Rainbows, Steganography and Malware in a new .NET cryptor

August 7, 2015 by hasherezade

Last updated: March 30, 2016

This post describes the process of unpacking a malicious payload delivered in a new spam campaign.

I often observe malicious samples, distributed in spam campaigns, that are packed by nifty, multilayer packers. This time, the first layer (the main file) is a heavily obfuscated .NET executable. It could be more difficult to crack, if the author had not neglected several details…

We will refer the used packer as **discuri**, since this name appears as the internal name of the files that we observed to be packed in this way. Some examples: (c215514941f8d99f23642050a6efbbf1, 7b29954d5cbe7ca9dcd3218476afa133)

Let's start from the brief look on the involved elements. Then, I will present the way that lead me to extract the malicious payload. Technical summary of the modules and their functionality is provided at the end.

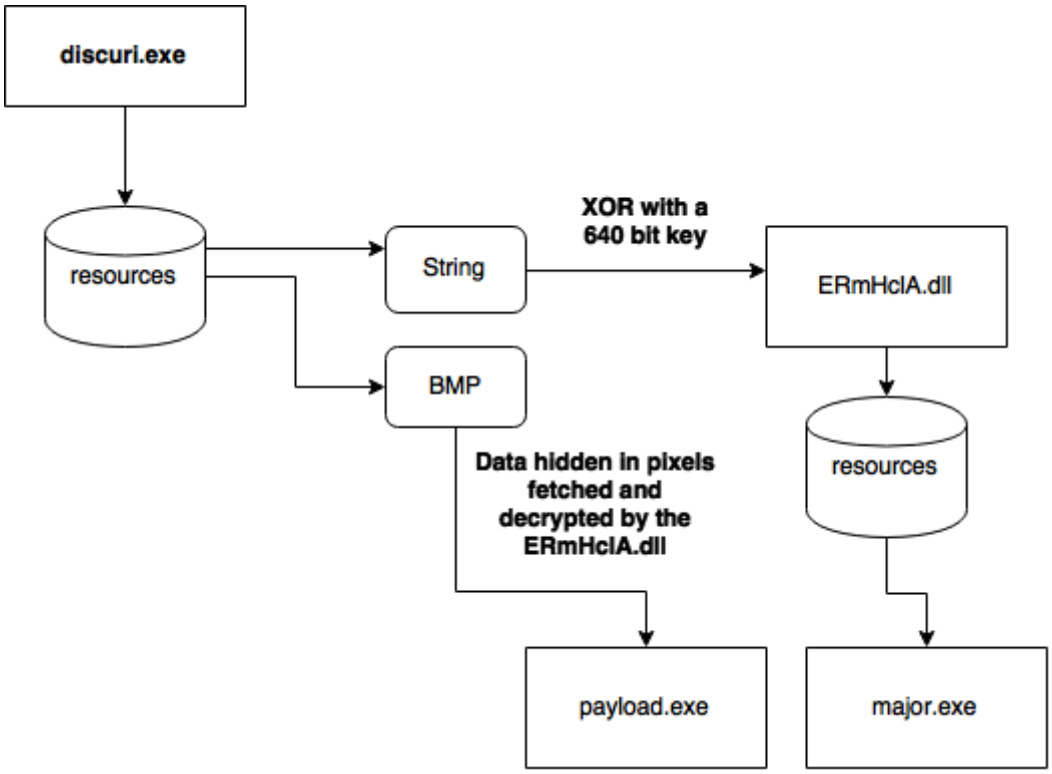
Elements involved

– **discuri.exe** (c215514941f8d99f23642050a6efbbf1) – the main file (obfuscated .NET exe)

– ERmHclA.dll (35d92229414f00a5335cc9957819b5d0) – an encrypted unpacker (.NET dll)

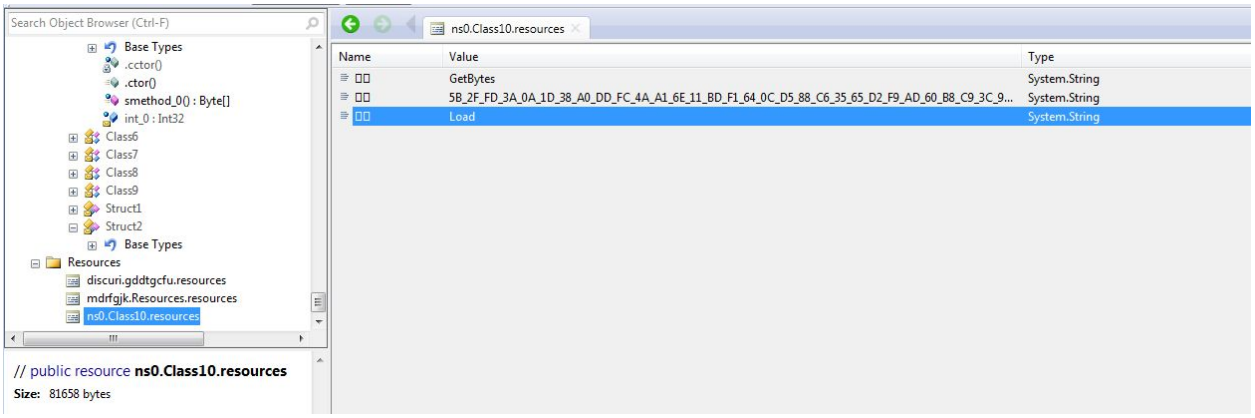
– major.exe (8b17d0360521852d87e07f3ca66a5ac7) – .NET exe

– *payload*.exe (88fbb83445929812deaae6da358d0b7c) – the malicious part



Unpacking

Let's start from decompiling the main .NET file. As we can see it is heavily obfuscated. We can take the long way and try to deobfuscate it, or search for some weak points. For example, the string below (starting from 5B_2F..), looks like a set of bytes. It can possibly be a payload.



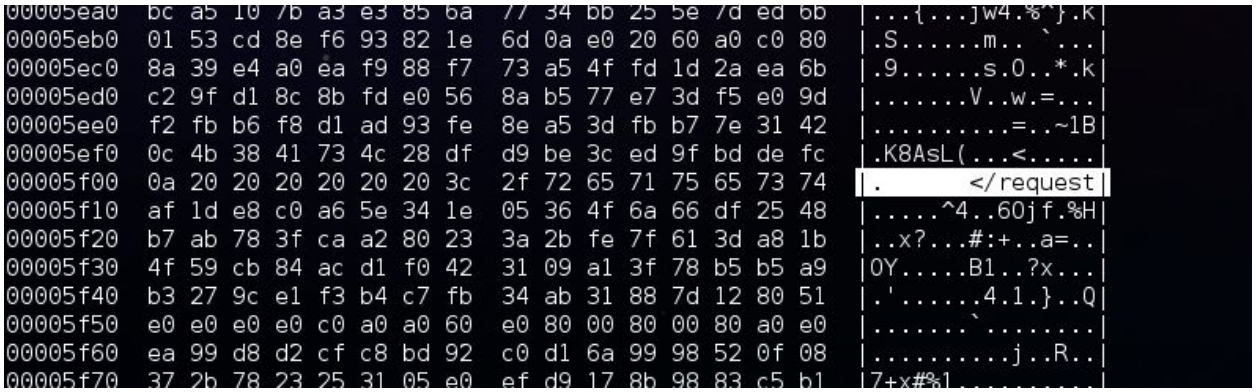
Converting it to bytes didn't make it readable:

```
00000000  5b 2f fd 3a 0a 1d 38 a0  dd fc 4a a1 6e 11 bd f1  |[/.:..8...J.n...|
00000010  64 0c d5 88 c6 35 65 d2  f9 ad 60 b8 c9 3c 92 99  |d....5e...`..<..|
00000020  81 fe 35 39 ec cc dd e0  96 a5 dd aa 89 ed 18 e0  |..59............|
00000030  79 0c 9a 91 d1 be bd 91  9c bc 85 f8 06 65 05 52  |y............e.R|
00000040  97 62 2a a6 89 78 9b f4  a0 96 44 65 21 dd 69 c8  |.b*..x....De!.i.|
00000050  9f e6 ad aa bb d2 ff b2  58 11 6a 42 f0 00 73 7e  |........X.jB..s~|
00000060  88 cc d7 8d e6 a7 f0 5c  39 44 4e 18 4d f3 e1 d9  |.......\9DN.M...|
00000070  4c 31 71 7c 02 21 30 4a  12 25 5d 2a 09 6d 78 40  |L1q|.!0J.%]*.mx@|
00000080  49 69 3a 31 1d 1f 1e b1  c9 f0 ff ad c6 e5 e5 f2  |Ii:1............|
00000090  f9 1d f0 c8 29 6c 30 f8  ea af 75 a9 ec 1e 9d c0  |....)l0...u.....|
000000a0  d6 f3 6d 3a c9 dd f8 a0  17 7c 8a a1 d1 ce 3d f1  |..m:.....|....=.|
000000b0  5c ec 95 c8 86 b5 e5 82  b9 8d a0 b8 89 3e 52 59  |\............>RY|
000000c0  c5 7e b5 b9 ac 8c 9d 60  d2 25 1d 2a c9 ed 18 e0  |.~.....`.%.*....|
000000d0  f9 2c da d1 91 bc bd d1  1c 3c 45 78 c5 65 05 97  |.,.......<Ex.e..|
000000e0  d9 fd 80 a8 c9 9c d2 39  41 2e 15 29 2c ec bd a0  |.......9A..),...|
000000f0  76 55 4d 1a 19 3d 18 00  39 7c 0a 21 51 6e 5d 51  |vUM..=..9|.!Qn]Q|
00000100  00 ec 35 68 49 15 45 b2  d9 0d 60 b8 a1 bf b2 f9  |..5hI.E...`.....|
00000110  a1 1e d5 d9 ec 2c 3d 80  b6 a5 1d aa c9 6d b8 80  |.....,=......m..|
00000120  d9 6c 3a 31 9d de dd b1  18 c3 85 f8 9a e5 65 f2  |.l:1..........e.|
00000130  79 5d b0 88 89 ec b2 99  e1 ae 85 a9 ac 7c 5d 00  |y]...........|].|
00000140  96 75 ed ba 89 9d b8 20  d9 7c 4a 21 91 ee 3d f1  |.u..... .|J!..=.|
00000150  dc 8c d5 88 c6 b5 e5 d2  39 0d 60 38 c1 3c 12 19  |........9.`8.<..|
00000160  81 fe b5 b9 ec cc dd 60  1e 05 5d 2a 41 ed 98 e0  |.......`..]*A...|
00000170  79 0c 1a 11 51 3e 3d 11  32 48 60 00 72 65 05 52  |y...Q>=.2H`.re.R|
00000180  8d 1d 10 28 09 6c 12 b9  81 cc 45 a9 ec fe bd a0  |...(.l....E.....|
00000190  f6 15 8d da c9 3d 18 c0  b9 fc 4a a1 b1 6e 9d f1  |.....=....J..n..|
000001a0  d2 9e 46 1a a5 d5 85 b2  f1 ae a0 b8 89 1c 32 f9  |..F...........2.|
000001b0  21 5a 95 99 ac c8 bd c0  b6 a5 dd aa 89 6d 78 40  |!Z...........mx@|
000001c0  99 2c ba b1 91 9e 9d 71  d2 4e 20 14 a9 86 65 f2  |.,.....q.N ...e.|
000001d0  f5 9d f0 c8 c9 2c b2 d9  61 2c 45 29 ec 14 1d 40  |.....,..a,E)...@|
000001e0  d6 f5 ed ba c9 dd f8 20  19 7c 0a 21 11 ee bd b3  |....... .|.!....|
000001f0  5c 4c 15 48 06 35 65 12  39 2d 20 38 09 3c 52 59  |\L.H.5e.9- 8.<RY|
```

But we can see some regularities that suggests that the data has been encrypted by XOR with a key. The key is obviously hidden in the obfuscated executable.

But the property of XOR function (XOR reverse XOR) gave me an idea on how to extract it in another way. I assumed that the output will be just another executable. Based on this assumption, we can try to extract the beginning of the key by XOR with the headers that are typical for Windows executables (PE).

I dumped the first 16 bytes of the typical PE file, and xored it with the first 16 bytes of the encrypted file. Then, with the help of the obtained key, I attempted unpacking the full file. The result confirmed the hypothesis that some valid strings would appear… but the file was still not decrypted meaning that the key is longer than 16 bytes.



After some research (thanks to **Joshua** for helping in this) we figured out that the valid key is 640 bytes long! It's far beyond the PE header… So we had to find another pattern, with the help of which the remaining part of the key could be extracted.

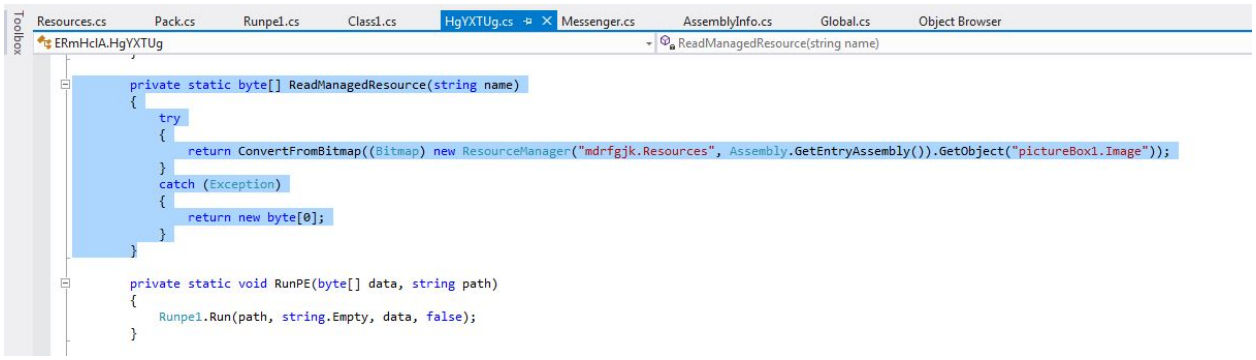
Fortunately, the payload turned out to be a .NET file with a typical XML manifest. Applying XOR with a typical manifest on appropriate fragment of the encrypted content gave us the full key.

Used elements [github]

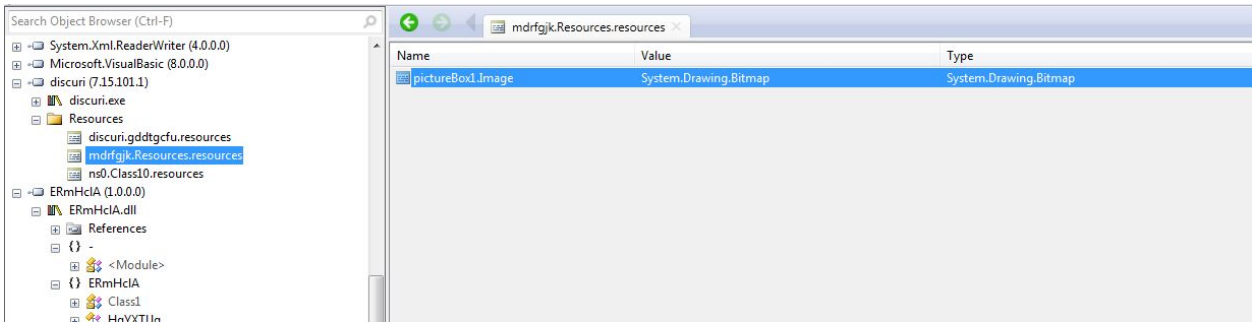
- bytes.txt – the dumped string of bytes (enctypted)
- rev_key.txt – the recovered key
- decoder.py – a python script decrypting the string of bytes

The output is a DLL, written in .NET and not obfuscated. It's original name is **ERmHclA.dll**.

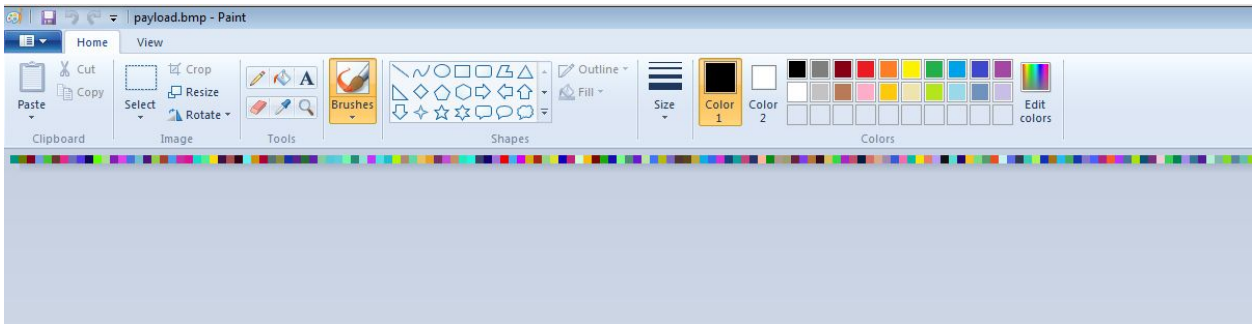
I noticed that it reads data from some BMP:



This BMP can be found in the resources of the main file:



After dumping the image we can see something very weird – a one pixel height, colorful strip. It's easy to guess, that it is not a real image, but a data represented by pixels.



As we can find out by reading the code that processes it, the BMP contains another executable (the malicious payload) and some configuration, packed as serialized objects of various types.

```
62  int timestamp = GetTimestamp();
63  byte[] data = ReadManagedResource(timestamp.ToString(CultureInfo.InvariantCulture));
64  object[] objArray = new Pack().Deserialize(Decrypt(data,
65  timestamp.ToString(CultureInfo.InvariantCulture)));
66  FileData = (byte[]) objArray[0]; //the malicious payload
67  int num2 = (int) objArray[1];
68  int num3 = num2 * 5;
69  bool flag5 = (bool) objArray[2 + num3];
70  bool flag6 = (bool) objArray[3 + num3];
71  byte num4 = (byte) objArray[4 + num3];
72  bool flag7 = (bool) objArray[5 + num3];
73  string caption = objArray[6 + num3].ToString();
74  string text = objArray[7 + num3].ToString();
75  bool flag8 = (bool) objArray[8 + num3];
76  bool flag9 = (bool) objArray[9 + num3];
77  bool flag10 = (bool) objArray[10 + num3];
78  bool flag11 = (bool) objArray[11 + num3];
79  string uriString = (string) objArray[12 + num3];
80  string key = (string) objArray[13 + num3];
81  bool flag12 = (bool) objArray[14 + num3];
82  bool flag13 = (bool) objArray[15 + num3];
83  int num5 = (int) objArray[0x10 + num3];
84  bool flag14 = (bool) objArray[0x11 + num3];
85  bool flag15 = (bool) objArray[0x12 + num3];
86  string str9 = objArray[0x13 + num3].ToString();
87  string str10 = objArray[20 + num3].ToString();
88  bool flag16 = (bool) objArray[0x15 + num3];
89  string str11 = objArray[0x16 + num3].ToString();
90  string str12 = objArray[0x17 + num3].ToString();
91  bool flag17 = (bool) objArray[0x18 + num3];
92  string str13 = objArray[0x19 + num3].ToString();
93  string str14 = objArray[0x1a + num3].ToString();
    injectionPath = string.Empty;
```

The functions used for its decompression are inside the DLL (not obfuscated):

```
372    private static byte[] ReadManagedResource(string name)
373    {
374      try
375      {
376        return ConvertFromBitmap((Bitmap) new ResourceManager("mdrfgjk.Resources",
377    Assembly.GetEntryAssembly()).GetObject("pictureBox1.Image"));
378      }
379      catch (Exception)
380      {
381        return new byte[0];
382      }
       }
```

```
303   private static byte[] ConvertFromBitmap(Bitmap bmp)
304   {
305     BitmapData bitmapdata = bmp.LockBits(new Rectangle(Point.Empty, bmp.Size),
306   ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);
307     byte[] destination = new byte[4];
308     Marshal.Copy(bitmapdata.Scan0, destination, 0, 4);
309     Array.Resize(ref destination, BitConverter.ToInt32(destination, 0));
310     Marshal.Copy(new IntPtr(bitmapdata.Scan0.ToInt64() + 4L), destination, 0,
311   destination.Length);
312     bmp.UnlockBits(bitmapdata);
313     return destination;
314   }
315
316   private static byte[] Decompress(byte[] data)
317   {
318     byte[] buffer = new byte[(BitConverter.ToInt32(data, 0) - 1) + 1];
319     DeflateStream stream1 = new DeflateStream(new MemoryStream(data, 4,
320   data.Length - 4, false), CompressionMode.Decompress);
321     stream1.Read(buffer, 0, buffer.Length);
322     stream1.Close();
323     return buffer;
324   }
325
326   private static byte[] Decrypt(byte[] data, string key)
327   {
328     Console.WriteLine("key:" + key);
329     byte[] bytes = Encoding.UTF8.GetBytes(key);
330     Random random = new Random(BitConverter.ToInt32(data, 0));
331     byte[] buffer2 = new byte[data.Length - 4];
332     for (int i = 0; i <= (buffer2.Length - 1); i++)
333     {
334       buffer2[i] = (byte) (data[i + 4] ^ Convert.ToByte((int) ((random.Next(0x100) + bytes[i
      % bytes.Length]) & 0xff)));
      }
      return buffer2;
    }
```

The key was supposed to make the decryption process dependable on the main module running the DLL. It is a timestamp of this module:

```
338     private static int GetTimestamp()
339     {
340       IntPtr baseAddress = Process.GetCurrentProcess().MainModule.BaseAddress;
341       return Marshal.ReadInt32(baseAddress, Marshal.ReadInt32(baseAddress, 60) + 8);
342     }
```

Base address + 60 -> address of PE header
address of PE header + 8 -> Timestamp

After figuring this out and dumping the proper timestamp, we got all the elements necessary to write an unpacker.

Running the unpacker, we obtained the configuration values and the ***payload.exe*** **(88fbb83445929812deaae6da358d0b7c)**

Functionality

discuri.exe

The role of this layer is shielding the other layers.

It is a heavily obfuscated .NET file, that generates the XOR key, unpacks the embedded DLL with it's help, then drops the DLL and deploys it.

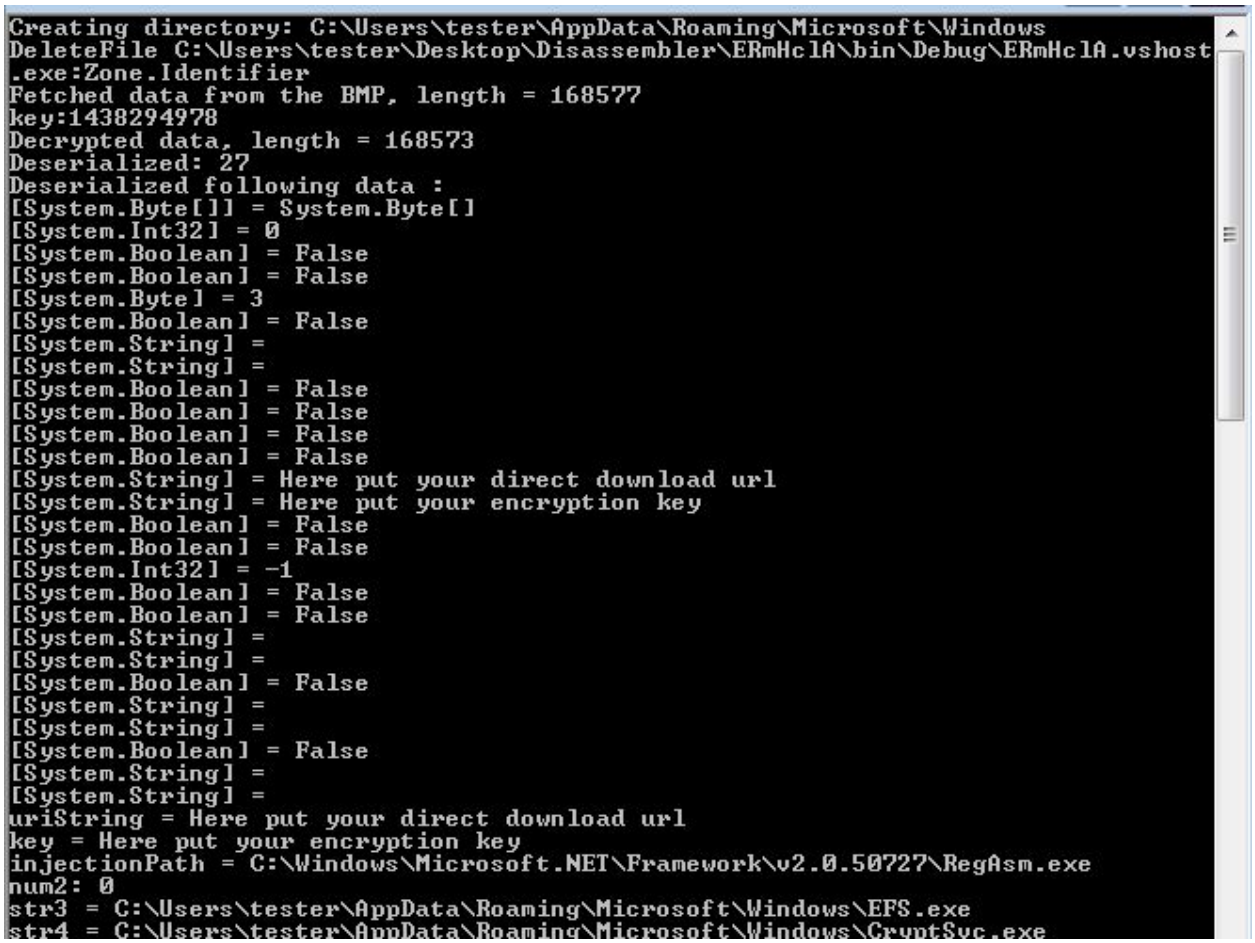
ERmHclA.dll

This module plays a crucial role in unpacking the payload and running it.

It has a rich set of options that can be configured by the user of this packer. In the analyzed case, most of the options have been disabled:

Below: values dumped from the sample's configuration (embedded in the BMP) by our unpacker:

Configuration values are in the format:
[Type] = Value



Additional paths of execution are enabled or disabled by the boolean flags. In case of the analyzed sample, as we see, all of them are disabled. Also, we can see some dummy text set as download URL (that can be used for providing additional payload) and a key (that could be used to decrypt it).

They are probably the hints for the packer's user, displayed in the generator's GUI – due to the fact, that in this case the user hasn't filled them, the default values have been embedded in the package.

This particular package, with the given configuration, is focused just on deploying the payload – disguised as RegAsm.exe with the hep of RunPE technique (running the original executable, suspending it, unmapping from the memory, mapping the payload on it's place and running it again).

The injection path is chosen from several options, depending on the value supplied in the configuration.

Available options:

0 : svchost.exe
1 : AppLaunch.exe
2 : vbc.ex
3 : RegAsm.exe
4 : RegSvcs.exe

Other interesting features provided by the DLL, that are DISABLED in the analyzed package:

1. Download additional payload from the provided URL, decrypting it with the help of the provided key and run
2. Check if the process runs under a vitrual machine/sandbox and exit in such case
3. Show *MessageBox* of a specified type displaying custom strings (given in the configuration)
4. Copy *N* additional payloads embedded in the configuration file into specific paths, set them specific attributes and run them (depending on chosen option: by starting them as a normal process, or by RunPE technique)
5. Copy 2 involved executables into **%APPDATA%\Microsoft\Windows**:
   – **main module** -> as **EFS.exe**
   – **major.exe** (carried in the DLL's resources)-> as **CryptSvc.exe**
6. Start the **major.exe** and persist it running (by checking in a loop if it haven't been killed)
7. Move the main file into:%APPDATA%\Microsoft\Windows\Templates**takshost.exe** and deploy

major.exe

This element is carried inside the DLL. Depending on the supplied configuration, it may (or not) be dropped *(into: **%APPDATA%\Microsoft\Windows**:**CryptSvc.exe**) and deployed. In the analyzed case it was not run – but still it is worth to take a brief look.

This file is small and simple – it's role is to run and kill the main module (dropped as **%APPDATA%/"Microsoft\Windows\EFS.exe**) in regular intervals of time. Also, it provides persistence by adding appropriate registry keys.

Below: experiment with fake **EFS.exe** shows the "pulse" of starting and killing the program by **major.exe**:

Fragment of code responsible for this actions:

```
1   public static void FilePersistance()
2   {
3     while (true)
4     {
5       if (Kill)
6       {
7         break;
8       }
9       Class2.GetValue(); // check if EFS.exe is running, if not -> run it
10      Class2.T(); //new ManualResetEvent(false).WaitOne(100);
11    }
12  }
13
14  public static void Main()
15    {
16        Messenger.Pong += new major.Messenger.PongEventHandler(Program.Pong);
17        new Thread(new ThreadStart(Program.Persistence) { IsBackground = true }.Start();
18        FilePersistance();
19  }
20
21  public static void Persistence()
22  {
23    while (true)
24    {
25      if (Kill)
26      {
27        break;
28      }
29      AddKey();
30      Class2.T(); //new ManualResetEvent(false).WaitOne(100);
31    }
32  }
```

payload.exe

The payload is a fully independent module, that can be a topic in itself, beyond the post about unpacking. Brief summary of its main features:

- creates mutexes typical to Zeus
- opens a port and starts listening
- periodically sends a GET request to it's C&C: *198.46.81.172*, trying to fetch: *imagess/panel/config.jpg*
  (probably waiting for additional configuration, but during the tests we haven't observed any other response than *Error 404 – Not found*)

The report from the dynamic analysis can be viewed here.

Conclusion

This packing has been recently observed in various samples. Its features depicts that it is designed especially for the purpose of protecting malware – and probably distributed on underground forums. It has high flexibility (with the help of configuration files) and a creatively designed structure. However, its weak points provided many shortcuts in the process of unpacking, not using the full potential of employed techniques.

We will keep eye on its evolution and update you in the future.