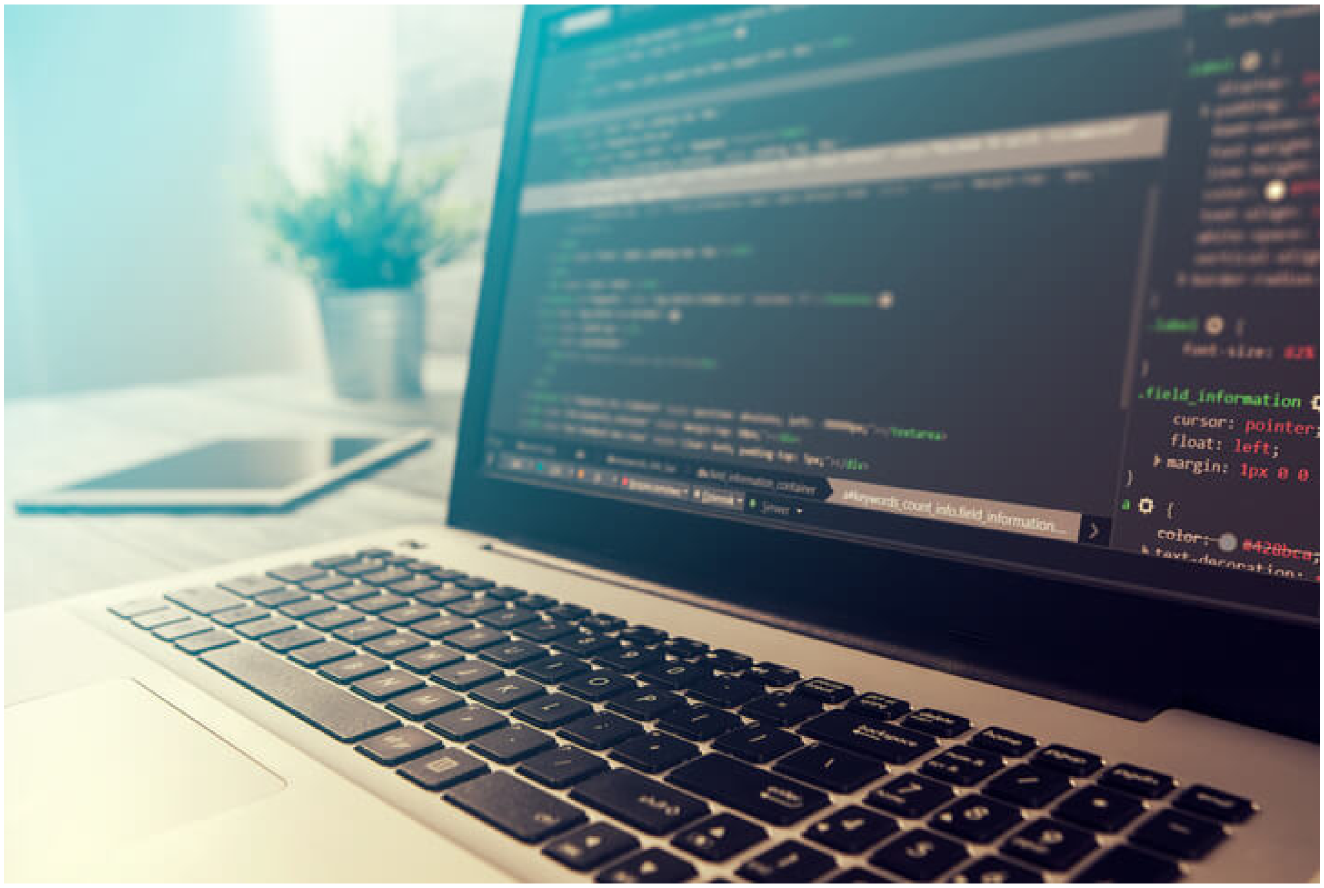
Analysis of New Agent Tesla Spyware Variant

fortinet.com/blog/threat-research/analysis-of-new-agent-tesla-spyware-variant.html

April 5, 2018



Threat Research

By Xiaopeng Zhang | April 05, 2018

Recently, FortiGuard Labs captured a new malware sample that was spread via Microsoft Word documents. After some quick research, I discovered that this was a new variant of the Agent Tesla spyware. I analyzed another sample of this spyware last June and published a blog about it. In this blog, I want to share what's new in this new variant.

This malware was spread via a Microsoft Word document that contained an embedded exe file. Figure 1 below shows what it looks like when you open the Word document.

Figure 1. Opening the malicious Word document

As you can see, it asks the victim to double click the blue icon to enable a "clear view." Once clicked, it extracts an exe file from the embedded object into the system's temporary folder and runs it. In this case, the exe file is called "POM.exe".

Figure 2. POM.exe is created in a temporary folder

Analysis of POM.exe

Figure 3. Looking at POM.exe in an analysis tool

In figure 3 we can see that the malware is written in the MS Visual Basic language. Based on my analysis, it’s a kind of installer program. When it runs, it drops two files: “filename.exe” and “filename.vbs” into the “%temp%\subfolder”. It then exits the process after executing the file “filename.vbs”. Below, in figure 4, is the content of “filename.vbs”.

Figure 4. The content of filename.vbs

To make it run automatically when the system starts, it adds itself (runs filename.vbs) to the system registry as a startup program. It then runs “%temp%\filename.exe”.

Figure 5. The malware adds itself into the system registry as “RunOnce” item

Analysis of filename.exe

When “filename.exe” starts, like most other malware it creates a suspended child process with the same name to protect itself. It then extracts a new PE file from its resource to overwrite the child process memory. Afterwards, it resumes the execution of the child process. This is when it executes the code of that new PE file, which is the main part of this malware.

Figure 6. Checking to see if the module mscorjit.dll is loaded

Let’s go on to the analysis of the child process. It first checks to see if the environment value of "Cor_Enable_Profiling" is set to 1, and if the modules "mscorjit.dll" and "clrjit.dll" have been loaded (see figure 6). If one of these checks is true, it exits the process without doing anything. So far, I have no idea what the purpose of doing that is, but it is likely anti-something.

If the process doesn’t exit, it loads a named resource. The resource name is "__", which is a string decrypted from a local variable. Afterwards, by calling the API functions “FindResource” and “LoadResource”, it can read the resource data to the process memory. Figure 7 shows the “__” resource in CFF Explorer. For sure, the data is encrypted.

Figure 7. Encrypted “__” resource

By decrypting the “__” data, we obtain another PE file, which is a .Net framework program. This is to be loaded into the child process memory. It reads sections of the .Net program into memory according to the PE file headers, imports APIs defined in the import table for .Net programs, relocates offset of the function “_CorExeMain”, as well as builds the .Net framework running environment by calling several APIs. Finally, it jumps to the entry point of the .Net program where it later jumps to "_CorExeMain" – which is the entry point of all .Net programs – to execute this .Net program. You can see in figure 8 how it jumps to the "_CorExeMain" function.

Figure 8. Jumping to the entry point of the .Net program

In order to further analyze the .Net program, I dumped it from the child process memory into a local file. This allowed me to launch it independently rather than running it within the child process. This also allowed me to load it into the .Net program analysis tools to analyze it.

Deep analysis of the .Net program

The dumped file has an incorrect PE header. I manually repaired it so that it can be executed, debugged, and parsed by .Net program analysis tools. Figure 8 shows the main function of the .Net program in an analysis tool.

Figure 9. The main function of the .Net program

As you may have already noticed, it uses some kind of code obfuscation technique to increase the difficulty of code analysis. In the following parts, you may see that some of the names of method, class, variable, etc. have been modified to make them understandable.

All the constant strings in the .Net program are encoded and saved within a large buffer, and every string is assigned an index. Whenever it needs to use the string, it calls a function with its string index to get the string. If the string is encoded, it throws the encoded string into another function to get it decoded. In figure 10 we can see that it reads the huge string into the big buffer—"Pkky9noglfauhKN1Fjq.QOZ4uWBaWw".

Here is an example:
"XtL6rF5GoidQVxdCxi.R6ybT342I(Pkky9noglfauhKN1Fjq.Y3LpEpC6nY(3172));"

"3172" is the string index.

The "Pkky9noglfauhKN1Fjq.Y3LpEpC6nY" function picks up the string of index 3172 from that large buffer. In this case, it's "hyNN5z+7qAsS695lDXLuHg==".

"XtL6rF5GoidQVxdCxi.R6ybT342I" is the decoding function. After decoding, we get the string "True\x00\x00\x00\x00\x00\x00\x00\x00". i.e. "True".

Figure 10. Reading strings in the large buffer

When the main function is called, it first pauses 15 seconds by calling "Thread::Sleep()" function. This allows it to potentially bypass sandbox detection.

As my analysis in the previous blog showed, Agent Tesla is a spyware. It monitors and collects the victim's keyboard inputs, system clipboard, screen shots of the victim's screen, as well as collects credentials of a variety of installed software. To do that it creates many different threads and timer functions in the main function. So far, through my quick analysis, this version is similar to the older one. As I did not find much change, I won't talk about it more here but simply refer you to the previous blog analysis.

However, the way of submitting data to the C&C server has changed. It used to use HTTP POST to send the collected data. In this variant, it uses SMTPS to send the collected data to the attacker's email box.

Based on my analysis, the commands used in the SMTP method include "Passwords Recovered", "Screen Capture", and "Keystrokes", etc. The commands are identified within the email's "Subject" field. For example:

"System user name/computer name Screen Capture From: *victim's IP"*

Here's an example to show you how it sends the collected credential data to the attacker's email address. Figure 10 shows the email content that will be sent out with my PC information along with the collected credentials. It enables an SSL function and uses TCP port 587. The "Body" field is the collected data in HTML format. The "Subject" field contains the command "Passwords Recovered" which tells the recipient that this email contains credentials.

Figure 11. Email content with collected data

The attacker registered a free zoho email account for this campaign to receive victims' credentials. Figure 11, below, shows the SMTP server and its login information. You can see the attacker's SMTP credential "UserName" and "Password" as well as the SMTP server.

Figure 12. Attacker's SMTP credential

When the email is sent out through the Wireshark tool, we were able to capture the packets shown in figure 12, below.

Figure 13. Collected data submission using SMTPS in wireshark

As I explained above, the collected data in the mail body is in html format. I copied the html content into a local html file and was able to open it in the IE brower to see what the malware had harvested from my test enviroment. In figure 13, you can see the screenshot of my PC information along with the related credentials in an IE browser.

Figure 14 Harvested Credentials

Daemon program

It also drops a daemon program from the .Net program’s resource named “Player” into the “%temp%” folder and run it up to protect “filename.exe” from being killed.

Figure 15. Dropping the daemon program and running it

The daemon program’s name is made up of three random letters, as you can see in figure 15. It’s also a .Net program and its main purpose is very clear and simple. Figure 16 shows the daemon program’s entire code in an analysis tool.

You can see that the main function receives a command line argument (for this sample, it’s the full path to “filename.exe”.) and saves it to a string variable called “filePath”. It creates a thread, and in the thread function it checks to see if the file “filename.exe” is running in each 900 millisecond. It runs it again whenever the “filename.exe” is killed.

Figure 16. Daemon program code

Solution

The file “PPSATV.doc” has been detected as “**W32/VBKrypt.DWSS!tr**”, and “POM.exe” has been detected as “**W32/VBKrypt.DWSS!tr**” by FortiGuard AntiVirus service.

We have informed Zoho of the email account which is being used in this AgentTesla campaign.

IoC:

Sample SHA256:

PPSATV.doc

13E9CDE3F15E642E754AAE63259BB79ED08D1ACDA93A3244862399C44703C007

POM.exe

A859765D990F1216F65A8319DBFE52DBA7F24731FBD2672D8D7200CC236863D7

filename.exe

B4F81D9D74E010714CD227D3106B5E70928D495E3FD54F535B665F25EB581D3A

Random name daemon program

C2CAE82E01D954E3A50FEAEBCD3F75DE7416A851EA855D6F0E8AAAC84A507CA3

Check out our latest Quarterly Threat Landscape report for Q4 of 2017 for more details about recent threats.

Sign up for our weekly FortiGuard intel briefs or to be a part of our open beta of Fortinet's FortiGuard Threat Intelligence Service.

Related Posts