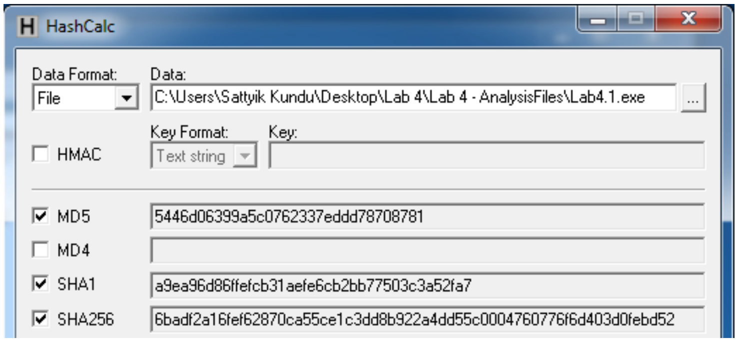
ISA 564: Lab 4 – Wireshark and Metasploit (by Sattyik Kundu)

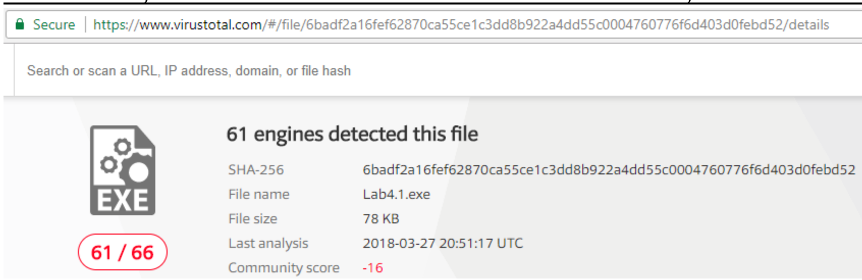
Task 1 Answers:

As stated in instructions for task 1, I need to first use *hashcalc* to get the MD5, SHA-1, and SHA-256 hashes of Lab4.1.exe binary:



Question 1.1 – Answers to Questions:

At this time, the most recent submission date is ***3/27/2018***; and the detection ratio is ***61/66*** as shown below:



Below is a *partial* list of the Detection results for Lab4.1.exe:

Detection | Details | Behavior | Community 2

Ad-Aware	Generic.PoisonIvy.1232DF25	AegisLab	Backdoor.W32.Poison.aec!c
AhnLab-V3	Trojan/Win32.Poison.R5433	ALYac	Generic.PoisonIvy.1232DF25
Antiy-AVL	Trojan[Backdoor]/Win32.Poison	Arcabit	Generic.PoisonIvy.1232DF25
Avast	Win32:Agent-AAGI [Trj]	AVG	Win32:Agent-AAGI [Trj]
Avira	TR/Dropper.Gen	AVware	Backdoor.Win32.Poison.Pg (v)
Baidu	Win32.Backdoor.Poison.a	BitDefender	Generic.PoisonIvy.1232DF25
Bkav	W32.eHeur.Virus02	CAT-QuickHeal	TrojanAPT.Poisonivy.D3
ClamAV	Win.Downloader.24465-1	CMC	Backdoor.Win32.Agent!O
Comodo	Backdoor.Win32.PoisonIvy.Gen	CrowdStrike Falcon	malicious_confidence_60% (D)
Cybereason	malicious.399a5c	Cylance	Unsafe
Cyren	W32/Agent.G.gen!Eldorado	DrWeb	Trojan.Proxy.3103
eGambit	RAT.PoisonIvy	Emsisoft	Generic.PoisonIvy.1232DF25 (B)

From the above long list of detection results, I found the most notable terms from above to be "PoisonIvy", "Poison", "W32", "Trojan", and "Backdoor". From inference, "**PoisonIvy**" and "**Poison**" are most likely the names of the malware. Next, "W32" likely means that the Lab4.1.exe file is a WIN 32 file type (a file that runs on Windows 32-bit systems). Finally, "Trojan" and "Backdoor" could describe the type or properties of the malware. For example: this malware could be a Trojan (disguised as a legitimate program) that is first downloaded onto the target host; then the malware can create a backdoor for the attacker to take over that target host.

Some additional information from under the other tabs of the VT analysis (shown below):

Under 'File detail', it states that the packer "PENinja" is used. Next, under 'Additional Information', the file has another name which is "SAST.exe". This maybe the real name of the binary when unpacked and successfully installed; the name "Lab4.1.exe" seems to only apply when the binary is packed and only for this lab. Finally, under "Comments" (where visitors can add their own comments about the malware), it is stated that this malware type is more specifically a RAT (Remote Access Trojan).

Under the ‘File details’ and ‘Additional Information’, there is additional information like file descriptions (like size and type), meta-data, various hashes, and etc. However, there is hardly anything else that could be easily understood regarding the actual function(s) of this malware file without additional research.

Question 1.2 – After some Google searching, I have found that some information corroborating my findings from earlier (the links below are my found sources) along with new information. To start off, I found that “PoisonIvy” (a.k.a. “Poison”) is indeed that malware’s name. The Poison Ivy malware injects itself into the browser process (possibly Internet Explorer since it is part of Windows) to gain outside access and attempt to evade the firewall if it disguises itself as part of the browser [1,2].

Additionally, the packer (PENinja) used on PoisonIvy enables it to look like a legitimate file/program so that it can pass anti-virus scans; packers are file compression programs (like ZIP, RAR, and etc). However, because packers can change the format and structure of the file/program during the compression, it can end up appearing as legitimate to anti-virus scans[3]. This in effect makes PoisonIvy a Trojan.

Once Poison Ivy is successfully embedded, a backdoor is established on the victim host. Using a RAT (Remote Access Toolkit), the attacking host can manipulate the target host though the established backdoor [1]. Variations of PoisonIvy can do the following including [4]:

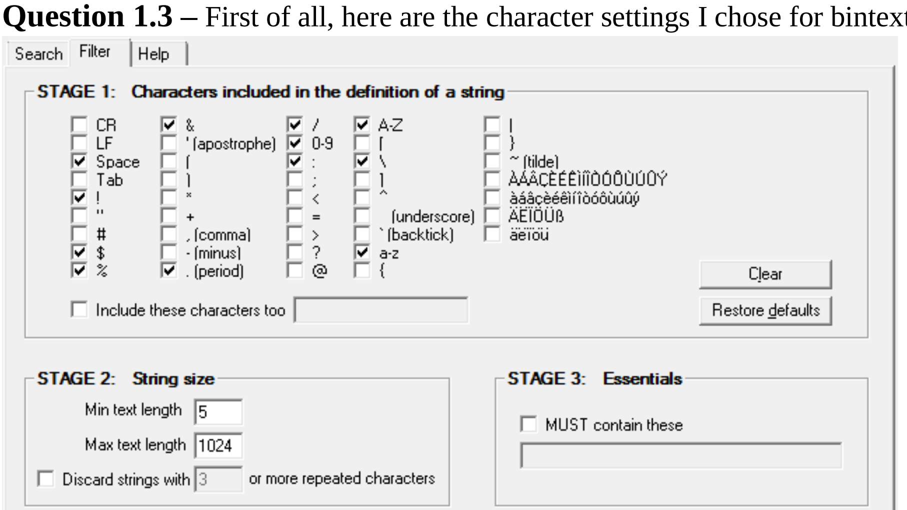
- Capture screen, audio, and webcam
- List active ports
- Log keystrokes
- Manage open windows
- Manage passwords
- Manage registry, processes, services, devices, and installed applications
- Perform remote shell
- Search files
- Update, restart, terminates itself
- Etc.

Lastly, the below sources [2,4] provide details with regards to how folders and registry keys are created/modified during the implementation of the malware.

<u>Sources</u>:

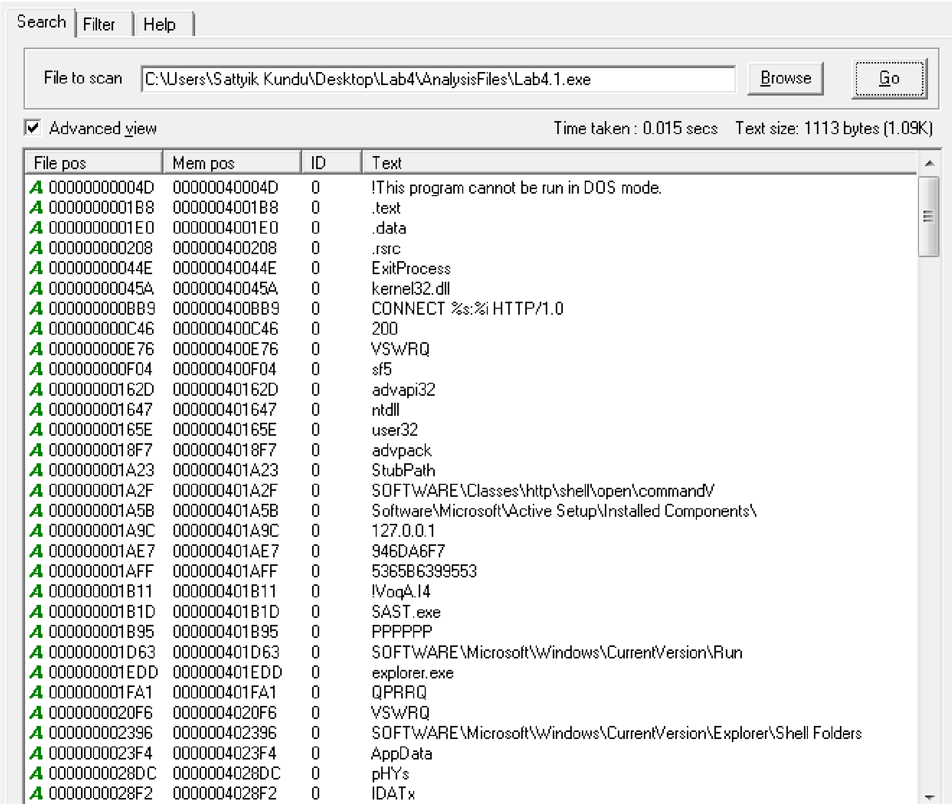
- [1] https://www.f-secure.com/v-descs/backdoor_w32_poisonivy.shtml
- [2]https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Backdoor%3AWin32%2FPoisonivy.I
- [3] https://www.welivesecurity.com/2008/10/27/an-introduction-to-packers/
- [4] https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/poisonivy

Question 1.3 – First of all, here are the character settings I chose for bintext:



I removed characters I think won’t be found in the needed results. For Registry Keys, URLs, and file names, the most important ones to highlight (to my understanding) are “0-9”, “a-z”, “A-Z”, “/”, “\”, “.”, and <space> as *most* Registry keys, URLs, and files names will make use of only these ones. The other ones I checked are ones I believe are likely and relevant enough to be found in a readable string.

Here is a screenshot of the (partial) search results which contain the needed information:



File pos	Mem pos	ID	Text
00000000004D	00000040004D	0	!This program cannot be run in DOS mode.
0000000001B8	0000004001B8	0	.text
0000000001E0	0000004001E0	0	.data
000000000208	000000400208	0	.rsrc
00000000044E	00000040044E	0	ExitProcess
00000000045A	00000040045A	0	kernel32.dll
000000000BB9	000000400BB9	0	CONNECT %s:%i HTTP/1.0
000000000C46	000000400C46	0	200
000000000E76	000000400E76	0	VSWRQ
000000000F04	000000400F04	0	sf5
00000000162D	00000040162D	0	advapi32
000000001647	000000401647	0	ntdll
00000000165E	00000040165E	0	user32
0000000018F7	0000004018F7	0	advpack
000000001A23	000000401A23	0	StubPath
000000001A2F	000000401A2F	0	SOFTWARE\Classes\http\shell\open\commandV
000000001A5B	000000401A5B	0	Software\Microsoft\Active Setup\Installed Components\
000000001A9C	000000401A9C	0	127.0.0.1
000000001AE7	000000401AE7	0	946DA6F7
000000001AFF	000000401AFF	0	5365B6399553
000000001B11	000000401B11	0	!VoqA.I4
000000001B1D	000000401B1D	0	SAST.exe
000000001B95	000000401B95	0	PPPPPP
000000001D63	000000401D63	0	SOFTWARE\Microsoft\Windows\CurrentVersion\Run
000000001EDD	000000401EDD	0	explorer.exe
000000001FA1	000000401FA1	0	QPRRQ
0000000020F6	0000004020F6	0	VSWRQ
000000002396	000000402396	0	SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
0000000023F4	0000004023F4	0	AppData
0000000028DC	0000004028DC	0	pHYs
0000000028F2	0000004028F2	0	IDATx

Above shows basically all the readable strings I found in the search result. After the above shown results were unintelligible strings that couldn't be used to describe registry keys, file paths, software, and etc. At the half-way point of the search results, the results repeat starting from the beginning.

From the above results, the relevant registry file paths are:

- SOFTWARE\Classes\http\shell\open\commandV
- Software\Microsoft\Active Setup\Installed Components\
- SOFTWARE\Microsoft\Windows\CurrentVersion\Run
- SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellFolders

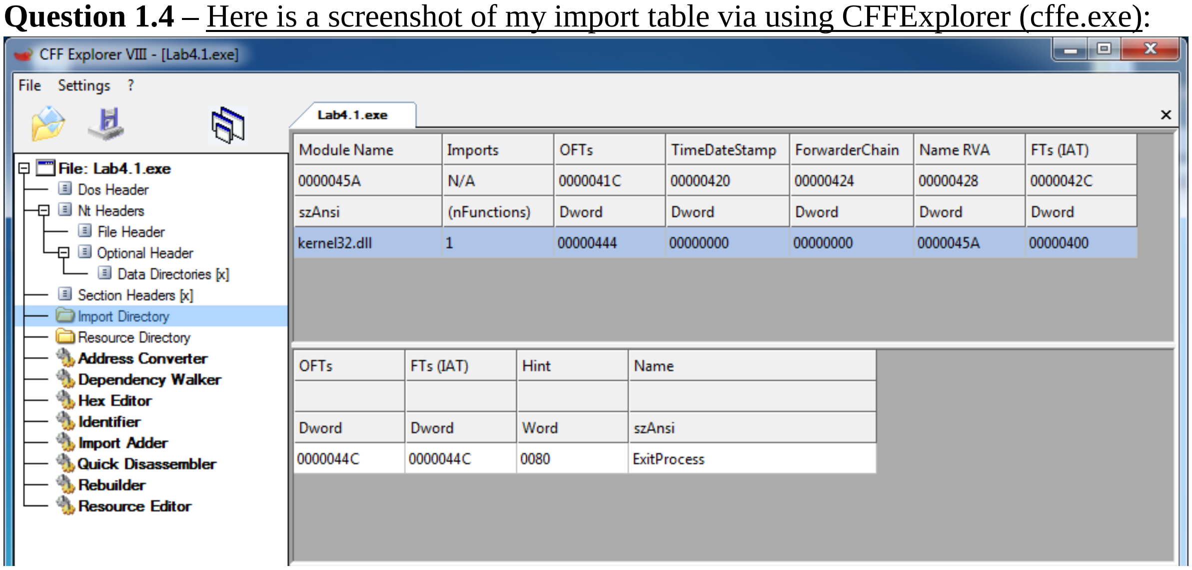
These strings suggest that Lab4.1.exe may add or alter values or software under these registry keys. Hence, this can enable an analyst to tracker where the changes occur.

There were no URLs or registry keys shown in the results. But the shown executables are:

- SAST.exe
- explorer.exe

SAST.exe is the name of the binary when unpacked and downloaded onto the host machine. explorer.exe refers to the malware being injected onto the Internet explorer browser process (as explained earlier).

Question 1.4 – Here is a screenshot of my import table via using CFFExplorer (cffe.exe):

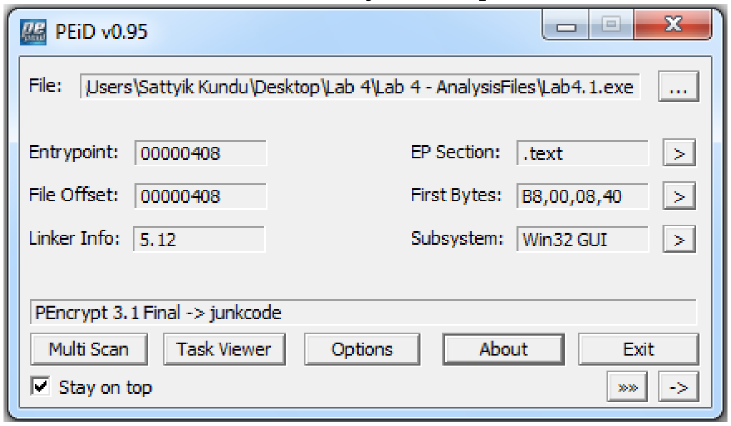


Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0000045A	N/A	0000041C	00000420	00000424	00000428	0000042C
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
kernel32.dll	1	00000444	00000000	00000000	0000045A	00000400

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
0000044C	0000044C	0080	ExitProcess

From the above screenshot, there seems to be only 1 function imported from the binary and according to the bottom table, it's called ExitProcess. The IAT (import address table) for the function points to the memory address of ExitProcess; which is 0x44C.

However, as stated, it should be impossible to only have 1 function. Looking into the binary via PEiD, it is shown that the binary is packed (as shown below). Because the binary is packed, it is only showing 1 function in the IAT table. If the binary was unpacked, it would then show more than 1 function.



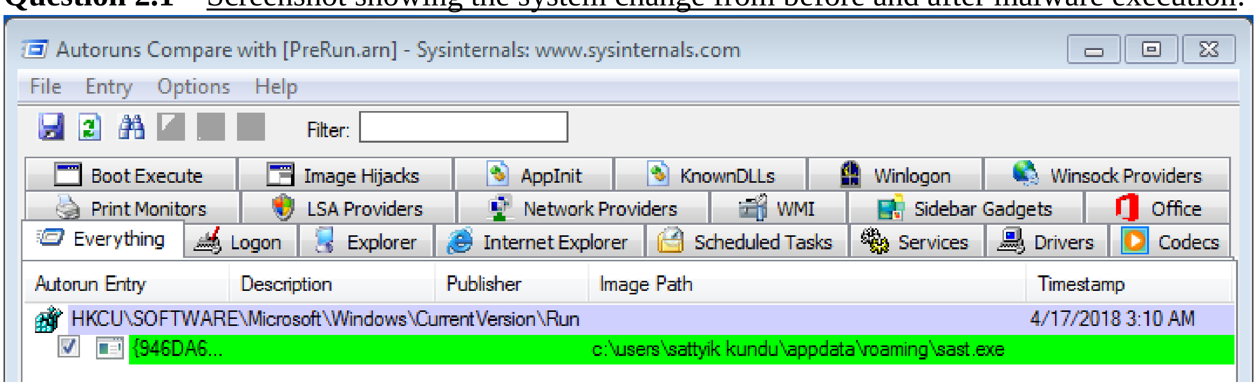
From above, it states that the packer "PEncrypt 3.1 Final" is used. This contradicts what was found earlier on Virus Total which stated that PENinja was used. Upon further Google research, I couldn't find anything on PENinja where as I was able to find information and downloadables for 'PEncrypt 3.1 Final'. I think this means that the packer name information from VirusTotal was incorrect.

Lastly, the fact only the Exit Process was shown on the IAT supports my earlier explanation on the use of packers to hide malware from anti-virus scans and other forms of detection; since only seeing the Exit Process would make Lab4.1.exe rather innocuous.

<u>Task 2 Answers</u>:

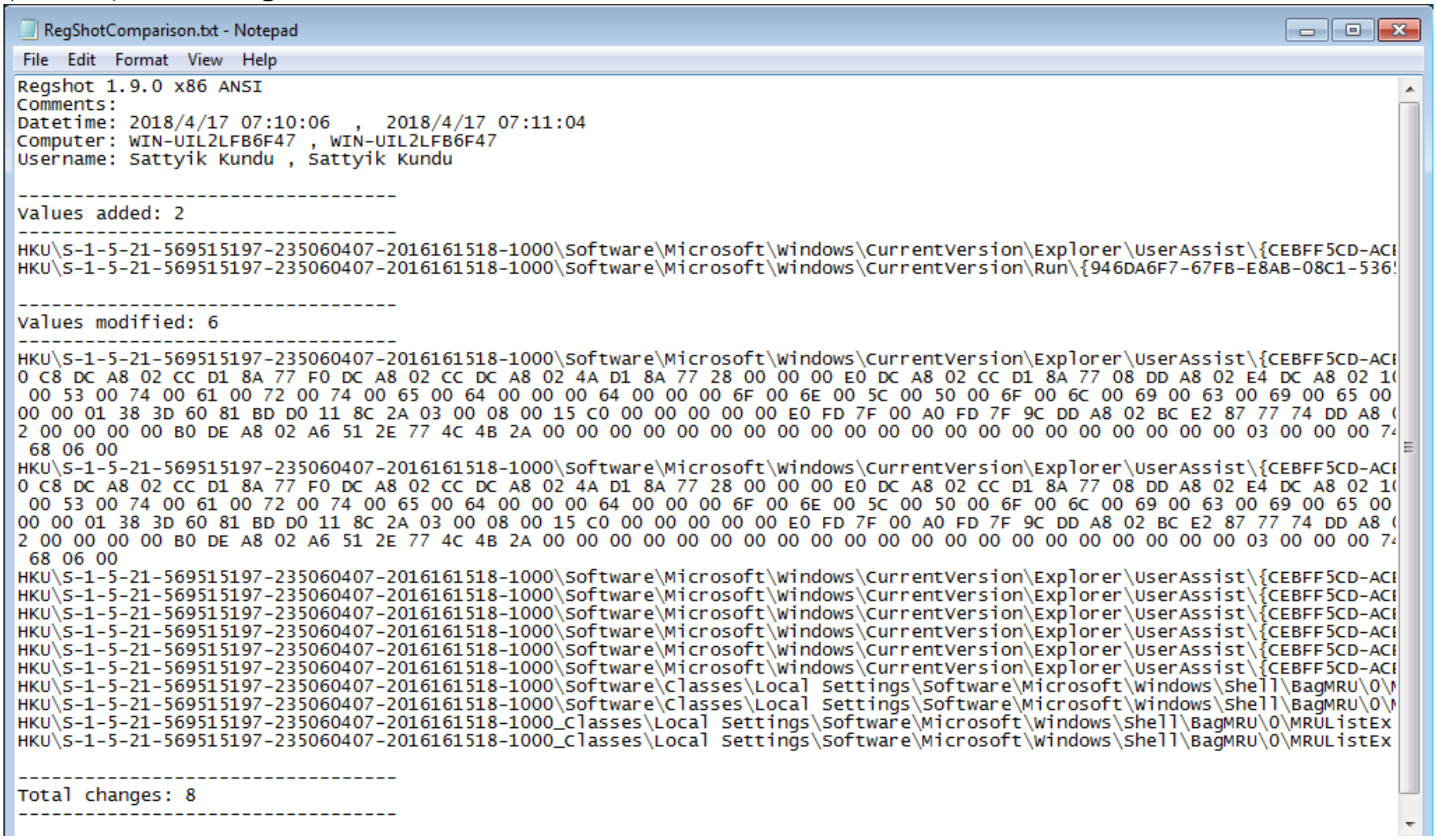
Task 2 requires a significant series of steps utilizing Autoruns, Procmon, and Regshot to compare Windows System settings from before and after Lab4.1.exe is executed (using Administrator privileges). Also, VMware system snapshots are used to "reset" the system to a stored point before the malware was run.

Question 2.1 – <u>Screenshot showing the system change from before and after malware execution</u>:



The above screenshot shows the comparison between the current and pre-execution snapshot. What was found from the comparison was that the executed malware(Lab4.1.exe) inserted the persistence mechanism sast.exe under the register key "HKCU\SOFTWARE\Microsoft\Windows\Current\Version\Run". This registry key tells Windows which programs to run when a specific user logs onto the computer. When the target machine starts up, the sast.exe binary would start up and enable the attacker to utilize the Poison Ivy backdoor to compromise the target machine.

Question 2.2 – Screenshots of the below file shows the changes in the system from before (1st shot) and after (2nd shot) the running of Lab4.1.exe:



```
RegShotComparison.txt - Notepad
File  Edit  Format  View  Help
Regshot 1.9.0 x86 ANSI
Comments:
Datetime: 2018/4/17 07:10:06  ,  2018/4/17 07:11:04
Computer: WIN-UIL2LFB6F47 , WIN-UIL2LFB6F47
Username: Sattyik Kundu , Sattyik Kundu

----------------------------------
Values added: 2
----------------------------------
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Run\{946DA6F7-67FB-E8AB-08C1-536

----------------------------------
Values modified: 6
----------------------------------
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
0 C8 DC A8 02 CC D1 8A 77 F0 DC A8 02 CC DC A8 02 4A D1 8A 77 28 00 00 00 E0 DC A8 02 CC D1 8A 77 08 DD A8 02 E4 DC A8 02 1
 00 53 00 74 00 61 00 72 00 74 00 65 00 64 00 00 00 64 00 00 00 6F 00 6E 00 5C 00 50 00 6F 00 6C 00 69 00 63 00 69 00 65 00
00 00 01 38 3D 60 81 BD D0 11 8C 2A 03 00 08 00 15 C0 00 00 00 00 00 E0 FD 7F 00 A0 FD 7F 9C DD A8 02 BC E2 87 77 74 DD A8
2 00 00 00 00 B0 DE A8 02 A6 51 2E 77 4C 4B 2A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 00 7
 68 06 00
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
0 C8 DC A8 02 CC D1 8A 77 F0 DC A8 02 CC DC A8 02 4A D1 8A 77 28 00 00 00 E0 DC A8 02 CC D1 8A 77 08 DD A8 02 E4 DC A8 02 1
 00 53 00 74 00 61 00 72 00 74 00 65 00 64 00 00 00 64 00 00 00 6F 00 6E 00 5C 00 50 00 6F 00 6C 00 69 00 63 00 69 00 65 00
00 00 01 38 3D 60 81 BD D0 11 8C 2A 03 00 08 00 15 C0 00 00 00 00 00 E0 FD 7F 00 A0 FD 7F 9C DD A8 02 BC E2 87 77 74 DD A8
2 00 00 00 00 B0 DE A8 02 A6 51 2E 77 4C 4B 2A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 00 7
 68 06 00
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\0\M
HKU\S-1-5-21-569515197-235060407-2016161518-1000\Software\Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\0\M
HKU\S-1-5-21-569515197-235060407-2016161518-1000_Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\0\MRUListEx
HKU\S-1-5-21-569515197-235060407-2016161518-1000_Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\0\MRUListEx

----------------------------------
Total changes: 8
----------------------------------
```

From the the above screenshot, there are 2 new registry key values added plus changes in 6 old key registry locations. The 'S-1-5-21-xxxxxxx-1000' is the unique SID(security identifier) for each Windows computer user account. When removed from the HKU (HKEY_USER) registry path, the above paths shorten to the actual registry key paths.

Hence, for the added values, the actual registry key paths are:

- HKU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\Count\P:\Hfref\Fngglvx Xhaqh\Qrfxgbc\Gbbyf\NanylfvfSvyrf\Yno4.1.rkr
- HKU\Software\Microsoft\Windows\CurrentVersion\Run\{946DA6F7-67FB-E8AB-08C1-5365B6399553}

The (shortened) actual HKU paths for the modified paths are:

- HKU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\Count\HRZR_PGYFRFFVBA
- HKU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\Count\{S38OS404-1Q43-42S2-9305-67QR0O28SP23}\rkcybere.rkr
- HKU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\Count\P:\Hfref\Fngglvx Xhaqh\Qrfxgbc\Gbbyf\cebpzba.rkr
- HKU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\Count\P:\Hfref\Fngglvx Xhaqh\Qrfxgbc\Gbbyf\ertfubg.rkr
- HKU\Software\Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\0\MRUListEx

In the last modified registry key file path, "S-1-5-21-xxxxxxx-1000_Classes" becomes "\Software\Classes"; hence:

- HKU\Software\Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\0\MRUListEx

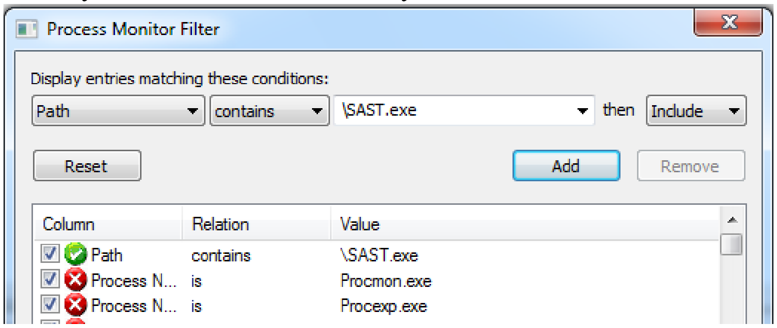
From the above list, most HKEYs contain "Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\". The registry keys at around this location are called UserAssist registry keys. UserAssist entries under these keys contain information about exe files and links that are opened frequently. If changes were made so this registry key file path can be accessed via PoisonIvy backdoor, the attacker can use the remote access toolkit to spy on what exe files and links are being run on the target machine.

Next, the registry key containing “\Software\Microsoft\Windows\CurrentVersion\Run\” refers to programs that are started up whenever a user logs in. This registry key path seems most useful to the attacker because any persistence mechanism put here (like malware) can automatically startup whenever the user logs in. The SAST.exe should be enabled though this registry key path (as already shown via Autoruns comparison).

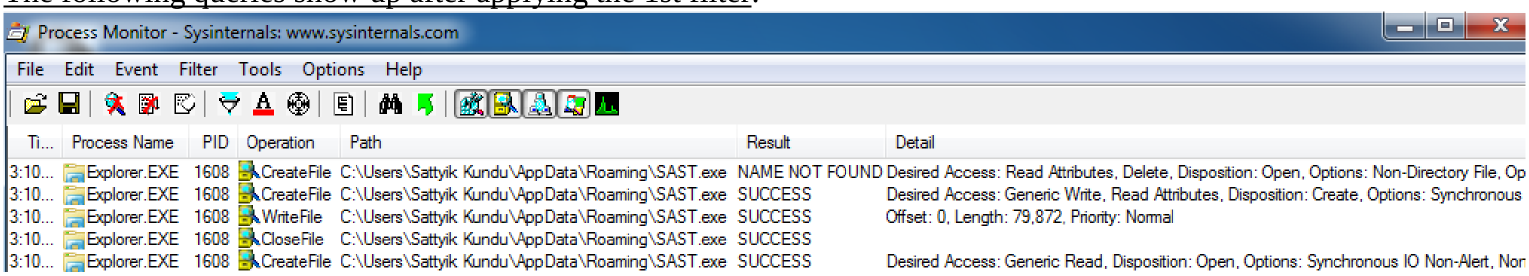
Finally, the last 2 registry keys (from the location paths containing “Local Settings\Software\Microsoft\Windows\Shell\ BagMRU\0\ MRUListEx”) refer to Shellbags which maintain and store information regarding size, view, icon, and position of a folder when using Explorer. The attacker can view this information; but these registry file path areas are not useful for placing a persistence mechanism in.

Question 2.3 – After finding various registry file paths in the previous question, I now need to use these given file paths to find/confirm the location of the persistence mechanism SAST.exe.

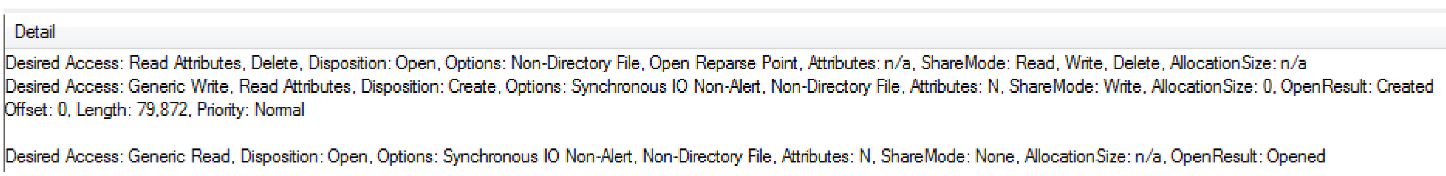
<u>For my first filter, I will directly search for “\SAST.exe” from the file path</u>:



<u>The following queries show up after applying the 1st filter</u>:



…………Entire Details shown below……….

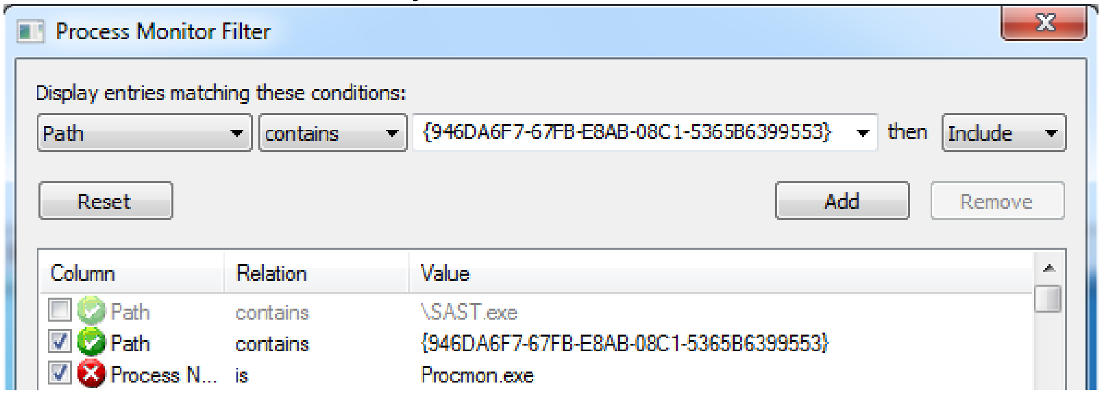


The 1st screenshot shows the output (albeit details are cutoff). The 2nd screenshot shows the full detail for each of the 5 entries. From the above entries, it is seen that the persistence mechanism of SAST.exe is found in the exact same path as was found in Autoruns.

The “…\AppData\Roaming” folder path is used to store data that would “roam” with a user account. Since SAST.exe is within this folder path, SAST.exe would follow the user account even if the owner opened his/her infected account on another computer within the Domain (a computer network where all accounts and devices are registered with a central database under a domain controller). This benefits the attacker in the sense that it makes it harder for the owner to sidestep the effects of SAST.exe by opening the infected account on another machine. Hence, this is the reason that SAST.exe was placed in the “…\AppData\Roaming” folder path by the attacker.

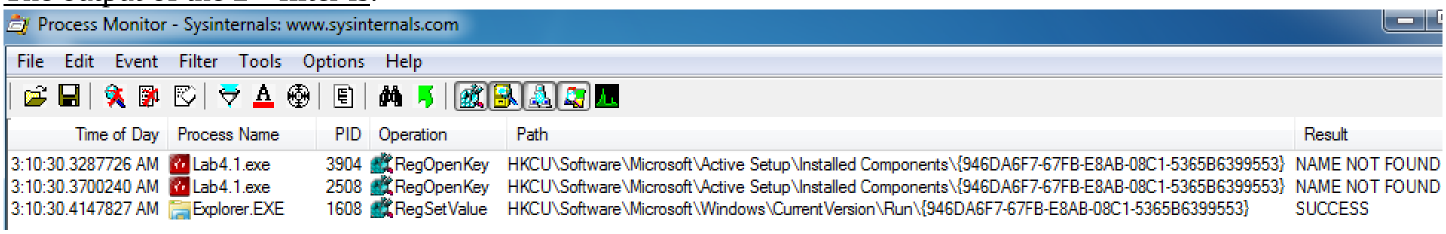
The 2nd required filter is to find the related registry key for where SAST.exe was located. The means searching through the changed registry keys found in Question 2.2(via regshot).

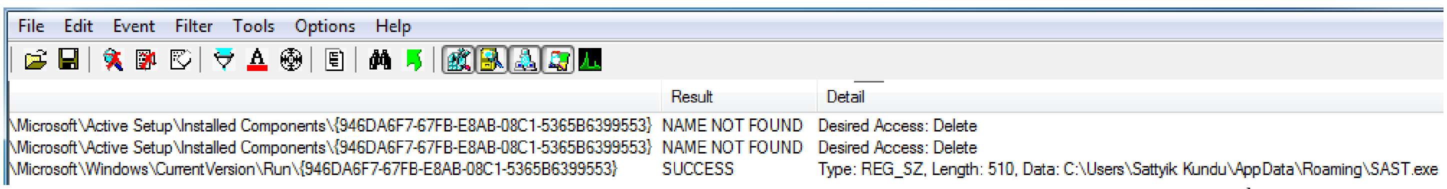
Here is the filter that I eventually used:



I decided to use the "{}" segment of the registry key path HKU\Software\Microsoft\Windows\CurrentVersion\Run\{946DA6F7-67FB-E8AB-08C1-5365B6399553} which is the GUID (globally unique identifier) which represents a globally unique Windows object.

The output of the 2nd filter is:





Seen above is the 1st screenshot which shows the output of the filter (albeit cutoff). The above 2nd screenshot shows the entire Detail that was cutoff in the 1st screenshot.

From above, in the 3rd row, it is shown that under the registry key "**HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{946DA6F7-67FB-E8AB-08C1-5365B6399553}**", the persistence mechanism SAST.exe will activate from its folder path (shown in the 3rd row's Detail), "**C:\Users\Sattyik Kundu\AppData\Roaming\SAST.exe**", whenever the computer starts up. This matches what was shown in the Autorun comparison screenshot under Question 2.1.

Question 2.4 – With regards to how my static analysis helped with task 2, the first thing was that I found out that the real name of the Lab4.1.exe binary was SAST.exe (since the 'Lab4.1.exe' name only has value in this class). When using Autorun, Procmon, and Regshot, I repeatedly have the binary being read as SAST.exe. If I thought I had to look for Lab4.1.exe, I wouldn't have found it and finished the lab.

The other major thing that helped me in Task 2 (from my Task 1 static analysis) was using the output strings from bintext as reference. From the 4 below strings:

- SOFTWARE\Classes\http\shell\open\commandV
- Software\Microsoft\Active Setup\Installed Components\
- SOFTWARE\Microsoft\Windows\CurrentVersion\Run
- SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellFolders

I was able to narrow down that the persistence mechanism(SAST.exe) file path would be related to one of these registry keys. As it turned out, the SAST.exe and its file path was eventually found under registry key "HKCU\Software\Microsoft\Windows\CurrentVersion\Run\{946DA6F7-67FB-E8AB-08C1-5365B6399553}" which contains the "SOFTWARE\Microsoft\Windows\CurrentVersion\Run" string from above.
The answers I got for questions 2.1, 2.2, & 2.3 supported this finding in the end.

The one time so far I saw anything unexpected was during Question 2.2 where I had to take a Regshot comparison of before and after executing Lab4.1.exe. Over multiple attempts, I noticed that the number of changes in the comparison output changed dramatically between each run. Sometimes, there were few total changes like 5-10; other times, there were as many as 26-40 total changes! Sometimes, I didn't get the needed change in "\Software\Microsoft\Windows\CurrentVersion\Run\". After a couple rewinds of the VM snapshot (including once rebuilding the Windows 7 VM and redoing the setup instructions in the rubric for good measure); I finally got a decent Regshot comparison output screenshot (as shown in Question 2.2) which included the all important "\Software\Microsoft\Windows\CurrentVersion\Run\".

Task 3 Answers:

Question 3.1 – After starting from 3 different call functions from within the main function(sub_401460), they all have a same call function pointing to this function, sub_40162F, which doesn't continue further. Thus, **sub_40162F** is most likely the delete function. Reading through the function, the Windows API call functions within this function are:

- **call GetModuleFileNameA**: this call function retrieves the full path and file name for the file during the current process. One of its inputs is *hmodule* which comes from the "sub esp, 234h" instruction within sub_40162F.
- **call strcat**: this call function concatenates 2 string operands. Although the 2 operands were not written together with this call function, I believe that this function maybe invoking the strings from "mov dword ptr [eax], 20632F20h" and "mov dword ptr [eax+4], 206C6564h" within sub_40162F.
- **call GetEnvironmentalVariableA**: reads the value of one of the computer's environment variables. The value of *lpName* is placed into the string buffer passed as the *lpBuffer* parameter. In the delete function, the *lpName* variable exists from the instruction "sub esp, 0ch" within sub_40162F.
- **call ShellExecuteA**: Given inputs, this call function performs an arbitrary operation on a specified file. This is arguably what sets off the self-deletion of the Lab4.1.exe binary. From the delete function code, the *hwnd* parameter is from the "sub esp, 0ch" instruction (the previous one is for the previous call) that is invoked after the call GetEnvironmentalVariableA. Additionally, although not explicitly stated, the input file should be this program's name (which is "Lab4.1.exe").

Finally, the 'A' at the end of three of the above call functions shows up because these function names are its ANSI names.

Question 3.2 – The three functions that can call the delete function are:

- **sub_401523**: The jump instruction at the end of the top/first block is "jz short loc_40153B". If the previous instruction, "cmp [ebp+arg_0], 3", equals zero, the jump will go to loc_40153B. The "call sub_40162F" instruction is in the other branch block aside from loc_40153B. So "cmp [ebp+arg_0], 3" has to be remade so it will always be equal to 0 to avoid going to "call sub_40162F" which leads to the delete function. For example, "cmp ebp,ebp" will always yield 0.
- **sub_401542**: Here, the jump instruction at the end of the top/first block is "jz short loc_40155D". If the previous instruction, "test eax, eax", equals zero, the jump will go to loc_40155D. The other branch block has the "call sub_40162F" instruction. So, "test eax, eax" has to be remade so it will always be equal to 0 to avoid going to "call sub_40162F" and go down to path leading to the program's end. Here, using "cmp eax,eax" will yield zero.
- **sub_401564**: Out of the many cascading blocks in this function, there is one called loc_401623 which contains the "call sub_40162F" instruction. All the blocks that can jump to it uses a "jnz" jump instruction which jumps to the prescribed loc_401623 when previous "test" or "cmp" instruction equals 1. One way to avoid going to loc_401623 and its "call sub_40162F" instruction is to first create a "cmp eax, eax" instruction(in the 1st code block) and then turn the following "jnz" jump instruction to "jz" to always force a jump all the way down to the loc_401623 block. Then, "call sub_40162F" can be disabled with NOP instructions. Finally, the "mov eax,0" can be changed to "mov eax,1" to ensure function completion.

Question 3.3 – Regarding conditions on how the complete the program**,** sub_401523 says that three command line arguments are needed to successfully execute the program:

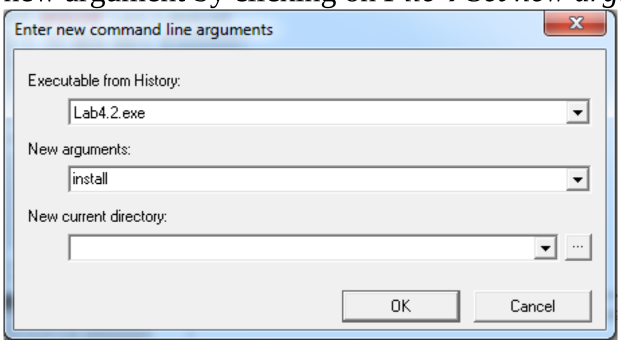
- 1st argument is to disable the debugger ("no debugger")
- 2nd argument is "install"
- 3rd argument is the password (which I don't know currently)

In practice, if the codes of line can be patched (like in later Task 4), inputs can generally be avoided. If the program is successfully executed, the call "CopyFileA" command within the Start function(sub_401460) will cause the program to copy itself into all users' startup folders.

Task 4 Answers:

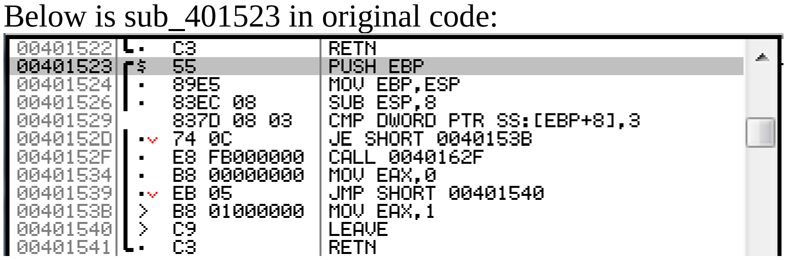
In this last task, OllyDbg needs to use patches (based on finding from Task 3) to get Lab4.2.exe to successfully run without hitting the deletion function sub_40162F.

Question 4.1 – I first start OllyDbg and then open Lab4.2.exe into OllyDbg. The first thing I need to do is set a new argument by clicking on *File→Set new arguments*:

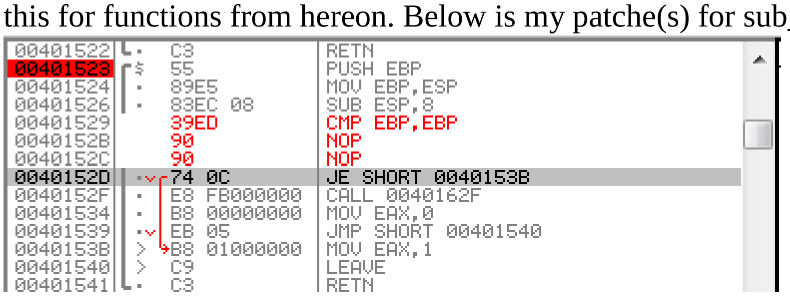


I set a new argument "install" which I will need later for executing sub_401564.

The next thing do is make patches in functions sub_401523, sub_401542, and sub_401564.
Below is sub_401523 in original code:



I first start with making a breakpoint at the start of function sub_401523 at assembly line 00401523; I will do this for functions from hereon. Below is my patche(s) for sub_401523:



Above, I changed the code in line 0040152D to cmp ebp,ebp (along with NOPS to preserve memory size). This makes it so JE SHORT 004153B is forced to jump to where MOV EAX,1 takes to skip CALL 0040162F and complete the function.

For sub_401542, here is the unchanged code:

```
00401542  $  55              PUSH EBP
00401543  .  89E5            MOV EBP,ESP
00401545  .  83EC 08         SUB ESP,8
00401548  .  E8 D3270000     CALL <JMP.&KERNEL32.IsDebuggerPresent>
0040154D     85C0            TEST EAX,EAX
0040154F  .v 74 0C           JZ SHORT 0040155D
00401551  .  E8 D9000000     CALL 0040162F
00401556  .  B8 00000000     MOV EAX,0
0040155B  .v EB 05           JMP SHORT 00401562
0040155D  >  B8 01000000     MOV EAX,1
00401562  >  C9              LEAVE
00401563  .  C3              RETN
```

Now, here is the patched code for function sub_401542:

```
00401542  $  55              PUSH EBP
00401543  .  89E5            MOV EBP,ESP
00401545  .  83EC 08         SUB ESP,8
00401548  .  E8 D3270000     CALL <JMP.&KERNEL32.IsDebuggerPresent>
0040154D     39C0            CMP EAX,EAX
0040154F  .v 74 0C           JZ SHORT 0040155D
00401551  .  E8 D9000000     CALL 0040162F
00401556  .  B8 00000000     MOV EAX,0
0040155B  .v EB 05           JMP SHORT 00401562
0040155D  >  B8 01000000     MOV EAX,1
00401562  >  C9              LEAVE
00401563  .  C3              RETN
```

Here, I changed TEST EAX,EAX to CMP EAX,EAX. Similar to before, this is meant to set the zero flag so JZ SHORT 0040155D will be forced to jump to MOV EAX,1 so the function can complete.

Here is the first part of the unchanged code for function sub_401564(the entire code is too big to show; so I'll show only the relevant parts):

```
00401564  $  55              PUSH EBP
00401565  .  89E5            MOV EBP,ESP
00401567  .  83EC 28         SUB ESP,28
0040156A  .  C745 F4 4400    MOV DWORD PTR SS:[LOCAL.3],44
00401571  .  C74424 04 AD    MOV DWORD PTR SS:[LOCAL.9],OFFSET 004050AD
00401579  .  8B45 08         MOV EAX,DWORD PTR SS:[ARG.1]
0040157C  .  890424          MOV DWORD PTR SS:[LOCAL.10],EAX
0040157F  .  E8 C4260000     CALL <JMP.&msvcrt.strcmp>
00401584     85C0            TEST EAX,EAX
00401586   v 0F85 97000000   JNZ 00401623
0040158C  .  8B45 0C         MOV EAX,DWORD PTR SS:[ARG.2]
0040158F  .  890424          MOV DWORD PTR SS:[LOCAL.10],EAX
00401592  .  E8 A1260000     CALL <JMP.&msvcrt.strlen>
```

Here are the patches needed to be added to this part of sub_401564 first:

```
00401564  $  55              PUSH EBP
00401565  .  89E5            MOV EBP,ESP
00401567  .  83EC 28         SUB ESP,28
0040156A  .  C745 F4 4400    MOV DWORD PTR SS:[LOCAL.3],44
00401571  .  C74424 04 AD    MOV DWORD PTR SS:[LOCAL.9],OFFSET 004050AD     string2 => "install"
00401579  .  8B45 08         MOV EAX,DWORD PTR SS:[ARG.1]
0040157C  .  890424          MOV DWORD PTR SS:[LOCAL.10],EAX                string1 => [ARG.1]
0040157F  .  E8 C4260000     CALL <JMP.&msvcrt.strcmp>                      MSVCRT.strcmp
00401584     39C0            CMP EAX,EAX
00401586   v 0F84 97000000   JZ 00401623
0040158C  .  8B45 0C         MOV EAX,DWORD PTR SS:[ARG.2]
0040158F  .  890424          MOV DWORD PTR SS:[LOCAL.10],EAX                string => [ARG.2]
00401592  .  E8 A1260000     CALL <JMP.&msvcrt.strlen>                      MSVCRT.strlen
00401597  .  83F8 06         CMP EAX,6
0040159A  .v 0F85 83000000   JNE 00401623
```

Above, I first changed TEST EAX, EAX to CMP EAX, EAX to force a zero flag. Then I changed JNZ 00401623 to JZ 00401623 to automatically jump in response to the zero flag. Additionally, at the right, it can be seen that input "install" is wanted by the call function at 00401592. This was why I had to put in install as an argument at the beginning (the code won't run without it).

Next, there is another code portion of sub_401564 that needs to be patched to complete the function. Here is that code section without the patches:

```
00401606  .  8B45 0C         MOV EAX,DWORD PTR SS:[ARG.2]
00401609  .  83C0 05         ADD EAX,5
0040160C  .  0FB600          MOVZX EAX,BYTE PTR DS:[EAX]
0040160F  .  0FBEC0          MOVSX EAX,AL
00401612  .  8B55 F4         MOV EDX,DWORD PTR SS:[LOCAL.3]
00401615  .  83C2 0A         ADD EDX,0A
00401618  .  39D0            CMP EAX,EDX
0040161A  .v 75 07           JNE SHORT 00401623
0040161C  .  B8 01000000     MOV EAX,1
00401621  .v EB 0A           JMP SHORT 0040162D
00401623     E8 07000000     CALL 0040162F
00401628     B8 00000000     MOV EAX,0
0040162D  >  C9              LEAVE
0040162E  .  C3              RETN
```

The red arrow above is pointing from the jump(JZ) patch I created earlier. Albeit concerning that it points to the deletion function, these patches will resolve that:

```
00401606  .  8B45 0C       MOV EAX,DWORD PTR SS:[ARG.2]
00401609  .  83C0 05       ADD EAX,5
0040160C  .  0FB600        MOVZX EAX,BYTE PTR DS:[EAX]
0040160F  .  0FBEC0        MOVSX EAX,AL
00401612  .  8B55 F4       MOV EDX,DWORD PTR SS:[LOCAL.3]
00401615  .  83C2 0A       ADD EDX,0A
00401618  .  39D0          CMP EAX,EDX
0040161A  .v 75 07         JNE SHORT 00401623
0040161C  .  B8 01000000   MOV EAX,1
00401621  .v EB 0A         JMP SHORT 0040162D
00401623  >  90            NOP
00401624     90            NOP
00401625     90            NOP
00401626     90            NOP
00401627     90            NOP
00401628     B8 01000000   MOV EAX,1
0040162D  >  C9            LEAVE
0040162E  .  C3            RETN
```

Above, the deletion function was turned into a series of NOP(whilst preserving the memory size and space). Also, MOV EAX,0 has been changed to MOV EAX,1 because the 1 value tells the program that this function is running to completion. With this, the patches for sub_401564 are complete.

The last place to put patches in within the Start function(sub_401560). Besides avoiding the deletion functions and completing the program; the program also needs to insert a persistence mechanism at a certain folder location. Do to this, the program has to reach an assembly command called "call CopyFileA" within the Start function. These last patches are for doing that before running the program.

Much of the Start function consists of several blocks of code where the program execution can be traversed via jump commands. Here is where the Start function begins (at instruction 00401460):

```
00401460  $  8D4C24 04     LEA ECX,[ARG.1]
00401464  .  83E4 F0       AND ESP,FFFFFFF0                       DQWORD (16.-byte) stack alignment
00401467  .  FF71 FC       PUSH DWORD PTR DS:[ECX-4]
0040146A  .  55            PUSH EBP
0040146B  .  89E5          MOV EBP,ESP
0040146D  .  57            PUSH EDI
0040146E  .  56            PUSH ESI
0040146F  .  53            PUSH EBX
00401470  .  51            PUSH ECX
00401471  .  83EC 78       SUB ESP,78
00401474  .  894D 94       MOV DWORD PTR SS:[LOCAL.28],ECX
00401477  .  E8 84070000   CALL 00401C00
0040147C  .  8D45 9F       LEA EAX,[LOCAL.26+3]
0040147F  .  BB 64504000   MOV EBX,OFFSET 00405064                ASCII "C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup\Persist.exe"
00401484  .  BA 49000000   MOV EDX,49
00401489  .  8B0B          MOV ECX,DWORD PTR DS:[EBX]
0040148B  .  8908          MOV DWORD PTR DS:[EAX],ECX
0040148D  .  8B4C13 FC     MOV ECX,DWORD PTR DS:[EDX+EBX-4]
00401491  .  894C10 FC     MOV DWORD PTR DS:[EDX+EAX-4],ECX
```

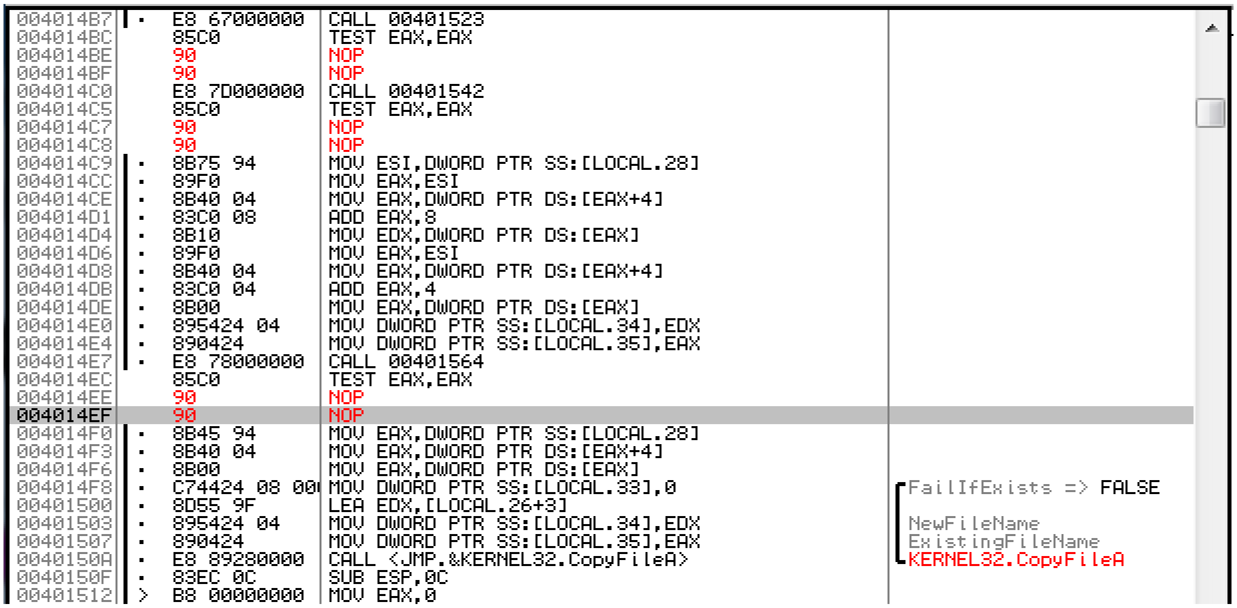
Above on the right, it is seen what file path the persistence mechanism should eventually be in.

Here is segment of the Start code where multiple patches need to be made:

```
004014B4  .  890424        MOV DWORD PTR SS:[LOCAL.35],EAX
004014B7  .  E8 67000000   CALL 00401523
004014BC     85C0          TEST EAX,EAX
004014BE   v 74 52         JZ SHORT 00401512
004014C0     E8 7D000000   CALL 00401542
004014C5     85C0          TEST EAX,EAX
004014C7   v 74 49         JZ SHORT 00401512
004014C9  .  8B75 94       MOV ESI,DWORD PTR SS:[LOCAL.28]
004014CC  .  89F0          MOV EAX,ESI
004014CE  .  8B40 04       MOV EAX,DWORD PTR DS:[EAX+4]
004014D1  .  83C0 08       ADD EAX,8
004014D4  .  8B10          MOV EDX,DWORD PTR DS:[EAX]
004014D6  .  89F0          MOV EAX,ESI
004014D8  .  8B40 04       MOV EAX,DWORD PTR DS:[EAX+4]
004014DB  .  83C0 04       ADD EAX,4
004014DE  .  8B00          MOV EAX,DWORD PTR DS:[EAX]
004014E0  .  895424 04     MOV DWORD PTR SS:[LOCAL.34],EDX
004014E4  .  890424        MOV DWORD PTR SS:[LOCAL.35],EAX
004014E7  .  E8 78000000   CALL 00401564
004014EC     85C0          TEST EAX,EAX
004014EE   v 74 22         JZ SHORT 00401512
004014F0  .  8B45 94       MOV EAX,DWORD PTR SS:[LOCAL.28]
004014F3  .  8B40 04       MOV EAX,DWORD PTR DS:[EAX+4]
004014F6  .  8B00          MOV EAX,DWORD PTR DS:[EAX]
004014F8  .  C74424 08 00( MOV DWORD PTR SS:[LOCAL.33],0          FailIfExists => FALSE
00401500  .  8D55 9F       LEA EDX,[LOCAL.26+3]
00401503  .  895424 04     MOV DWORD PTR SS:[LOCAL.34],EDX        NewFileName
00401507  .  890424        MOV DWORD PTR SS:[LOCAL.35],EAX        ExistingFileName
0040150A  .  E8 89280000   CALL <JMP.&KERNEL32.CopyFileA>         KERNEL32.CopyFileA
0040150F  .  83EC 0C       SUB ESP,0C
00401512  >  B8 00000000   MOV EAX,0
00401517  .  8D65 F0       LEA ESP,[LOCAL.5]
0040151A  .  59            POP ECX
0040151B  .  5B            POP EBX
0040151C  .  5E            POP ESI
```

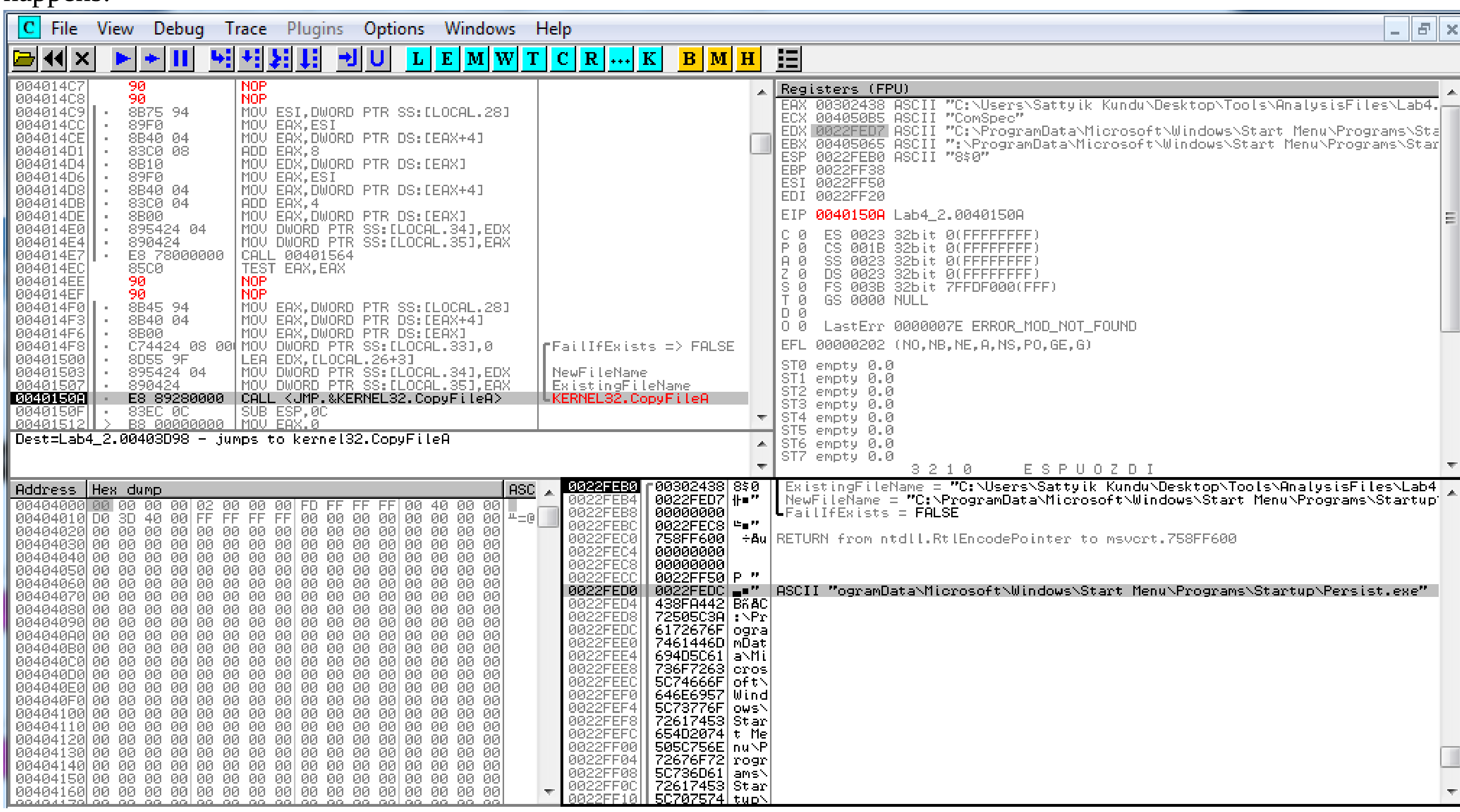
Above in three instruction values (004014BE, 004014C7, and 004014EE), those JZ SHORT 00401512 instructions need to be overwritten. If any of those jump, then the program will automatically jump to the end of the Start function and miss "call CopyFileA". One way to do it is to overwrite those three JZ instructions with NOP commands.

Here are the patches:



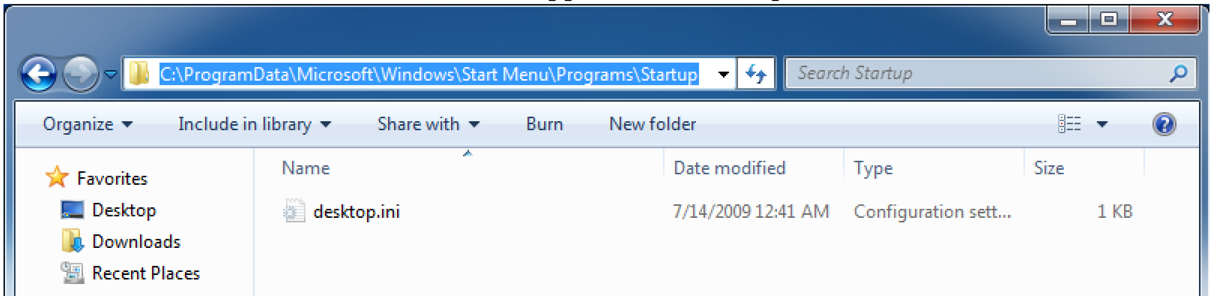
Above you can see I put NOP instructions(with memory size maintained) to overwrite those JZ instructions and force the Start function to eventually call CopyFileA. Now I can truly run the program.

I click on Stepover(F8) until execute the instruction for CopyFileA. At this point, some **VERY WRONG** happens:



In the top-left pane, I have already reached and executed the call CopyFileA function. However, if you look at the bottom-right pan, you can see that grey highlighted text which shows the file path of the persistence function is missing the “Pr” from “ProgramData\”. This means because the executing program WILL NOT see the correct file path as it is misspelled; and it won’t be able to put the Persist.exe file in the correct place.

Below shows where the Persist.exe was supposed to show up but didn't:



Alas, this is far as I am able to go. I have shown that I got all the correct patches; and I was careful not to affect the instruction sizes(using NOP) during the patches to avoid memory overwriting.