FLARE-ON CHALLENGE 9 SOLUTION

BY PAUL TARTER (@HEFRPIDGE)

Challenge 6: à la mode

Challenge Prompt

FLARE FACT #824: Disregard flare fact #823 if you are a .NET Reverser too.

We will now reward your fantastic effort with a small binary challenge. You've earned it kid!

7-zip password: flare

Solution

This challenge was created from a malware sample I analyzed that led me to learn about the beautiful world of mixed-mode assemblies in .NET. The challenge creates a named pipe in unmanaged code and then connects to it with managed code. The managed code is straightforward but finding and analyzing the native code was the purpose of this challenge. Most samples I have analyzed that are mixed-mode use an empty class in .NET and have unmanaged code run in the background. This attempt is used to fool automated analysis and hide functionality from manual analysis. In this write-up while going over the challenge I will also give some education into what mixed-mode assemblies are, how they are built and how they execute.

What is Mixed Mode

No need for me to re-word the official documentation of lovely Microsoft so here is the big doc-dump

Mixed assemblies can contain both unmanaged machine instructions and MSIL instructions. This allows them to call and be called by .NET components, while retaining compatibility with native C++ libraries. Using mixed assemblies, developers can author applications using a mixture of .NET and native C++ code.

*For example, an existing library consisting entirely of native C++ code can be brought to the .NET platform by recompiling just one module with the **/clr** compiler switch. This module is then able to use .NET features but remains compatible with the remainder of the application. It is even possible to decide between managed and native compilation on a function-by-function basis within the same file*

There are multiple ways to incorporate managed code into an unmanaged project, some are just simple switches in a Visual Studio project. When approaching this simple method of implementation, the unmanaged code gets translated to managed code behind the scenes for you. This is very interesting but would have made this challenge much easier as all code is visible in dnSpy as seen in Figure 1, Figure 2, and Figure 3. Figure 1 is a *c++* source file that contains a call to both managed and unmanaged code, the most obvious are *printf* being a standard C function used in unmanaged code and *Console::WriteLine* being a managed method.

```
using namespace System;
#include <stdio.h>
#include "rc4.h"

unsigned char key[] = { 'A', 'B', 'C', 'D' };
unsigned char message[22] = {
    0x70, 0x01, 0x96, 0xDB, 0x60, 0xF4, 0x4C, 0x2F,
    0xAF, 0x49, 0x46, 0xA1, 0xC9, 0xD6, 0xAA, 0x27,
    0x72, 0x2E, 0xB4, 0x5C, 0xE4, 0x67
};



int main(int argc, char* argv[])
{
    RC4_STATE rc4;

    rc4_init(&rc4, (unsigned char*)key, sizeof(key));
    rc4_crypt(&rc4, message, sizeof(message));

    printf("unmanaged  : %s\n", message);

    System::String^ managed_message =
    System::Runtime::InteropServices::Marshal::PtrToStringAnsi((IntPtr)(char*)message);
    Console::WriteLine("managed    : {0}", managed_message);

    return 0;
}
```

Figure 1: main.cpp

```
#include "rc4.h"

void rc4_init(PRC4_STATE s, unsigned char* key, int length)
{
    int i, j, k, * m, a;

    s->x = 0;
    s->y = 0;
    m = s->m;

    for (i = 0; i < 256; i++)
    {
        m[i] = i;
    }

    j = k = 0;

    for (i = 0; i < 256; i++)
    {
        a = m[i];
        j = (unsigned char)(j + a + key[k]);
        m[i] = m[j]; m[j] = a;
        if (++k >= length) k = 0;
    }
}
```

Figure 2: rc4.cpp

```
// Token: 0x06000001 RID: 1 RVA: 0x00001090 File Offset: 0x00000490
internal unsafe static int main(int argc, sbyte** argv)
{
    RC4_STATE_ rc4_STATE_;
    <Module>.rc4_init(&rc4_STATE_, (byte*)(&<Module>.key), 4);
    <Module>.rc4_crypt(&rc4_STATE_, (byte*)(&<Module>.message), 22);
    <Module>.printf(ref <Module>.??_C@_0BB@HBDIFFAF@unmanaged?5?5?3?5?$CFs?6@, ref <Module>.message);
    string arg = Marshal.PtrToStringAnsi((IntPtr)((void*)(&<Module>.message)));
    Console.WriteLine("managed    : {0}", arg);
    return 0;
}

// Token: 0x06000002 RID: 2 RVA: 0x00001178 File Offset: 0x00000578
internal unsafe static void rc4_init(RC4_STATE_* s, byte* key, int length)
{
    *(int*)s = 0;
    *(int*)(s + 4L / (long)sizeof(RC4_STATE_)) = 0;
    int* ptr = (int*)(s + 8L / (long)sizeof(RC4_STATE_));
    int num = 0;
    int* ptr2 = ptr;
    do
    {
        *ptr2 = num;
        num++;
        ptr2 += 4L / 4L;
    }
    while (num < 256);
    int num2 = 0;
    long num3 = 0L;
    long num4 = 0L;
    long num5 = (long)length;
    do
    {
        int* ptr3 = num3 * 4L / 4L + ptr;
        int num6 = *ptr3;
        num2 = (int)((byte)((int)num4[key] + num6 + num2));
        int* ptr4 = (long)num2 * 4L + ptr / 4;
        *ptr3 = *ptr4;
        *ptr4 = num6;
        num4 += 1L;
        num4 = ((num4 >= num5) ? 0L : num4);
        num3 += 1L;
    }
    while (num3 < 256L);
}
```

Figure 3: dnSpy representation

Headers and Flags

A question one might have, while building the sample is, how does the code get executed? A humorous flag, ***/ijw***, that gets passed to the linker when building a mixed-mode assembly stands for *It Just Works.* While this is true, it is always nice for us as reverse engineers to get a view into how this craziness just works. I don't intend to go into heavy depth on this subject but show some ways that this is viewable to us as reverse engineers and things that are useful to understand.

First, inspecting the PE Headers, the .NET directory is found in a directory commonly called the *.NET Directory* which is in the data directory index, *14*, defined as *IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR.* This entry will point to the IMAGE_COR20_HEADER structure. To quote from the almighty useful MSDN docs and a classic article once again, (2019)

> *Executables produced for the Microsoft .NET environment are first and foremost PE files. However, in most cases normal code and data in a .NET file are minimal. The primary purpose of a .NET executable is to get the .NET-*

specific information such as metadata and intermediate language (IL) into memory. In addition, a .NET executable links against MSCOREE.DLL. This DLL is the starting point for a .NET process. When a .NET executable loads, its entry point is usually a tiny stub of code. That stub just jumps to an exported function in MSCOREE.DLL (_CorExeMain or _CorDllMain). From there, MSCOREE takes charge, and starts using the metadata and IL from the executable file. This setup is similar to the way apps in Visual Basic (prior to .NET) used MSVBVM60.DLL. The starting point for .NET information is the IMAGE_COR20_HEADER structure, currently defined in CorHDR.H from the .NET Framework SDK and more recent versions of WINNT.H. The IMAGE_COR20_HEADER is pointed to by the IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR entry in the DataDirectory.

```
// CLR 2.0 header structure.
typedef struct IMAGE_COR20_HEADER
{
    // Header versioning
    DWORD                   cb;
    WORD                    MajorRuntimeVersion;
    WORD                    MinorRuntimeVersion;

    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY    MetaData;
    DWORD                   Flags;

    // If COMIMAGE_FLAGS_NATIVE_ENTRYPOINT is not set, EntryPointToken represents a
       managed entrypoint.
    // If COMIMAGE_FLAGS_NATIVE_ENTRYPOINT is set, EntryPointRVA represents an RVA
       to a native entrypoint.
    union {
        DWORD               EntryPointToken;
        DWORD               EntryPointRVA;
    } DUMMYUNIONNAME;

    // Binding information
    IMAGE_DATA_DIRECTORY    Resources;
    IMAGE_DATA_DIRECTORY    StrongNameSignature;

    // Regular fixup and binding information
    IMAGE_DATA_DIRECTORY    CodeManagerTable;
    IMAGE_DATA_DIRECTORY    VTableFixups;
    IMAGE_DATA_DIRECTORY    ExportAddressTableJumps;

    // Precompiled image info (internal use only - set to zero)
    IMAGE_DATA_DIRECTORY    ManagedNativeHeader;

} IMAGE_COR20_HEADER, *PIMAGE_COR20_HEADER;
```

Figure 4: IMAGE_COR20_HEADER structure

As noted in the comments in Figure 4, the *Flags* field has a constant *COMIMAGE_FLAGS_NATIVE_ENTRYPOINT*. This states whether the EntryPointRVA is to be interpreted as a managed entrypoint or a native entrypoint. The sample given above has a Cor20 Header that parsed by dnSpy looks like the Figure 5.

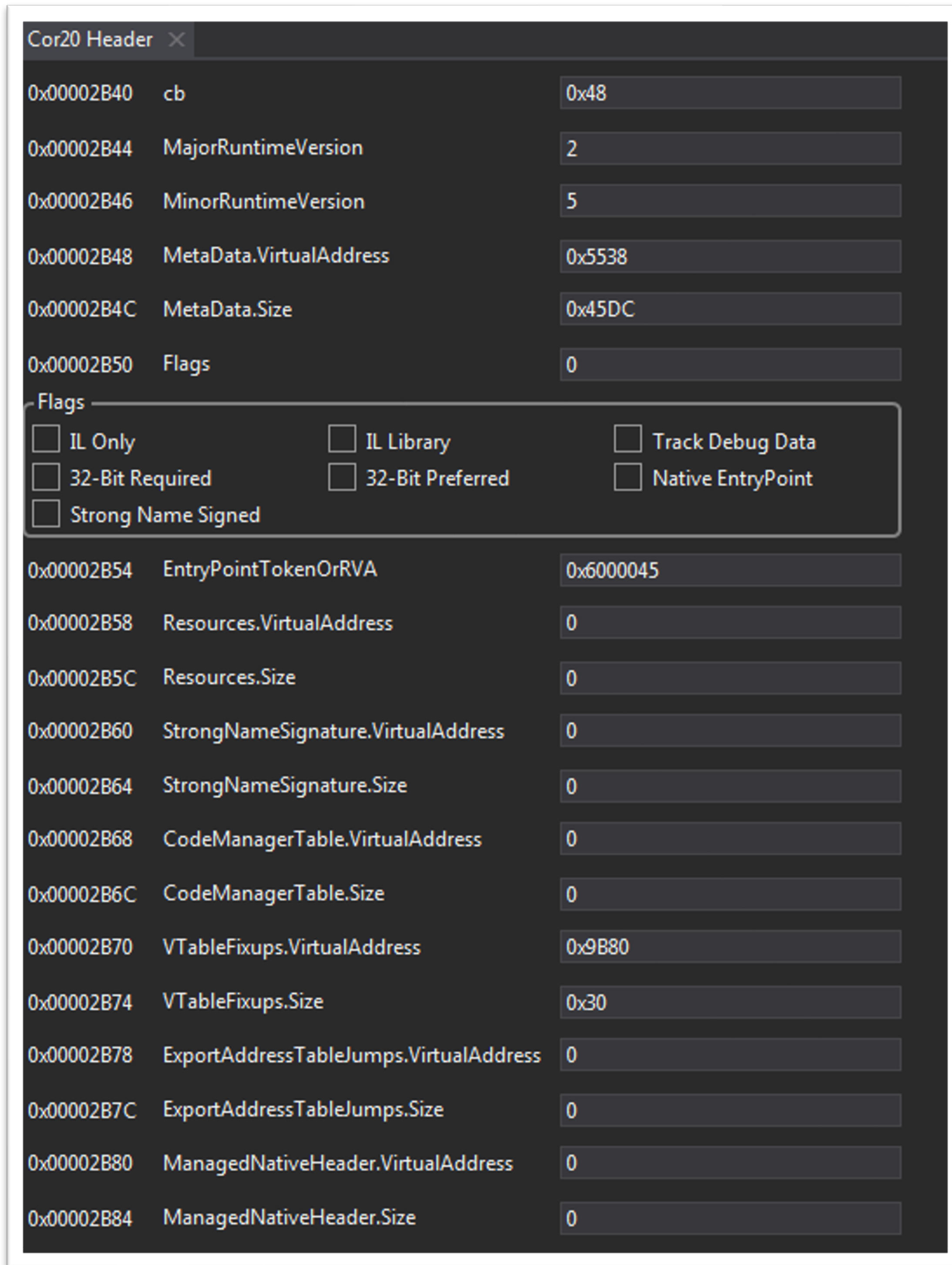

Figure 5: Cor20 Header in dnSpy

The *Native EntryPoint* flag is clear and contains a token value 0x06000045, which immediately calls unmanaged code:

```
// Token: 0x06000045 RID: 69 RVA: 0x000015AC File Offset: 0x000009AC
[SuppressUnmanagedCodeSecurity]
```

MANDIANT PROPRIETARY AND CONFIDENTIAL

```
[MethodImpl(MethodImplOptions.Unmanaged | MethodImplOptions.PreserveSig)]
internal static extern uint mainCRTStartup();
```

Figure 6: Managed EntryPoint

This sample is a mixed-mode sample but doesn't have a native entry point and therefore needs the CRT initialized so .NET makes sure the CRT is the first thing that gets called and then calls main. This is good that this happens because as already seen in *Figure 1* main calls *printf*. How does *mainCRTStartup* get back to main Token *0x06000001*? The answer is shown in *Figure 7*, the interesting note is that when debugging this sample, the global variable *_mep_main* gets modified while loading and changed from *0x06000001* to an executable address. This code will eventually call token main 0x06000001.

```
int __cdecl main_0(int argc, const char **argv, const char **envp)
{
    return _mep_main(argc, argv, envp);
}

.data:000000014000B028  ; __int64 (__fastcall *_mep_main)(_QWORD, _QWORD, _QWORD)
.data:000000014000B028 _mep_main dq 6000001h
```

Figure 7: mainCRTStartup Calling main()

That might have all seemed difficult to follow and jumping back and forth between managed and unmanaged can cause headaches. This previous example would have been much easier to solve in FlareOn because all the code that was relevant would be provided inside dnSpy. So how does one do a better job of hiding unmanaged code and not have it converted to ILCode?

Building Mixed-Mode to keep unmanaged code

Building a mixed-mode assembly with unmanaged code and guaranteeing it stayed unmanaged meant creating my own builder script. The simplified idea is to compile all unmanaged code with *cl.exe* and then compile all managed code with *csc.exe*. We are just compiling and not linking all the source code first and next the object files need to be linked together. With *.obj* files and *.netmodule* files all objects are linked together with *link.exe* with the magic flag */CLRIMAGETYPE:IJW.*

With the previous knowledge given one can now analyze This years FlareOn Challnge #6 with much more ease

Analyzing FlareOn Challenge 6: A La Mode

An initial triage of Challenge 6 quickly leads one to see that the sample is a .NET Assembly that has one class named Flag with an empty constructor and a method containing simple function named GetFlag as seen in *Figure 8* .

```
public class Flag
{
  // Token: 0x06000001 RID: 1 RVA: 0x0000D078 File Offset: 0x0000C478
  public string GetFlag(string password)
  {
    Decoder decoder = Encoding.UTF8.GetDecoder();
    UTF8Encoding utf8Encoding = new UTF8Encoding();
    string text = "";
    byte[] array = new byte[64];
    char[] array2 = new char[64];
    byte[] bytes = utf8Encoding.GetBytes(password + "\0");
    using (NamedPipeClientStream namedPipeClientStream = new
               NamedPipeClientStream(".", "FlareOn", PipeDirection.InOut))
    {
      namedPipeClientStream.Connect();
      namedPipeClientStream.ReadMode = PipeTransmissionMode.Message;
      namedPipeClientStream.Write(bytes, 0, Math.Min(bytes.Length, 64));
      int byteCount = namedPipeClientStream.Read(array, 0, array.Length);
      int chars = decoder.GetChars(array, 0, byteCount, array2, 0);
      text += new string(array2, 0, chars);
    }
    return text;
  }
}
```

Figure 8: GetFlag

There are a couple big flags that should raise awareness when triaging this sample. First, when looking at this in CFF explorer or most PE viewers it will be immediately obvious it is a .NET sample. In CFF Explorer the File Type is filled out at Portable Executable 32 .NET Assembly and there is a visible .NET Directory in the explorer tree as seen in *Figure 9*.

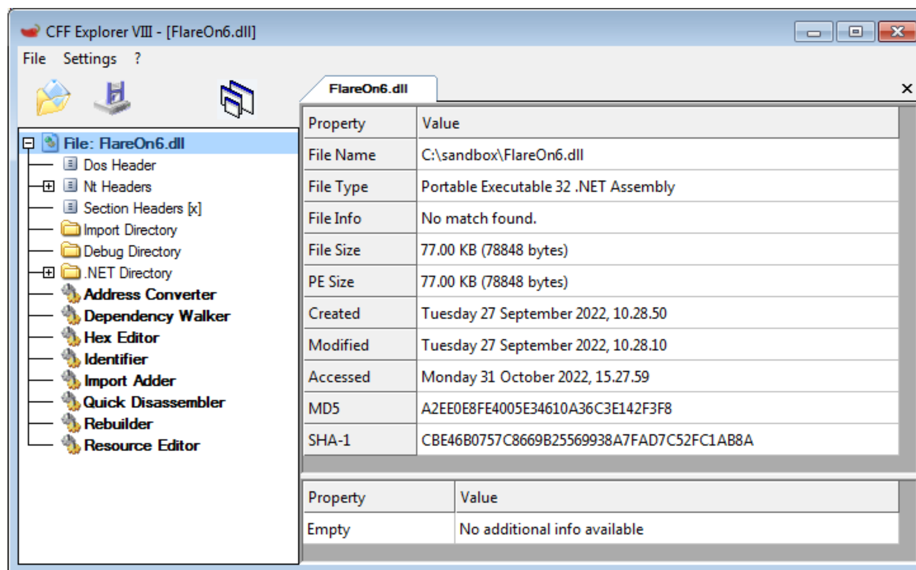

Figure 9: CFF Explorer FlareOn6 Triage

Some might at that point jump straight to dnSpy, a quick look at the import directory can give some interesting information. Generally, the only export you should see in an import directory for a .NET assembly is *mscoree.dll* In this challenge's case there is also kernel32.dll as an import, this is indicative of a mixed-mode sample. This can quickly be

confirmed by looking at the COR20 Header. The IL only flag is cleared stating it is a mixed-mode assembly and the Native EntryPoint flag is set. The Native EntryPoint is a different scenario than seen before and will be covered.

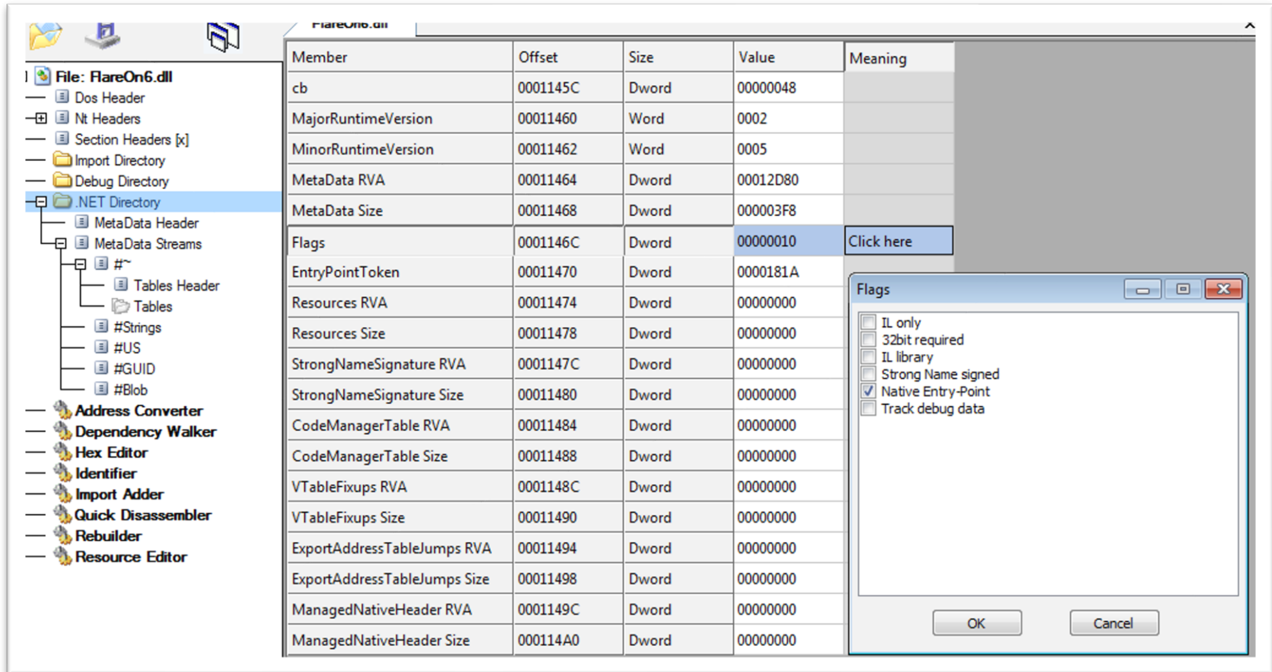

Figure 10: CFF Explorer Triage Mixed-Mode Flags

The same information is available in dnSpy by parsing the structures, but, dnSpy will also give a summary of the sample, this summary is very giving in information as seen in [Figure 11](#)

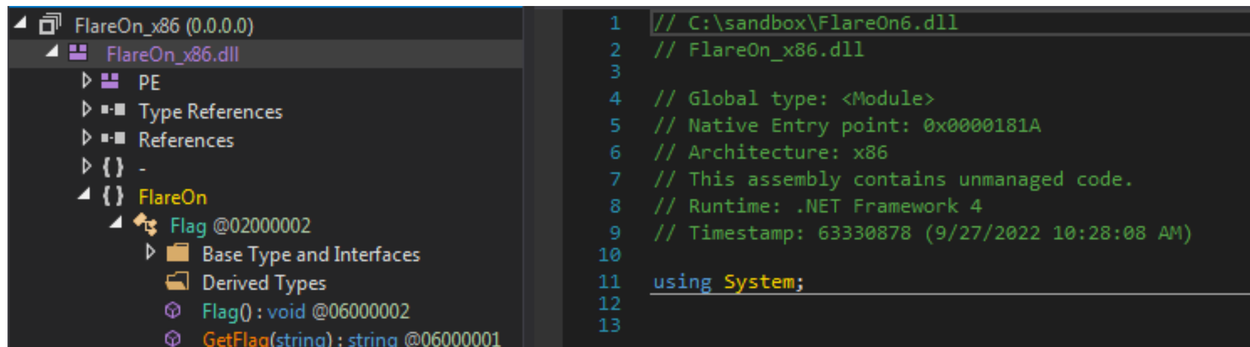

Figure 11: dnSpy File Summary

The real meat of this challenge is in the unmanaged code, as the managed code is just taking input and writing it to a pipe, reading back a response and printing it.

First a little sidetrack to learn a little bit more under the hood fun. How is this native entry getting called? What about the entry point that the Optional Header specifies? So as with the previous example analyzed this sample will have an AddressOfEntryPoint defined in the optional header, the challenge's entry point is *0xD16C* or ends up being loaded at *0x1000D16C*, *DllEntryPoint*. The *DllEntryPoint* immediately calls *mscoree!CorDllMain*. Open your trusty disassembler or decompiler of choice and find *CorDllMain* and it is an interesting experiment to trace through, as there are symbols and it isn't hard to see what is going on. One will see the header's flags being checked and loading the entry point accordingly, either calling *DllMain* or a managed token entry. This is how native code gets executed, and it quickly

fires off a thread in *DllMain* (classic malware, not following *DllMain* rules), then the sample's DllMain returns leaving it as a normal .NET assembly waiting to be used. *CreateThread* being called in *DLLMain* is mainly a no-no because the will not get kicked off until *DLLMain* exits, and synchronization should not take place during *DLLMain*, this challenge is okay with those caveats.

Analyzing the unmanaged code is as simple as opening it in your favorite disassembler / decompiler. With IDA Pro, ensure to change the Loader from *Microsoft.Net assembly [pe.dll]* to *Portable executable for 80386 (PE) [pe.dll]* as seen in Figure 12

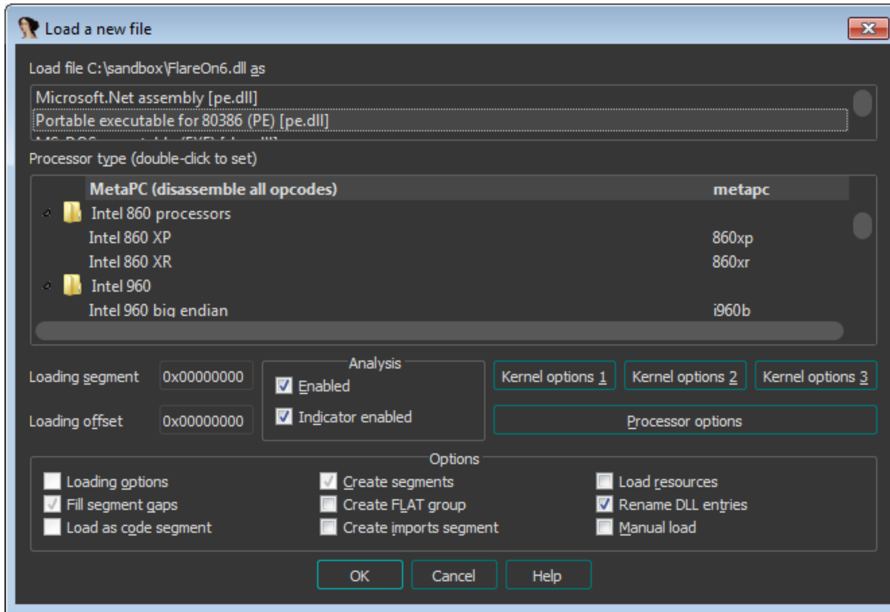

Figure 12: PE Loader in IDA Pro

Once opened the entry point can be obtained from the COR20 Header. In this case the EntryPointToken value is *0x181A* as seen in Figure 13, so the virtual address would be *0x1000181A*.

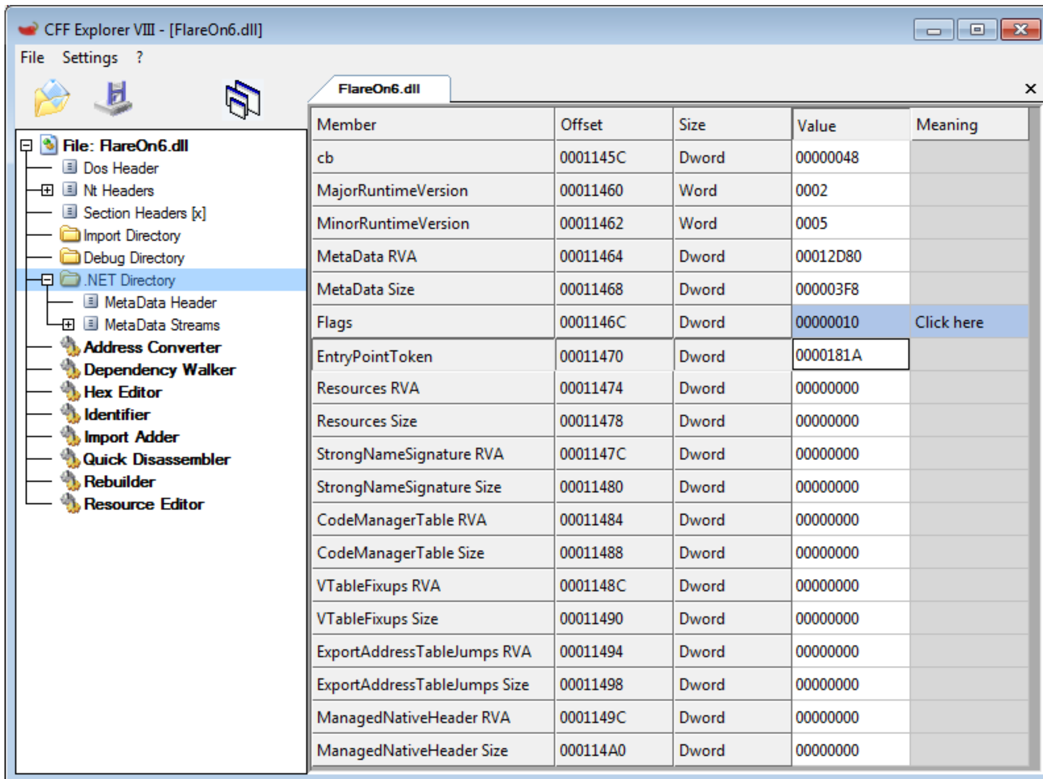

Figure 13: Unmanaged Entry Point

This code is not recognized by IDA when loading as a function, it is disassembled but ignored. This is most likely because it has no reference to it and isn't recognized as an entry point. Simply mark this as a function and it will start to look like a standard *dllmain_dispatch* function. Keep tracing through dispatch land until you get to *DllMain* as seen in *Figure 14*

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  if ( fdwReason == 1 )
  {
    sub_100012F1();
    dword_10015A30(0, 0, sub_10001094, 0, 0, 0);
  }
  return 1;
}
```

Figure 14: DllMain 0x10001163

There are two functions, one of them can be inspected the other is just a global variable, which would give the idea of dynamic API resolution for this sample. One might identify the second function as *CreateThread* or suspect it just by the prototype. Cross-referencing dword_10015a30 does show it being written to in function sub_100012f1 as seen in *Figure 15*

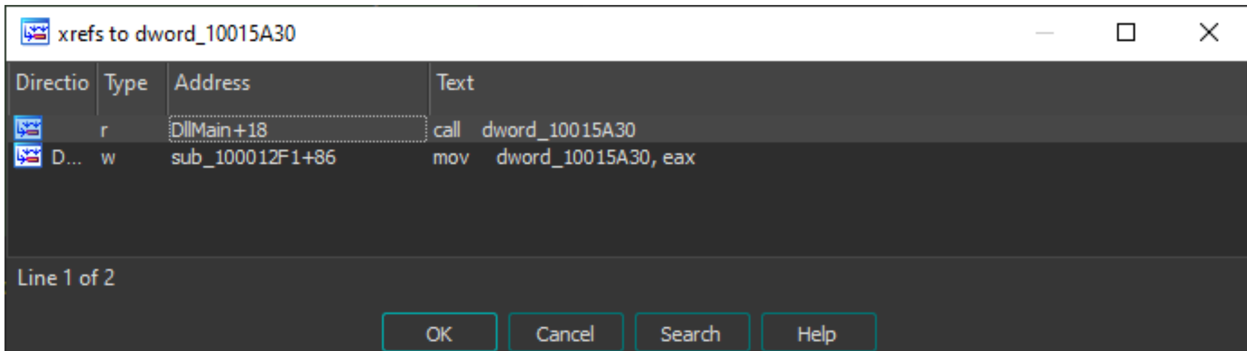

Figure 15: dynamic API CreateThread xref

The first function called in *DllMain* at address *0x100012F1* is a dynamic API resolver. The function dynamically resolves functions manually by re-implementing *GetProcAddress*. All the strings that are passed into *GetProcAddress* are single-byte xor encoded with the value *0x17*.

Some might have found that depending on how this sample was run it might not work properly. This was intended, there was no verification on the function at *0x100012DB* that performed parsing the PEB to find kernel32. When a .NET application is loaded the load order is different, mscoree.dll is loaded and puts kernel32.dll lower in the load order. If this challenge isn't loaded by a .NET application (as it should, it is a .NET assembly) then it will crash because it will resolve kernelbase.dll (which is next in line after kernel32.dll). This means if analyzed dynamically in x64dbg or something else that didn't load it properly as a .NET assembly the challenge would not work properly.

Some may have noticed a bug in which *CloseHandle* had an invalid string when xor-decoded and therefore did not resolve properly. This never causes an issue with the challenge and if it was any other API it would have been a lot more noticeable.

After resolving the APIs the sample is fairly straightforward for analyzing. The sample connects to a named pipe with the name *FlareOn* and then reads from the pipe up to 64 bytes. The string received over the pipe is validated by string comparison against a RC4 encrypted string. The cipher that is used to decrypt the string is then used without re-initializing to decrypt the flag. The password required to pass is *MyV0ic3!*

Solutions

Two simple solutions for solving this solution are: create a project that uses the DLL as intended or decrypt the bytes using your favorite method, I will show a Python script that accomplishes that.

Visual Studio Solution

By adding the DLL to a .NET C# Console Application as a reference once can use the challenge and call GetFlag as seen in *Figure 16*

```
using System;
using FlareOn;

namespace FlareOn_app
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Flag: {0}", new Flag().GetFlag("MyV0ic3!"));
        }
    }
}
```

Figure 16: Visual Studio Solution

Python Solution

It becomes more obvious that one really doesn't need to know the authentication password to solve this challenge, just find the decryption area and understand its logic. This is made obvious by looking at the Python solution in *Figure 17*

```
from Crypto.Cipher import ARC4

key = b'\x55\x8B\xEC\x83\xEC\x20\xEB\xFE'

password = b'\x3e\x39\x51\xfb\xa2\x11\xf7\xb9\x2c'

flag = b'\xE1\x60\xA1\x18\x93\x2E\x96\xAD'+\
       b'\x73\xBB\x4A\x92\xDE\x18\x0A\xAA'+\
       b'\x41\x74\xAD\xC0\x1D\x9F\x3F\x19'+\
       b'\xFF\x2B\x02\xDB\xD1\xCD\x1A'

cipher = ARC4.new(key)
print('Password: {}'.format(cipher.decrypt(password).decode()))
print('Flag:     {}'.format(cipher.decrypt(flag).decode()))
```

Figure 17: Python Solution

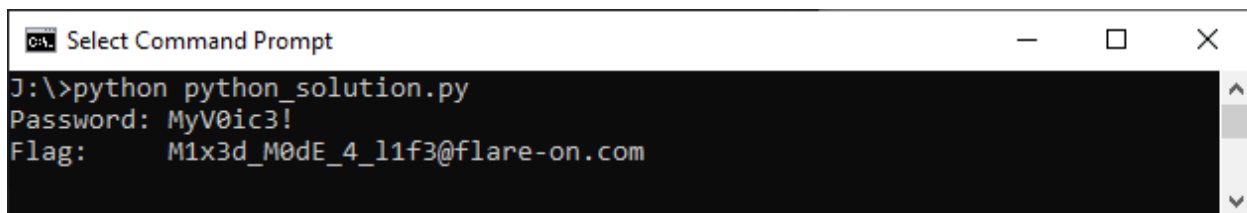

Figure 18: Python Solution

©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