



FLARE-ON CHALLENGE 9 SOLUTION
BY MORITZ RAABE (@M_R_TZ)

Challenge 5: T8

Challenge Prompt

FLARE FACT #823: Studies show that C++ Reversers have fewer friends on average than normal people do. That's why you're here, reversing this, instead of with them, because they don't exist.

We've found an unknown executable on one of our hosts.

The file has been there for a while, but our networking logs only show suspicious traffic on one day.

Can you tell us what happened?

7-zip password: flare

Solution

This challenge tests your ability to reverse engineer a small program written in C++. The challenge consists of an executable and a packet capture file. Overall, this sample requires you to reverse engineer C++ objects and functions, overcome some anti-analysis techniques, and understand networking, encryption and encoding.

This writeup focuses on the key components and does not describe every functionality in detail. The main analysis tools we use are IDA Pro, capa, FLOSS, and CyberChef – all run within [FLARE VM](#).

Basic Analysis

Looking at the program's strings we notice mangled function names like `?AV?Sctype@_W@std@@` that indicate C++ code. Besides this no useful strings seem to stand out. Running [FLOSS](#), we get two potentially interesting strings. One of them being flare-on.com.

To get a first idea of the file, we run [capa](#). capa skips about 1,000 runtime and library functions (53% of all identified functions) in this binary. The capa results are shown in [Figure 1](#).

CAPABILITY	NAMESPACE
get geographical location	collection
initialize WinHTTP library	communication/http
prepare HTTP request	communication/http/client
receive HTTP response	communication/http/client
encode data using Base64 (2 matches)	data-manipulation/encoding/base64
reference Base64 string	data-manipulation/encoding/base64
encode data using XOR	data-manipulation/encoding/xor
encrypt data using RC4 KSA	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
hash data with MD5	data-manipulation/hashing/md5
contain a resource (.rsrc) section	executable/pe/section/rsrc
print debug messages (2 matches)	host-interaction/log/debug/write-event
allocate RWX memory	host-interaction/process/inject

Figure 1: capa results for the challenge binary

capa identifies various interesting program capabilities – including communication, encoding, and encryption. Before we investigate these in the disassembled file, we start the program and observe its run-time activities. For my execution I did not get any useful results from dynamic analysis tools such as FakeNet-NG or Process Monitor.

In the packet capture file, we notice two HTTP POST requests and responses. The HTTP data appears to be Base64 encoded, however decoding it does not result in any useful data.

Advanced Analysis

We pop open IDA Pro to investigate how the data is encoded. To find the interesting code sequences we use one of the verbose capa output modes (-v or -vv) or the [capa explorer IDAPython plugin](#).

capa identifies two functions related to Base64 encoding. The one at 0x4014C0 contains the standard Base64 characters ABC... and is the Base64 encoding function. This function is called once in the program, namely in the function starting at address 0x403C20. This function is only referenced once from the .rdata section. A few lines above we see IDA Pro's helpful annotation that indicates the start of a virtual function table. To understand where and how the function is used, we'll have to do a little more work.

Dealing with virtual function calls

We first trace back to where the virtual function table (vftable) is used and end up in the constructor of the CClientSock class (the second function referencing the vftable is the destructor [it contains various calls to free]).

The program calls the constructor once. Just before this, it allocates 0x4C (76) bytes of memory using the new operator. Based on the known size and the referenced offsets in the constructor we create and apply a struct definition like shown in [Figure 2](#).

```

mov    [ebp+var_4], 0
lea    edx, [esi+CClientSock.field_14]
xor    eax, eax
mov    [esi+CClientSock.vftable], offset const CClientSock::`vftable'
mov    [esi+CClientSock.field_4], ax
xorps xmm0, xmm0
movq   qword ptr [esi+6], xmm0
mov    [esi+CClientSock.field_E], eax
mov    [esi+CClientSock.field_12], ax
mov    [edx+10h], eax
mov    dword ptr [edx+14h], 7
mov    [edx], ax
mov    [esi+CClientSock.field_3C], eax
mov    dword ptr [esi+40h], 7
mov    [esi+CClientSock.field_2C], ax

```

Figure 2: Excerpt of CClientSock fields in the constructor

At the vftable offset 0x44B918 we see twelve function pointers and create the according struct as shown in [Figure 3](#). To ease later navigation, we add the respective function offsets as repeatable comments (IDA shortcut ; [semicolon]).

```
CClientSock_vftable struct ; (sizeof=0x30, mappedt
field_0          dd ? ; 4035F0
field_4          dd ? ; 403770
field_8          dd ? ; 4037A0
field_C          dd ? ; 403C20
field_10         dd ? ; 403CE0
field_14         dd ? ; 4036D0
field_18         dd ? ; 403860
field_1C         dd ? ; 403D70
field_20         dd ? ; 404200
field_24         dd ? ; 4043F0
field_28         dd ? ; 403910
field_2C         dd ? ; 405530
CClientSock_vftable ends
```

Figure 3: CClientSock vftable struct

[Figure 4](#) shows the CClientSock constructor call and subsequent code. Applying the structures like shown here, we can now follow the virtual function calls.

```
mov    ecx, esi
call   CClientSock__ctor
; } // starts at 4047BA
; try {
mov    byte ptr [ebp+var_4], 0
lea    ecx, [ebp+var_20]
movq   xmm0, ds:qword_44B8A0
mov    edi, eax
mov    ax, ds:word_44B8A8
movq   [ebp+var_20], xmm0
xor    word ptr [ebp+var_20], 45h ; 'E'
xor    word ptr [ebp+var_20+2], 45h ; 'E'
xor    word ptr [ebp+var_20+4], 45h ; 'E'
xor    word ptr [ebp+var_20+6], 45h ; 'E'
mov    [ebp+var_18], ax
mov    al, ds:byte_44B8AA
mov    [ebp+var_16], al
mov    [ebp+var_15], 0
mov    [ebp+var_11], 0
mov    eax, [edi+CClientSock.vftable]
push   ecx
mov    ecx, edi
mov    [ebp+var_9C], edi
call   [eax+CClientSock_vftable.field 4] ; 403770
```

Figure 4: CClientSock constructor call and subsequent code

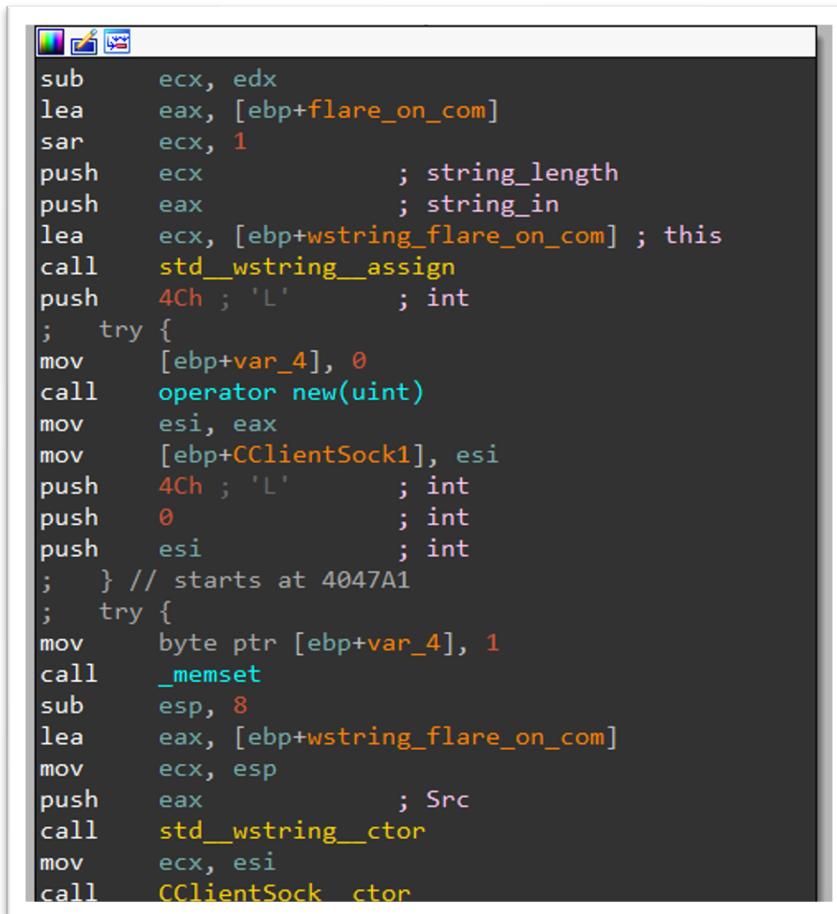
The called function located at 0x403770 sets a wide character string in the object. The string is XOR decoded with the byte 0x45 just before the virtual function call. The decoded string is POST. A little further above, FLOSS identified the flare-on.com string. Tracking its usage, we see that this host string gets passed to a function called briefly before the CClientSock constructor. Annoyingly, we often must recover string functions in C++ binaries and this sample is no exception.

Recognizing string functions

Frequently, we can recognize string functions based on the referenced offsets and called subfunctions. The relevant offsets for wide character strings can be deduced from the structure definition shown below. Offset 0 stores the string data (for strings up to 7 characters long) or a pointer to the string on the heap. Offset 4 stores the string size/length and offset 8 stores the reserved capacity of a string.

```
struct std::wstring
{
    wchar_t chars_or_pheap[8];
    int size;
    int reserved;
};
```

With the offsets in mind, we recover the assign and constructor functions as shown in [Figure 5](#).



The screenshot shows assembly code in a debugger window. The code is as follows:

```

sub    ecx, edx
lea    eax, [ebp+flare_on_com]
sar    ecx, 1
push   ecx          ; string_length
push   eax          ; string_in
lea    ecx, [ebp+wstring_flare_on_com] ; this
call   std_wstring_assign
push   4Ch ; 'L'      ; int
; try {
mov   [ebp+var_4], 0
call   operator new(uint)
mov   esi, eax
mov   [ebp+CClientSock1], esi
push   4Ch ; 'L'      ; int
push   0           ; int
push   esi          ; int
; } // starts at 4047A1
; try {
mov   byte ptr [ebp+var_4], 1
call   _memset
sub   esp, 8
lea    eax, [ebp+wstring_flare_on_com]
mov   ecx, esp
push   eax          ; Src
call   std_wstring_ctor
mov   ecx, esi
call   CClientSock_ctor

```

Figure 5: std::wstring::assign and std::wstring::wstring calls before CClientSock constructor call

The wide string (UTF-16LE encoded) flare-on.com is passed into the CClientSock constructor and assigned to offset 0x14.

Identifying core functionality

The main routine references various global offsets. Tracing back to the offset references leads us to a function that calls GetLocalTime as well as srand and rand. This function is called as part of the program initialization – it executes before the main function. It looks like this creates a global object.

Using static analysis or with the help of a debugger we see that in the main function the string F09 and a randomly created number are concatenated. The resulting string is passed to another virtual function call that calculates the MD5 hash (easy to see via the capa hit) and assigns it to a CClientSock field at offset 0x2C.

Further following the virtual function calls we identify usage of the WinHTTP API in the function at 0x403D70. We skip over the details here, but the program constructs a User-Agent string containing the random number, connects to the configured host (at offset 0x14, flare-on.com), RC4 encrypts and Base64 encodes the input data and sends it in an HTTP POST request. The program then receives HTTP data back and stores it into the CClientSock object at offset 0x44.

In the function at 0x4043F0 the program Base64 and RC4 decrypts the received data. The RC4 decryption uses the previously configured MD5 hash (offset 0x2C) as a password. In the provided PCAP file the first HTTP request contains the number 11950 in the User-Agent string. The concatenated string F0911950 (UTF-16LE encoded) results in the MD5 hash a5c6993299429aa7b900211d4a279848.

We extract the data from the first HTTP response in the PCAP and decode it using CyberChef as shown in [Figure 6](#).

Input	Output
54645164425261316e78475530366462423274537535137544a322b6364377a73744c585251c 34c626d68326e5476446d3170354966542f4375304a7853686b3674485142525777506c6f397i 413164495366736c6b4c674744733431574b313269625749666c714c45345971334f5949456e c46a775648726a4c3255344c75336d732d485163346e664d57585067634f48623466686f6b6l 39332f414a643547547543357a2b3459736d675268315a393879696e4c424b422b666d475579i 1675436676f6e2f4b486d4a647641f51386e416e6c38402f3058472b38745951625a5277675i 36744876767066796e394f48379756374352f634f69384b5767414c76564851576166727038: 1422f4a74542b74357a6d6e657a516c70337a504c34736a3243a666355544b35636f70625a4: 794865785644346a4a4e2b4c657a4a45747244585031444a4e673d3d	00000000 e5 07 09 00 03 00 0f 00 0d 00 25 00 03 00 62 02 å.....%...b. 00000010 2c 00 dc 07 0a 00 06 00 0d 00 25 00 09 00 ,.,U.....%... 00000020 2a 03 2c 00 e1 07 0c 00 04 00 07 00 0d 00 25 00 *,.,å.....%... 00000030 24 00 e5 00 2c 00 e0 07 05 00 05 00 06 00 d0 00 \$.å..,å..... 00000040 25 00 0b 00 26 00 2c 00 e2 07 0a 00 01 00 08 00 %,.,&.,å..... 00000050 0d 00 25 00 1f 00 45 03 2c 00 e6 07 03 00 02 00 ..%,.,E.,æ..... 00000060 01 00 0d 00 25 00 32 00 da 00 2c 00 de 07 07 00 2.Ú.,P... 00000070 02 00 16 00 0d 00 25 00 36 00 d1 02 2c 00 de 07 %,6.N.,P... 00000080 05 00 03 00 0e 00 0d 00 25 00 01 00 e8 00 2c 00 %,è., 00000090 da 07 04 00 01 00 05 00 0d 00 25 00 3a 00 0b 00 Ú.....%... 000000a0 2c 00 dd 07 0a 00 04 00 03 00 0d 00 25 00 16 00 .,V.....%... 000000b0 16 03 2c 00 de 07 01 00 02 00 0e 00 0d 00 25 00 ...,P.....%... 000000c0 10 00 c9 00 2c 00 dc 07 0c 00 01 00 0a 00 0d 00 ..É.,,U..... 000000d0 25 00 30 00 0c 02 2c 00 e6 07 02 00 01 00 1c 00 %,0...,æ..... 000000e0 0d 00 25 00 22 00 4b 01 2c 00 e6 07 09 00 05 00 ..%,.,K.,æ..... 000000f0 99 00 ad aa 75 aa 71 aa fd a1 %,!,m.

Figure 6: Decoding the data of the first HTTP response

After decoding the received data, the program uses the wcstok_s function to identify tokens separated by comma (,) character – note the UTF16-LE encoding (\x2c\x00). Here, this results in fourteen 16-byte tokens as shown in [Figure 7](#).

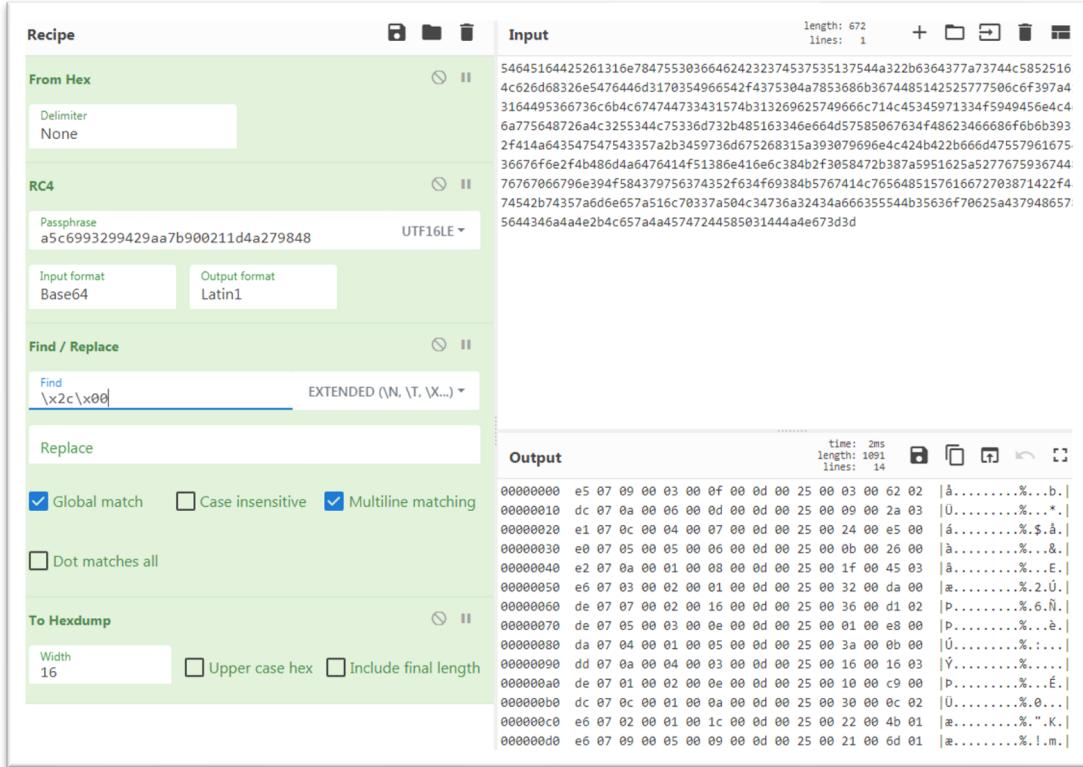


Figure 7: Decoded data split on UTF16-LE encoded comma character

Each 16-byte token is passed to the function at 0x404570 in a loop. The function uses a few unusual instructions (at least in a malware analysis context) and several hard-coded constants to calculate a value¹. To avoid understanding the calculations in detail, we can use the debugger to calculate the results. Alternatively, we can reimplement the function. Reimplementing is easier to do from the decompilation output shown in [Figure 8](#).

¹ Bonus points if you can identify what value this algorithm computes.

```

int __caec1_sub_404570(unsigned int a1, unsigned int a2)

{
    unsigned int v2; // ecx
    int v3; // esi
    unsigned int v4; // eax
    float v5; // xmm0_4
    float v7; // [esp+2Ch] [ebp+14h]

    v2 = HIWORD(a1);
    v3 = (unsigned __int16)a1 - 1;
    if ( HIWORD(a1) > 2u )
        v3 = (unsigned __int16)a1;
    v4 = v2 + 12;
    if ( v2 > 2 )
        v4 = HIWORD(a1);
    v7 = (float)((float)((double)(int)(v3 / 100 / 4
                                + HIWORD(a2)
                                + (int)((double)(v3 + 4716) * 365.25)
                                - (int)((double)(int)(v4 + 1) * -30.6001)
                                - v3 / 100
                                + 2)
                                - 1524.5)
                                - 2451549.5)
                                / 29.53;
    v5 = floor(v7);
    return (int)roundf((float)(v7 - v5) * 29.53);
}

```

Figure 8: Decompiled function that calculates a value

Each calculated value is passed to the function at 0x4041E0 which converts it to a character. We can again leverage the debugger or reimplement this function from the disassembly or decompilation.

The decoded raw bytes, intermediate values, and resulting characters are shown in [Figure 9](#).

Raw bytes	-> value	-> character
e5 07 09 00 03 00 0f 00 0d 00 25 00 03 00 62 02	-> 9	-> i
dc 07 0a 00 06 00 0d 00 0d 00 25 00 09 00 2a 03	-> 28	-> _
e1 07 0c 00 04 00 07 00 0d 00 25 00 24 00 e5 00	-> 19	-> s
e0 07 05 00 05 00 06 00 0d 00 25 00 0b 00 26 00	-> 29	-> 3
e2 07 0a 00 01 00 08 00 0d 00 25 00 1f 00 45 03	-> 29	-> 3
e6 07 03 00 02 00 01 00 0d 00 25 00 32 00 da 00	-> 28	-> _
de 07 07 00 02 00 16 00 0d 00 25 00 36 00 d1 02	-> 25	-> y
de 07 05 00 03 00 0e 00 0d 00 25 00 01 00 e8 00	-> 15	-> o
da 07 04 00 01 00 05 00 0d 00 25 00 3a 00 0b 00	-> 21	-> u
dd 07 0a 00 04 00 03 00 0d 00 25 00 16 00 16 03	-> 28	-> _
de 07 01 00 02 00 0e 00 0d 00 25 00 10 00 c9 00	-> 13	-> m
dc 07 0c 00 01 00 0a 00 0d 00 25 00 30 00 0c 02	-> 27	-> 0
e6 07 02 00 01 00 1c 00 0d 00 25 00 22 00 4b 01	-> 27	-> 0
e6 07 09 00 05 00 09 00 0d 00 25 00 21 00 6d 01	-> 14	-> n

Figure 9: Decoded bytes, intermediate value, and resulting characters

Together the characters form the string `i_s33_you_m00n`. The main function concatenates this string with the @ character and the configured host flare-on.com. This results in the challenge flag:

<code>i_s33_you_m00n@flare-on.com</code>
--

CHALLENGE 5: T8 | FLARE-ON 9

For extra bonus points check out the following program functionality and decode the remaining network traffic. If you have any questions on this challenge or would like to learn more, please contact the challenge author directly, for example on Twitter.

CHALLENGE 5: T8 | FLARE-ON 9

©2021 Mandiant, Inc. All rights reserved. Mandiant is a registered trademark of Mandiant, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners.

MANDIANT®
NOW PART OF Google Cloud