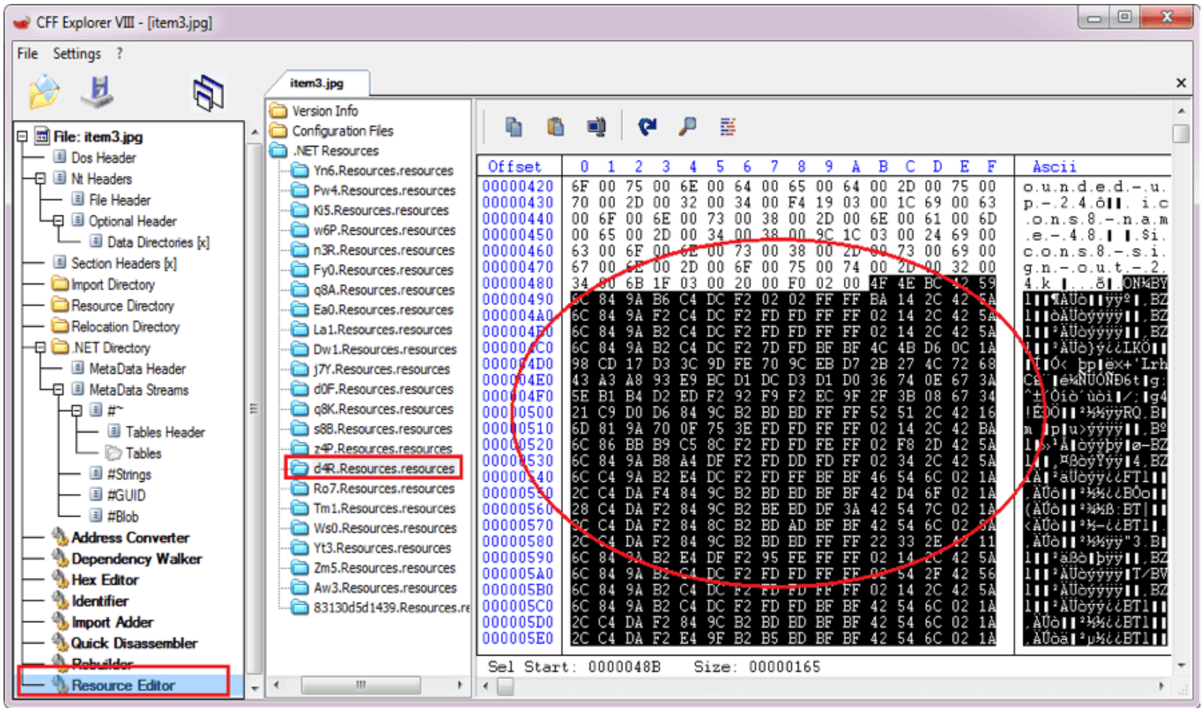
Deep Analysis: New FormBook Variant Delivered in Phishing Campaign – Part I

fortinet.com/blog/threat-research/deep-analysis-new-formbook-variant-delivered-phishing-campaign-part-I

April 12, 2021



```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00000420  6F 00 75 00 6E 00 64 00 65 00 64 00 2D 00 75 00  o.u.n.d.e.d.-.u.
00000430  70 00 2D 00 32 00 34 00 F4 19 03 00 1C 69 00 63  p.-.2.4.ô▮▮. i.c
00000440  00 6F 00 6E 00 73 00 38 00 2D 00 6E 00 61 00 6D  .o.n.s.8.-.n.a.m
00000450  00 65 00 2D 00 34 00 38 00 9C 1C 03 00 24 69 00  .e.-.4.8.▮ ▮.$i.
00000460  63 00 6F 00 6E 00 73 00 38 00 2D 00 73 00 69 00  c.o.n.s.8.-.s.i.
00000470  67 00 6E 00 2D 00 6F 00 75 00 74 00 2D 00 32 00  g.n.-.o.u.t.-.2.
00000480  34 00 6B 1F 03 00 20 00 F0 02 00 4F 4E BC 42 59  4.k ▮...ð▮.ON¼BY
00000490  6C 84 9A B6 C4 DC F2 02 02 FF FF BA 14 2C 42 5A  l▮▮¶ÄÜò▮▮ÿÿº▮,BZ
000004A0  6C 84 9A F2 C4 DC F2 FD FD FF FF 02 14 2C 42 5A  l▮▮òÄÜòýýÿÿ▮▮,BZ
000004B0  6C 84 9A B2 C4 DC F2 FD FD FF FF 02 14 2C 42 5A  l▮▮²ÄÜòýýÿÿ▮▮,BZ
000004C0  6C 84 9A B2 C4 DC F2 7D FD BF BF 4C 4B D6 0C 1A  l▮▮²ÄÜò}ý¿¿LKÖ▮▮
000004D0  98 CD 17 D3 3C 9D FE 70 9C EB D7 2B 27 4C 72 68  ▮Í▮Ó< þp▮ë×+'Lrh
000004E0  43 A3 A8 93 E9 BC D1 DC D3 D1 D0 36 74 0E 67 3A  C£¨▮é¼ÑÜÓÑÐ6t▮g:
000004F0  5E B1 B4 D2 ED F2 92 F9 F2 EC 9F 2F 3B 08 67 34  ^±´Òíò´ùòì▮/;▮g4
00000500  21 C9 D0 D6 84 9C B2 BD BD FF FF 52 51 2C 42 16  !ÉÐÖ▮▮²½½ÿÿRQ,B▮
00000510  6D 81 9A 70 0F 75 3E FD FD FF FF 02 14 2C 42 BA  m ▮p▮u>ýýÿÿ▮▮,Bº
00000520  6C 86 BB B9 C5 8C F2 FD FD FE FF 02 F8 2D 42 5A  l▮»¹Å▮òýýþÿ▮ø-BZ
00000530  6C 84 9A B8 A4 DF F2 FD DD FD FF 02 34 2C 42 5A  l▮▮¸¤ßòýÝýÿ▮4,BZ
00000540  6C C4 9A B2 E4 DC F2 FD FF BF BF 46 54 6C 02 1A  lÄ▮²äÜòýÿ¿¿FTl▮▮
00000550  2C C4 DA F4 84 9C B2 BD BD BF BF 42 D4 6F 02 1A  ,ÄÚô▮▮²½½¿¿BÔo▮▮
00000560  28 C4 DA F2 84 9C B2 BE BD DF 3A 42 54 7C 02 1A  (ÄÚò▮▮²¾½ß:BT|▮▮
00000570  3C C4 DA F2 84 8C B2 BD AD BF BF 42 54 6C 02 0A  <ÄÚò▮▮²½-¿¿BTl▮.
00000580  2C C4 DA F2 84 9C B2 BD BD FF FF 22 33 2E 42 11  ,ÄÚò▮▮²½½ÿÿ"3.B▮
00000590  6C 84 9A B2 E4 DF F2 95 FE FF FF 02 14 2C 42 5A  l▮▮²äßò▮þÿÿ▮▮,BZ
000005A0  6C 84 9A B2 C4 DC F2 FD FD FF FF 02 54 2F 42 56  l▮▮²ÄÜòýýÿÿ▮T/BV
000005B0  6C 84 9A B2 C4 DC F2 FD FD FF FF 02 14 2C 42 5A  l▮▮²ÄÜòýýÿÿ▮▮,BZ
000005C0  6C 84 9A B2 C4 DC F2 FD FD BF BF 42 54 6C 02 1A  l▮▮²ÄÜòýý¿¿BTl▮▮
000005D0  2C C4 DA F2 84 9C B2 BD BD BF BF 42 54 6C 02 1A  ,ÄÚò▮▮²½½¿¿BTl▮▮
000005E0  2C C4 DA F2 E4 9F B2 B5 BD BF BF 42 54 6C 02 1A  ,ÄÚòä▮²µ½¿¿BTl▮▮
Sel Start: 0000048B    Size: 00000165
```

Threat Research

By Xiaopeng Zhang | April 12, 2021

FortiGuard Labs Threat Research Report

Affected platforms: Microsoft Windows
Impacted parties: Windows Users
Impact: Collect Sensitive Information from Victim’s Device.
Severity level: Critical

FortiGuard Labs captured a phishing campaign that was sending a Microsoft PowerPoint document as an email attachment to spread the new variant of the FormBook malware. FormBook is a well-known commercial malware, so dubbed because it has been sold “as-a-service” on hacking forums since 2016. It is designed to steal personal information from victims' devices and manipulate their devices using control commands from a C2 server. FormBook, which has been detected in the wild for over five years, is designed to steal personal information through the use of keyloggers and form grabbers to collect victim input along with the data of some software, such as browsers, IM, Email clients, and FTP clients.

I recently conducted in-depth research on the latest campaign we captured, starting with the phishing email and the attached PowerPoint document that delivers the malware. This is the first part of that analysis. In it, I will demonstrate all my findings from the research, including but not limited to how the malicious VBA code is executed in the PowerPoint file; how the FormBook payload file is downloaded by the PowerPoint file; as well as how the FormBook main file (module) is finally extracted from a .Net module. In part II, I will look at what the FormBook malware does once loaded, and in particular, the new functions and features in this latest variant.

Phishing Email and PowerPoint Document

The phishing email used to deliver the FormBook malware looks like a reply to a request for a purchase order. Of course, this is simply a crafted fake message to the victim. Figure 1.1 shows the email content. It is designed to lead the victim to open the attached PowerPoint file to view the details of “brochures and prices” in a video.

Figure 1.1 - The phishing email captured by FortiGuard Labs

The PowerPoint file is delivered with a “.pps” file extension, which tells PowerPoint to open it directly in “slide show” view (not in “edit” view, which is the default for other file extensions like “.ppt”) once the recipient double-clicks on the file.

Figure 1.2 is the screenshot of the slide content once it begins. While the victim is moving their mouse around the slide show or clicking on items, the malicious VBA code is being executed in background. Through my analysis, it sets two actions (“Mouse Click” and “Mouse Over”) to run the Macro function MRmgEG() once triggered. Figure 1.3 only shows the “Action Settings” for the “Mouse Over” event. These are the same as the settings for the “Mouse Click” event.

Figure 1.2 – Beginning of the slide show after double-clicking on the PowerPoint file

Figure 1.3 - The Mouse Over action is set to execute the Macro MRmgEG()

Once MRmgRE() is called, it then calls several methods and finally runs a piece of PowerShell code by calling the function RPUJob() (which is an alias of API WinExec()), as displayed in Figure 1.4. I have deobfuscated the PowerShell command at the bottom of the Figure for you to clearly observe it.

Figure 1.4 - Macro executing a PowerShell code

The method Invoke-RestMethod() requests a PowerShell code from “hxxps[:]//bit[.]ly/2NDvCoG”. “bit.ly” is a website that provides a URL shortening service. In this case, the short URL links to “hxxp[:]//kiibra[.]com/images/index.jpg”. “index.jpg” is not a real picture file, but a PowerShell file that will be executed by IEX().

I manually downloaded the file “index.jpg” and my static analysis found that the PowerShell code would extract two files (“item1.gif” and “item2.png”) into the system’s temporary folder from two variables with base64 encoded data. Below is a key piece of the PowerShell code that I copied from “index.jpg”.

```
[IO.FIlE]::WRiTeAlLBYtES("$Env:TMP\item1.gif",
        [SySTeM.ConveRT]::frOMbaSe64CHArARRay($SAKeLtXeLERX, 0,

        $SAKeLtXeLERX.LENgTH));

[IO.FilE]::WrItEALLByTeS("$Env:TMP\item2.png",
        [SyStem.CONVErt]:: fROmbaSe64ChaRaRrAy($vVXFiQawOSBg, 0,
        $vVXFiQawOSBg.LENGthh));

cMd.exe /C COPy /B '%TMP%\item1.gif' + '%TMP%\item2.png'
'%TMP%\item3.jpg';

del "$Env:TMP\item1.gif";

deL "$Env:TMP\item2.png";

CmD.eXe /c '%TMP%\item3.jpg'
```

These two variables, with huge amounts of base64 encoded data, are “$SAKeLtXeLERX” (that saves to “item1.gif”) and “$vVXFiQawOSBg” (that saves to “item2.png”). The two files are actually pieces of an executable file, which then are combined into a single file called “item3.jpg” (by executing the “Copy” command), which is eventually run by calling the command “cmd.exe /c %TMP%\item3.jpg”. Figure 1.5 is a screenshot of the files residing in my testing environment’s %TEMP% folder.

Figure 1.5 - Generated files used to execute the PowerShell code

Initial Static Analysis on item3.jpg in Phishing Campaign

I then dragged the file into a PE analysis tool, as shown in Figure 2.1 below. We can see that it is a 32-bit executable file that was developed in Microsoft Visual C#.

Figure 2.1 - “item3.jpg” in PE analysis tool

Using the view provided in the .Net analysis tool “dnSpy”, shown in Figure 2.2, I noticed that the code of “item3.jpg” is fully obfuscated, including classes, methods, properties, and code flow. All the names are totally randomly generated and meaningless so you cannot obtain any clues from them. In addition, an obfuscated code-flow can confuse the code order so you cannot easily know where the next code goes. All of these techniques are a huge challenge for an analyst trying to comprehend the code and trace the code-flow. Since the module name is “Li7f”, I will refer to it using the module name in following content.

Figure 2.2 - The obfuscated code of “item3.jpg” and its entry point function

Analyzing the “Li7f” Module in Debugger

The malware developers not only obfuscated the code, but also encrypted all constant strings. However, there is a special method that can be used for dynamically decrypting constant strings by their string ID from a large string array, just before the constant string is being used. Just as you can see in Figure 3.1, it decrypts two strings: “https://www.bing.com” and “https://www.google.com”. You may have noticed that the names of class and method have been renamed to be meaningful for better tracking and to better understand its code. The renamed method is now “cls_x2C.decrypt_string_g4J()”, while its original name was “x2C.g4J()”. During my analysis, I renamed many other classes, methods, and variables, which you will see in this blog, to make them easier to track.

Figure 3.1 - Decrypting two Host strings

To confirm that the victim’s device is connecting to the internet, the “Li7f” module sends normal “Get” requests to the two hosts, as shown above in Figure 3.1. It then continues its task only when it can receive responses from both of the two host servers.

There are twenty-three .Net resources in “item3.jpg” that are used to save encrypted data. “Li7f” retrieves the data of “d4R.Resources” into memory, which is Dictionary data (key/value). To finish this, it creates a ResourceManager object with a resource name and a ResourceSet object (Dictionary object). It then obtains the value of the key "LoginDataBase", which is an encrypted PE file. Figure 3.2 shows all of the .Net resources of “item3.jpg” and the encrypted PE (value of key “LoginDataBase”) in the resource “d4R.Resources”.

Figure 3.2 - .Net Resources and the encrypted PE in “d4R.Resources”

Figure 3.3 - The decrypted value of the key “LoginDataBase”

Figure 3.3 is the decrypted PE file in the memory from the key “LoginDataBase” of the resource “d4R.Resources”. The PE file is a .Net DLL module that is dynamically deployed by the “Li7f” module. I will refer to this .Net module as “zOAI” in this blog, according to its project name.

The “zOAI” module’s entry method is invoked in the “Li7f” module, which defines a delegate type “Fx7” and assigns it to link to the entry method “CaCl.aXt()“ of the “zOAI” module. So, calling a delegate method defined by “Fx7” type in the “Li7f” module is equivalent to calling “CaCl.aXt()“ in the “zOAI” module.

An “Fx7” type is defined in the class “cls_form_m3N” in the “Li7f” module as follows:

internal delegate void Fx7(); //It is similar to the function pointer in C and C++ language.

It is assigned by calling “obj = Delegate.CreateDelegate (A_0, A_1, A_2, A_3, A_4);”, where “A_0” is the type (“cls_form_m3N.Fx7”) of delegate to create; “A_1” is a target class name(“zOAI!Cacl”) that develops the target method; “A_2” is the target method “aXt”.

After that, the variable “obj” is linked to the method “zOAI!Cacl.aXt()”.

And finally, “Li7f” invokes the “zOAI”’s entry method “zOAI!Cacl.aXt()” using the pseudo code “((cls_form_m3N.Fx7)(delegate)obj)();”.

Deep Dive into “zOAI” Module

The “zOAI” module is same as the “Li7f” module that is fully obfuscated. As you can see in Figure 4.1, the names of classes and methods are random strings, and the code-flow is also obfuscated.

Figure 4.1 – The obfuscated “zOAI” module and entry method “aXt()”

The “zOAI” module retrieves the current call back frame by calling StackFrame Last<StackFrame>(). Through the last one (which is “Li7f!Zx4.m3N.Rt7()”—refer to Figure 2.2), “zOAI” obtains the image of the module “Li7f”. Furthermore, it proceeds to retrieve another .Net resource from “Li7f” (item3.jpg). It parses the TimeStamp value from the PE structure of item3.jpg to generate the name (“83130d5d1439.Resources”) of the .Net resource to read, which is shown in Figure 4.2. Looking back at Figure 3.2, you will find that “83130d5d1439.Resources” is the last of the .Net resources.

Figure 4.2 - Generated resource name from the TimeStamp value

The data of this resource that is also encrypted contains a dictionary data structure with many key/value pairs. Figure 4.3, below, is the decrypted and loaded dictionary.

Figure 4.3 – The Dictionary data that is read from “83130d5d1439.Resources”

It contains many switches explaining what “aOZI” module is able to do. For example,

- How to perform startup persist (%StartupPersist%)
- How to show up a fake message to victim (%Message%)
- How to do the anti-VM environment (%VM%) and anti-sandbox detection (%SB%)
- And so on

As shown in Figure 4.3, all of them above are set to false; hence, the “zOAI” module will bypass those operations and detections. The most significant data is %MainFile%, which is a decrypted PE file (file size 0x2d600). It is the payload file of this variant of FormBook, which I’ll talk about later.

It then decrypts a constant string, "AddInProcess32.exe", with the index 2444184564U. This file belongs to the Microsoft® .NET Framework and is located in the .Net Framework installation folder. It then copies this file from "C:\Windows\Microsoft.NET\Framework\v4.0.30319\AddInProcess32.exe" (in my testing environment) to "%Temp%\AddInProcess32.exe" by calling an imported native API CopyFileEx, defined below in module “zOAI”:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto, EntryPoint = "CopyFileEx", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
internal static extern bool FO18xSYnaRX8DJ59Xlc280GJqLA([MarshalAs(UnmanagedType.VBByRefStr)] ref string, [MarshalAs(UnmanagedType
VeQ.MAN5S2ZmxZ35Z7m3YKy0VksroQA, IntPtr, ref int, VeQ.Qs6h1EUA15Rsz9R5qN5eapIAj63);
```

The last thing that the “zOAI” module does is to decrypt another .Net module from a local array variable and then deploys it in the memory as well as invokes its entry method with parameters. Again, this module is fully obfuscated. Its name is “AMe8” and the entry method is “AMe8!eUI.C74()”.

Figure 4.4 – Invoking the entry method of the “AMe8” module

It breaks in “Invoke()”, as shown in Figure 4.4, and is going to call method “AMe8!eUI.C74()” with parameters, which contains the full path of the copied “AddInProcess32.exe” and a buffer address of the FormBook payload (the same data in %MainFile%, shown in Figure 4.3).

Let’s move on to next section for more information about the module “AMe8”.

Using the “AMe8” Module to Run FormBook

The entry method “AMe8!eUI.C74()” parses the parameters passed from the “zOAI” module and it then runs the “AddInProcess32.exe” that has copied into the %temp% folder. As I explained earlier, “AddInProcess32.exe" is an official program of the Microsoft .Net Framework. Why does “AMe8” module run it? Actually, it is a puppet program, because the “AMe8” module will inject the FormBook payload into it and then perform malicious tasks on its behalf.

To implement this, it creates the process "%Temp%\AddInProcess32.exe". This is a code segment of the “AMe8” module, shown in Figure 5.1, which calls an imported native API CreateProcessAsUser() with a CreationFlag “4U” that refers to **CREATE_SUSPENDED**.

Figure 5.1 – The API CreateProcessAsUser() used to create a suspended process

After calling this API, the “AddInProcess32.exe” process is created and entered into a suspended state before executing any instructions. Next, the “AMe8” module calls a group of imported native APIs, such as GetThreadContext(), ReadProcessMemory(), VirtualAllocEx(), WriteProcessMemory(), SetThreadContext(), and ResumeThread().

It also allocates memory space in the “AddInProcess32.exe” process for the FormBook payload by calling the API VirtualAllocEx() and injecting the FormBook payload into it by calling the API WriteProcessMemory() and modifying the process’s registers using the API SetThreadContext(), so that it runs the FormBook payload’s entry point function when it resumes to run by calling the API ResumeThread(). Figure 5.2 is a screenshot of the Process Tree, which shows the process of “AddInProcess32.exe” just after it is created.

Figure 5.2 - Process tree view of the newly-created “AddInProcess32.exe”

Conclusion on Phishing Campaign

This is the first part of my analysis of the FormBook phishing campaign. In this part, I have elaborated how the PowerPoint document runs its VBA code to download PowerShell code, which extracts the .Net file “item3.jpg”.

In order to execute the main file (payload) of FormBook, it is transferred through three .Net modules, ”Li7f”, “zOAI”, and “AMe8”, and is finally executed by “AMe8”. I also explained the relationship between these modules and how they connect to each other. I also displayed the anti-analysis technique they use – including the full obfuscation, which really takes a significant amount analyst time to defeat.

We know that the payload file of FormBook is injected into a newly-created “AddInProcess32.exe” process and is executed on its behalf to secretly perform malicious tasks. In Part II of this blog, I will provide additional analysis to explain what malicious things it is going to do and how, including what data it can steal from a victim’s device and how it controls the victim via control commands. Please stay tuned.

Fortinet Protections

Fortinet customers are already protected from this FormBook variant with FortiGuard’s Web Filtering and AntiVirus services, as follow:

The download URL launched from the PowerPoint sample is rated as "**Malicious Websites**" by the FortiGuard Web Filtering service.

The attached PowerPoint file is detected as “**VBA/FormBook.C393!tr**” and the “item3.jpg” file is detected as "**MSIL/FormBook.ZXL!tr**" and blocked by the FortiGuard AntiVirus service.

The FortiGuard AntiVirus service is supported by FortiGate, FortiMail, FortiClient, and FortiEDR. The Fortinet AntiVirus engine is a part of each of those solutions as well. As a result, customers who have these products with up-to-date protections are protected.

We also suggest our readers to go through the free NSE training -- NSE 1 – Information Security Awareness, which has a module on Internet threats designed to help end users learn how to identify and protect themselves from phishing attacks.

IOCs:

URLs

hxxp[:]//kiibra[.]com/images/index[.]jpg

Sample SHA-256

[Video Drawing P.O.#4210020253.pps]

82BE061B2BFE48EE3A9F76EE99CF6F3ED712C0C1393AD4A9F064CFC4D11CB53D

[item3.jpg, module name “Li7f”]

F45A363A86D38E5814D41908C0EA5A13F8A89AF2AFEA931472F9905B87FB2ADC

Learn more about FortiGuard Labs threat research and the FortiGuard Security Subscriptions and Services portfolio.

Learn more about Fortinet’s free cybersecurity training initiative or about the Fortinet NSE Training program, Security Academy program, and Veterans program.

Related Posts