Reverse Engineering Malware

Part 1

Author :Arunpreet Singh

Blog : https://reverse2learn.wordpress.com

MD5 Hash : 1d8ea40a41988b9c3db9eff5fce3abe5

This is First Part of 2 Part Series .This Malware Drops A File (All malwares do it usually)..So in This Part We will only Analyze Dropper and Next Part We will Analyze Dropped File.

Originally Sample is Downloaded from KernelMode.info ..It is very Good Place for Malware Samples and Reverse Engineering .I Uploaded the Sample to sendspace ..The Password to File is “infected”

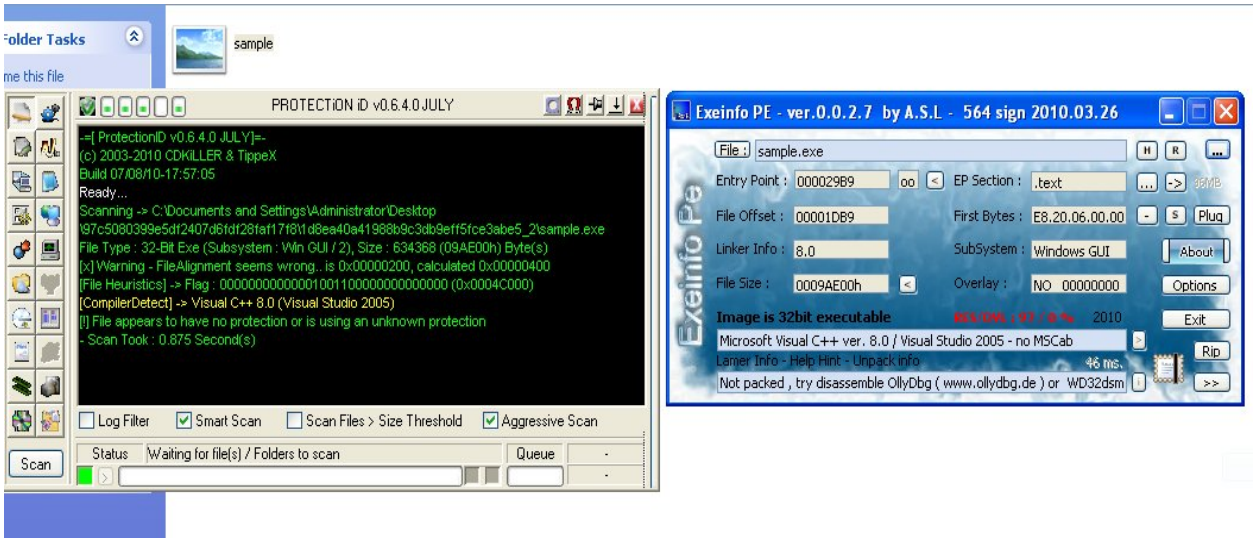
Link

http://www.sendspace.com/file/to53wo

Anyway Start With Basic Stuff..Check it with ExeInfo/Protection ID for Packer Detection or Compiler Detection .

NOTE: I Have Dedicated Virtual Machine For Malware Analysis .I recommend You to have same..

Here are Results From ExeInfo/Protection ID



So Sample is Not Packed .:D (Normally Malwares Are packed)

Compiler Detected : Visual C++ 2008

Fine Till Now

Visual C++ Targets are Kind of Ideal For Reversing ..Unlike Delphi Targets That Contain annoying Calls..VC++ Targets are Relatively Easier to Reverse .

Debugger /Disassembler we are going to use are

1)Ollydbg

2)IDA

I have a habit of Running Both IDA and Ollydbg parallely .IDA is very Powerful Due to Its Features Like Renaming the Variables, Functions ,Locations and Cross Reference etc ..Ollydbg is my Personal Favorite Debugger.

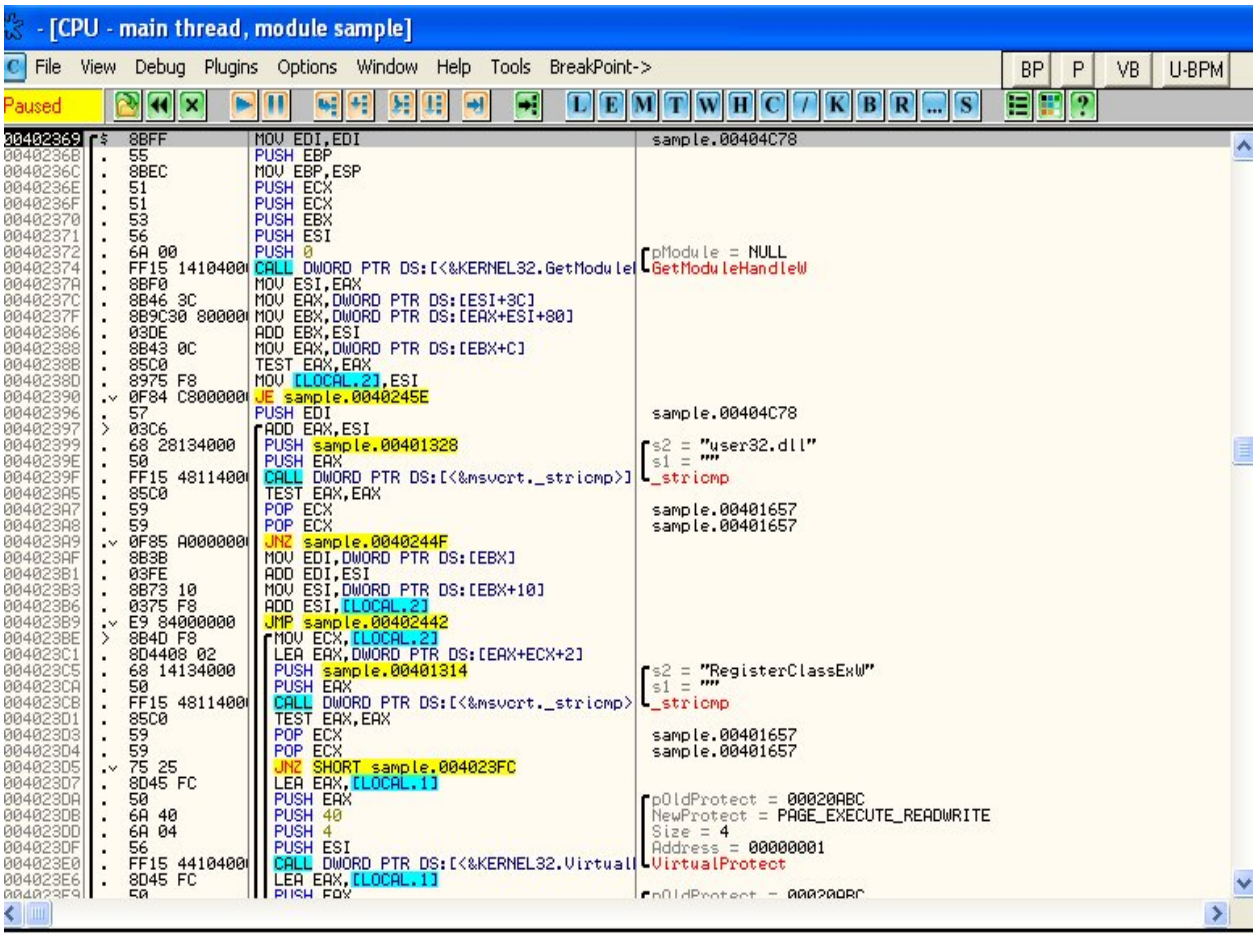
Also This Article is mainly to demonstrate Reverse Code Engineering ..I will try to Reverse Engineer Important parts of Malware .

Trace into Ollydbg Till WinMain = 00401648 or Use IDA ..IDA By default Start From WinMain

So lets start Analyzing from WinMain

```
00401648  /$  8BFF        MOV EDI,EDI                  //Do Nthing
0040164A  |.  55          PUSH EBP                     //Standard Function Start –Save  Frame Pointer
0040164B  |.  8BEC        MOV EBP,ESP                  //Move Stack Pointer to  EBP
0040164D  |.  83EC 1C     SUB ESP,1C                   //Allocate 1C (28) Bytes For Local Variables
00401650  |.  56          PUSH ESI                     //Save Registers Before Calling
00401651  |.  57          PUSH EDI
00401652  |.  E8 120D0000   CALL sample.00402369
```

Lets Trace Into This Call.. Disassembly Inside Call Looks Like This



Lets Start From Something Interesting.. We have A Call To API "GetModuleHandleW" ,the argument passed is 0 .

We All Know GetModuleHandleW(NULL)..Returns Imagebase of Currently Loaded Executable in EAX ..So This Call returns the Imagebase of sample.exe ..Next few Lines are Interesting

0040237A |. 8BF0 MOV ESI,EAX //Now ESI Contain Imagebase

0040237C |. 8B46 3C MOV EAX,DWORD PTR DS:[ESI+3C] //Get NT HEADER OFFSET

0040237F |. 8B9C30 800000 MOV EBX,DWORD PTR DS:[EAX+ESI+80] //Image_import_Directory

00402386 |. 03DE ADD EBX,ESI //Address Of _IMAGE_IMPORT_DESCRIPTOR Structure

00402388 |. 8B43 0C MOV EAX,DWORD PTR DS:[EBX+C] //Point to Name Field of _IMAGE_IMPORT_DESCRIPTOR

To Understand Above Code ..You Need Some Basic Understanding of PE Format ...

First Here We Have = Imagebase+3c

In PE Format First We have IMAGE_DOS_HEADER .....Lets Explore IMAGE_DOS_HEADER in Windbg

```
0:000> dt nt!_IMAGE_DOS_HEADER
ntdll!_IMAGE_DOS_HEADER
   +0x000 e_magic          : Uint2B
   +0x002 e_cblp           : Uint2B
   +0x004 e_cp             : Uint2B
   +0x006 e_crlc           : Uint2B
   +0x008 e_cparhdr        : Uint2B
   +0x00a e_minalloc       : Uint2B
   +0x00c e_maxalloc       : Uint2B
   +0x00e e_ss             : Uint2B
   +0x010 e_sp             : Uint2B
   +0x012 e_csum           : Uint2B
   +0x014 e_ip             : Uint2B
   +0x016 e_cs             : Uint2B
   +0x018 e_lfarlc         : Uint2B
   +0x01a e_ovno           : Uint2B
   +0x01c e_res            : [4] Uint2B
   +0x024 e_oemid          : Uint2B
   +0x026 e_oeminfo        : Uint2B
   +0x028 e_res2           : [10] Uint2B
   +0x03c e_lfanew         : Int4B
```

Ignore Other Fields.. Here we have e_lfanew at offset 0x3C.

e_lfanew Actually Contains the offset to PE Header

MOV EAX,DWORD PTR DS:[ESI+3C]

So above instruction is to get NT Header Offset

Imagebase is added as we are parsing the File in Memory ..Hope it is Clear Now

MOV EBX,DWORD PTR DS:[EAX+ESI+80]

So What We have is Load value at Imagebase+NT_HEADER+0x80 into EBX ..

Each PE File Contains Array of IMAGE_DATA_ DIRECTORY Structures .Lets Look Into IMAGE_DATA_DIRECTORY Structure

```
0:000> dt nt!_IMAGE_DATA_DIRECTORY
ntdll!_IMAGE_DATA_DIRECTORY
   +0x000 VirtualAddress   : Uint4B
   +0x004 Size             : Uint4B
```

So Each IMAGE_DATA_DIRECTORY Contains Two Fields Virtual Address and Size

NT_HEADER+80 Points to Import_table_address .. The Values of that Directory are

00400170 D4320000 DD 000032D4 ; Import Table address = 32D4

00400174 78000000 DD 00000078 ; Import Table size = 78 (120.)

I Took these values from Memory Window of Ollydbg ..

So what above instruction doing is getting the import table address ..

*Import table is very important concept ..It basically contains the info about imported functions/Dlls

Add Imagebase with Import Table address as we are parsing file in memory

ADD EBX,ESI

Next Instruction is

MOV EAX,DWORD PTR DS:[EBX+C]

This is Interesting .. Import Table is actually a Array of IMAGE_IMPORT_DESCRIPTOR.. Each IMAGE_IMPORT_DESCRIPTOR Structure Contains a Info about Single DLL and Info about Functions imported from this DLL ..SO NO. of IMAGE_IMPORT_DESCRIPTOR= No. of DLLs

Lets Look Into IMAGE_IMPORT_DESCRIPTOR Closely

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
  _ANONYMOUS_UNION union {
    DWORD Characteristics;
    DWORD OriginalFirstThunk;
  } DUMMYUNIONNAME;
  DWORD TimeDateStamp;
  DWORD ForwarderChain;
  DWORD Name;                          //Offset 0xC
  DWORD FirstThunk;                    // offset  0x10
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

So in Above Instruction ESI is Pointing to Name From IMAGE_IMPORT_DESCRIPTOR

0040238D |. 8975 F8 MOV [LOCAL.2],ESI //Save ESI Into Local_var

So Next is Loop ..We Can Easily See this in our dear Ollydbg ..Lets Look Into Loop

```
00402397  |> /03C6         /ADD EAX,ESI                          //Get Name of DLL In Memory
00402399  |. |68 28134000   |PUSH sample.00401328                 ; /s2 = "user32.dll"   //Constant String
0040239E  |. |50           |PUSH EAX                             ; |s1 = 000036E4 ???              /
0040239F  |. |FF15 48114000 |CALL DWORD PTR DS:[<&msvcrt._stricmp>]  ; \_stricmp
004023A5  |. |85C0         |TEST EAX,EAX
004023A7  |. |59           |POP ECX                              ;  sample.00404C78
004023A8  |. |59           |POP ECX                              ;  sample.00404C78
004023A9  |. |0F85 A0000000 |JNZ sample.0040244F
```

So what above instructions Doing are

1)Get Address of DLL_NAME in Memory

2) Compare the DLL_NAME with " user32.dll"

3) IF DLL_NAME !="user32.dll" then Go to NEXT IMAGE_IMPORT_DESCRIPTOR Structure

4)Go to Step 1

So this Loop Continues Untill DLL NAME IS "user32.dll"

So lets Look What Happen When condition is True ..I mean DLL NAME == "user32.dll"

Jump AT 004023A9 is Conditional Jump ..That Not taken If DLL Name ==”user32.dll” .So lets Look Into Code Below the conditional Jump when we Got a Match with DLL Name .

004023AF |. 8B3B MOV EDI,DWORD PTR DS:[EBX] //Get Address OF RVA of IAT in EDI

004023B1 |. 03FE ADD EDI,ESI // Get In Memory Address

004023B3 |. 8B73 10 MOV ESI,DWORD PTR DS:[EBX+10] // RVA of FirstThunk

004023B6 |. 0375 F8 ADD ESI,[LOCAL.2] // IN Memory Address of FirstThunk

So what Above Code Does is Get IN Memory Address(Virtual Address of FirstThunk)..

(Look into PE format to Know More About IAT)

This Whole Procedure is Actually to Parse the Names of APIs Imported By DLL.

```
004023BE  |> /8B4D F8      MOV ECX,[LOCAL.2]
004023C1  |. |8D4408 02    LEA EAX,DWORD PTR DS:[EAX+ECX+2]    //Point To Name Of API
```

So finally Now it Point to Name of APIs Imported By User32.dll

```
004023BE  |> /8B4D F8          MOV ECX,[LOCAL.2]                  ;  sample.00400000
004023C1  |. |8D4408 02       LEA EAX,DWORD PTR DS:[EAX+ECX+2]
004023C5  |. |68 14134000     PUSH sample.00401314               ; /s2 = "RegisterClassExW"
004023CA  |. |50              PUSH EAX                           ; |s1 = "TranslateMessage"
004023CB  |. |FF15 48114000 CALL DWORD PTR DS:[<&msvcrt._stricmp>] ; \_stricmp
```

Check If Current API Name == RegisterClassExW (here it is not equal as First API Imported is Translate Message).

```
004023D1  |. |85C0          TEST EAX,EAX                       ;  sample.004037D2
004023D3  |. |59            POP ECX                            ;  sample.004037D2
004023D4  |. |59            POP ECX                            ;  sample.004037D2
004023D5  |. |75 25         JNZ SHORT sample.004023FC
```

If API Name Matched then DO not JUMP(Execute the Code Below) If Not Matched then JUMP

```
004023D7  |. |8D45 FC       LEA EAX,[LOCAL.1]
004023DA  |. |50            PUSH EAX                           ; /pOldProtect = sample.004037D2
004023DB  |. |6A 40         PUSH 40                            ; |NewProtect = PAGE_EXECUTE_READWRITE
004023DD  |. |6A 04         PUSH 4                             ; |Size = 4
004023DF  |. |56            PUSH ESI                           ; |Address = <&USER32.TranslateMessage>
```

004023E0 |. |FF15 44104000 CALL DWORD PTR DS:[<&KERNEL32.VirtualP>; \VirtualProtect

004023E6 |. |8D45 FC LEA EAX,[LOCAL.1]

IF API NAME MATCHED THEN CHANGE THE PERMISSION FOR THAT ADDRESS (ESI POINT TO ADDRESS OF API) BY USING VirtualProtect

New Protect = PAGE_EXECUTE_READWRITE

MAKE IT WRITABLE

Size= 4

HERE SIZE = 4 Bytes as Probably it Going to Overwrite the API Address (As we are on 32 bit Arcitecture so Address = 4bytes=32 bits)

Address = ESI (API ADDRESS) (Address of Target API)

004023E9 |. |50 PUSH EAX ; /pOldProtect = sample.004037D2

004023EA |. |C706 EF194000 MOV DWORD PTR DS:[ESI],sample.004019EF

OverWrite the API Address With 004019EF (Other Function Address)

004023F0 |. |FF75 FC PUSH [LOCAL.1] ; |NewProtect = PAGE_READONLY|PAGE_WRITECOPY

004023F3 |. |6A 04 PUSH 4 ; |Size = 4

004023F5 |. |56 PUSH ESI ; |Address = <&USER32.TranslateMessage>

004023F6 |. |FF15 44104000 CALL DWORD PTR DS:[<&KERNEL32.VirtualP>; \VirtualProtect

Restore the Original Permission Using VirtualProtect

Same Is For Next Part Of LOOP .. it checks API Against “CreateWindowExW”.If Name Matched then Use VirtualProtect to Make that Memory portion Writable .Then Change Address and Again Restore Permission

So let me Write A Pseudo Code To Describe what Just Happened in This LOOP

```
Parse IMAGE_IMPORT_DESCRIPTOR
If   stricmp(Image_Import_descriptor->Name,"user32.dll)  //Label2
{
      Parse  using  FirstThunk ..Get API NAMES..
     If stricmp(Current_API ,"RegisterClassExW")       //Label1
     {
       VirtualProtect(Address_of_API,Size(4Bytes), PAGE_EXECUTE_READWRITE,PoldProtec)
       //  Make it Writable
        Address_of_API= 004019EF
      VirtualProtect()   //Restore original Permissions
           }
       Else if(Stricmp(Current API,"CreateWIndowExW")
         {
           VirtualProtect(Address_of_API,Size(4Bytes), PAGE_EXECUTE_READWRITE,PoldProtec)
          //  Make it Writable
             Address_of_API= 00402228
         VirtualProtect()   //Restore orginal Permissions
              }
        Else
      {
       Get Next  API NAME
          }     //Start From Label 1
Else
{Get Next IMAGE_IMPORT_DESCRIPTOR TABLE
}  //Start From Label 2   ..Once Two Functions are matched Loop Terminates
```

SO After End OF LOOP We have

Address_RegisterClassExW=004019EF

Address_CreateWindowEx=00402228

Finally Functions Ends and Return ...So main Motive is this Function to Make Some Modification in IAT

After Call ..There are some Calls to Resources..More Likely Fake Calls ..As Called Resource does not Exist

```
00401657  |.  8B7D 08      MOV EDI,[ARG.1]                          // Move ImageBase Into EDI
0040165A  |.  8B35 C8104000 MOV ESI,DWORD PTR DS:[<&USER32.LoadStringW>]     ;  USER32.LoadStringW
00401660  |.  6A 64        PUSH 64                                  ; /Count = 64 (100.)
00401662  |.  68 C04A4000   PUSH sample.00404AC0                     ; |Buffer = sample.00404AC0
00401667  |.  6A 67        PUSH 67                                  ; |RsrcID = 67 (103.)          //It Actually Never Exist
00401669  |.  57          PUSH EDI                                  ; |hInst = 00400000
0040166A  |.  FFD6        CALL ESI                                  ; \LoadStringW
```

This ResourceID does not Exist .Check the GetLastError Field Under the ollydbg ERROR_RESOURCE_TYPE_NOT_FOUND.So Look Like A Fake Call to make Program Look Legitmate(may be)

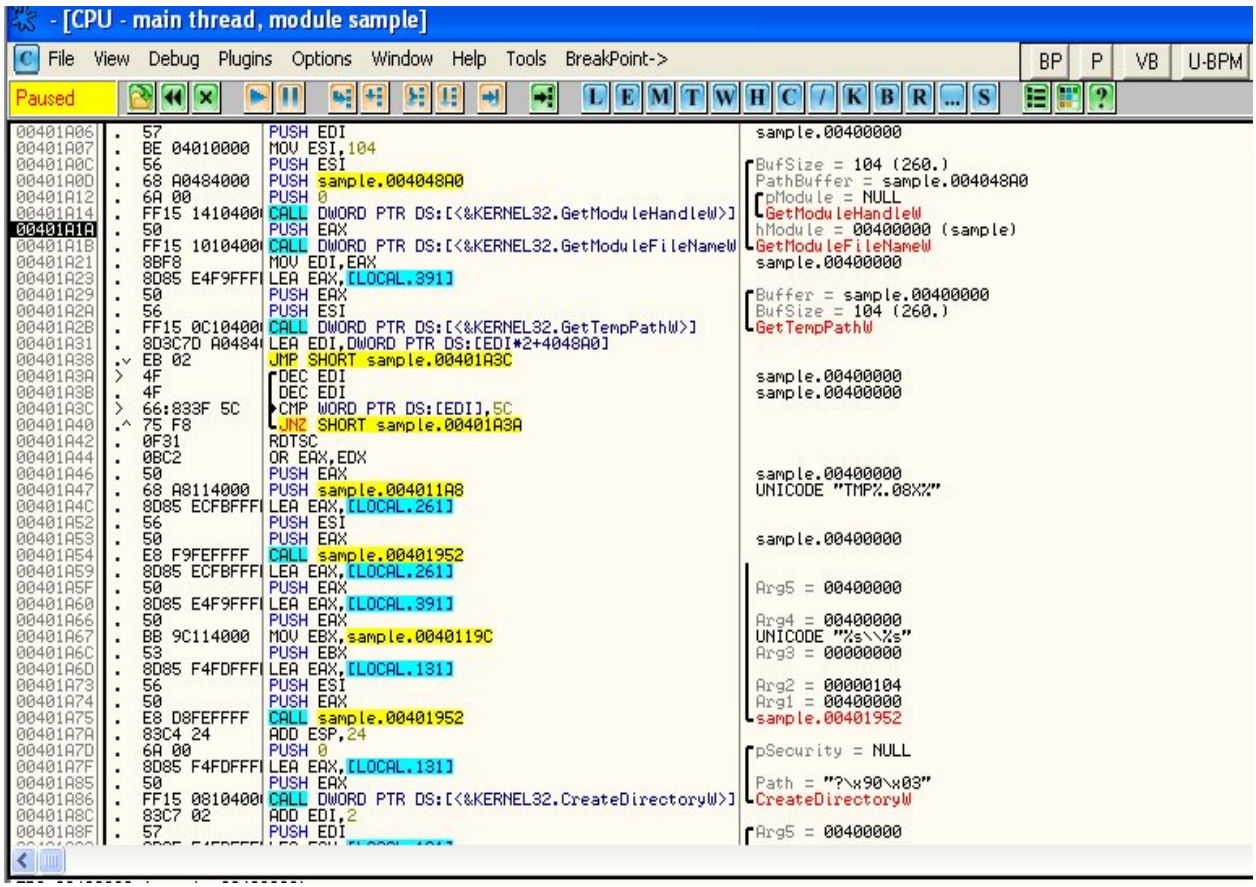
```
00401678  |.  57          PUSH EDI                          ; Arg1 = 00400000       // ImageBase As Parameter
00401679  |.  E8 4DFFFFFF   CALL sample.004015CB
```

Trace Into This Call... Again Few Call to Resources ....LoadIcon,LoadResouce ..Nthing Important

After that We See a call

```
00401637  |.  50          PUSH EAX                          ; pWndClassEx = 0006FECC
00401638  |.  FF15 00114000 CALL DWORD PTR DS:[<&USER32.RegisterClassExW>]    ;\RegisterClassExW
```

Remember the Address of RegisterClassExW is altered in starting ..Now Instead of Going to user32.dll ,it points to another function inside the executable..Trace Into this Function



Here In this Function We can See Some Interesting Calls Such As

GetModuleFileNameA=>Here it is to get the full path of Currently Executing File (As GetModuleHandleW with Argument NULL is used to Get Handle For It)

GetTempPathW=> As Name Suggests Retrieves a Path to Temp Files

Then We can see a Call to Function 00401952 =>

I Checked this Function ..This internally Calls CRT function _vsnwprintf...Which is used For String Manipulation (String Formatting)..

First Call to this Function Returns =TMP1CDFDEBF (It is Directory name..i know it as I analyzed it)

Second Call To This Function Returns a String = C:\\DOCUME~1\\ADMINI~1\\LOCALS~1\\Temp\\\\TMP1CDFDEBF

This LookLike A Location To Drop A File

00401A7D |. 6A 00 PUSH 0 ; /pSecurity = NULL

00401A7F |. 8D85 F4FDFFFF LEA EAX,[LOCAL.131] ; |

00401A85 |. 50 PUSH EAX ; |Path = "C:\\DOCUME~1\\ADMINI~1\\LOCALS~1\\Temp\\\\TMP1CDFDEBF"

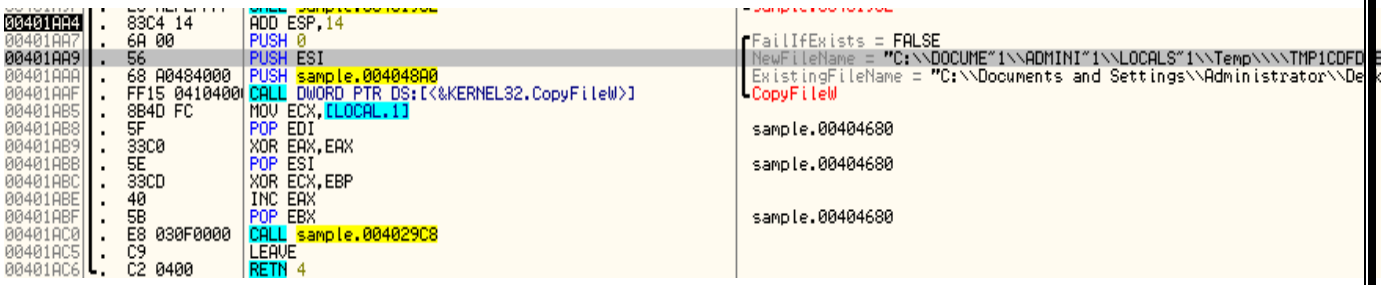
00401A86 |. FF15 08104000 CALL DWORD PTR DS:[<&KERNEL32.CreateDirectoryW>]

So Here it Creates A Directory ..Nthing to Explain..

Then Again it Call to 401952(String Formatting) to Generate File Path ..output is

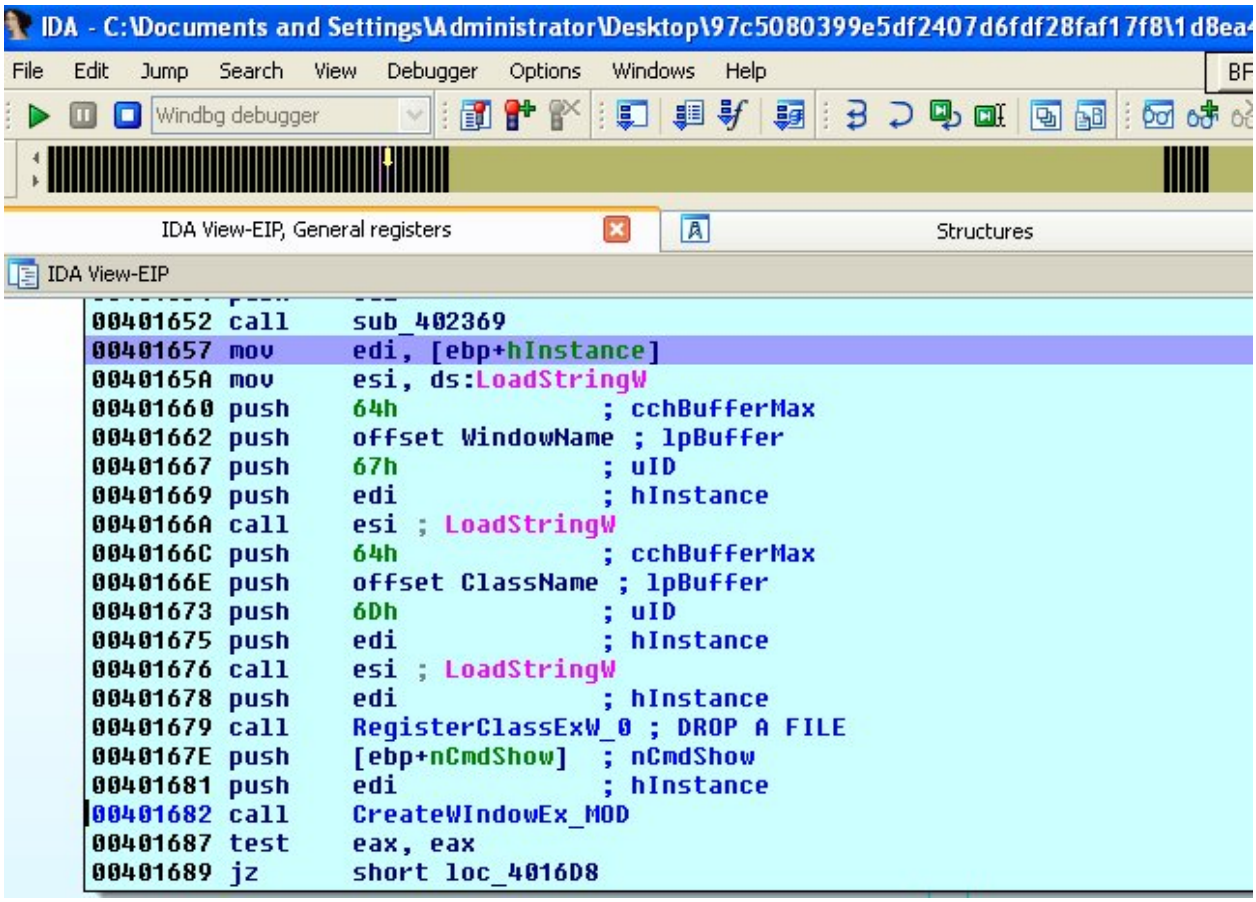
C:\\DOCUME~1\\ADMINI~1\\LOCALS~1\\Temp\\\\TMP1CDFDEBF\\sample.exe

So finally this is Path to Drop File



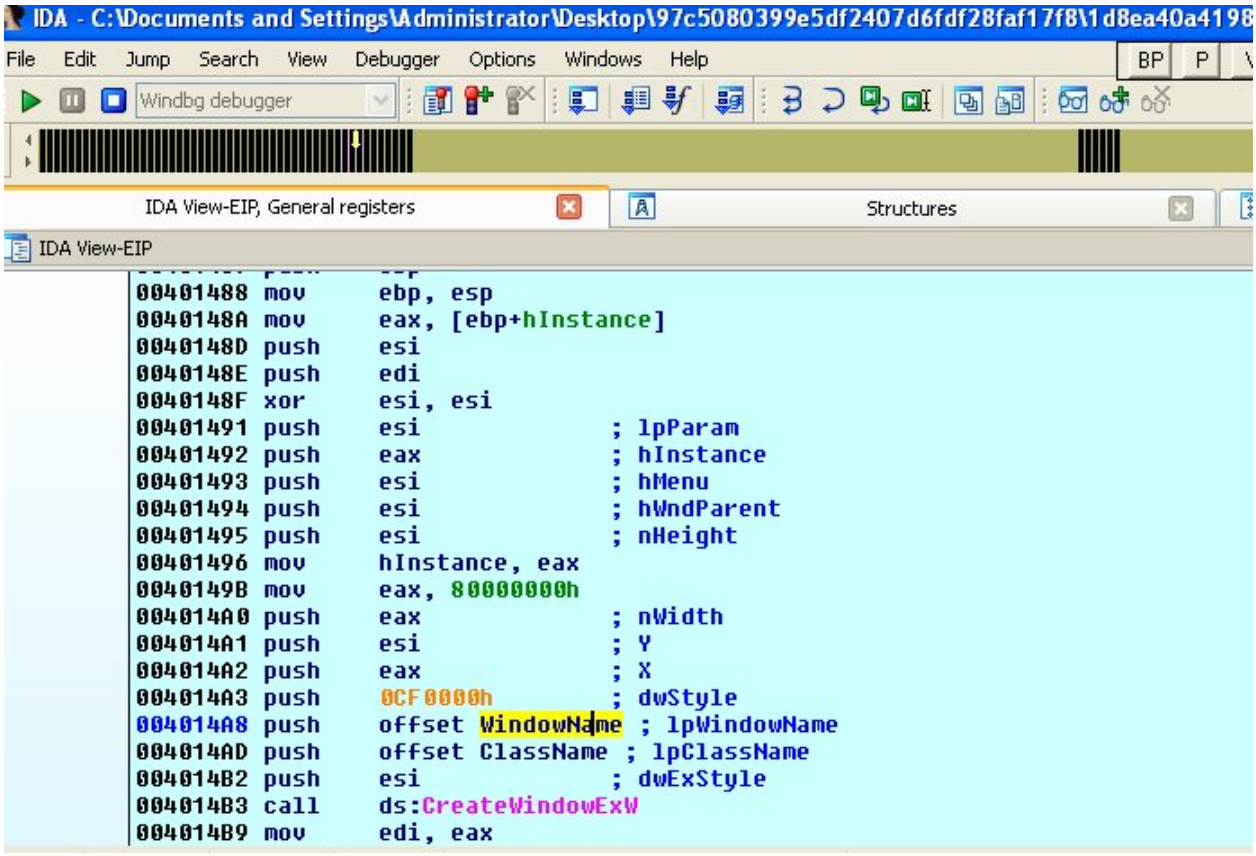
As Shown in Pic ..then Finally there is call to CopyFileW ..So finally it Drops File to Location Mentioned above..It Actually Copy/Drop the Same File That is being Executed ..

So After Dropping A File Our Function Ends ..



So Till Now We Analyzed the RegisterClassExW_0 Function ... Now Trace Into CreateWindowEx_MOD(Modified CreateWindowEx)...I Call this Function CreateWindowEx_MOD as it internally Calles Modified CreateWIndowExW API..

Lets Trace Into This



All Parameters Original/Necessary are passed to CreateWindowEx to make it Look genuine ..Now Step Into CreateWindowExW

So inside CreateWindowExW(that actually is Function 00402228)..We can See some Intersting API Calls

Such As CreateProcessW,GetThreadContext,SetThreadContext, ,WriteProcessMemory...Lets Check what They Exactly Doing ..

Then we have a call to CreateProcessW(W in the end is to indicate a Unicode Version)... CreateProcessW in simple words used to Create a Process ...check MSDN For Other info

http://msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx

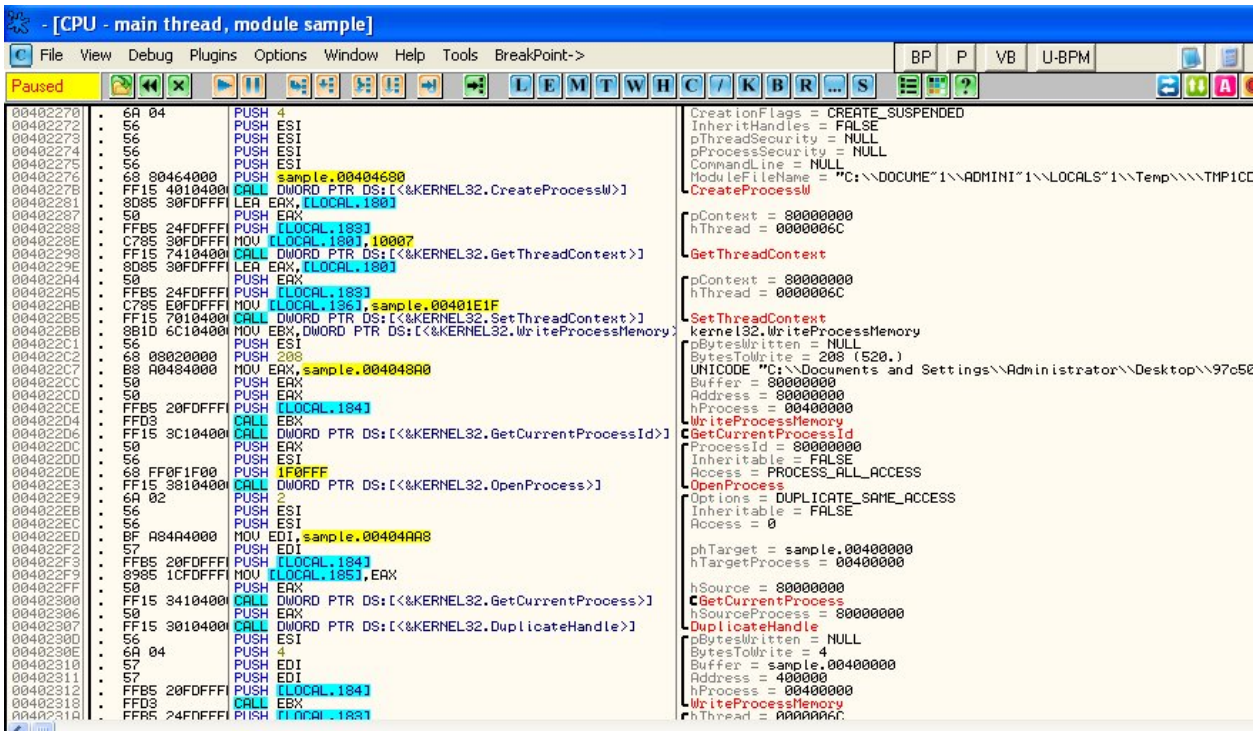
Lets Check the Paramters Passed to CreateProcessW

00402266 |. 50 PUSH EAX ; /pProcessInfo = 0006FB90

00402267 |. 8D85 D8FCFFFF LEA EAX,[LOCAL.202] ; |

0040226D |. 50 PUSH EAX ; |pStartupInfo = 0006FB90

```
0040226E  |.  56          PUSH ESI                 ; |CurrentDir = NULL
0040226F  |.  56          PUSH ESI                 ; |pEnvironment = NULL
00402270  |.  6A 04       PUSH 4                   ; |CreationFlags = CREATE_SUSPENDED
00402272  |.  56          PUSH ESI                 ; |InheritHandles = FALSE
00402273  |.  56          PUSH ESI                 ; |pThreadSecurity = NULL
00402274  |.  56          PUSH ESI                 ; |pProcessSecurity = NULL
00402275  |.  56          PUSH ESI                 ; |CommandLine = NULL
00402276  |.  68 80464000   PUSH sample.00404680   ; |ModuleFileName =
"C:\\DOCUME~1\\ADMINI~1\\LOCALS~1\\Temp\\\\TMP1CDFDEBF\\sample.exe"
0040227B  |.  FF15 40104000 CALL DWORD PTR DS:[<&KERNEL32.CreateProcessW>]      ; \CreateProcessW
```



The Paramters Highlighted in Red Color are Important ...Let me explain it

```
00402270  |.  6A 04       PUSH 4                   ; |CreationFlags = CREATE_SUSPENDED
```

Acc. To MSDN

CREATE_SUSPENDED
0x00000004

The primary thread of the new process is created in a suspended state, and does not run until the ResumeThread function is called.

Hope it is Clear Now....In Case of Malware if Process is created in SUSPENDED mode then it Most probably means it will be modified

Other interesting Paramter is

```
00402276  |.  68 80464000   PUSH sample.00404680                    ; |ModuleFileName =
"C:\\DOCUME~1\\ADMINI~1\\LOCALS~1\\Temp\\\\TMP1CDFDEBF\\sample.exe"
```

So this mean our sample file starts the dropped file into SUSPENDED Mode ...

*Also u can think like that ...what is meaning of dropping Duplicate/Same File and then Run it ..Does not making sense ..Dropping File and then run it ..then again the file will do same (Off course u can think that file can check its running location and can change its behavior acc. To it but it is not in this case..)...So it will be kind of very Stupid malware that Just Drops itself and do nthing ..:P.. so this Philosophy also provide some hint that there will be some modification in the Dropped File Process..Also We can See Some APIs Like WriteProcessMemory

WriteProcessMemory is basically used for InterProcess Communication ...to Write the Given Data in Desired Location in Remote Process.

So All this make Sense that Our malware will make Some Modification in its child Process i.e Dropped File Process .Lets Continue Analyzing

```
00402287  |.  50          PUSH EAX                               ; /pContext = 0006FBE8
00402288  |.  FFB5 24FDFFFF PUSH [LOCAL.183]                     ; |hThread = 00000048 (window)
0040228E  |.  C785 30FDFFFF>MOV [LOCAL.180],10007                ; |
00402298  |.  FF15 74104000 CALL DWORD PTR DS:[<&KERNEL32.GetThreadContext>]     ; \GetThreadContext
```

GetThreadContext = Retrieves the context of the specified thread (Simple and smart Defination from MSDN)

pContext = Holds the CONEXT Structre..I.e it Value Of registers obtained ..Here it is 006FBE8

hThread =Handle of thread....Here in this Case it Contains the Handle of main thread of Dropped File Process(I will call it Dropped Process)

Check Context Structre in Windbg ..WIndbg is Pretty Handy Tool to Examine the Data Structres in Windows ..also Shows Offsets ...that's Really Useful...

```
0:000> dt nt!_CONTEXT
ntdll!_CONTEXT
   +0x000 ContextFlags     : Uint4B
   +0x004 Dr0              : Uint4B
   +0x008 Dr1              : Uint4B
   +0x00c Dr2              : Uint4B
   +0x010 Dr3              : Uint4B
   +0x014 Dr6              : Uint4B
   +0x018 Dr7              : Uint4B
   +0x01c FloatSave        : _FLOATING_SAVE_AREA
   +0x08c SegGs            : Uint4B
   +0x090 SegFs            : Uint4B
   +0x094 SegEs            : Uint4B
   +0x098 SegDs            : Uint4B
   +0x09c Edi              : Uint4B
   +0x0a0 Esi              : Uint4B
   +0x0a4 Ebx              : Uint4B
   +0x0a8 Edx              : Uint4B
   +0x0ac Ecx              : Uint4B
   +0x0b0 Eax              : Uint4B
   +0x0b4 Ebp              : Uint4B
   +0x0b8 Eip              : Uint4B
   +0x0bc SegCs            : Uint4B
   +0x0c0 EFlags           : Uint4B
   +0x0c4 Esp              : Uint4B
   +0x0c8 SegSs            : Uint4B
   +0x0cc ExtendedRegisters : [512] UChar
```

As You can See EAX is at offset 0xB0 ..

We have Context Structre Starting at 006FBE8

Context.EAX in Memory = 006FBE8+B0=006FC98

Why EAX is So Important ... in Case of SUSPENDED Process EAX Always Point To Entry Point

After Executing GetThreadContext

```
tack address=0006FBE8
AX=00000001

ddress  Hex dump
006FC98 B9 29 40 00 00 00 00 00
006FCA8 00 02 00 00 FC FF 06 00
006FCB8 4F 00 43 00 55 00 4D 00
```

We Have Value of Context.Eax

We have 006Fc98 = 004029B9 ..As Described Earlier it is Entry Point of Dropped Process

Now Examine Next Few Intersting Lines

```
004022A4  |.  50          PUSH EAX                          ; /pContext = 0006FBE8
004022A5  |.  FFB5 24FDFFFF PUSH [LOCAL.183]          ; |hThread = 00000048 (window) //Dropped Process thread
004022AB  |.  C785 E0FDFFFF>MOV [LOCAL.136],sample.00401E1F                //OverWrite EAX
004022B5  |.  FF15 70104000 CALL DWORD PTR DS:[<&KERNEL32.SetThreadContext>]
```

SetThreadContext= Sets the context for the specified thread....

As You Can See it Points to Same Address 0006FBE8

Check the Highlighted... Here LOCAL.136 = 006Fc98 ..So what this instruction doing is Overwriting the value at Location 006Fc98 with 00401E1F..

And Then We have a call to SetThreadContext...

So all this to change the Entry Point of Dropped Process By Overwriting the Eax in Context Structre.

Check the Below Snapshot to get things more Clear way ...IDA's naming feature make this tool ideal for reversing .

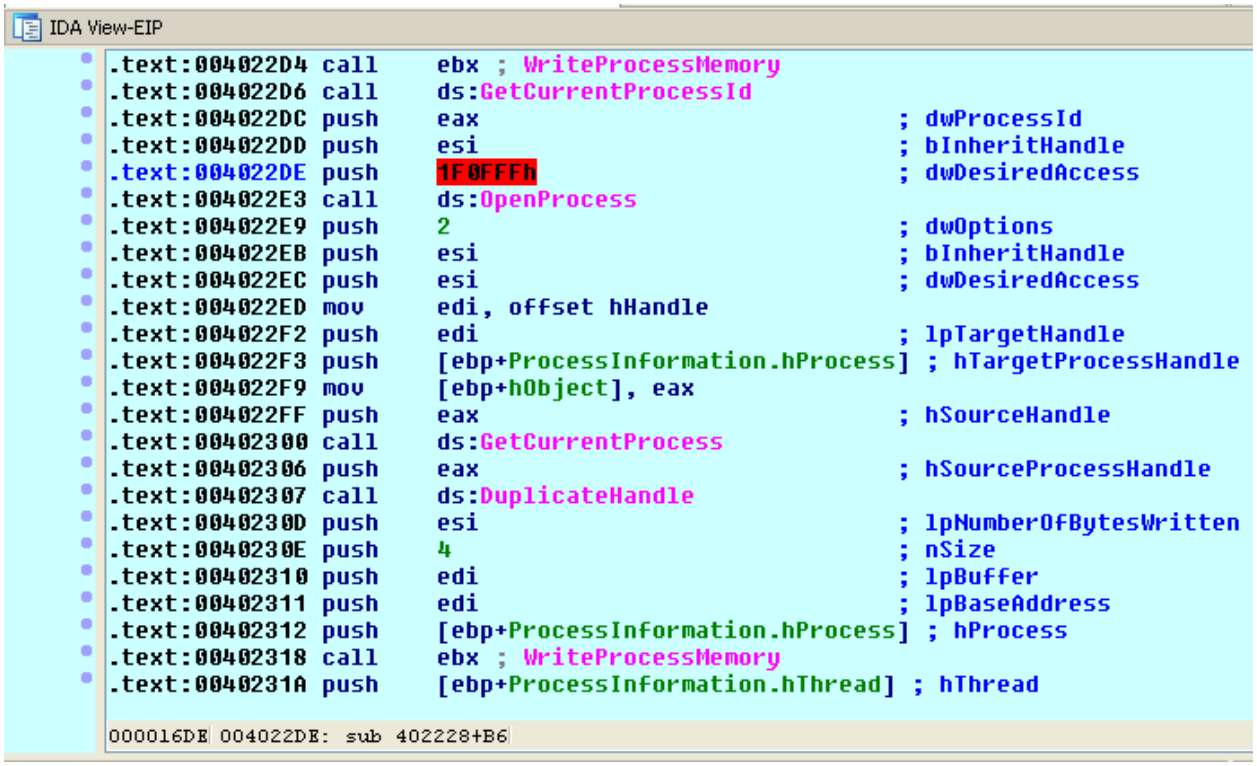
```
00402288 push    [ebp+ProcessInformation.hThread] ; hThread
0040228E mov     [ebp+Context.ContextFlags], CONTEXT_FULL
00402298 call    ds:GetThreadContext
0040229E lea     eax, [ebp+Context]
004022A4 push    eax              ; lpContext
004022A5 push    [ebp+ProcessInformation.hThread] ; hThread
004022AB mov     [ebp+Context._Eax], 401E1Fh ; NEW_ENTRY_POINT
004022B5 call    ds:SetThreadContext
```

So Entry Point is Changed ...Lets see what happened Next..Lets Analyze what happen Next

```
004022C1  |.  56          PUSH ESI                          ; /pBytesWritten = NULL
004022C2  |.  68 08020000   PUSH 208                         ; |BytesToWrite = 208 (520.)
004022C7  |.  B8 A0484000   MOV EAX,sample.004048A0                  ; |UNICODE "C:\\Documents and Settings\\Administrator\\Desktop\\97c5080399e5df2407d6fdf28faf17f8\\1d8ea40a41988b9c3db"
```

```
004022CC  |. 50          PUSH EAX                          ; |Buffer = sample.004048A0
004022CD  |. 50          PUSH EAX                          ; |Address = 4048A0
004022CE  |. FFB5 20FDFFFF PUSH [LOCAL.184]                |hProcess = 00000044     //Handle of  Dropped Process
004022D4  |. FFD3        CALL EBX                          ; \WriteProcessMemory
```

As I commented hProcess is Handle of Dropped Process...So what WriteProcessMemory Here Doing is Copying the Orginal Path of Sample To Dropped Process.(4048A0 contains path of Current Executable).You will come to know why this is copied to dropped process



Call to GetCurrentProcessId = This Returns the Process ID of Current Process in Eax

Next , OpenProcess API is called and ProcessID of Current Process is Passed a Parameter..This Means OpenProcess Attempt to Currently Executing Process with PROCESS_ALL_ACCESS (Red mark 1F0FFF=PROCESS_ALL_ACCESS)...If Evrything Fine then OpenProcess Will Return A handle to Local Process Object.

Next We Have a Call to DuplicateHanlde...This is Best Explained in MSDN ..Read it

http://msdn.microsoft.com/en-us/library/windows/desktop/ms724251%28v=vs.85%29.aspx

Then we have a call to WriteProcessMemory...In this call..we are Passing the Real Handle Obtained By Using DuplicateHandle to Dropped Process.. We Write The Handle at =404AA8 (Keep this in Mind)

If u read the MSDN ..then the Purpose of this WriteProcessMemory will be Clear to You ..

Then We have a call to Resume Thread

```
0040231A  |.  FFB5 24FDFFFF PUSH [LOCAL.183]                ; /hThread = 00000048   //Dropped Process

00402320  |.  FF15 1C104000 CALL DWORD PTR DS:[<&KERNEL32.ResumeThread>]
```

So Finally After making all the Necessary Changes in Dropped Process ...It Resumes The Dropped Process and then Dropped Process Start Executing ..

I did not execute ResumeThread Till Now ...All I want is to attach Ollydbg to to dropped process..

Here is a way how to do it ..

We have entry point of Dropped Process =401E1F h

So what we are Going to do is to trap the the Dropped Process in Infinite Loop...

I am going to Use PUPE Suite .. And Chaned First Two Bytes at 401E1F to EB FE (Write Down Orginal Bytes Before Changing Orginal Ones)

Check my previous post where i use same method if you are not getting it

https://reverse2learn.wordpress.com/2011/09/01/unprotecting-the-crypter/

So Now Execute ResumeThread.Now Dropped Process is Trapped in Infinite Loop..Attach Ollydbg to it ..Replace EB FE with Orginal Bytes (orginal Bytes : 8B FF)..

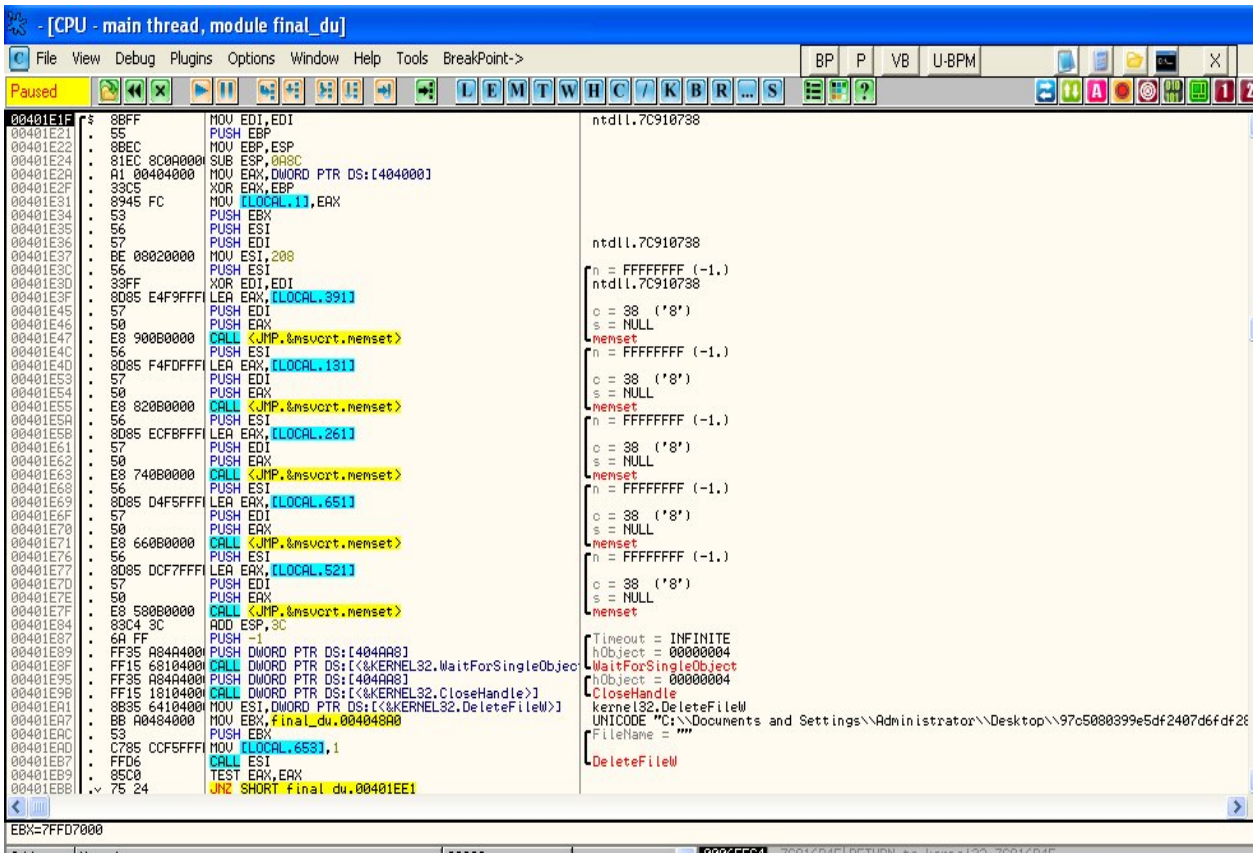
*I Recommend to DUMP the Process at this point as I include all the changes made by parent process..we are going to analyze this dumped process in next part of this series

So Finally What we have is Two Instances of Ollydbg one Debugging sample and other Debugging Dropped Process.

This is What We have in Sample Process...After Resuming Main Thread of Dropped Process ...it closes Hanldes and Finally Exits

```
.text:0040231A push    [ebp+ProcessInformation.hThread] ; hThread
.text:00402320 call    ds:ResumeThread
.text:00402326 push    [ebp+hObject]                    ; hObject
.text:0040232C mov     edi, ds:CloseHandle
.text:00402332 call    edi ; CloseHandle
.text:00402334 push    [ebp+ProcessInformation.hProcess] ; hObject
.text:0040233A call    edi ; CloseHandle
.text:0040233C push    [ebp+ProcessInformation.hThread] ; hObject
.text:00402342 call    edi ; CloseHandle
.text:00402344 push    esi                              ; uExitCode
.text:00402345 call    ds:GetCurrentProcess
.text:0040234B push    eax                              ; hProcess
.text:0040234C call    ds:TerminateProcess
```

Look at Code From Dropped Process



So What We have Calls To Memset ..After Memset Calls We have Very Intersting API Call

WaitForSingleObject (hObject,TimeOut)

hObject=404AA8 (Remember 2nd WriteProcessMemory Call ,Where we Write the handle obtained from DuplicateHandle at 404AA8 of Dropped Process)

TimeOut = INFINITE // Wait Until Object is Signaled

And After that Handle is Closed by Using CloseHandle

Then We Have a Call to DeleteFileW ...The Filename passed is Location of Sample Process

*Remember when we passed the location/path of sample Executable using first WriteProcessMemory Call

So I think Now it is clear how this implement Self Delete(Melt Feature)/Drop File ...

Dropped Process Wait for the Event From Sample Process ...When it is Signaled it go ahead and delete that file.

So First Part Ends Here ...In Next Part We Will Analyze the Dropped Process(I Dumped it when I restore the orginal Bytes after attaching Ollydbg to it).

I Love to get Your Feedback .You can Email me Comment on my blog

Blog :https://reverse2learn.wordpress.com/

Email :arunpreet90@gmail.com