Unpacking the spyware disguised as antivirus

August 25, 2016 by Malwarebytes Labs

Last updated: January 7, 2020

Recently we got access to several elements of the espionage toolkit that has been captured attacking Vietnamese institutions. During the operation, the malware was used to dox 400,000 members of Vietnam Airlines.

The payload, distributed disguised as antivirus, is a variant of Korplug RAT (aka PlugX) – a spyware with former associations with Chinese APT groups, and known from targeted attacks at important institutions of various countries.

In this article we will describe the process of extracting the final payload out of it's cover.

Analyzed samples

Set #1:

- 884d46c01c762ad6ddd2759fd921bf71 – McAfee.exe (harmless: reference)

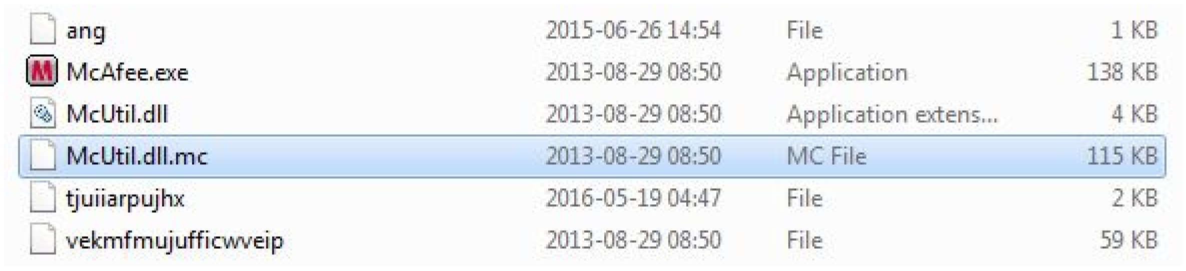
- c52464e9df8b3d08fc612a0f11fe53b2 – McUtil.dll (shellcode loader)
- 28f151ae7f673c0cf369150e0d44e415 – McUtil.dll.mc – shellcode
    - 321a2f0abe47977d5c8663bd7a7c7d28 – unpacked payload (DLL)

Execution flow:

McAfee.exe -> McUtil.dll -> McUtil.dll.mc -> payload (DLL)

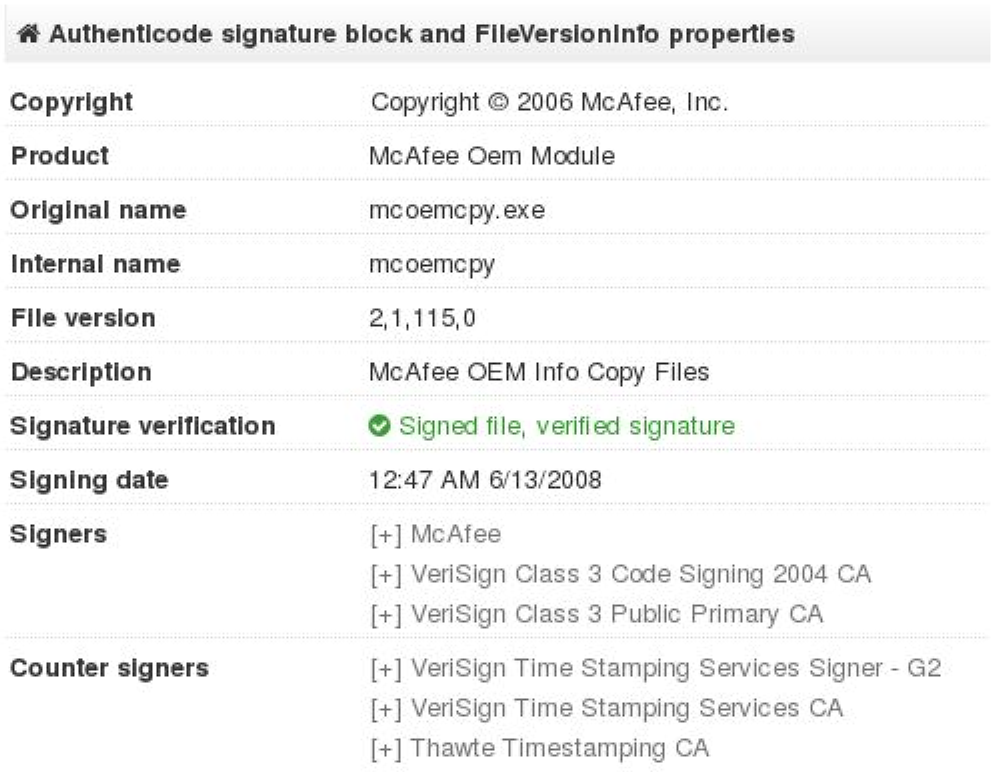
A look at the package

This spyware has an interesting, modular package. As a whole, it tries to pretend to be McAfee antivirus:



ang	2015-06-26 14:54	File	1 KB
McAfee.exe	2013-08-29 08:50	Application	138 KB
McUtil.dll	2013-08-29 08:50	Application extens...	4 KB
McUtil.dll.mc	2013-08-29 08:50	MC File	115 KB
tjuiiarpujhx	2016-05-19 04:47	File	2 KB
vekmfmujufficwveip	2013-08-29 08:50	File	59 KB

If we take a look at the executable, we see that is has been signed by the original certificate:


Authenticode signature block and FileVersionInfo properties

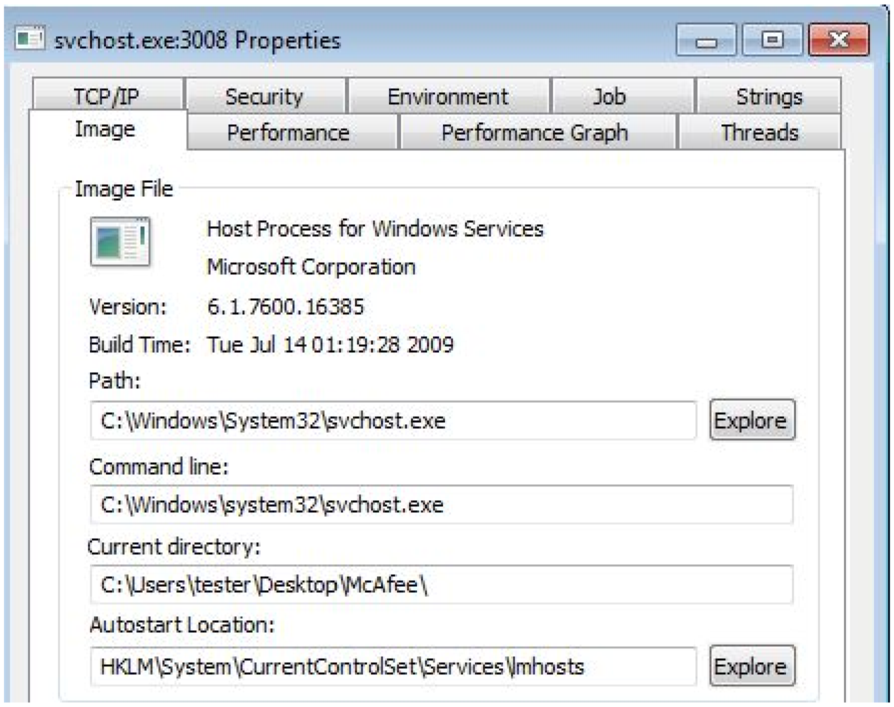
Copyright	Copyright © 2006 McAfee, Inc.
Product	McAfee Oem Module
Original name	mcoemcpy.exe
Internal name	mcoemcpy
File version	2,1,115,0
Description	McAfee OEM Info Copy Files
Signature verification	✓ Signed file, verified signature
Signing date	12:47 AM 6/13/2008
Signers	[+] McAfee [+] VeriSign Class 3 Code Signing 2004 CA [+] VeriSign Class 3 Public Primary CA
Counter signers	[+] VeriSign Time Stamping Services Signer - G2 [+] VeriSign Time Stamping Services CA [+] Thawte Timestamping CA

It is not fake – the executable is a legitimate product. However, it is bundled with the DLL that is not signed – and this it the point that attackers used in order to hijack the execution.

Note that the app used in the attacks is very old (compiled in 2008). The current versions of McAfee Antivirus that we managed to test are no longer vulnerable to this type of abuse.

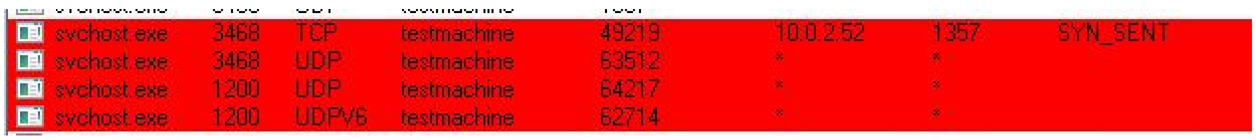
Behavioral analysis

After being deployed, the application runs silently. We can see the main component executing svchost.exe, and then terminating itself. It is caused by the fact that the malicious code has been injected into svchost, and will continue operating from there. Looking at the current directory of svchost.exe we can find that it inherits default directory of the malicious app:

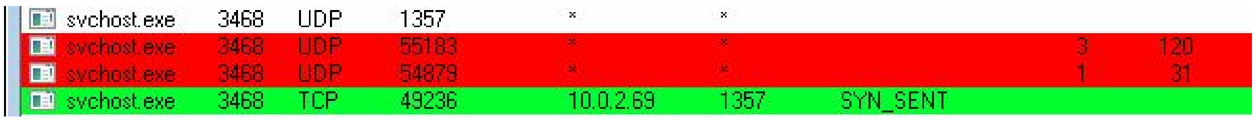

svchost.exe:3008 Properties

Host Process for Windows Services
Microsoft Corporation
Version: 6.1.7600.16385
Build Time: Tue Jul 14 01:19:28 2009
Path:
C:\Windows\System32\svchost.exe
Command line:
C:\Windows\system32\svchost.exe
Current directory:
C:\Users\tester\Desktop\McAfee\
Autostart Location:
HKLM\System\CurrentControlSet\Services\lmhosts

The bot makes reconnaissance in the LAN by scanning for other computers. It enumerates full range of local addresses, from the lowest to the highest:

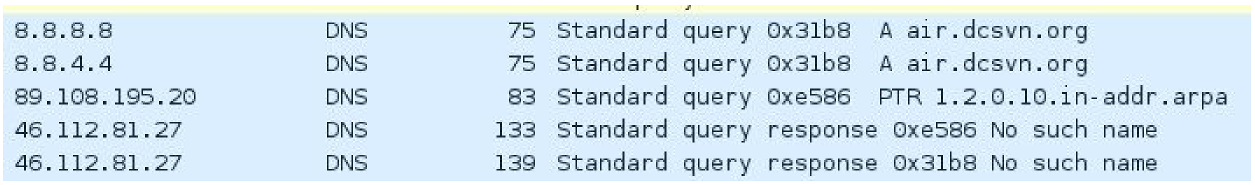


svchost.exe	3468	TCP	testmachine	49219	10.0.2.52	1357	SYN_SENT
svchost.exe	3468	UDP	testmachine	63512	*	*	
svchost.exe	1200	UDP	testmachine	64217	*	*	
svchost.exe	1200	UDPV6	testmachine	62714	*	*	



svchost.exe	3468	UDP	1357	*	*			
svchost.exe	3468	UDP	55183	*	*		3	120
svchost.exe	3468	UDP	54879	*	*		1	31
svchost.exe	3468	TCP	49236	10.0.2.69	1357	SYN_SENT		

It also tried to connect with it’s C&C (air.dscvn.org), however, at the moment of tests the domain was down:



```
8.8.8.8          DNS    75  Standard query 0x31b8  A air.dcsvn.org
8.8.4.4          DNS    75  Standard query 0x31b8  A air.dcsvn.org
89.108.195.20    DNS    83  Standard query 0xe586  PTR 1.2.0.10.in-addr.arpa
46.112.81.27     DNS   133  Standard query response 0xe586 No such name
46.112.81.27     DNS   139  Standard query response 0x31b8 No such name
```

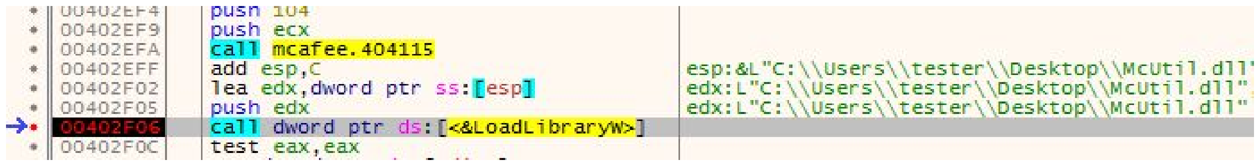
Unpacking

The application have several layers of loaders before it reach the final functionality. The exe file, as well as the DLL are harmless. All the the malicious features lies in the external file, that is a blocks of obfuscated shellcode. Within the shellcode, another DLL is hidden – that is the core spy bot.

Loading the shellcode

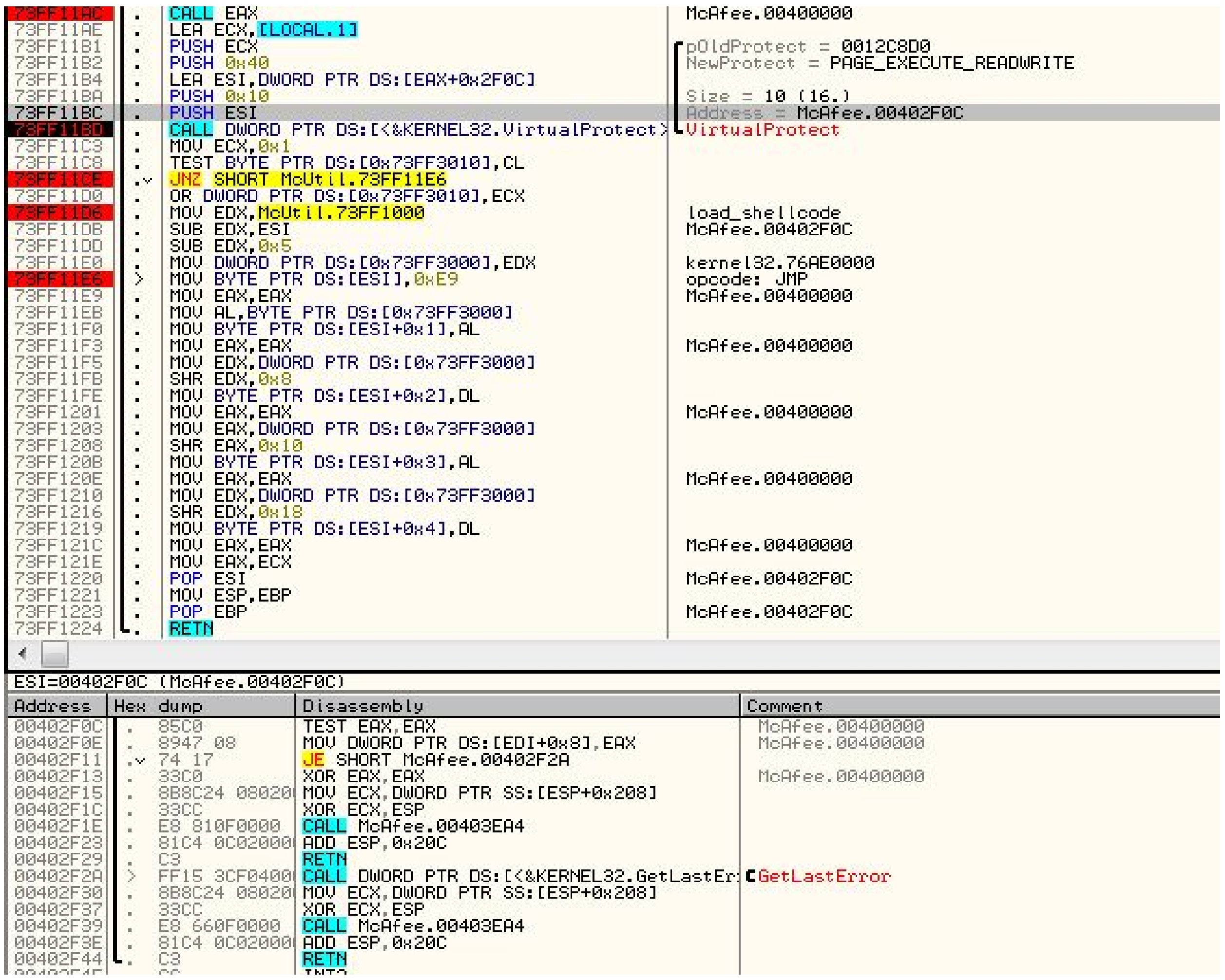
The payload is loaded in an obfuscated way containing some interesting tricks. The authors took great care that it will not be easy to analyze the modules separately.

Execution starts from the harmless *McAfee.exe*. Malware utilized the fact that this application loads a library called *McUtil.dll* from the startup directory. It doesn't make any integrity check, so in fact, if we rename any library to the desired name, the executable will just load it:



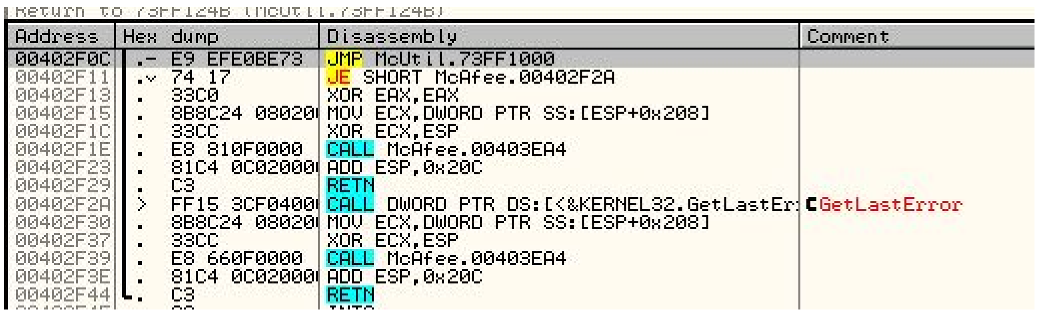
```
00402EF4   push 104
00402EF9   push ecx
00402EFA   call mcafee.404115
00402EFF   add esp,C                     esp:&L"C:\\Users\\tester\\Desktop\\McUtil.dll"
00402F02   lea edx,dword ptr ss:[esp]    edx:L"C:\\Users\\tester\\Desktop\\McUtil.dll"
00402F05   push edx                      edx:L"C:\\Users\\tester\\Desktop\\McUtil.dll"
00402F06   call dword ptr ds:[<&LoadLibraryW>]
00402F0C   test eax,eax
```

McUtil.dll is supposed to deploy the next file: *McUtil.dll.mc* – however, to make the flow more difficult to follow, it doesn't run it directly. Instead, it patches the caller executable (*McAfee.exe*) and makes it execute the function responsible for reading and loading the next file. Below we can see the fragment of code, that writes the hook into the memory:



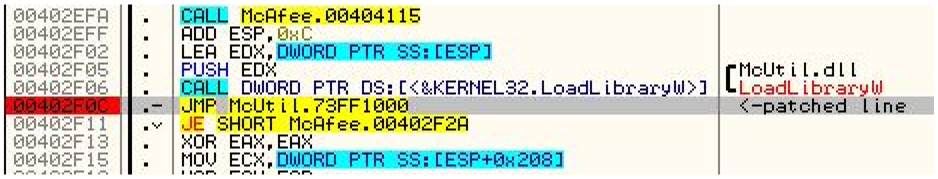
```
73FF11AC  .  CALL EAX                                       McAfee.00400000
73FF11AE  .  LEA ECX,[LOCAL.1]
73FF11B1  .  PUSH ECX                                       pOldProtect = 0012C8D0
73FF11B2  .  PUSH 0x40                                      NewProtect = PAGE_EXECUTE_READWRITE
73FF11B4  .  LEA ESI,DWORD PTR DS:[EAX+0x2F0C]
73FF11BA  .  PUSH 0x10                                      Size = 10 (16.)
73FF11BC  .  PUSH ESI                                       Address = McAfee.00402F0C
73FF11BD  .  CALL DWORD PTR DS:[<&KERNEL32.VirtualProtect>  VirtualProtect
73FF11C3  .  MOV ECX,0x1
73FF11C8  .  TEST BYTE PTR DS:[0x73FF3010],CL
73FF11CE  .v JNZ SHORT McUtil.73FF11E6
73FF11D0  .  OR DWORD PTR DS:[0x73FF3010],ECX
73FF11D6  .  MOV EDX,McUtil.73FF1000                        load_shellcode
73FF11DB  .  SUB EDX,ESI                                    McAfee.00402F0C
73FF11DD  .  SUB EDX,0x5
73FF11E0  .  MOV DWORD PTR DS:[0x73FF3000],EDX              kernel32.76AE0000
73FF11E6  >  MOV BYTE PTR DS:[ESI],0xE9                     opcode: JMP
73FF11E9  .  MOV EAX,EAX                                    McAfee.00400000
73FF11EB  .  MOV AL,BYTE PTR DS:[0x73FF3000]
73FF11F0  .  MOV BYTE PTR DS:[ESI+0x1],AL
73FF11F3  .  MOV EAX,EAX                                    McAfee.00400000
73FF11F5  .  MOV EDX,DWORD PTR DS:[0x73FF3000]
73FF11FB  .  SHR EDX,0x8
73FF11FE  .  MOV BYTE PTR DS:[ESI+0x2],DL
73FF1201  .  MOV EAX,EAX                                    McAfee.00400000
73FF1203  .  MOV EAX,DWORD PTR DS:[0x73FF3000]
73FF1208  .  SHR EAX,0x10
73FF120B  .  MOV BYTE PTR DS:[ESI+0x3],AL
73FF120E  .  MOV EAX,EAX                                    McAfee.00400000
73FF1210  .  MOV EDX,DWORD PTR DS:[0x73FF3000]
73FF1216  .  SHR EDX,0x18
73FF1219  .  MOV BYTE PTR DS:[ESI+0x4],DL
73FF121C  .  MOV EAX,EAX                                    McAfee.00400000
73FF121E  .  MOV EAX,ECX
73FF1220  .  POP ESI                                        McAfee.00402F0C
73FF1221  .  MOV ESP,EBP
73FF1223  .  POP EBP                                        McAfee.00402F0C
73FF1224  .  RETN

ESI=00402F0C (McAfee.00402F0C)
Address   Hex dump         Disassembly                               Comment
00402F0C  .  85C0          TEST EAX,EAX                              McAfee.00400000
00402F0E  .  8947 08       MOV DWORD PTR DS:[EDI+0x8],EAX            McAfee.00400000
00402F11  .v 74 17         JE SHORT McAfee.00402F2A
00402F13  .  33C0          XOR EAX,EAX                               McAfee.00400000
00402F15  .  8B8C24 08020  MOV ECX,DWORD PTR SS:[ESP+0x208]
00402F1C  .  33CC          XOR ECX,ESP
00402F1E  .  E8 810F0000   CALL McAfee.00403EA4
00402F23  .  81C4 0C02000  ADD ESP,0x20C
00402F29  .  C3            RETN
00402F2A  >  FF15 3CF0400  CALL DWORD PTR DS:[<&KERNEL32.GetLastEr  GetLastError
00402F30  .  8B8C24 08020  MOV ECX,DWORD PTR SS:[ESP+0x208]
00402F37  .  33CC          XOR ECX,ESP
00402F39  .  E8 660F0000   CALL McAfee.00403EA4
00402F3E  .  81C4 0C02000  ADD ESP,0x20C
00402F44  .  C3            RETN
```

That’s how the above fragment of caller’s code looks after patching. Instead of the first two lines we can see a jump into the *McUtil.dll*:



```
Return to 73FF124B (McUtil.73FF124B)
Address   Hex dump         Disassembly                               Comment
00402F0C  .- E9 EFE0BE73   JMP McUtil.73FF1000
00402F11  .v 74 17         JE SHORT McAfee.00402F2A
00402F13  .  33C0          XOR EAX,EAX
00402F15  .  8B8C24 08020  MOV ECX,DWORD PTR SS:[ESP+0x208]
00402F1C  .  33CC          XOR ECX,ESP
00402F1E  .  E8 810F0000   CALL McAfee.00403EA4
00402F23  .  81C4 0C02000  ADD ESP,0x20C
00402F29  .  C3            RETN
00402F2A  >  FF15 3CF0400  CALL DWORD PTR DS:[<&KERNEL32.GetLastEr  GetLastError
00402F30  .  8B8C24 08020  MOV ECX,DWORD PTR SS:[ESP+0x208]
00402F37  .  33CC          XOR ECX,ESP
00402F39  .  E8 660F0000   CALL McAfee.00403EA4
00402F3E  .  81C4 0C02000  ADD ESP,0x20C
00402F44  .  C3            RETN
```

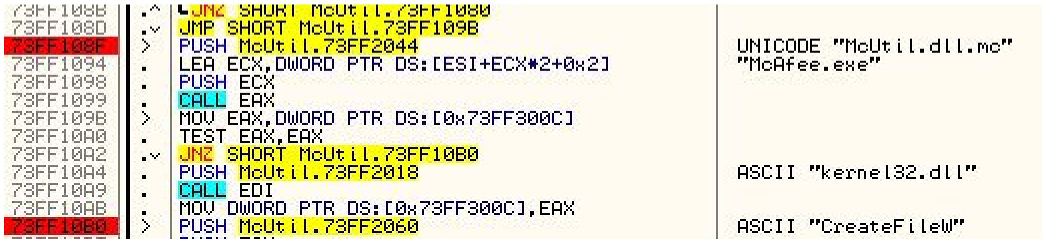
Patching function is in DllMain of the *McUtil.dll* – so, it is called on load. The patched line is just after the call that loaded the library:



```
00402EFA  .  CALL McAfee.00404115
00402EFF  .  ADD ESP,0xC
00402F02  .  LEA EDX,DWORD PTR SS:[ESP]
00402F05  .  PUSH EDX                                        ┌McUtil.dll
00402F06  .  CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryW>]    └LoadLibraryW
00402F0C  .- JMP McUtil.73FF1000                              <-patched line
00402F11  .v JE SHORT McAfee.00402F2A
00402F13  .  XOR EAX,EAX
00402F15  .  MOV ECX,DWORD PTR SS:[ESP+0x208]
```

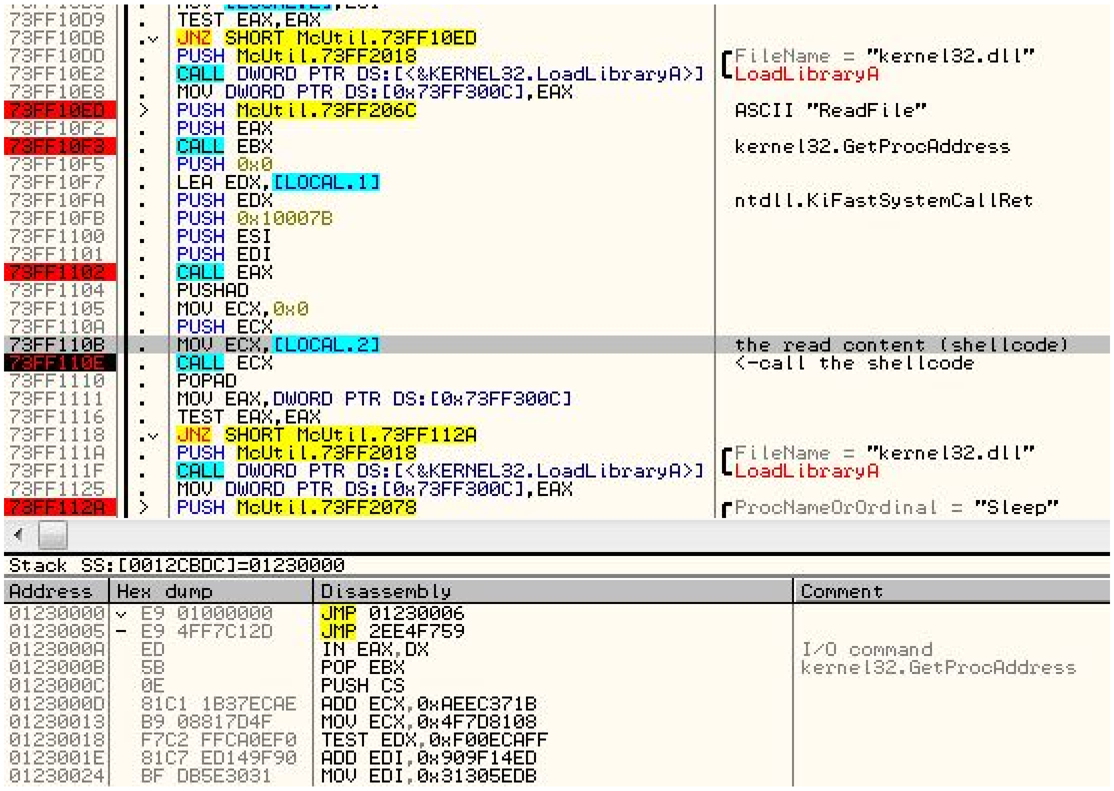
So, the hook will be executed as soon as the loading function returns.

Inside the function called by the hook, the external file is open:



```
73FF108B  .^ JNZ SHORT McUtil.73FF1080
73FF108D  .v JMP SHORT McUtil.73FF109B
73FF108F  >  PUSH McUtil.73FF2044                  UNICODE "McUtil.dll.mc"
73FF1094  .  LEA ECX,DWORD PTR DS:[ESI+ECX*2+0x2]  "McAfee.exe"
73FF1098  .  PUSH ECX
73FF1099  .  CALL EAX
73FF109B  >  MOV EAX,DWORD PTR DS:[0x73FF300C]
73FF10A0  .  TEST EAX,EAX
73FF10A2  .v JNZ SHORT McUtil.73FF10B0
73FF10A4  .  PUSH McUtil.73FF2018                  ASCII "kernel32.dll"
73FF10A9  .  CALL EDI
73FF10AB  .  MOV DWORD PTR DS:[0x73FF300C],EAX
73FF10B0  >  PUSH McUtil.73FF2060                  ASCII "CreateFileW"
```

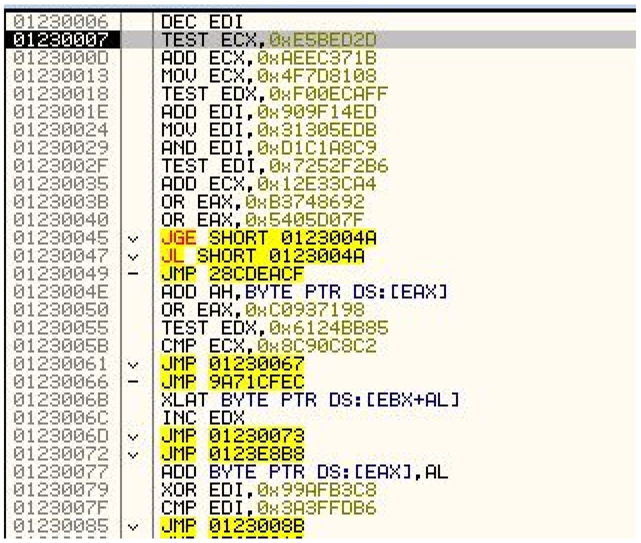
It is read into the memory and then execution is redirected there:



```
73FF10D9  .  TEST EAX,EAX
73FF10DB  .v JNZ SHORT McUtil.73FF10ED
73FF10DD  .  PUSH McUtil.73FF2018                           ┌FileName = "kernel32.dll"
73FF10E2  .  CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryA>]   └LoadLibraryA
73FF10E8  .  MOV DWORD PTR DS:[0x73FF300C],EAX
73FF10ED  >  PUSH McUtil.73FF206C                            ASCII "ReadFile"
73FF10F2  .  PUSH EAX
73FF10F3  .  CALL EBX                                        kernel32.GetProcAddress
73FF10F5  .  PUSH 0x0
73FF10F7  .  LEA EDX,[LOCAL.1]
73FF10FA  .  PUSH EDX                                        ntdll.KiFastSystemCallRet
73FF10FB  .  PUSH 0x10007B
73FF1100  .  PUSH ESI
73FF1101  .  PUSH EDI
73FF1102  .  CALL EAX
73FF1104  .  PUSHAD
73FF1105  .  MOV ECX,0x0
73FF110A  .  PUSH ECX
73FF110B  .  MOV ECX,[LOCAL.2]                               the read content (shellcode)
73FF110E  .  CALL ECX                                        <-call the shellcode
73FF1110  .  POPAD
73FF1111  .  MOV EAX,DWORD PTR DS:[0x73FF300C]
73FF1116  .  TEST EAX,EAX
73FF1118  .v JNZ SHORT McUtil.73FF112A
73FF111A  .  PUSH McUtil.73FF2018                           ┌FileName = "kernel32.dll"
73FF111F  .  CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryA>]   └LoadLibraryA
73FF1125  .  MOV DWORD PTR DS:[0x73FF300C],EAX
73FF112A  >  PUSH McUtil.73FF2078                           ┌ProcNameOrOrdinal = "Sleep"

Stack SS:[0012CBDC]=01230000

Address   Hex dump        Disassembly              Comment
01230000  v E9 01000000   JMP 01230006
01230005  - E9 4FF7C12D   JMP 2EE4F759
0123000A    ED            IN EAX,DX                I/O command
0123000B    5B            POP EBX                  kernel32.GetProcAddress
0123000C    0E            PUSH CS
0123000D    81C1 1B37ECAE ADD ECX,0xAEEC371B
01230013    B9 08817D4F   MOV ECX,0x4F7D8108
01230018    F7C2 FFCA0EF0 TEST EDX,0xF00ECAFF
0123001E    81C7 ED149F90 ADD EDI,0x909F14ED
01230024    BF DB5E3031   MOV EDI,0x31305EDB
```

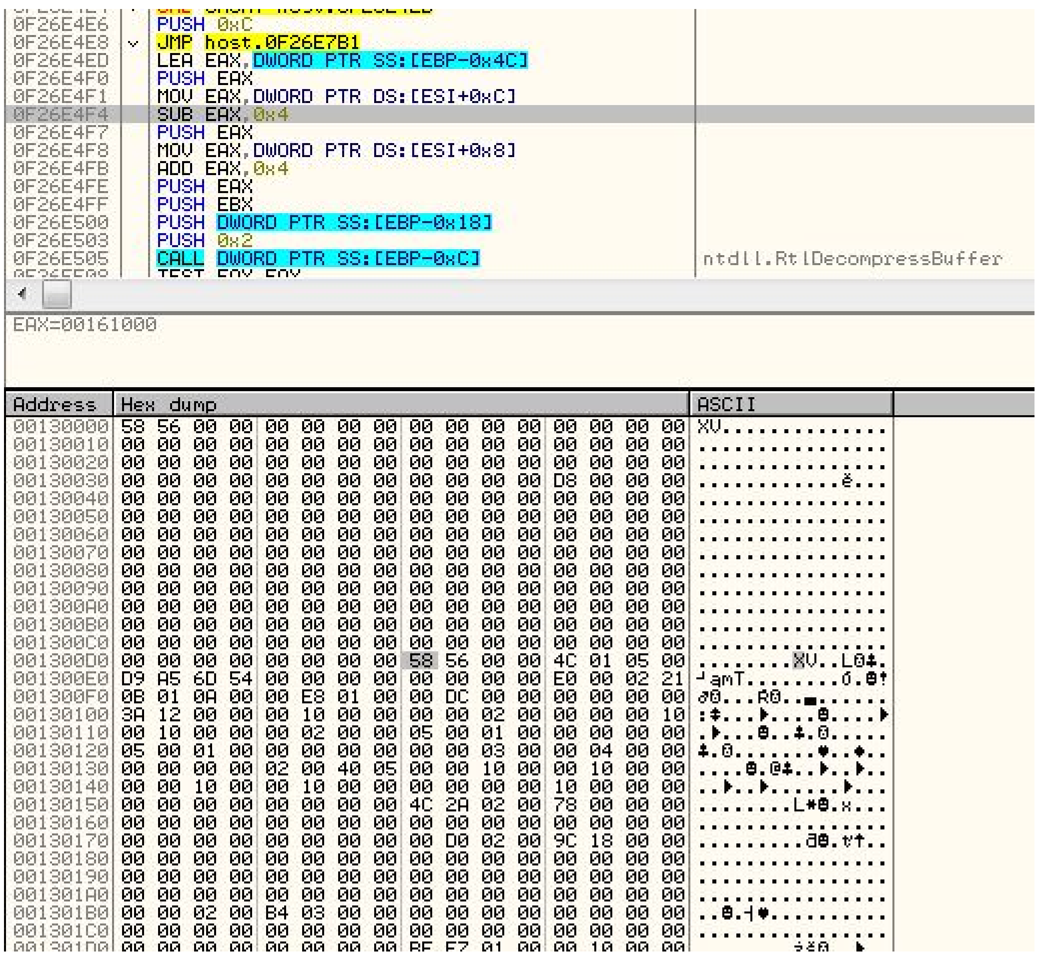
Unpacking the final payload

The shellcode is heavily obfuscated:



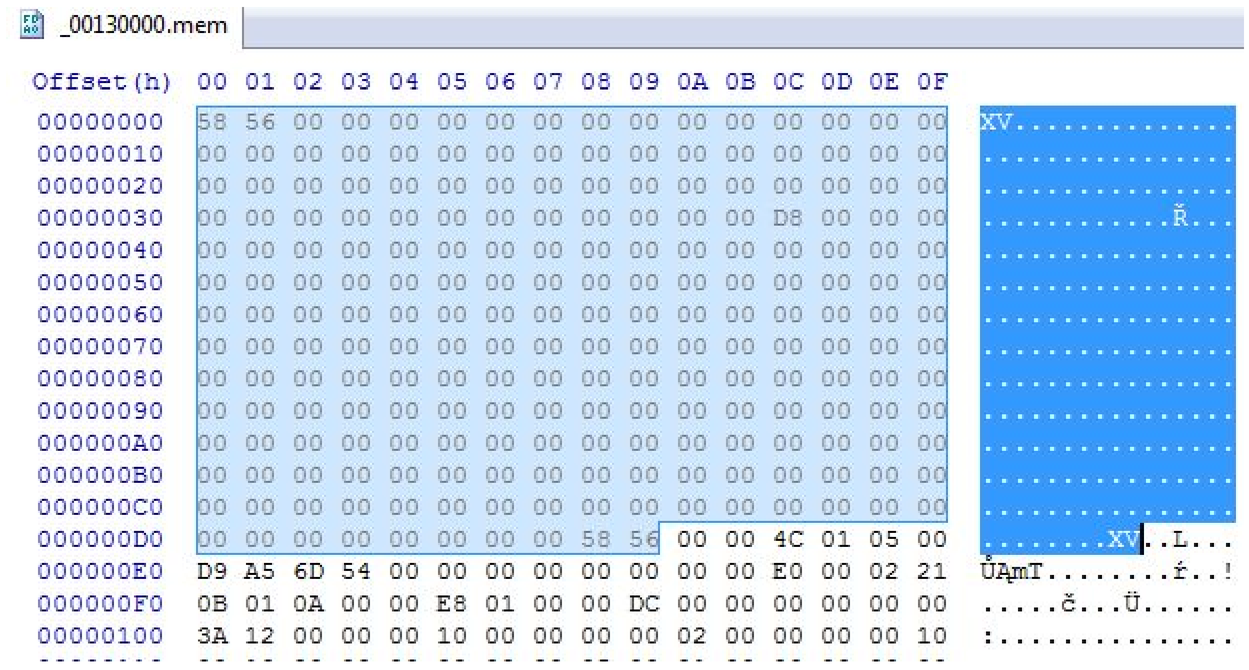
```
01230006     DEC EDI
01230007     TEST ECX,0xE5BED2D
0123000D     ADD ECX,0xAEEC371B
01230013     MOV ECX,0x4F7D8108
01230018     TEST EDX,0xF00ECAFF
0123001E     ADD EDI,0x909F14ED
01230024     MOV EDI,0x31305EDB
01230029     AND EDI,0xD1C1A8C9
0123002F     TEST EDI,0x7252F2B6
01230035     ADD ECX,0x12E33CA4
0123003B     OR EAX,0xB3748692
01230040     OR EAX,0x5405D07F
01230045  v  JGE SHORT 0123004A
01230047  v  JL SHORT 0123004A
01230049  -  JMP 28CDEACF
0123004E     ADD AH,BYTE PTR DS:[EAX]
01230050     OR EAX,0xC0937198
01230055     TEST EDX,0x6124BB85
0123005B     CMP ECX,0x8C90C8C2
01230061  v  JMP 01230067
01230066  -  JMP 9A71CFEC
0123006B     XLAT BYTE PTR DS:[EBX+AL]
0123006C     INC EDX
0123006D  v  JMP 01230073
01230072  v  JMP 0123E8B8
01230077     ADD BYTE PTR DS:[EAX],AL
01230079     XOR EDI,0x99AFB3C8
0123007F     CMP EDI,0x3A3FFDB6
01230085  v  JMP 0123008B
```

This is not the main stage, but an unpacker and loader of the main spyware. It decompresses the following content into a buffer:



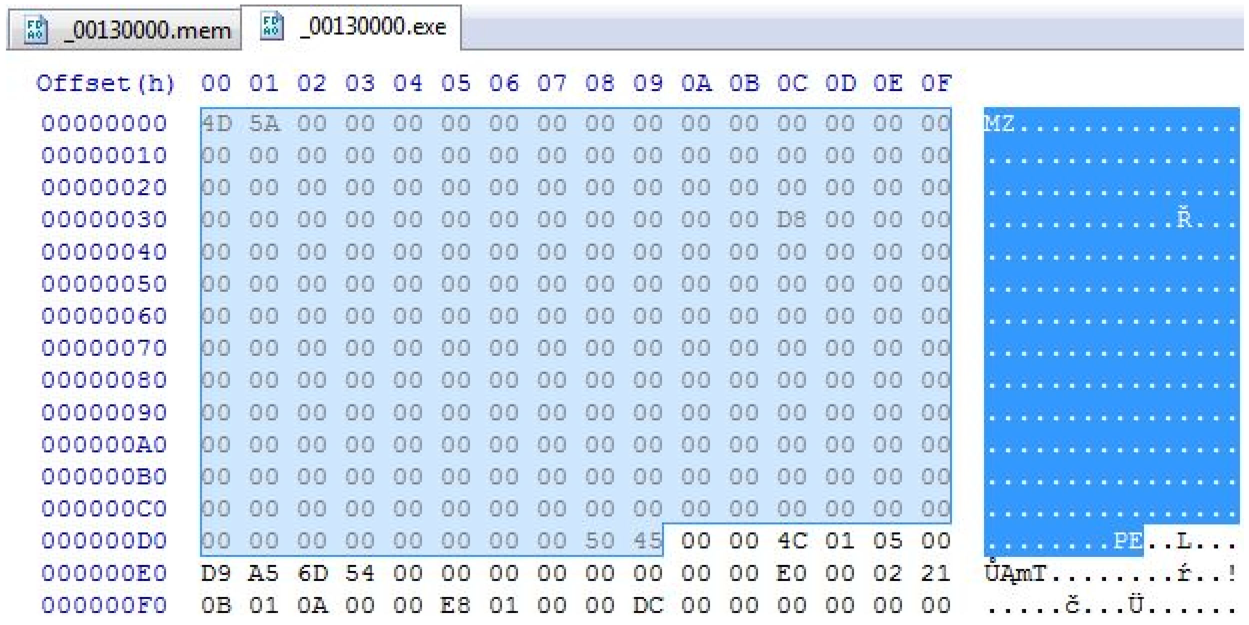
```
0F26E4E6     PUSH 0xC
0F26E4E8  v  JMP host.0F26E7B1
0F26E4ED     LEA EAX,DWORD PTR SS:[EBP-0x4C]
0F26E4F0     PUSH EAX
0F26E4F1     MOV EAX,DWORD PTR DS:[ESI+0xC]
0F26E4F4     SUB EAX,0x4
0F26E4F7     PUSH EAX
0F26E4F8     MOV EAX,DWORD PTR DS:[ESI+0x8]
0F26E4FB     ADD EAX,0x4
0F26E4FE     PUSH EAX
0F26E4FF     PUSH EBX
0F26E500     PUSH DWORD PTR SS:[EBP-0x18]
0F26E503     PUSH 0x2
0F26E505     CALL DWORD PTR SS:[EBP-0xC]                    ntdll.RtlDecompressBuffer

EAX=00161000

Address   Hex dump                                          ASCII
00130000  58 56 00 00 00 00 00 00 00 00 00 00 00 00 00 00  XV..............
00130010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130030  00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00  ............ě...
00130040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001300A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001300B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001300C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001300D0  00 00 00 00 00 00 00 00 58 56 00 00 4C 01 05 00  ........XV..L0‡.
001300E0  D9 A5 6D 54 00 00 00 00 00 00 00 00 E0 00 02 21  ┘ąmT........Ó.☻!
001300F0  0B 01 0A 00 00 E8 01 00 00 DC 00 00 00 00 00 00  ♂0...R0..▬......
00130100  3A 12 00 00 00 10 00 00 00 00 02 00 00 00 00 10  :‡...►....☻....►
00130110  00 10 00 00 00 02 00 00 05 00 01 00 00 00 00 00  .►...☻..‡.0.....
00130120  05 00 01 00 00 00 00 00 00 00 03 00 00 04 00 00  ‡.0.......♥..♦..
00130130  00 00 00 00 02 00 40 05 00 00 10 00 00 10 00 00  ....☻.@‡..►..►..
00130140  00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00  ..►..►......►...
00130150  00 00 00 00 00 00 00 00 4C 2A 02 00 78 00 00 00  ........L*☻.x...
00130160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130170  00 00 00 00 00 00 00 00 00 D0 02 00 9C 18 00 00  .........Đ☻.ť↑..
00130180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00130190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001301A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001301B0  00 00 02 00 B4 03 00 00 00 00 00 00 00 00 00 00  ..☻.┤♥..........
001301C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Then it reserves additional memory and starts remapping this content, chunk by chunk. By the way in which it parses it, we can notice similarity with process of remapping raw PE file into a virtual image. And indeed, the unpacked content is a PE file – only the headers are distorted. Delimiters XV were used to substitute the typical “MZ”.. “PE” values:



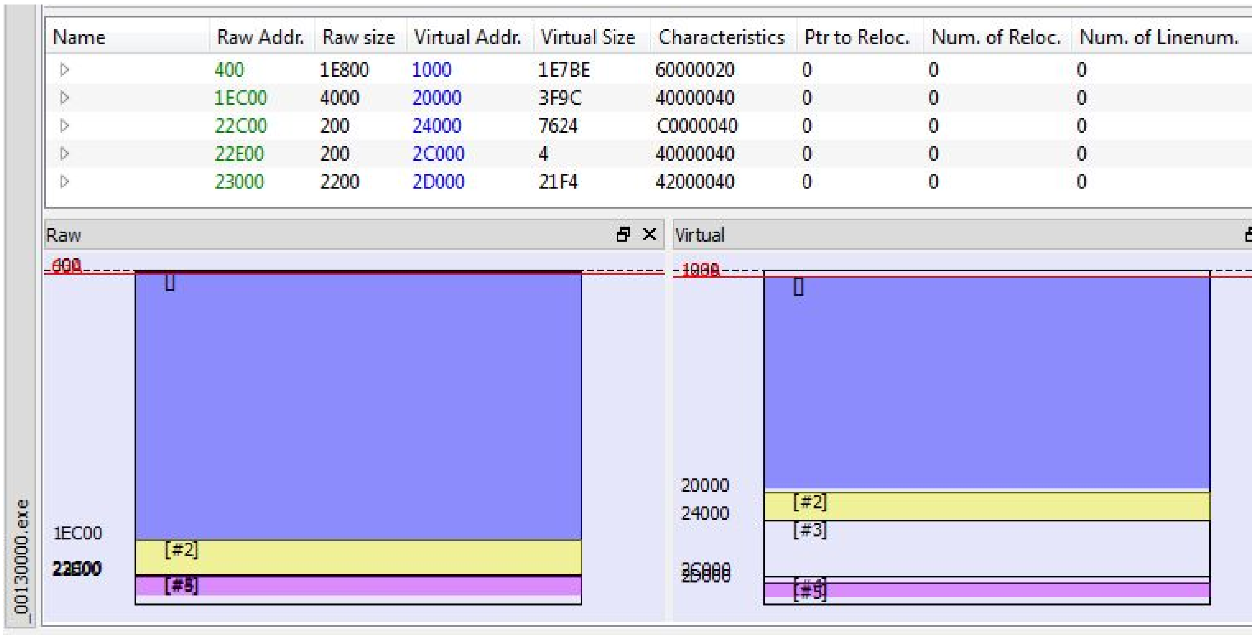
```
_00130000.mem

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  58 56 00 00 00 00 00 00 00 00 00 00 00 00 00 00  XV..............
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00  ............Ř...
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 58 56 00 00 4C 01 05 00  ........XV..L...
000000E0  D9 A5 6D 54 00 00 00 00 00 00 00 00 E0 00 02 21  ŮĄmT........ŕ..!
000000F0  0B 01 0A 00 00 E8 01 00 00 DC 00 00 00 00 00 00  .....č...Ü......
00000100  3A 12 00 00 00 10 00 00 00 00 02 00 00 00 00 10  :...............
```

Reconstructing the header is not difficult – we must just substitute back those values by their real meaning:



```
_00130000.mem   _00130000.exe

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4D 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00  MZ..............
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00  ............Ř...
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 05 00  ........PE..L...
000000E0  D9 A5 6D 54 00 00 00 00 00 00 00 00 E0 00 02 21  ŮĄmT........ŕ..!
000000F0  0B 01 0A 00 00 E8 01 00 00 DC 00 00 00 00 00 00  .....č...Ü......
```

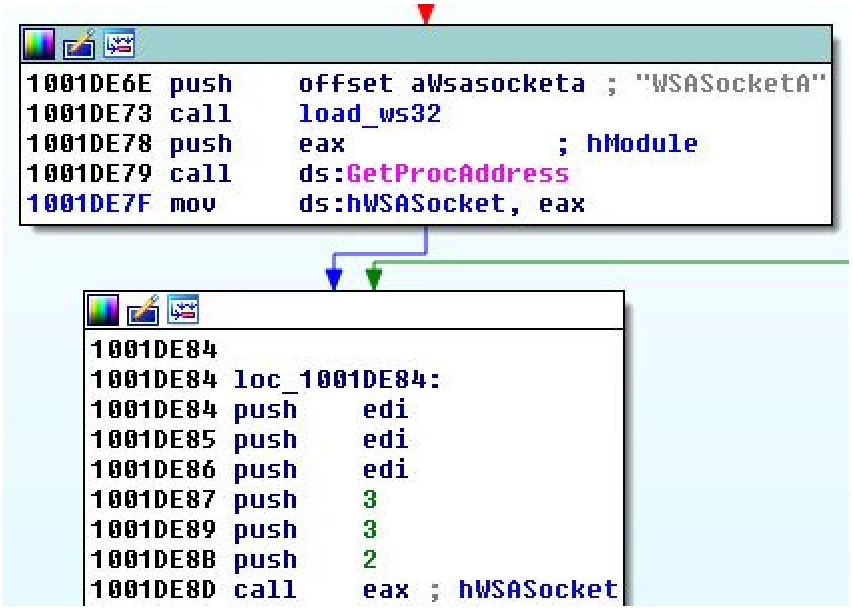
After this small modification, the dumped image can be parsed as a normal PE file (321a2f0abe47977d5c8663bd7a7c7d28). Sections are not named, but all the content is valid:



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▷	400	1E800	1000	1E7BE	60000020	0	0	0
▷	1EC00	4000	20000	3F9C	40000040	0	0	0
▷	22C00	200	24000	7624	C0000040	0	0	0
▷	22E00	200	2C000	4	40000040	0	0	0
▷	23000	2200	2D000	21F4	42000040	0	0	0

File characteristics describes the payload as a DLL, however, it doesn't have any export table, so we cannot read it's original name.

Looking at the imports loaded by this piece we can suspect that it is the final payload. It loads and uses many functions related to the network communication, i.e:

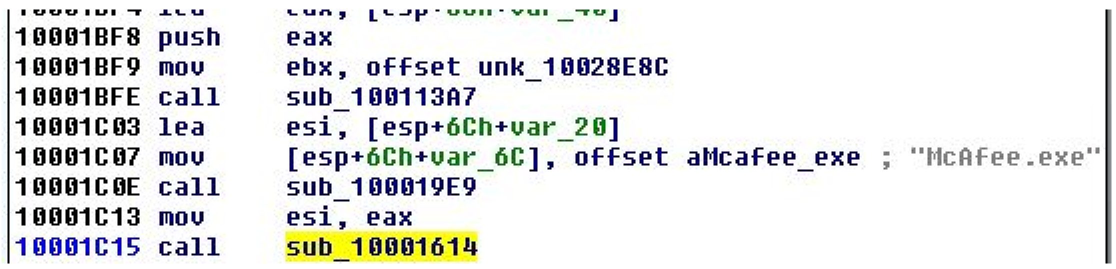


```
1001DE6E push    offset aWsasocketa ; "WSASocketA"
1001DE73 call    load_ws32
1001DE78 push    eax                ; hModule
1001DE79 call    ds:GetProcAddress
1001DE7F mov     ds:hWSASocket, eax
```

```
1001DE84
1001DE84 loc_1001DE84:
1001DE84 push    edi
1001DE85 push    edi
1001DE86 push    edi
1001DE87 push    3
1001DE89 push    3
1001DE8B push    2
1001DE8D call    eax ; hWSASocket
```

We can also find the fragment responsible for retrieving the local IP of the current machine and performing LAN scanning that we observed during behavioral analysis.

Authors took care so that the payload will not be run independently. That’s why they checks if all the elements are called in the expected order. We can find hardcoded names of the main elements, used for the check:



```
10001BF8 push    eax
10001BF9 mov     ebx, offset unk_10028E8C
10001BFE call    sub_100113A7
10001C03 lea     esi, [esp+6Ch+var_20]
10001C07 mov     [esp+6Ch+var_6C], offset aMcafee_exe ; "McAfee.exe"
10001C0E call    sub_100019E9
10001C13 mov     esi, eax
10001C15 call    sub_10001614
```

Conclusion

Malware authors often use fake icons and descriptions in order to disguise as a legitimate product, but this type of attack is going a step forward. Authors used an original McAfee application and hijacked the DLL that it uses, in order to run the malicious code. To make detection more difficult, they tangled elements with each other. None of them can do malicious actions on it’s own. That’s why, tools that scan each module separately may fail to detect the malicious behavior.

Users are more vigilant about executables – but this time, neither EXE nor DLL file contained the malicious code – they were just used as loaders of the shellcode.

Malwarebytes Anti-Malware detects this threat as ‘Trojan.Korplug’.

Appendix

http://e.gov.vn/theo-doi-ngan-chan-ket-noi-va-xoa-cac-tap-tin-chua-ma-doc-a-NewsDetails-37486-14-186.html – info from Vietnamese CERT

http://blog.trendmicro.com/trendlabs-security-intelligence/new-wave-of-plugx-targets-legitimate-apps/ – similar attack from 2013

http://www.welivesecurity.com/2014/11/12/korplug-military-targeted-attacks-afghanistan-tajikistan/ – about the Korplug RAT targeting military of Afganistan and Tajikistan

https://www.blackhat.com/docs/asia-14/materials/Haruyama/Asia-14-Haruyama-I-Know-You-Want-Me-Unplugging-PlugX.pdf – Korplug RAT analysis (presentation from BlackHat) https://www.f-secure.com/documents/996508/1030745/nanhaishu_whitepaper.pdf – about NanHaiShu APT

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.