Assignment 5

Adam Livingston

University Of Arizona

CYBV 454 MALWARE THREATS & ANALYSIS

Professor Galde

5 Apr 2023

LAB 7-1

- LAB07-01.exe : c04fd8d9198095192e7d55345966da2e (Figure 1)

Basic properties

MD5	c04fd8d9198095192e7d55345966da2e
SHA-1	86ee262230cbf6f099b6086089da9eb9075b4521
SHA-256	0c98769e42b364711c478226ef199bfbba90db80175eb1b8cd565aa694c09852
Vhash	024036551d1038z2brz2bz
Authentihash	d914849f01250b4c5b2b4bfcd3db0d4bfa0027075ad39d64bb8d5f3cf7f07b2e
Imphash	8da16e39c9a232fcb6894ec30bf5bdbe
Rich PE header hash	322b4bd14501e689bfb776cdfff8d6cb

Figure 1: Virus Total MD5 Hash for file Lab07-01.exe.

Virus Total found 41 of 69 matching security vendor signatures for this malware (Figure 2) and has a compilation timestamp of 2011-09-30 at 19:49:12 UTC (Figure 3).

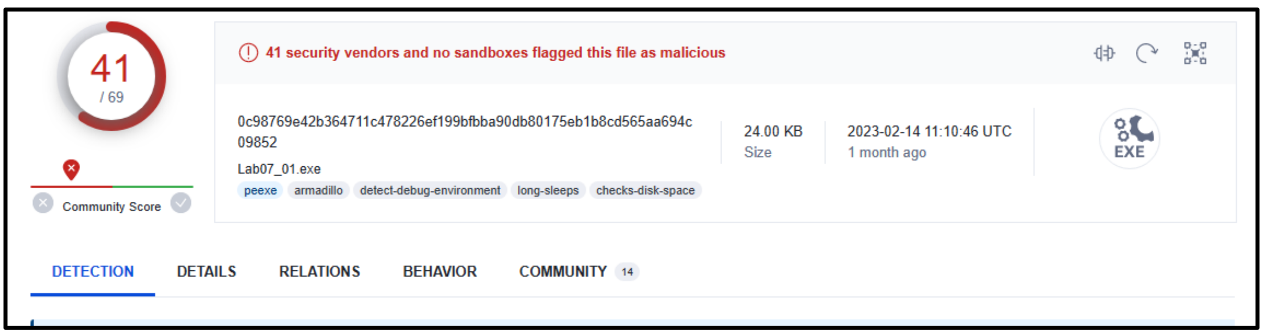


Figure 2: Virus Total Findings for file Lab07-01.exe.

Header

Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2011-09-30 19:49:12 UTC
Entry Point	4496
Contained Sections	3

Figure 3: Virus Total compilation timestamp for file Lab07-01.exe.

The file appears to import three dynamic linked libraries: kerenel32, advapi, and wininet. Kernel32.dll indicates that it has the capability to access and modify the core OS functions. Wininet.dll shows that it imports and implements functions related to networking protocols. Advapi32.dll indicates that core Windows components will be altered, such as the Service Manager and Registry.

Imports

+ ADVAPI32.dll
+ KERNEL32.dll
+ WININET.dll

Figure 4: Virus Total imports for file Lab07-01.exe.

The networking protocol functions that are supported within wininet.dll may be related to the detected network connections identified by Virus Total for Lab07-01.exe. There is an HTTP request to practicalmalwareanalysis.com and malwareanalysisbook.com (Figure 5) and lots of IP traffic with potential Transport Layer Security (TLS) implemented for practicalmalwareanalysis.com (Figure 6).

HTTP Requests

+ http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/CABD2A79A1076A31F21D253635CB039D4329A5E8.crt?48c5e15bd6f2d359
+ http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/CABD2A79A1076A31F21D253635CB039D4329A5E8.crt?7f8174eae7f5cd77
+ http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab?6ec0cc6145b15d4a
+ http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab?a5da56a174b07afd
+ http://www.malwareanalysisbook.com/
+ http://www.practicalmalwareanalysis.com/

Figure 5: Virus Total HTTP requests for file Lab07-01.exe.

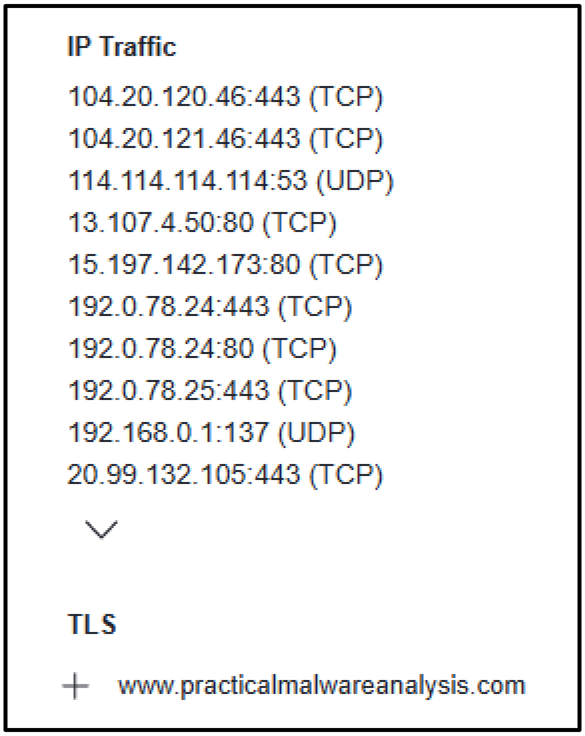


Figure 6: Virus Total IP traffic for file Lab07-01.exe.

Virus Total also reports that the file has behavior of persistence, privilege escalation, and defense evasion (Figures 7 and 8). It also shows behavior of downloading files using HTTP and using HTTPS for encrypted channels (Figure 8), most likely in reference for the TLS networking behavior identified in Figure 6. Based on these findings, this malware is most likely a generic trojan and uses HTTP and HTTPS (ports 80 and 443) to download additional packages onto the infected machine when the file is run by the user. It is possible that this malware reads code from the domain in order to execute more commands as seen in the malware analyzed in Assignment 4.

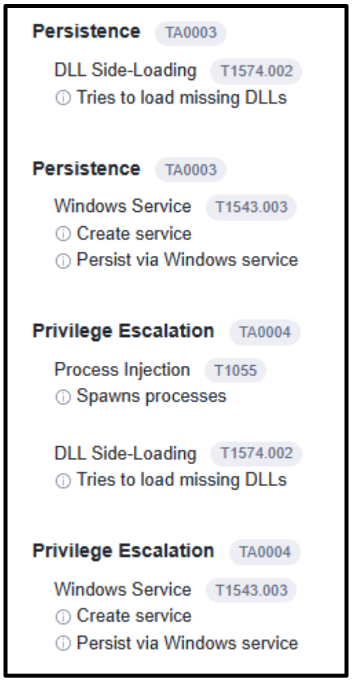


Figure 7: Virus Total behavior for file Lab07-01.exe.

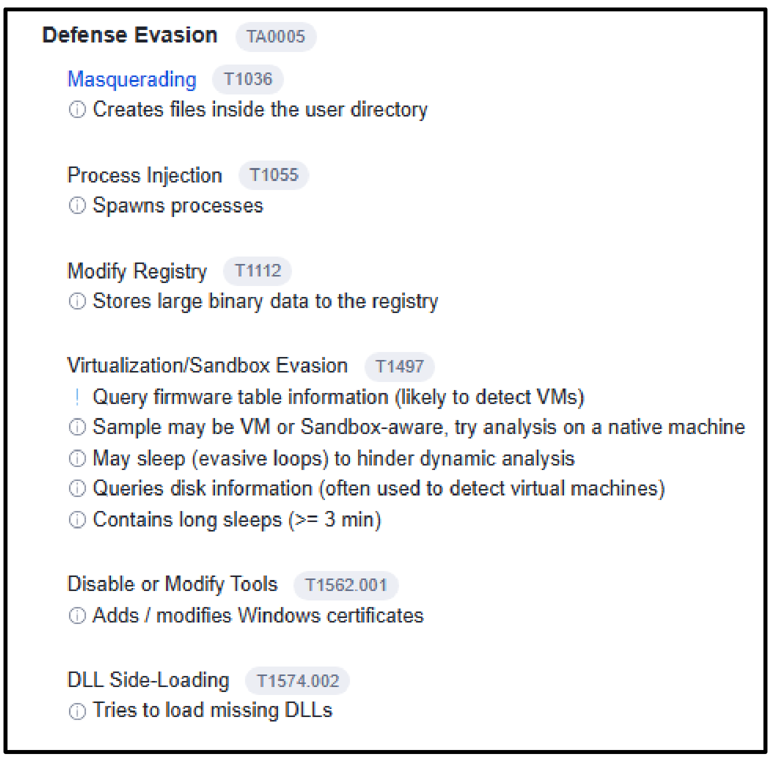


Figure 8: Virus Total behavior for file Lab07-01.exe.

Command and Control TA0011

Application Layer Protocol T1071

ⓘ Uses HTTPS
ⓘ Downloads files from webservers via HTTP
ⓘ Performs DNS lookups

Non-Application Layer Protocol T1095

ⓘ Downloads files from webservers via HTTP
ⓘ Performs DNS lookups

Ingress Tool Transfer T1105

ⓘ Downloads files from webservers via HTTP

Encrypted Channel T1573

ⓘ Uses HTTPS
ⓘ Uses HTTPS for network communication, use the SSL MITM Proxy cookbook for further analysis

Figure 9: Virus Total network-based behavior for file Lab07-01.exe.

LAB 7-1

LAB 7-1 Question 1

How does this program ensure that it continues running (achieves persistence) when the computer is restarted?

BLUF: Creates the service “Malservice”.

To start, the malware was loaded into IDA Pro for static analysis. Since malware that runs on Windows Operating systems often achieve persistence through services which allow it to run on startup, we first look at the Imports Window to identify any suspicious functions/API keys. Looking for functions/API keys that relate to services, we immediately notice two: OpenSCManagerA and StartServiceCtrlDispatcherA (Figure 10).

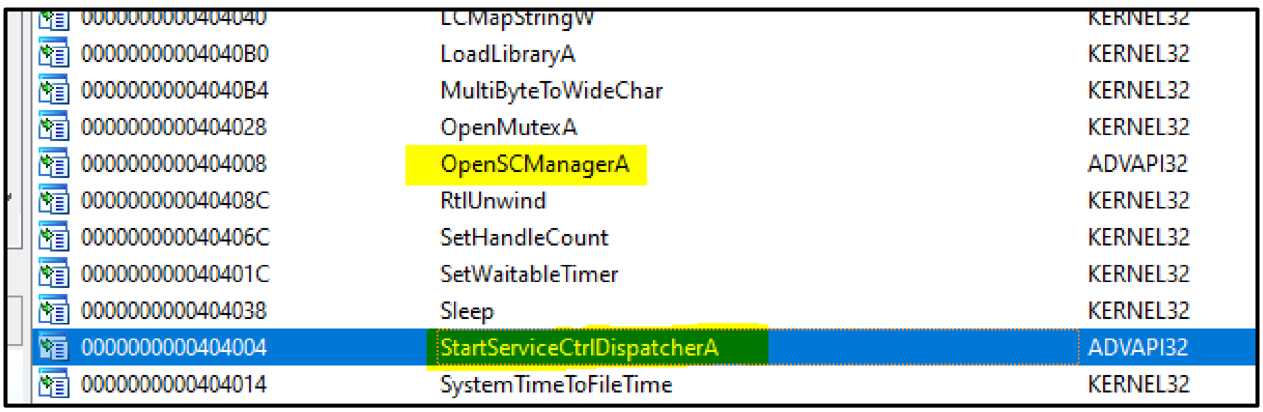


Figure 10: Imports Window in IDA and service-related functions.

StartServiceCtrlDispatcherA is part of the winsvc.h header file and connects the main thread of a service process to the service control manager (documentation here). OpenSCManagerA is also part of winsvc.h and establishes a connection to the service control manager (documentation here). We also see an import for CreateServiceA (Figure 11), also part of winsvc.h, which creates a service object and adds it to the specified service control manager database (documentation).

Because of these service-related functions, it is reasonable to conclude that this malware most likely achieves persistence through the creation of a service.

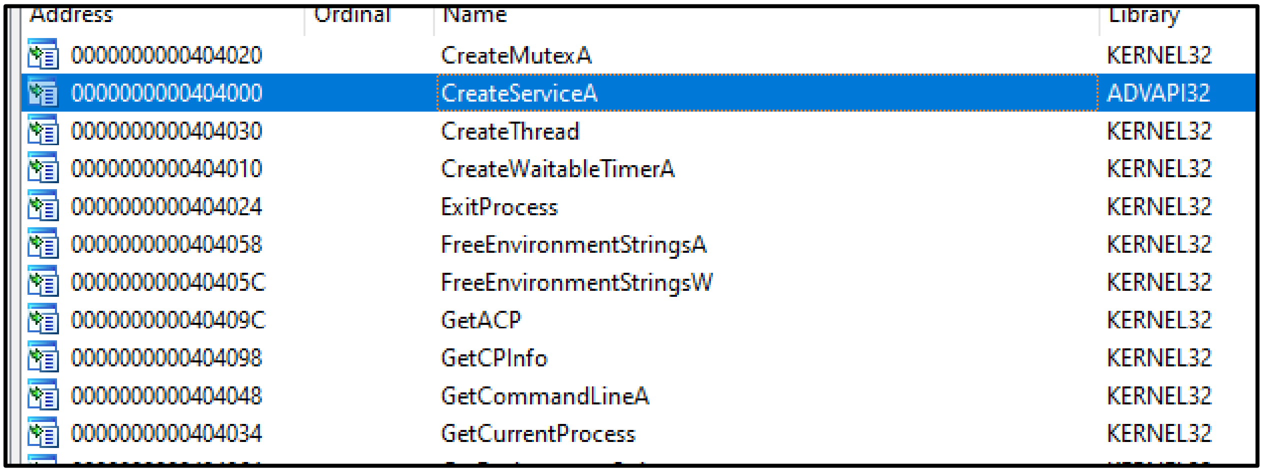


Figure 11: CreateServiceA import in IDA Imports Window.

To begin, we examine the xrefs for StartServiceCtrlDispatcherA and see there is one location where the function is called (Figure 12).

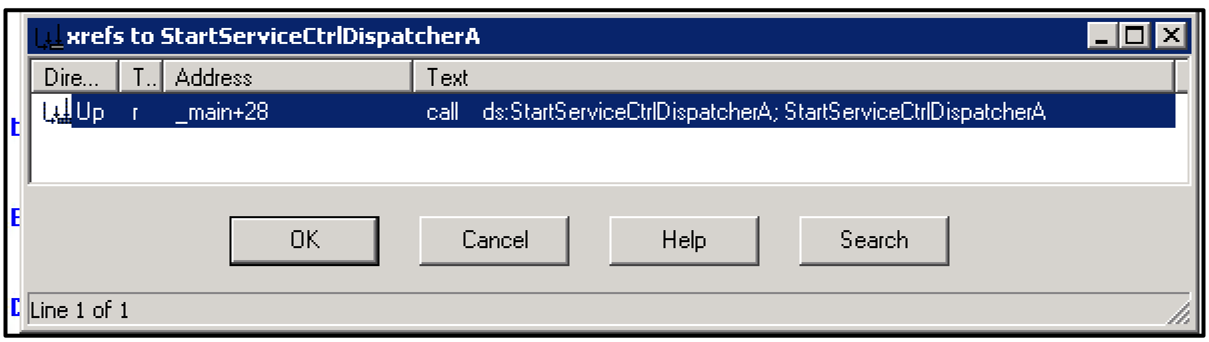


Figure 12: One Xref to StartServiceCtrlDispatcherA.

Reviewing the documentation for this function, we can expect to find one argument passed into it, being lpServiceStartTable. This argument is “a pointer to an array of SERVICE_TABLE_ENTRY structures containing one entry for each service that can execute in the calling process.” Therefore, this malware is the calling process and whatever is passed into the function is the service that can execute the malware on startup. Navigating to where the function is called, we find ourselves in _main and see that indeed lpServiceStartTable (located in

EAX) is pushed onto the stack prior to the function being called at 0x0040100F (Figure 13). We also see that 0x00401003 that the effective address of [esp+10h+ServiceStartTable] was loaded into EAX, then on the next line we see [esp+10h+ServiceStartTable.lpServiceName] is identified as “MalService”. Therefore we can conclude that the service name which executes the malware on startup is titled “MalService” which is stored in the aMalservice variable.

```
00401000     ; int __cdecl main(int argc,const char **argv,const char *envp)
00401000     _main proc near
00401000
00401000     ServiceStartTable= SERVICE_TABLE_ENTRYA ptr -10h
00401000     var_8= dword ptr -8
00401000     var_4= dword ptr -4
00401000     argc= dword ptr  4
00401000     argv= dword ptr  8
00401000     envp= dword ptr  0Ch
00401000
00401000 000 sub     esp, 10h          ; Integer Subtraction
00401003 010 db 8Dh,44h,24h,0 ; <BAD>lea     eax, [esp+10h+ServiceStartTable] ; Load Effective Address
00401007 010 db 0C7h,44h,24h,0,30h,50h,40h,0 ; <BAD>mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalservice ; "MalService
0040100F 010 push    eax               ; lpServiceStartTable
00401010 014 mov     [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
00401018 014 mov     [esp+14h+var_8], 0
00401020 014 mov     [esp+14h+var_4], 0
00401028 014 call    ds:StartServiceCtrlDispatcherA ; Indirect Call Near Procedure
0040102E 010 push    0
00401030 014 push    0
00401032 018 call    sub_401040        ; Call Procedure
00401037 018 add     esp, 18h          ; Add
0040103A 000 retn                      ; Return Near from Procedure
0040103A     _main endp
0040103A
```

Figure 13: One Xref to StartServiceCtrlDispatcherA.

aMalservice is located in the .data section at 0x00405030 (Figure 14). Here, we also see here another reference to the string “Malservice” and is defined as the variable “DisplayName”.

```
.data:0040502F 00                                db    0
.data:00405030 4D 61 6C 53 65 72+    aMalservice     db 'MalService',0
.data:0040503B 00                                align 4
.data:0040503C                       ; char DisplayName[]
.data:0040503C 4D 61 6C 73 65 72+    DisplayName     db 'Malservice',0       ; DATA XREF: sub_401040+71↑o
.data:0040503C 76 69 63 65 00                                               ; sub_401040+76↑o
.data:00405047 00                                align 4
.data:00405048                       ; char Name[]
.data:00405048 48 47 4C 33 34 35+    Name            db 'HGL345',0           ; DATA XREF: sub_401040+6↑o
.data:00405048 00                                                           ; sub_401040+25↑o
.data:0040504F 00                                align 10h
```

Figure 14: Malservice strings and locations in .data.

To confirm this, we do a dynamic analysis. The first thing noticed when running the malware is the appearance of a cmd.exe window and its persistence (Figure 15). Although suspicious, it doesn't confirm the name of the service. However, when clicking on “Run” within the Start menu and typing “msconig”, we notice Malservice residing within the Services tab in the System Configuration Utility (Figure 16).

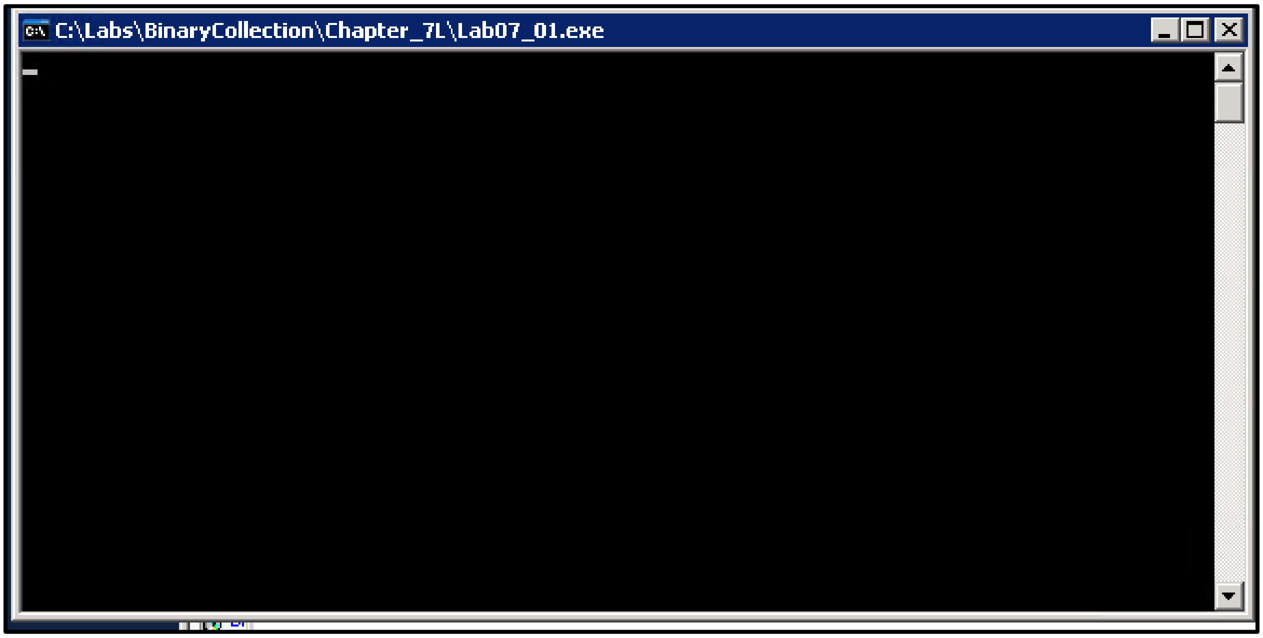


Figure 15: Cmd.exe window persists after malware is run.

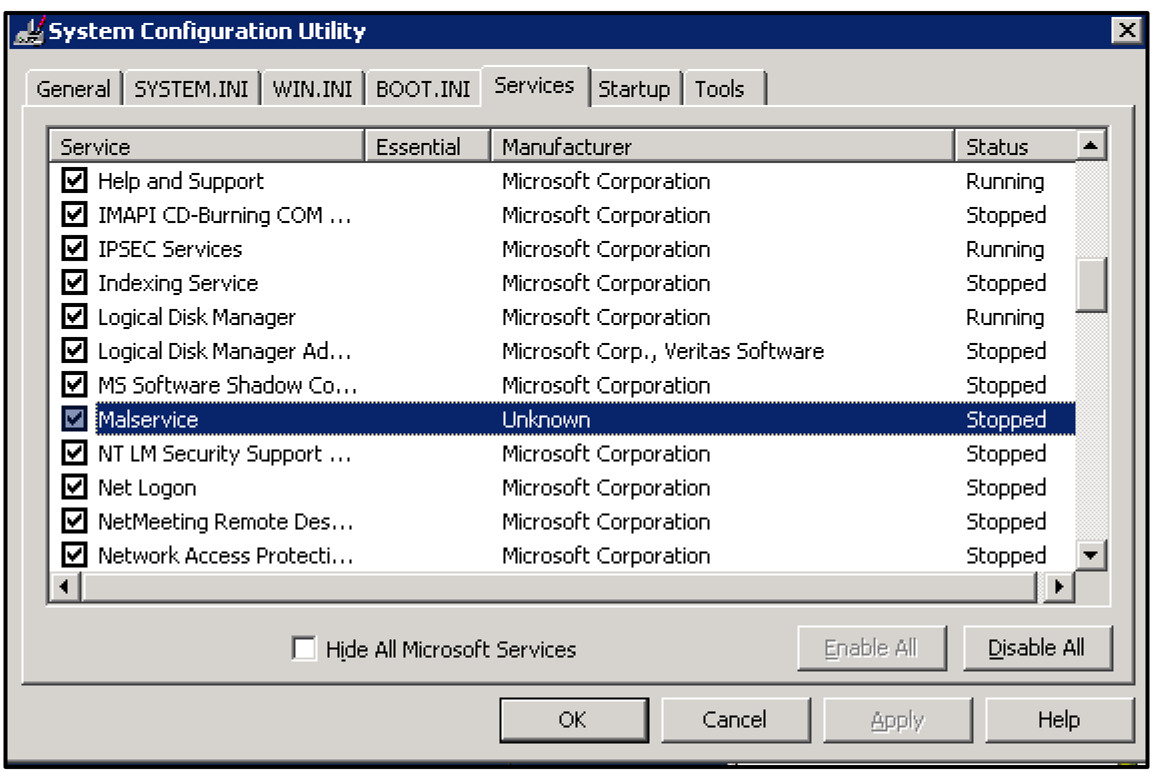


Figure 16: Malservice in Services.

The capture created by procmon was filtered to find the registry hive for services in HKLM\SYSTEM\CurrentControlSet\services. We see that services.exe created Malservice in the registry (Figure 17). It has a parent PID of 512 (Figure 18). PID 512 was identified as winlogon.exe within Process Explorer (Figure 19).

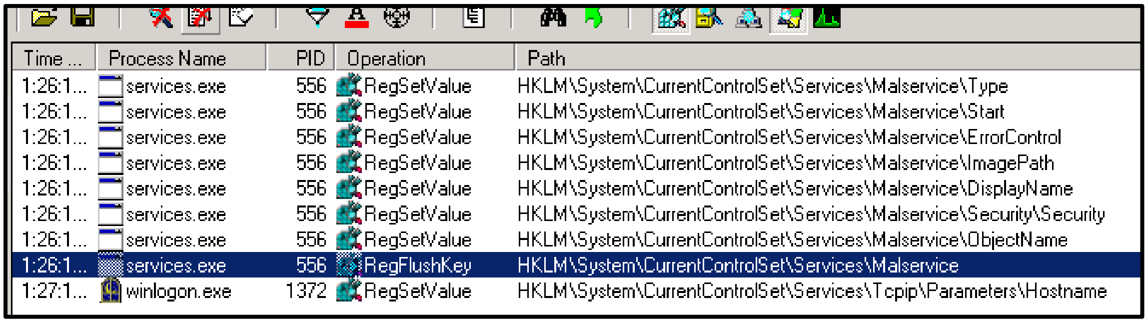


Figure 17: Malservice registry keys set.

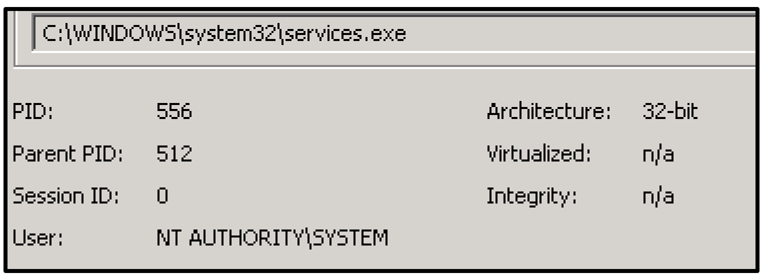


Figure 18: Services.exe parent PID.

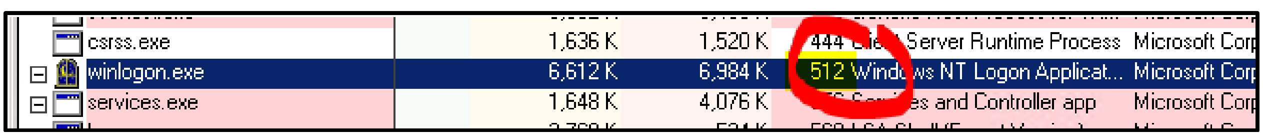


Figure 19: winlogon.exe PID

PID 512 did not have any activity within procmon (Figure 20). However, it is clear that at Lab07-01.exe, through the use of StartServiceCtrlDispatcherA, used services.exe to change the registry values of the startup programs to include Malservice. We can confirm that Malservice exists within the registry by opening the registry editor. The image path is Lab07-01.exe (Figure 21).

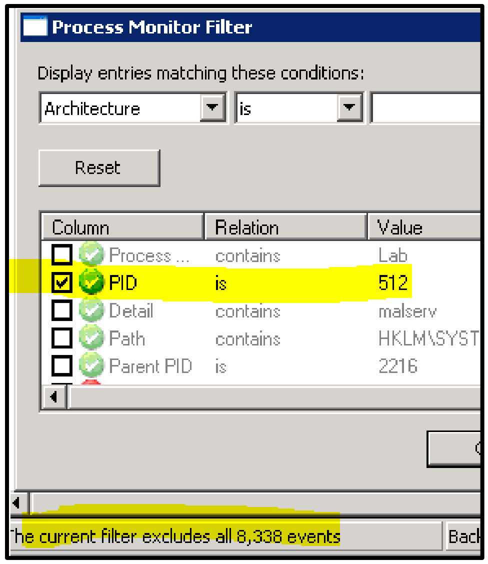


Figure 20: winlogon.exe PID.

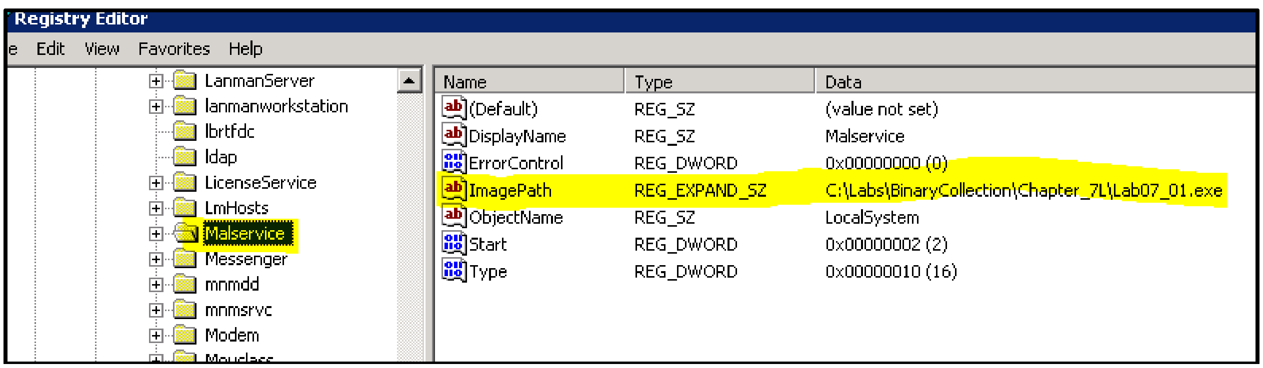


Figure 21: Lab07-01.exe is Malservice.

LAB 7-1 Question 2

Why does this program use a mutex?

BLUF: To ensure only one instance of Lab07-01.exe is running at a time.

The _main function has subroutine 401040 and is the last call before the _main endpoint and after StartServiceCtrlDispatcherA (Figure 22).

```
401010 014 mov     [esp+14h+ServiceStartTable.lpServiceProc], off
401018 014 mov     [esp+14h+var_8], 0
401020 014 mov     [esp+14h+var_4], 0
401028 014 call    ds:StartServiceCtrlDispatcherA ; Indirect Call
40102E 010 push    0
401030 014 push    0
401032 018 call    sub_401040         ; Call Procedure
401037 018 add     esp, 18h           ; Add
40103A 000 retn                       ; Return Near from Procedure
40103A     _main endp
40103A
```

Figure 22: sub_401040 in _main.

The beginning portion of the subroutine calls the external function OpenMutexA (Figure 23). Documentation could not be found for OpenMutexA, but a similar function OpenMutexW has the same argument names as OpenMutexA in this malware. Of note, we can deduce that the name of the mutex attempted to be opened is titled “HGL345”. The return value in EAX is tested against itself and will jump if zero to a location that skips over an ExitProcess function call (if it cannot obtain a handle to HGL345, then the program exits).

```
00401040
00401040
00401040
00401040     sub_401040 proc near
00401040
00401040     SystemTime= SYSTEMTIME ptr -400h
00401040     DueTime= LARGE_INTEGER ptr -3F0h
00401040     BinaryPathName= byte ptr -3E8h
00401040
00401040 000 sub     esp, 400h          ; Integer Subtraction
00401046 400 push    offset Name        ; "HGL345"
0040104B 404 push    0                  ; bInheritHandle
0040104D 408 push    1F0001h            ; dwDesiredAccess
00401052 40C call    ds:OpenMutexA      ; Indirect Call Near Procedure
00401058 400 test    eax, eax           ; Logical Compare
0040105A 400 jz      short loc_401064 ; Jump if Zero (ZF=1)
```

Figure 23: Beginning of sub_401040 with OpenMutexA.

If the test returns a zero flag, then we see that “HGL345” is pushed onto the stack again along with other function arguments for CreateMutexA (Figure 24). Within the documentation, we see that if a 0 is pushed for the Mutex Attributes argument then the mutex cannot be inherited by a child process. The 0 passed into bInitialOwner argument means that no thread will own the mutex initially. This means that when this mutex is assigned to the service, only one copy of the program will run at a time.

```
00401064
00401064     loc_401064:
00401064 400 push    esi
00401065 404 push    offset Name        ; "HGL345"
0040106A 408 push    0                  ; bInitialOwner
0040106C 40C push    0                  ; lpMutexAttributes
0040106E 410 call    ds:CreateMutexA    ; Indirect Call Near Procedure
00401074 404 push    3                  ; dwDesiredAccess
00401076 408 push    0                  ; lpDatabaseName
00401078 40C push    0                  ; lpMachineName
0040107A 410 call    ds:OpenSCManagerA  ; Establish a connection to the service
0040107A                                ; control manager on the specified computer
0040107A                                ; and opens the specified database
00401080 404 mov     esi, eax
00401082 404 call    ds:GetCurrentProcess ; Indirect Call Near Procedure
00401088 404 lea     eax, [esp+404h+BinaryPathName] ; Load Effective Address
```

Figure 24: sub _401040 calling CreateMutexA.

To dynamically see the mutex, we can use Process Explorer and search for the mutex "HGL345" with the "Find Handle" tool after running the malware. We immediately see that the mutex HGL345 is only used by Lab07-01.exe (Figure 25).

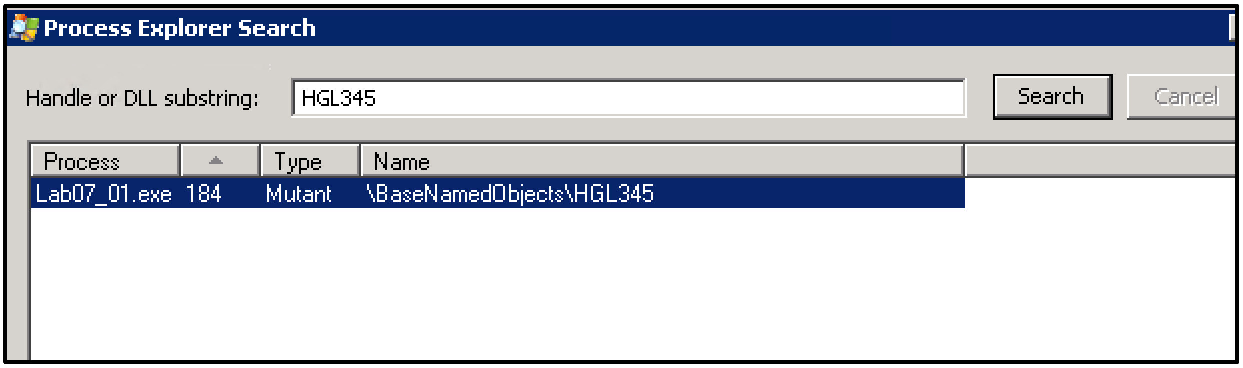


Figure 25: HGL345 used by Lab07_01.exe process.

Using the WinObj.exe utility, we can also see the existence of HGL345 in the BaseNamedObjects directory as a type of "Mutant", aka mutex (Figure 26). It has the lock symbol due to it currently in use by Lecture07-01.exe. Unfortunately, the creation of the mutex was not captured on procmon. But the existence of the mutex and association with Lab07-01.exe within process explorer and being shown as locked within the WinObj utility is sufficient enough evidence to show that HGL345 is only associated with the malware file.

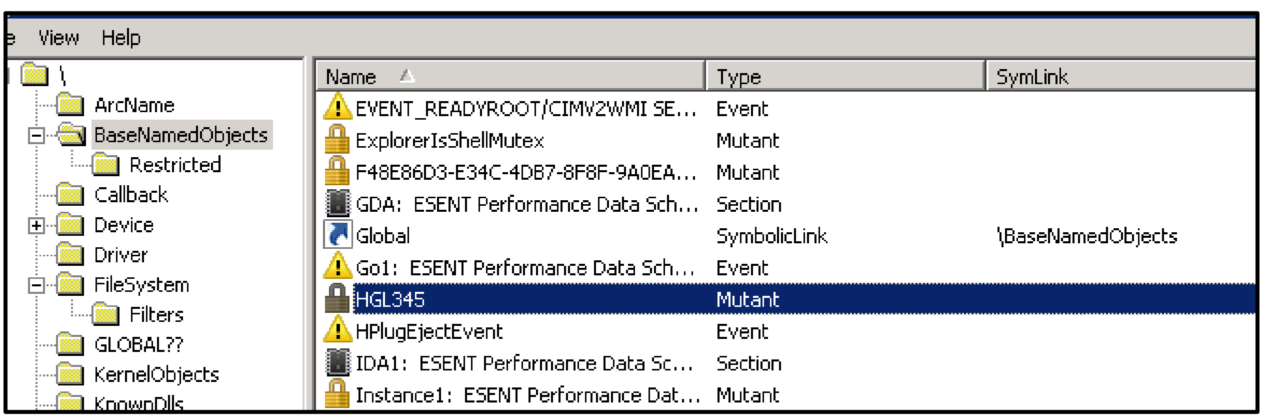


Figure 26: HGL345 in WinObj utility.

LAB 7-1 Question 3

What is a good host-based signature to use for detecting this program?

BLUF: HGL345 mutex and Malservice.

In the both the static and dynamic analysis in Question 1 and Question 2, it is clear that the two main host-based signatures to look for on an infected machine are the existence of the HGL345 mutex and the Malservice service. To look for Malservice, the user needs to look in the system registry at HKLM\SYSTEM\CurrentControlSet\services for the self-titled entry (as seen in Figure 21 here). Additionally, the existence of the mutex HGL345 can be searched for in two places: In process explorer as shown in Figure 25 here, and in WinObj.exe as shown in Figure 26 here.

An additional, yet more obvious host-based indicator would be the existence of a cmd.exe window that won't go away, observed when the malware was running as seen in Figure 15 here.

The aim of this question is to identify crucial host-based indicators for potential endpoint signatures using static and dynamic analysis. Provide a concise yet thorough analysis of your findings.

LAB 7-1 Question 4

What is a good network-based signature for detecting this malware?

BLUF:

Moving further down in the code within IDA, there is a CreateThread external function call (documentation here) whose purpose is to create a thread to execute within the virtual address space of the calling process. There are five arguments passed into the function, only one of which is not 0 but is the offset "StartAddress" into the lpStartAddress parameter (Figure 27). This is a pointer to the application-defined function to be executed by the thread. Therefore it is reasonable to conclude that whatever is defined within the variable StartAddress is a function. This will most likely contain some information related to network-based indicators.

```
00401126
00401126                          loc_401126:              ; lpThreadId
00401126 408 6A 00                push    0
00401128 40C 6A 00                push    0                ; dwCreationFlags
0040112A 410 6A 00                push    0                ; lpParameter
0040112C 414 68 50 11 40 00       push    offset StartAddress ; lpStartAddress
00401131 418 6A 00                push    0                ; dwStackSize
00401133 41C 6A 00                push    0                ; lpThreadAttributes
00401135 420 FF D7                call    edi ; CreateThread ; Indirect Call Near Procedure
00401137 408 4E                   dec     esi              ; Decrement by 1
00401138 408 75 EC                jnz     short loc_401126 ; Jump if Not Zero (ZF=0)
```

Figure 27: CreateThread function called by sub_401040.

In Figure 28, we indeed see that there are some network-based indicators within this function. Particularly, we see the call to InternetOpenA for the user-agent of Internet Explorer 8.0. We then see the call to InternetOpenUrlA which passes the URL of malwareanalysisbook.com into the lpszUrl parameter. This parameter specifies the URL to begin reading. Therefore, a good network-based indicator for an infected machine would be the attempt to connect to www. malwareanalysisbook.com.

Figure 28: StartAddress function.

Back in Question 1's dynamic analysis, it was discovered that Lab07-01.exe was run as the service entitled "Malservice" and was created by services.exe (seen in Figure 17 here). Since this is running as a service, the program that will run it will be svchost.exe. Using a procmon capture, we see numerous TCP Send and Receive operations by svchost.exe from the infected machine's IP address (Figure 29) over port 3389 to IP address 10.139.4.146 on port 51305 (traffic can be seen in Figure 30). At no point was another internet-based application used during the procmon capture. Port 3389 is especially concerning since it supports Remote Desktop Protocol (RDP). The machine being connected to will use port 3389 while the connecting machine will use a varied port. In this case, port 51305 is being used by the connecting machine which is a dynamic/private port. Therefore, another good network-based indicator would be traffic to unknown and unverified IP addresses via port 3389 over TCP.

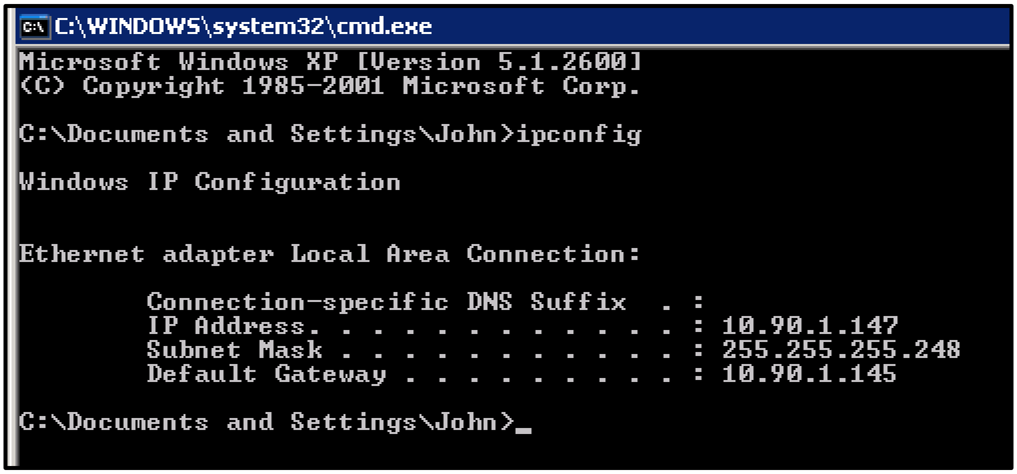
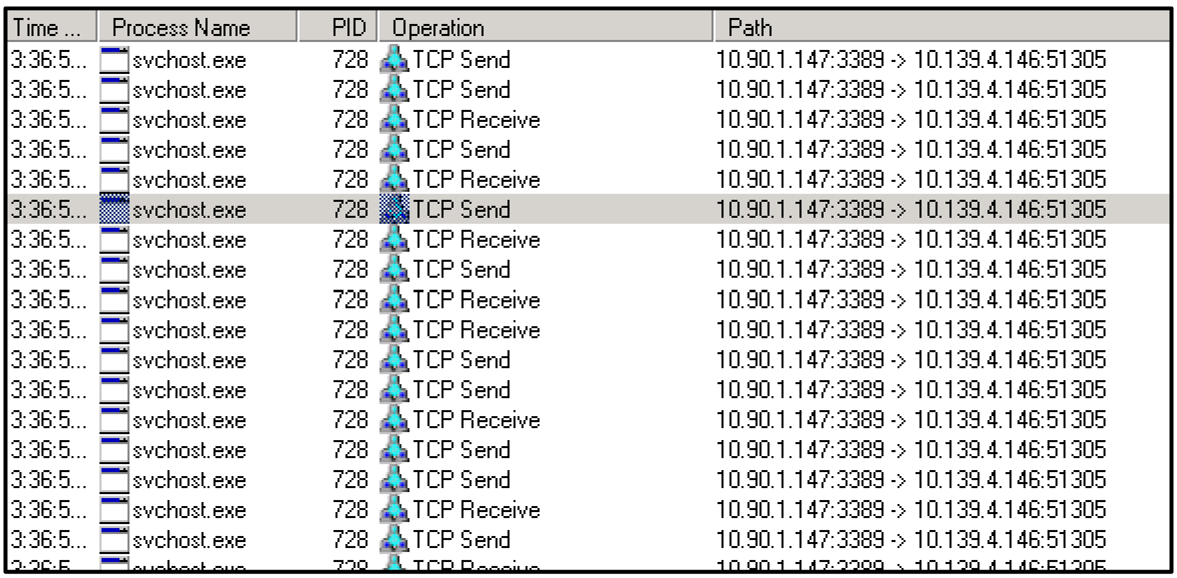


Figure 29: Host machine IP address.



Time ...	Process Name	PID	Operation	Path
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Receive	10.90.1.147:3389 -> 10.139.4.146:51305
3:36:5...	svchost.exe	728	TCP Send	10.90.1.147:3389 -> 10.139.4.146:51305

Figure 30: Procmon network traffic capture.

Figure 26 shows that we can see that the next call is to open the service manager at 0x004017A. It then calls the external function GetCurrentProcess followed by followed by GetModuleFileNameA. GetModuleFileNameA will return the size of the string upon success and store the name of the file into a buffer. The effective address at [esp+404h+BinaryPathName], now stored in EAX and is the lpFileName argument, is a pointer to that buffer. This allows the malware to dynamically obtain its information instead of relying on hard-coding its storage location.

```
00401078 40C push    0                   ; lpMachineName
0040107A 410 call    ds:OpenSCManagerA ; Establish a connection to the service
0040107A                                 ; control manager on the specified computer
0040107A                                 ; and opens the specified database
00401080 404 mov     esi, eax
00401082 404 call    ds:GetCurrentProcess ; Indirect Call Near Procedure
00401088 404 lea     eax, [esp+404h+BinaryPathName] ; Load Effective Address
0040108C 404 push    3E8h                ; nSize
00401091 408 push    eax                 ; lpFilename
00401092 40C push    0                   ; hModule
00401094 410 call    ds:GetModuleFileNameA ; Indirect Call Near Procedure
0040109A 404 push    0                   ; lpPassword
0040109C 408 push    0                   ; lpServiceStartName
0040109E 40C push    0                   ; lpDependencies
004010A0 410 push    0                   ; lpdwTagId
004010A2 414 lea     ecx, [esp+414h+BinaryPathName] ; Load Effective Address
004010A6 414 push    0                   ; lpLoadOrderGroup
004010A8 418 push    ecx                 ; lpBinaryPathName
004010A9 41C push    0                   ; dwErrorControl
004010AB 420 push    2                   ; dwStartType
004010AD 424 push    10h                 ; dwServiceType
004010AF 428 push    2                   ; dwDesiredAccess
004010B1 42C push    offset DisplayName ; "Malservice"
004010B6 430 push    offset DisplayName ; "Malservice"
004010BB 434 push    esi                 ; hSCManager
004010BC 438 call    ds:CreateServiceA ; Indirect Call Near Procedure
004010C2 404 xor     edx, edx            ; Logical Exclusive OR
```

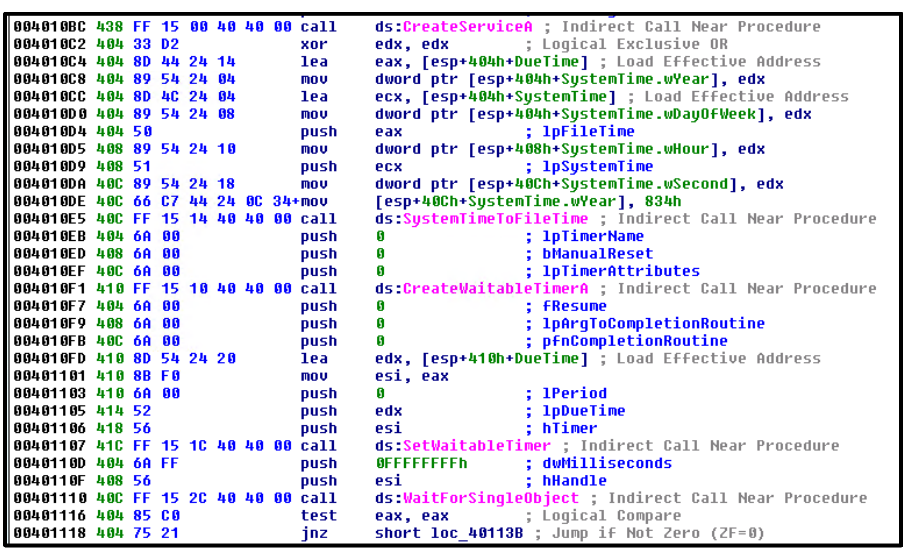
Figure 26: Next set of instructions in sub_401040.

LAB 7-1 Question 5

What is the purpose of this program?

BLUF: Logic/time bomb Denial of Service.

After CreateServiceA is called to create Malservice, there are some interesting pieces of code that use external time-related functions (Figure 31). The first is SystemTimeToFileTime after manipulating the SystemTime structure's year, day of week, hour, and second values. We see that EDX has a XOR instruction performed on it, setting it to 0, then that value is moved into each portion of the structure. It then passes hex value 834 (decimal 2100) into the year value. We also see lpFileTime and lpSystemTime pushed onto the stack for the function parameters, each of which were loaded with pointers to the effective address of whatever was stored the DueTime and SystemTime. Even though the SystemTime year value of 2100 was set after ECX was pushed onto the stack, the value is still passed into the function due to the pointer. The function converts the system time format into a file time format for the date of 1 Jan 2100 at midnight UTC.



```
004010BC 438 FF 15 00 40 40 00 call    ds:CreateServiceA ; Indirect Call Near Procedure
004010C2 404 33 D2             xor     edx, edx        ; Logical Exclusive OR
004010C4 404 8D 44 24 14       lea     eax, [esp+404h+DueTime] ; Load Effective Address
004010C8 404 89 54 24 04       mov     dword ptr [esp+404h+SystemTime.wYear], edx
004010CC 404 8D 4C 24 04       lea     ecx, [esp+404h+SystemTime] ; Load Effective Address
004010D0 404 89 54 24 08       mov     dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
004010D4 404 50                push    eax             ; lpFileTime
004010D5 408 89 54 24 10       mov     dword ptr [esp+408h+SystemTime.wHour], edx
004010D9 408 51                push    ecx             ; lpSystemTime
004010DA 40C 89 54 24 18       mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
004010DE 40C 66 C7 44 24 0C 34+mov     [esp+40Ch+SystemTime.wYear], 834h
004010E5 40C FF 15 14 40 40 00 call    ds:SystemTimeToFileTime ; Indirect Call Near Procedure
004010EB 404 6A 00             push    0               ; lpTimerName
004010ED 408 6A 00             push    0               ; bManualReset
004010EF 40C 6A 00             push    0               ; lpTimerAttributes
004010F1 410 FF 15 10 40 40 00 call    ds:CreateWaitableTimerA ; Indirect Call Near Procedure
004010F7 404 6A 00             push    0               ; fResume
004010F9 408 6A 00             push    0               ; lpArgToCompletionRoutine
004010FB 40C 6A 00             push    0               ; pfnCompletionRoutine
004010FD 410 8D 54 24 20       lea     edx, [esp+410h+DueTime] ; Load Effective Address
00401101 410 8B F0             mov     esi, eax
00401103 410 6A 00             push    0               ; lPeriod
00401105 414 52                push    edx             ; lpDueTime
00401106 418 56                push    esi             ; hTimer
00401107 41C FF 15 1C 40 40 00 call    ds:SetWaitableTimer ; Indirect Call Near Procedure
0040110D 404 6A FF             push    0FFFFFFFFh      ; dwMilliseconds
0040110F 408 56                push    esi             ; hHandle
00401110 40C FF 15 2C 40 40 00 call    ds:WaitForSingleObject ; Indirect Call Near Procedure
00401116 404 85 C0             test    eax, eax        ; Logical Compare
00401118 404 75 21             jnz     short loc_40113B ; Jump if Not Zero (ZF=0)
```

Figure 31: Time-related functions after Malservice is created.

Then, CreateWaitableTimerA is called which returns a handle to a waitable timer object. This object is used to synchronize one or more threads of a specified time interval. It is then followed by SetWaitTableTimer. The parameters passed are handle return from CreateWaitableTimerA, stored in EAX then moved into ESI. Most importantly, the value returned from SystemTimeToFileTime is loaded into EDX and pushed as the lpDueTime parameter. This parameter specifies the amount of time that should elapse before the waitable timer object signals waiting threads. Therefore, the waitable timer object will signal Malservice at 1 Jan 2100 at midnight.

We then see that WaitForSingleObject is called and passes a -1 into the dwMilliseconds parameter. This will have the function wait indefinitely until the waitable timer object that was created earlier signals. From these functions, this malware is a type of logic/time bomb that will occur at the turn of the century at midnight UTC.

To determine what happens on that date and time, we can look at Figure 32. We see the value of 0x14 (20 decimal) placed into ESI and that it is decremented in a loop, performing CreateThread with the offset StartAddress placed into the lpStartAddress parameter. As discussed in Question 4, the StartAddress function is used to open Internet Explorer 8 to connect to connect to www.malwareanalysisbook.com. We can then interpret this malware as being a logic/time bomb Denial of Service type malware against the malwareanalysisbook URL, so long as the malware infects enough machines that are still in use for 77 more years.

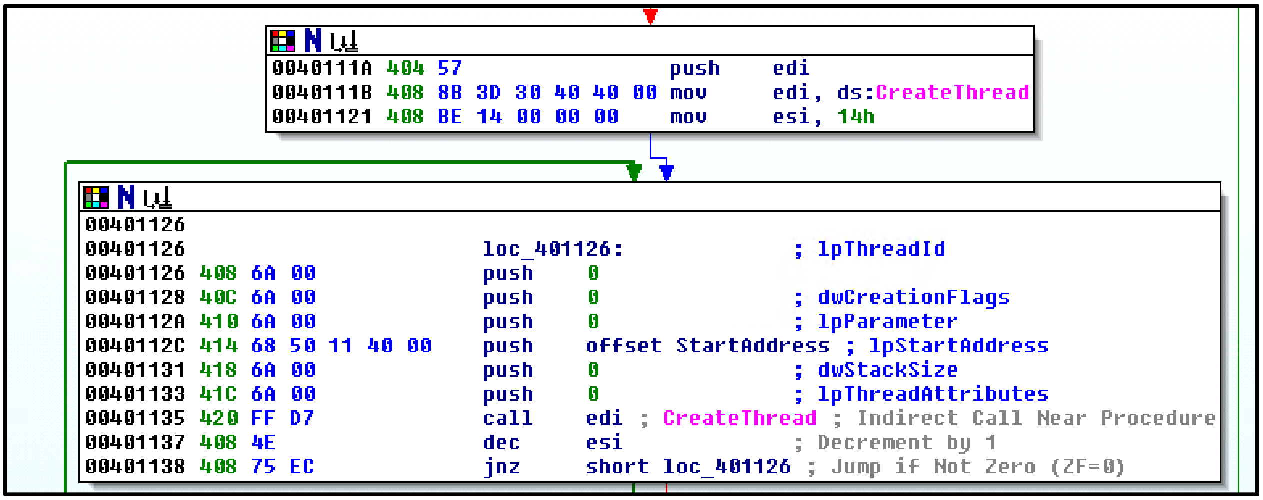


Figure 32: Loop creating 20 threads to malwareanalysisbook.com

LAB 7-1 Question 6

When will this program finish executing?

BLUF: Never.

In the previous questions, it was determined that this malware will continue to indefinitely execute until the time-based trigger of 1 January 2100 at midnight is reached, where it will then create 20 threads to www.malwareanalysisbook.com.

An attempt to patch the program was conducted within x32dbg in order to monitor traffic to the domain. The method of patching the program is to pass in desired values into the SystemTime structure prior to SystemTimeToFileTime is called. However, the malware does not provide fields for the SystemTime.wMinute field in the SystemTime structure, therefore it is extraordinarily time-consuming to test out certain variables. However, instead of passing the hex value for 2100 into the wYear field, the value of 0x7E7 was passed for the year 2023. The wHour portion of the code at address 0x0040100D5 has EDX moved into it. Simply setting a breakpoint at that address and modifying the EDX register value to whatever hour (in 24hr format) the program is to execute is the correct way of going about patching this program.

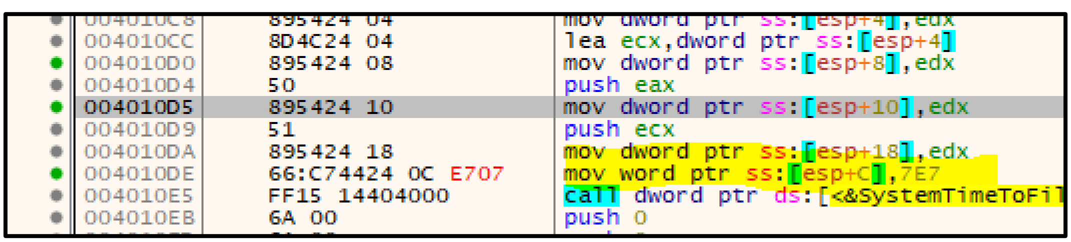


Figure 33: x32dbg editing SystemTime structure values.