


Another country-sponsored #malware: Vietnam APT Campaign

 blog.malwaremustdie.org/2014/08/another-country-sponsored-malware.html

The background

This is a team work analysis, we have at least 5 (five) members involved with this investigation.

The case that is about to be explained here is an APT case. Until now, we were (actually) avoiding APT cases for publicity in Malware Must Die! posts. But due to recent progress in "*public privacy violation or power-abuse/bullying*" malware cases, we improved our policy, so for several cases fit to "*a certain condition*", i.e. malware developed by "*powerful actors with budget*" aiming weak victims including the APT method, or, intimidation for public privacy cases using a crafted-malware, are going to be disclosed and reported here "**ala MMD**", along w/public criminal threat too. So don't use malware if you don't want to look BAD :-)

This case is NOT a new threat, for the background this threat was written in the Infosec Island blog, written by By Eva Galperin and Morgan Marquis-Boire in the good report of article: "Vietnamese Malware Gets Very Personal" which is posted several months ago, access is in here-->[[LINK](#)], the post was very well written as heads up for this threat. Also, there are similar article supported to this threat and worth reading beforehand like:

- <https://www.hostragon.com/shadowy-pro-government-hacking-squad-spying-vietnamese-bloggers/>
- <http://english.vietnamnet.vn/fms/science-it/102484/chinese-hackers-set-malware-to-trap-vietnamese-internet-users.html>

- <http://www.nytimes.com/aponline/2014/01/20/world/asia/ap-as-vietnam-online-wars.html>

You can consider this post is made as additional for the previous writings, to disclose deeper of what public and the victims actually SHOULD know in-depth about the malicious activity detail, that is performed by this malware. To be more preventive in the future for the similar attack that is possibly occurred.

We suspect a group with good budget is in behind of this malware, aiming and bullying privacy of specific individuals who against one country's political method. In a glimpse, the malware, which is trying hard to look like a common-threat, looks like a simple backdoor & connecting/sending some stuffs to CNC. But if you see it closely to the way it works, you will be amazed of the technique used to fulfill its purpose, and SPYING is the right word for that purpose.

The sample we analyzed in this post was received from the victims side, we picked the one file called "Thu moi.7z" which contains the "Thu moi.hta" snippet below:

Name	Date modified	Type	Size
Thu moi.7z	2014/08/17 22:33	WinRAR archive	401 KB
Thu moi.hta	2013/01/26 1:41	HTML Application	1,338 KB

3aefa7a49e75e871959365ce65e60037.#Thu moi.7z
6e667d6c9e527ada1a3284aa333d954d.#Thu moi.hta

..which was reported as the latest of this series.

From the surface, if "Thu moi.hta" file is being executed (double clicked), it will extract (drop) and opening a Microsoft Word DOC file, to camouflage the victim to make them believe that they are opening an archived document file, while what had actually happened is, in the background a series of infection activities happened in the victim's PC.

Malware installer scheme

How the file was extracted from "Thu moi.hta" is by utilizing a simple embedded VB Script, you can see it started in the line 307 (of that .hta sample file) as per shown below in any text editor you pick:

```

306 .....
307 .....
308 .....
309 .....
310 .....
311 .....
312 .....
313 .....
314 .....
315 .....
316 .....
317 .....
318 .....

```

At the starting part of this script. you can see three points was used to camouflage, which are : (1) The usage of the long white space to cover the evil script start tag from the eye-sight, (2) the effort to minimize the "window" for the shell used to run this evil VB Script, and (3) the effort to NOT showing the window taskbar during the script running.

I will try to peel the evil script used, with the explanation I commented within the lines, as per below:

We also check bit-by-bit to make sure which samples belong to which installers, since this malware looks hit some victims / more than one time.

So what does this ".exe" malware do?

Polymorphic self-copy & new process spawner

I picked the .exe file dropped by this .hta installer with the MD5 hash f38d0fb4fiac3571fo7006fb85130aod, this malware was uploaded to VT about 7 months ago.

The malware is the one was dropped by the installer, you can see the same last bits before blobs of "00" hex were written in the malware binary as per snipped and red-marked color in the VB script mentioned in the previous section:

This binary is having an interesting functionality. There's so much to write from it..but I will go to important highlights, or this post is going to be a book. Among all usual malicious tricks for evasion & "reverse/debug checking" tricks used, it was designed to detect the way it was called. When it was initially executed as the form of the dropped .exe from the .hta installer it will delete the original file and rewrite itself to the %Temp% folder using the random Hex-filename with ".tmp" extension, below is the partial writing codes snipped for it:

The self-copied files are polymorphic, below some PoC, one AV evasion detection designed:

2

	Size	Exec Date	Filename	MD5
1				
2				
3	438272	Aud 23 01:28	10.tmp*	577237bfd9c40e7419d27b7b884f95d3
4	438272	Aud 23 07:22	17.tmp*	9451a18db0c70960ace7d714ac0bc2d2
5	438272	Aud 23 07:36	18.tmp*	53d57a45d1b05dce56dd139fc985c55e
6	438272	Aud 23 07:39	19.tmp*	387321416ed21f31ab497a774663b400
7	438272	Aud 23 07:43	1A.tmp*	0a65ecc21f16797594c53b1423749909
8	438272	Aud 23 07:44	1B.tmp*	91a49ed76f52d5b6921f783748edab01
9	438272	Aud 23 07:44	1C.tmp*	f89571efe231f9a05f9288db84dcb006
10	438272	Aud 23 07:45	1D.tmp*	7ca95b52ed43d71e2d6a3bc2543b4ee1
11	438272	Aud 23 07:46	1E.tmp*	faec9c62f091dc2163a38867c28c224d
12	438272	Aud 23 07:47	1F.tmp*	4b02063c848181e3e846b59cbb6b3a46
13	438272	Aud 23 08:14	20.tmp*	5c8f2f581f75beff1316eee0b5eb5f6d
14	438272	Aud 23 01:19	F.tmp*	b466cb01558101d934673f56067f63aa
15	:	:	:	:

It'll then create the process (with the command line API), which will be executed at the function reversed below, I put default IDA commented information since it is important for all of us (not only reverser) to understand flow used below, pls bear the length, just please scroll down to skip these assembly explanation (unless you interest to know how it works):

2

```

1 0x40BF20 sub_40BF20 proc near
2 0x40BF20
3 0x40BF20 StartupInfo= _STARTUPINFOFOW ptr -8508h
4 0x40BF20 ProcessInformation= _PROCESS_INFORMATION ptr -84C4h
5 0x40BF20 var_84B4= dword ptr -84B4h
6 0x40BF20 CommandLine= word ptr -84B0h
7 0x40BF20 FileName= word ptr -4B0h
8 0x40BF20 ApplicationName= dword ptr -2A8h
9 0x40BF20 var_A0= dword ptr -0A0h
10 0x40BF20 var_1C= dword ptr -1Ch
11 0x40BF20 var_18= dword ptr -18h
12 0x40BF20 var_10= dword ptr -10h
13 0x40BF20 var_8= dword ptr -8
14 0x40BF20 var_4= dword ptr -4
15 0x40BF20 arg_8= dword ptr 10h
16 0x40BF20
17 0x40BF20 push ebp
18 0x40BF21 mov ebp, esp
19 0x40BF23 push 0FFFFFFFh
20 0x40BF25 push offset unk_4284D0
21 0x40BF2A push offset sub_416480
22 0x40BF2F mov eax, large fs :0
23 0x40BF35 push eax
24 0x40BF36 sub esp, 8
25 0x40BF39 mov eax, 84F0h
26 0x40BF3E call sub_4207F0
27 0x40BF43 mov eax, dword_42A520

```

```

[0x00069e00:0x00472800]>
6A000 E8 3C EC 3C F0 3C F4 3C F8 3C FC 3C 00 3D 04 3D
6A010 08 3D 0C 3D 10 3D 14 3D 18 3D 1C 3D 20 3D 24 3D
6A020 28 3D 34 3D 38 3D 3C 3D 40 3D 44 3D 48 3D 4C 3D 50 3D
6A030 FC 3D 00 3E 04 3E 08 3E 0C 3E 10 3E 14 3E 18 3E 1C 3E
6A040 60 3F 64 3F 68 3F 6C 3F 70 3F 74 3F 78 3F 7C 3F
6A050 88 3F 8C 3F 90 3F 94 3F 98 3F 9C 3F A0 3F A4 3F
6A060 A8 3F B0 3F B4 3F B8 3F BC 3F C0 3F C4 3F C8 3F
6A070 CC 3F D0 3F D4 3F D8 3F DC 3F E0 3F E4 3F E8 3F
6A080 EC 3F F0 3F F4 3F F8 3F FC 3F 00 00 00 00 00 00
6A090 50 00 00 00 00 00 00 00 04 30 08 30 0C 30 10 30 14 30
6A0A0 18 30 1C 30 20 30 24 30 28 30 2C 30 30 30 34 30
6A0B0 38 30 3C 30 40 30 44 30 48 30 4C 30 50 30 54 30
6A0C0 58 30 5C 30 64 30 68 30 6C 30 70 30 74 30 78 30
6A0D0 7C 30 80 30 84 30 88 30 90 30 94 30 00 A0 06 00
6A0E0 1C 00 00 00 04 30 08 30 B0 30 D0 30 EC 30 08 31
6A0F0 28 31 4C 31 6C 31 00 00 00 00 00 00 00 00 00 00
6A100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
6A110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

```

0x0040AB11 push esi ; contains path of this exe
0x0040AB12 call ds:PathFileExistsW ;
0x0040AB18 test eax, eax+
0x0040AB1A jz short loc_40AB91;
0x0040AB1C push esi ; the file name of this exe+
0x0040AB1D call ds>DeleteFileW ; // self deletion
0x0040AB23 test eax, eax+
;
0x0040AB32 xor edx, edx+
0x0040AB34 push ebx+
0x0040AB35 push eax+
0x0040AB36 mov [ebp+NewFileName], dx+
0x0040AB3D call sub_412510+
0x0040AB42 add esp, 0Ch+
0x0040AB45 lea ecx, [ebp+NewFileName]+
0x0040AB4B push ecx ; lpBuffer+
0x0040AB4C push 104h ; nBufferLength+
0x0040AB51 call ds:GetTempPathW+
0x0040AB57 test eax, eax+
0x0040AB59 jz loc_40AD0A+
0x0040AB5F lea edx, [ebp+NewFileName]+
0x0040AB65 push edx ; lpTempFileName+
0x0040AB66 push ebx ; uUnique+
0x0040AB67 push ebx ; lpPrefixString+
0x0040AB68 mov eax, edx+
0x0040AB6A push eax ; lpPathName+
0x0040AB6B call ds:GetTempFileNameW+
0x0040AB71 test eax, eax+
0x0040AB73 jz loc_40AD0A+
0x0040AB79 push 1 ; dwFlags+
0x0040AB7B lea ecx, [ebp+NewFileName]+
0x0040AB81 push ecx ; lpNewFileName+
0x0040AB82 push esi ; lpExistingFileName+
0x0040AB83 call ds:MoveFileExW+
0x0040AB89 test eax, eax+
0x0040AB8B jz loc_40AD0A+

```

```

28 0x40BF48 xor [ebp+var_8], eax
29 0x40BF4B xor eax, ebp
30 0x40BF4D mov [ebp+var_1C], eax
31 0x40BF50 push ebx
32 0x40BF51 push esi
33 0x40BF52 push edi
34 0x40BF53 push eax
35 0x40BF54 lea eax, [ebp+var_10]
36 0x40BF57 mov large fs:0, eax
37 0x40BF5D mov [ebp+var_18], esp
38 0x40BF60 mov esi, [ebp+arg_8]
39 0x40BF63 xor ebx, ebx
40 0x40BF65 push ebx
41 0x40BF66 call ds:CoInitialize
42 0x40BF6C mov [ebp+var_4], ebx
43 0x40BF6F push 6
44 0x40BF71 push offset aHelp
45 0x40BF76 push esi
46 0x40BF77 call sub_41196F
47 0x40BF7C add esp, 0Ch
48 0x40BF7F test eax, eax
49 0x40BF81 jz loc_40C13E
50 :
51 0x40BF87 call sub_409740
52 0x40BF8C xor eax, eax
53 0x40BF8E mov [ebp+FileName], ax
54 0x40BF95 push 206h
55 0x40BF9A push ebx
56 0x40BF9B lea ecx, [ebp-4AEh]
57 0x40BFA1 push ecx
58 0x40BFA2 call sub_412510
59 0x40BFA7 add esp, 0Ch
60 0x40BFAA push 104h
61 0x40BFAF lea edx, [ebp+FileName]
62 0x40BFB5 push edx
63 0x40BFB6 push ebx
64 0x40BFB7 call ds:GetModuleFileNameW
65 0x40BFBD test eax, eax
66 0x40BFBF jz loc_40C15D
67 :
68 0x40BFC5 xor eax, eax
69 0x40BFC7 mov word ptr [ebp+ApplicationName], ax
70 0x40BFCE push 206h
71 0x40BFD3 push ebx
72 0x40BFD4 lea ecx, [ebp+ApplicationName+2]
73 0x40BFDA push ecx
74 0x40BFDB call sub_412510
75 0x40BFDE add esp, 0Ch
76 0x40BFE3 lea edx, [ebp+ApplicationName]
77 0x40BFE9 push edx
78 0x40BFEA push 104h
79 0x40BFEF call ds:GetTempPathW
80 0x40BFF5 test eax, eax
81 0x40BFF7 jz loc_40C15D
82 :
83 0x40BFFD lea eax, [ebp+ApplicationName]
84 0x40C003 push eax
85 0x40C004 push ebx
86 0x40C005 push ebx
87 0x40C006 mov ecx, eax
88 0x40C008 push ecx
89 0x40C009 call ds:GetTempFileNameW
90 0x40C00F test eax, eax
91 0x40C011 jz loc_40C15D
92 :
93 0x40C017 call sub_4079C0
94 0x40C01C test eax, eax
95 0x40C01E jz loc_40C15D
96 :
97 0x40C024 mov byte ptr [ebp+var_A0], bl
98 0x40C02A push 80h
99 0x40C02F push ebx
100 0x40C030 lea edx, [ebp+var_A0+1]
101 0x40C036 push edx
102 0x40C037 call sub_412510
103 0x40C03C add esp, 0Ch
104 0x40C03F mov [ebp+var_84B4], 81h
105 0x40C049 lea edx, [ebp+var_84B4]
106 0x40C04F lea eax, [ebp+var_A0]
107 0x40C055 call sub_40A300
108 0x40C05A test eax, eax
109 0x40C05C jz loc_40C15D
110 :
111 0x40C07B xor eax, eax

```

```

112 0x40C07D mov [ebp +CommandLine], ax
113 0x40C084 push 7FFEh
114 0x40C089 push ebx
115 0x40C08A lea ecx , [ ebp -84AEh]
116 0x40C090 push ecx
117 0x40C091 call sub_412510
118 0x40C096 lea edx , [ ebp +var_A0]
119 0x40C09C push edx
120 0x40C09D lea eax , [ ebp +FileName]
121 0x40C0A3 push eax
122 0x40C0A4 lea ecx , [ ebp +ApplicationName]
123 0x40C0AA push ecx
124 0x40C0AB push offset aSHelpSS
125 0x40C0B0 push 4000h
126 0x40C0B5 lea edx , [ ebp +CommandLine]
127 0x40C0BB push edx
128 0x40C0BC call sub_411448
129 0x40C0C1 mov [ ebp +StartupInfo.cb], ebx
130 0x40C0C7 push 40h
131 0x40C0C9 push ebx
132 0x40C0CA lea eax , [ ebp +StartupInfo.lpReserved]
133 0x40C0D0 push eax
134 0x40C0D1 call sub_412510
135 0x40C0D6 add esp , 30h
136 0x40C0D9 mov [ ebp +StartupInfo.cb], 44h
137 0x40C0E3 xor ecx , ecx
138 0x40C0E5 mov [ ebp +StartupInfo.wShowWindow], cx
139 0x40C0EC mov [ ebp +StartupInfo.dwFlags], 1
140 0x40C0F6 mov [ ebp +ProcessInformation.hProcess], ebx
141 0x40C0FC xor eax , eax
142 0x40C0FE mov [ ebp +ProcessInformation.hThread], eax
143 0x40C104 mov [ ebp +ProcessInformation.dwProcessId], eax
144 0x40C10A mov [ ebp +ProcessInformation.dwThreadId], eax
145 0x40C110 lea edx , [ ebp +ProcessInformation]
146 0x40C116 push edx
147 0x40C117 lea eax , [ ebp +StartupInfo]
148 0x40C11D push eax
149 0x40C11E push ebx
150 0x40C11F push ebx
151 0x40C120 push 8000000h
152 0x40C125 push ebx
153 0x40C126 push ebx
154 0x40C127 push ebx
155 0x40C128 lea ecx , [ ebp +CommandLine]
156 0x40C12E push ecx
157 0x40C12F lea edx , [ ebp +ApplicationName]
158 0x40C135 push edx
159 0x40C136 call ds :CreateProcessW
160 0x40C13C jmp short loc_40C15D
161
162

```

if the .hta dropped malware named "sample.exe", new process will be started by launching command line contains parameters described below:

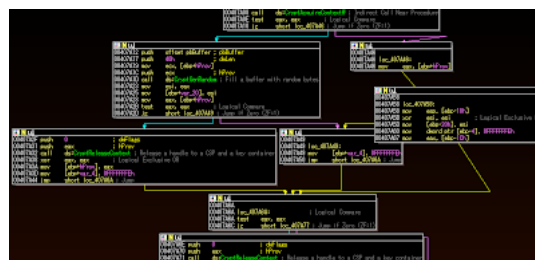
2

```

1 "CreateProcessW" , "C:\DOCUME~1\...\LOCALS~1\Temp\RANDOM[0-9A-F]{1,2}.tmp" , "SUCCESS|FAIL" , "PID: xxx,
2 Command line: "" C : \DOCUME~1\...\LOCALS~1\Temp\RANDOM[0-9A-F]{1,2}.tmp "" \n
3 --helpC:\DOCUME~1\...\LOCALS~1\Temp\sample.exe \n
4 BCE6D32D8CD4F1E6A1064F66D561FDA47E0CD5F8F330C4856A250BB104BC18320FF75E6E56A1741C6770AD238DCFD23DD8A82DDF332FDC8

```

The decryption function used is as per below:



And this malware will end its process here, raising new process that has just been executed..

More drops & payload installation

The process RANDOM[o-9A-F]{1,2}.tmp started by allocated memory, loading rpsess.dll, uxtheme.dll, MSCTF.dll before it self deleting the dropper .exe. The snip code for the deletion is as per below, this isn't also an easy operation, it checks whether the file is really there, if not it makes sure it is there..

2

```
1 0x40A648 push edi
2 0x40A649 call ds :PathFileExistsW
3 :
4 0x40A657 push 0Ah
5 0x40A659 push 65h
6 0x40A65B push ebx
7 0x40A65C call ds :FindResourceW
8 0x40A662 mov esi , eax
9 0x40A664 cmp esi , ebx
10 0x40A666 jz loc_0x40A7CB
11 :
12 0x40A7CB loc_0x40A7CB:
13 0x40A7CB push edi
14 0x40A7CC call ds :DeleteFileW
15 0x40A7D2 mov [ ebp +var_18], 1
16 0x40A779 mov ecx , [ ebp +lpFile]
17 0x40A77C mov edx , [ ebp +lpExistingFileName]
18 0x40A77F push ecx
19 0x40A780 push edx
20 :
21 0x40A78B mov eax , [ ebp +lpFile]
22 0x40A78E push 1
23 0x40A790 push ebx
24 0x40A791 push ebx
25 0x40A792 push eax
26 0x40A793 push ebx
27 0x40A794 push ebx
28 0x40A795 call ds :ShellExecuteW
29 0x40A79B mov [ ebp +var_18], 1
30 :
31
32
33
```

..up to this point I know that we're dealing with a tailored-made malware.

Back to the highlights, RANDOM[o-9A-F]{1,2}.tmp executed with the right condition will drop payloads of this threat, the first drop is the real real payload, following by the second drop as the its driver. The file creation of first payload is handled in function ox41FC90, with the related snip below:

2

```
1 0x41FEAF mov eax , [ ebp +arg_0]
2 0x41FEB2 mov edi , ds :CreateFileW
3 0x41FEB8 push 0
4 0x41FEB8 push [ ebp +dwFlagsAndAttributes]
5 0x41FEBD mov dword ptr [ eax ], 1
6 0x41FEC3 push [ ebp +dwCreationDisposition]
7 0x41FEC6 lea eax , [ ebp +SecurityAttributes]
8 0x41FEC9 push eax
9 0x41FECA push [ ebp +dwShareMode]
10 0x41FECF push [ ebp +dwDesiredAccess]
11 0x41FED0 push [ ebp +lpFileName]
12 0x41FED0
13 0x41FED0
14 0x41FED0
15 0x41FED0
16 0x41FED0
17 0x41FED0
18 0x41FED0
19 0x41FED0
20 0x41FED3 call edi
21 0x41FED5 mov [ ebp +hHandle], eax
22
```

And the writing this file is written in function ox418EC2 after deobfuscating data part, as per snipped here:

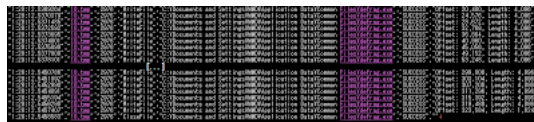
2


```

1  0x418FB9  mov     eax , [ eax +6Ch]
2  0x418FBC  xor     ecx , ecx
3  0x418FBE  cmp     [ eax +14h], ecx
4  0x418FC1  lea    eax , [ ebp +CodePage]
5  0x418FC7  setz   cl
6  0x418FCA  push   eax
7  0x418FCB  mov     eax , [ ebx ]
8  0x418FCD  push   dword ptr [ edi + eax ]
9  0x418FD0  mov     esi , ecx
10 0x418FD2  call   ds :GetConsoleMode
11 : (etc etc)
12 0x4194F0  push   ecx
13 0x4194F1  lea    ecx , [ ebp +var_1AD8]
14 0x4194F7  push   ecx
15 0x4194F8  push   [ ebp +nNumberOfBytesToWrite]
16 0x4194FB  push   [ ebp +lpBuffer]
17 0x419501  push   dword ptr [ eax + edi ]
18 0x419504  call   ds :WriteFile
19 0x41950A  test   eax , eax
20 0x41950C  jz     short loc_0x419523
21 :
22 0x419523  call   ds :GetLastError
23 0x419529  mov     dword ptr [ ebp +WideCharStr],

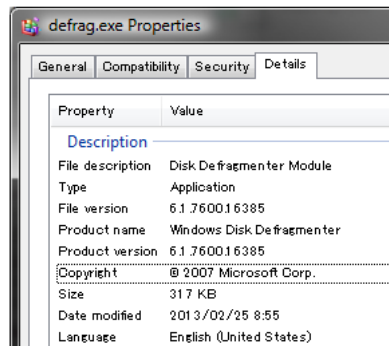
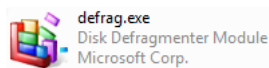
```

we recorded this drop operation in the forensics way too, as per below as evidence:



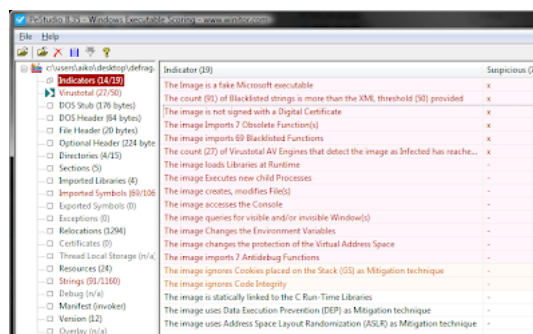
As you can see the wiring method is in redundancy per 4096 bytes.

This first drop called defrag.exe looks pretty much like Windows harddisk defragmentation tool, down to its property, a perfectly crafted evil file:



90F5BBBA8760F964B933C5F0007592D2

Only by using good analysis binary static analysis tool like PESTudio ([maker](#): Marc Oschenmeier), we can spot and focus investigation to the badness indicators right away:



[@MalwareMustDie](#) Thx for using PESTudio for your investigation. In that case, PESTudio indicating that the image is a fake Microsoft EXE! :-)

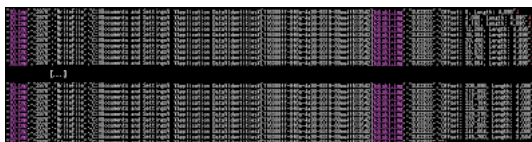
— Marc Ochschenmeier (@ochsenmeier) August 25, 2014

The next drop is the next task of this binary, noted that none of these drops were fetched from internet instead the data is already included in .hta or .[random].exe or [random.tmp].

Using the exactly the same functions described above, 0x41FC90 for creation and 0x418EC2 for writing, the second drop operation were also performed. The file name is formed as per below strings:

```
2
1 "%USERPROFILE%\AppData\Identities\{RANDOM-ID}\disk1.img"
2 like:
3 "C:\Documents and Settings\MMD\Application Data\Identities\{116380ff-9f6a-4a90-9319-89ee4f513542}\disk1.img"
```

the forensics PoC is:



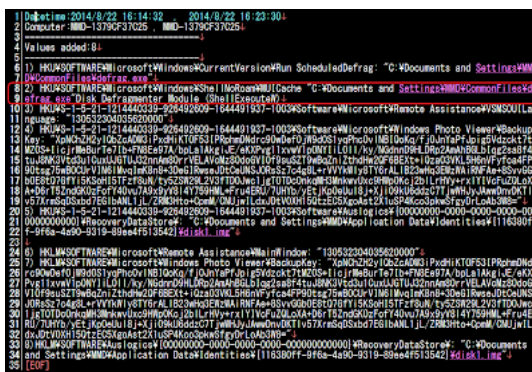
This file is actually a DLL file, here's some peframe:

```
2
1 File Name: disk1.img
2 PE32 executable for MS Windows (DLL) (GUI) Intel 80386 32-bit
3 File Size: 249344 byte
4 Compile Time: 2010-08-14 17:16:08
5 "DLL: True"
6 Entry Point: 0x0001BBD1
7 Sections: 4
8 MD5 hash: 62646ea0a4ce1e6d955cbaef8c4a510d
9 SHA-1 hash: 10116a65e19a7ebc6702250cc1caabf755ce8e7f
10 Anti Debuq: Yes
11 Anti VM: None
```

And Virus Total showing the good infection info:

```
2
1 First submission 2013-03-11 10:38:19 UTC ( 1 year, 5 months ago )
2 Last submission 2014-01-21 12:49:00 UTC ( 7 months ago )
3 File names disk1.dl, disk1.img
```

This file is then performing registry query and writing operations, I will skip some assembly for this, so shortly, these are the 8 keys added, below data I snip from forensics result:



We can see the autostart, and the way it camouflages malicious data in registry using legit scattered softwares and Windows components. Like: Auslogic (RecoveryDataStore), Photo Viewer, Disk Defragment Module, Microsoft Remote Assistance. This all means to hide and prevent the quick notice of this malware in the infected PC, it is a well thought plan.

To be noted that one of the key is used to run the defrag.exe execution via ShellExecuteW by the [Random].tmp file, and also you can see the "key" used for this malware saved, one last thing to be noticed is the bot ID used.

PS: There are also more drops made which are the Windows task installer for this malware

```
2
1 C:\Windows\Tasks\ScheduledDefrag.job
2 C:\Windows\Tasks\ScheduledDefrag_admin.job
```

It is the Windows scheduler (kinda crond) to execute the EXE payload (defrag.exe). Pic:

```

[0x00000000]> x
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00000000 2006 0120 6331 b678 9643 4c4d 887e 766d .. c1.x.CL.M.™.vm
0x00000010 27c7 1913 4620 3a01 2020 2020 3c20 0a20 ...F!<.
0x00000020 2020 2020 ffff ffff 2020 2020 0313 0420 ~ ....
0x00000030 2022 2001 2020 2020 2020 2020 2020 2020
0x00000040 2020 2020 2020 3520 4320 3a20 5c20 5520
0x00000050 7320 6520 7220 7320 5c20 4d20 4d20 4420
0x00000060 5c20 4120 7020 7020 4420 6120 7420 6120
0x00000070 5c20 5220 6f20 6120 6d20 6920 6e20 6720
0x00000080 5c20 4320 6f20 6d20 6d20 6f20 6e20 2020
0x00000090 4620 6920 6c20 6520 7320 5c20 6420 6520
0x000000a0 6620 7220 6120 6720 2e20 6520 7820 6520
0x000000b0 2020 2020 2020 0420 4d20 4d20 4420 2020
0x000000c0 3620 5420 6820 6920 7320 2020 7420 6120
0x000000d0 7320 6b20 2020 6420 6520 6620 7220 6120
0x000000e0 6720 6d20 6520 6e20 7420 7320 2020 7420
0x000000f0 6820 6520 2020 6320 6f20 6d20 7020 7520
0x00001000 7420 6520 7220 7320 2020 6820 6120 7220
0x00001010 6420 2020 6420 6920 7320 6b20 2020 6420
0x00001020 7220 6920 7620 6520 7320 2e20
[0x00000000]>

```

```

5 C:\U
sers\MMD
\AppData
\Roaming
\Commo
Files\de
frag.exe
.MMD
6 This ta
sk defra
gments t
he compu
ters har
d disk d
rives.

```

What this payload does

First thing that caught interest and attention is these obfuscation constant variables saved in .rdata section:

```

2
1 0x40F3AC
2 0x40F3AC aTztzpx75Xtdsia:
3 0x40F3AC unicode 0, < "tztzpx75]xtdsjqu/fyf" >,0
4 0x40F3D6 align 4
5 0x40F3D8
6 0x40F3D8 aTztufn43Xtdsia:
7 0x40F3D8 unicode 0, < "tztufn43]xtdsjqu/fyf" >,0
8 0x40F402 align 4
9 0x40F404
10 0x40F404 a2e6a3ddEmm:
11 0x40F404 unicode 0, < "2e6g3dd/emm" >,0
12 0x40F430
13 0x40F430 a0spsbnGimftY9:
14 0x40F430 unicode 0, < "Qspsbn!Gjmf!)y97*]Joufsofu!Fyqmpsf]jfmvxujm/fyf" >,0
15 0x40F498
16 0x40F498 a0spsbnGimftNf:
17 0x40F498 unicode 0, < "Qspsbn!Gjmf]Nftfohs]ntntht/fyf" >,0
18 0x40F4DE align 10h
19 0x40F4E0
20 0x40F4E0 a0spsbnGimft 0:
21 0x40F4E0 unicode 0, < "Qspsbn!Gjmf!)y97*]Joufsofu!Fyqmpsf]jfyqmpsf/fyf" >,0
22 0x40F548
23 0x40F548 a0spsbnGimftJo:
24 0x40F548 unicode 0, < "Qspsbn!Gjmf]Joufsofu!Fyqmpsf]jfyqmpsf/fyf" >,0
25 0x40F5A2 align 4
26

```

We have good decoder team in MMD. Soon these data were translated as per below:

```

405 // Strings Obfuscation:-
406 Qspsbn!Gjmf!)y97*]Joufsofu!Fyqmpsf]jfmvxujm/fyf Program Files (x86)\Internet Explorer\iexplore.exe
407 Qspsbn!Gjmf]Nftfohs]ntntht/fyf Program Files\Messenger\msg.exe
408 Qspsbn!Gjmf]Joufsofu!Fyqmpsf]jfyqmpsf/fyf Program Files\Internet Explorer\iexplore.exe
409 tztzpx75]xtdsjqu/fyf system32\cmd.exe
410 Qspsbn!Gjmf!)y97*]Joufsofu!Fyqmpsf]jfyqmpsf/fyf Program Files (x86)\Internet Explorer\iexplore.exe
411 tztzpx75]xtdsjqu/fyf system32\cmd.exe
412

```

When these data formed in the functions where they were called, we will have better idea of WHY these strings were obfuscated. This time we will take a look at the dump analysis in disassembly, to seek the executed code parts only:

2

```

1 0x0C22D37 call 0x0C28720h target: 0x0C28720
2 0x0C22D3C add esp , 0Ch
3 0x0C22D3F push 0x0C2F404h <== UTF-16 "2e6q3dd/emm"
4 0x0C22D44 lea edx , dword ptr [ ebp -00000084h]
5 0x0C22D4A push edx
6 0x0C22D4B call dword ptr [0x0C2D06Ch] lstrcpyW@KERNEL32.DLL
7 0x0C2207C lea ecx , dword ptr [ ebp -00000082h]
8 0xC22082 push ecx
9 0xC22083 mov word ptr [ ebp -00000080h], ax
10 0xC2208A call 00C28720h target: 00C28720
11 0xC2208F add esp , 0Ch
12 0xC22092 push 00C2F278h <== UTF-16
13 "Tpguxbsf]Bvtmphjdt]|11111111.1111.1111.1111.111111111111~]SfdpwwszEbubTupsf"
14 0x0C22A4E call ebx PathFileExistsW@SHLWAPI.DLL ( Import , 1 Params)
15 0x0C22A50 test eax , eax
16 0x0C22A52 jne 0x0C22AB8h target: 0x0C22AB8
17 0x0C22A54 push 0x0C2F4E0h <== UTF-16 "Qsphsbn!Gjmf!)y97*]Joufsofu!Fyqmpsf]jfyqmpsf/fyf"
18 0x0C22625 xor eax , eax
19 0x0C22627 push 0000007Eh
20 0x0C22629 push eax
21 0x0C2262A lea ecx , dword ptr [ ebp -0x000086h]
22 0x0C22630 push ecx
23 0x0C22631 mov word ptr [ ebp -0x000088h], ax
24 0x0C22638 call 0x0C28720h target: 0x0C28720
25 0x0C2263D mov esi , dword ptr [0x0C2D06Ch] lstrcpyW@KERNEL32.DLL
26 0x0C22643 add esp , 0Ch
27 0x0C22646 push 0x0C2F360h <== UTF-16 "///]tlzqf/fyf"
28 0x0C2264B lea edx , dword ptr [ ebp -0x000088h]
29 0x0C22651 push edx
30 0x0C22652 call esi lstrcpyW@KERNEL32.DLL
31 0x0C229DB push edx
32 0x0C229DC call ebx PathFileExistsW@SHLWAPI.DLL
33 0x0C229DE test eax , eax
34 0x0C229E0 jne 0x0C22A46h target: 0x0C22A46
35 0x0C229E2 push 0x0C2F498h <== UTF-16 "Qsphsbn!Gjmf]Nfttfohs]ntntht/fyf"
36 0x0C229E7 lea eax , dword ptr [ esp +74h]
37 0x0C229EB push eax
38 0x0C229EC call esi lstrcpyW@KERNEL32.DLL
39 0x0C22876 call dword ptr [0x0C2D090h] GetVersion@KERNEL32.DLL ( Import , 0 Params)
40 0x0C2287C mov esi , dword ptr [0x0C2D06Ch] lstrcpyW@KERNEL32.DLL ( Import , 2 Params)
41 0x0C22882 push 0x0C2F3ACh <== UTF-16 "tztzpx75]xtdsjqu/fyf" ; DECODED: "syswow64\wscript.exe"
42 0x0C22887 lea eax , dword ptr [ esp +74h]
43 0x0C2288B push eax
44 0x0C2288C call esi lstrcpyW@KERNEL32.DLL ( Import , 2 Params)
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63

```

Found this function is interesting, I found the check for username "Administrator" and SUID "system" are checked:

2

```

1 0x0C21FAB xor bl , bl
2 0x0C21FAD call dword ptr [0xC2D00Ch] GetUserName@ADVAPI32.DLL ( Import , 2 Params)
3 0x0C21FB3 test eax , eax
4 0x0C21FB5 je 0x0C21FCEh target: 0xC21FCE
5 0x0C21FB7 push 0x0C2F22Ch <== UTF-16 "system"
6 0x0C21FBC lea ecx , dword ptr [ ebp -0x000204h]
7 0x0C21FC2 push ecx
8 0x0C21AC9 call dword ptr [0x0C2D014h] LookupAccountSid@ADVAPI32.DLL
9 0x0C21ACF test eax , eax
10 0x0C21AD1 je 0x0C21AFDh target: 0xC21AFD
11 0x0C21AD3 lea ecx , dword ptr [ ebp -0x000204h]
12 0x0C21AD9 push ecx
13 0x0C21ADA push 0x0C2F1FCh <== UTF-16 "administrators"
14 0x0C21ADF call dword ptr [0x0C2D030h] lstrcpw@KERNEL32.DLL
15 0x0C21AE5 test eax , eax
16
17
18
19
20

```

Suspicious isn't it?

I go back to the binary for understanding the related functions, which is in 0x4027F0. I was wondering of what is the part of **wscript.exe** (not again!?) mentioned by this binary. So I trailed the path of the **wscript.exe** starting here, assumed that the Windows architecture is x64:

2

```

1 0x40286E call sub_408720
2 0x402873 add esp , 0Ch
3 0x402876 call ds :GetVersion
4 0x402876
5 0x40287C mov esi , ds :lstrcpyW
6 0x402882 push offset aTztxpx75Xtdsjq <== Push : "tztzpx75]xtdsjqu/fyf" to stack
7 0x402882
8 0x402887 lea eax , [ esp +694h+pMore]
9 0x40288B push eax
10 0x40288C call esi
11 0x40288E mov dx , [ esp +690h+pMore]
12 0x402893 xor edi , edi
13 0x402895 xor ecx , ecx
14 0x402897 movzx eax , dx
15 0x40289A cmp di , dx

```

then found the binary wscript.exe is executed in this part:

2

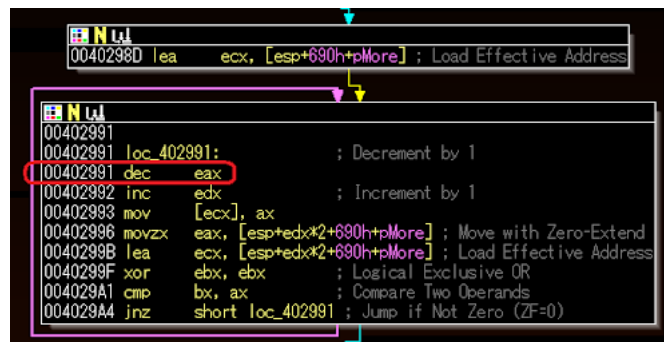
```

1  0x402B54 xor    eax ,  eax
2  0x402B56 push   40h
3  0x402B58 push   eax
4  0x402B59 mov    [ esp +698h+ProcessInformation.hThread], eax
5  0x402B5D mov    [ esp +698h+ProcessInformation.dwProcessId], eax
6  0x402B61 mov    [ esp +698h+ProcessInformation.dwThreadId], eax
7  0x402B65 lea   eax , [ esp +698h+StartupInfo.lpReserved]
8  0x402B69 push   eax
9  0x402B6A mov    [ esp +69Ch+ProcessInformation.hProcess], 0
10 0x402B72 call  sub_408720
11 0x402B77 add    esp , 0Ch
12 0x402B7A xor    ecx ,  ecx
13 0x402B7C lea   edx , [ esp +690h+ProcessInformation]
14 0x402B80 push   edx
15 0x402B80
16 0x402B81 lea   eax . [ esp +694h+StartupInfo]
17 0x402B81
18 0x402B85 push   eax
19 0x402B86 push   offset Buffer
20 0x402B8B push   ecx
21 0x402B8B
22 0x402B8C push   ecx
23 0x402B8D push   ecx
24 0x402B8E push   ecx
25 0x402B8F push   ecx
26 0x402B90 mov    [ esp +6B0h+StartupInfo.wShowWindow], cx
27 0x402B95 lea   ecx , [ esp +6B0h+CommandLine]
28 0x402B9C push   ecx
29 0x402B9D lea   edx , [ esp +6B4h+ApplicationName]
30 0x402BA4 push   edx
31 0x402BA5 mov    [ esp +6B8h+StartupInfo.cb], 44h
32 0x402BAD mov    [ esp +6B8h+StartupInfo.dwFlags], 1
33 0x402BB5 call  ds :CreateProcessW
34 0x402BBB test   eax ,  eax

```

So we have the wscript.exe process up and running.

Up to this part our teammate poke me in DM, and he asked me what can he helped, so I asked our friend (Mr. Raashid Bhat) to take over the further analysis of this defrag.exe and disk1.img, while I went to other parts, and after a while he came up straight forward with (1) decoder logic, which is match to our crack team did:



And (2) the conclusion of what "defrag.exe" is actually doing, is a loader which patches the executed wscript.exe's ExitProcess to load the DLL "disk1.img"....Well, it's all starts to make more sense now.

Checking the reported data. I confirmed to find the "process was read" from here:

2

```

1  0x4014BB mov    edx , [ ebp +nSize]
2  0x4014C1 lea   ecx , [ ebp +NumberOfBytesRead]
3  0x4014C7 push   ecx
4  0x4014C8 mov    ecx , [ ebp +lpAddress]
5  0x4014CE push   edx
6  0x4014CF lea   eax , [ ebp +Buffer]
7  0x4014D2 push   eax
8  0x4014D3 push   ecx
9  0x4014D4 push   esi
10 0x4014D5 mov    [ ebp +NumberOfBytesRead], ebx
11 0x4014DB call  ds :ReadProcessMemory
12 0x4014E1 test   eax ,  eax
13
14

```

As for the "Exit Process patching" itself, it is a quite sophisticate technique was used. It used a tiny shellcode that was observed within Mem Loc 1 : 009C0000 to 009D0000 (by Raashid).

The shellcode then was saved in binary which I received and then I was reversing it deeper, it looks like as per following snips:

```
[0x00000000]> x
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00000000 9090 6890 009c 00e8 c7ac e37b bb04 009c ..h.....{...
0x00000010 0089 03e8 1903 f47b bb08 009c 0089 03bb .....{.....
0x00000020 0090 9c90 c603 0068 e803 0000 e811 24e3 .....h.....$.
0x00000030 7beb f490 ffff ffff ffff ffff {.....
```

This shellcode I tweaked a bit, is in a plain assembly, contains three addresses of Windows static API call to (I wrote these API in order of calls from top to bottom) **LoadLibraryW@kernel32.dll**, **RtlGetLastWin32Error@ntdll.dll**, **Sleep@kernel32.dll** which can be shown in assembly code of the code as per snips below:

```
[0x00000000]> pd
0x00000000 90          nop
0x00000001 90          nop
0x00000002 680009c00  push 0x9c0000 ; 0x009c0000
0x00000007 e8c7ace37b call 0x7be3acd3
0x7be3acd3(unk)
0x0000000c bb04009c00 mov ebx, 0x9c0004
0x00000011 8903       mov [ebx], eax
0x00000013 e81903f47b call 0x7bf40331
0x7bf40331()
0x00000018 bb08009c00 mov ebx, 0x9c0008
0x0000001d 8903       mov [ebx], eax
0x0000001f bb0009c00  mov ebx, 0x9c0000
0x00000024 c60300    mov byte [ebx], 0x0
0x00000027 68e8030000 push 0x3e8 ; 0x000003e8
0x0000002c e81124e37b call 0x7be32442
0x7be32442(unk)
=< 0x00000031 ebf4       jmp 0x100000027
0x00000033 90          nop
0x00000034 ff         invalid
0x00000035 ff         invalid
0x00000036 ff         invalid
0x00000037 ff         invalid
```

So now we know that defrag.exe is actually hacked wscript.exe, hooks ExitProcess Function of kernel32.dll and patches it with a LoadLibraryW@kernel32.dll and loads a DLL string in local (for further execution), does some error-trapping and gives time for the DLL to be processed (loaded and executed).

OK. So now we have the idea on how this binary sniffs for account, checks for processes and load and use the DLL (disk1.img). There are many more details for more operation in defrag.exe, like searching the process of Auslogic and that skype/messenger buff (also many registry values sniffed too) , but those will be added later after this main course..

The DLL Payload

This DLL is the goal of this infection. It has operations for networking functionality, contains the CNC information and the data to be sent to the CNC. If you do forensics, you may never see disk1.img or the deobfuscated DLL filename in the process, but you will see its operation by the patched wscript.exe (for it was hacked to load this DLL, the wscript.exe process should appear).

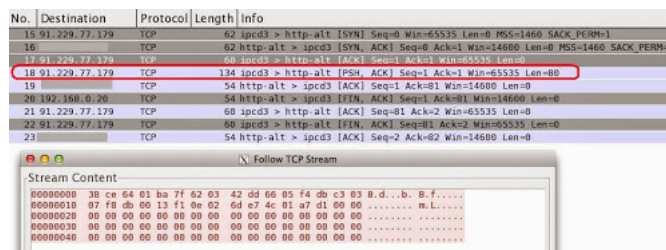
Below is the DLL part that in charge for the socket connections...

2

When I received the result, since I had the report that the CNC was down at the time reversed, I used the local dummy DNS to seek whether the requests was made to those CNC hosts, and is proven:

DNS	77	Standard query	0x4b92	A	menmin.strezf.com
DNS	74	Standard query	0x5202	A	imaps.qki6.com
DNS	74	Standard query	0x8aec	A	static.jg7.org
DNS	74	Standard query	0x8aec	A	static.jg7.org
DNS	77	Standard query	0xa620	A	menmin.strezf.com
DNS	74	Standard query	0xb3c3	A	imaps.qki6.com
DNS	74	Standard query	0xb975	A	static.jg7.org
DNS	74	Standard query	0xb975	A	static.jg7.org
DNS	74	Standard query	0xb975	A	static.jg7.org
DNS	74	Standard query	0xb975	A	static.jg7.org
DNS	74	Standard query	0xe67b	A	imaps.qki6.com
DNS	74	Standard query	0xf1da	A	static.jg7.org
DNS	74	Standard query	0xf1da	A	static.jg7.org

Furthermore, using the different method of networking (I won't explain this for the security purpose), I could find the alive connection to the CNC's IP and PoC'ing the blob binary sent to initiate the connection. Noted, again the data matched, the reversing blob binary is actually the CNC sent data used to initiate the CNC communication, as per captured in the PCAP below, same bits:



Does it means the CNC still alive?

I am not so sure. It was connected. The CNC "allowed" the bot to send the data to them, yet it was not responding back afterward and led the communication becoming in "pending" stage. So, there is many possibility can be happened, like: CNC is gone, or CNC specs has changed, etc. After all this APT sample is about 6-7months old.

So please allow me to take a rain check for analysis the blob binary used (still on it..among tons of tasks..). Let's investigate this CNC related network.

The CNC investigation

Based on the reverse engineering, forensics & behavior analysis we did, we found the CNC is actually 3 (three) hostnames matched to the 6 (six) IP addresses as per listed below:

2

- 1 static.ia7.ora
- 2 imaps.qki6.com
- 3 menmin.strezf.com

Which historically are using the below IP addresses:

2

- 1 8.5.1.38
- 2 64.74.223.38
- 3 208.73.211.66
- 4 91.229.77.179
- 5 124.217.252.186
- 6 212.7.198.211

The first three domains is having a very bad reputation in phishing & malware infection globally. PoC-->[\[here\]](#)

For the location of these IP are shown in the below details:

P Address	Country Code	Location	Postal Code	Coordinates	ISP	Organization	Domain	Metro Code
8.5.1.38	US	Costa Mesa, California, United States, North America		33.6411, -117.9187	Level 3 Communications	eNom, Incorporated		803
64.74.223.38	US	Atlanta, Georgia, United States, North America	30303	33.7518, -84.3915	Internap Network Services Corporation	eNom, Incorporated		524
91.229.77.179	UA	Ukraine, Europe		49, 32	FOP Zemlyanyi Dmitro Leonidowich	FOP Zemlyanyi Dmitro Leonidowich	dehabod.com.ua	
124.217.252.186	MY	Malaysia, Asia		2.5, 112.5	Piradius Net	Piradius Net		
208.73.211.66	US	Los Angeles, California, United States, North America	90071	34.0533, -118.2549	Oversee.net	Oversee.net		803
212.7.198.211	NL	Netherlands, Europe		52.5, 5.75	Dedisev Dedicated Servers Sp. z o.o.	LeaseWeb B.V.		

And the period time for each CNC's used subdomains VS IP addresses above can be viewed clearly below (Thank you FairSight team):

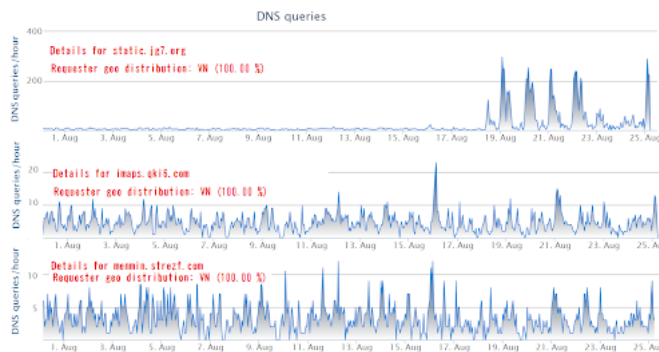
2

```

1 first seen 2013-11-01 21:17:45 -0000
2 last seen 2013-11-04 05:22:20 -0000
3 static.jg7.org. A 8.5.1.41
4
5 first seen 2013-10-07 13:10:00 -0000
6 last seen 2013-11-18 14:38:32 -0000
7 static.jg7.org. A 64.74.223.41
8
9 first seen 2013-08-26 10:01:39 -0000
10 last seen 2013-10-07 12:34:21 -0000
11 static.jg7.org. A 91.229.77.179
12
13 first seen 2012-12-17 04:20:19 -0000
14 last seen 2013-06-20 05:53:03 -0000
15 static.jg7.org. A 124.217.252.186
16
17 first seen 2013-06-20 08:00:28 -0000
18 last seen 2013-08-26 09:00:42 -0000
19 static.id7.org. A 212.7.198.211
20 first seen 2013-11-01 21:22:55 -0000
21 last seen 2013-11-04 05:24:20 -0000
22 imaps.qki6.com. A 8.5.1.38
23
24 first seen 2013-10-07 13:10:18 -0000
25 last seen 2013-11-18 14:38:38 -0000
26 imaps.qki6.com. A 64.74.223.38
27
28 first seen 2013-08-26 10:02:05 -0000
29 last seen 2013-10-07 12:33:13 -0000
30 imaps.qki6.com. A 91.229.77.179
31
32 first seen 2012-12-17 04:19:46 -0000
33 last seen 2013-06-20 05:52:30 -0000
34 imaps.qki6.com. A 124.217.252.186
35
36 first seen 2014-01-06 01:21:07 -0000
37 last seen 2014-01-11 14:30:44 -0000
38 imaps.qki6.com. A 208.73.211.66
39
40 first seen 2013-06-20 07:07:43 -0000
41 last seen 2013-08-26 09:01:08 -0000
42 imaps.qki6.com. A 212.7.198.211
43 first seen 2013-08-26 10:02:31 -0000
44 last seen 2014-08-22 04:06:36 -0000
45 menmin.strezf.com. A 91.229.77.179
46
47 first seen 2013-10-05 11:54:26 -0000
48 last seen 2013-10-07 13:45:55 -0000
49 menmin.strezf.com. A 208.91.197.101
50
51 first seen 2013-06-20 06:26:33 -0000
52 last seen 2013-08-26 09:01:34 -0000
53 menmin.strezf.com. A 212.7.198.211
54
55

```

And below is the DNS queries for these hostname (not IP) recorded in the recent terms, thank's to OpenDNS:



Cross checking various similar samples with the all recorded domains & IPs for the related CNC we found more possibility related hostnames to the similar series of the threat, suggesting the same actor(s), noted the usage of DDNS domains:

2

- 1 foursquare.dyndns.tv
- 2 neuro.dyndns-at-home.com
- 3 tripadvisor.dyndns.info
- 4 wowwiki.dynalias.net
- 5 yelp.webhop.org
- 6 (there are some more but we are not 100% sure of them yet..is a TBA now..)

The bully actor(s) who spread this APT loves to hide their domain behind various of services like:

2

- 1 nsX.dreamhost.com
- 2 nsX.cloudns.net
- 3 nsXX.ixwebhosting.com
- 4 nsXX.domaincontrol.com
- 5 dnsX.name-services.com
- 6 nsXX.dsredirection.com
- 7 dnsX.parkpage.foundationapi.com

With noted that these THREE CNC domains used by this sample, are made on this purpose only, and leaving many traceable evidence in the internet that we collected all of those successfully. Trailing every info leaves by this domains: **ig7.org**, **qki6.com**, **strezf.com** will help you to know who is actually behind this attack. Noted: see the time frame data we disclosed above. If there any malware initiators and coders think they can bully others and hide their ass in internet is a BIG FAIL.

The data is too many to write it all here, by the same method of previous check we can find the relation between results. It is an interesting investigation.

Samples

What we analyzed is shared only in KernelMode, link-->[\[here\]](#)

With thankfully to KM team (rocks!) I am reserving a topic there for the continuation disclosure for same nature of sample and threat.

The epilogue

This series of APT attack looks come and go, it was reported back then from 2009. This one campaign looks over, but for some reason that we snipped in above writing, there is no way one can be sure whether these networks used are dead. The threat is worth to investigate and monitor deeper. Some posts are suspecting political background supporting a government mission of a certain group is behind this activities, by surveillance to the targeting victims. Avoiding speculation, what we saw is a spyware effort, with a good quality...a hand-made level, suggesting a custom made malware, and I bet is not a cheap work too. We talked and compare results within involved members and having same thought about this.

If you received the sample, or, maybe got infected by these series, I suggest to please take a look at the way it was spread, dropped techniques used binaries, and the many camouflage tricks used. Further, for the researchers involved, we should add that the way to hide the CNC within crook's network is the PoC for a very well-thought & clever tricks. We have enough idea for whom is capable to do this, and now is under investigation.

We are informing to all MMD friends, this investigation is OPEN, please help in gathering information that is related to this threat for the future time frame too, as much as possible. We are opposing whoever group that is backing up this evil operation, and believe me, the dots are started to connect each other..

We are going to handle the similar threat from now on, so IF you have the abuse case by malware and need the deep investigation of what that malware does, do not hesitate to send us sample, archive the samples and text contains the explanations of how you got the sample and how can we contact you, with the password "infected", and please upload it in this link-->[\[DropBin\]](#).

Don't use malware, we never believe that any usage of malware can achieve any goodness. We will battle the malware initiators and its coders for the sake to support a better humanity and better internet usage.

