010DF          call    crackme      ; address of flag pushed into
stack
.text:004010DF start    endp
.text:004010DF
.text:004010E4 flag     db 0AFh, 0AAh, 0ADh, 0EBh, 0AEh, 0AAh, 0ECh, 0A4h,
0BAh
.text:004010E4          db 0AFh, 0AEh, 0AAh, 8Ah, 0C0h, 0A7h, 0B0h, 0BCh, 9Ah
.text:004010E4          db 0BAh, 0A5h, 0A5h, 0BAh, 0AFh, 0B8h, 9Dh, 0B8h,
0F9h
.text:004010E4          db 0AEh, 9Dh, 0ABh, 0B4h, 0BCh, 0B6h, 0B3h, 90h, 9Ah,
0A8h

.text:00401000 crackme  proc near
.text:00401000          pop     eax             ; fetch address of flag
.text:00401001          push    ebp
.text:00401002          mov     ebp, esp
.text:00401004          sub     esp, 10h
.text:00401007          mov     [ebp-10h], eax  ; write address of flag

.text:00401054          push    dword ptr [ebp-4]    ; input_sz
.text:00401057          push    offset input         ; input
.text:0040105C          push    dword ptr [ebp-10h]  ; flag
.text:0040105F          call    validate
```

Validation Routine:

```
.text:0040108C          xor     ebx, ebx
.text:0040108E          mov     ecx, 37         ; length
.text:00401093          cmp     [ebp+input_sz], ecx
.text:00401096          jl      short fail
.text:00401098          mov     esi, [ebp+input]
.text:0040109B          mov     edi, [ebp+email]
.text:0040109E          lea     edi, [edi+ecx-1] ; fetch array in reverse
.text:004010A2
.text:004010A2 loop:                            ; CODE XREF: validate+4Fj
.text:004010A2          mov     dx, bx
.text:004010A5          and     dx, 3           ; DX = BX & 3
.text:004010A9          mov     ax, 1C7h        ; AH = 1, AL = 0xC7
.text:004010AD          push    eax
.text:004010AE          sahf                    ; CF = AH = 1
.text:004010AF          lodsb                   ; AL = BYTE PTR [ESI]
.text:004010B0          pushf
.text:004010B1          xor     al, [esp+4]     ; AL = AL ^ 0xC7
.text:004010B5          xchg    cl, dl
.text:004010B7          rol     ah, cl          ; AH = ROTATE_LEFT [AH = 1],
DX
.text:004010B9          popf
.text:004010BA          adc     al, ah          ; AL = AL + AH + [CF = 1]
.text:004010BC          xchg    cl, dl
```

```
.text:004010BE          xor     edx, edx
.text:004010C0          and     eax, 0FFh
.text:004010C5          add     bx, ax             ; BX += AL
.text:004010C8          scasb                      ; ZF = AL == BYTE PTR [EDI];
EDI++
.text:004010C9          cmovnz  cx, dx             ; IF ZF == 0: CX = [DX = 0]
.text:004010CD          pop     eax
.text:004010CE          jecxz   short fail         ; jump on ECX == 0
.text:004010D0          sub     edi, 2
.text:004010D3          loop    loop
```

Solver:

```
email = [0xAF, 0xAA, 0xAD, 0xEB, 0xAE, 0xAA, 0xEC, 0xA4, 0xBA,
         0xAF, 0xAE, 0xAA, 0x8A, 0xC0, 0xA7, 0xB0, 0xBC, 0x9A,
         0xBA, 0xA5, 0xA5, 0xBA, 0xAF, 0xB8, 0x9D, 0xB8, 0xF9,
         0xAE, 0x9D, 0xAB, 0xB4, 0xBC, 0xB6, 0xB3, 0x90, 0x9A, 0xA8]

BX = 0
DX = 0
key = ''

for i in reversed(range(len(email))):
    AH = ((1 << DX) | (1 >> (8 - DX))) # ROL AH, DX
    AL = (email[i] - AH - 1) ^ 0xC7
    BX = BX + email[i]
    DX = BX & 3
    key += chr(AL)

print key
```

Flag: a_Little_b1t_harder_plez@flare-on.com

FLARE-ON CHALLENGE #3

Here, dumping the strings, gives valuable information regarding the binary. This could be a python script, compiled into an executable:

```
$ strings -a elfie.exe | grep -i python
Error loading Python DLL: %s (error code %d)
Error detected starting Python VM.
PYTHONHOME
PYTHONPATH
Cannot GetProcAddress for Py_SetPythonHome
Py_SetPythonHome
bpython27.dll
bpyside-python2.7.dll
bshiboken-python2.7.dll
python27.dll
```

This is using Python 2.7, so use PyInstaller Extractor to get the python source from executable:

```
$ python2 pyinstxtractor.py elfie.exe
Successfully extracted Pyinstaller archive : elfie.exe

$ ls
bz2.pyd                      msvcm90.dll              pyi_archive     pyside-
python2.7.dll  QtGui4.dll              struct
elfie                        msvcp90.dll              _pyi_bootstrap
PySide.QtCore.pyd     select.pyd              unicodedata.pyd
elfie.exe.manifest           msvcr90.dll              pyi_carchive
PySide.QtGui.pyd      shiboken-python2.7.dll
_hashlib.pyd                 out00-PYZ.pyz            pyi_importers
python27.dll          _socket.pyd
Microsoft.VC90.CRT.manifest  out00-PYZ.pyz_extracted  pyi_os_path
QtCore4.dll           _ssl.pyd
```

One of the extracted file is **elfie.** Elfie constructs base64 encode code and finally executes it as exec(base64.b64decode(base64_encoded_code)). Now, dump the code as print(base64.b64decode(base64_encoded_code))

```
$ python elfie > elfie_b64_decode.py
```

At the tail of the decoded source file, the email is found, in reverse :

```
if (O0O0000000OO00O00OO0O000OO0OO000 ==
''.join((OO0O0O0OO00O0OO0OOO0O000O00OOO000 for
O00O0O00O0000O0O0O0OO00O0OO0OO000 in reversed('moc.no-
eralf@OOOOY.sev0000L.eiflE')))):
```

Flag: Elfie.L0000ves.YOOOO@flare-on.com

FLARE-ON CHALLENGE #4

Disassembly of youPecks.exe shows that it was packed with UPX.

```
UPX1:0040B440 start:
UPX1:0040B440                  pusha
UPX1:0040B441                  mov     esi, offset dword_409000
UPX1:0040B446                  lea     edi, [esi-8000h]
UPX1:0040B44C                  push    edi
```

$ upx -d youPecks.exe

```
       File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     25088 <-     12800   51.02%    win32/pe     youPecks.exe
```

Unpacked 1 file.

The program skips major part of main routine, due to an unsolvable comparison, leading to dead code.

```
.text:00401442 push    offset Str      ; "5"
.text:00401447 call    ds:atoi
.text:0040144D add     esp, 4
.text:00401450 mov     esi, eax
.text:0040147C cmp     esi, 5
.text:0040147F jnz     short do_something
```

On patching JNZ to JZ, the execution continues into the do_something block. This turns out to be a wrong decision. Further, analyzing the binary with PEiD Krypto ANALyzer, shows the presence of hashing and base64 encoding:

```
MarkPosition(0x004051B8, 0, 0, 0, slotidx + 0, "BASE64 table");
MakeComm(PrevNotTail(0x004051B9), "BASE64 table\nBASE64 encoding (used e.g.
in e-mails - MIME)");
MarkPosition(0x004063E6, 0, 0, 0, slotidx + 1, "CryptCreateHash [Name]");
MakeComm(PrevNotTail(0x004063E7), "CryptCreateHash [Name]\nMicrosoft
CryptoAPI function name");
MarkPosition(0x004063D6, 0, 0, 0, slotidx + 2, "CryptHashData [Name]");
MakeComm(PrevNotTail(0x004063D7), "CryptHashData [Name]\nMicrosoft CryptoAPI
function name");
```

```
.text:00401314 push    8003h           ; Algid
.text:00401319 push    edx             ; hProv
.text:0040131A call    ds:CryptCreateHash
```

ALG_ID 0x8003 is CALG_MD5. The program takes only a single byte of input, so possible input values range from 0-255.

```
.text:00401563 push    eax             ; argv
.text:00401564 call    ds:atoi
.text:0040156A add     esp, 4
```

```
.text:0040156D lea     ecx, [esp+0F4h+md5_of_input]
.text:00401574 push    ecx              ; md5_of_user_input
.text:00401575 lea     edx, [esp+0F8h+pbData]
.text:0040157C push    edx              ; user_input
.text:0040157D mov     [esp+0FCh+pbData], al
.text:00401584 call    ComputeMD5
.text:00401589 lea     eax, [esp+0FCh+Time_sec]
```

Another input the program takes, is by using localtime64_s, which populates tm struct.

```
struct tm time;
_time64_t time_sec;
time_sec = _time64(0);
_localtime64_s(&time, &time_sec);
```

```
.text:004013E8 call    ds:_time64
.text:004013EE add     esp, 4
.text:004013F1 mov     [esp+0F4h+Time_sec], eax

.text:00401589 lea     eax, [esp+0FCh+Time_sec]
.text:0040158D push    eax              ; Time
.text:0040158E lea     ecx, [esp+100h+Tm]
.text:00401592 push    ecx              ; Tm
.text:00401593 call    ds:_localtime64_s
```

The binary performs several operations, including the decoding of hardcoded base64 strings. Then it compares MD5 of user supplied input, with another 16 byte computed array.

Out of all the elements in tm struct, only tm.tm_hour seems to be used by the program. The value of tm_hour ranges from 0 to 23.

```
.text:004015A0     mov     edi, [esp+0F4h+Tm.tm_hour]
```

Many other elements seems to be set to constant values.

```
.text:004036A6     mov     [esi+tm.tm_year], 0Fh
.text:004036AD     mov     [esi+tm.tm_mon], 0
.text:004036B4     mov     byte ptr [esi+tm.tm_sec], 0
```

So, as far as I understand, there exists a value between 0-255 whose md5 hash will equal, the value computed using all the base64 decode operations. But interestingly bruteforce of input 0-255 did not work.

To check the dependency of tm_hour and the other elements, I wrote a gdb-python script to modify the tm struct in memory, and observe the change in computation of target hash. It confirmed that only tm_hour is used in the computation of target hash.

```python
import gdb
import pprint
from random import randint

counter = 0
tm_struct_addr = 0
hashlist = []

def exit_handler(event):
    pprint.pprint(hashlist)
    gdb.execute("quit")

def callback_restart_prog():
    # Entry Point
    gdb.execute("set $eip = 0x00403A8A")

def callback_fetch_addr_tm():
    global tm_struct_addr
    tm_struct_addr = int(gdb.parse_and_eval("$ecx"))

def update_tm_element(address, value):
    gdb.execute("set *%s = %s" %
                (hex(address), hex(value)))

def callback_update_tm():
    global tm_struct_addr, counter

    #update_tm_element(tm_struct_addr, randint(0,60))        #tm_sec
    #update_tm_element(tm_struct_addr+4, randint(0,60))      #tm_min
    update_tm_element(tm_struct_addr+8, counter)             #tm_hour
    #update_tm_element(tm_struct_addr+12, randint(0,32))     #tm_mday
    #update_tm_element(tm_struct_addr+16, randint(0,12))     #tm_mon
    #update_tm_element(tm_struct_addr+20, randint(0,200))    #tm_year
    #update_tm_element(tm_struct_addr+24, randint(0,7))      #tm_wday
    #update_tm_element(tm_struct_addr+28, randint(0,366))    #tm_yday
    #update_tm_element(tm_struct_addr+32, randint(0,2))      #tm_isdst

    counter += 1
    if counter == 24: gdb.execute("clear *0x004025AF")

def callback_copy_hash():
    global counter
    calc_addr = gdb.parse_and_eval("$edi")
    inferior = gdb.selected_inferior()
    calc = inferior.read_memory(calc_addr, 16)
    calc = "%s" % (calc)
    hashlist.append(calc.encode('hex'))

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

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

```
OnBreakpoint("*0x00401592", callback_fetch_addr_tm)
OnBreakpoint("*0x00401599", callback_update_tm)
OnBreakpoint("*0x00401BAE", callback_copy_hash)
OnBreakpoint("*0x004025AF", callback_restart_prog)
gdb.events.exited.connect(exit_handler)
```

Different hashes generated for tm_hour values 0-23, can be collected by running the script using gdbmingw

(gdb) source check_tm_struct.py
Breakpoint 1 at 0x401592
Breakpoint 2 at 0x401599
Breakpoint 3 at 0x401bae
Breakpoint 4 at 0x4025af
(gdb) run 0
[New Thread 8156.0x1b28]
2 + 2 = 5
………….
2 + 2 = 5
[Inferior 1 (process 8156) exited normally]
['2bb21f445e273a2367f49b2ab7dc050a',
'bc0bda724d013f23108aad0c902845aa',
'34c226c24be2138e9500234769f46a55',
'1cccce9ea0106736a556f3f445eec501',
…………………………………………………………
'f4014a0fcd16a91b002a2c7089616791',
'576d521b3ee30c16dd452b9c7cd5bd7e',
'1e9f2ef8ac3ea64ad934d59c0d710baa']

None of extracted hash values matched hash values of 0-255. I was stuck here for quite some time. Later, on brute forcing the one byte input with the original unpacked binary, dumped the flag.

```
import subprocess

for c in range(256):
    msg = subprocess.check_output(['youPecks.exe', str(c)])
    print msg.strip()
```

>python brute.py
2 + 2 = 4
2 + 2 = 4
…………
2 + 2 = 4
Uhr1thm3tic@flare-on.com

There seemed to be an issue with unpacking, so the unpacked binary was not working the right way. Opening the UPX packed executable with vicOlly, the last jump statement looks like 0x0040B621 JMP youPecks.00403A8A. So set breakpoint at 0x0040B621. When execution hits the breakpoint, single step and the actual entry point looks like

```
00403A8A    CALL youPecks.00403F23
```

Then use OllyDump plugin in vicOlly, to dump the binary with current EIP as entry point. Run the gdb-python script on this unpacked binary

```
(gdb) source  check_tm_struct.py
Breakpoint 1 at 0x401592
Breakpoint 2 at 0x401599
Breakpoint 3 at 0x401bae
Breakpoint 4 at 0x4025af
(gdb) run 0

2 + 2 = 4
Uhr1thm3tic@flare-on.com
2 + 2 = 4
............
2 + 2 = 4
2 + 2 = 4

 ['93b885adfe0da089cdf634904fd59f71',
 '55a54008ad1ba589aa210d2629c1df41',
 '9e688c58a5487b8eaf69c9e1005ad0bf',
 '8666683506aacd900bbd5a74ac4edf68',
 'ec7f7e7bb43742ce868145f71d37b53c',
 ………………………………………………………….
 'ffe51d3e7d8297237588704eeddc6ab2',
 '15f41a2e96bae341dde485bb0e78f485',
 'f5a7e477cd3042b49a9085d62307cd28',
 'bf6d6c819ec975b043aec502167c3d15',
 '84ff14fa45be3ca4739e7c027717a541']
```

These are hash values are of integers 0-23. From this it’s evident that, the binary checks if md5(input) == md5(tm_hour) to print the flag. So if time is 10:00 PM, the valid input for binary is 22.

Later on I came to know that, there are anti-unpacking techniques against upx -d as mentioned by @corkami

FLARE-ON CHALLENGE #5

This challenge had two files - challenge.pcap and sender.exe. The function at 0x00401100 in sender.exe, reads a key.txt file. Then the key data is transformed by function at 0x00401250

```
.rdata:00410EA8 flarebearstare  db 'flarebearstare',0

void transform(char *key, unsigned int size) {
    for(unsigned int i = 0; i < size; i++) {
        key[i] += flarebearstare[i % 0xE];
    }
}
```

Following this, comes the below calculations:

```
if (key_sz % 3)
    enc_sz = 3 * (key_sz / 3) - key_sz + 3;     // 3 - (key_sz % 3)
else
    enc_sz = 0;

pad_enc_sz = 4 * ((key_sz + enc_sz) / 3);
```

This is size computation for base64 encoding. Quick analysis of the function 0x004012A0 and information from the previous size computation, shows a possibility of base64 encoding:

```
.rdata:00410EB8 charseta        xmmword 'ponmlkjihgfedcba'
.rdata:00410EC8 charsetb        xmmword 'FEDCBAzyxwvutsrq'
.rdata:00410ED8 charsetc        xmmword 'VUTSRQPONMLKJIHG'
.rdata:00410EE8 charsetd        xmmword '/+9876543210ZYXW'

.text:004013D8 padding:
.text:004013D8                 cmp     ecx, edi
.text:004013DA                 jnb     short success
.text:004013DC                 mov     byte ptr [esi+ecx], '='
.text:004013E0                 inc     edx
.text:004013E1                 inc     ecx
.text:004013E2                 cmp     edx, 3
.text:004013E5                 jl      short padding
```

After base64 encoding, the transformed key bytes is, sent in chunks of 4 bytes to a HTTP server listening on port 80. Let's analyze the pcap file and filter 'http' traffic. It is noticed that, string UDYs1D7bNmdE1o3g5ms1V6RrYCVvODJF1DpxKTxAJ9xuZW== is sent in chunks of 4 bytes. As per the analysis, I wrote the inverse algorithm:

```
import base64

enc = 'UDYs1D7bNmdE1o3g5ms1V6RrYCVvODJF1DpxKTxAJ9xuZW=='
dec = base64.b64decode(enc)

secret = 'flarebearstare'
key = ''

for i in range(len(dec)):
    key += chr((ord(dec[i]) - ord(secret[i%0xE])) & 0xFF)
print key
```

But this did not give a printable text. So the base64 encode seemed to be some modified version. Let us analyze further. The length of key is found to be 34 bytes.

Create a key file with content 'AAAAAAAAAAAAAAAAAAAAA@flare-on.com' and debug the binary.

(gdb) break *0x00401203
Breakpoint 1 at 0x401203
(gdb) x/s $ecx
0x87bf50: "P62IS6AJPQkZTlwIS6ANRAkZPQoMODJF1DpxKTxAJ9xuZW=="

The suffix of the encoded data, matches 'ODJF1DpxKTxAJ9xuZW==' from pcap file.

```
import base64

secret = 'flarebearstare'
key = 'AAAAAAAAAAAAAAAAAAAAA@flare-on.com'
transform = ''

for i in range(len(key)):
    transform += chr(ord(key[i]) + ord(secret[i%0xE]))
print base64.b64encode(transform)
```

This gives the output p62is6ajpqKztLWis6anraKzpqOmodjf1dPXktXaj9XUzw==. The difference between normal base64, and the one in binary, is the case inversion. Therefore, modify the solver, to invert cases, before performing base64 decode, to get the flag.

```
import base64

enc = 'UDYs1D7bNmdE1o3g5ms1V6RrYCVvODJF1DpxKTxAJ9xuZW=='
enc = ''.join(c.lower() if c.isupper() else c.upper() for c in enc)
dec = base64.b64decode(enc)

secret = 'flarebearstare'
key = ''

for i in range(len(dec)):
    key += chr(ord(dec[i]) - ord(secret[i%0xE]))
print key
```

Flag: Sp1cy_7_layer_OSI_dip@flare-on.com

FLARE-ON CHALLENGE #6

This challenge involves reversing an APK file. Extracting the APK file, an ARM shared object file libvalidate.so could be found in lib/armeabi. Analyzing the file with IDA, there is an interesting function *Java_com_flareon_flare_ValidateActivity_validate*. Decompile the function using IDA, and rewrite it to an understandable form:

```
j_j_memset(exponent, 0, 0x1B28);
j_j_memcpy(ptrs_to_exponent, &ptrs, 92);

user_key = user_arg;
if (user_arg && j_j_strlen(user_arg) <= 46) {
    cmp_counter = 0;
    is_valid = true;

    for (i = 0; i < j_j_strlen(user_key); i += 2) {
        j_j_memset(exponent, 0, 0x1B28);
        key_word = 0;
        if (user_key[i]) {
            key_word = user_key[i];
            if (user_key[i+1]) {
               key_word = 0x7E7E
               >= ((user_key[i] << 8) | user_key[i + 1]) ?
                   (user_key[i] << 8) | user_key[i + 1] : 0;
            }
        }
        for (j = 0; j != 0xD94; j++) {
            factor = factor_array[j];      // type uint16_t
            while ( !(key_word % factor & 0xFFFF)) {
                ++*&exponent[j * 2];       // type uint16_t
                key_word = key_word / factor & 0xFFFF;
                if (key_word == 1)
                    goto do_comparison;
            }
        }
    do_comparison:
        if (j_j_memcmp(*&ptrs_to_exponent[4 * cmp_counter], exponent, 0xD94))
            is_valid = false;
        else
            ++cmp_counter;
    }

    if (cmp_counter == 23 && is_valid)
        msg = "That's it!";
    else
        msg = "No";
}
```

The length of user input is 46 bytes. For each processed WORD, the factors are found using division technique. If WORD % factor == 0, the number is considered a valid factor, and another array which stores the exponent of a factor is incremented. Finally, an array of exponents are compared using memcmp. So, hard coded array values of factor_array and ptrs_to_exponent can be used, to find the valid key.

`pow(factor[j=0], exponent[j=0]) * … * pow(factor[j=n], exponent[j=n])`

Using the above equation, valid key could be reversed. I mmaped the library, to read data from it. Below is the complete solution:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>

// gcc -m32 -o solve solve.c --std=c99 -lm

// pointers from libvalidate.so as in IDA
uint32_t ptrs[23] = {
0x2A5D0,0x28AA8,0x26F80,0x25458,
0x23930,0x21E08,0x202E0,0x1E7B8,
0x1CC90,0x1B168,0x19640,0x17B18,
0x15FF0,0x144C8,0x129A0,0x10E78,
0x0F350,0x0D828,0x0BD00,0x0A1D8,
0x086B0,0x06B88,0x05060};

// 0x00002214 in IDA
uint16_t *factors = (uint16_t *)0x00003214;

int main(int argc, char **argv)
{
    uint8_t key[64] = {0};
    int fd = open("./libvalidate.so", O_RDONLY);
    off_t fsize = lseek(fd, 0, SEEK_END);

    if (mmap((void *)0x1000, fsize, PROT_READ, MAP_FIXED | MAP_SHARED, fd, 0)
    == MAP_FAILED) {
        perror("run as root");
        exit(EXIT_FAILURE);
    }

    for (uint32_t i = 0; i < 23; i++) {
        uint16_t c = 1;

        for (uint32_t j = 0; j < 0xd94; j++) {
            uint16_t factor = factors[j];
            uint16_t exponent = *((uint16_t *)ptrs[i]+j);
            if (exponent > 0) c = c * pow(factor, exponent);
        }

        key[i*2]   = c >> 8;
        key[i*2+1] = c & 0xff;
    }

    printf("%s\n", key);
    return 0;
}
```

```
$ gcc -m32 -o solve solve.c --std=c99 -lm
$ sudo ./solve
Should_have_g0ne_to_tashi_$tation@flare-on.com
```

Flag: Should_have_g0ne_to_tashi_$tation@flare-on.com

FLARE-ON CHALLENGE #7

Opening the YUSoMeta.exe with IDA, shows that it is a Microsoft.Net assembly. So, I decided to try decompiling using .NET Reflector.

Viewing the binary using the Reflector, gave the below details:

```
[assembly: AssemblyAlgorithmId(0)]
[assembly: AssemblyProduct("FLARE-On Challenge")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows=true)]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyCopyright("Copyright 2015, FireEye Inc.")]
[assembly: AssemblyTitle("Yo dawg, I heard you were meta")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("deadbeef-1337-beef-babe-f33dc00ffeee")]
[assembly: AssemblyFileVersion("12.34.56.78")]
[assembly: SuppressIldasm]
[assembly: PoweredBy("Powered by SmartAssembly 6.9.0.114")]
```

And the code is obfuscated using SmartAssembly

```
public static void CreateMemberRefsDelegates(int typeID)
{
    // This item is obfuscated and can not be translated.
}
```

Deobfuscating the binary with de4dot, dumped YUSoMeta-cleaned.exe. On analyzing the deobfuscated binary with the Reflector, gave some interesting information.

```
    Console.WriteLine(Encoding.ASCII.GetString(bytes));
    Console.Write(Encoding.ASCII.GetString(buffer4));
    string str = Console.ReadLine().Trim();
    string password = smethod_0(class2, buffer2) + '_' + smethod_3();// build 
password from some routine
    if (str == password)
    {
        Console.WriteLine(Encoding.ASCII.GetString(buffer6));
        Console.Write(Encoding.ASCII.GetString(buffer7));
        Console.WriteLine(smethod_1(str, buffer));  // crypto routine, 
decrypt and print flag
    }
    else
    {
        Console.WriteLine(Encoding.ASCII.GetString(buffer5));
    }
```

If the correct password is supplied, the flag will be printed. My first attempt was to patch the bytecodes in the deobfuscated executable, to print the password/flag. For this, I loaded the executable in IDA, and browsed to:

```
.method privatescope static hidebysig void Main(string[] args)

    call      string ns2.Class3::smethod_0(class ns1.Class1 class1_0, unsigned
int8[] byte_0)
    ldc.i4.s 0x5F
    box       [mscorlib]System.Char
    call      string ns2.Class3::smethod_3()
    call      string [mscorlib]System.String::Concat(object, object, object)
    stloc.3
    ldloc.2
    ldloc.3
    call      bool [mscorlib]System.String::op_Equality(string, string)
    brfalse.s loc_316
    call      class [mscorlib]System.Text.Encoding
[mscorlib]System.Text.Encoding::get_ASCII()
    ldloc.s   8
    callvirt instance string
[mscorlib]System.Text.Encoding::GetString(unsigned int8[])
    call      void [mscorlib]System.Console::WriteLine(string)
    call      class [mscorlib]System.Text.Encoding
[mscorlib]System.Text.Encoding::get_ASCII()
    ldloc.s   9
    callvirt instance string
[mscorlib]System.Text.Encoding::GetString(unsigned int8[])
    call      void [mscorlib]System.Console::Write(string)
```

A good reference to byte codes is found in List of CIL instructions Wikipedia. Using the details in wiki, I patched the byte codes. But this ended in printing only garbage. Next, I decided to set breakpoint in the string comparison check:

```
call      bool [mscorlib]System.String::op_Equality(string, string)
brfalse.s loc_316
```

For this, I analyzed mscorlib.dll and picked up a few functions. Then, I attached windbg to the process and set the breakpoints:

>YUSoMeta-cleaned.exe
Warning! This program is 100% tamper-proof!
Please enter the correct password:

.loadby sos clr
0:007> !bpmd mscorlib.dll System.String.Compare
0:007> !bpmd mscorlib.dll System.String.Equals
0:007> !bpmd mscorlib.dll System.String.op_Equality

Supplying a junk password, I hit breakpoint in [System.String.Equals(System.String, System.String)]
mscorlib_ni+0x4ff0c3:
00000644`784ff0c3 4c3bc2   cmp   r8,rdx

Dumping the memory pointed by R8 had the input string, but RDX had some junk values. This reminded me of challenge #4. So, I did the same against the original obfuscated executable, and dumped the memory pointed by R8 and RDX.

```
>YUSoMeta.exe
Warning! This program is 100% tamper-proof!
Please enter the correct password: sdsf
```

```
0:000> dd r8
00000000`0f0575c0  7869b320 00000644 00000004 00640073
00000000`0f0575d0  00660073 00000000 00000000 00000000
00000000`0f0575e0  00000000 00000000 7869d610 00000644
00000000`0f0575f0  00000000 00000000 00000000 00000000
00000000`0f057600  00541118 00000000 02d045a8 00000000
00000000`0f057610  00000001 00000000 00000000 00000000
00000000`0f057620  786afa00 00000644 0f0575e8 00000000
00000000`0f057630  00000000 00000000 00000000 00000000
```

```
0:000> du r8+c
00000000`0f0575cc  "sdsf"
```

```
0:000> dd rdx
00000000`0f07b1e0  7869b320 00000644 00000036 0065006d
00000000`0f07b1f0  00610074 00720070 0067006f 00610072
00000000`0f07b200  006d006d 006e0069 00690067 00680073
00000000`0f07b210  00720065 005f0064 00440044 00420039
00000000`0f07b220  00310045 00300037 00430034 00390036
00000000`0f07b230  00460030 00340042 00320032 00310046
00000000`0f07b240  00300035 00410039 00360034 00420041
00000000`0f07b250  00390043 00380038 00000000 00000000
```

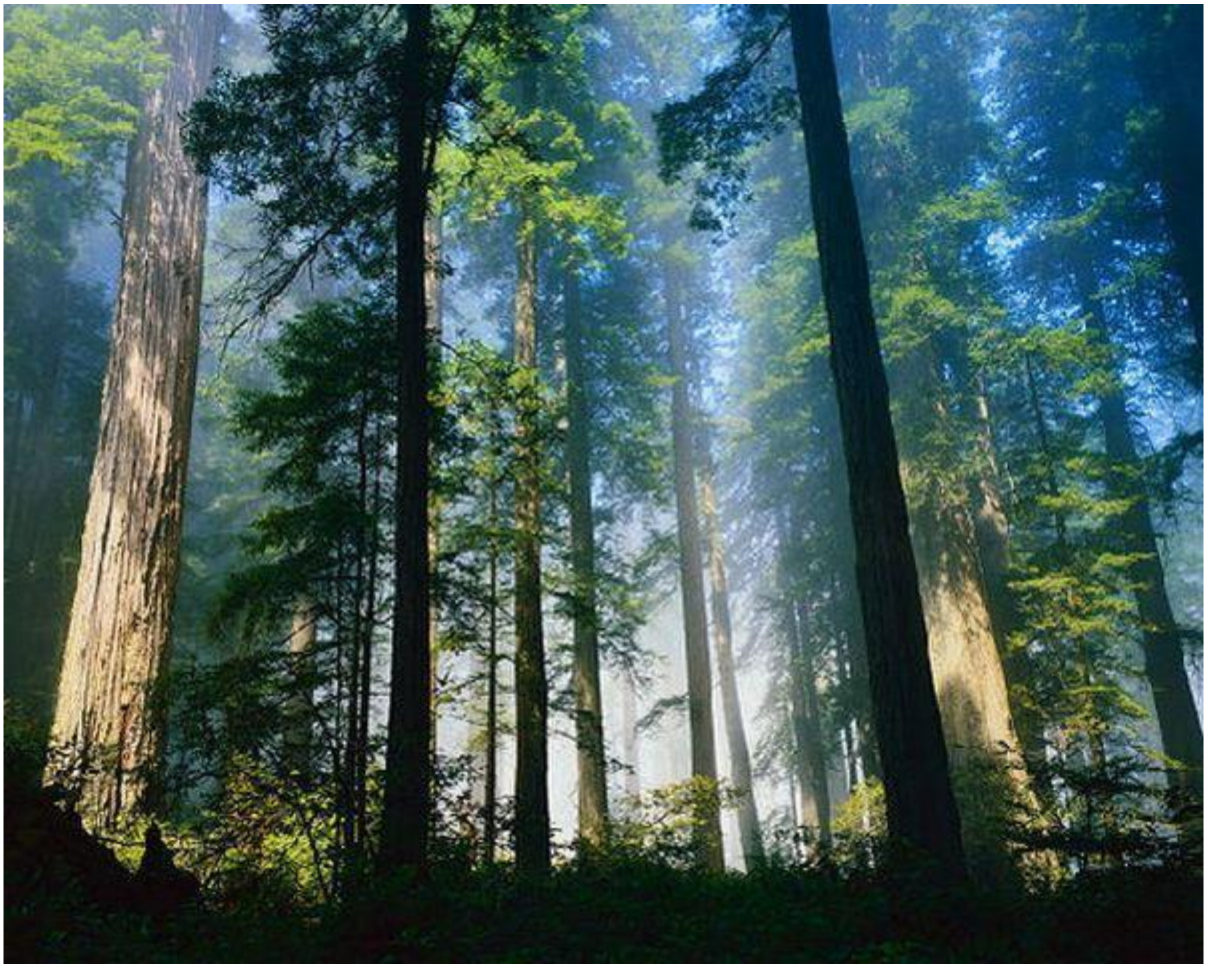
```
0:000> du rdx+c
00000000`0f07b1ec  "metaprogrammingisherd_DD9BE1704C"
00000000`0f07b22c  "690FB422F1509A46ABC988"
```

Highlighted DWORDS are the length, followed by Unicode string.
metaprogrammingisherd_DD9BE1704C690FB422F1509A46ABC988 is the password.

```
>YUSoMeta.exe
Warning! This program is 100% tamper-proof!
Please enter the correct password: metaprogrammingisherd_DD9BE1704C690FB422F1509A46ABC988
Thank you for providing the correct password.
Use the following email address to proceed to the next challenge: Justr3adth3sourc3@flare-on.com
```

FLARE-ON CHALLENGE #8

The binary gdssagh.exe has nothing more than a huge base64 encoded string. Dumping and decoding the string, I got a PNG image file.



This should be some steganography challenge! On that note, LSB steganography technique came to my mind. Below is the code to dump LSB bits:

```
from PIL import Image

img = Image.open("chall.png")
steg = ''

for rgb in img.getdata():
    for c in rgb: steg += str(c & 1)

flag = ''
for i in range(0, len(steg), 8):
    flag += chr(int(steg[i:i+8], 2))

open('flag','w').write(flag)
```

Fetched LSB bits were neither text, shellcode, nor any valid file format. Well, I failed to notice a lot of NUL bytes at the start of the dumped data.

$ cat flag | ndisasm -u - | less

```
00000000  B25A              mov dl,0x5a
00000002  0900              or [eax],eax
00000004  C00000            rol byte [eax],byte 0x0
00000007  0020              add [eax],ah
00000009  0000              add [eax],al
0000000B  00FF              add bh,bh
0000000D  FF00              inc dword [eax]
0000000F  001D00000000      add [dword 0x0],bl
00000015  0000              add [eax],al
00000017  0002              add [edx],al
00000019  0000              add [eax],al
0000001B  0000              add [eax],al
```

After some reading, I came across an LSB enhancement for visual steganalysis – if LSB is 0 then set the color value to 0, if LSB is 1 set color value to 255. Below is the generated image:

```python
from PIL import Image

img = Image.open("chall.png")
pixels = img.load()
w, l = img.size

for x in range(w):
    for y in range(l):
        enhance = [255 if c&1 == 1 else 0 for c in pixels[x,y]]
        pixels[x,y] = tuple(enhance)

img.save("chall_lsb_en.png")
```

Top portion of the image is fully black [color code 000000], which means there are lot of NUL bytes in the hidden data [Now I noticed!]. Since executables have a lot of NUL bytes at the start, there could be an embedded executable inside the image. So I checked the hex difference of the LSB dumped data with another PE file [gdssagh.exe] using vbindiff.

```
flag
0000 0000: B2 5A 09 00 C0 00 00 00  20 00 00 00 FF FF 00 00  .Z......  ........
0000 0010: 1D 00 00 00 00 00 00 00  02 00 00 00 00 00 00 00  ........  ........
0000 0020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........  ........
0000 0030: 00 00 00 00 00 00 00 00  00 00 00 00 0D 00 00 00  ........  ........
0000 0040: 70 F8 5D 70 00 2D 90 B3  84 1D 80 32 B3 84 2A 16  p.]p.-..  ...2..*.
0000 0050: 96 CE 04 0E 4E F6 E6 4E  86 B6 04 C6 86 76 76 F6  ....N..N  .....vv.
0000 0060: 2E 04 46 A6 04 4E AE 76  04 96 76 04 22 F2 CA 04  ..F..N.v  ..v."...
0000 0070: B6 F6 26 A6 74 B0 B0 50  24 00 00 00 00 00 00 00  ..&.t..P  $.......
0000 0080: BA 3A B6 83 98 BC C0 49  98 BC C0 49 98 BC C0 49  .:.....I  ...I...I
0000 0090: E9 44 08 49 78 BC C0 49  A7 B8 88 49 18 BC C0 49  .D.Ix..I  ...I...I
0000 00A0: 4A 96 C6 16 98 BC C0 49  00 00 00 00 00 00 00 00  J......I  ........
0000 00B0: 0A A2 00 00 32 80 C0 00  17 9C 85 AA 00 00 00 00  ....2...  ........
0000 00C0: 00 00 00 00 07 00 F0 80  D0 80 A0 30 00 40 00 00  ........  ...0.@..
0000 00D0: 00 20 00 00 00 00 00 00  00 08 00 00 00 08 00 00  . ......  ........
0000 00E0: 00 04 00 00 00 00 02 00  00 08 00 00 00 40 00 00  ........  .....@..
0000 00F0: 20 00 00 00 20 00 00 00  20 00 00 00 00 00 00 00   ... ...   .......

gdssagh.exe
0000 0000: 4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ......  ........
0000 0010: B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........  @.......
0000 0020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........  ........
0000 0030: 00 00 00 00 00 00 00 00  00 00 00 00 B0 00 00 00  ........  ........
0000 0040: 0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ........  !..L.!Th
0000 0050: 69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is progr  am canno
0000 0060: 74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run   in DOS 
0000 0070: 6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode....  $.......
0000 0080: 5D 5C 6D C1 19 3D 03 92  19 3D 03 92 19 3D 03 92  ]\m..=..  .=...=..
0000 0090: 97 22 10 92 1E 3D 03 92  E5 1D 11 92 18 3D 03 92  ."...=..  .....=..
0000 00A0: 52 69 63 68 19 3D 03 92  00 00 00 00 00 00 00 00  Rich.=..  ........
0000 00B0: 50 45 00 00 4C 01 03 00  E6 47 B6 55 00 00 00 00  PE..L...  .G.U....
0000 00C0: 00 00 00 00 E0 00 0F 01  0B 01 05 0C 00 BE 0C 00  ........  ........
0000 00D0: 00 04 00 00 00 00 00 00  00 10 00 00 00 10 00 00  ........  ........
0000 00E0: 00 D0 0C 00 00 00 40 00  00 10 00 00 00 02 00 00  ......@.  ........
0000 00F0: 04 00 00 00 04 00 00 00  04 00 00 00 00 00 00 00  ........  ........
```

There is a perfect pattern. All the NUL bytes are in the same index. Non-NULL bytes remain as non-NULL bytes, but there is some transformation. Inspecting individual bytes, I noticed that, the byte in the LSB dumped file, is the reverse of the one in the gdssagh.exe. So change the endianness of the bytes, and dump the hidden data.

```python
from PIL import Image

im = Image.open("chall.png")
steg = ''

for rgb in im.getdata():
    for c in rgb: steg += str(c & 1)

flag = ''
for i in range(0, len(steg), 8):
    flag += chr(int(steg[i:i+8][::-1], 2))

open('flag','w').write(flag)
```

$ file flag
flag: PE32 executable (console) Intel 80386, for MS Windows

Execute the binary, and it prints the flag **Im_in_ur_p1cs@flare-on.com**

FLARE-ON CHALLENGE #9

Initial analysis on you_are_very_good_at_this.exe using IDA, shows this is some obfuscated executable. The function at 0x00401000 is the main routine, which reads input. The input could be a maximum of 50 bytes. Later I realized that, this code path is never executed.

```
.text:0040103C                push    32h               ; nNumberOfBytesToRead
.text:0040103E                push    offset input      ; lpBuffer
.text:00401043                push    dword ptr [ebp-0Ch] ; hFile
.text:00401046                call    ReadFile
```

Running vicOlly trace, it is noticed that the instructions are executed from the stack as well. And the execution switches between the stack and the main executable.

```
Main    you_are_  0040116D  MOV DWORD PTR SS:[ESP+108],EAX
Main    you_are_  00401174  LEA EAX,DWORD PTR SS:[EBP-8]      EAX=0018FF80
Main    you_are_  00401177  LEA EBX,DWORD PTR SS:[ESP-10]     EBX=0018FE40
Main    you_are_  0040117B  MOV DWORD PTR DS:[EBX],10248489
Main    you_are_  00401181  MOV DWORD PTR SS:[ESP-C],C300000
Main    you_are_  00401189  MOV DWORD PTR SS:[ESP-8],EBX
Main    you_are_  0040118D  MOV DWORD PTR SS:[ESP-4],119D
Main    you_are_  00401195  ADD DWORD PTR SS:[ESP-4],EDI      FL=0
Main    you_are_  00401199  SUB ESP,8                         FL=PA, ESP=0018FE48
Main    you_are_  0040119C  RET                               ESP=0018FE4C
Main              0018FE40  MOV DWORD PTR SS:[ESP+110],EAX
Main              0018FE47  RET                               ESP=0018FE50
Main    you_are_  0040119D  XOR DWORD PTR SS:[ESP+108],4000   FL=P
Main    you_are_  004011A8  ADD DWORD PTR SS:[ESP+C0],1736
Main    you_are_  004011B3  SUB DWORD PTR SS:[ESP+B8],-1736   FL=CPA
Main    you_are_  004011BE  JMP SHORT you_are_.004011BF
Main    you_are_  004011BF  INC EAX                           FL=CP, EAX=0018FF81
Main    you_are_  004011C1  MOV EAX,EDI                       EAX=00400000
Main    you_are_  004011C3  OR EAX,2068                       FL=0, EAX=00402068
Main    you_are_  004011C8  MOV EAX,DWORD PTR DS:[EAX]        EAX=7DD71282
Main    you_are_  004011CA  OR DWORD PTR SS:[ESP+FC],FFFFFF   FL=PS
Main    you_are_  004011D2  MOV DWORD PTR SS:[ESP-4],1
Main    you_are_  004011DA  MOV DWORD PTR SS:[ESP-10],824848
Main    you_are_  004011E2  SHL DWORD PTR SS:[ESP-4],16       FL=P
Main    you_are_  004011E7  JNZ SHORT you_are_.004011EB
Main    you_are_  004011EB  MOV DWORD PTR SS:[ESP-C],C300000
Main    you_are_  004011F3  ADD DWORD PTR SS:[ESP+D4],1736
Main    you_are_  004011FE  MOV DWORD PTR SS:[ESP-8],EBX
Main    you_are_  00401202  ADD DWORD PTR SS:[ESP-4],120E     FL=0
Main    you_are_  0040120A  ADD ESP,-8                        FL=CP, ESP=0018FE48
Main    you_are_  0040120D  RET                               ESP=0018FE4C
Main              0018FE40  MOV DWORD PTR SS:[ESP+108],EAX
Main              0018FE47  RET                               ESP=0018FE50
Main    you_are_  0040120E  MOV DWORD PTR SS:[ESP+100],EDI
```

At this point, I decided to analyze the program using PIN, for better control. First, trace execution when EIP either points to the main executable or the stack.

>pin -t itrace.dll -- you_are_very_good_at_this.exe
I have evolved since the first challenge. You have not. Bring it.
Enter the password> ABCDEFGH
You are failure

Then I searched for CMP instructions, for any hardcoded constraint check during execution.

```
$ cat itrace | grep cmp
0x18fdc0 : cmpxchg bl, dl
0x18fdc0 : cmpxchg bl, dl
0x401bbd : cmp ebx, 0x7
```

```
0x18fdc4 : cmpxchg bl, dl
......
0x18fe60 : cmpxchg bl, dl
0x18fe60 : cmpxchg bl, dl
0x401c27 : cmp eax, 0x29
```

The CMP EAX, 0x29 instruction looked interesting as the challenge binary reads max 0x32 bytes as input. So I assumed this to be possible length check. Generate the trace of the program, by fetching the register values.

>pin -t exectrace.dll -- you_are_very_good_at_this.exe
I have evolved since the first challenge. You have not. Bring it.
Enter the password> ABCD

```
$ cat exectrace | grep 0x401c27
0x401c27 : cmp eax, 0x29
0x401c27 : [0] [0x29]
```

I passed ABCD as input and was expecting CMP 4, 0x29 to be the comparison. But EAX is still 0. Since suffix of flag is known, I passed in **AAAAAAAAAAAAAAAAAAAAAAAAAAAA@flare-on.com**

>pin -t exectrace.dll -- you_are_very_good_at_this.exe
I have evolved since the first challenge. You have not. Bring it.
Enter the password> AAAAAAAAAAAAAAAAAAAAAAAAAAAA@flare-on.com
You are failure

```
$ cat exectrace | grep 0x401c27
0x401c27 : cmp eax, 0x29
0x401c27 : [0xd] [0x29]
```

Now, that's interesting – CMP 0xD, 0x29. Note that strlen ('@flare-on.com') == 0xD. The binary seems to increment a counter for every valid comparison. Finally, if the counter is 0x29, that is a valid flag. So the idea is to bruteforce byte by byte, and see if the EAX values increase during comparison. If yes, consider that as a valid byte.

>python pin.py
I
Is
Is_
Is_t
Is_th
……………………………….
Is_th1s_3v3n_mai_finul_foa
Is_th1s_3v3n_mai_finul_foar
Is_th1s_3v3n_mai_finul_foarm

Flag: Is_th1s_3v3n_mai_finul_foarm@flare-on.com

```python
from subprocess import PIPE, Popen

pintool = 'count.dll'
program = 'you_are_very_good_at_this.exe'
payload = list('AAAAAAAAAAAAAAAAAAAAAAAAAAAA@flare-on.com')

def getcount(payload):
    global pintool, program
    p = Popen(["pin", "-t", pintool, "--", program],
                   stdin=PIPE, stdout=PIPE, stderr=PIPE)
    a = p.stdout.readline()
    count = p.communicate(payload)[1][11:13]
    return int(count)

flag = ''
success = 13

for a in range(28):
    for b in range(32, 127):
        payload[a] = chr(b)
        count = getcount(''.join(payload))
        if count <= success: continue
        success = count
        flag += chr(b)
        print flag
        break
```

I revisited this challenge once I finished the contest, to check the validation algorithm and other possible solutions. My PIN tracer needed a small modification. I added tracing for XED_CATEGORY_SEMAPHORE instructions to support CMPXCHG, in addition to XED_CATEGORY_BINARY, XED_CATEGORY_LOGICAL, XED_CATEGORY_ROTATE and XED_CATEGORY_SHIFT.

Once this is done, I got a neat trace for the user input AAAAAAAAAAAAAAAAAAAAAAAAAAAA@flare-on.com. This had enough details to reverse the algorithm.

```
0x18fdc0 : xor al, ah
0x18fdc0 : [0x41] [0x46] := [0x7]    -> 'A' used for operations
0x401b14 : rol al, cl
0x401b14 : [0x7] [0x56] := [0xc1]
0x18fdc0 : cmpxchg bl, dl
0x18fdc0 : [0xc1] [0xc3] [0x1]
0x18fdc0 : cmpxchg bl, dl
0x18fdc0 : [0x1] [0] [0x1]
0x401b6d : test eax, eax
0x401b6d : [0] [0]
......
0x401bc6 : add eax, ebx
0x401bc6 : [0x1] [0xc] := [0xd]
0x401c27 : cmp eax, 0x29
0x401c27 : [0xd] [0x29]
```

So the algorithm looks like this:

```
AL = input[i] ^ xor_key[i]
AL = ROL(AL, shift[i])
IF AL == check[i] : success++
```

Solver

```
xorkey = [0x46, 0x15, 0xf4, 0xbd, 0xff, 0x4c, 0xef, 0x46, 0xeb, 0xe6,
          0xb2, 0xeb, 0xf1, 0xc4, 0x34, 0x67, 0x39, 0xb5, 0x8e, 0xef,
          0x40, 0x1b, 0x74, 0x0d, 0x60, 0x26, 0x45, 0xa8, 0x4a, 0x96,
          0xc9, 0x65, 0xe2, 0x32, 0x60, 0x64, 0x8c, 0x65, 0xe3, 0x8e, 0x9f]

shift = [0x56, 0xf5, 0xac, 0x1b, 0xb5, 0x93, 0x7e, 0xb8, 0x23, 0xda,
         0x0a, 0xf2, 0x01, 0x61, 0x5c, 0xc8, 0x4c, 0xd6, 0x16, 0x55,
         0x67, 0xb8, 0xc1, 0xf8, 0xbc, 0x11, 0xfa, 0x9b, 0x6b, 0xf9,
         0xd4, 0x75, 0x87, 0xca, 0xce, 0xbe, 0x4e, 0x6e, 0xf1, 0xb9, 0x6e]

check = [0xc3, 0xcc, 0xba, 0x4e, 0xf2, 0xeb, 0x27, 0x19, 0xc6, 0x42,
         0x06, 0x16, 0x5d, 0x53, 0x55, 0x0e, 0x66, 0xf4, 0xf9, 0x30,
         0x9a, 0x77, 0x56, 0x6b, 0xf0, 0x8e, 0xdc, 0x2e, 0x50, 0xe1,
         0x5a, 0x80, 0x48, 0x5d, 0x53, 0xc2, 0xb8, 0xd2, 0x01, 0xc3, 0xbc]

def ROR(a, b):
    b = b % 8
    return (a >> b) | (a << (8 - b)) & 0xff

flag = ''
for i in range(41):
    AL = ROR(check[i], shift[i])
    AL = AL ^ xorkey[i]
    flag += chr(AL)

print flag
```

Importing PIN trace to IDA

PIN trace results can be parsed, and imported into IDA Pro for better static analysis. I did not explore this much, but below is the code:

```
import idaapi

def PatchNOP(address, sz):
    for addr in range(address, address+sz):
        PatchByte(addr, 0x90)
        MakeCode(addr)

TRACE = open('itrace').readlines()
EA_LIST = []
RETADDR = {}

# fetch address of instructions from trace
for index, line in enumerate(TRACE):
    # e.g. 0x4010ad : mov dword ptr [esp+eax*1], ebx
```

```python
    disass = line.split(':')
    EA = int(disass[0], 16)
    Mnem = disass[1]
    # skip stack address
    if GetFlags(EA) == 0: continue
    if EA not in EA_LIST: EA_LIST.append(EA)

    # fetch target address of ret instruction
    if 'ret' in Mnem and index < len(TRACE)-1:
        ret = TRACE[index+1].split(':')[0]
        RETADDR[EA] = int(ret, 16)

# undefine overlapped ins and re-create code
for address in reversed(EA_LIST):
    MakeUnkn(address, DOUNK_EXPAND)
    MakeCode(address)
    idaapi.set_item_color(address, 0xD8D8D8)

seg_s = SegByBase(SegByName(".text"))
seg_e = SegEnd(seg_s)

# replace undefined/data/not_traced address with NOP
for address in range(seg_s, seg_e):
    flag = GetFlags(address)
    if isData(flag) == True: PatchNOP(address, 1)
    elif isUnknown(flag) == True: PatchNOP(address, 1)

# remove invalid jump sequence
# jmp + xor + jz, jnz + jz, jnz/jz + nop
address = seg_s
cjumps = ['jnz', 'jz']

while address != BADADDR and address < seg_e:
    c_address  = address
    n_address  = NextHead(address)
    nn_address = NextHead(n_address)
    c_ins  = GetMnem(c_address)
    n_ins  = GetMnem(n_address)
    nn_ins = GetMnem(nn_address)
    # address to update
    up_addr = n_address

    if c_ins == 'jmp' and n_ins == 'xor' and nn_ins == 'jz':
        up_addr = NextHead(nn_address)
        PatchNOP(c_address, up_addr - c_address)

    elif c_ins in cjumps and n_ins in cjumps:
        up_addr = NextHead(n_address)
        PatchNOP(c_address, up_addr - c_address)

    elif c_ins in cjumps and n_ins == 'nop':
        up_addr = NextHead(n_address)
        PatchNOP(c_address, up_addr - c_address)
    address = up_addr

for address, ret in RETADDR.items():
    MakeComm(address, "ret[%s]" % hex(ret))
```

Sample results using IDAPython + PIN for partial deobfuscation

```
.text:00401BBD                 cmp     ebx, 7
.text:00401BC0                 setz    bl
.text:00401BC3                 sub     al, bl
.text:00401BC5                 pop     ebx
.text:00401BC6                 add     eax, ebx
.text:00401BC8                 nop
.text:00401BC9                 nop
.text:00401BCA                 nop
.text:00401BCB                 nop
.text:00401BCC                 nop
.text:00401BCD                 retn                       ; ret[0x40173f]

.text:00401B87                 mov     dword ptr [esp-10h], 301D8B64h
.text:00401B8F                 mov     dword ptr [esp-0Ch], 0C3000000h
.text:00401B97                 push    offset loc_401BA2
.text:00401B9C                 push    esp
.text:00401B9D                 sub     dword ptr [esp], 0Ch
.text:00401BA1                 retn                       ; ret[0x18fe98]

.text:0040188D                 mov     dword ptr [ebx+84h], 4EF5BEACh
.text:00401897                 nop
.text:00401898                 nop
.text:00401899                 nop
.text:0040189A                 mov     dword ptr [ebx+80h], 2C2B2A07h
.text:004018A4                 nop
.text:004018A5                 nop
.text:004018A6                 nop
.text:004018A7                 nop
.text:004018A8                 nop
.text:004018A9                 mov     dword ptr [ebx+7Ch], 0E051511h
```

The binary worked just fine, after patching using IDA generated DIF file:

```
>patched.exe
I have evolved since the first challenge. You have not. Bring it.
Enter the password> Is_th1s_3v3n_mai_finul_foarm@flare-on.com
```

```
CPU Disasm
Address         Hex dump        Command
00401C27    |.  83F8 29        CMP EAX, 29
CPU - main thread, module patched

EAX 00000029
ECX 00000068
EDX 00000001
EBX 00000028
ESP 0018FF50
EBP 0018FF88
```

FLARE-ON CHALLENGE #10

Loader.exe is a 3MB large executable. Running the binary, I got an error message saying "Must be run on x86 architecture". But I could not find any string references to this. So, I decided to inspect the binary using Rohitab API Monitor. The binary tries to write challenge.sys and ioctl.exe to the system32 folder.

```
#     API   Return      Value       Error
CreateFileW ( "C:\Windows\system32\challenge.sys", GENERIC_READ |
GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, 0x00b2e404, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL ) INVALID_HANDLE_VALUE    5 = Access is denied.
#     API   Return      Value       Error
CreateFileW ( "C:\Windows\system32\ioctl.exe", GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, 0x00b2e404, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL ) INVALID_HANDLE_VALUE    5 = Access is denied.
```

Running the executable with higher privileges, dumps the binaries to system32. In ioctl.exe, the first argument passed to the program, should be a hexadecimal string, which is encoded as a hexadecimal number e.g. input *aaaaaaaa* is encoded as 0xaaaaaaaa. This is done by the function at 0x004014AF.

```
(gdb) break *0x0040103B
Breakpoint 1 at 0x40103b
(gdb) run abcdef02
Breakpoint 1, 0x0040103b in ?? ()
(gdb) p/x $eax
$1 = 0xabcdef02
```

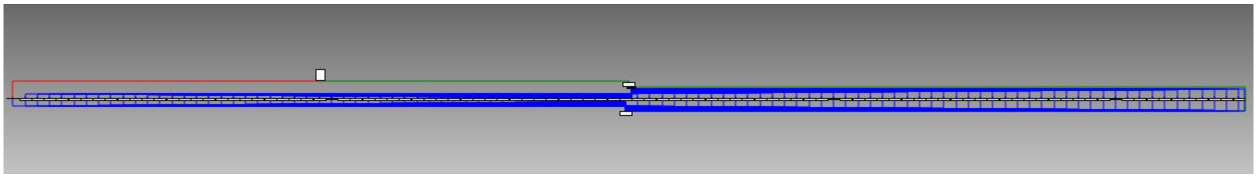
Ioctl.exe communicates to driver using device \\.\challenge and passes the IOCTL number/Control Code read, as user input.

```
hDevice   =   CreateFileA("\\.\challenge",   GENERIC_READ   |   GENERIC_WRITE,
FILE_SHARE_READ|FILE_SHARE_WRITE, 0, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);

DeviceIoControl(hDevice, arg_ioctl_num, 0, 0, &OutBuffer, 8, &BytesReturned,
&Overlapped)
```

Then I started analyzing the challenge.sys using IDA. The DriverEntry is at 0x002A203E. IRP routines are registered as below:

```
.text:0029D86B inc:
.text:0029D86B                 mov     edx, [ebp+counter]
.text:0029D86E                 add     edx, 1
.text:0029D871                 mov     [ebp+counter], edx
.text:0029D874
.text:0029D874 register_irp_routines:
.text:0029D874                 cmp     [ebp+counter], 27
.text:0029D878                 jge     short out_of_loop
.text:0029D87A                 mov     eax, [ebp+counter]
.text:0029D87D                 mov     ecx, [ebp+DRIVER_OBJECT]
.text:0029D880                 mov
[ecx+eax*4+DRIVER_OBJECT.MajorFunction], offset GenericHandler
.text:0029D888                 jmp     short inc
```

Include DRIVER_OBJECT structure in IDA to make the analysis easier. The GenericHandler handles the I/O request to the driver. The handler is very huge, majorly a dispatch routine, based on the ioctl number.



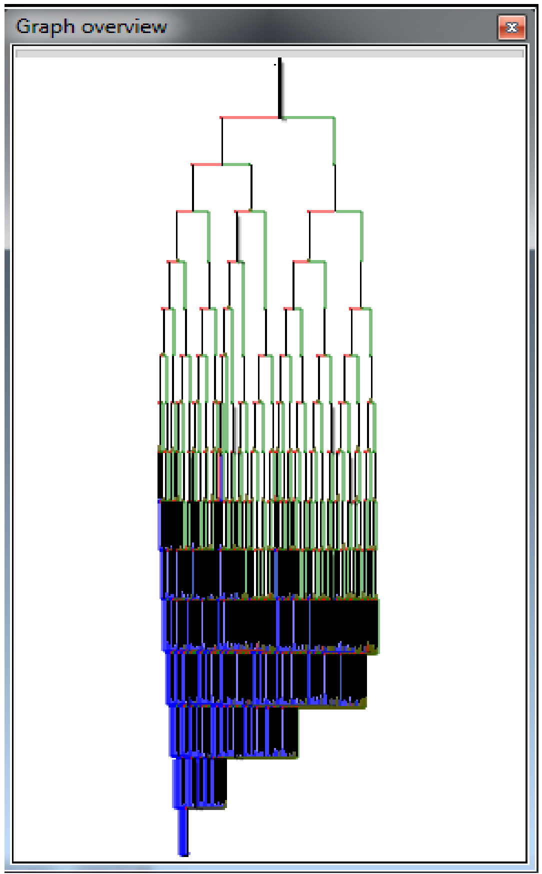
Then I started analyzing the handler, for which I included the IRP and _IO_STACK_LOCATION structures. Below are some details:

```
.text:0029CD40                 mov     edx, [ebp+IRP]
.text:0029CD43                 push    edx
.text:0029CD44                 call    IoGetCurrentIrpStackLocation
.text:0029CD49                 mov     [ebp+CurrentStackLocation], eax
.text:0029CD4C                 mov     eax, [ebp+CurrentStackLocation]
.text:0029CD4F                 mov     cl,
[eax+_IO_STACK_LOCATION.MajorFunction]
.text:0029CD51                 mov     [ebp+irp_major_func_code], cl
.text:0029CD54                 cmp     [ebp+irp_major_func_code], 0Eh ;
IRP_MJ_DEVICE_CONTROL
.text:0029CD58                 jz      short on_device_control

.text:0029CD5F                 mov     edx, [ebp+CurrentStackLocation]
.text:0029CD62                 mov     eax,
[edx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
.text:0029CD65                 mov     [ebp+_io_control_code], eax
.text:0029CD68                 mov     ecx, [ebp+_io_control_code]
.text:0029CD6B                 mov     [ebp+io_control_code], ecx
.text:0029CD6E                 mov     edx, [ebp+io_control_code]
.text:0029CD71                 sub     edx, 22E004h
.text:0029CD77                 mov     [ebp+io_control_code], edx
.text:0029CD7A                 cmp     [ebp+io_control_code], 190h
.text:0029CD81                 ja      complete_request
```

IOCTL number from userland is fetched as _IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode. The PEiD Krypto analyzer plugin showed the use of TEA decryption algorithm

```
MarkPosition(0x0001102C, 0, 0, 0, slotidx + 0, "TEAN [32 rounds]");
MakeComm(PrevNotTail(0x0001102D), "TEAN [32 rounds]\n2 DWORDs of 32-round
TEAN: Delta and precomputed initial sum for decryption (32*Delta)");
```

The TEA decryption routine, is referenced in function 0x000110F0, which chunks data to be encrypted into TEA block size. This function is further referenced in 3 other places [0x2D2E0, 0xADC40, 0x1A4EC0], which looked like obfuscated IOCTL handlers. Below is the graph overview:



I decided to take a closer look at these 3 IOCTL handlers where TEA decryption is performed.

IOCTL@0x2D2E0

```
.text:000ADC24                 lea     ecx, [ebp+key]
.text:000ADC27                 push    ecx
.text:000ADC28                 lea     edx, [ebp+sz]    ; 40 bytes
.text:000ADC2B                 push    edx
.text:000ADC2C                 push    offset IOCTLA_DATA
.text:000ADC31                 call    DoTEADecrypt
.text:000ADC36                 mov     esp, ebp
.text:000ADC38                 pop     ebp
.text:000ADC39                 retn
```

IOCTL@0xADC40

```
.text:001A4E99                 lea     edx, [ebp+key]
.text:001A4E9C                 push    edx
.text:001A4E9D                 lea     eax, [ebp+sz]    ; 80 bytes
.text:001A4EA0                 push    eax
.text:001A4EA1                 push    offset IOCTLB_DATA
.text:001A4EA6                 call    DoTEADecrypt
.text:001A4EAB                 mov     esp, ebp
.text:001A4EAD                 pop     ebp
.text:001A4EAE                 retn
```

IOCTL@0x1A4EC0

```
.text:0029C119                  lea     edx, [ebp+key]
.text:0029C11C                  push    edx
.text:0029C11D                  lea     eax, [ebp+sz]    ; 80 bytes
.text:0029C120                  push    eax
.text:0029C121                  push    offset IOCTLC_DATA
.text:0029C126                  call    DoTEADecrypt
.text:0029C12B                  mov     esp, ebp
.text:0029C12D                  pop     ebp
.text:0029C12E                  retn
```

All three TEA decryption operation uses the same hardcoded key.

```
.text:000ADC48 mov      [ebp+key], 33323130h
.text:000ADC4F mov      [ebp+key+4], 37363534h
.text:000ADC56 mov      [ebp+key+8], 42413938h
.text:000ADC5D mov      [ebp+key+0Ch], 46454443h
```

Then, let us inspect the data that is getting decrypted i.e. arrays IOCTLA_DATA[40], IOCTLB_DATA[80] and IOCTLC_DATA[80]

IOCTLA_DATA

```
.data:0029F210 IOCTLA_DATA      db ?
.data:0029F211 byte_29F211      db ?
.data:0029F212 byte_29F212      db ?
.data:0029F213 byte_29F213      db ?
.data:0029F214 byte_29F214      db ?
.data:0029F215 byte_29F215      db ?
.data:0029F216 byte_29F216      db ?
.data:0029F217 byte_29F217      db ?
.data:0029F218 byte_29F218      db ?
.data:0029F219 byte_29F219      db ?
.data:0029F21A byte_29F21A      db ?
```

IOCTLB_DATA

```
.data:0029F1D4 IOCTLB_DATA      db 0
.data:0029F1D5 byte_29F1D5      db 0
.data:0029F1D6 byte_29F1D6      db 0
.data:0029F1D7 byte_29F1D7      db 0
.data:0029F1D8 byte_29F1D8      db 0
.data:0029F1D9 byte_29F1D9      db 0
.data:0029F1DA byte_29F1DA      db 0
.data:0029F1DB byte_29F1DB      db 0
.data:0029F1DC                  db 0
.data:0029F1DD byte_29F1DD      db 0
.data:0029F1DE byte_29F1DE      db 0
.data:0029F1DF byte_29F1DF      db 0
.data:0029F1E0                  db 0
.data:0029F1E1 byte_29F1E1      db 0
.data:0029F1E2 byte_29F1E2      db 0
.data:0029F1E3 byte_29F1E3      db 0
.data:0029F1E4 byte_29F1E4      db 0
.data:0029F1E5 byte_29F1E5      db 0
.data:0029F1E6 byte_29F1E6      db 0
```

```
.data:0029F1E7 byte_29F1E7     db 0
.data:0029F1E8                 db 0
.data:0029F1E9 byte_29F1E9     db 0
.data:0029F1EA                 db 0
.data:0029F1EB byte_29F1EB     db 0
```

IOCTLC_DATA

```
.data:0029F1FC IOCTLC_DATA     db 0
.data:0029F1FD                 db 0
.data:0029F1FE byte_29F1FE     db 0
.data:0029F1FF byte_29F1FF     db 0
.data:0029F200 byte_29F200     db ?
.data:0029F201                 db ?
.data:0029F202                 db ?
.data:0029F203 byte_29F203     db ?
.data:0029F204 byte_29F204     db ?
.data:0029F205 byte_29F205     db ?
.data:0029F206 byte_29F206     db ?
.data:0029F207                 db ?
.data:0029F208                 db ?
.data:0029F209 byte_29F209     db ?
.data:0029F20A byte_29F20A     db ?
.data:0029F20B                 db ?
.data:0029F20C                 db ?
.data:0029F20D                 db ?
.data:0029F20E                 db ?
.data:0029F20F byte_29F20F     db ?
```

Except for array IOCTLA_DATA[40], rest of the 2 arrays had unreferenced addresses. This is not good for decryption. So I inspected array IOCTLA_DATA[40], to check how they are being populated. Xrefs showed that each of the bytes are populated using hardcoded values.

```
.text:000257E0 mov     [ebp+var_40], 56h
.text:000257E4 mov     al, [ebp+var_40]
.text:000257E7 mov     IOCTLA_DATA, al
```

Collect all bytes using an IDAPython script, and decrypt the extracted data using TEA. This will give the flag.

IDAPython script:

```
IOCTLA_DATA = 0x0029F210
sz = 40
tea_data = []

for address in range(IOCTLA_DATA, IOCTLA_DATA+sz):
    ref = list(DataRefsTo(address))[0]
    ins = PrevHead(PrevHead(ref))
    c = DecodeInstruction(ins).Op2.value
    tea_data.append(c)

print tea_data
```

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

//gcc -o solver solver.c --std=c99

uint32_t key[4] = {0x33323130, 0x37363534, 0x42413938, 0x46454443};

uint8_t cipher[40] = {
0x56, 0x7F, 0xDC, 0xFA, 0xAA, 0x27, 0x99, 0xC4, 0x6C, 0x7C,
0xFC, 0x92, 0x61, 0x61, 0x47, 0x1A, 0x19, 0xB9, 0x63, 0xFD,
0x0C, 0xF2, 0xB6, 0x20, 0xC0, 0x2D, 0x5C, 0xFD, 0xD9, 0x71,
0x54, 0x96, 0x4F, 0x43, 0xF7, 0xFF, 0xBB, 0x4C, 0x5D, 0x31};

void decrypt(uint32_t *buf, uint32_t *key)
{
    uint32_t sum = 0xC6EF3720;
    uint32_t delta = 0x9E3779B9;
    uint32_t b0 = buf[0], b1 = buf[1];
    uint32_t k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];

    for (uint32_t i = 0; i < 32; i++) {
        b1 -= ((b0 << 4) + k2) ^ (b0 + sum) ^ ((b0 >> 5) + k3);
        b0 -= ((b1 << 4) + k0) ^ (b1 + sum) ^ ((b1 >> 5) + k1);
        sum -= delta;
    }

    buf[0] = b0; buf[1] = b1;
}

int main(int argc, char **argv)
{
    uint8_t flag[64] = {0};

    for (uint32_t i = 0; i < 40; i = i+8) {
        decrypt((uint32_t *)&cipher[i], key);
        memcpy(&flag[i], &cipher[i], 8);
    }

    printf("%s\n", flag);
    return 0;
}
```

Flag: unconditional_conditions@flare-on.com

FLARE-ON CHALLENGE #11

Name of the challenge binary CryptoGraph.exe suggests, that it involves some crypto. PEiD Krypto ANALyzer gives the below results:

```
MarkPosition(0x00410008, 0, 0, 0, slotidx + 0, "CryptGenRandom [Import]");
MakeComm(PrevNotTail(0x00410009), "CryptGenRandom [Import]\nMicrosoft
CryptoAPI import");
MarkPosition(0x004021E4, 0, 0, 0, slotidx + 1, "MD5");
MakeComm(PrevNotTail(0x004021E5), "MD5\nMD5 transform (\"compress\")
constants");
MarkPosition(0x00402A17, 0, 0, 0, slotidx + 2, "RC5 / RC6 [Init, -Delta]");
MakeComm(PrevNotTail(0x00402A18),  "RC5  /  RC6  [Init,  -Delta]\nRC5/6  32bit
magic constants, negative Delta");
```

There are references to MD5, RC5/RC6 and CryptGenRandom. The main routine is 0x00401F60 and 0x00402E77 is atoi, returning an integer for the user supplied argument. The function at 0x00401910 operates on resources.

Function 0x00401910 fetches resources 0x78 and 0x79. Resource ID 0x78 is 48 bytes in size. XOR transformations are performed and the first 16 bytes of the resource ID 0x78 is updated.

```
.text:00401A57                  movdqu  xmm0, xmmword ptr [esi+16]
.text:00401A5D                  movdqu  xmm1, xmmword ptr [esi+32]
.text:00401A64                  pxor    xmm1, xmm0
.text:00401A6E                  movdqu  xmm0, xmmword ptr [esi]
.text:00401A79                  pxor    xmm1, xmm0
.text:00401A7D                  movdqu  xmmword ptr [esi], xmm1
```

Function 0x00401090 initializes a few DWORDS in crypto structure and generates a random number [could not find its usage though]. IDA Class Informer fetches the below details:

```
.text:004010A1  lea     esi, [edi+4]
.text:004010A4  mov     dword ptr [esi], 0
.text:004010AA  push    0                   ; szContainer
.text:004010AC  push    esi                 ; phProv
.text:004010AD  mov     dword ptr [edi], offset ??_7RandomClass@@6B@ ; const
RandomClass::`vftable'
.text:004010B3  call    ds:CryptAcquireContextW
```

Function at 0x004015D0 is renamed to create_key_buffer. This routine takes user input, resource ID 0x78 [Named as RESA] and resource ID 0x79 [Named as RESB] as input. Some information on routines:

MD5Init@00401FE0
MD5Update@00402040
MD5Final@004020F0

RESB is a 1584 byte structure chunked into 48 byte structures. The first 48 bytes were not encrypted. Consider this as META chunk. It has magic bytes FLARE-ON and MD5 hash of the remaining 1536 bytes. The hash validation and header check is performed in function validate_resourceb@00401000

```
.text:00401013                  lea     ecx, [ebp+context]
.text:00401016                  call    MD5Init
.text:0040101B                  push    1536
```

```
.text:00401020                  lea     eax, [esi+Crypto::resource.rc5]
.text:00401023                  push    eax
.text:00401024                  lea     ecx, [ebp+context]
.text:00401027                  call    MD5Update
.text:0040102C                  lea     eax, [ebp+hash]
.text:0040102F                  push    eax
.text:00401030                  lea     ecx, [ebp+context]
.text:00401033                  call    MD5Final
.text:00401038                  cmp     dword ptr
[esi+Crypto::resource.magic], 'RALF'
.text:0040103E                  jnz     short fail
.text:00401040                  cmp     dword ptr [esi+4], 'NO-E'
.text:00401047                  jnz     short fail
.text:00401049                  lea     edx, [esi+Crypto.resource.rc5hash]
.text:0040104C                  mov     esi, 12
.text:00401051                  lea     ecx, [ebp+hash]
```

Then one byte of the user input, is used to modify the META chunk:

```
.text:0040162E                  lea     edx, [ebx+Crypto.resource.userbyte]
.text:0040163A                  mov     al, [ebp+arg]
.text:0040163D                  or      [edx], al
```

Next, gen_rc5_key@004014E0 computes the 16 byte key material, using RESA and userbyte [one byte from user + remaining from META chunk] using MD5 and HMAC routines.

MD5HMAC@00402870
HMACLoop@00401170

I did not reverse the above routines, there were multiple calls to MD5 routines along with IPAD, OPAD operations. So this must be some HMAC related calculation.

```
.text:00401214                  movdqa  xmm1, ds:IPAD
.text:0040121C                  movdqa  xmm2, ds:OPAD
.text:00401240                  movdqu  xmm0, [ebp+eax+k_ipad]
.text:00401246                  pxor    xmm0, xmm1
.text:0040124A                  movdqu  [ebp+eax+k_ipad], xmm0
.text:00401250                  movdqu  xmm0, [ebp+eax+k_opad]
.text:00401259                  pxor    xmm0, xmm2
.text:0040125D                  movdqu  [ebp+eax+k_opad], xmm0

.rdata:00414690 IPAD            xmmword 36363636363636363636363636363636h
.rdata:004146A0 OPAD            xmmword 5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5Ch
```

Decryption is performed on bytes [48 - 96] of RESB, using the already computed key. After decryption, it is checked if the first DWORD == 0:

```
.text:004016D4                  cmp     dword ptr [esi], 0
```

This constraint should be satisfied. Below is the script to find the byte:

```python
import gdb

value = 0
def quit():
    gdb.execute("set confirm off")
    gdb.execute("quit")

def callback_check_zf():
    global value
    EFLAGS = gdb.parse_and_eval("$eflags")
    ZF = int(EFLAGS) & 0x40
    if ZF:
        print "FOUND BYTE : %d" % (value)
        quit()
    value += 1
    # restart program
    gdb.execute("set $eip = 0x00402F77")

def callback_update_input():
    global value
    gdb.execute("set $edx = " + str(value))

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

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

OnBreakpoint("*0x004016D7", callback_check_zf)
OnBreakpoint("*0x00401F9B", callback_update_input)
```

(gdb) source brute.py
Breakpoint 1 at 0x4016d4
Breakpoint 2 at 0x401f9b

(gdb) run 0
FOUND BYTE: 205

Below are the routines dealing with RC5 algorithm:
InitRC5Table@00402A10
RC5MixSecret@00402AC0
RC5Decrypt@00402BE0

Once the valid byte is provided, MD5 hash of 32 bytes of RC5 structure is computed. The last 16 bytes of the structure, is the hash of the first 32 bytes i.e. MD5(RC5[0][0-32]) == STRUCT RC5[0][32-48]

After this, the next RC5 structure is chosen i.e. the address after META CHUNK + RC5[0]. The data from the decrypted RC5[0] is used for computing key material for decrypting the next struct i.e. RC5[1]. This operation is further performed in a loop, for decrypting structures RC5[1] to RC5[32]

Also, there is an integer in each structure, which is used for computing a loop counter. The loop counter is passed as an argument to HMACLoop@00401170. This routine takes a lot of time to execute, depending on the value of loop counter.

```
.text:00401757        mov     esi, eax      ; EAX is rc5 struct index
.text:00401759        lea     ecx, [esp+0E0h+context]
.text:0040175D        shr     esi, 4
.text:00401760        imul    esi, [edx+Crypto.key_index]
.text:00401767        add     esi, [edi-4]  ; integer in struct rc5 [ebx + 60]
.text:0040176A        mov     [edx+Crypto.key_index], esi
```

Overall, the structures look something similar to below:

```
// struct.h
struct RC5 {
    int padding;
    int counter;                          // HMAC loop
    char next_block_data_hmac_key[8];     // MD5 of element is HMAC key
    char next_block_data_hmac_msg[16];    // HMAC data, for next strcture key
    char current_block_md5_hash[16];      // MD5 of first 32 bytes
};

struct Crypto {
    DWORD *vftable;
    HCRYPTPROV *phProv;
    struct Resource {
        char magic[8];        // FLARE-ON
        int counter;          // HMAC loop
        int allocsz;
        char userbyte[8];     // one byte from user, MD5 of element is HMAC key
        char rc5hash[16];     // MD5 of RC5 structures
        char padding[8];
        struct RC5 rc5[32];   // RC5 structures
    } resource;
    int key_index;            // used for computing key index for secret.jpg
};
```

Import this into IDA using Parse C header file option and synchronize the structures to IDB using Local Types. This would assist in the analysis.

Once all the structures are decrypted, the program writes secret.jpg. But since the program runs forever, on higher counter values, some alternative is needed.

Analyzing the function DecryptSecret@00401CF0, an index into decrypted key buffer is computed using function @00401B60. The only variable this function relies on, comes from the Crypto structure.

```
.text:00401BB0 mov     ebx, [ebx+Crypto.key_index]
```

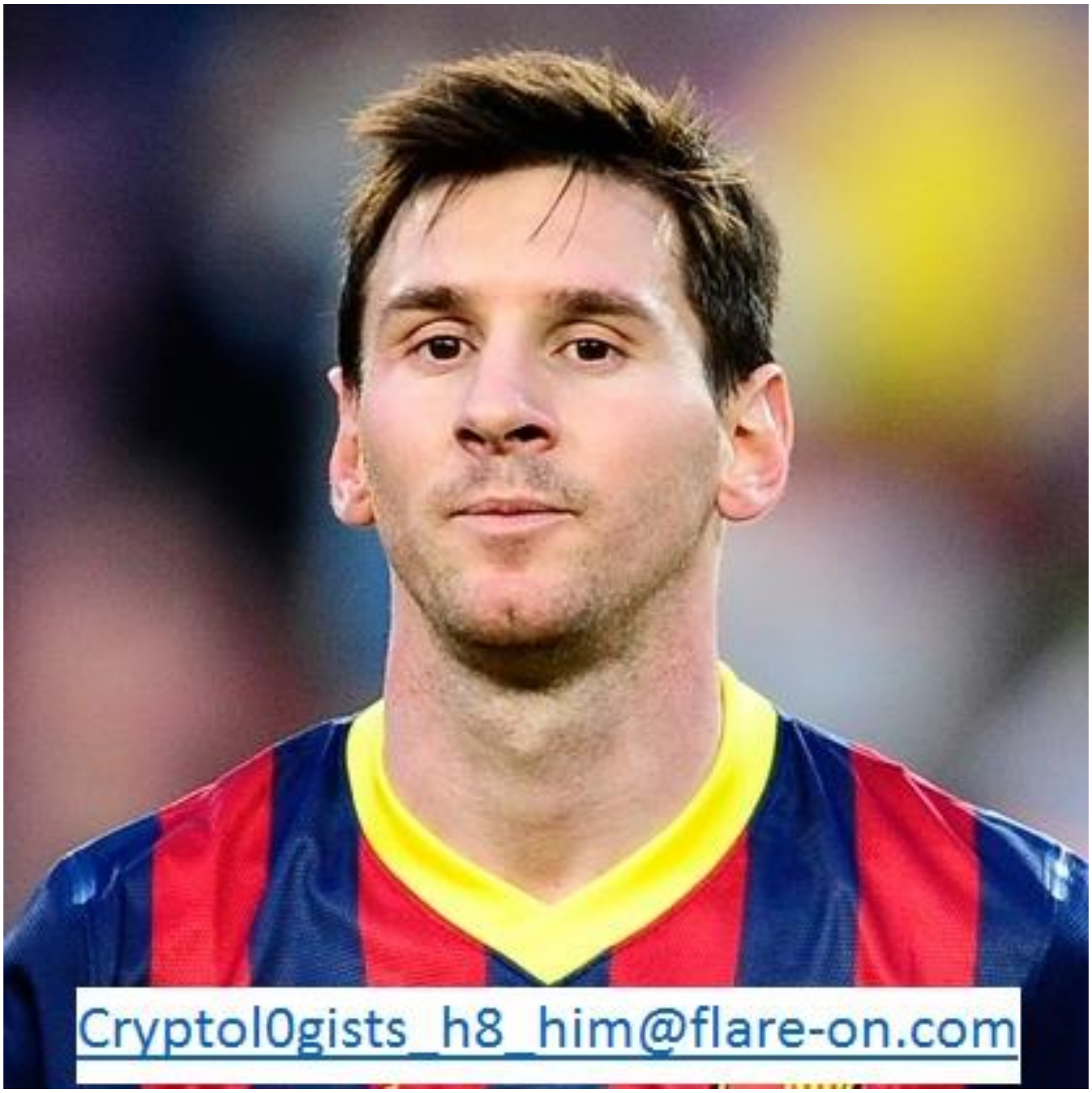
The return value of this function could be a maximum of 0xF. So the entire RC5 key material need not be decrypted, and an early termination of decryption loop should work.

```
.text:00401DAF     call    GetStructIndex
.text:00401DB4     mov     esi, eax          ; index into rc5 struct
.text:00401DB6     mov     eax, [ebp+Crypto]
.text:00401DB9     push    8                 ; size_t
.text:00401DBB     lea     ecx, [esi+esi*2]
.text:00401DBE     shl     ecx, 4            ; multiple of 48 ; (n+n*2) << 4
.text:00401DC1     add     ecx, 64           ; index to next_block_data_hmac_key
.text:00401DC4     add     ecx, eax          ; computed address
.text:00401DC6     push    ecx               ; RC5 key material
```

I decided to allow decryption to a certain structure index, then break out of loop and modify the index value returned by GetStructIndex. A simple bruteforce would dump the jpeg file.

```
(gdb) source solve.py
Breakpoint 1 at 0x4018c0
Breakpoint 2 at 0x401db4
Breakpoint 3 at 0x401fb6
(gdb) run 205
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 1
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 2
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 3
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 4
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 5
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 6
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 7
The "secret.jpg" has been written to the current execution folder.
Invalid file for index 8
The "secret.jpg" has been written to the current execution folder.
Valid file for index 9
```

```python
import gdb

index = 1
def quit():
    gdb.execute("set confirm off")
    gdb.execute("quit")

def callback_check_file():
    global index
    JPEG = 'ffd8ffe0'
    header = open('secret.jpg').read(4).encode("hex")

    if header == JPEG:
        print "Valid file for index %d" %(index)
        quit()
    else:
        print "Invalid file for index %d" %(index)
        index += 1
        # restart program
        gdb.execute("set $eip = 0x00403006")

def callback_break_loop():
    global index
    eax = int(gdb.parse_and_eval("$eax"))
    if index == eax - 1:
        # break loop
        gdb.execute("set $eax = 32")

def callback_update_index():
    global index
    gdb.execute("set $eax = " + str(index))

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

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

OnBreakpoint("*0x004018C0", callback_break_loop)
OnBreakpoint("*0x00401DB4", callback_update_index)
OnBreakpoint("*0x00401FB6", callback_check_file)
```



Flag : Cryptol0gists_h8_him@flare-on.com