BFOR - 418/618 - Reverse Engineering Malware
Advanced Static and Dynamic Analysis

Katta Sreenivasulu, Tarun Sai

November 11, 2023

Dr. Prinkle Sharma
University at Albany, SUNY

Contents

1 Advanced Static Dynamic Analysis

1.1 Advanced Static Analysis

- Disassembly/decompilation using IDA Pro or Ghidra to view assembly code and reconstruct high-level constructs
- Data flow analysis to track how data values are manipulated
- Control flow analysis to understand execution paths
- Symbolic execution to analyze multiple paths symbolically
- Taint analysis to track flow of malicious data
- Abstract interpretation to approximate program semantics
- Program slicing to extract relevant code portions
- Constraint solving to generate inputs that trigger behaviors
- Hybrid analysis combining static and dynamic techniques

1.2 Advanced Dynamic Analysis

- Debugging using IDA Pro or x64dbg to control execution and inspect internals
- Instrumentation to monitor memory, registry, API calls, network activity etc. at runtime
- Multi-path exploration to trigger different execution paths
- Taint tracking to follow data flows during execution
- Anti-anti-debug and anti-VM techniques to evade detection
- Dynamic symbolic execution to force exploration of multiple paths
- High-interaction honeypots to observe full attack behavior
- Sandbox automation and scripting to increase efficiency
- Hybrid analysis combining dynamic and static techniques

By analyzing the code and structure of malware, researchers can also better understand how it works and develop new techniques for defending against it.

In this lab, Advanced Static analysis is performed using a disassembly tool - IDA Pro.

1.3 Disassembler - IDA Pro

Dynamic analysis involves examining and analyzing malware after executing or running it. It mainly focuses on understanding the true functionality of malware, which cannot be determined using static analysis techniques to packed or obfuscation.

Static malware analysis involves examining the malware code without actually executing it. This allows analysts to understand the structure and logic of the malware. Advanced static analysis uses disassemblers like IDA Pro to convert the binary executable code into human-readable assembly code.

IDA Pro is one of the most popular disassemblers used for malware analysis. It allows analysts to interactively explore the disassembled code, rename variables, add comments, and understand the program logic. The key benefits of IDA Pro are its robust disassembly capabilities, extensive scripting and plug-in support, and advanced data reconstruction features.

However, static analysis has limitations - it cannot reveal behavior that emerges only when the code executes. This is where dynamic analysis comes in. Dynamic analysis executes the malware in a controlled environment like a sandbox and observes its behaviors at runtime. Advanced dynamic analysis uses tools like debuggers and instrumentation to analyze memory, system calls, network traffic etc. while the malware runs.

So in summary, IDA Pro plays a key role in conducting static analysis to understand malware code, which complements dynamic analysis to observe actual behaviors. The combination of advanced static and dynamic techniques provides comprehensive malware analysis capabilities.

Pro's:

- Powerful disassembly and decompilation capabilities
- Advanced manual and automated analysis.
- Extensive tooling ecosystem via plugins.
- Reconstructs code and data structures

Con's:

- Static analysis alone misses runtime behaviors
- Advanced use requires significant expertise
- Manual analysis is time consuming
- Difficult to analyze obfuscated malware

2 Methodology

Write Methodology here...

2.1 Analyzing Malware file Lab05-01.dll

1. **What is the address of DllMain?**

   The address of DLLMain is **.text:1000D02E**

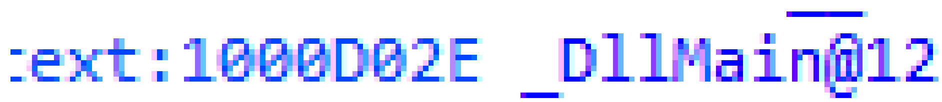


Figure 1: DLLMain Address

2. **Use the Imports window to browse to gethostbyname. Where is the import located?**

   It is located at address **100163CC**

Figure 2: gethostbyname

3. **How many functions call gethostbyname?**

   In Xrefs, we have gethostbyname referred in **18** for both pointer and read types, however it is only considered a call when a pointer is pointing to the function. So, we have **9 gethostbyname** function calls.

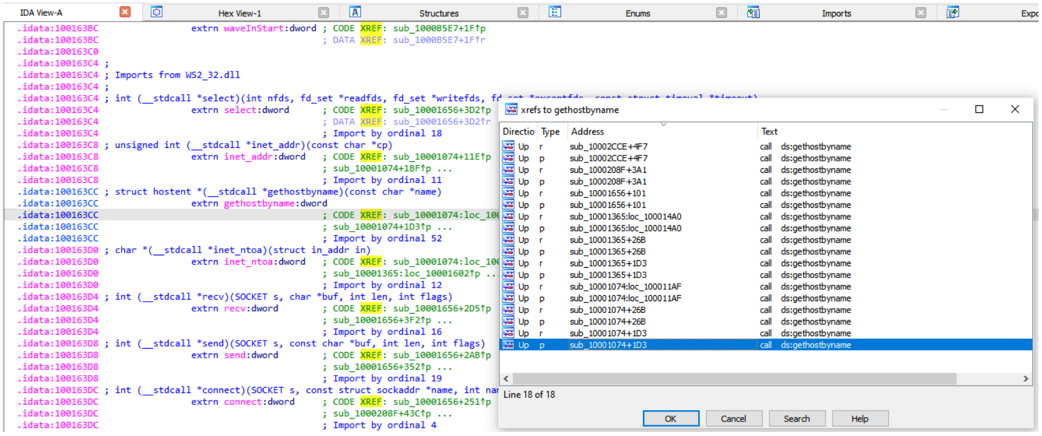


Figure 3: XREF's for gethostbyname

4. **Focusing on the call to gethostbyname located at 0x10001757, can you figure out which DNS request will be made?**

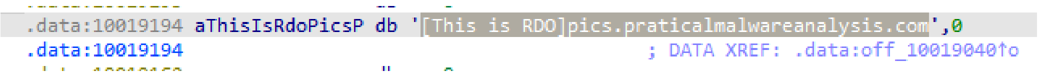


Figure 4: DNS request made by gethostbyname

5. **How many local variables has IDA Pro recognized for the subroutine at 0x10001656?**

   We have 23 local variables in total. From figure 5, they are present between **var_675 and WSAData**.

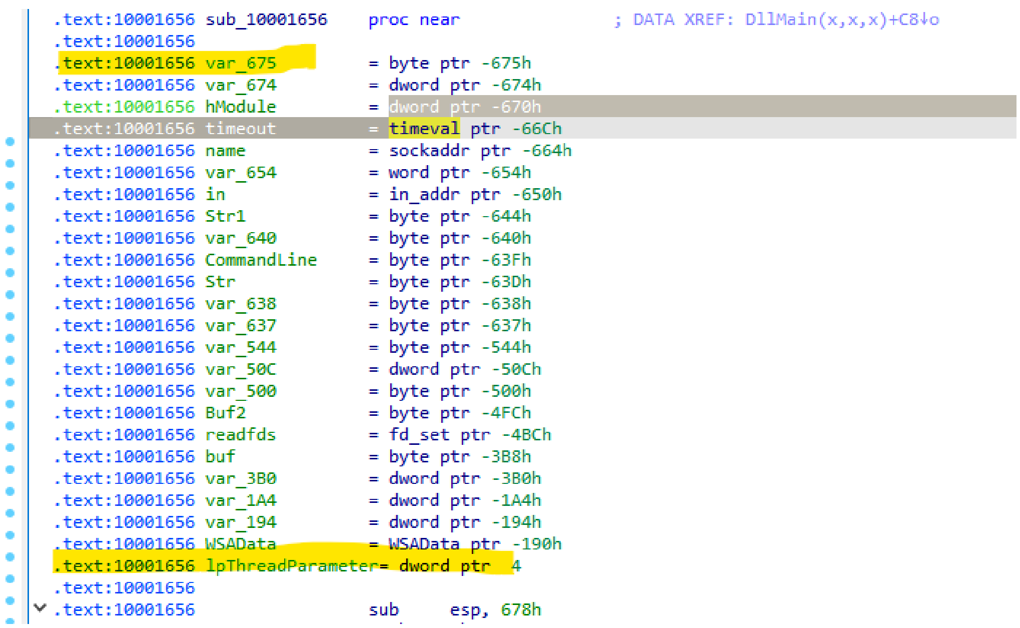


Figure 5: Local variables recognized

6. **How many parameters has IDA Pro recognized for the subroutine at 0x10001656?**

   For subroutine at **0x10001656**, we have only one parameter. From figure 6 below, it is

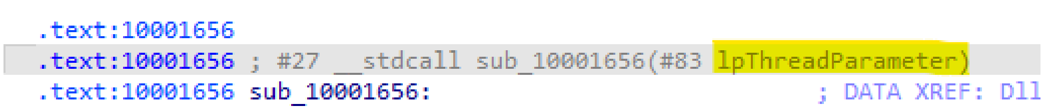
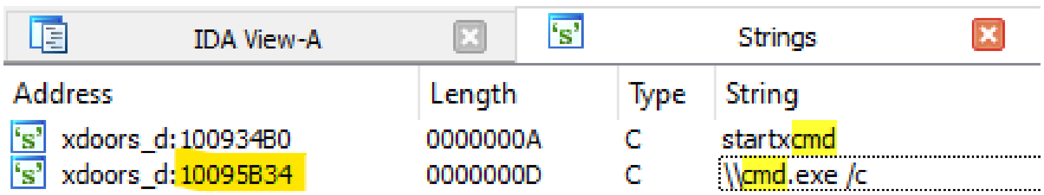


Figure 6: Parameter to subroutine sub_10001656 - lpThreadParameter

7. **Use the Strings window to locate the string cmd.exe in the disassembly. Where is it located?**

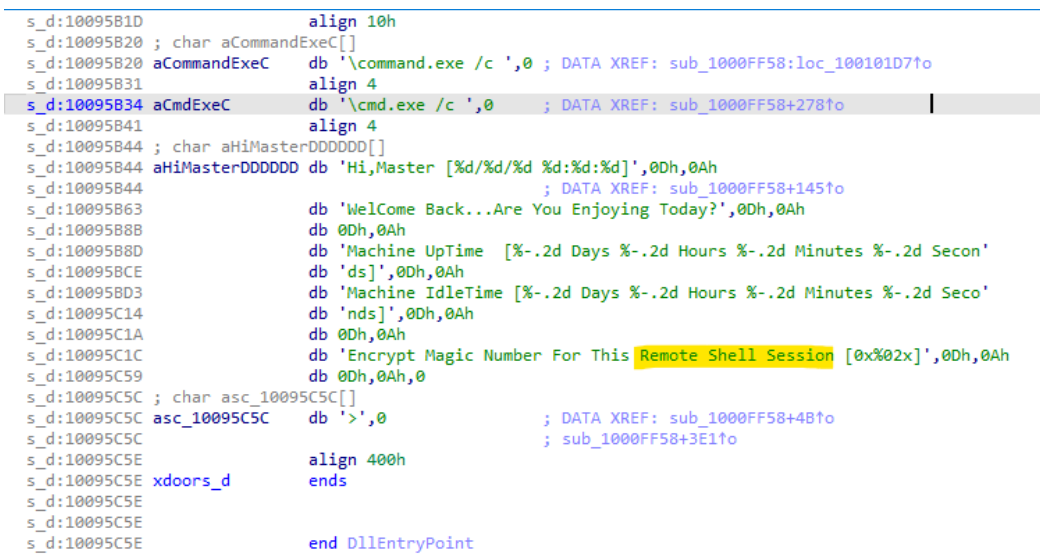


Address	Length	Type	String
xdoors_d:100934B0	0000000A	C	startxcmd
xdoors_d:10095B34	0000000D	C	\\cmd.exe /c

Figure 7: Strings output for cmd.exe

8. **What is happening in the area of code that references cmd.exe /c?**

   By examining the graph view from figure 8. It appears to be creating a remote shell session from the host.



```
s_d:10095B1D                 align 10h
s_d:10095B20 ; char aCommandExeC[]
s_d:10095B20 aCommandExeC     db '\command.exe /c ',0 ; DATA XREF: sub_1000FF58:loc_100101D7↑o
s_d:10095B31                 align 4
s_d:10095B34 aCmdExeC         db '\cmd.exe /c ',0      ; DATA XREF: sub_1000FF58+278↑o
s_d:10095B41                 align 4
s_d:10095B44 ; char aHiMasterDDDDDD[]
s_d:10095B44 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
s_d:10095B44                                         ; DATA XREF: sub_1000FF58+145↑o
s_d:10095B63                 db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
s_d:10095B8B                 db 0Dh,0Ah
s_d:10095B8D                 db 'Machine UpTime  [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
s_d:10095BCE                 db 'ds]',0Dh,0Ah
s_d:10095BD3                 db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
s_d:10095C14                 db 'nds]',0Dh,0Ah
s_d:10095C1A                 db 0Dh,0Ah
s_d:10095C1C                 db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
s_d:10095C59                 db 0Dh,0Ah,0
s_d:10095C5C ; char asc_10095C5C[]
s_d:10095C5C asc_10095C5C     db '>',0                ; DATA XREF: sub_1000FF58+4B↑o
s_d:10095C5C                                         ; sub_1000FF58+3E1↑o
s_d:10095C5E                 align 400h
s_d:10095C5E xdoors_d         ends
s_d:10095C5E
s_d:10095C5E
s_d:10095C5E                 end DllEntryPoint
```

Figure 8: Remote Shell description available

From the description of the code and data strings, it is conclusive that the code is trying to build a **remote shell session**.

9. **In the same area, at 0x100101C8, it looks like dword_1008E5C4 is a global variable that helps decide which path to take. How does the malware set dword_1008E5C4? (Hint: Use dword_1008E5C4's cross-references.)**

   Using Xref, it is easier to check for all the references and based on the figure 9, there is only one **MOV** which is storing some value into dword_1008E5C4.

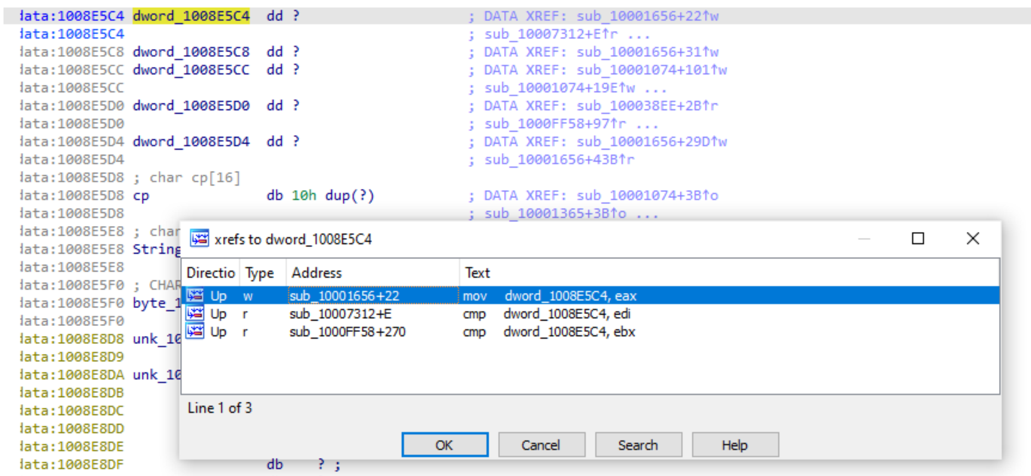


Figure 9: Xref's for dword_1008E5C4

   The value is being moved from **eax**, whose value is being returned based on return type of a subroutine, From figure 10, this subroutine is renamed as **IsWin32NT**.

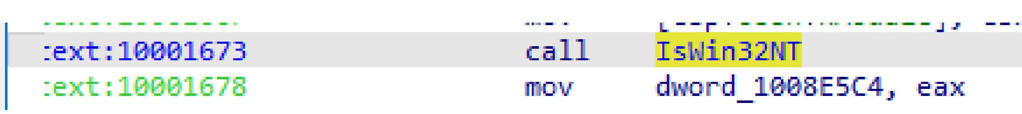


Figure 10: Subroutine renamed as IsWin32NT at text:10001673

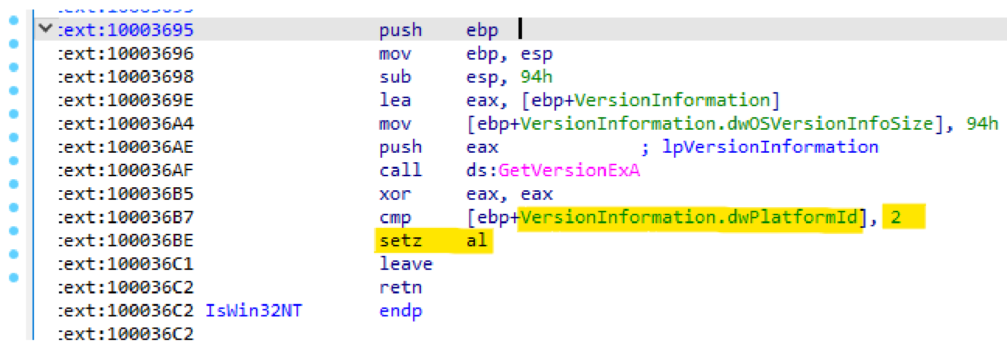
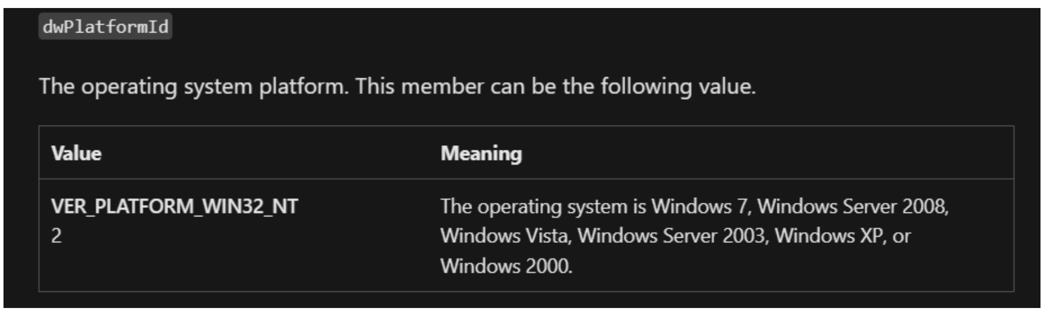


Figure 11: Version Check being done with integer 2



dwPlatformId

The operating system platform. This member can be the following value.

Value	Meaning
VER_PLATFORM_WIN32_NT 2	The operating system is Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, or Windows 2000.

Figure 12: dwPlatformId definition from MSDN

Inside this **IsWin32NT** subroutine from figure 11, there is a version comparison check done between **VersionInformation.dwPlatformId** and 2. Register AL will be set if PlatformId is **VER_PLATFORM_WIN32_NT**.

10. **A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?**

    The remote shell function contains series of memcmp functions. String comparision with robotwork at 0x10010452.

```
:ext:10010444
:ext:10010444 loc_10010444:                            ; CODE XREF: sub_1000FF58+4E0↑j
:ext:10010444                 push    9                ; Size
:ext:10010446                 lea     eax, [ebp+Buf1]
:ext:1001044C                 push    offset aRobotwork ; "robotwork"
:ext:10010451                 push    eax              ; Buf1
:ext:10010452                 call    memcmp
:ext:10010457                 add     esp, 0Ch
:ext:1001045A                 test    eax, eax
:ext:1001045C                 jnz     short loc_10010468
:ext:1001045E                 push    [ebp+s]          ; s
:ext:10010461                 call    sub_100052A2
:ext:10010466                 jmp     short loc_100103F6
:ext:10010468 ; ---------------------------------------------------------------------------
```

Figure 13: robotwork string comparision

The **jnz** will not be done, if string matches with **robotwork** and instead calls **sub_100052A2**.

```
100052DC                 lea     eax, [ebp+phkResult]
100052DF                 push    eax             ; phkResult
100052E0                 push    0F003Fh         ; samDesired
100052E5                 push    0               ; ulOptions
100052E7                 push    offset aSoftwareMicros ; "SOFTWARE\\Microsof
100052EC                 push    80000002h       ; hKey
100052F1                 call    ds:RegOpenKeyExA
100052F7                 test    eax, eax
100052F9                 jz      short loc_10005309
100052FB                 push    [ebp+phkResult] ; hKey
100052FE                 call    ds:RegCloseKey
10005304                 jmp     loc_100053F6
10005309 ; ---------------------------------------------------------------
```

Figure 14: sub_100052A2 subroutine

11. **What does the export PSLIST do?**

The PSLIST export in its location 0x10007025, has a subroutine checks to see if the OS version is Windows Vista/7 or XP/2003/2000.

```
:ext:100036C3 VersionInformation= _OSVERSIONINFOA ptr -94h
:ext:100036C3
:ext:100036C3                 push    ebp
:ext:100036C4                 mov     ebp, esp
:ext:100036C6                 sub     esp, 94h
:ext:100036CC                 lea     eax, [ebp+VersionInformation]
:ext:100036D2                 mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
:ext:100036DC                 push    eax             ; lpVersionInformation
:ext:100036DD                 call    ds:GetVersionExA
:ext:100036E3                 cmp     [ebp+VersionInformation.dwPlatformId], 2
:ext:100036EA                 jnz     short loc_100036FA
:ext:100036EC                 cmp     [ebp+VersionInformation.dwMajorVersion], 5
:ext:100036F3                 jb      short loc_100036FA
:ext:100036F5                 push    1
:ext:100036F7                 pop     eax
:ext:100036F8                 leave
:ext:100036F9                 retn
:ext:100036FA ; ---------------------------------------------------------------------------
:ext:100036FA
:ext:100036FA loc_100036FA:                           ; CODE XREF: sub_100036C3+27↑j
:ext:100036FA                                         ; sub_100036C3+30↑j
:ext:100036FA                 xor     eax, eax
:ext:100036FC                 leave
:ext:100036FD                 retn
:ext:100036FD sub_100036C3    endp
:ext:100036FD
:ext:100036FE
:ext:100036FE ; =============== S U B R O U T I N E =======================================
```

Figure 15: OS Version comparision using dwMajorVersion

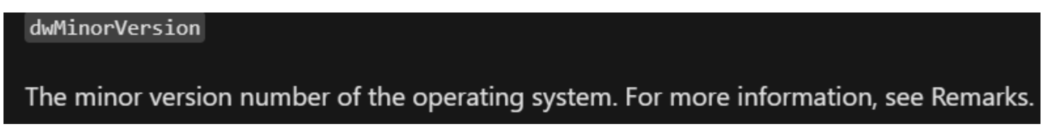


Figure 16: Comparision check done on dwMajorVersion

Operating system	Version number	dwMajorVersion	dwMinorVersion	Other
Windows 10	10.0*	10	0	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2016	10.0*	10	0	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows 8.1	6.3*	6	3	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2012 R2	6.3*	6	3	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows 8	6.2	6	2	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2012	6.2	6	2	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows 7	6.1	6	1	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2008 R2	6.1	6	1	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows Server 2008	6.0	6	0	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows Vista	6.0	6	0	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2003 R2	5.2	5	2	GetSystemMetrics(SM_SERVERR2) != 0
Windows Server 2003	5.2	5	2	GetSystemMetrics(SM_SERVERR2) == 0

Figure 17: Official Documentation table for dwMajorVersion

From figure 15, and 16, it is conclusive that this subroutine is checking if host Operating system is above Windows Server 2003 in table in figure 17.

12. **Use the graph mode to graph the cross-references from sub_10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?**

    Using Xref's from View Tab, by setting start and end address as **sub_10004E79** and setting depth as 1, the following graph is returned from WinGraph32.exe

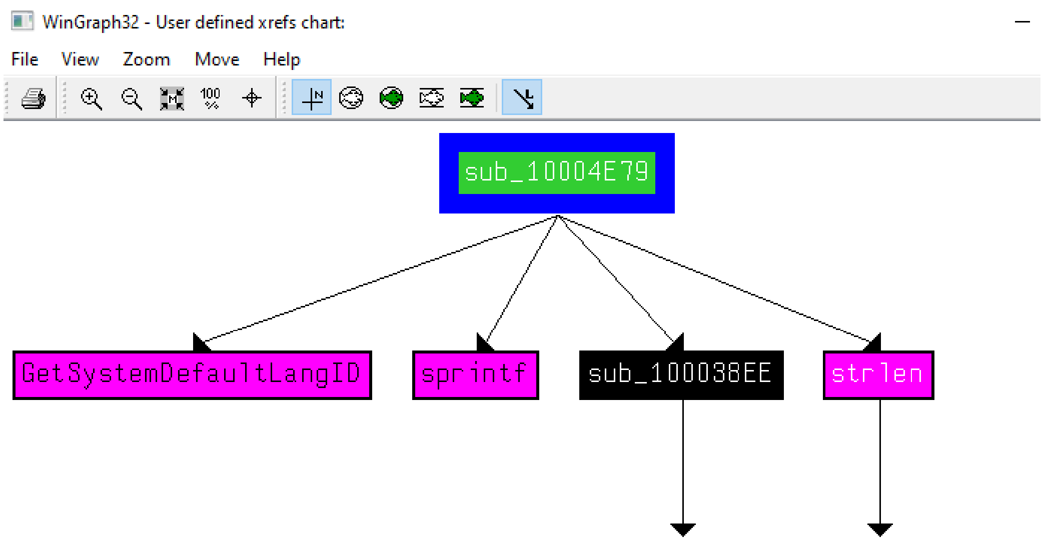


Figure 18: Wingraph32 output for sub_10004E79

GetSystemDefaultLangID, strlen, and sprintf are API calls made from sub_10004E79. This function could be renamed to something useful like GetSystemDefaultLanguage.

13. **How many Windows API functions does DllMain call directly? How many at a depth of 2?**

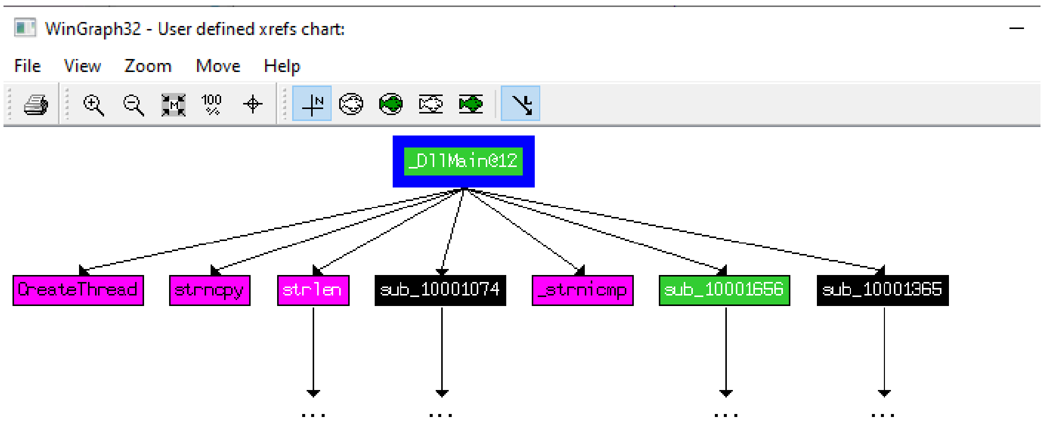


Figure 19: WinGraph.exe returned tree from DLLMain@12

From figure 19, .DllMain calls strncpy, strnicmp, CreateThread, and strlen directly. At a depth of 2, it calls Sleep, WinExec, gethostbyname, and many other networking function calls.

14. **At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?**

    This code makes the program sleep for 30 seconds.

15. **At 0x10001701 is a call to socket. What are the three parameters?**

    The values of arguments sent to socket subroutine call are 6, 1 and 2. The 3 parameters are af, type and protocol based on SOCKET subroutine definition.

```
.text:100016FB                 push    6               ; protocol
.text:100016FD                 push    1               ; type
.text:100016FF                 push    2               ; af
.text:10001701                 call    ds:socket
```

Figure 20: SOCKET subroutine parameters

16. **Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?**

    This can be performed by inserting a enum in Enum Subview and click Insert on keyboard, click **Add standard enum by symbolic name**, search for appropriate value and hit Enter.
    Then select the number, right click and select the appropriate symbolic constant.

```
.text:100016FB                 push    IPPROTO_TCP     ; protocol
.text:100016FD                 push    SOCK_STREAM     ; type
.text:100016FF                 push    AF_INET         ; af
.text:10001701                 call    ds:socket
```

Figure 21: Symbolic Constants from MSDN

17. **Search for usage of the in instruction (opcode 0xED). This instruction is used with a magic string VMXh to perform VMware detection. Is that in use in this malware? Using the cross-references to the function that executes the in instruction, is there further evidence of VMware detection?**

.text:100061C7	sub_10006196	mov eax, 'VMXh'
.text:100061DC	sub_10006196	cmp ebx, 'VMXh'

Figure 22: VMXh search found at 0x100061C7

Instruction has been searched using Search -¿ Text search (slow!). at address 0x100061C0.

```
.text:100061C0 ;    __try { // __except at loc_100061EF
.text:100061C0                  and     [ebp+ms_exc.registration.TryLevel], 0
.text:100061C4                  push    edx
.text:100061C5                  push    ecx
.text:100061C6                  push    ebx
.text:100061C7                  mov     eax, 'VMXh'
.text:100061CC                  mov     ebx, 0
.text:100061D1                  mov     ecx, 0Ah
.text:100061D6                  mov     edx, 'VX'
.text:100061DB                  in      eax, dx
.text:100061DC                  cmp     ebx, 'VMXh'
.text:100061E2                  setz    [ebp+var_1C]
.text:100061E6                  pop     ebx
.text:100061E7                  pop     ecx
.text:100061E8                  pop     edx
.text:100061E9                  jmp     short loc_100061F6
```

Figure 23: VMXh instruction found at 0x100061C7

Using Xref's to find the function that executes leads to output as shown in figure 24.

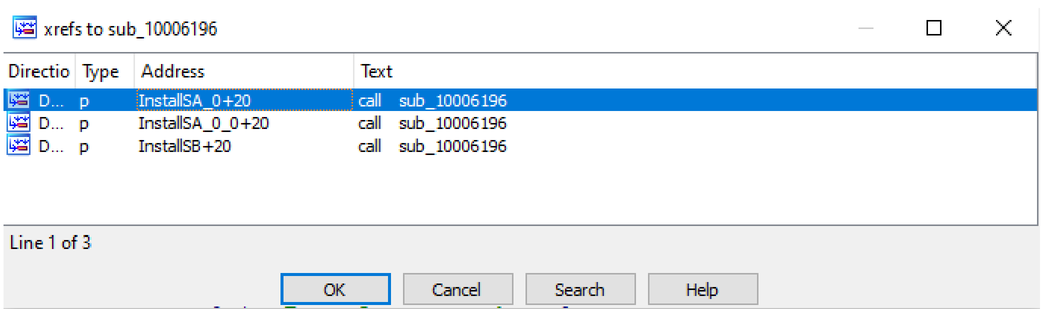


Figure 24: Xref from 0x100061C0

18. **Jump your cursor to 0x1001D988. What do you find?**

    At 0x1001D998, there is a data defined of double size

```
.data:1001D997                  db  34h ; 4
.data:1001D998                  db  36h ; 6
.data:1001D999                  db  3Eh ; >
```

Figure 25: Address 0x1001D998

19. **If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?**

    After executing the script while cursor is at 0x1001D998, the result is shown in figure below.

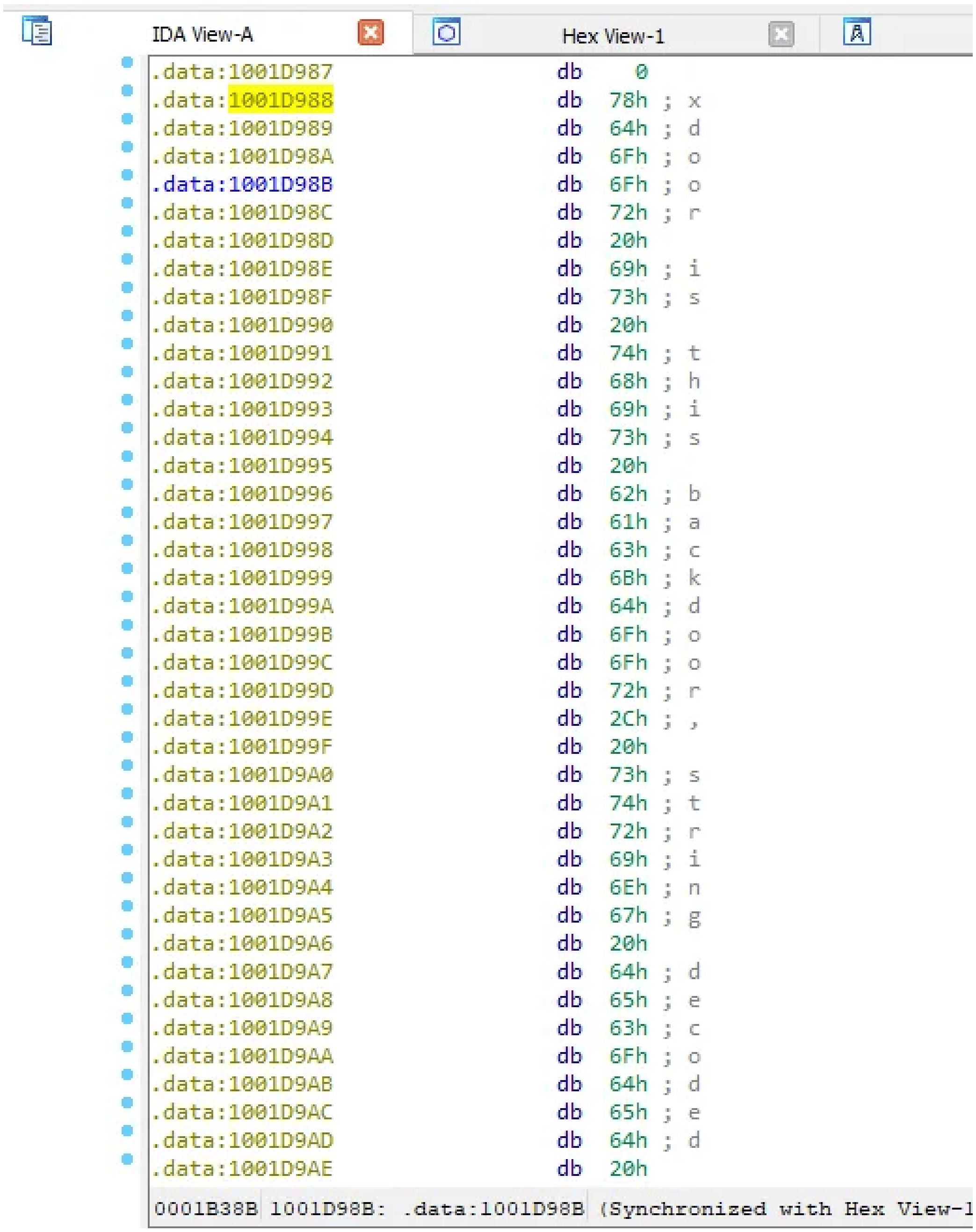


Figure 26: Readable Strings after Python Script execution

20. **With the cursor in the same location, how do you turn this data into a single ASCII string?**

    Pressing key “A” (uppercase or lowercase) converts to ASCII text.

21. **Open the script with a text editor. How does it work?**

    Data iterated from 0 to 0x50 and byte is XORed with value 0x55.

2.2 Analyze the malware found in the file Lab06-01.exe.

1. **What is the major code construct found in the only subroutine called by main?**

   IF Statement.

2. **What is the subroutine located at 0x40105F?**

   printf

3. **What is the purpose of this program?**

   The malware constantly if there is a network connection then connect to the internet then output success but if there is not a connection it will output error.

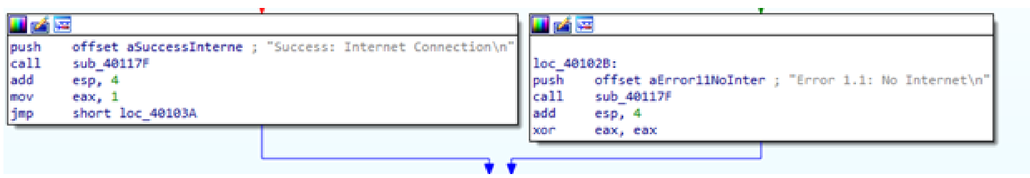


Figure 27: Connection Check

2.3 Analyze the malware found in the file Lab06-02.exe

1. **What operation does the first subroutine called by the main perform?**

   If statement

2. **What is the subroutine located at 0x40117F?**

   printf

3. **What does the second subroutine called by the main do?**

   It downloads a file from the practical malware analysis page.

```
.text:00401051                 push    offset szAgent  ; "Internet Explorer 7.5/pma"
.text:00401056                 call    ds:InternetOpenA
.text:0040105C                 mov     [ebp-0Ch], eax
.text:0040105F                 push    0               ; dwContext
.text:00401061                 push    0               ; dwFlags
.text:00401063                 push    0               ; dwHeadersLength
.text:00401065                 push    0               ; lpszHeaders
.text:00401067                 push    offset szUrl    ; "http://www.practicalmalwareanalysis.com"...
.text:0040106C                 mov     eax, [ebp-0Ch]
.text:0040106F                 push    eax             ; hInternet
.text:00401070                 call    ds:InternetOpenUrlA
.text:00401076                 mov     [ebp-10h], eax
.text:00401079                 cmp     dword ptr [ebp-10h], 0
.text:0040107D                 jnz     short loc_40109D
.text:0040107F                 push    4070A8h         ; "Error 2.1: Fail to OpenUrl\n"
.text:00401084                 call    sub_40117F
.text:00401089                 add     esp, 4
.text:0040108C                 mov     ecx, [ebp-0Ch]
.text:0040108F                 push    ecx             ; hInternet
.text:00401090                 call    ds:InternetCloseHandle
.text:00401096                 xor     al, al
```

Figure 28: Trying to open a URL

4. **What type of code construct is used in this subroutine?**

   InternetReadFile

```
.text:004010AD                 mov     ecx, [ebp-10h]
.text:004010B0                 push    ecx             ; hFile
.text:004010B1                 call    ds:InternetReadFile
.text:004010B7                 mov     [ebp-4], eax
.text:004010BA                 cmp     dword ptr [ebp-4], 0
```

Figure 29: InternetReadFile

5. **Are there any network-based indicators for this program?**

   Yes, there are 2 network-based indicators.

6. **What is the purpose of this malware?**

   This program looks for a network connection then downloads files from a targeted website if an internet connection if found. This can be used in real life by attackers to download malware onto a targeted device and run it.

2.4 Analyze the malware found in Lab11-02.dll. Assume that a suspicious file named Lab11-02.ini was also found with this malware:

1. **What are the exports for this DLL malware?**

   There is only one export for this DLL malware, It is **installer**.

2. **What happens after you attempt to install this malware using rundll32.exe?**

   The malware is creating file Lab11-02.ini. It is trying to open files with read permission.

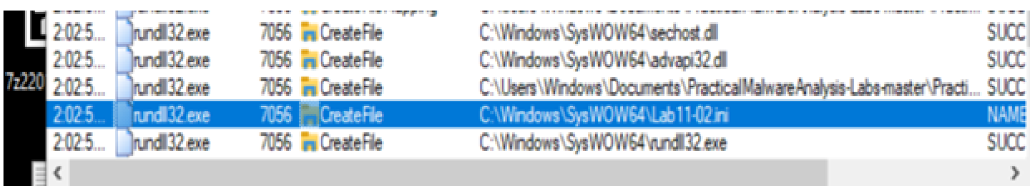


Figure 30: Lab11-02.ini file created by malware

It is also setting a registry key value at **HKLM\SOFTWARE\WOW6432Node\Microsoft \Windows NT \CurrentVersion\Windows\AppInit_DLLs**

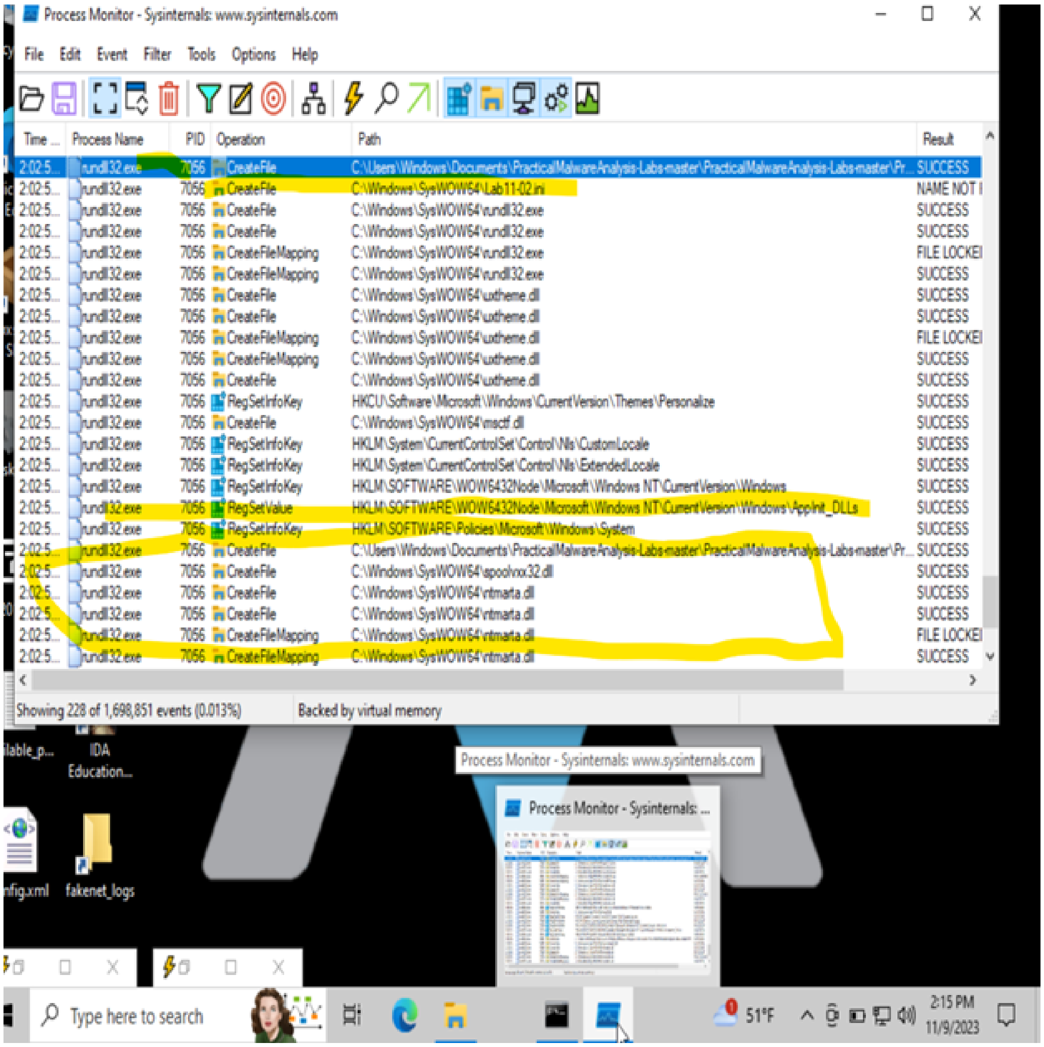


Figure 31: AppInit_DLL registry key set

After setting Registry Key values at AppInit_DLL's, they malware is creating a file spoolvxx32.dll at System32 directory.

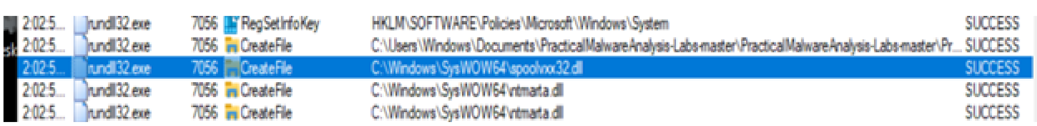


Figure 32: spoolvxx32.dll file created

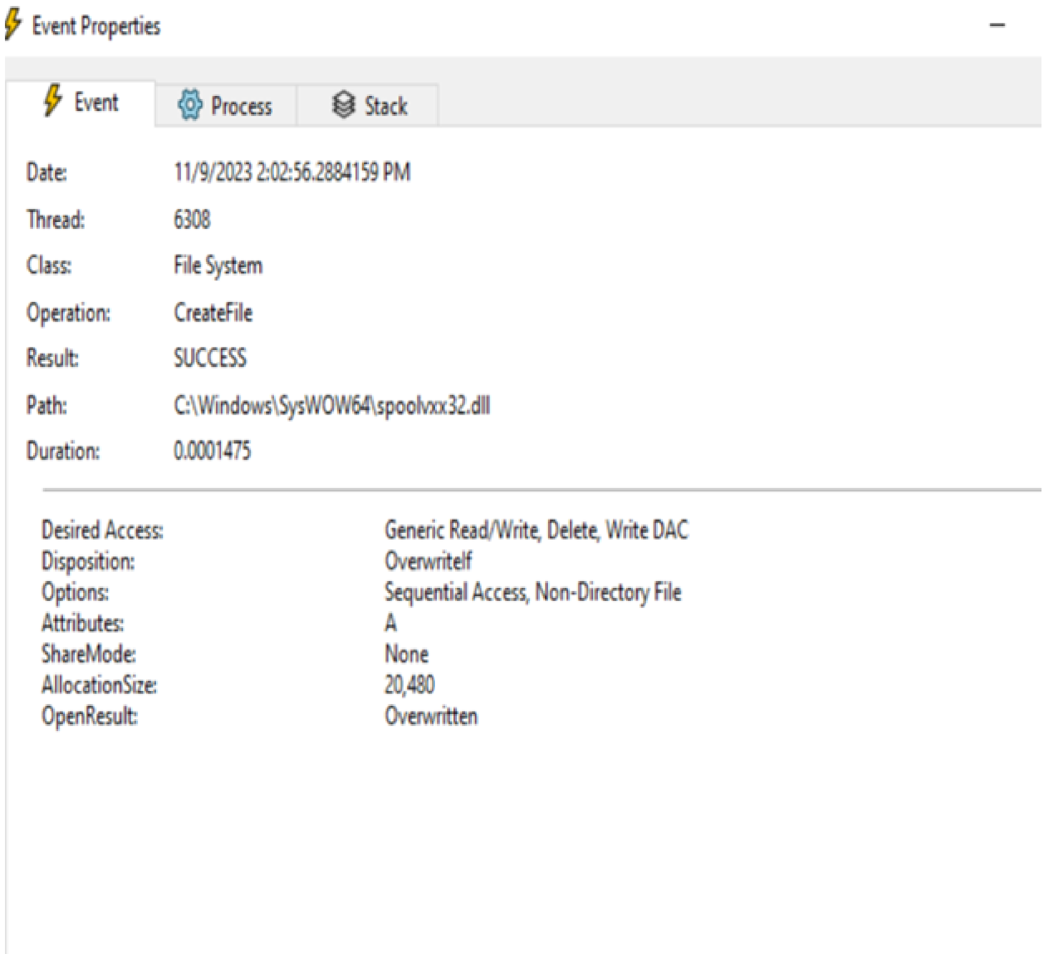


Figure 33: spoolvxx32.dll create event details

3. **Where must Lab11-02.ini reside in order for the malware to install properly?**

   Lab11-02.ini must reside in %SystemAdminName%\System32\in order for the malware to run properly with Administrator Privileges. Usually, by default this directory will be in Windows directory like shown below **C:\Windows\System32\Lab11-02.ini**

4. **How is this malware installed for persistence?**

   Based on the answer of previous question, the sets Registry Key AppInit_DLL as its persistance mechanism.

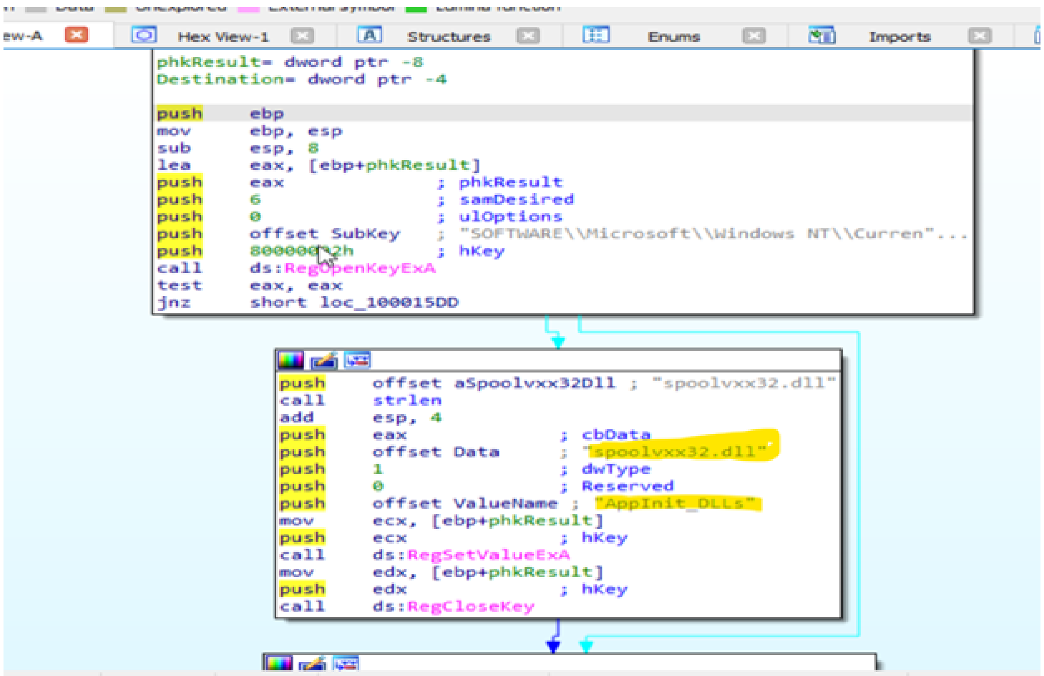


Figure 34: Persistence set using AppInit_DLL Key in Registry

5. **What user-space rootkit technique does this malware employ?**

   From figure **??**, it is conclusive that the malware is using Inline hooking technique.

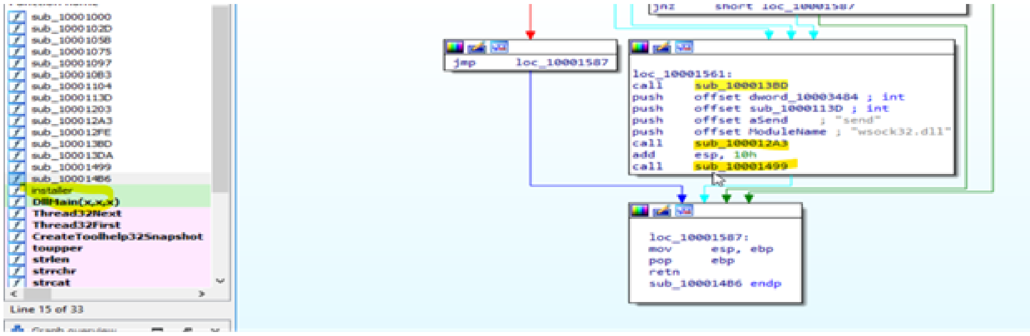


Figure 35: wsock32.dll offset used in hooking

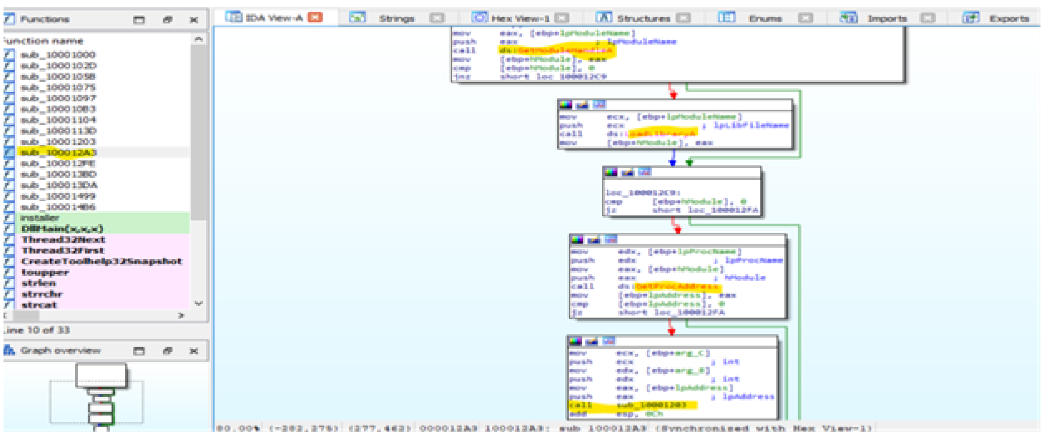


Figure 36: GetModuleA, LoadLibraryA, GetProcAddress API's used

6. **What does the hooking code do?**

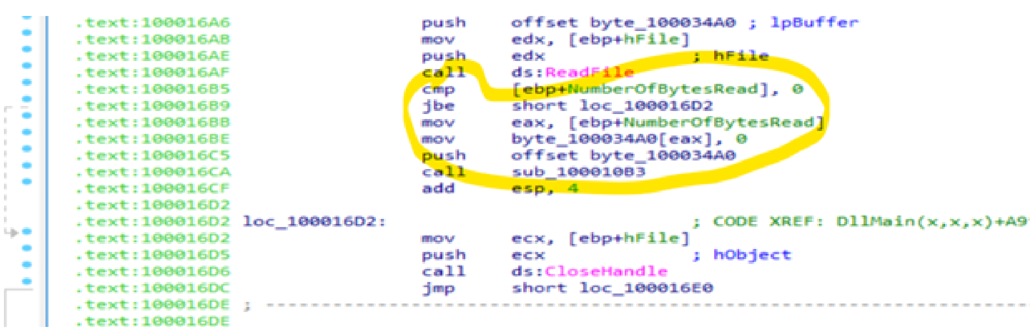


Figure 37: Hooking Code

7. **Which process(es) does this malware attack and why?**

   Based on question 5 and 6 it is attacking MSIMN.exe and THEBAT.exe and OUTLOOK.exe because they are are email clients and the malware purpose was made hook those API

8. **What is the significance of the .ini file?**

   The .ini file is encoded and when it is decoded it is used to specify an email. this email can be added when sending an email to any recipient.After decrypting Lab11-02.ini, we see it contains billy@malwareanalysisbook.com.

9. **How can you dynamically capture this malware's activity with Wireshark?**

   It is possible to dynamically capture network traffic of this malware using Wireshark on the network interface through which the default traffic is directed at.

   For this, select the interface and capture packets in real-time. While at it, execute the malware sample and stop the packet capture once the executable is successfully run and the process is exited.

   While Fakenet is running in background to divert the traffic, wireshark enables to see the raw packets transmitted in real-time.

3 Conclusion

In summary, advanced static analysis provides in-depth understanding of malware code, while advanced dynamic analysis reveals the full scope of its behaviors. Combining these techniques allows comprehensive analysis to dissect functionality, determine capabilities, and assess threat potential.

Using IDA Pro to it's full extent opens windows to understand complex malware better, using exisiting toolkits and integrations with Python enable the whole reverse engineering process, Disassembly and De-compilation automated, thereby reducing workload of analyst without missing any minute details that can be found during the process.