Analysis of an Unusual HawkEye Sample

govcert.ch/blog/analysis-of-an-unusual-hawkeye-sample/

GovCERT.ch

- Homepage
- GovCERT.ch Blog

Close

Blog Posts

Recently published blog posts:

- 12.12.2021Zero-Day Exploit Targeting Popular Java Library Log4j
- 09.03.2021Exchange Vulnerability 2021
- 27.10.2020Cyber Security for the Healthcare Sector During Covid19

Blog Archive

Go to the blog archive and browse all previous blog posts we have published so far.

RSS Feed

Subscribe to the GovCERT.ch blog RSS feed to stay up to date and get notified about new blog posts.

- Whitepapers

Close

Whitepapers

Recently published whitepapers:

  - 23.09.2019Trickbot - An analysis of data collected from the botnet
  - 03.03.2017Scripting IDA Debugger to Deobfuscate Nymaim
  - 23.05.2016Technical Report about the Malware used in the Cyberespionage against RUAG

Whitepapers RSS feed

Subscribe to the whitepapers RSS feed to stay up to date and get notified about new whitepapers.

- Report an Incident

Close

Report an incident to NCSC

Report an incident: incidents[at]govcert{dot}ch
General inquiries: outreach[at]govcert{dot}ch

Point of contact for CERTs and CSIRTs

The following email address can be considered as point of contact for FIRST members and other CERTs/CSIRTs:

incidents[at]govcert{dot}ch

Encrypted Email

GovCERT.ch PGP-Key (preferred)
Alternative GovCERT.ch PGP Key (for older versions of PGP without Curve25519 support)
GovCERT.ch SMIME

- Statistics

Breadcrumbs

Introduction

Currently, we are observing HawkEye samples being distributed by large malspam waves. HawkEye[1] is a keylogger which has been around quite a long time (since 2013) and has evolved since then and gained more functionality. There are several good blog posts about HawkEye in general [2] [3]. Recently we observed an interesting obfuscation method in a HawkEye binary [4], which we are going to describe in this blog post.

Extracting Base32 encoded DLL

HawkEye is written in .NET and thus we can analyze it rather easily with the help of *dnSpy* [5].

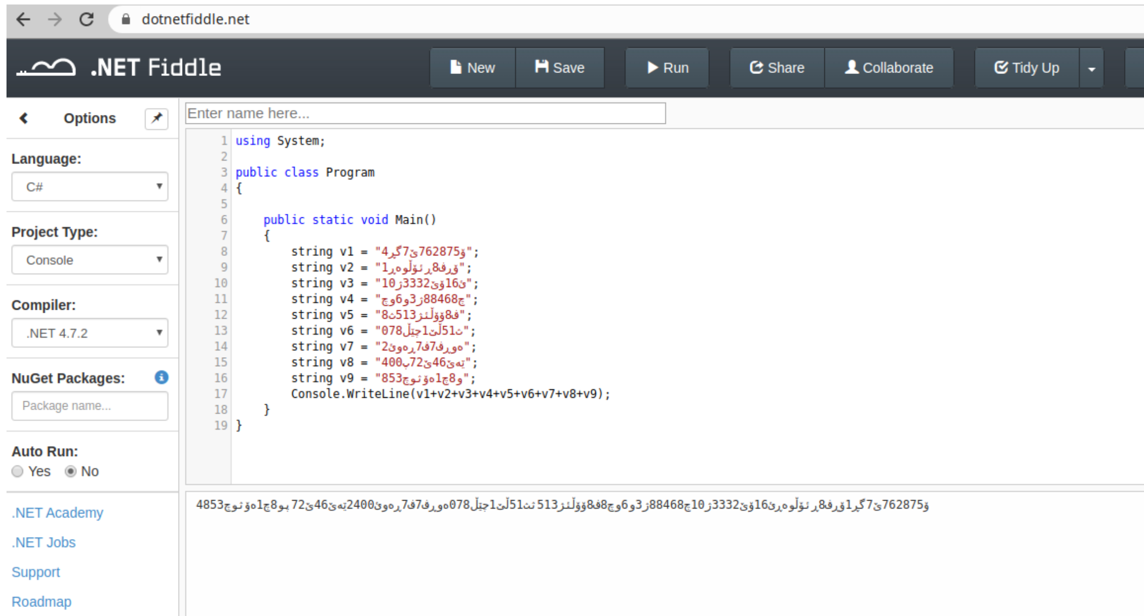
Looking through the decompiled source code in *dnSpy* we find the method `gate` in the class `Sinister`.

The Base32[6] encoded strings are concatenated, then reversed and decoded. The output of the decoded string is a DLL written in .NET as well. We can easily decode the string using `rev` and `base32` on a Linux system or with the help of similar tools.

After decoding, the DLL is loaded and the property `iraq` is set via reflection [7]. This value is a concatenated string consisting of the following nine static strings.

```csharp
using System;

namespace ParameterTransition
{
    // Token: 0x02000003 RID: 3
    internal class StructureCodes
    {
        // Token: 0x04000001 RID: 1
        public string v1 = "4ڕگ7ێ762875ۆ";

        // Token: 0x04000002 RID: 2
        public string v2 = "1ڕەولۆئڕ8ڨڕۆ";

        // Token: 0x04000003 RID: 3
        public string v3 = "10ژ3332ێۆ16ێ";

        // Token: 0x04000004 RID: 4
        public string v4 = "جو6و3ژ88468ج";

        // Token: 0x04000005 RID: 5
        public string v5 = "8ث513ژئلۆژ8ڨ";

        // Token: 0x04000006 RID: 6
        public string v6 = "078ڵچنڵ1یڵ51ث";

        // Token: 0x04000007 RID: 7
        public string v7 = "2ێوەڕ7ڨ7ڨڕوە";

        // Token: 0x04000008 RID: 8
        public string v8 = "400پ72ێ46ێەێ";

        // Token: 0x04000009 RID: 9
        public string v9 = "853جوتۆە1ج8و";
    }
}
```

Side note: The concatenated string used here is encoded with non-latin letters (maybe Farsi, any feedback most welcome). Trying to concatenate the strings in a text editor did not really work, however, using dotnetfiddle.net revealed the correct string. Update: According to our colleagues at FedPol/BKP the string is actually Urdu. Thanks for the hint, most appreciated.



)

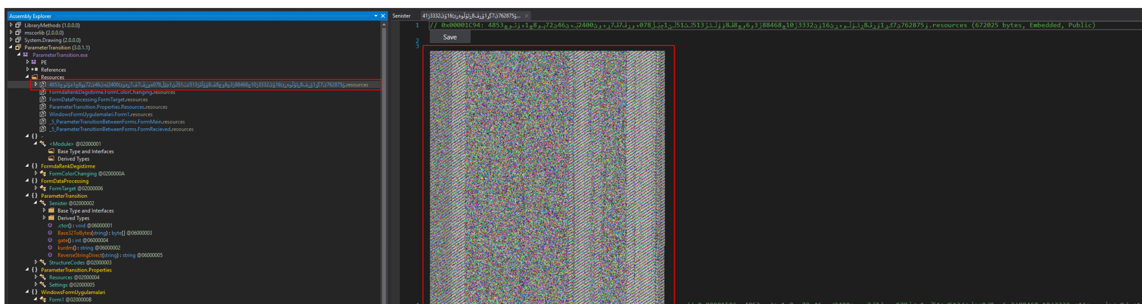
Extracting EXE file from embedded PNG

After opening the DLL in `dnspy`, we can examine the `set_iraq` method. The only thing this method does is to pass the argument to the method `ArgueMents.e1`.

The method `e1` loads and starts another executable file (lines 34–36). In order to analyze this executable, we need to know how it is loaded into the `rawAssembly` array and how we can extract it.

```
27          // Token: 0x06000003 RID: 3 RVA: 0x0000207C File Offset: 0x0000027C
28          public static void e1(string e4)
29          {
30              Bitmap i = ArgueMents.i10(e4);
31              byte[] i2 = ArgueMents.e5(i);
32              byte[] array = ArgueMents.e9(i2);
33              byte[] rawAssembly = array;
34              Assembly assembly = AppDomain.CurrentDomain.Load(rawAssembly);
35              MethodInfo entryPoint = assembly.EntryPoint;
36              entryPoint.Invoke(null, null);
37          }
38
39          // Token: 0x06000005 RID: 5 RVA: 0x00002190 File Offset: 0x00000390
40          public static Bitmap i10(string r11)
41          {
42              ResourceManager resourceManager = new ResourceManager(r11, Assembly.GetEntryAssembly());
43              return (Bitmap)resourceManager.GetObject(r11);
44          }
45
46          // Token: 0x06000004 RID: 4 RVA: 0x000020C4 File Offset: 0x000002C4
47          public static byte[] e5(Bitmap i1)
48          {
49              List<byte> list = new List<byte>();
50              checked
51              {
52                  int num = i1.Size.Width - 1;
53                  for (int j = 0; j <= num; j++)
54                  {
55                      int num2 = i1.Size.Height - 1;
56                      for (int k = 0; k <= num2; k++)
57                      {
58                          Color left = ArgueMents.i6(i1, j, k);
59                          bool flag = left != Color.FromArgb(0, 0, 0, 0);
60                          if (flag)
61                          {
62                              list.Add(left.R);
63                              list.Add(left.G);
64                              list.Add(left.B);
65                          }
66                      }
67                  }
68                  return list.ToArray();
69              }
70          }
71
72
73          // Token: 0x06000007 RID: 7 RVA: 0x0000221C File Offset: 0x0000041C
74          public static Color i6(Bitmap i18, int i19, int i20)
75          {
76              return i18.GetPixel(i19, i20);
77          }
```

Line 30 instantiates a `Bitmap` object with the value that was passed to the `set_iraq` method. Looking at the code of the method `i10` we see, that the image is loaded from the `ResourceManager` of the main binary. The bitmap is a PNG file, located in the resource section of the main executable.



Next, the `Bitmap` is passed to the method `e5`. This method reads the width and the height of the PNG. It then loops over the height (inner loop) and the width (outer loop) and stores the red, green and blue value of every pixel into a array and returns it. The alpha values as well as all pixels that are `r,g,b,a = (0,0,0,0)` are omitted.

The resulting array is then passed to the decryption method `e9`.

```
71
72      // Token: 0x06000006 RID: 6 RVA: 0x000021BC File Offset: 0x000003BC
73      public static byte[] e9(byte[] i15)
74      {
75          checked
76          {
77              byte[] array = new byte[i15.Length - 16 - 1 + 1];
78              Buffer.BlockCopy(i15, 16, array, 0, array.Length);
79              int num = array.Length - 1;
80              for (int j = 0; j <= num; j++)
81              {
82                  byte[] array2 = array;
83                  int num2 = j;
84                  array2[num2] ^= i15[j % 16];
85              }
86              return array;
87          }
88      }
```

In this method, a new array is created which is 16 bytes smaller than the array containing the color values. Then, the image array is copied to the new array (the first 16 bytes are omitted). The first 16 bytes are in fact the key needed to decrypt the rest of the data using `XOR`. There is a `for` loop, which will iterate over every value in the smaller array and XORs the value with the corresponding value of the key.

The decoded data is a PE file (exe), which is loaded into the memory. Finally the `EntryPoint` of the File is called.

As the key is stored in the Image itself, we can write a small python script to decrypt this and similar images.

```
#!/usr/bin/python3
from PIL import Image
import argparse

KEY_LENGTH = 16


def get_color_values(file_name):
    arr = bytearray()
    im = Image.open(file_name)
    w, h = im.size
    for i in range(w):
        for j in range(h):
            r, g, b, t = im.getpixel((i, j))
            # ignore zero values
            if (r, g, b, t) != (0, 0, 0, 0):
                arr.extend([r, g, b])
    return arr


def process(input_file, output_file):
    arr = get_color_values(input_file)
    key = arr[:KEY_LENGTH]
    data = arr[KEY_LENGTH:]

    for i in range(len(data)):
        data[i] ^= key[i % KEY_LENGTH]

    with open(output_file, "wb") as o:
        o.write(data)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("input_file")
    parser.add_argument("output_file")
    args = parser.parse_args()
    process(args.input_file, args.output_file)
```

In this case, the resulting PE file is once again heavily obfuscated, the first stage is obfuscated with *Babel Obfuscator* [8]. The final payload after several obfuscation rounds is HawkEye.

Conclusion

The obfuscation technique using a PNG file to store a PE file is neither new, nor very advanced. However we found it to be noteworthy nevertheless as we do not see it often. It could be an interesting way to bypass anti virus products, because the PE file is loaded directly into the memory, however in this case, the resulting HawkEye binary stores a copy of itself on the disk after the infection and therefore may be detected by AV products.

The detection rate of VirusTotal shows, that the “smuggling” technique is actually working. The initial malicious file [4] is detected by only 17 AV engines (two weeks after the first upload). The decrypted PE file[9] was detected by 34 engines right after upload.