Unpacking yet another .NET crypter

July 29, 2016 by Malwarebytes Labs

Last updated: August 17, 2016

In this post, we will study one of the malicious executables recently delivered by RIG Exploit Kit. It is packed in a .NET cryptor and includes similar features as one that we described some time ago (here). Similar packers are widespread and commonly used to protect various malware samples, that's why it is worth to know their common building blocks and methods of defeating them.

Analyzed sample

- 388548d34807fee28dede8b411063927 – original sample
  - 3a5cc47413cd815b44a0329100e552da – loader
  - 07a08cf5211665dfcd090e7bab6c8608 – payload (Neurevt Bot)

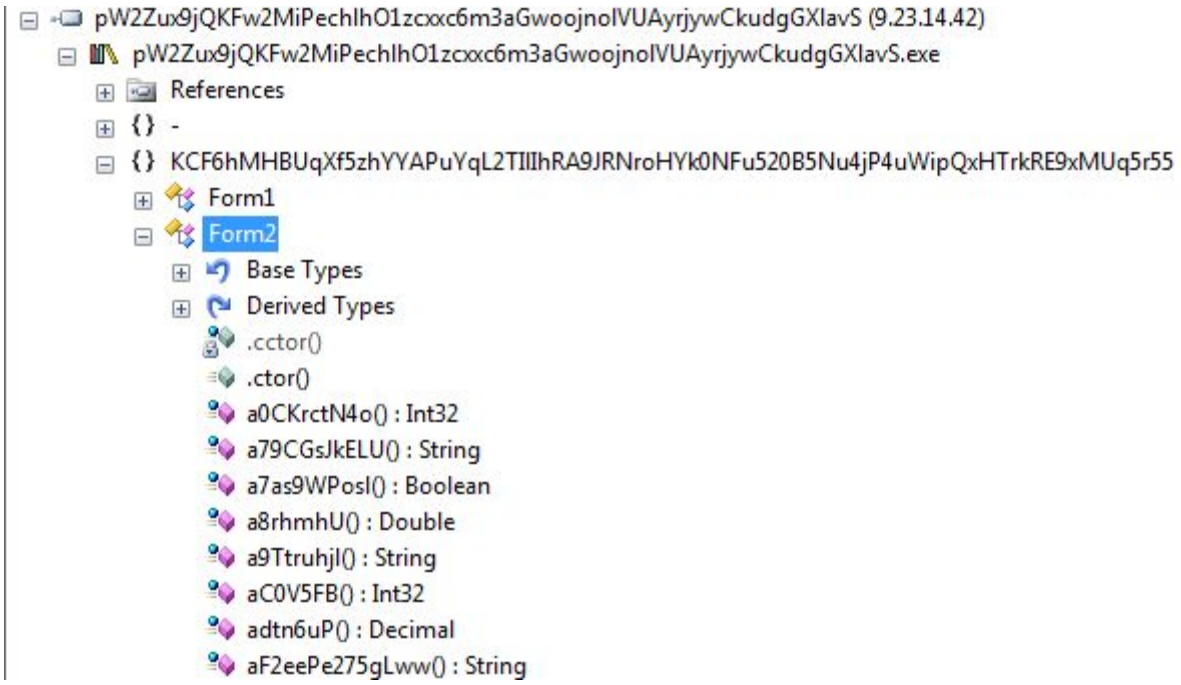
The interesting fact about this sample is that it comes signed:

Authenticode signature block and FileVersionInfo properties	
Signature verification	Signed file, verified signature
Signing date	9:55 PM 7/17/2016
Signers	[+] www.suaq.com

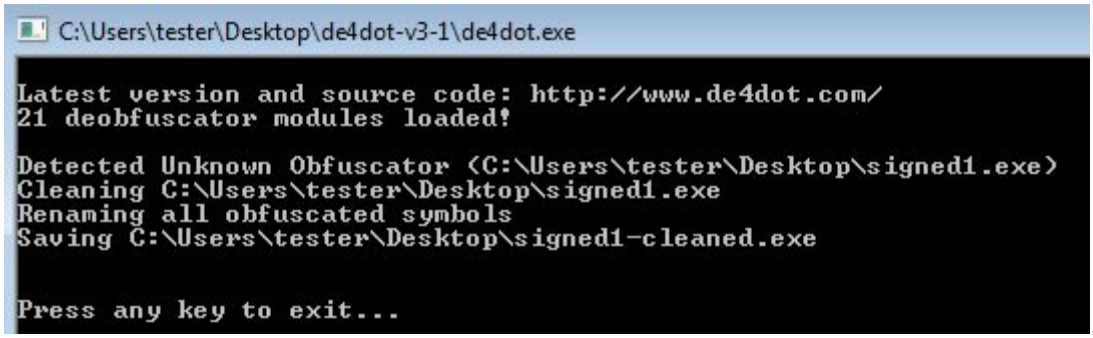
PE header basic information	
Target machine	Intel 386 or later processors and compatible processors
Compilation timestamp	2016-07-17 20:55:06
Entry Point	0x0003D9DE
Number of sections	3

Unpacking

The executable is written in .NET so we can decompile it using some of the popular tools made for this purpose (.NET Reflector, dnSpy, etc).

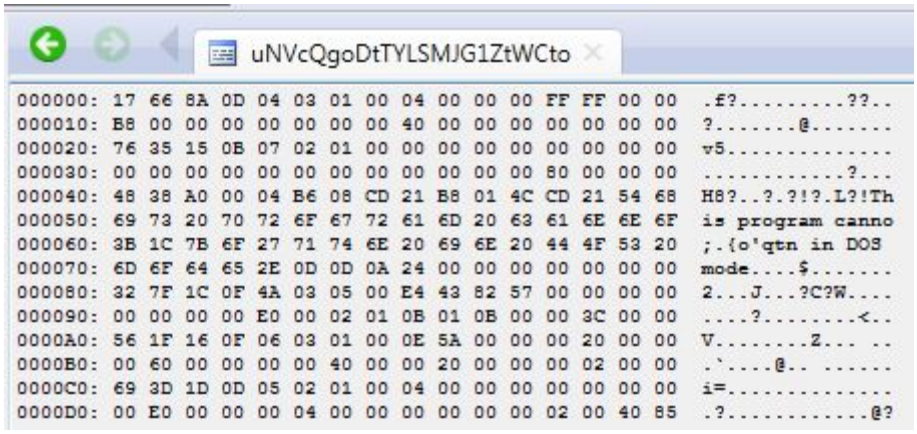


As we can see, the code is obfuscated – functions have garbled, meaningless names. Also the code inside contains lot of junk instructions and is difficult to follow. Even applying a known tool for .NET deobfuscation (de4dot) didn't helped much.



Anyway, let's start by finding the possible payload that is going to be unpacked.

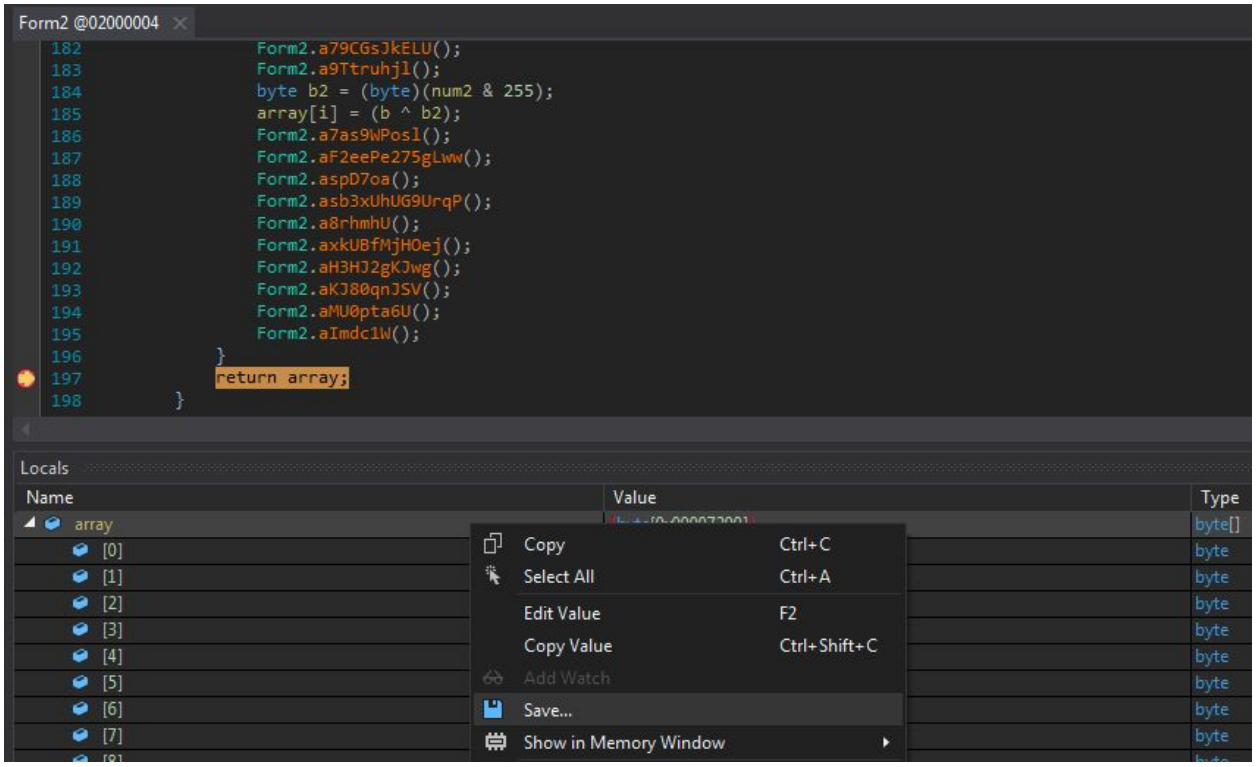
Looking at the resources we can see one element that looks like a distorted PE file:



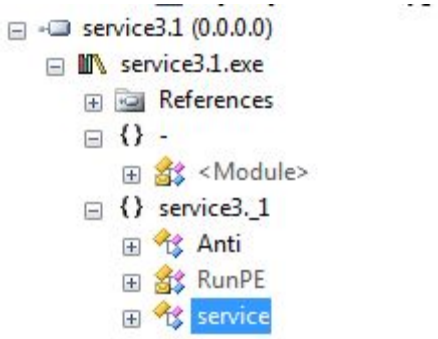
It is loaded and processed by the following function:

```
public static byte[] Aw0thGO34Z72LsABWmZu5kQDNQrNlBU1sQq5xoR6()
{
    byte[] buffer = (byte[]) Resources.ResourceManager.GetObject("uNVcQgoDtTYLSMJG1ZtWCto");
    byte[] bytes = Encoding.ASCII.GetBytes(kQDNQrNlBU1sQq5xoR5E2X2QDArgLHRsFtgsih6u8[1]);
    for (int i = 0; i < buffer.Length; i++)
    {
        int num2 = (i + 5) + bytes.Length;
        ao5CrzX1W6();
        aC0V5FB();
        aOr5gT49Qk();
        aivcCiy();
        aQB0so9q6DcVP();
        arm7Y7UM();
        byte num3 = buffer[i];
        int num4 = bytes[i % 0x1b] >> num2;
        aXr1OSNotZ();
        aFpJ0IaEhy();
        aTFEErLZ3hcoLt();
        adtn6uP();
        aRh7fySCat9rEI();
        a0CKrctN4o();
        aHLb6ia();
        a79CGsJkELU();
        a9TtruhjI();
        byte num5 = (byte) (num4 & 0xff);
        buffer[i] = (byte) (num3 ^ num5);
        a7as9WPosI();
        aF2eePe275gLww();
        aspD7oa();
        asb3xUhUG9UrqP();
        a8rhmhU();
        axkUBfMjHOej();
        aH3HJ2gKJwg();
        aKJ80qnJSV();
        aMU0pta6U();
        aImdc1W();
    }
    return buffer;
}
```

Using dnSpy we can set breakpoint at the end of this function, run it and dump the output buffer.



The dumped binary turned out to be another PE file written in .NET (3a5cc47413cd815b44a0329100e552da). However, it is not the malicious payload that we are looking for, but just another element of the crypter – a loader. It unpacks the real payload and injects it in another binary using RunPE technique (also known as process hollowing).



The loader is not independent – it relies on resources from the previous file. We can see from the code that the resource "*varitoyp*" contains a set of parameters. It is decrypted by a function ***DeCrypt***, using a word "*params*" as the decryption key:

```
[MethodImpl(MethodImplOptions.NoOptimization | MethodImplOptions.NoInlining)]
public static void service()
{
    try
    {
        ResourceManager manager = new ResourceManager(Assembly.GetEntryAssembly().GetManifestResourceNames()[0].Replace(".resources", ""), Assembly.GetEntryAssembly());
        byte[] data = (byte[]) manager.GetObject("varitoyp");
        param = Strings.Split(Encoding.ASCII.GetString(DeCrypt(ref data, "params", 0)), "|", -1, CompareMethod.Binary);
        Thread.Sleep(Conversions.ToInteger(param[10]));
        string keys = param[30];
        if (param[0] == "1")
```

The real payload is hidden inside of another encrypted resource. The name of the file, as well as the decryption key is included in the parameters that are decrypted in the previous step:

```
data = (byte[]) manager.GetObject(param[0x1d]);
mainFile = DeCrypt(ref data, keys, 0);
if (param[0x1f] == "1")
{
    InPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), "vbc.exe");
}
else if (param[0x1f] == "2")
{
    InPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), "RegAsm.exe");
}
else if (param[0x1f] == "3")
{
    InPath = Path.Combine(RuntimeEnvironment.GetRuntimeDirectory(), "AppLaunch.exe");
}
else if (param[0x1f] == "4")
{
    InPath = Path.Combine(Environment.SystemDirectory, "svchost.exe");
}
else if (param[0x1f] == "5")
{
    InPath = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.System), "notepad.exe");
}
else if (param[0x1f] == "6")
{
    InPath = Application.ExecutablePath;
}
if (param[0x20] == "1")
{
    Thread thread = new Thread(new ThreadStart(service.loader));
    thread.SetApartmentState(ApartmentState.STA);
    thread.Start();
}
```

The payload can be injected into one of the predefined executables: *vbc.exe*, *RegAsm.exe*, *AppLaunch.exe*, *notepad.exe* – or, eventually, its own process. The choice is made based on one of the parameters from the encrypted set.

The decryption algorithm is custom XOR based:

```
public static byte[] DeCrypt(ref byte[] Data, string Keys, uint ExtraRounds = 0)
{
   byte[] bytes = Encoding.Default.GetBytes(Keys);
   int num2 = (int) ((Data.Length - 1) * (ExtraRounds + 1L));
   for (int i = 0; i <= num2; i++)
   {
      Data[i % Data.Length] = (byte) ((((Data[i % Data.Length] ^ bytes[i % bytes.Length]) - Data[(i + 1) % Data.Length]) + 0x100) % 0x100);
   }
   Array.Resize<byte>(ref Data, Data.Length - 1);
   return Data;
}
```

Using a copy of this function we can easily decrypt the dumped resources from the initial binary. We were able to reconstruct a sample decoder, you can find the python script here: msil_dec.py.

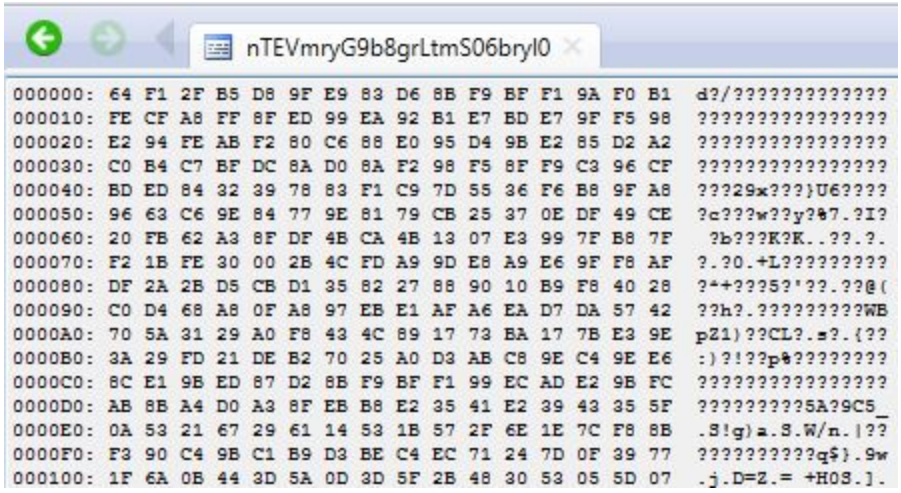
Decrypting parameters:

```
./msil_dec.py --file varitoyp --key params
0|0|0|0|0|0|0|0|0|0|10000|0|0|0|0|0|0|0|0|0|0|LgunkLBEWL7f5asOISuri|0|0|0
|0|0|0|nTEVmryG9b8grLtmS06bryl0|ZxjmzvjUYrFNhuAOygWpbtsxcVZ|6|0|
```

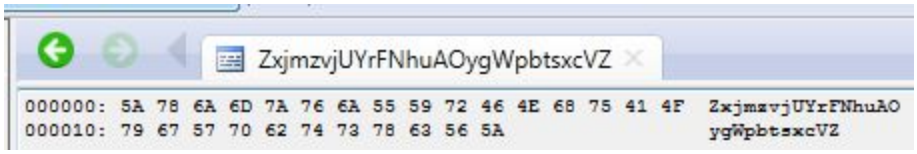
The parameters are in the form of a string, containing values separated by a delimiter. Parameters 30 and 31 contains the name of the resource hiding the encrypted payload and the key.

```
nTEVmryG9b8grLtmS06bryl0, //binary
ZxjmzvjUYrFNhuAOygWpbtsxcVZ, //key
```

The encrypted executable is stored in the resources of the initial binary:



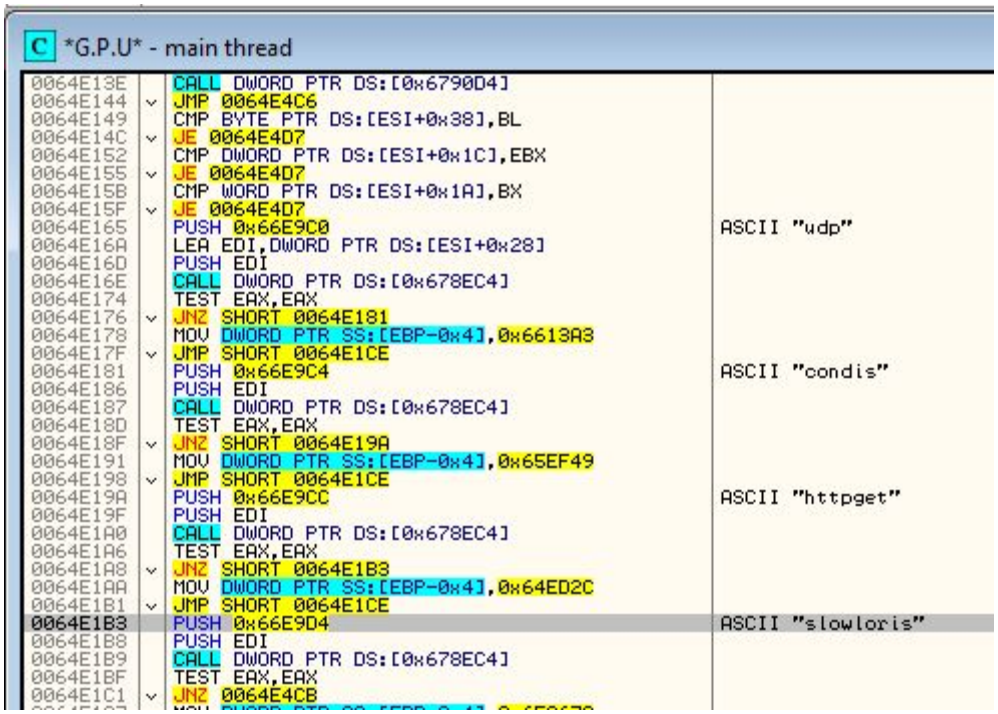
So is the key:



Decrypting:

./msil_dec.py --file nTEVmryG9b8grLtmS06bryl0 --key ZxjmzvjUYrFNhuAOygWpbtsxcVZ > payload.exe

As a result we get the final payload (07a08cf5211665dfcd090e7bab6c8608) – it is a Neurevt Bot, used i.e. for credential stealing and DDoS attacks (read more here).



Conclusion

This cryptor probably shares some code with the previous described one – it might even be the work of the same authors. Again, we see a loader with another PE file packed inside. Also, again there is an array of parameters. Finally, the list of the applications where the payload is injected is exactly the same in both cases. In the previous cryptor, a BMP file was used to hide

encrypted data (configuration and the final payload). This time authors gave up applying any steganographic tricks.

After almost a year from the previous release, we cannot say that the product evolved to something more complex. Instead – we see the same ideas, however mutated and implemented differently.

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.