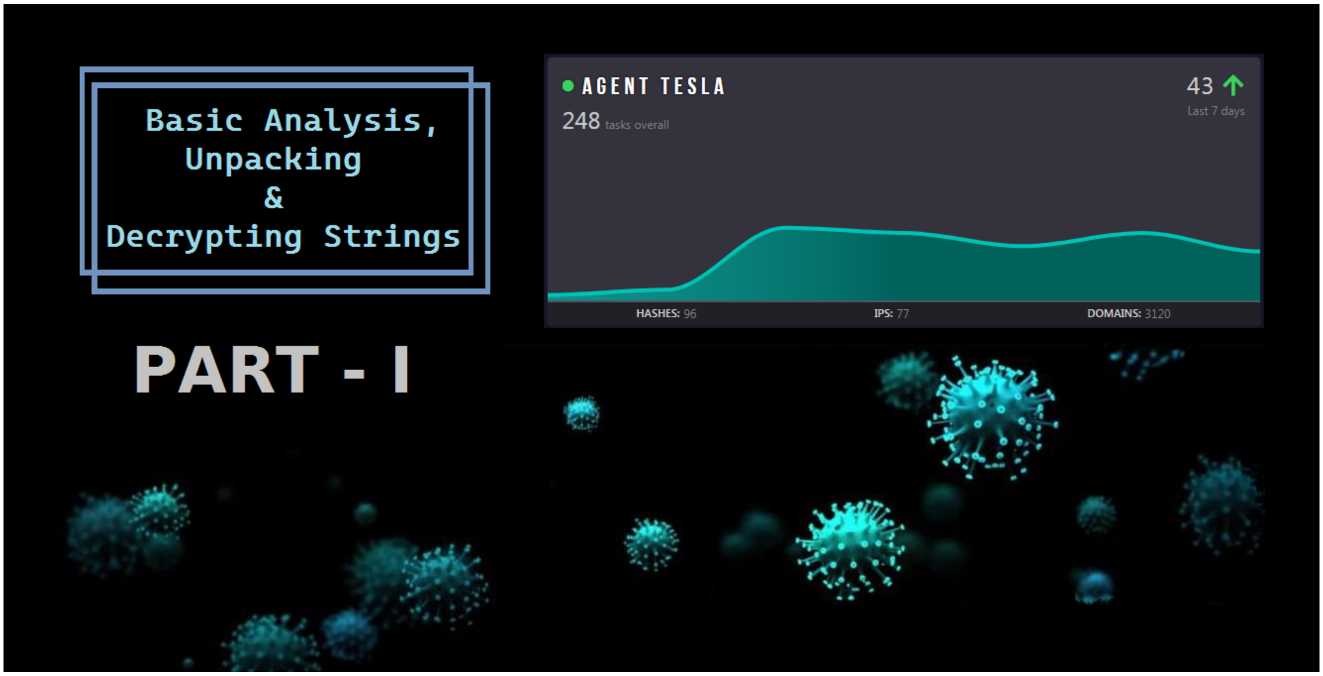
How Analysing an AgentTesla Could Lead To Attackers Inbox - Part I

mrt4ntr4.github.io/How-Analysing-an-AgentTesla-Could-Lead-To-Attackers-Inbox-1/

Suraj Malhotra April 13, 2020



If you'd read my previous articles I assured that I'll be releasing some article every week but now that seems nearly impossible due to some time constraints. I would have shared some things from my real life and new interesting security related things I come across but I don't think that will happen too coz I think it will decrease the quality of the blog somehow If i begin to post my random findings which may seem boring to some other readers.

What is the thing I love most about Security in general is the research part.. How we can get to real low level to find vulns. And this can happen only if we spend weeks.. maybe months reading and testing it out to give a detailed explanation.
Anyways if you have any suggestion/advice regarding this you can always comment and let me know.

Introduction

So as I promised previously this one is going to be .NET.
PS This is my first post on analysing a live malware sample and I'm not experienced in this field.

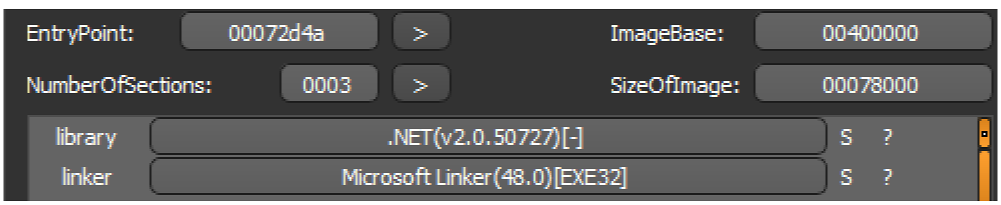
I know there are other blogposts on AgentTesla online but I didn't find them as detailed as this one's going to be.
And Yeah I also know the title seems to be a clickbait but its true XD
I have divided it into 2 parts and they have around 70+ screenshots as I believe in the fact that *Pictures are louder than words* :)

> This is a live malware and I don't want anyone to maliciously use the attacker's credentials, so obviously I would not give out this sample's hash and will be redacting a few things as well.

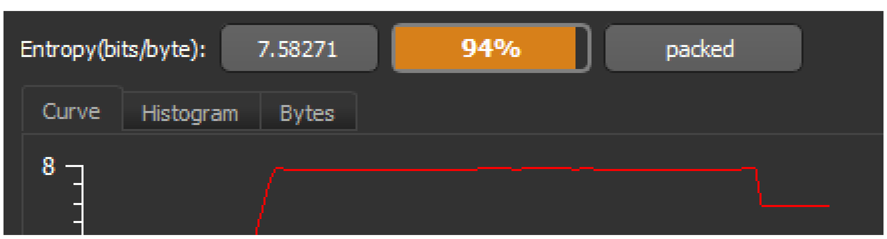
To start with, I found this sample on Malware Bazaar and it is tagged as a **COVID19** malware spead through spearphishing.
Luckily this sample doesn't have any anti-debug/vm techniques implemented. Also I've not setup my Sniffer VM with inetsim etc. I found it to be fileless. It has a Virustotal Score of 22/71 at time of writing this.

Static Properties Analysis

I started off with DIE and observed that its .NET based.



Also DIE shows that its Packed as its entropy is basically greater than 7.



Next we can check for some strings in the binary, ANSI doesn't show anything usually and its same in this case too. Observing the UNICODE strings it looks like this was basically based on a photo manager or something.

Unrecognized album file format	0000b578	0000003c
\Album	0000b5b6	0000000c
MyPhotos {0:#}.{1:#}	0000b5c4	00000028
{0} - MyPhotos {1:#}.{2:#}	0000b5ee	00000034
Open Album	0000b624	00000014
Album files (*.abm)|*.abm|All files (*.*)|*.*	0000b63a	0000005a
Add Photos	0000b696	00000014
Image Files (JPEG, GIF, BMP, etc.)|*.jpg;*.jpeg;*.gif;*.bmp;*.tif;*.t...	0000b6ad	00000204
D:\CurrentWork\Tmp	0000b8b3	00000024

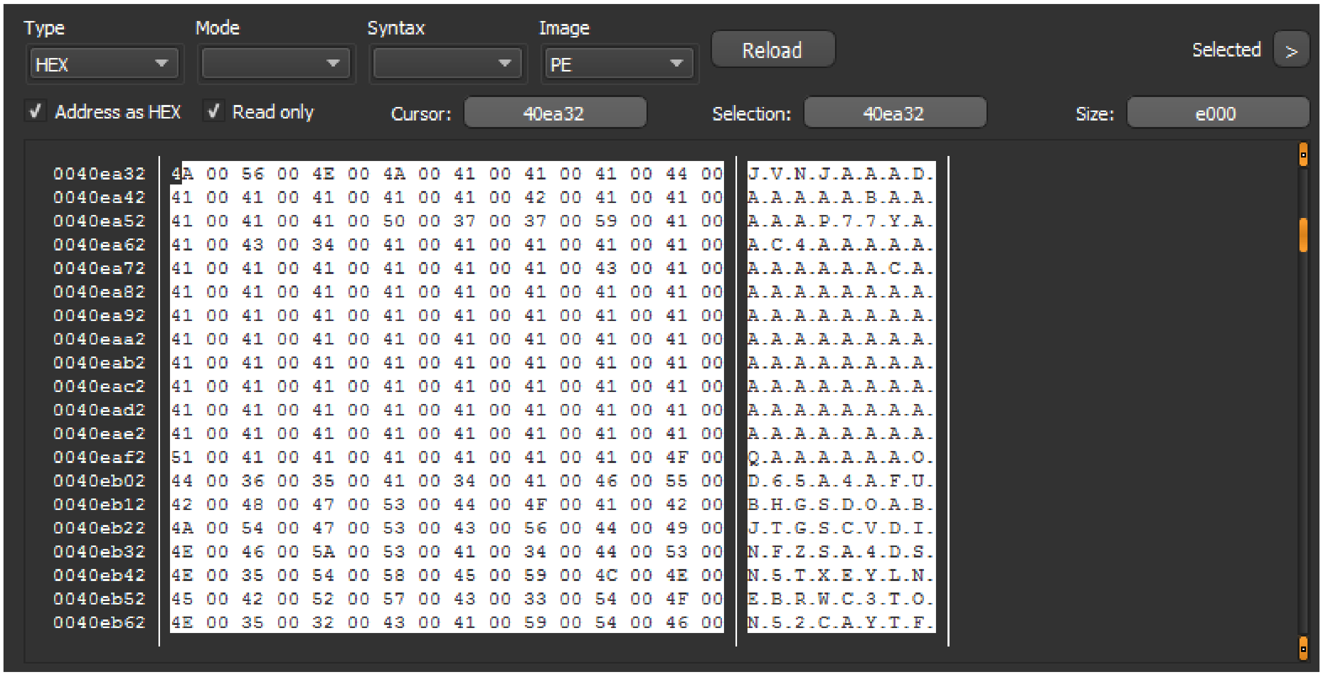
But wait if we scroll down we find something interesting…
Yeah It looks similar to base32 encoded string and below it we can see some Game related strings such as *frmGameOver*, *You win!*, etc.

ANSI
UNICODE
Crypto
Links

Size	Size	Size
Bettye Soderberg	0000cbc3	00000020
Tammera Strebel	0000cbe5	0000001e
ABHqTRJFnsWBEzLtXeCZ	0000cc05	00000028
JVNJAAADAAAAABAAAAAP77YAAC4AAAAAAAAAAACAAAAAAAAAA...	0000cc32	0000e000
_2048	0001ac3c	0000000a
plTitle	0001ac62	0000000e
lblInfo	0001ac72	0000000e
plSelect	0001acf2	00000010
btnExit	0001ad04	0000000e
btnAgain	0001ad1a	00000010
frmGameOver	0001ad36	00000016
You win!	0001ad4e	00000010

```
echo JVNJAA |
base32 -d
MZ▒base32: invalid
input
```

Cool It starts with the **MZ** Header and this confirms that its base32 encoded.
Unfortunately we can’t copy the whole string here but we can just view it in hexdump by right clicking it in DIE.

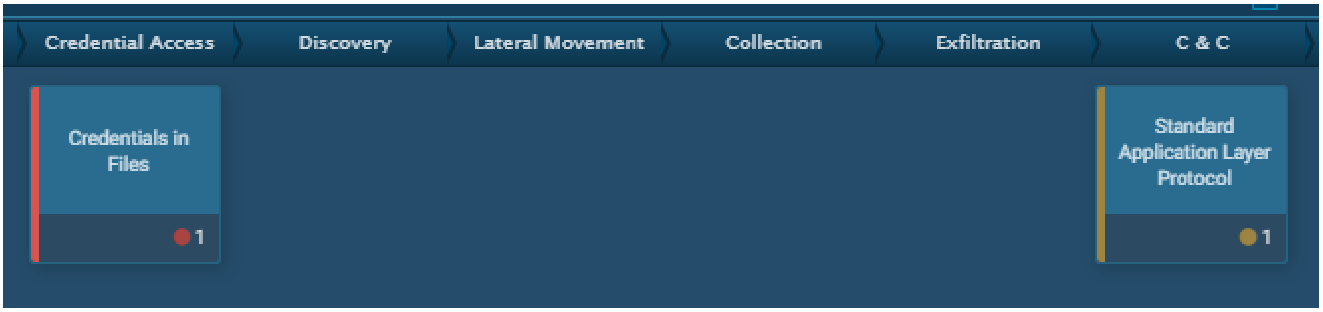


Behavioral Analysis

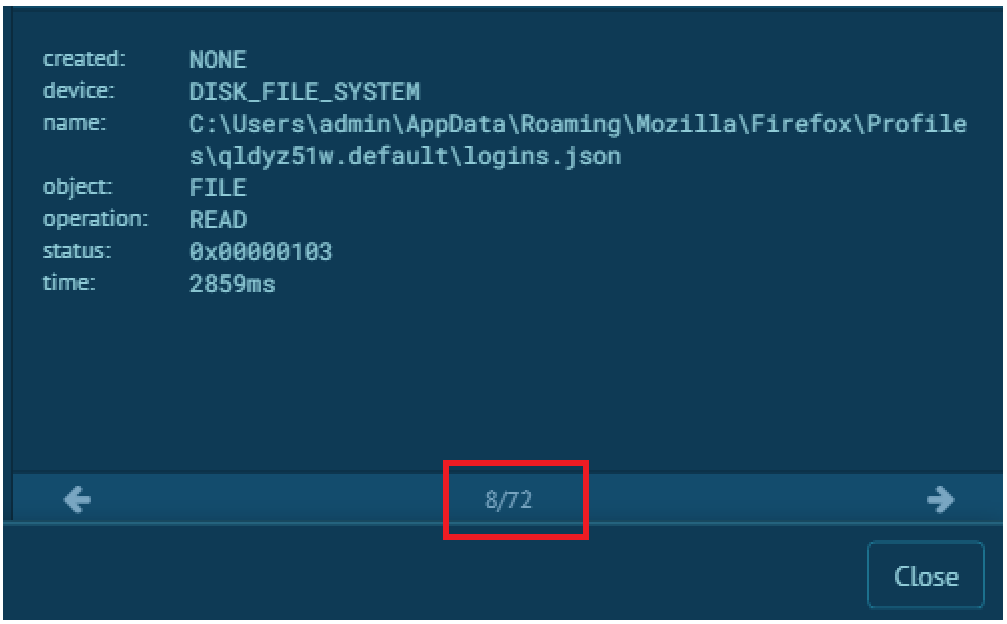
Now we have the coolest part of running it in a sandbox environment.

I used any.run and selected a Win7 32 bit VM (Basic plan) and noticed its execution. Hmmm.. So Its silent and doesn't do any activity on the screen.

So, Any.run has this feature of mapping MITRE Techniques it notices through the malware activity.

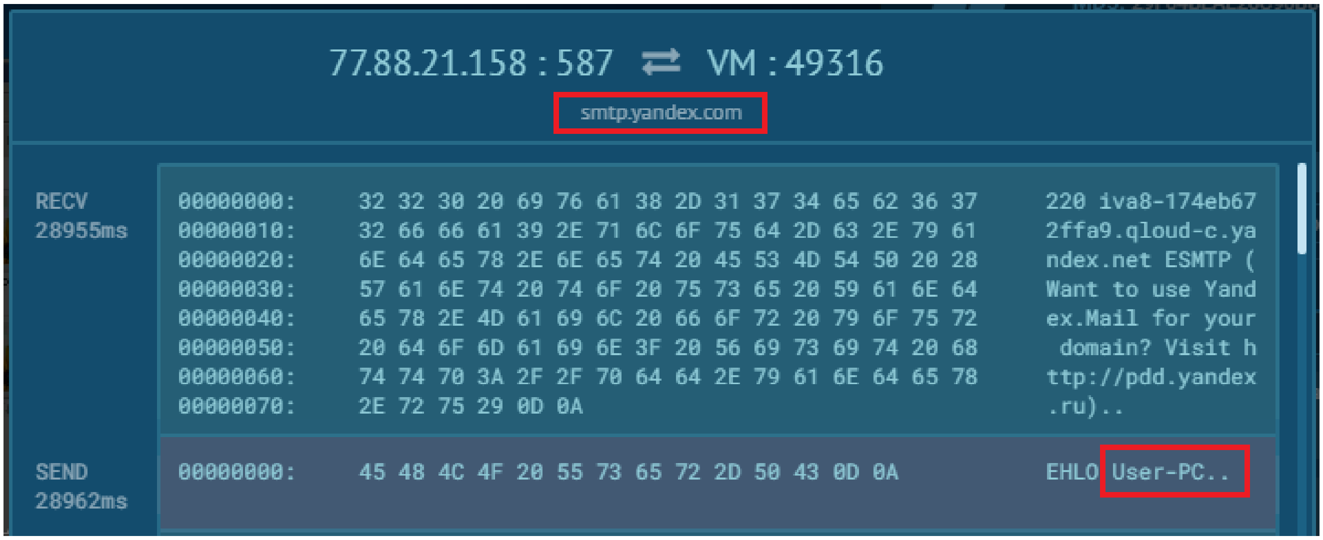


We observe that its basically a credential stealer and tries to communicate with a C&C server.



Woah!! It accesses over 72 files and is basically looking for browsers, ftp clients etc. So Any.run has 2 additional browsers I know of ie. Firefox & Opera. And Firefox for instance requires logins.json and key4.db for the passwords which it accesses obviously. [1]

We can also view the connection requests and looks like its sending data over smtp with *smtp.yandex.com* and sends some data which includes *User-PC*.



I also downloaded and analysed the pcap file from any.run but it doesn't look suspicious as I don't think the browsers in any.run had some saved passwords.

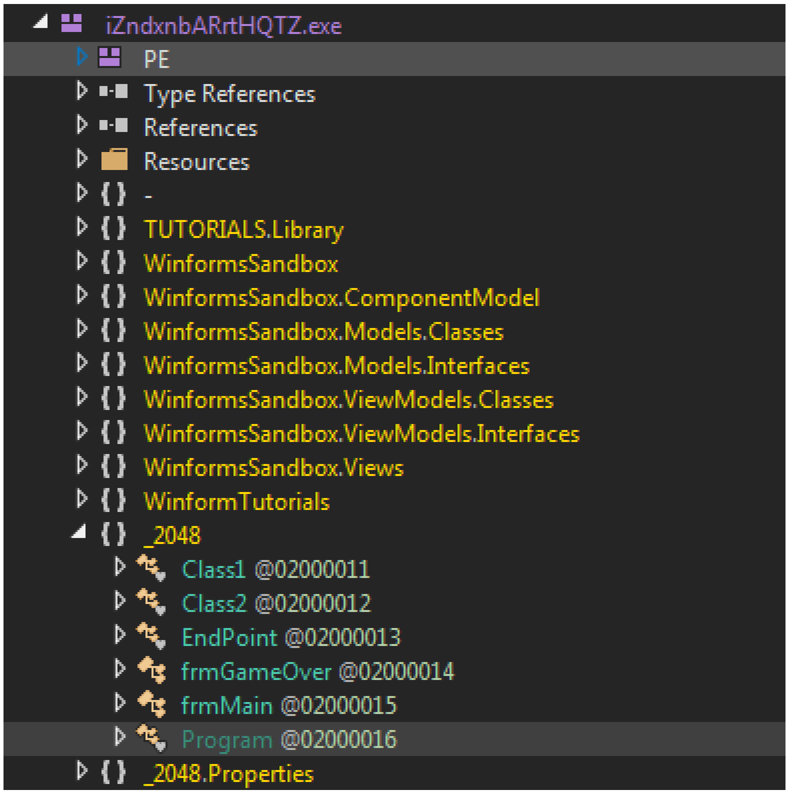
Dynamic Analysis

So to check what it does under the hood we can use dnspy and get on with debugging stuff. I moved over to my setup of Victim VM for which I use Win7 x64.

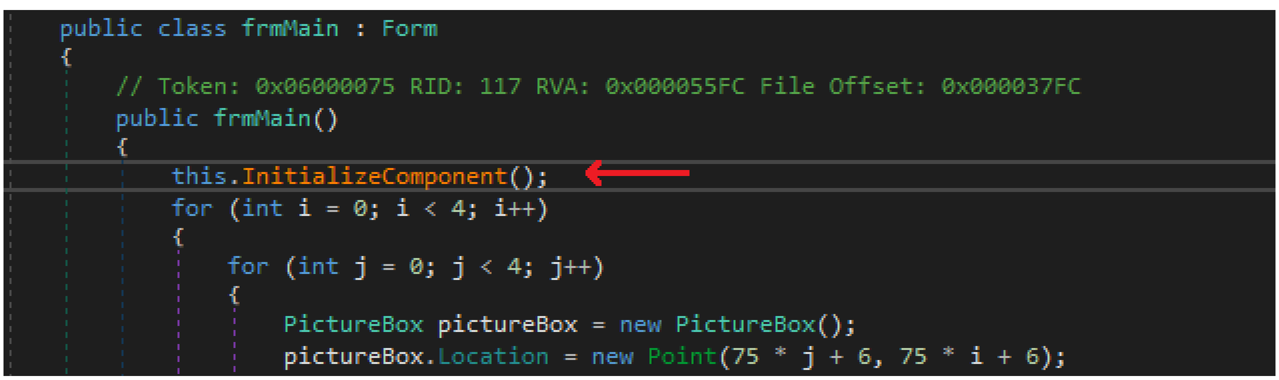
Unpacking Methods

PS I also tried unpac.me for the first time and I am very much impressed with it. For this sample it resulted in 3 children.

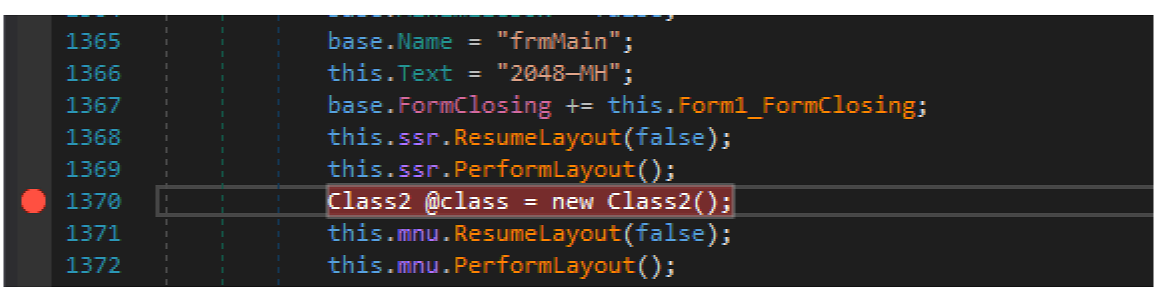
Lets see how far can we make it manually.
Hmm.. It doesn't look quite obfuscated right now.



Also It doesn't have any constructor, So we just place a breakpoint on its Entrypoint and run it.



Nice, We end up in **frmMain** and then we can just step in **InitializeComponent**. I noticed that class2 looked suspicious and setup a breakpoint there.



Ahh actually the base32 encoded payload was used here.

```
internal class Class2
{
    // Token: 0x06000066 RID: 102 RVA: 0x00004F2E File Offset: 0x0000312E
    public Class2()
    {
        this.vanilla();
    }

    // Token: 0x06000067 RID: 103 RVA: 0x00004F40 File Offset: 0x00003140
    public int vanilla()
    {
        string encoded =
          "JVNJAAADAAAAABAAAAAP77YAAC4AAAAAAAAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
          ABJTGSCVDINFZSA4DSN5TXEYLNEBRWC3TON52CAYTFEBZHK3RANFXCARCPKMQG233EMUXA2DIKEQAAAAAAAAAAAUCFAAAB
          YAAEIQAWAJQAAAD4AAAAADAAAAAAAAAABWS5AAAAAIAAAAAGAAAAAAAAAAEAAEAAAAAAACAAAAIAAAAAAAAAAAAQAAAAAAAA
          BBIAAAIAAAAQAAAAAAAQAAABAAAAAAAAAAAAQAAAAAAAAAAAAAAAAACEF2AAAJ4AAAAAAMAAABOADAAAAAAAAAAAAAAAAA
          FYAAAHAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEAAAACAAAAAAAAAA
          AAAAAAAAXHIZLYOQAAAAHAHUAAAABAAAAAAPQAAAAAEAAAAAAAAAAAAAAAAAAAAAAAAACAAAAMAXHE43SMMAAAAFYAMAAAADA
          AAAAAAAAAAEAAAAIAXHEZLMN5RQAAAMAAAAAAEAAAAAAAQAAAAEIAAAAAAAAAAAAAAAAAAAAAAAEAAAAIIAAAAAAAAAAAA
          JAAAAAACAACQB7BNAAALIEAAAAAQAAAAAAAAAAAFQHYAAAKA6AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
          SAEAAA4EAAEAAABBKAAAAAEZQAUAFQAAAAAAQAAARAABI42I7CDNBPWQX22GR2AAAAEFAEHYQAYLANDTJFAHAAAAKAADI4
          INBEEUOAQID4IF3ELB2JJAACAX2YGAQB76AILP4AITAQIQILOZAYJQKKYACECSUEZQAYAMUAAAAABAAAARABZQ6AAABIFA
```

So this is the first level of unpacking where it simply invokes a function named *f20* with arguments as *Class1.Myproperty* & *_2048*. [2]

```
          AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
          AAACAAIAAAAAAAAAAAEQAAAAAAAAAAAAAAAAAAAIAAAAACAAAAAAAAAAABAEAQAAQCAIAAGAYDAACAIBAAAUCQKAAGAYDAABYHA4AAQCAI
          BMFQADAMBQAA2DINAAHA4DQAB4HQ6AAQCAIAAEIRCEABEEQSAAJRGEYACQKBIAAVCUKQAFQWCYABOFYXAAMBQGAADEMRSAA2DINAAGY
          ADYPB4AA7[...string is too long...]";
        Type x = Assembly.Load(Base32.Decode(encoded)).GetTypes()[0];
        this.sdasad(x);
        Environment.Exit(0);
        return 0;
    }

    // Token: 0x06000068 RID: 104 RVA: 0x00004F7C File Offset: 0x0000317C
    private void sdasad(Type x)
    {
        object obj = x.InvokeMember("f20", BindingFlags.InvokeMethod, null, null, new object[]
        {
            Class1.MyProperty,
            "_2048"
        });
    }
  }
}
```

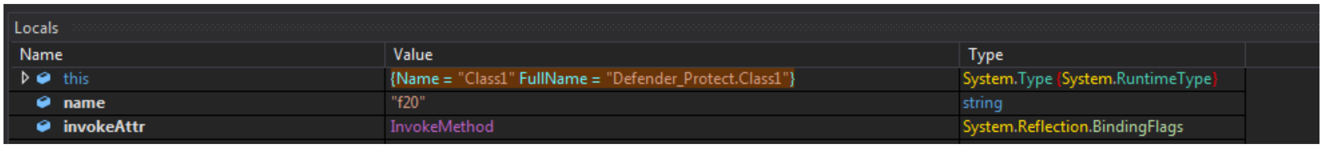
Now I place a breakpoint on return in InvokeMember and just continue.

```
            object result = ((MethodInfo)methodBase).Invoke(target, bindingFlags, binder, providedArgs, culture);
            if (obj != null)
            {
                binder.ReorderArgumentArray(ref providedArgs, obj);
            }
            return result;
        }
    }
```

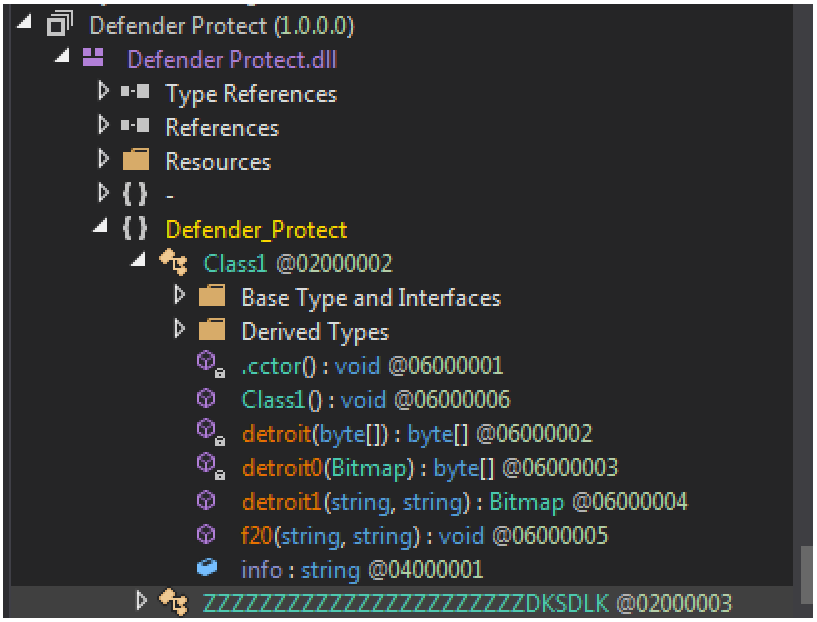
Now stepping in we find some interesting locals and the payload file is a dll named **DefenderProtect.dll**.

Locals

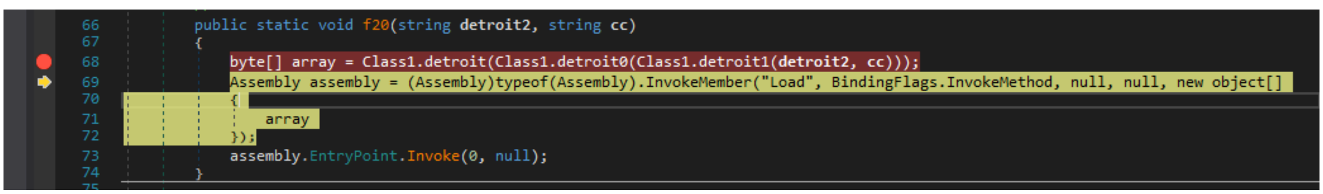
Name	Value	Type
this	{_2048.Class2}	_2048.Class2
x	{Name = "Class1" FullName = "Defender_Protect.Class1"}	System.Type {System.RuntimeType}
Assembly	{Defender Protect, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null}	System.Reflection.Assembly
AssemblyQualifiedName	"Defender_Protect.Class1, Defender Protect, Version=1.0.0.0, Culture=neutral,...	string
Attributes	VisibilityMask	System.Reflection.TypeAttributes
BaseType	{Name = "Object" FullName = "System.Object"}	System.Type {System.RuntimeType}
Cache (System.Reflection.MemberInfo)	{System.Reflection.Cache.InternalCache}	System.Reflection.Cache.InternalCac...
Cache	{System.RuntimeType.RuntimeTypeCache}	System.RuntimeType.RuntimeTypeC...
ContainsGenericParameters	false	bool



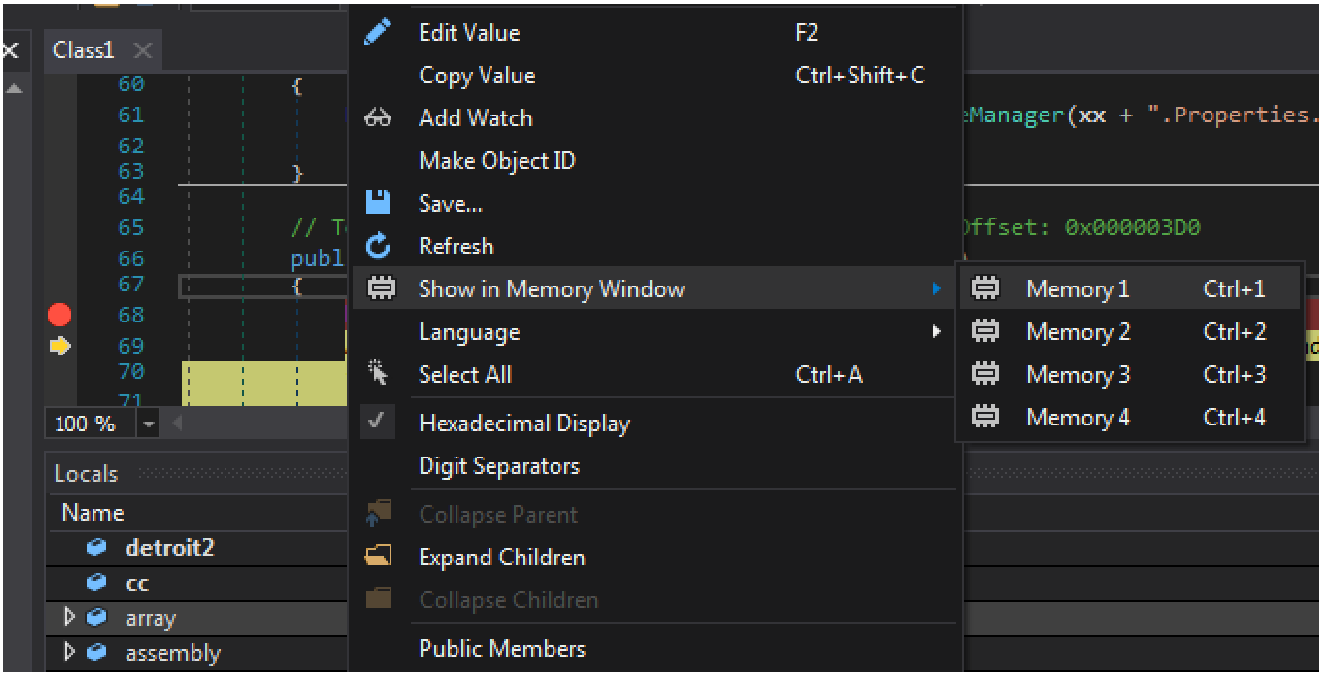
Also It has another methods as well which are used in *f20*.



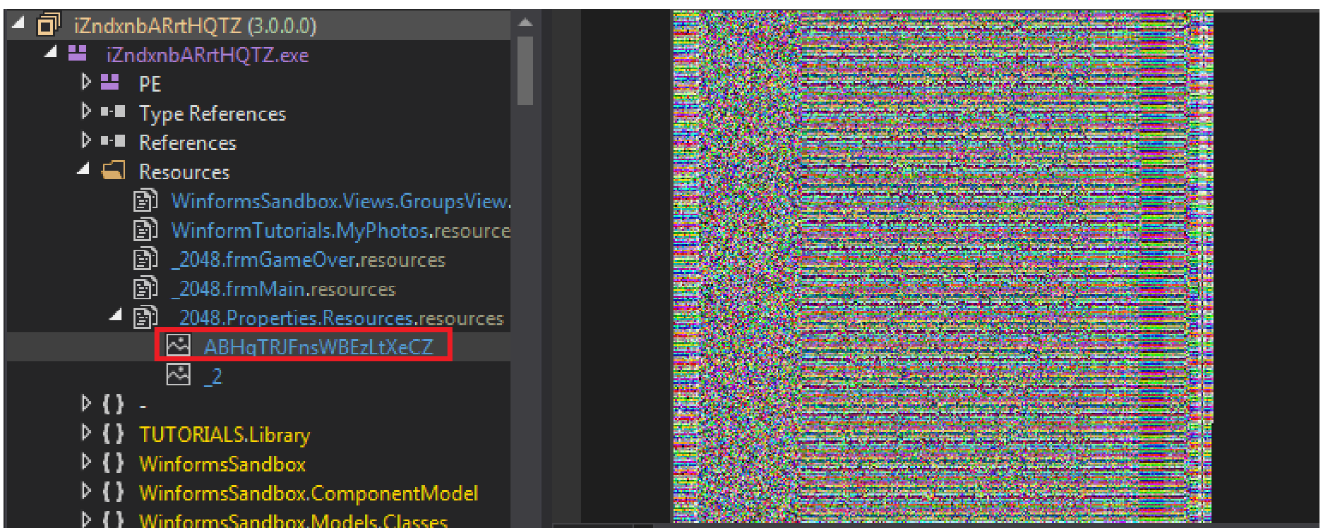
So *f20* is basically used to unpack another file



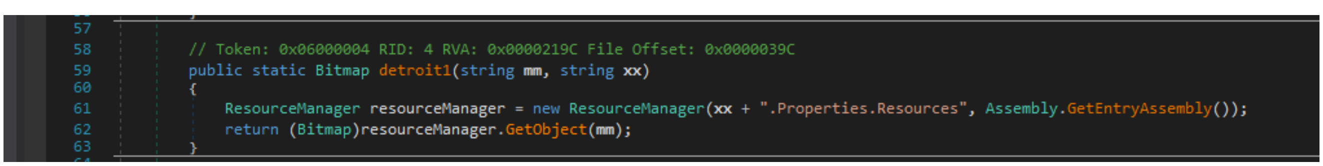
The *array* has the final decrypted 2nd payload file so we can dump it using Memory Window too.



But whats the fun in doing that, instead we can try to understand the unpacking algo. Hmm.. the resource from *_2048* named *ABHqTRJFnsWBEzLtXeCZ* is used in this process.



fcn. *detroit1* just returns that resource as a handle to a bitmap image.



The main algo resides in fcn *detroit1* and *detroit*
detroit1 adds a pixel's rgb value to a list when it is non-black.

```
private static byte[] detroit0(Bitmap aa)
{
    List<byte> list = new List<byte>();
    checked
    {
        int num = aa.Size.Width - 1;
        for (int i = 0; i <= num; i++)
        {
            int num2 = aa.Height - 1;
            for (int j = 0; j <= num2; j++)
            {
                Color pixel = aa.GetPixel(i, j);
                bool flag = !pixel.Equals(Color.FromArgb(0, 0, 0, 0));
                if (flag)
                {
                    list.InsertRange(list.Count, new byte[]
                    {
                        pixel.R,
                        pixel.G,
                        pixel.B
                    });
                }
            }
        }
        return list.ToArray();
    }
}
```

Then *detroit* does a repeating key xor on the list returned by *detroit0* where the key is first 16 bytes of the list.

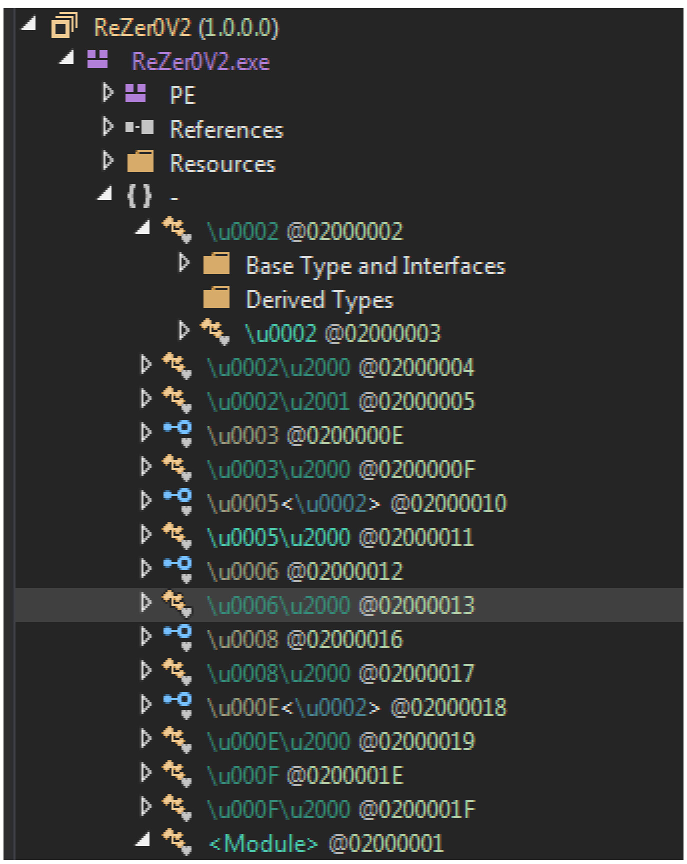
```
public class Class1
{
    // Token: 0x06000002 RID: 2 RVA: 0x00002060 File Offset: 0x00000260
    private static byte[] detroit(byte[] ii)
    {
        checked
        {
            byte[] array = new byte[ii.Length - 16 - 1 + 1];
            Buffer.BlockCopy(ii, 16, array, 0, array.Length);
            int num = array.Length - 1;
            for (int i = 0; i <= num; i++)
            {
                ref byte ptr = ref array[i];
                ptr ^= ii[i % 16];
            }
            return array;
        }
    }
```

So I just saved the bitmap image and wrote a python script to test the algo as well.

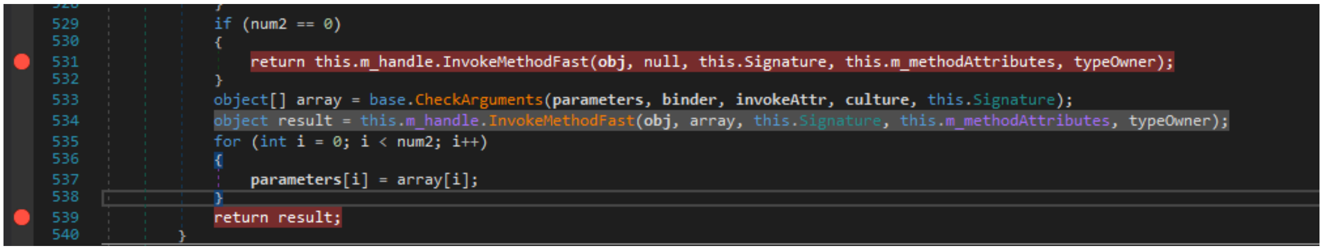
```
from PIL import Image
from hexdump import *

img = Image.open("ABHqTRJFnsWBEzLtXeCZ")

pixels = img.load()
pixList = []
width, height = img.size

for x in range(width):
    for y in range(height):
        cpixel = pixels[x, y]
        if(cpixel != (0,0,0,0)):
            for value in cpixel[:3]:
                        pixList.append(value)

xorkey = pixList[:16]
encPayload = pixList[16:]

i = 0
while(i < len(encPayload)):
    encPayload[i] ^= xorkey[i%16]
    i+=1

dec = ''.join([chr(d) for d in
encPayload[:9200]])
print hexdump(dec)

payload = open('dontopen.gg','wb')
for lol in dec:
    payload.write(chr(lol))
```

```
00000000: 4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00
MZ..............
00000010: B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00
........@.......
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00
................
00000040: 0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68
........!..L.!Th
00000050: 69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program
canno
00000060: 74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in
DOS
00000070: 6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00
mode....$.......
00000080: 50 45 00 00 4C 01 03 00  0D C8 84 5E 00 00 00 00
PE..L......^....
```

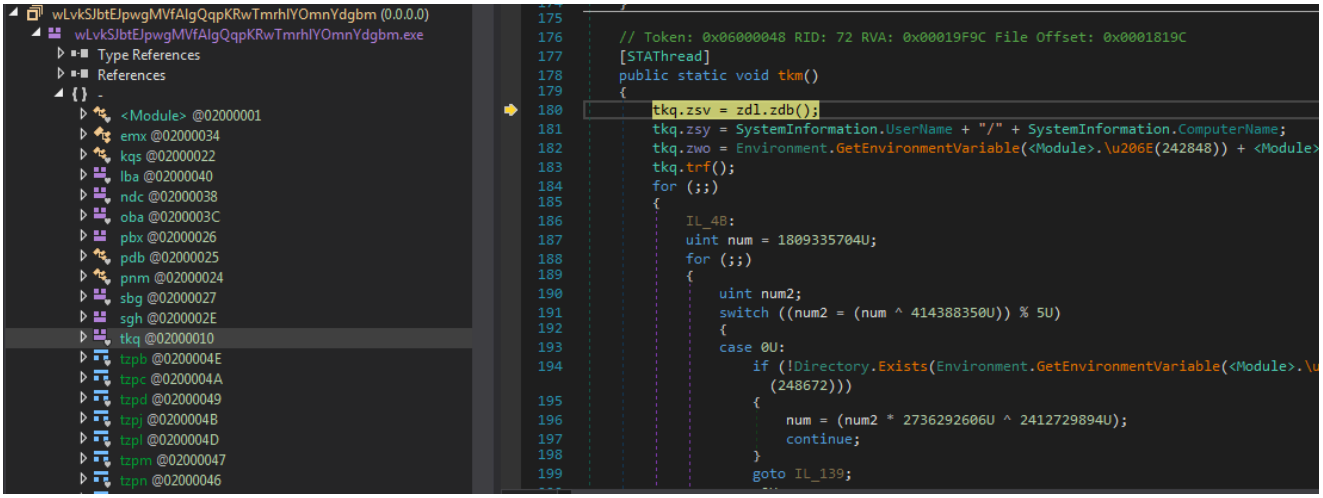
Now this boy looks obfuscated.



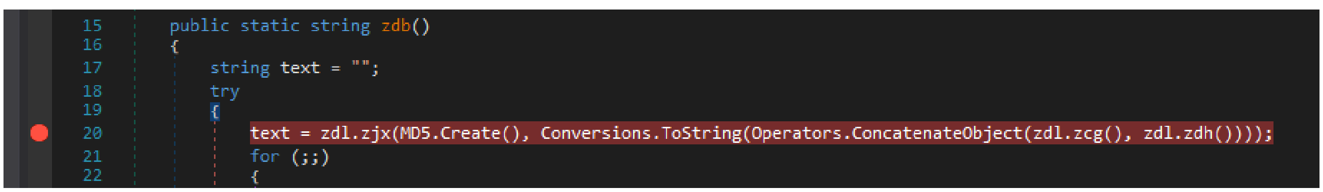
But it just starts a thread and I was unable to debug it.
Also that isn't a big deal as this was mainly invoking a method from another file it just unpacked.. ¯_(ツ)_/¯



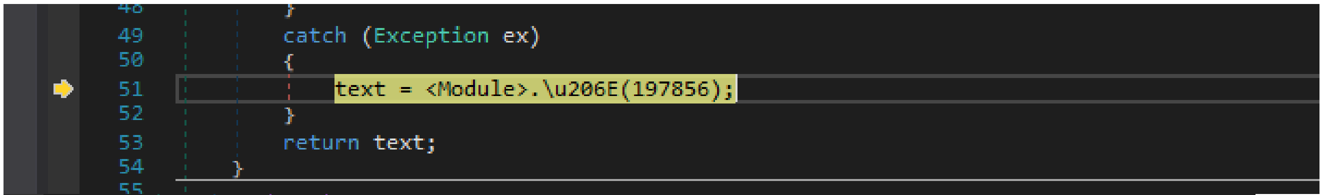
Now finally we will be analysing the last and third file which resulted in unpacme.



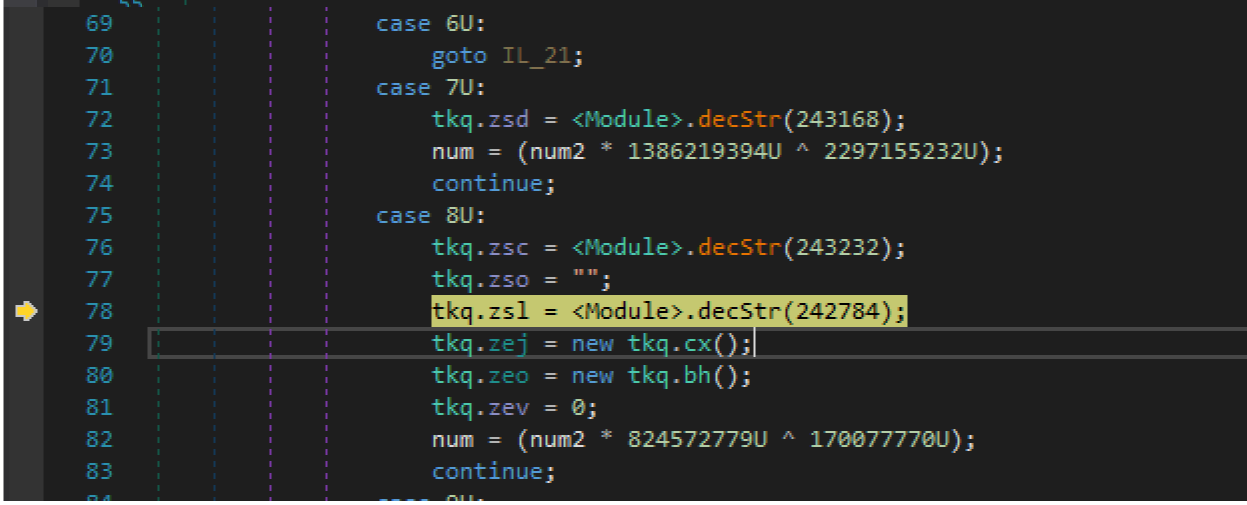
We step in and are currently in *zdb* method.



Now when I stepped in the above line to get value for text I observed that a function is called repeatedly. Hmm.. Maybe It is used for some deobfuscation or decryption of suspicious.



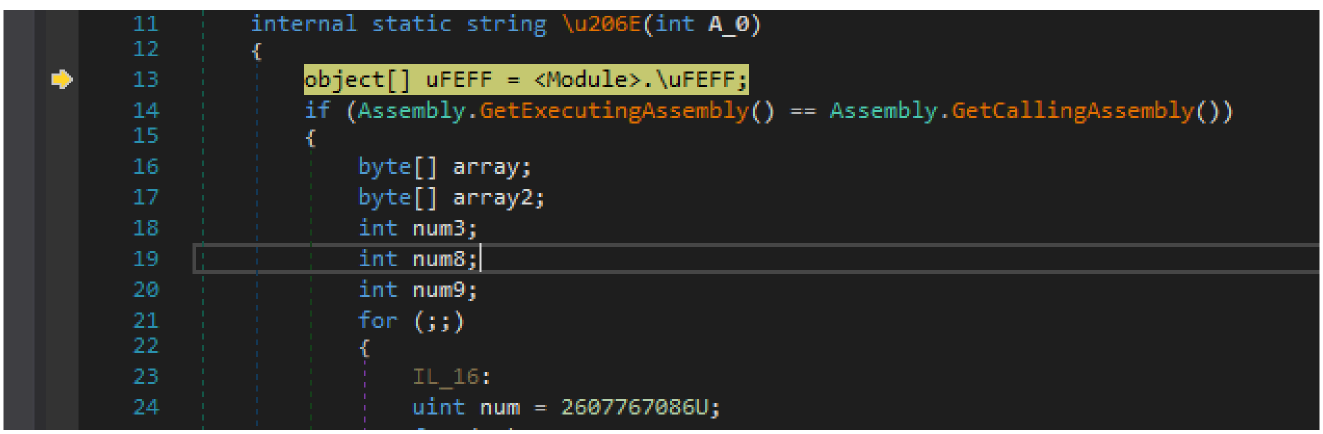
```
catch (Exception ex)
{
    text = <Module>.\u206E(197856);
}
return text;
}
```



```
case 6U:
    goto IL_21;
case 7U:
    tkq.zsd = <Module>.decStr(243168);
    num = (num2 * 1386219394U ^ 2297155232U);
    continue;
case 8U:
    tkq.zsc = <Module>.decStr(243232);
    tkq.zso = "";
    tkq.zsl = <Module>.decStr(242784);
    tkq.zej = new tkq.cx();
    tkq.zeo = new tkq.bh();
    tkq.zev = 0;
    num = (num2 * 824572779U ^ 170077770U);
    continue;
```

Decrypting Strings

We step into that suspicious function obfuscated as *\u206E* and at first It looks like assigning a list of objects from *\uFEFF*



```
internal static string \u206E(int A_0)
{
    object[] uFEFF = <Module>.\uFEFF;
    if (Assembly.GetExecutingAssembly() == Assembly.GetCallingAssembly())
    {
        byte[] array;
        byte[] array2;
        int num3;
        int num8;
        int num9;
        for (;;)
        {
            IL_16:
            uint num = 2607767086U;
```

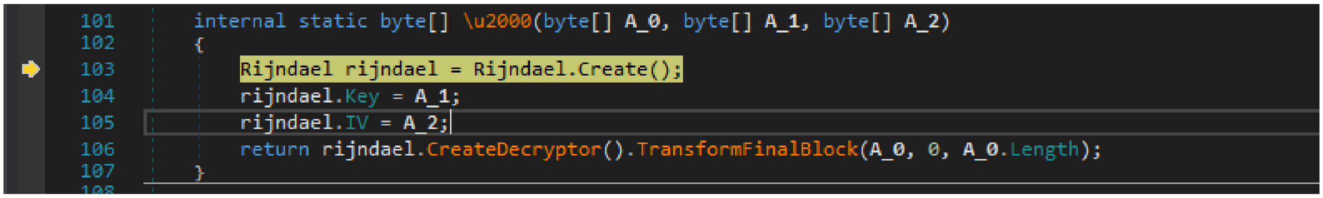
The object array looks like the following in the locals window and contains integer arrays.

Locals	
Name	Value
value	0x000304E0
uFEFF	{object[0x00000362]}
[0]	{uint[0x00000010]}
[0]	0xC6B9D9EF
[1]	0x0C3BB9E5
[2]	0x0AD85631
[3]	0xD340E817
[4]	0xADA72279
[5]	0x7880CC14
[6]	0x3CB3DDCF
[7]	0x7403B9F4
[8]	0x93DFCD69
[9]	0x2A623833
[10]	0x5027A082
[11]	0x09AFA6FC
[12]	0x9A3D3387
[13]	0xFFD4DBA4
[14]	0x8E4FE42D
[15]	0xFF88D934
[1]	{uint[0x00000010]}
[0]	0xBB6D39B5
[1]	0x0D3ADB3F
[2]	0xC2778268
[3]	0xE7FE4105
[4]	0x82A4E9D9
[5]	0x082F196E

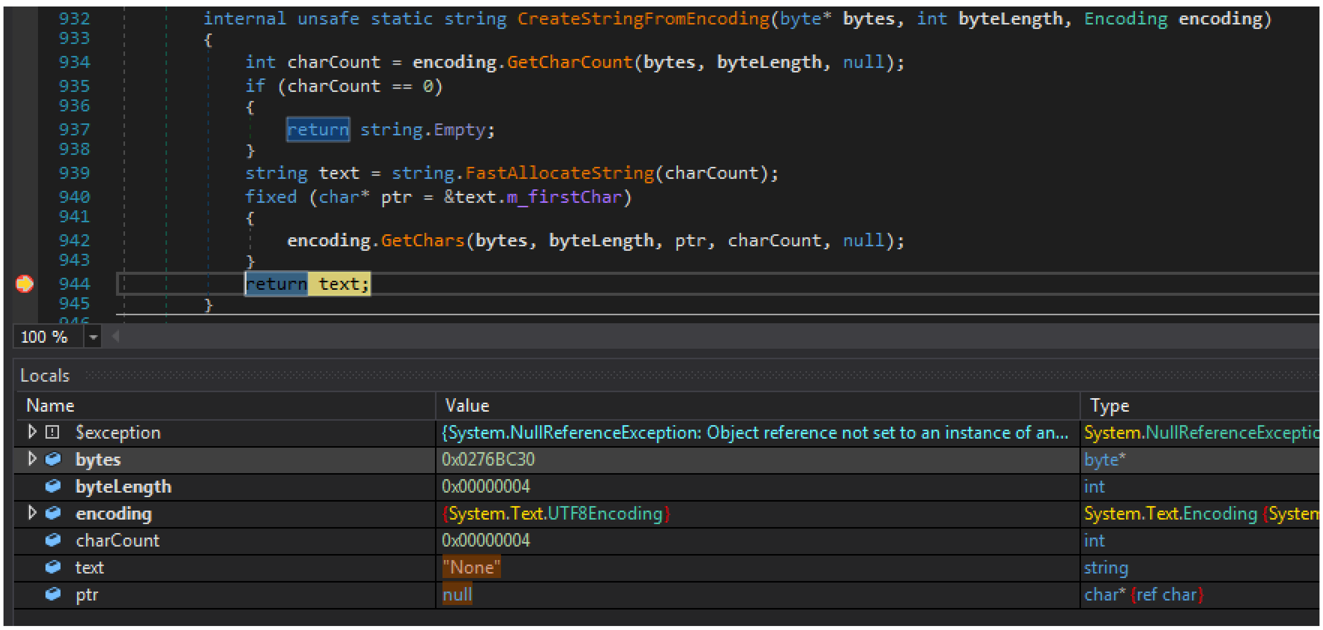
So we setup a normal breakpoint at the function return. It passes the beginning 32 bytes of the string as key and the next 16 bytes as the IV to the Decryption function.

```
                }
                goto Block_1;
            }
        }
        Block_1:
        goto IL_1E4;
        IL_15A:
        uint[] array3 = (uint[])uFEFF[num3];
        byte[] array4 = new byte[array3.Length * 4];
        Buffer.BlockCopy(array3, 0, array4, 0, array3.Length * 4);
        byte[] array5 = array4;
        int num10 = array5.Length - (num8 + num9);
        byte[] array6 = new byte[num10];
        Buffer.BlockCopy(array5, 0, array, 0, num8);
        Buffer.BlockCopy(array5, num8, array2, 0, num9);
        Buffer.BlockCopy(array5, num8 + num9, array6, 0, num10);
        return Encoding.UTF8.GetString(<Module>.\u2000(array6, array, array2));
    }
    IL_1E4:
    return "";
}
```

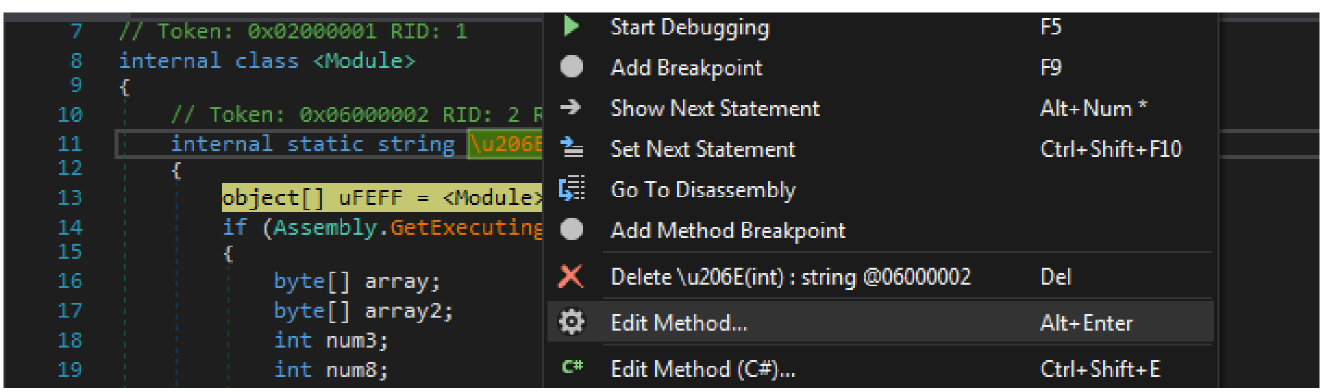
And Now execution is passed over to Rijndael(AES) decryption function and we can clearly see that it isn't obfuscated and has variable names as key & IV, and looks like CBC mode ezpz :)



We can just place a Breakpoint at *return text* in *CreateStringFromEncoding* and we will get the decoded string in the locals window and we'd get to know whenever this decryption func is invoked as well.



So this time it returns "None" due to exception but sometimes the same gives "WinMgmt:". Also we can now rename it to **decStr()** for our ease.



Moving on.. It access/creates some environment variables.

encoding	{System.Text.UTF8Encoding}
charCount	0x0000000F
text	"%startupfolder%"

Locals

Name	Value
bytes	0x0277AF88
byteLength	0x00000016
encoding	{System.Text.UTF8Encoding}
charCount	0x00000016
text	@"\%insfolder%\%insname%"
ptr	null

Now I thought of decrypting all of the strings with python.

Unfortunately I was not able to copy the content of the encrypted int array from the locals and copying it from the declaration was not efficient.
The problem with dumping a array local from the memory window (in this case) in dnspy is it just shows it in reverse (maybe coz of little endian).
But the array starts below what it refers to there and I was somehow able to select it manually and dumped it finally.

Then I tried to implement the algo in python and coz of my weird workaround for dumping it, the script resulted in some errors. But I noticed that the error arised due to the values in list strEnds(below) and they were pretty common at the string end and I used them to split the dump and get a single string. I think this was because of the uint[] initialization in the object array.
Anyways It finally worked and there were around 865 enc strings.

PS The Key and IV for every cipher is different and is taken from the encoded string as well.

```
import string
from Crypto.Cipher import AES

def decipher(dd):
    key = dd[0:0x20]
    iv = dd[0x20:0x30]
    cipher = dd[0x30:]
    rijn = AES.new(key, AES.MODE_CBC, iv)
    decipher = rijn.decrypt(cipher).strip()
    plain = filter(lambda x: x in string.printable,
decipher)
```

```
    print plain

dmp = open('dump.txt').read()

strEnds = ["0000000048191F0110000000",
"0000000048191F0114000000",
"0000000048191F0118000000",
"0000000048191F011C000000",
"0000000048191F0124000000",
"0000000048191F0120000000",
"0000000048191F012C000000",
"0000000048191F0128000000",
"000000004819C4001C000000"]

for end in strEnds:
    dmp = dmp.replace(end, " ")

lol = dmp.split()
c = 0

for x in lol:
    print c,
    try:
        decipher(x.decode('hex'))
    except:
        print "[!] ERROR :", x , "Length :",
len(x.decode('hex'))
    c+=1
```

> Full Results : dec.txt

Some of the decrypted strings are as follows :

```
WScript.Shell
Software\Microsoft\Windows\CurrentVersion\Run
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\StartupApprove
d\Run
SELECT * FROM Win32_Processor
Opera Software\Opera Stable
Yandex\YandexBrowser\User Data
Chrome\Chrome\User Data
\FTP Navigator\Ftplist.txt
HKEY_CURRENT_USER\Software\FTPWare\COREFTP\Sites
```

Hmm so it also uses WScript.Shell, maybe for executing some system commands.
Also it uses some registry keys for persistence and adding itself to the startup.
And Gets some info about our system using Win32_Processor
Access locations associated with browsers mainly “User Data” and looks for some FTP credentails too.

> So Now I guess Its enough for Part-1, Head over here for the 2nd Part.