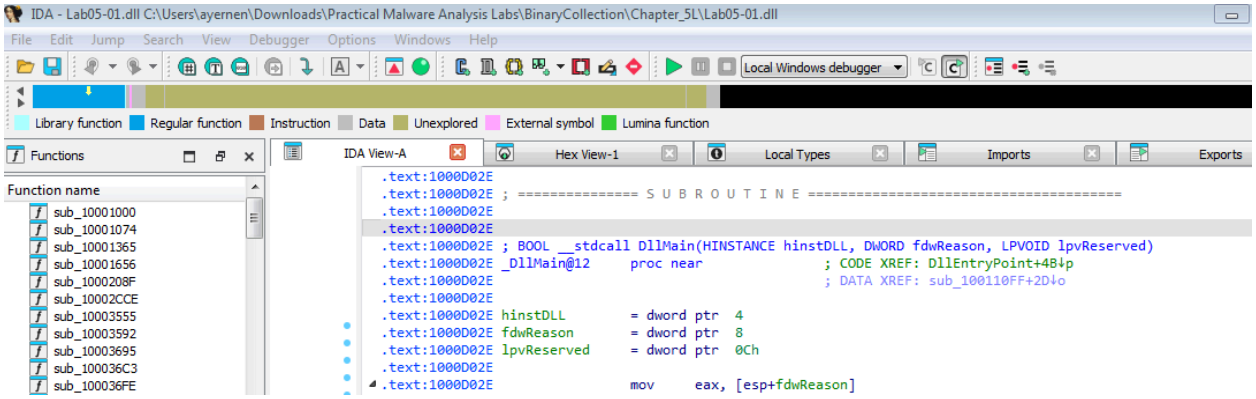
Anvitha Yerneni
ayernen@g.clemson.edu

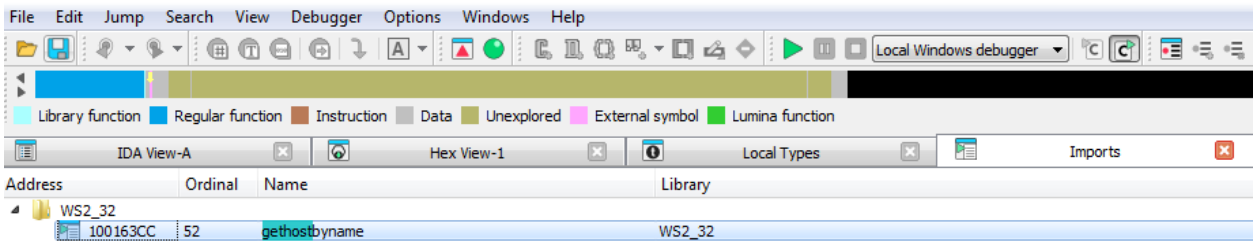
Malware Reverse Engineering - MidTerm

Question1. LAB 5-1

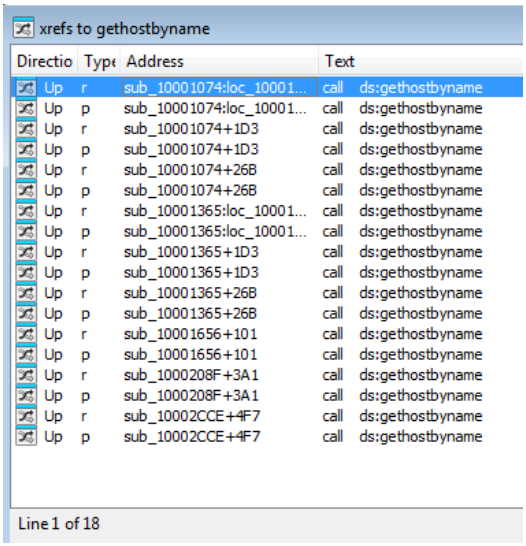
1) 1000D02E is the address of dll



2) It is located at 100163CC



3) There are a total of 18 function calls. In which there are 6 common with 10001074 address.6 common with 10001365 address. 2 with 10001656. 2 with 1000208F and 2 with 10002CC. Therefore 5 function calls are said to be unique.



Anvitha Yerneni
ayernen@g.clemson.edu

4) gethostbyname has only 1 parameter called name and it is going to pull the parameter off of the top of stack. Through reverse engineering we find it comes from the offset off_10019040.

```
.text:10001748                jnz     loc_100017ED
.text:1000174E                mov     eax, off_10019040 ; "[This is RDO]pics.praticalmalwareanalys"...
.text:10001753                add     eax, 0Dh
.text:10001756                push    eax             ; name
.text:10001757                call    ds:gethostbyname
```

And the offset makes an DNS request to pic.practicalmalwareanalysis.com

```
.data:10019040 off_10019040    dd offset aThisIsRdoPicsP
.data:10019040                                         ; DATA XREF: sub_10001656:loc_10001722↑r
.data:10019040                                         ; sub_10001656+F8↑r ...
.data:10019040                                         ; "[This is RDO]pics.praticalmalwareanalys"...
```

5) When examining the subroutine at the address 0x10001656 using IDA, it's observed that there are 23 local variables identified. This determination comes from noticing that these variables have a negative offset relative to the EBP register, indicating their local scope.

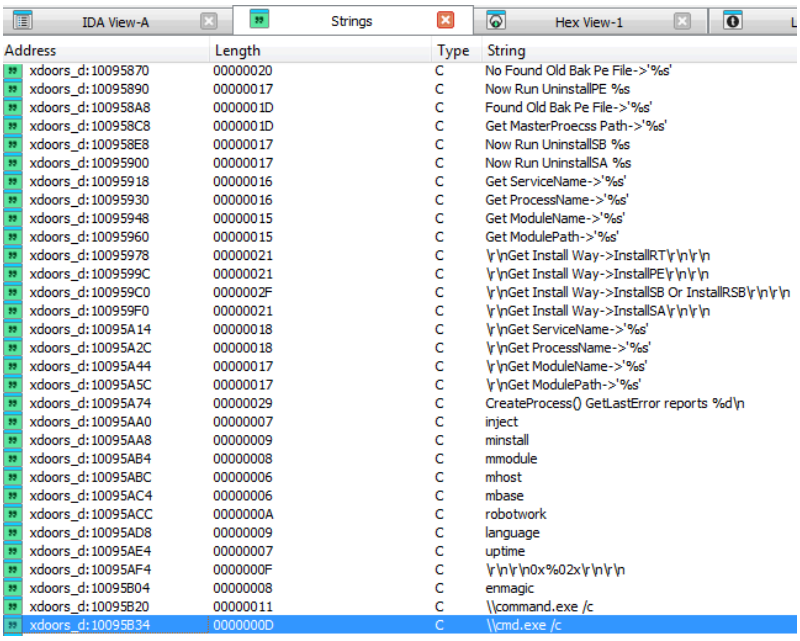
```
.text:10001656
.text:10001656 ; =============== S U B R O U T I N E =======================================
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656 sub_10001656    proc near               ; DATA XREF: DllMain(x,x,x)+C8↓o
.text:10001656
.text:10001656 var_675         = byte ptr -675h
.text:10001656 var_674         = dword ptr -674h
.text:10001656 hModule         = dword ptr -670h
.text:10001656 timeout         = timeval ptr -66Ch
.text:10001656 name            = sockaddr ptr -664h
.text:10001656 var_654         = word ptr -654h
.text:10001656 in              = in_addr ptr -650h
.text:10001656 Str1            = byte ptr -644h
.text:10001656 var_640         = byte ptr -640h
.text:10001656 CommandLine     = byte ptr -63Fh
.text:10001656 Str             = byte ptr -63Dh
.text:10001656 var_638         = byte ptr -638h
.text:10001656 var_637         = byte ptr -637h
.text:10001656 var_544         = byte ptr -544h
.text:10001656 var_50C         = dword ptr -50Ch
.text:10001656 var_500         = byte ptr -500h
.text:10001656 Buf2            = byte ptr -4FCh
.text:10001656 readfds         = fd_set ptr -4BCh
.text:10001656 buf             = byte ptr -3B8h
.text:10001656 var_3B0         = dword ptr -3B0h
.text:10001656 var_1A4         = dword ptr -1A4h
.text:10001656 var_194         = dword ptr -194h
.text:10001656 WSAData         = WSAData ptr -190h
.text:10001656 lpThreadParameter= dword ptr  4
.text:10001656
.text:10001656                 sub     esp, 678h
```

6) The presence of a positive offset in relation to EBP indicates that "lpThreadParameter" is functioning as a parameter.

Anvitha Yerneni
ayernen@g.clemson.edu

```
.text:10001656
.text:10001656 ; =============== S U B R O U T I N E ========
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThread
.text:10001656 sub_10001656    proc near               ; DATA
.text:10001656
.text:10001656 var_675         = byte ptr -675h
.text:10001656 var_674         = dword ptr -674h
.text:10001656 hModule         = dword ptr -670h
.text:10001656 timeout         = timeval ptr -66Ch
.text:10001656 name            = sockaddr ptr -664h
.text:10001656 var_654         = word ptr -654h
.text:10001656 in              = in_addr ptr -650h
.text:10001656 Str1            = byte ptr -644h
.text:10001656 var_640         = byte ptr -640h
.text:10001656 CommandLine     = byte ptr -63Fh
.text:10001656 Str             = byte ptr -63Dh
.text:10001656 var_638         = byte ptr -638h
.text:10001656 var_637         = byte ptr -637h
.text:10001656 var_544         = byte ptr -544h
.text:10001656 var_50C         = dword ptr -50Ch
.text:10001656 var_500         = byte ptr -500h
.text:10001656 Buf2            = byte ptr -4FCh
.text:10001656 readfds         = fd_set ptr -4BCh
.text:10001656 buf             = byte ptr -3B8h
.text:10001656 var_3B0         = dword ptr -3B0h
.text:10001656 var_1A4         = dword ptr -1A4h
.text:10001656 var_194         = dword ptr -194h
.text:10001656 WSAData         = WSAData ptr -190h
.text:10001656 lpThreadParameter= dword ptr  4
.text:10001656
.text:10001656                 sub     esp, 678h
```

7) It is located at 10095B34



IDA View-A | Strings | Hex View-1

Address	Length	Type	String
xdoors_d:10095870	00000020	C	No Found Old Bak Pe File->'%s'
xdoors_d:10095890	00000017	C	Now Run UninstallPE %s
xdoors_d:100958A8	0000001D	C	Found Old Bak Pe File->'%s'
xdoors_d:100958C8	0000001D	C	Get MasterProecss Path->'%s'
xdoors_d:100958E8	00000017	C	Now Run UninstallSB %s
xdoors_d:10095900	00000017	C	Now Run UninstallSA %s
xdoors_d:10095918	00000016	C	Get ServiceName->'%s'
xdoors_d:10095930	00000016	C	Get ProcessName->'%s'
xdoors_d:10095948	00000015	C	Get ModuleName->'%s'
xdoors_d:10095960	00000015	C	Get ModulePath->'%s'
xdoors_d:10095978	00000021	C	\r\nGet Install Way->InstallRT\r\n\r\n
xdoors_d:1009599C	00000021	C	\r\nGet Install Way->InstallPE\r\n\r\n
xdoors_d:100959C0	0000002F	C	\r\nGet Install Way->InstallSB Or InstallRSB\r\n\r\n
xdoors_d:100959F0	00000021	C	\r\nGet Install Way->InstallSA\r\n\r\n
xdoors_d:10095A14	00000018	C	\r\nGet ServiceName->'%s'
xdoors_d:10095A2C	00000018	C	\r\nGet ProcessName->'%s'
xdoors_d:10095A44	00000017	C	\r\nGet ModuleName->'%s'
xdoors_d:10095A5C	00000017	C	\r\nGet ModulePath->'%s'
xdoors_d:10095A74	00000029	C	CreateProcess() GetLastError reports %d\n
xdoors_d:10095AA0	00000007	C	inject
xdoors_d:10095AA8	00000009	C	minstall
xdoors_d:10095AB4	00000008	C	mmodule
xdoors_d:10095ABC	00000006	C	mhost
xdoors_d:10095AC4	00000006	C	mbase
xdoors_d:10095ACC	0000000A	C	robotwork
xdoors_d:10095AD8	00000009	C	language
xdoors_d:10095AE4	00000007	C	uptime
xdoors_d:10095AF4	0000000F	C	\r\n\r\n0x%02x\r\n\r\n
xdoors_d:10095B04	00000008	C	enmagic
xdoors_d:10095B20	00000011	C	\\command.exe /c
xdoors_d:10095B34	0000000D	C	\\cmd.exe /c

Anvitha Yerneni
ayernen@g.clemson.edu

8) The string in question is exclusively referenced within a single function, which can be found at the memory address 1000FF58.

Address	Function	Instruction
.text:100101D0	sub_1000FF58	push offset aCmdExeC ; "\\cmd.exe /c "

In this code segment, an offset value associated with a string is being placed onto the stack.

```
.text:100101CE                  jz      short loc_100101D7
.text:100101D0                  push    offset aCmdExeC ; "\\cmd.exe /c "
.text:100101D5                  jmp     short loc_100101DC
```

I then examined the graph representation of the function to gain an overall understanding of its behavior. It's not feasible to share the entire graph here, but I did come across several strings during my review.

- closesocket
- minstall
- robotwork
- cd
- mhost
- inject
- svchost.exe
- Get Install Way
- xinstall.log
- Detect VM
- Inject To Process Sucessfully
- Robot_Worktime
- Machine IdleTime:

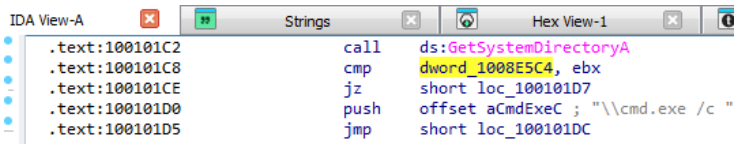
Then i found this:

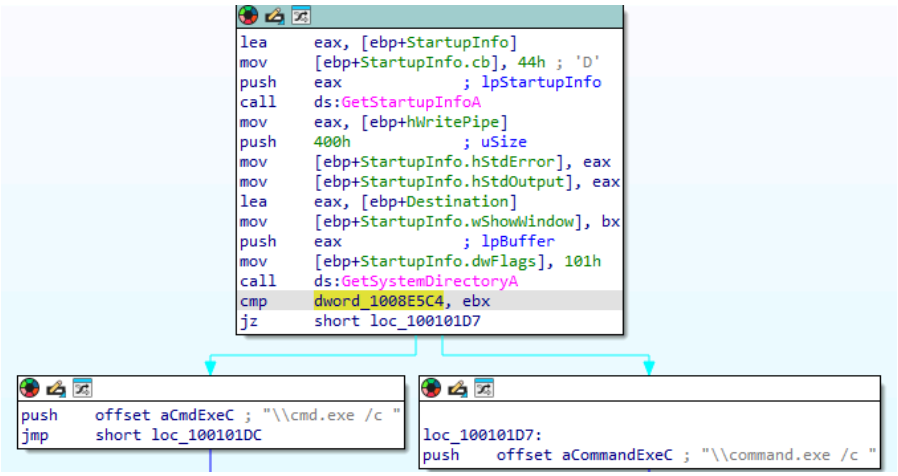
```
aHiMasterDDDDDD: db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
db 'WelCome Back...Are You Enjoying Tod'
db 'ay?',0Dh,0Ah
db 0Dh,0Ah
db 'Machine UpTime  [%-.2d Days %-.2d H'
db 'ours %-.2d Minutes %-.2d Seconds]',0Dh
db 0Ah
db 'Machine IdleTime [%-.2d Days %-.2d '
db 'Hours %-.2d Minutes %-.2d Seconds]',0Dh
db 0Ah
db 0Dh,0Ah
db 'Encrypt Magic Number For This Remot'
db 'e Shell Session [0x%02x]',0Dh,0Ah
db 0Dh,0Ah,0
```

Now it's understood that the code which points to the specific string is in charge of establishing a remote access shell.

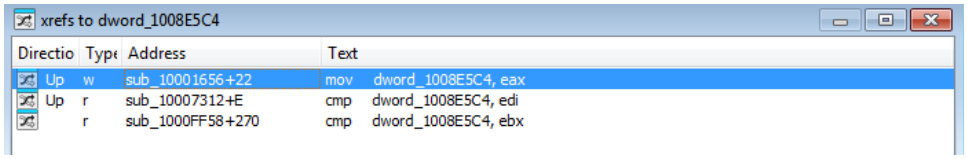
Anvitha Yerneni
ayernen@g.clemson.edu

9) At address 100101C8, the value stored in the ebx register is checked against the value at memory location dword_1008E5C4. If these values match, the memory address pointing to the string "\cmd.exe /c" is placed on the stack. Conversely, if there's no match, the memory address for the string "\command.exe /c" is added to the stack instead.

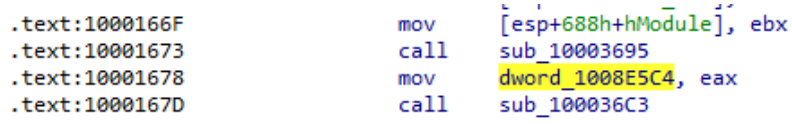




When examining the cross-references to dword_1008E5C4, the initial entry is particularly noteworthy.



The MOV command transfers the data held in the EAX register to the memory location at dword_1008E5C4. We'll proceed to this command to discover what additional information is available.

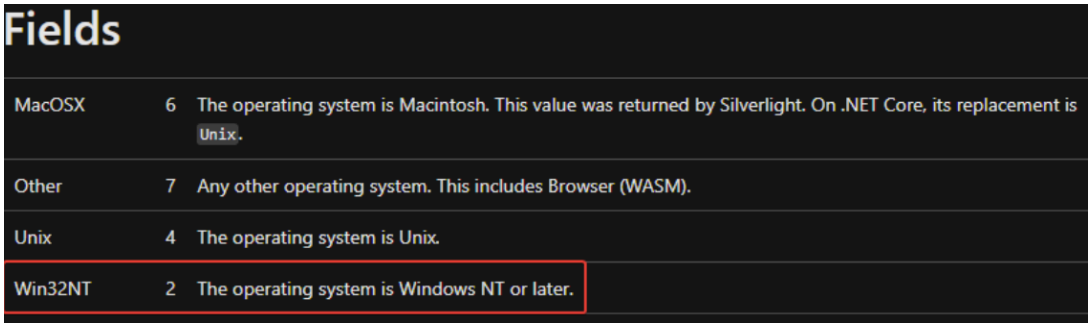


Anvitha Yerneni
ayernen@g.clemson.edu

To gain a better understanding, let's examine the instruction "call sub_10003695" that precedes the "mov" instruction, as it might provide some insight into what is being assigned to the eax register.

```
.text:10003695
.text:10003695 ; =============== S U B R O U T I N E =======================================
.text:10003695
.text:10003695 ; Attributes: bp-based frame
.text:10003695
.text:10003695 sub_10003695    proc near               ; CODE XREF: sub_10001656+1D↑p
.text:10003695                                         ; sub_10003B75+7↓p ...
.text:10003695
.text:10003695 VersionInformation= _OSVERSIONINFOA ptr -94h
.text:10003695
.text:10003695                 push    ebp
.text:10003696                 mov     ebp, esp
.text:10003698                 sub     esp, 94h
.text:1000369E                 lea     eax, [ebp+VersionInformation]
.text:100036A4                 mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036AE                 push    eax             ; lpVersionInformation
.text:100036AF                 call    ds:GetVersionExA
.text:100036B5                 xor     eax, eax
.text:100036B7                 cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:100036BE                 setz    al
.text:100036C1                 leave
.text:100036C2                 retn
.text:100036C2 sub_10003695    endp
```

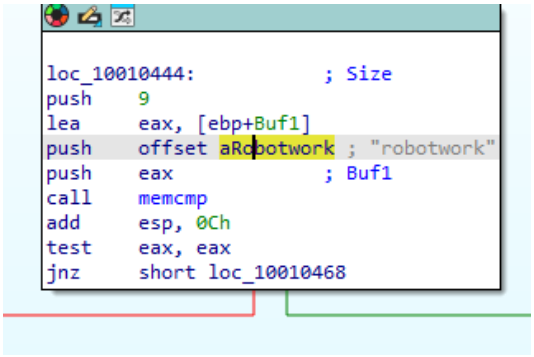
This function utilizes the OSVERSIONINFOA structure from the Win32 API to determine the operating system version of the target. It performs a comparison with a PlatformID value of 2. Upon reviewing the significance of PlatformID values, it's evident that the function is verifying whether the operating system version is at least Windows NT or more recent.



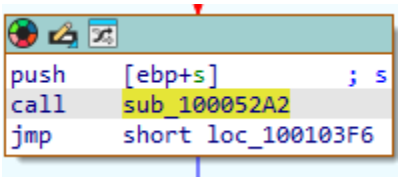
Fields		
MacOSX	6	The operating system is Macintosh. This value was returned by Silverlight. On .NET Core, its replacement is `Unix`.
Other	7	Any other operating system. This includes Browser (WASM).
Unix	4	The operating system is Unix.
Win32NT	2	The operating system is Windows NT or later.

10) When the string comparison with "robotwork" is successful, leading memcmp to return 0, the program will not execute the conditional jump at address 100145C (which occurs if the result is non-zero). Consequently, we proceed along the path indicated by the red arrow.

Anvitha Yerneni
ayernen@g.clemson.edu



The red arrow points us towards a newly identified function, situated at the memory address 100052A2.

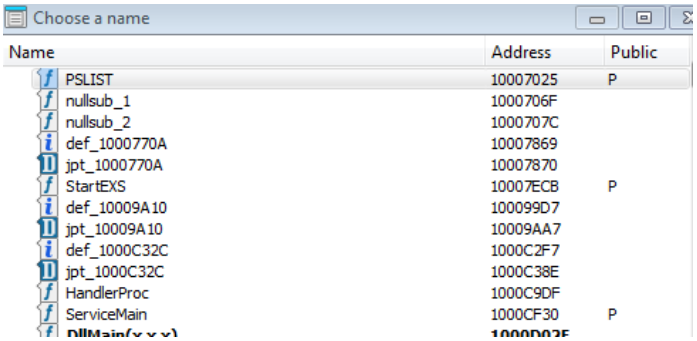


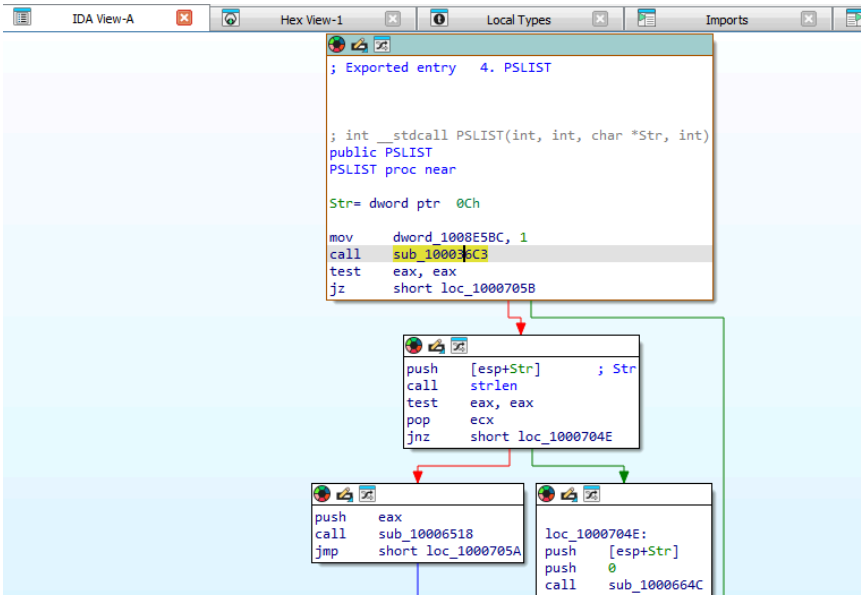
When examining this function, it appears to be interacting with registry entries located under the path SOFTWARE\Microsoft\Windows\CurrentVersion.

Upon reviewing the graph of the function, it's observed that the function retrieves data from the registry paths SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTimes and SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime.

Anvitha Yerneni
ayernen@g.clemson.edu

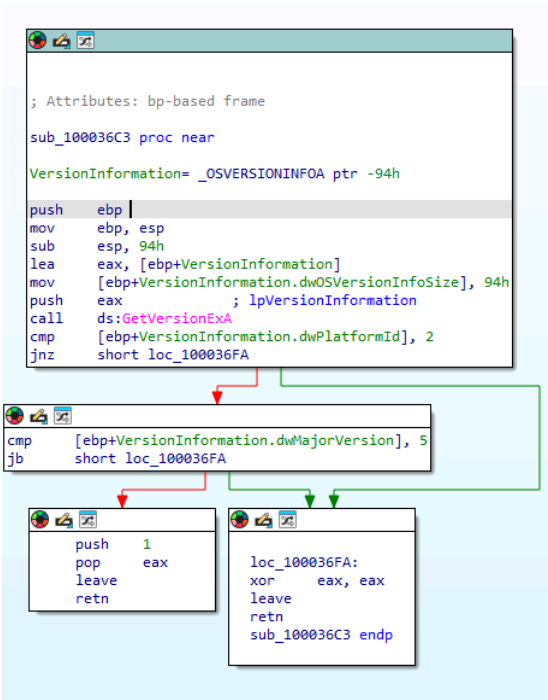
11) The PSLIST function is situated at the memory address 10007025. Upon navigating to this location, it's observed that the flow of execution diverges based on the outcome produced by the function located at sub_100036C3.



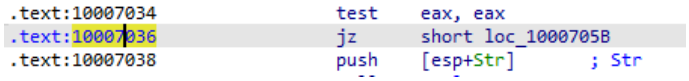


Upon examining the `sub_100036C3` function, it's evident that the initial sequence of instructions bears a striking resemblance to a function we explored previously in question 9. However, to describe it uniquely, we notice that the foundational operations and their progression in this section mirror those found in the earlier mentioned function, displaying a similar pattern in execution and logic flow, albeit within a different context or application. This observation suggests a potential reuse or adaptation of certain algorithmic strategies or code snippets, underlining a consistency in approach or methodology between the two instances.

Anvitha Yerneni
ayernen@g.clemson.edu



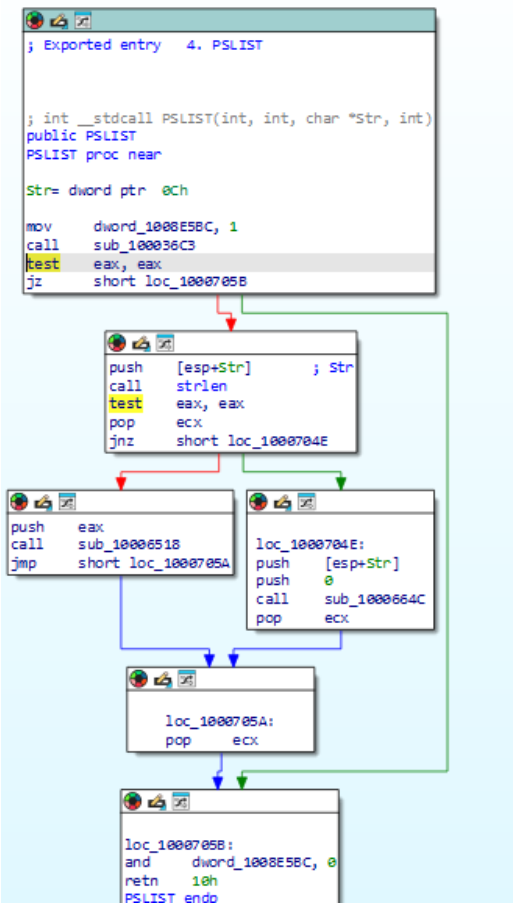
When analyzing the `sub_100036C3` function, it becomes clear that the beginning set of operations closely parallels a function discussed earlier in question 9. However, distinguishing this particular instance, it is observed that the core processes and their sequence share resemblances to the previously discussed function, indicating a similar method of execution and logical progression, yet applied in a distinct scenario or purpose. This insight points to the possibility of leveraging or modifying specific algorithms or code fragments, highlighting a uniformity in strategy or technique across both examples.



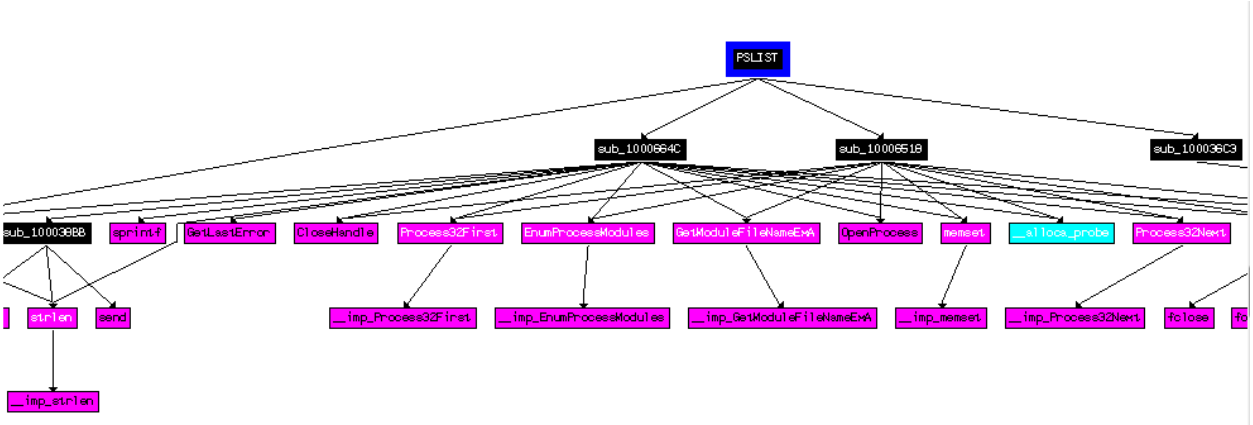
After completing the execution of sub_100036C3, control is transferred back to the PSLIST function, located at the memory address 10007034.

The "test eax, eax" instruction checks if the eax register is 0, setting the Zero Flag (ZF) accordingly. Following a prior function, if the OS is newer than Windows 2000 (dwMajorVersion ≥ 5), eax is 1; if older, eax is 0. A subsequent Jump if Zero at 1007036 leads to 1000705B for older OSes, showing no notable action. For newer OSes, execution might split to either sub_10006518 or sub_1000664C.

Anvitha Yerneni
ayernen@g.clemson.edu

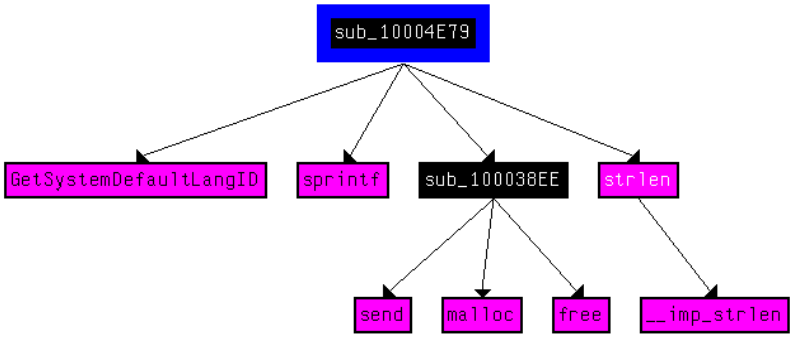


These snippets appear to describe functions designed to enumerate the active processes on a target computer. The function graphs provided give an indication of this functionality, although the full graphs are too extensive to share here.

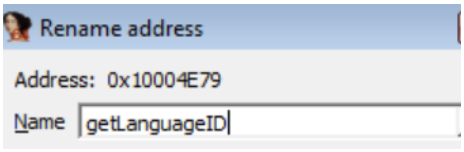


Anvitha Yerneni
ayernen@g.clemson.edu

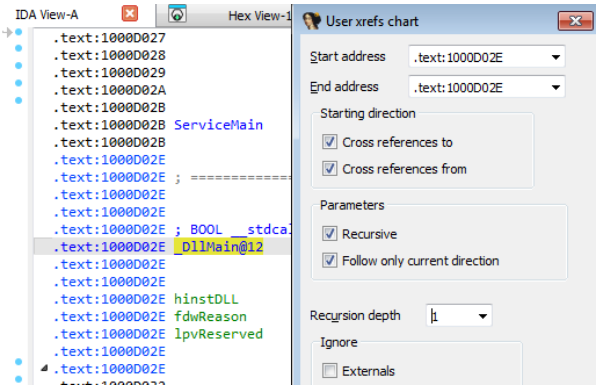
12) The chart below showcases various API functions, with GetSystemDefaultLangID standing out as particularly noteworthy. This function appears to transmit the language ID of the target to the operator of the malware.



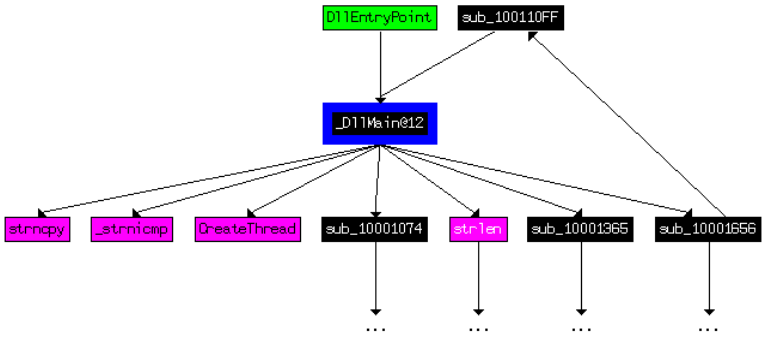
It can be renamed as get_LanguageID.



13) I constructed a graph to identify the Windows API functions directly invoked by DllMain, using a recursion depth of one.



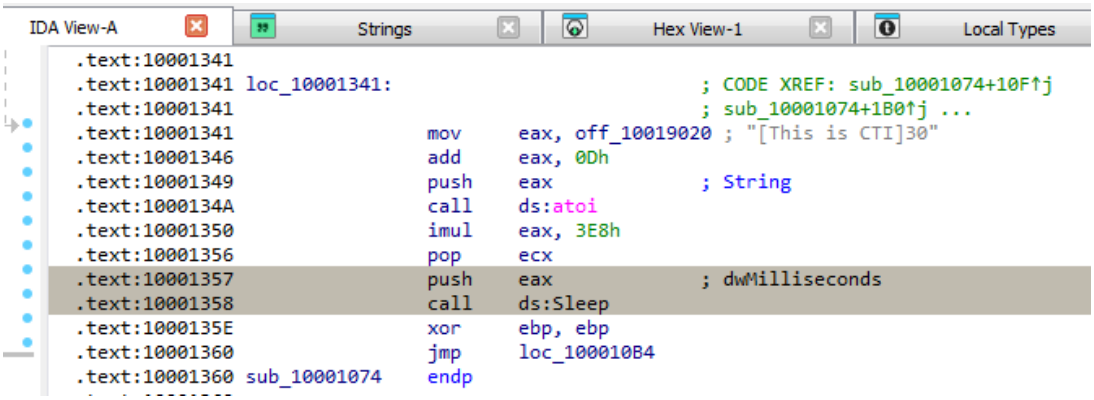
There are four API functions directly invoked by DllMain:



Anvitha Yerneni
ayernen@g.clemson.edu

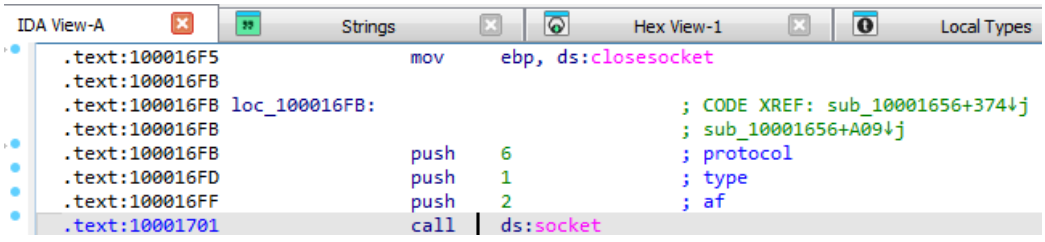
When the recursion depth reaches 2, the complexity of the graph significantly increases, with a total of 33 API functions engaged. Among these, noteworthy functions include Sleep, gethostbyname, closesocket, WinExec, send, recv, and socket.

14) Tracing the code in reverse order, it is observed that the duration for the sleep, expressed in milliseconds, is transferred into the eax register right before the API function is invoked:



In the loc_10001341 section, the code initially assigns the string [This is CTI]30 to the EAX register, then adds 13 to EAX, making it point to the "30" part of the string. This "30" is pushed onto the stack and converted to the integer 30 via the a toi function. The value is then multiplied by 1000, resulting in 30000, which is pushed onto the stack again. Finally, the program calls the Sleep function with this value, causing it to pause for 30000 milliseconds, or 30 seconds.In the loc_10001341 section, the code initially assigns the string [This is CTI]30 to the EAX register, then adds 13 to EAX, making it point to the "30" part of the string. This "30" is pushed onto the stack and converted to the integer 30 via the atoi function. The value is then multiplied by 1000, resulting in 30000, which is pushed onto the stack again. Finally, the program calls the Sleep function with this value, causing it to pause for 30000 milliseconds, or 30 seconds.

15) 3 parameters are 6, 1 & 2



Anvitha Yerneni
ayernen@g.clemson.edu

16) After reviewing the documentation provided by Microsoft regarding the socket function, I have gathered the following information:

- TCP protocol (6):

IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_STREAM**.

- sock_stream(1):

SOCK_STREAM 1	A socket type that provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. This socket type uses the Transmission Control Protocol (TCP) for the Internet address family (AF_INET or AF_INET6).

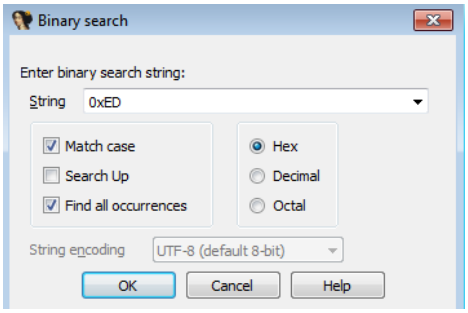
- IPv4 (2):

AF_INET 2	The Internet Protocol version 4 (IPv4) address family.

We can enhance the clarity of the parameters by updating them to their accurately named symbolic constants as defined in IDA.

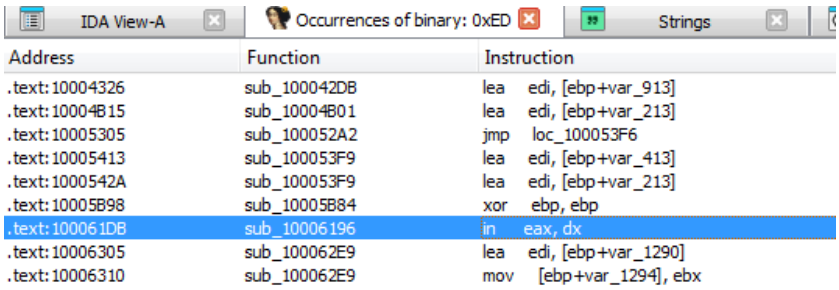
```
.text:100016FB
.text:100016FB loc_100016FB:           ; protocol
.text:100016FB push    IPPROTO_TCP
.text:100016FD push    SOCK_STREAM     ; type
.text:100016FF push    AF_INET         ; af
.text:10001701 call    ds:socket       ; Indirect Call Near Procedure
.text:10001707 mov     edi, eax
```

17) I utilized the shortcut ALT+B to locate every instance of the hexadecimal code 0xED within the text.

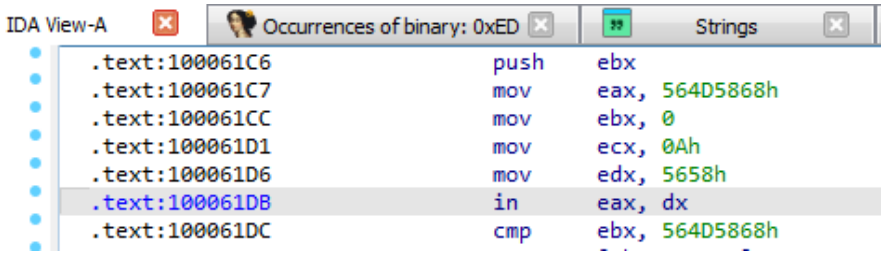


This results in 157 findings, however, only a single instance of the "in" command can be observed, and it is situated at the address 100061DB.

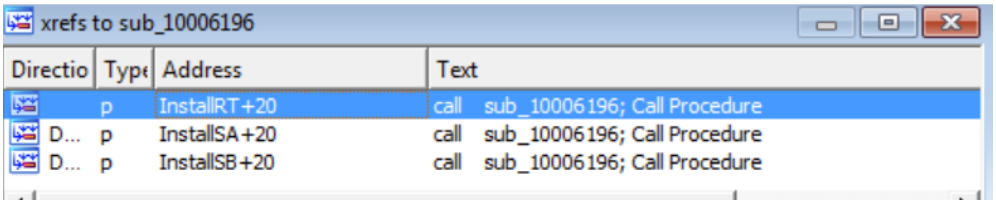
Anvitha Yerneni
ayernen@g.clemson.edu



Upon examining the specified memory address, references to the "564D5868h" magic string are observed. This suggests that the malware employs techniques to identify the presence of VMware.

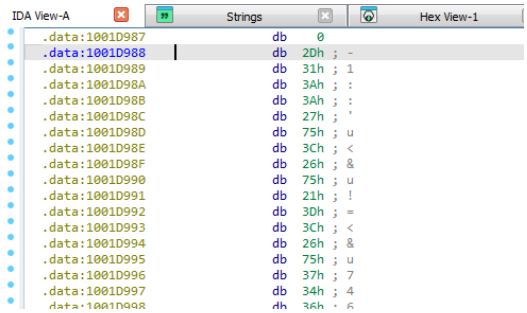


The function utilizing the "in" instruction is identified as sub_10006196, which is referenced three times throughout the documentation.



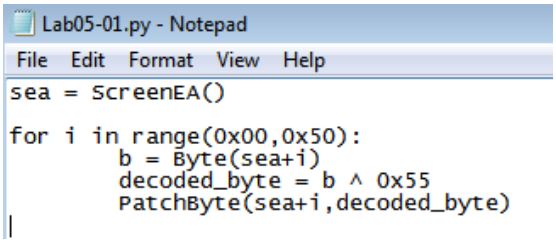
From the initial cross-reference, it's evident that the malware employs virtual machine detection techniques.

18) Accessing this location displays seemingly random bytes of data.



Anvitha Yerneni
ayernen@g.clemson.edu

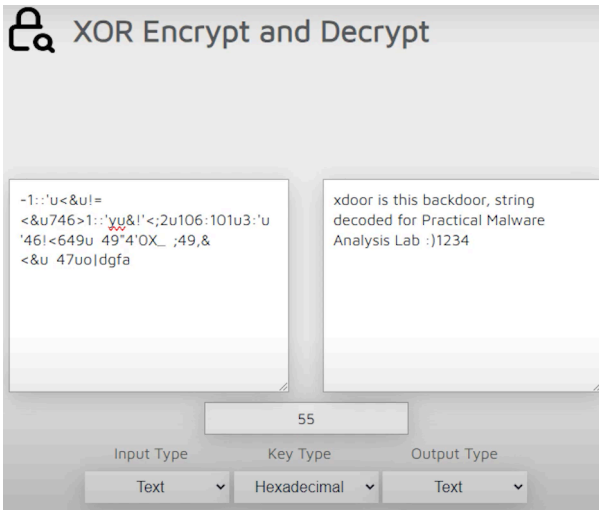
19) The script processes each byte, applying an XOR operation with the hexadecimal value 0x55 to decode it. And gives an output by generating from ASCII value strings



20) Right click on the characters and use (A hotkey) which builds a string out of characters

```
.data:1001D988 a1UUU7461Yu2u10 db '-1::',27h,'u<&u!=<&u746>1::',27h,'yu&!',27h,'<;2u106:101u3:',27h,'u'
.data:1001D9B3                 db    5
.data:1001D9B4 a46649u         db 27h,'46!<649u'
.data:1001D9BD                 db  18h
.data:1001D9BE a4940u          db '49"4',27h,'0u'
.data:1001D9C5                 db  14h
.data:1001D9C6 a49U            db ';49,&<&u'
.data:1001D9CE                 db  19h
.data:1001D9CF a47uoDgfa       db '47uo|dgfa',0
```

Copy the strings and using online XOR compiler generated the output:



21)

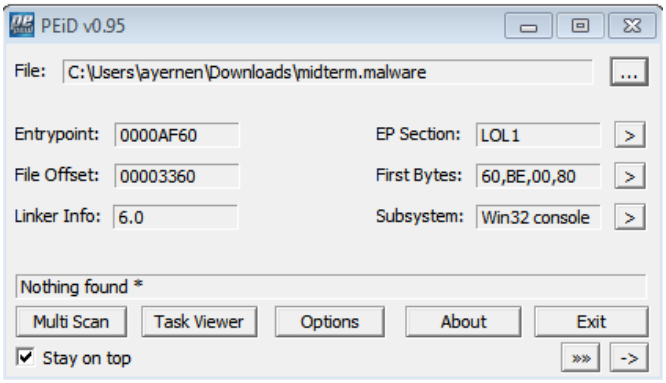
When working in a text editor rather than IDA Pro or a similar tool, these specific functions like `ScreenEA()`, `Byte()`, and `PatchByte()` won't be available because a text editor does not inherently understand the structure of binary files or provide an API for editing binary data. These functions are part of the IDA Python API, which allows users to automate analysis and modification tasks within the IDA environment.

Anvitha Yerneni
ayernen@g.clemson.edu

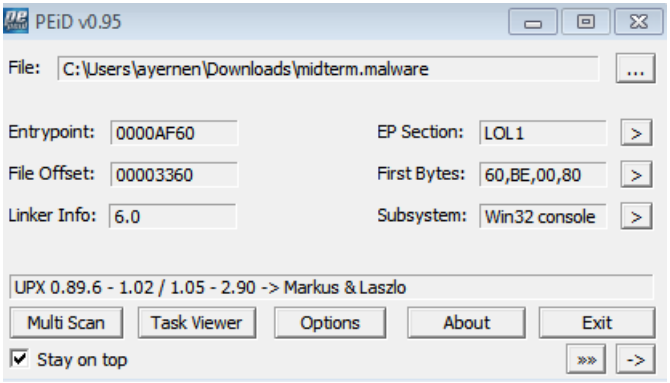
Question 2:

1) A packed file essentially contains malware that's been encrypted within its code. To determine if a file was packed, I utilized a tool called PEiD. Several distinct signs suggested that this file was indeed containing packed malware.

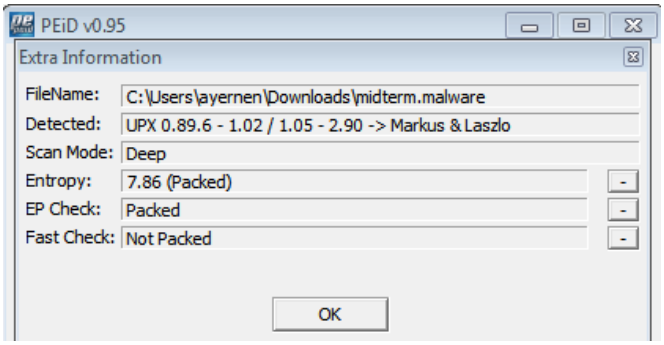
Indication 1:



Below figure displays the output from PEiD upon analyzing the malicious software file.



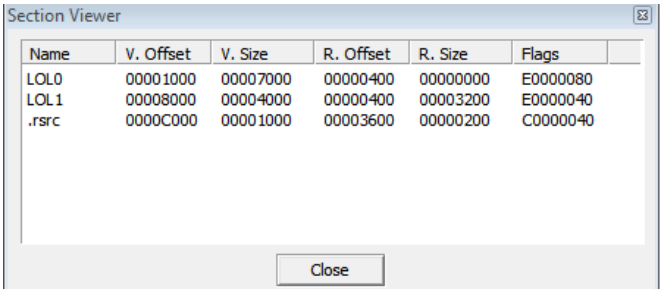
Below figure shows detailed analysis reveals that the file undergoes compression following an in-depth examination.



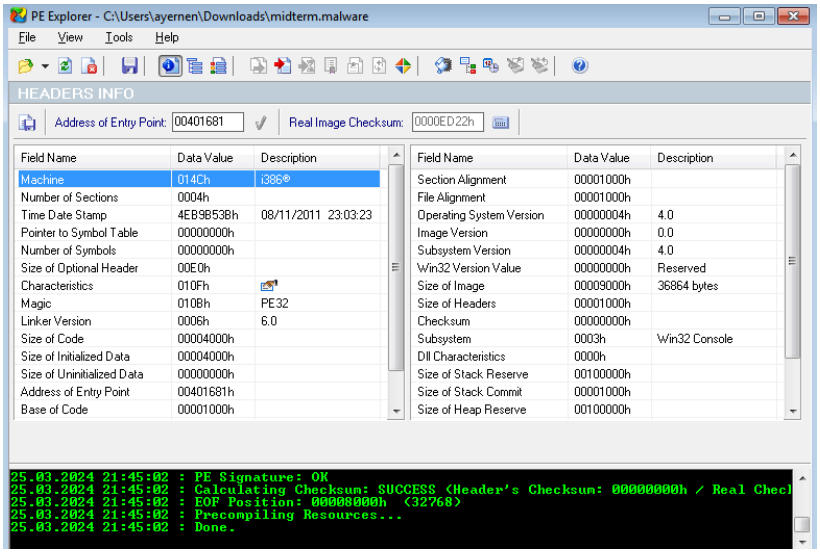
Anvitha Yerneni
ayernen@g.clemson.edu

Upon scanning the malware file with PEiD software, it becomes evident from the displayed image that the file has been compressed. The compression of this file was achieved using an open-source compression tool known as the UPX packer, which is known for its efficient compression capabilities for various executable file formats.

Indication 2:
In the provided screenshot, by using the section viewer, it's apparent that all rows, with the exception of one labeled as ".rsrc," are not decipherable. This strongly suggests that the file has been compressed.



2) We can successfully decompress the malware file by employing a specific unpacker, specifically the UPX unpacker, to handle the decompression process. However, due to the potential for the file to be modified or secured, we will also utilize PE Explorer. This tool allows us to inspect, modify, and fix the binary files before proceeding with the decompression using the UPX unpacker. This combination ensures we can manage and understand the file's contents safely and effectively.
Below figure displays the PE Explorer interface following the execution of the malicious software.



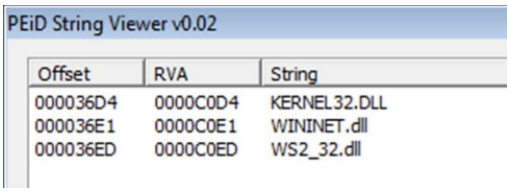
Anvitha Yerneni
ayernen@g.clemson.edu

Typically, there's a variation in size between a compressed file and its original version. This discrepancy signals that the file has been decompressed.

Below figure shows a comparison of file sizes before and after the compression process

```
25.03.2024 21:45:02 : UPX Unpacker Plug-in: <UPX> Compression method: NRV2B_LE32
25.03.2024 21:45:02 : UPX Unpacker Plug-in: <UPX> Compression level: 9
25.03.2024 21:45:02 : UPX Unpacker Plug-in: <UPX> Uncompressed size: 34025 bytes
25.03.2024 21:45:02 : UPX Unpacker Plug-in: <UPX> Compressed size: 12122 bytes
25.03.2024 21:45:02 : UPX Unpacker Plug-in: <UPX> Original file size: 32768 bytes
25.03.2024 21:45:02 : UPX Unpacker Plug-in: <UPX> Filter ID: 26h
```

3) Given the program's compact nature, it appears improbable that we will uncover any significant strings, as the malware creator is evidently taking measures to obscure their activities. I used the PEiD String Viewer v0.02 tool to examine the malware sample for any embedded strings. According to the screenshot provided, the only strings identified were the import libraries previously mentioned in the VirusTotal analysis. Nonetheless, these imported libraries may offer clues regarding the program's intended functions.



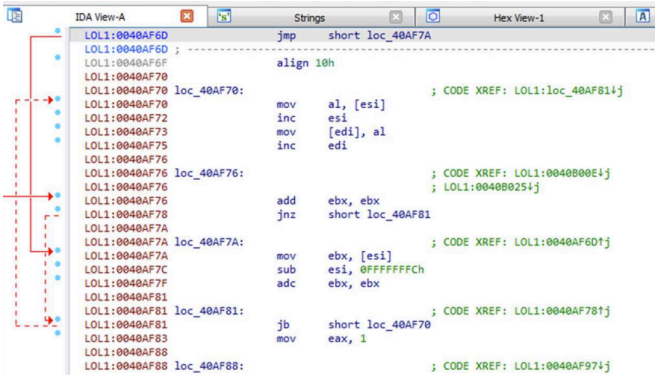
Next, I employed the PEview and Resource Hacker tools to meticulously search for any strings that might have been overlooked by the PEiD String Viewer v0.02 tool. Despite my efforts, I was only able to find the import declarations of the program. The presence of functions like WSAStartup and InternetOpenA indicates that the program is setting up for network library usage and starting the WinINet library. This information about the malware's potential internet usage will be instrumental in guiding our dynamic analysis process.

pFile	Data	Description	Value
000036A8	0000C0F8	Hint/Name RVA	0000 LoadLibraryA
000036AC	0000C106	Hint/Name RVA	0000 GetProcAddress
000036B0	0000C116	Hint/Name RVA	0000 VirtualProtect
000036B4	0000C126	Hint/Name RVA	0000 VirtualAlloc
000036B8	0000C134	Hint/Name RVA	0000 VirtualFree
000036BC	0000C142	Hint/Name RVA	0000 ExitProcess
000036C0	00000000	End of Imports	KERNEL32.DLL
000036C4	0000C150	Hint/Name RVA	0000 InternetOpenA
000036C8	00000000	End of Imports	WININET.dll
000036CC	80000073	Ordinal	0073
000036D0	00000000	End of Imports	WS2_32.dll

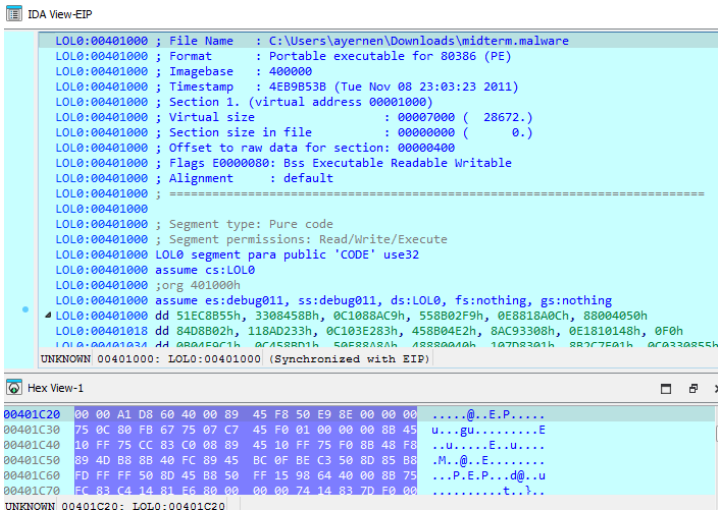
Anvitha Yerneni
ayernen@g.clemson.edu

The displayed information indicates that the malware utilizes host-based signatures and processes. It employs basic dynamic link libraries (DLLs) such as Kernel32.dll, Wininet.dll, and Ws2_32.dll for accessing host and network services. The presence of strings such as "Virtual Free" suggests that the malware might be employing strategies to manipulate strings for the purpose of extracting or altering information within files. Furthermore, the malware appears to be establishing internet connections through various network-related strings. The string "Internet Open A," in particular, reveals the malware's activities in connecting to the web, accessing files, and executing DNS lookups.

4)

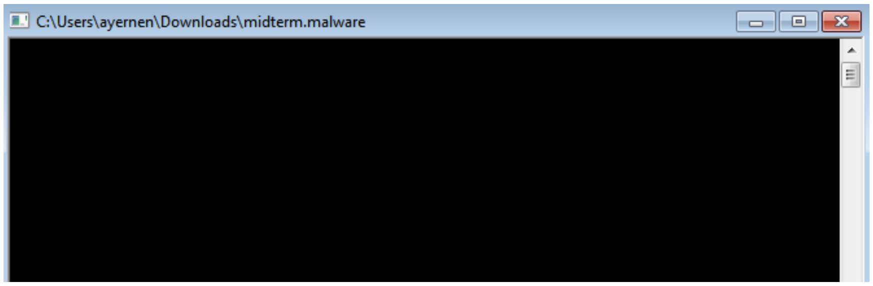


Following the completion of the static examination of this malware specimen, we are proceeding to the dynamic evaluation phase. As illustrated in the provided screenshot, I have employed the IDA Freeware tool for an in-depth analysis of this malware instance.



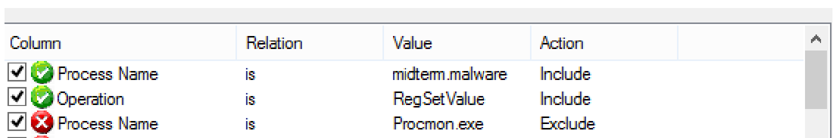
After disassembling the malware sample using the IDA Freeware tool, I identified and set breakpoints on points of interest. The screenshot previously shared illustrates how I executed the malware sample within the IDA Freeware environment.

Anvitha Yerneni
ayernen@g.clemson.edu

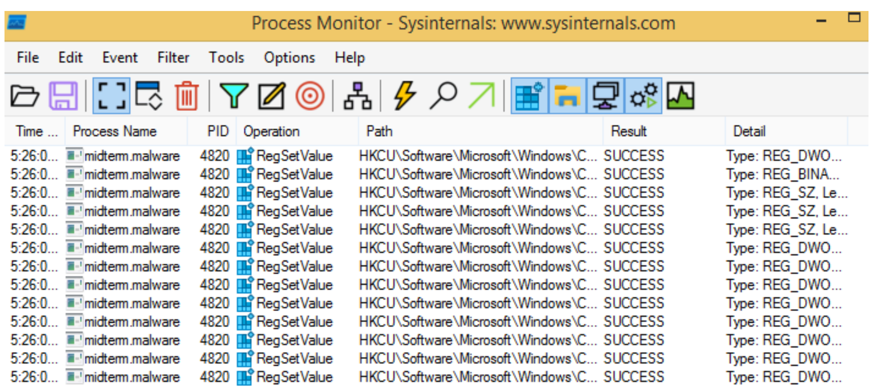


Surprisingly, nothing noticeable happened, apart from the appearance of a blank command prompt window. It is advisable to conduct a more thorough examination of the system using the Process Monitor tool.

5)

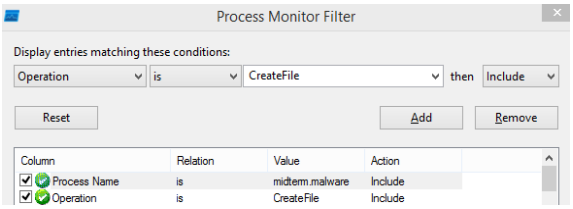


I used the Process Monitor tool to identify any host-based indicators left by the malware sample. In the screenshot provided, I set the filter to only display activities related to the "midterm.malware" process. Furthermore, I included a filter for "RegSetValue" operations to highlight the alterations the malware made to the registry.

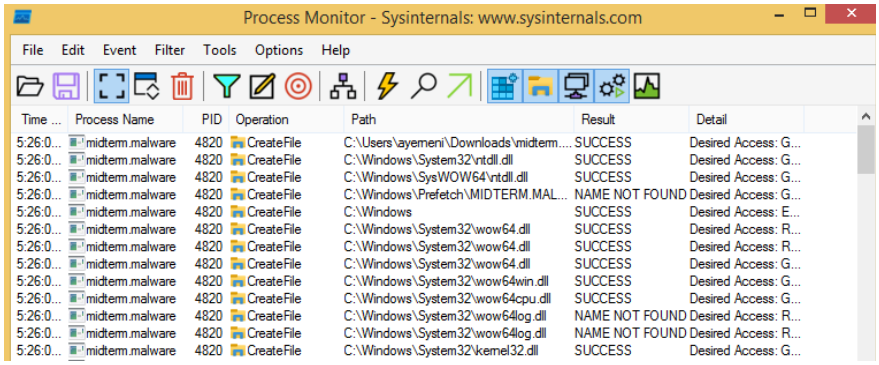


After implementing the previously mentioned filters, I went back to the Process Monitor tool to delve into the findings. In the displayed screenshot, I opted to explore the "Show Registry Activity" feature to scrutinize the registry interactions and uncover how this particular malware sample embedded itself within the registry. This approach yielded a list aligning with my filters, aiding in the assessment of whether a specific computer was compromised by this malware variant.

Anvitha Yerneni
ayernen@g.clemson.edu

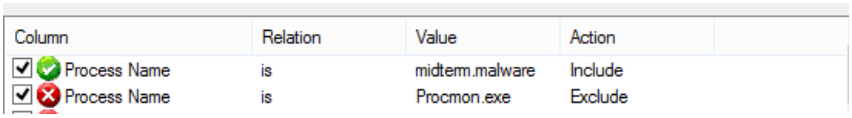


Next, I used the Process Monitor tool to identify more host-based indicators left by the malware sample. In the screenshot provided, I set a filter to only display activities related to the malware sample by using “Process Name is midterm.malware”. Furthermore, I applied a “CreateFile” operation filter to highlight the alterations made by the malware to the file system.



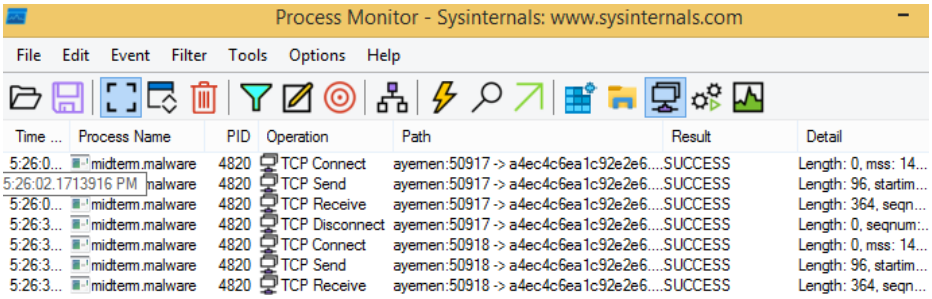
After implementing the previously mentioned filters, I navigated back to the Process Monitor application to delve into the findings. The screenshot depicted my selection of the "Show File System Activity" feature, aiming to scrutinize the files generated by the malware or any configuration files it might employ. A generated list that conformed to my set filters was available, serving as a tool to assess whether a particular system was compromised by the specified malware sample.

6)



I used the Process Monitor tool to identify any network-related traces left by this specific malware. In the screenshot shown, I filtered the results to only show activities related to the malware sample by applying the filter “Process Name is midterm.malware.”

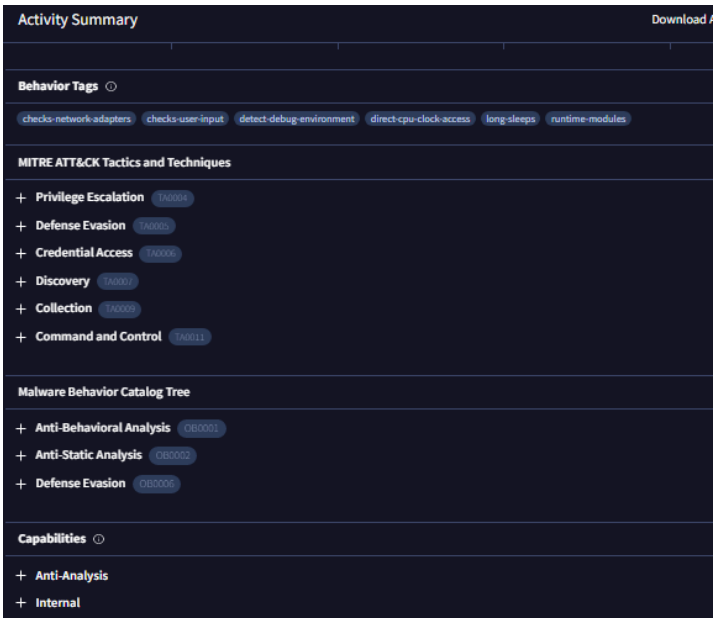
Anvitha Yerneni
ayernen@g.clemson.edu



After applying the previously mentioned filters, I navigated back to the main window of the Process Monitor tool to examine the findings. The screenshot above highlights my selection of the "Show Network Activity" feature, aimed at pinpointing network connections that align with my set criteria. Notably, the connection path “ayernen:50917 -> a4ec4c6ea1c92e2e6.awsglobalaccelerator.com:http
” caught my attention as a potential indicator of whether the examined machine was compromised by the specific malware sample under investigation.

7) I had to use IDA Freeware as an alternative to IDA Pro. The restricted features in this version hindered my analysis process. Acquiring a license for IDA Pro compatible with Windows 7, as opposed to the one for Linux that I was given, could address these limitations.

8)



Following the completion of both static and dynamic assessments, a clearer understanding of the objectives of the examined malware has emerged. The behavior section of the VirusTotal

Anvitha Yerneni
ayernen@g.clemson.edu

the report, as mentioned, sheds light on various actions undertaken by the malware, including alterations to the registry, process initiation, event logging, and establishing communication with an external server. Such a mix of functions suggests a high risk, as it enables the malware's author to access information on the infected machine. The sophistication of this malware is further evidenced by its use of UPX compression, obfuscation techniques, and stealthy execution, complicating the task of discerning its true motives. This thorough analysis has been instrumental in revealing the malware's potential impact and guiding the detection of infections on affected systems.