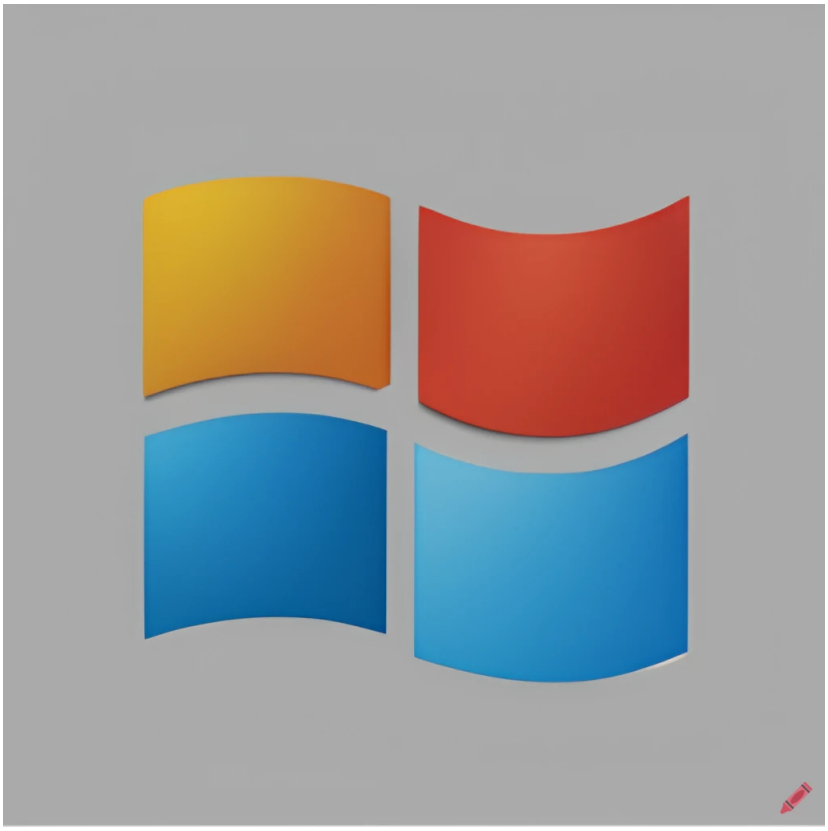
The Windows Process Journey

version 2.0 (beta)
May-2023

By Dr. Shlomi Boutnaru



Created using Craiyon, AI Image Generator

Table of Contents

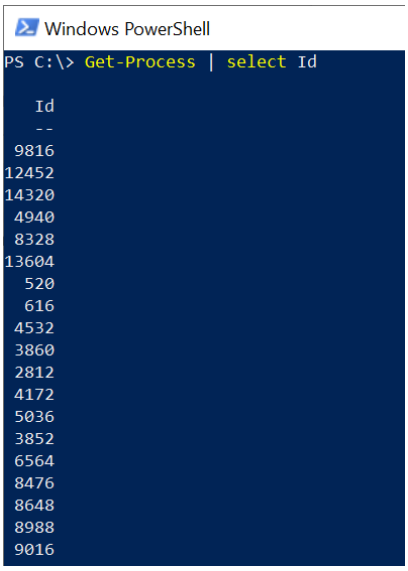
Introduction

Before speaking about a specific process I wanted to talk about an attribute related to all processes on Windows which is not so well known among all administrators/users/programmers etc.

I encourage you before reading the next lines to open any process listing app/program that you like in Windows (tasklist, task manager, process explorer or anything else) and go over PID numbers of all the processes - What can you learn from those numbers?

You probably saw that all of them are even numbers, what is more interesting is that if you divide them by two you will still get an even number - thus all the PIDs are divisible by 4!!!! BTW, the same is true for TIDs (Thread IDs) under Windows. A screenshot from

The reason for that is due to code reuse in the Windows kernel. The PIDs/TIDs are allocated by the same code which allocates kernel handles. Thus, since kernel handles are divisible by 4 so are PIDs/TIDs. We can also use the following powershell command to list only the PIDs: “Get-Process | select ID” - as shown in the screenshot below.



But why are the handles divisible by 4? Because the two bottom bits can be ignored by Windows and could be used for tagging. You can verify it by going over the comments in ntdef.h -https://github.com/tpn/winsdk-10/blob/master/Include/10.0.10240.0/shared/ntdef.h#L846. Think about the pattern for each PID/TID in binary form to fully understand it.

Lastly, you can follow me on twitter - @boutnaru (https://twitter.com/boutnaru). Also, you can read my other writeups on medium - https://medium.com/@boutnaru. Lets GO!!!!!!

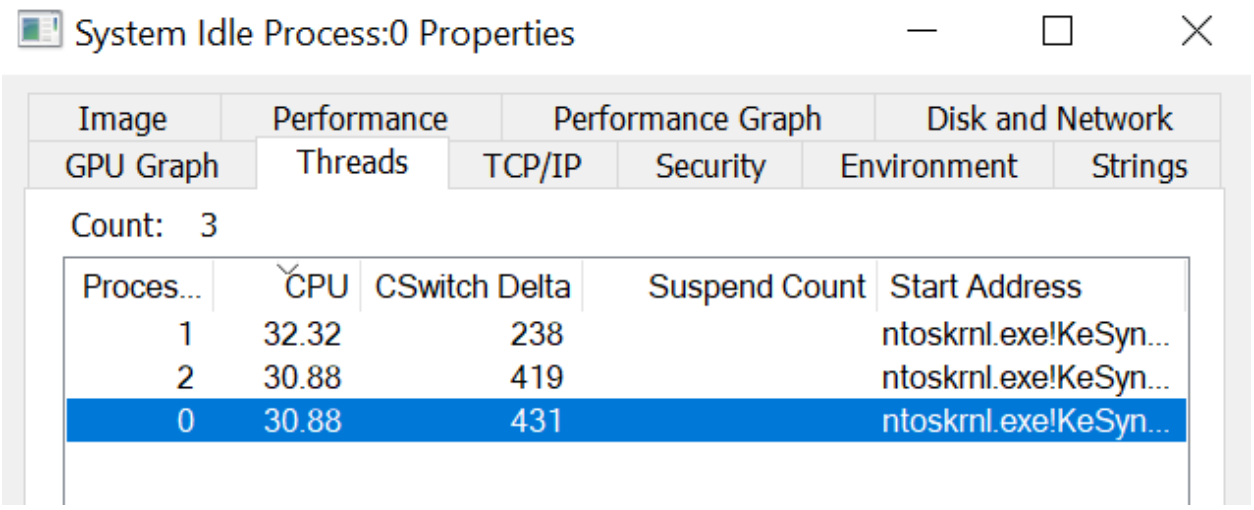
System Idle Process (PID 0)

The goal of this process is to give the CPU something to execute in case there is nothing else to do (thus it is called idle ;-). Let's think about the next situation, we have a process using 30% of CPU, in that case PID 0 (System Idle) will consume the remaining 70%. Also, Idle is the first process that the kernel starts.

Moreover, there is a kernel thread of System Idle for each vCPU the OS has identified (check out the screenshot below which shows that. The VM which I have used had 3 vCPUs - also see the first field in the table showing the “Processor”).

The reason for having an “Idle Process” is to avoid an edge case in which the scheduler (Windows schedule based on threads) does not have any thread in a “Ready” state to execute next. By the way, there are also other schedulers IO and Memory, which we will talk about in one of the next posts/writeups.

When the kernel threads are executed they can also perform different power saving tricks regarding the CPU. One of them could be halting different components which are not in use until the next interrupt arrives. The kernel threads can also call functions in the HAL (hardware abstraction layer, more on that in the future) in order to perform tasks such as reducing the CPU clock speed. Which optimization is performed is based on the version of Windows, hardware and the firmware installed.



smss.exe (Session Manager Subsystem)

“smss.exe” is the first user-mode process, it is executed from the following location: %SystemRoot%\System32\smss.exe. It’s part of Windows since Windows NT 3.1 (1993). Thus, it starts as part of the OS startup phase and performs different tasks such as those we are doing to detail next (The order of writing is not the order of execution).

Performing delayed renaming/file deletion changes based on configuration in the Registry - “HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\FileRenameOperations” (for now we should know the Registry central repository for Windows configuration, more on this in the future).

Creation of DOS device mapping based on “HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\DOS Devices” such as AUX, CON, PIPE and more (a short explanation could be found here - http://winapi.freetechsecrets.com/win32/WIN32DefineDosDevice.htm).

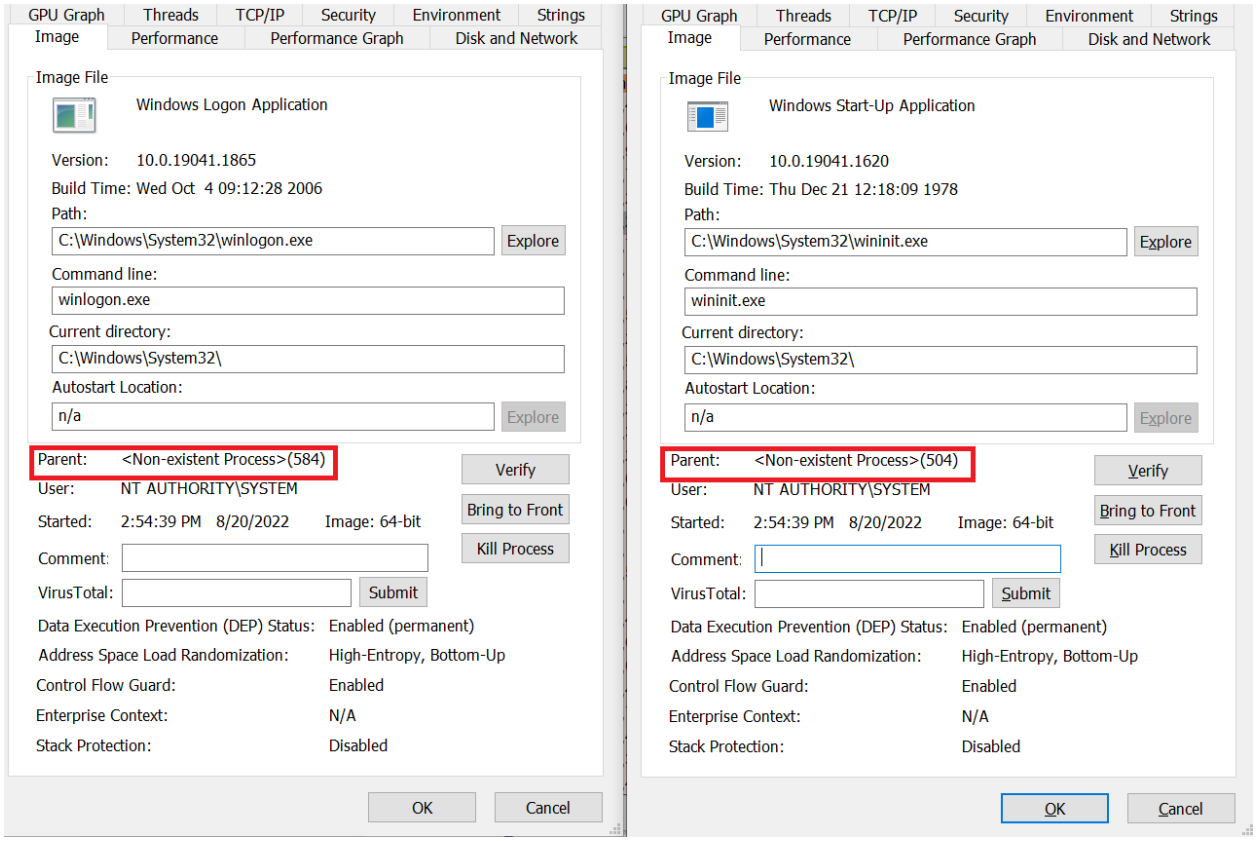
Loading the subsystems which are configured in the Registry - “HKLM\System\CurrentControlSet\Control\Session Manager\SubSystems”. At minimum we have have the kernel part of the Win32 Subsystem (aka win32k.sys) and on session 0, which is the session in which Windows’ services are executed - smss.exe starts “csrss.exe” and “wininit.exe” (you can also read about them in the following pages).

Also, on session 1, which is the first user session - smss.exe starts “csrss.exe” and “winlogon.exe”. Of course, they could be multiple sessions if more users are logged on (locally or using RDP).

Moreover, both the page files (used for virtual memory) and environment variables (“HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Environment”) are created by “smss.exe”. There are also more actions regarding memory management, KnownDlls, power management and more that are going to be discussed in the future. “smss.exe” also takes part when creating a new RDP session, we will detail this process after taking more in depth about sessions, desktops and windows stations in a future writeup - so stay tuned.

Anyhow, we should expect only one instance of “smss.exe” running without any children processes on session 0, with PPID 4 (“System Process”, on which we will also have a seperate writeup). This “smss.exe” is called the master,it is responsible for creating at minimum 2 instances of itself for session 0 and 1 (in order to do the work we detailed above). The other instances of “smss.exe” (the non-master) will terminate after finishing the session initialization

phase of a new session. On the screenshot below we can see “wininit.exe” from session 0 and “winlogon.exe” from session 1 both of them having a non-existent parent.



csrss.exe (Client Server Runtime Subsystem)

The goal of "csrss.exe" (Client Server Runtime Subsystem) is to be the user-mode part of the Win32 subsystem (which is responsible for providing the Windows API). "csrss.exe" is included in Windows from Windows NT 3.1. It is located at "%windir%\System32\csrss.exe" (which is most of the time C:\Windows\System32\csrss.exe).

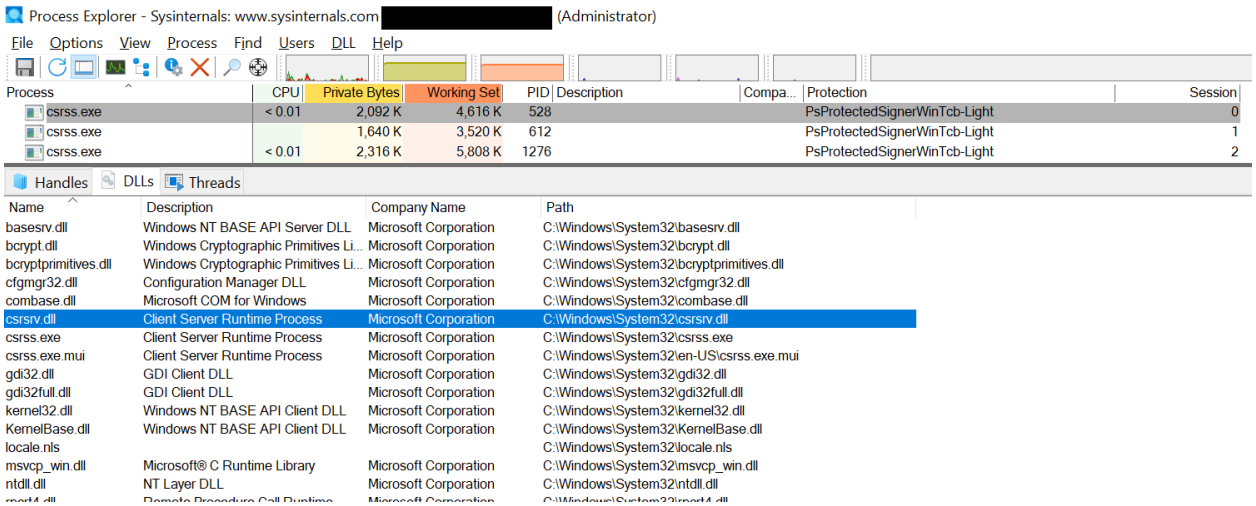
From Windows NT 4.0 most of the Win32 subsystem has been moved to kernel mode - "With this new release, the Window Manager, GDI, and related graphics device drivers have been moved to the Windows NT Executive running in kernel mode"[1]. Thus "csrss.exe" manages today GUI shutdowns and windows console (today it is "cmd.exe").

Overall, we can say that today "csrss.exe" handles things like process/threads, VDM (Visual DOS machine emulation), creating of temp files and more[2] . It is executed by "local system" and there is one instance per user session. Thus, at minimum we will have two (one for session 0 and on for session 1) - as shown in the screenshot below. "csrss.exe" has a handle for each process/thread in the specific session it is part of. Also, for each running process a CSR_PROCESS structure is maintained[3], by the way we can leverage this fact for identifying hidden processes (like by using "psxview"[4] from the volatility framework).

"smss.exe" is the process which starts "csrss.exe" together with "winlogon.exe" (more about it in a future writeup), after finishing "smss.exe" exits. In case you want to read more about "smss.exe"[5]. By the way, from Windows 7 (and later) "csrss.exe" executes "conhost.exe" instead of drawing the console windows by itself (I am going to elaborate about that in the next writeup).

Lastly, "csrss.exe" loads "csrsrv.dll", "basesrv.dll" and "winsrv.dll" as shown in the screenshot below. If we want to go over some of the source code of "csrss.exe" we can use the ReactOS which is a "A free Windows-compatible Operating System", which is hosted in github.com. The relevant code of the entire subsystem can be found at https://github.com/reactos/reactos/tree/master/subsystems/csr. We can also debug "csrss.exe" using WinDbg, it is important to know that since Windows "csrss.exe" is a protected process so it can be debugged form kernel mode only[6]. A list of all the "csrss.exe" API list can be found here https://j00ru.vexillium.org/csrss_list/api_table.html.

[1] https://learn.microsoft.com/en-us/previous-versions//cc750820(v=technet.10)?redirectedfrom=MSDN#XSLTsection124121120120
[2] https://j00ru.vexillium.org/2010/07/windows-csrss-write-up-the-basics/
[3] https://www.geoffchappell.com/studies/windows/win32/csrsrv/api/process/process.htm
[4] https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#psxview
[5] https://medium.com/@boutnaru/the-windows-process-journey-smss-exe-session-manager-subsystem-bca2cf748d33
[6] https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-csrss



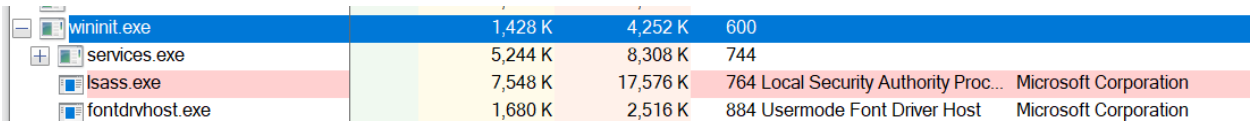
wininit.exe (Windows Start-Up Application)

“wininit.exe” is an executable which is responsible for different initialization steps as described next. The executable is located at “%windir%\System32\wininit.exe” (On 64 bit systems there is only a 64 bit version with no 32 bit version—in contrast to other executables such as cmd.exe). It is started by the first “smss.exe” at session 0 under LocalSystem (S-1–5–18). Overall there should be only one running instance of “wininit.exe”.

Historically, “wininit.exe” was used mainly in order to allow uninstallers to process commands stored in the “WinInit.ini” file. By doing so it allowed programs to take action while the system is booting[7].

Moreover, “wininit.exe” is responsible for a couple of system initialization steps. Among them are: creating the %windir%\temp folder, initializing the user-mode scheduling infrastructure, creating a window station (Winsta0) and two desktops (Winlogon and Default) for processes to run on in session 0, marking itself critical so that if it exits prematurely and the system is booted in debugging mode (it will break into the debugger) and waiting forever for system shutdown[8].

Also, “wininit.exe” launches “services.exe” (SCM—Service Control Manager) , “lsass.exe” (Local Security Authority Subsystem) and “fontdrvhost.exe” (Usermode Font Driver Host)—as seen in the screenshot below. If you want more information about service management I suggest reading https://medium.com/@boutnaru/windows-services-part-1-5d6c2d25b31c and https://medium.com/@boutnaru/windows-services-part-2-7e2bdab5bce4. Regarding the last two (“lsass.exe” and “fontdrvhost.exe”) I am going to write something in the near future.



[7] https://social.technet.microsoft.com/Forums/ie/en-US/df6f5eeb-cbb9-404f-9414-320ea02b4a60/wininitexe-what-is-is-and-why-is-it-constantly-running

[8] https://learn.microsoft.com/en-us/answers/questions/405417/explanation-of-windows-processes-and-dlls.html

winlogon.exe (Windows Logon Application)

“winlogon.exe” is an executable which is located at “%windir%\System32\winlogon.exe“ (On 64 bit systems there is only a 64-bit version with no 32-bit version like with other executables such as cmd.exe). It is executed under the “NT AUTHORITY\SYSTEM” (S-1-5-18) user. “Winlogon.exe” provides interactive support for interactive logons[9].

Overall, “winlogon.exe” manages user interactions which are related to the security of the system. Among them are: coordination of the logon flow, handling logout (aka logoff), starting “LogonUI.exe”[10], allowing the alteration of the ussr’s password and locking/unlocking the server/workstation[11]. In order to obtain user information for logon “winlogon.exe” uses credentials providers which are loaded by “LogonUI.exe” - more on them in a future writeup. For authenticating the user “winlogon.exe” gets help from “lsass.exe”.

In its initialization phase “winlogon.exe” registers the “CTRL+ALT+DEL” secure attention sequence[12] before any application can do that. Also, “winlogon.exe” creates three desktops within WinSta0: “Winlogon Desktop” (it is the desktop that the user is switched to when SAS is received), “Application Desktop” (this is the desktop created for the logon session of the user) and “ScreenSaver Desktop” (this is the desktop used when a screensaver is running). For more information I suggest reading “Initializing Winlogon”[13].

Before any logon is performed to the system, the visible desktop is Winlogon’s. Moreover, the number of instances that we expect to have is one for each interactive logon session that is present (as the number of “explorer.exe”) as minimum and in some case another one which is for the next session that can be created - as seen in the screenshot below.

Lastly, I think it is a good idea to go over the reference implementation in ReactOS for “winlogon.exe”[14].

```
C:\>tasklist | findstr  explorer.exe
explorer.exe                 6568 31C5CE94259D4006          2

C:\>tasklist | findstr  winlogon
winlogon.exe                  708 Console                   1
winlogon.exe                 3292 31C5CE94259D4006          2
```

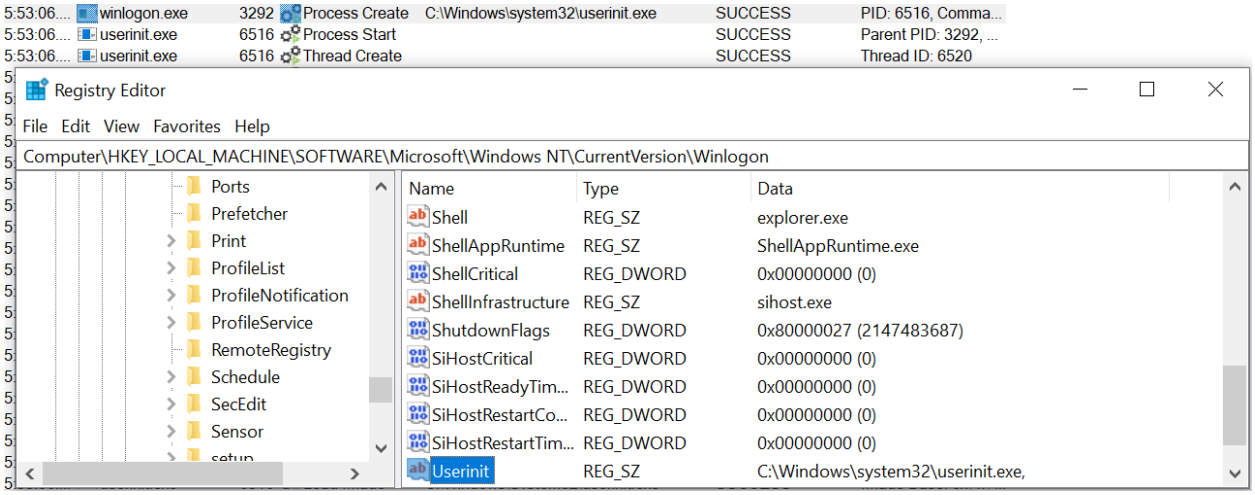
[9] https://learn.microsoft.com/en-us/windows/win32/secgloss/w-gly
[10] https://medium.com/@boutnaru/the-windows-process-journey-logonui-exe-windows-logon-user-interface-host-4b5b8b6417cb
[11] https://www.microsoftpressstore.com/articles/article.aspx?p=2228450&seqNum=8
[12] https://medium.com/@boutnaru/security-sas-secure-attention-sequence-da8766d859b5
[13] https://learn.microsoft.com/en-us/windows/win32/secauthn/initializing-winlogon
[14] https://github.com/reactos/reactos/tree/2752c42f0b472f2db873308787a8b474c4738393/base/system/winlogon

userinit.exe (Userinit Logon Application)

“userinit.exe” is an executable which is located executable is located at “%windir%\System32\userinit.exe“ (On 64 bit systems there is only a 64 bit there is also a 32 bit version located at “%windir%\SysWOW64\userinit.exe”). It is started by the “winlogon.exe” - as seen in the screenshot below (taken from ProcMon). Also, “userinit.exe” is executed with the permissions of the user which is logging in to the system.

Overall, “userinit.exe” is responsible for loading the user’s profile and executing startup applications while the logon process of the user is being performed. Thus, it will execute logon scripts[15].

“C:\Windows\System32\userinit.exe” is defined by default as the executable for the UserInit phase under the “userinit” key in the registry[16] - as shown in the screenshot below (taken from “regedit.exe”). Moreover, “userinit.exe” runs the shell of the logged on user, which is by default “explorer.exe” as configured in the registry under the “shell” key[17] - as shown in the screenshot below (taken from “regedit.exe”).



I think it is a good idea to go over the reference implementation in ReactOS for “userinit.exe” (https://github.com/reactos/reactos/tree/3fa57b8ff7fcee47b8e2ed869aecaf4515603f3f/base/system/userinit).

[15] https://www.minitool.com/news/userinit-exe.html
[16] HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\UserInit
[17] HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

dwm.exe (Desktop Window Manager)

"dwm.exe" (Desktop Window Manager) is the executable which handles different tasks in the display process of the Windows UI like rendering effects. Among those efforts are: live taskbar thumbnails, Flip3D, transparent windows and more[18]. The executable is located at "%windir%\System32\dwm.exe" (On 64 bit systems there is only a 64 bit version with no 32 bit version like with other executables such as cmd.exe).

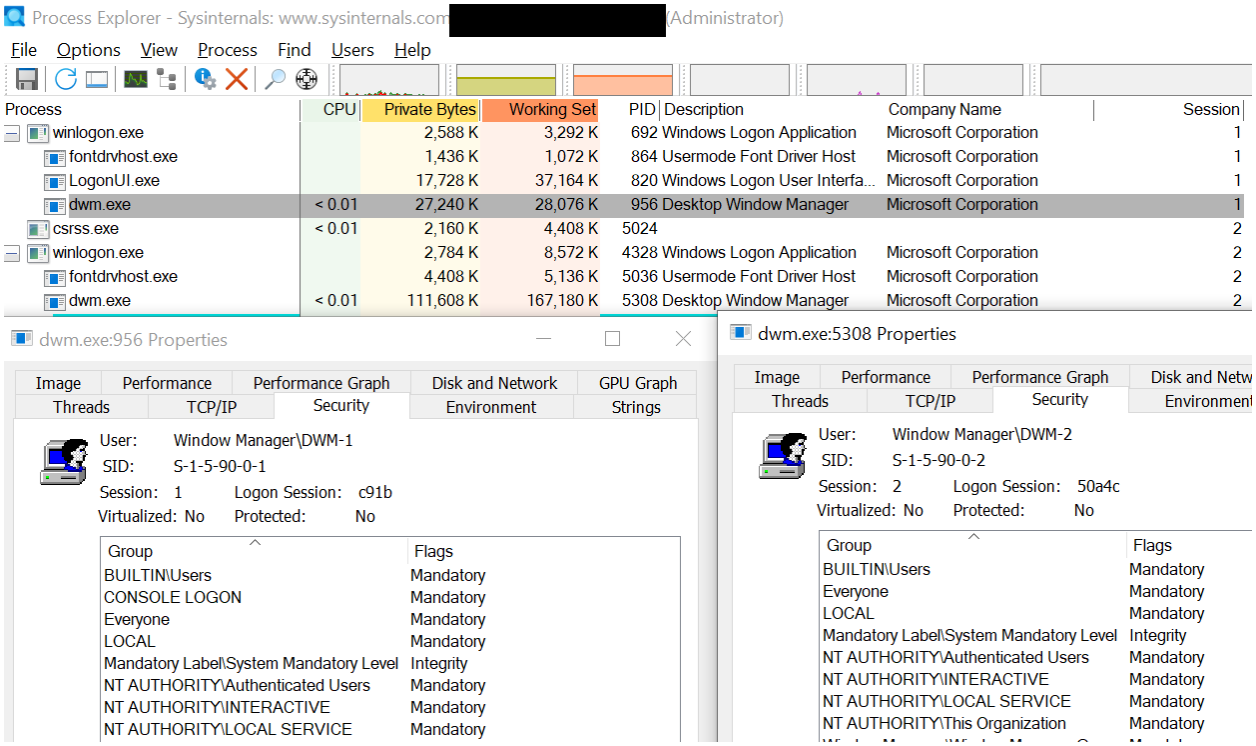
Thus, we can think about "dwm.exe" as a "compositing windows manager". A "windows manager" is computer software that controls the placement and appearance of a window as part of a "window system" in a GUI environment[19]. So, a "compositing windows manager" is a "window manager" that provides applications with an off-screen buffer for each window. The goal of the manager is to composite all the windows' buffers into an image representing the screen and commit it to the display memory[20].

The desktop composition feature was introduced in Windows Vista. It changed the way applications display pixels on the screen (as it was until Windows XP). When desktop composition is enabled, individual windows no longer draw directly to the screen (or primary display device). Their drawings are redirected to off-screen surfaces in video memory, which are then rendered into a desktop image and presented on the display.

For more information I suggest reading the following links https://learn.microsoft.com/en-us/windows/win32/dwm/dwm-overview and https://learn.microsoft.com/en-us/archive/blogs/greg_schechter/under-the-hood-of-the-desktop-window-manager.

Under Windows 10, there is one instance of "dwm.exe" for each session (excluding session 0). The parent process for each "dwm.exe" is "winlogon.exe". The user which is associated with the security token of each "dwm.exe" has a the pattern of "Window Manager\DWM-{SESSION_ID}" and a SID of pattern "S-1–5–90–0-{SESSION_ID}" as shown in the screenshot below (taken from Process Explorer).

[18] https://learn.microsoft.com/en-us/windows/win32/dwm/dwm-overview
[19] https://en.wikipedia.org/wiki/Window_manager
[20] https://en.wikipedia.org/wiki/Compositing_window_manager



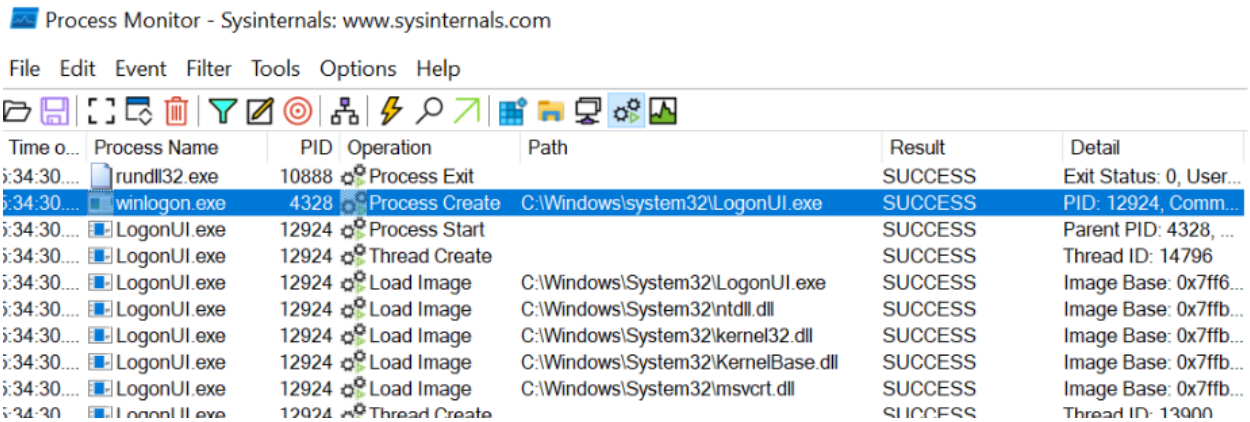
LogonUI.exe (Windows Logon User Interface Host)

"LogonUI.exe" (Windows Logon User Interface Host) is responsible for the graphical user interface which asks the user to logon into the system (aka logon screen/lock screen). The executable file is located at "%SystemRoot%\System32\LogonUI.exe" (On 64 bit systems there is only a 64 bit version with no 32 bit version like with other executables such as cmd.exe).

Moreover, "LogonUI.exe" is executed under the Local System user (S-1–5–18) for every session (excluding session 0). "winlogon.exe" is the process which is responsible for running "LogonUI.exe" as we can see in the screenshot below, which was taken from Process Monitor[21]. Also, if you want to see how "LogonUI.exe" GUI looks in different versions of Windows[22].

In the perspective of the data flow between "LogonUI.exe" and "winlogon.exe" the basic phases are as follows (after "LogonUI.exe" was launched by "winlogon.exe"). "LogonUI.exe" gets credentials from the user (like username and password) and sends them to "winlogon.exe". "winlogon.exe" performs the authentication (since Windows Vista it is done using a credential provider, before that it was done by msgina.dll). If the authentication process succeeds, it sends a message back to "LogonUI.exe" to indicate that the user has been authenticated[23]. We will get deeper into this flow after talking about "winlogon.exe", sessions, ALPC (which is the communication line between the processes) and more.

In addition, settings for LogonUI.exe are stored in the registry in the following branch: "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\LogonUI". Among those settings we can find the user list that should be shown, the last user that logged-on and the background image. Lastly, if you want to see a reference code for "LogonUI.exe" you can check out ReactOS[24].



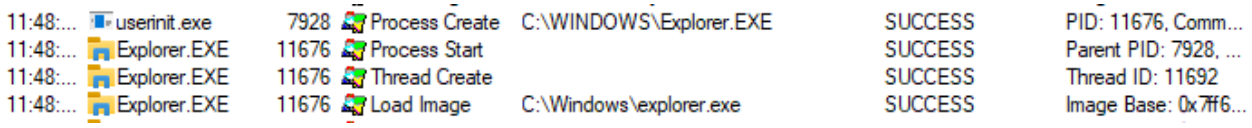
[21] https://learn.microsoft.com/en-us/sysinternals/downloads/procmon
[22] https://media.askvg.com/articles/images3/Windows_Login_Screen.png
[23] https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/credentials-processes-in-windows-authentication
[24] https://github.com/reactos/reactos/tree/3647f6a5eb633b52ef4bf1db6e43fc2b3fc72969/base/system/logonui

explorer.exe (Windows Explorer)

“explorer.exe” is an executable which is the “Windows Explorer”. The executable is located at “%windir%\explorer.exe (On 64 bit systems there is also a 32 bit version located in %windir%\SysWOW64\explorer.exe). It is responsible for handling elements of the graphical user interface in Windows (including the taskbar, start menu, and desktop), the “File Explorer” and more. Thus, we can think about it as a graphical shell[25].

In case we terminate “explorer.exe” the taskbar will disappear and also the desktop both the shortcuts and the wallpaper itself[26]. For more understanding about “exeplorer.exe” I think it is a good idea to go over the reference implementation in ReactOS[27].

Every time a user logins interactively “explorer.exe” is executed under the user which logged on to the system[28]. The process which starts “explorer.exe” is “userinit.exe” (I will post on it in the near future) - as can be seen in the screenshot below.



I also suggest going over the following link https://ss64.com/nt/explorer.html to checkout all the arguments that can be passed to “exeplorer.exe” while launching it. There are also several examples of usage there. By the way, it seems that Microsoft wants to decouple features from “explorer.exe” in order to make Windows 11 faster[29].

[25] https://www.pcmag.com/encyclopedia/term/explorerexe
[26] https://copyprogramming.com/howto/what-happens-if-i-end-the-explorer-exe-process
[27] https://github.com/reactos/reactos/tree/81db5e1da884f76e6cee66b8cb1c7a2f6ff791eb/base/shell/explorer
[28] https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/windows-logon-scenarios
[29] https://www.windowslatest.com/2022/12/22/microsoft-wants-to-make-windows-11-faster-by-decoupling-features-from-explorer-exe/

svchost.exe (Host Process for Windows Services)

“svchost.exe” is probably the builtin executable which has the most instances (for example 78 on the my testing VM) among all the running processes in Windows. We can split its name to “Svc” and “Host”, that is service host which hits its responsibility (more on that later).

The executable “svchost.exe” is located in %windir%\System32\svchost.exe. In case we are talking about the 64 bit version of Windows, there is also %windir%\SysWOW64\svchost.exe (which is a 32 bit version). Both of the files are signed digitally by Microsoft. It was introduced during Windows 2000, even though there was support for “shared service processes” already in Windows NT 3.1 (more on this in the following paragraphs).

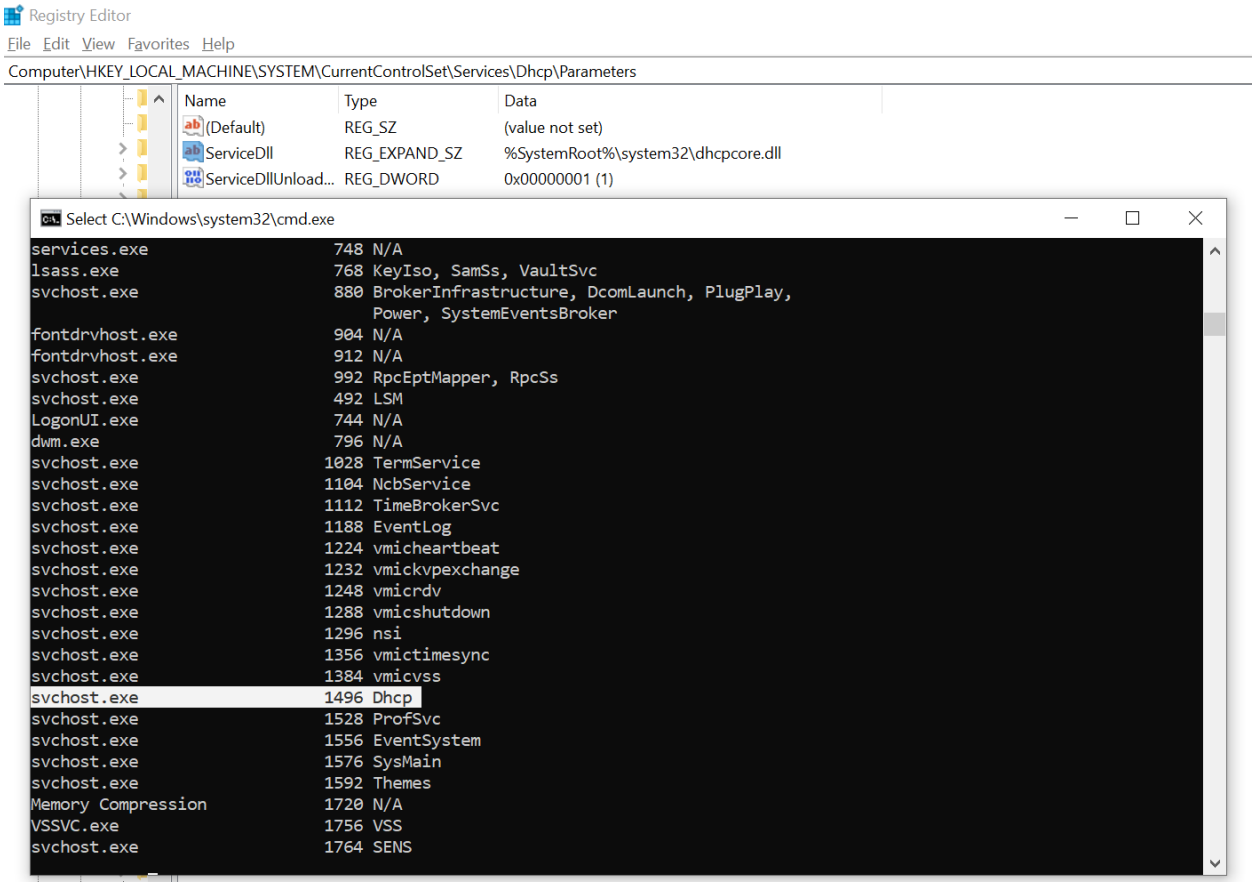
Due to the fact, many of the Windows’ services (you can read on Wndows’ Services on https://medium.com/@boutnaru/windows-services-part-2-7e2bdab5bce4) are implemented as DLLs (Dynamic Link Libraries) there is a need for an executable to host them. Thus, you can think about “svchost.exe” as the implementation of “shared service process” - A process which hosts/executes/runs multiple services in a single memory address space.

The configuration of services is stored in the registry (“HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services”), for each service which is hosted the name of the DLL is stored under the “Parameter” subkey in a value named “ServiceDll”. For example, in the case of the DHCP client is “HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Dhcp\Parameters\ServiceDll” - as shown in the screenshot below. The ImagePath (which stores the path to the executable to run when starting the service) will be “svchost.exe” with a command line parameter of “-k” and the name of the service groups (like netsvcs, Dcomlaunch, utcsvc, and LocalServiceNoNetwork, LocalSystemNetworkRestricted).

At the end services are splitted into different groups, every group is hosted by one host process which is a single instance of “svchost.exe”. If we want to see which services are hosted on which “svchost.exe” you can use tools like “Process Explorer” and “tasklist” - as you can see in the screenshot below. The configuration of which services are part of what group we can see at “HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost” (on my test VM a total of 49 groups are defined).

It is important to know that from Windows 10 (version 1903) on systems with more than 3.5GB or RAM by default there is no grouping. That is, every service will be executed in a single instance of “svchost.exe” for better security and reliability. Of course there are exceptions for that[30].

[30] https://learn.microsoft.com/en-us/windows/application-management/svchost-service-refactoring

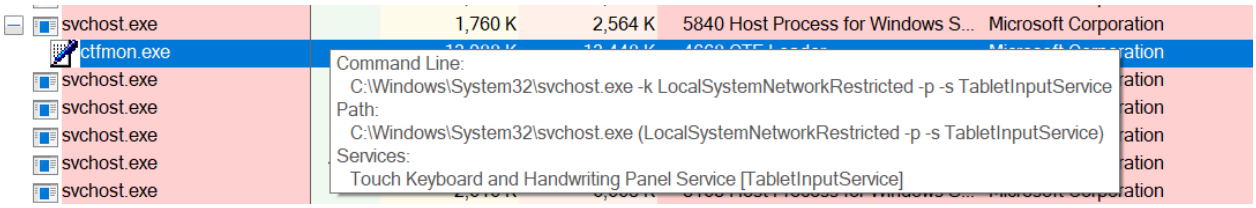


ctfmon.exe (CTF Loader)

“ctfmon.exe” is a user-mode process which is executed from the following location %SystemRoot%\System32\ctfmon.exe. If you are using a 64 bit version of Windows, there is also a 32 bit version of “ctfmon.exe” located at C:\Windows\SysWOW64\ctfmon.exe. By parsing the file information we can see that it is described as a “CTF Loader”. CTF stands for “Collaboration Translation Framework”, it is used by Microsoft Office.

The goal of “ctfmon.exe” is to provide different input capabilities for users such as speech and handwriting recognition. By the way, it will run even if you are not using Microsoft Office.

“Ctfmon.exe” is launched as a child process of the service TabletInputService ("Touch Keyboard and Handwriting Panel Service"), which is hosted by “svchost.exe” - as shown in the screenshot below. Thus, if we want to stop “ctfmon.exe” we can just disable/stop that service. For more information about what is “svchost.exe” you can read the following link https://medium.com/@boutnaru/the-windows-process-journey-svchost-exe-host-process-for-windows-services-b18c65f7073f.

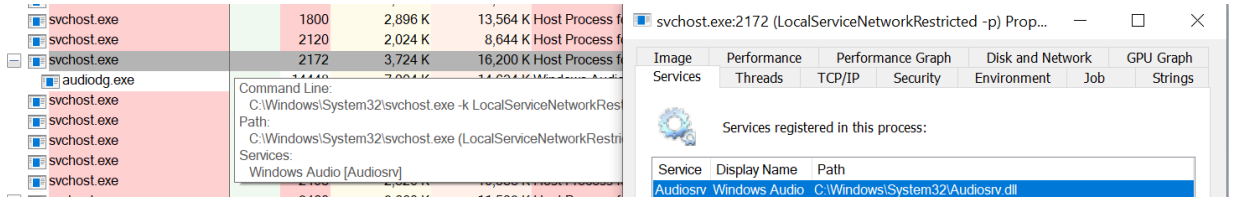


audiodg.exe (Windows Audio Device Graph Isolation)

“audiodg.exe” is an executable which is part of the Windows shared-mode audio engine as described next. The executable is located at “%windir%\System32\audiodg.exe” (On 64 bit systems there is only a 64 bit version with no 32 bit version—in contrast to other executables such as cmd.exe). The process is running under the user “NT AUTHORITY\LOCAL SERVICE”.

In Windows the audio engine runs in user mode. We have the "Windows Audio" service which is implemented in AudioSrv.dll, it is hosted using the “svchost.exe” process. The service launches a helper process “audiodg.exe”[31]. All of that is demonstrated in the screenshot below. It runs in a different login session from the logged on user (isolated) in order to that content and plug-ins cannot be modified[32].

Thus, we can say that “audiodg.exe” is being utilized for all audio processing[33]. It hosts the audio engine for Windows so all the digital signal processing (DSP) is performed by “audiodg.exe”. Vendors can install their own audio effects which will be processed by “audiodg.exe”[34]. There should be one instance only of “audiodg.exe” at a specific time.



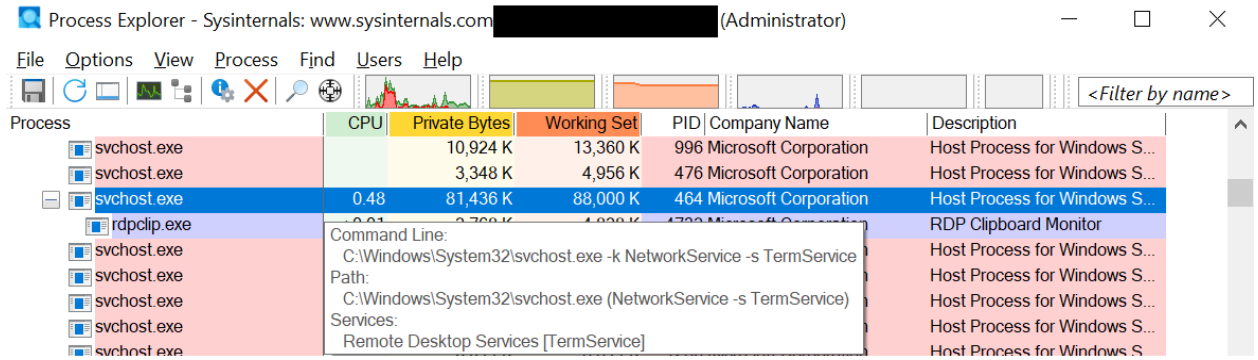
[31] https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/audio-measures
[32] https://answers.microsoft.com/en-us/windows/forum/all/audiodgexe/0c86aef4-81a5-480e-9389-d9652fee1d21
[33] https://answers.microsoft.com/en-us/windows/forum/all/windows-10-audiodgexe/af1b70e0-06fe-4952-8205-b6191ccb8882
[34] https://answers.microsoft.com/en-us/windows/forum/all/audiodgexe-high-cpu-and-memory/42b3f122-87bf-45cd-8ea7-08abafa9442c

rdpclip.exe (RDP Clipboard Monitor)

"rdpclip.exe" (RDP Clipboard Monitor) is responsible for managing the shared clipboard between the local computer and the remote desktop which the user is interacting with[35]. The executable file is located at "%windir%\System32\rdpclip.exe" (On 64 bit systems there is only a 64 bit version with no 32 bit version like with other executables such as cmd.exe).

By enabling the "Remote Desktop" capability[36] on Windows it allows remote management of a system using a GUI (graphical user interface) by leveraging the Remote Desktop Protocol (RDP). The default port of the protocol is TCP/3389. For more information about the protocol I suggest reading the following link https://www.cyberark.com/resources/threat-research-blog/explain-like-i-m-5-remote-desktop-protocol-rdp.

"rdpclip" is started when a new remote desktop session is created by the service which is called "Remote Desktop Services" - as shown in the screenshot below. Fun fact, the old display name of the service was "Terminal Services" which was changed while the service name is still "TermService".



Lastly, the description of the service states "it allows users to connect interactively to a remote computer. Remote Desktop and Remote Desktop Session Host Server depend on this service. To prevent remote use of this computer, clear the checkboxes on the Remote tab of the System properties control panel item".

[35] https://www.winosbite.com/rdpclip-exe/

[36] https://learn.microsoft.com/en-us/windows-server/remote/remote-desktop-services/clients/remote-desktop-allow-access

smartscreen.exe (Windows Defender SmartScreen)

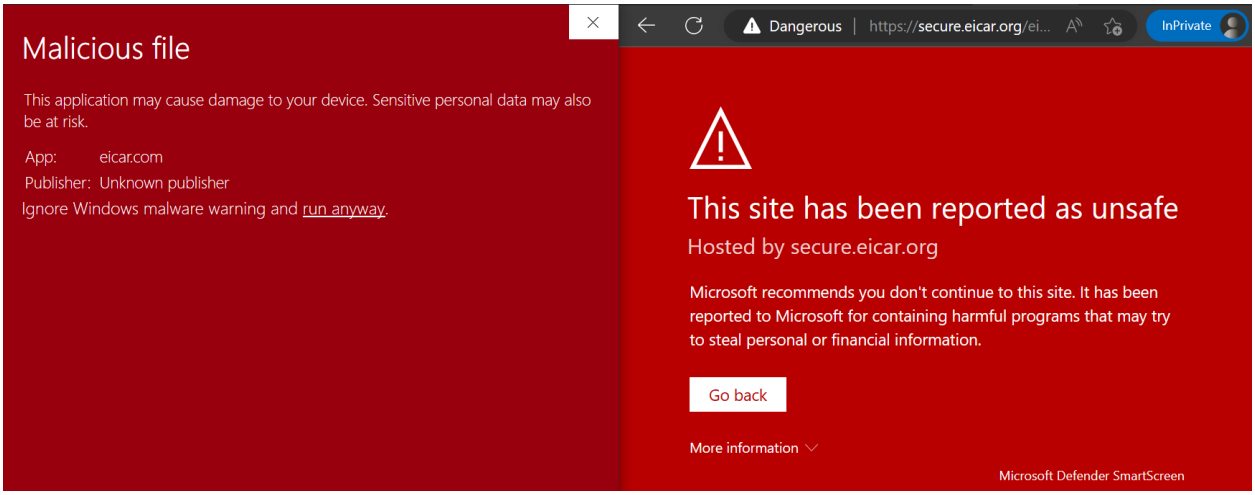
“smartscreen.exe” is an executable which is the “Windows Defender SmartScreen”. The executable is located at “%windir%\System32\smartscreen.exe” (On 64 bit systems there is only a 64 bit version with no 32 bit version—in contrast to other executables such as cmd.exe).

SmartScreen is a cloud-based anti-phishing/anti-malware component which is included in different Microsoft products such as: Windows, Internet Explorer and Microsoft Edge (https://en.wikipedia.org/wiki/Microsoft_SmartScreen).

Microsoft Defender SmartScreen helps with determining whether a site is potentially malicious and by determining if a downloaded application/installer is potentially malicious. We can sum up the benefits of SmartScreen as follows: anti-phishing/anti-malware support, reputation-based URL/application protection, operating system integration, ease of management using group policy/Microsoft Intune and blocking URLs associated with potentially unwanted applications. (https://learn.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-smartscreen/microsoft-defender-smartscreen-overview).

In order to demonstrate the working of SmartScreen I have tried to download (using Edge) - you can see the warning in the left side of the screenshot below. Moreover, after downloading it using a different browser I have executed the EICAR test file - you can see the result in the left side of the screenshot below. By the way, the EICAR (European Institute from Computer Antivirus Research) test file was created to test the response of AV software (https://en.wikipedia.org/wiki/EICAR_test_file).

Lastly, we can enable/disable SmartScreen using the settings window, bot for the OS/browser (https://www.digitalcitizen.life/how-disable-or-enable-smartscreen-filter-internet-explorer-or-windows-8/).

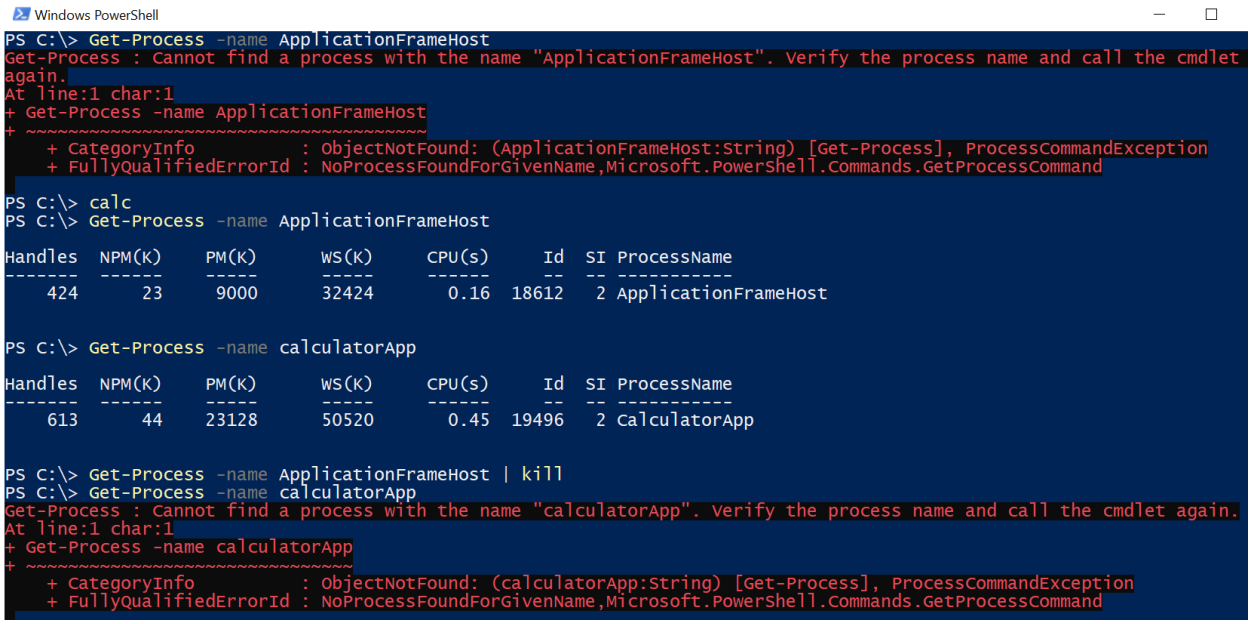


ApplicationFrameHost.exe

The "ApplicationFrameHost.exe" executable is located at the following directory - "%windir%\system32\ApplicationFrameHost.exe". On 64-bit systems there is only a 64-bit version with no 32 bit version—in contrast to other executables such as cmd.exe.

Overall, the goal of "ApplicationFrameHost.exe" is to display the frames (windows) of the applications whether we are in desktop/tablet mode[37]. By the way, if we kill "ApplicationFrameHost.exe" all the UWP applications will be closed also - as we can see in the screenshot below.

There is one instance per session for the "ApplicationFrameHost.exe" in case one or more "Window Store App" which is also known as "Universal Windows Platform App"[38] - I will elaborate about them in a separate writeup. An example for a UWP app is the Calculator ("%windir%\system32\calc.exe"). Also, "ApplicationFrameHost.exe" is running with the permissions of the logged on user (that from whom the session was created).



[37] https://www.howtogeek.com/325127/what-is-application-frame-host-and-why-is-it-running-on-my-pc/
[38] https://www.file.net/process/applicationframehost.exe.html

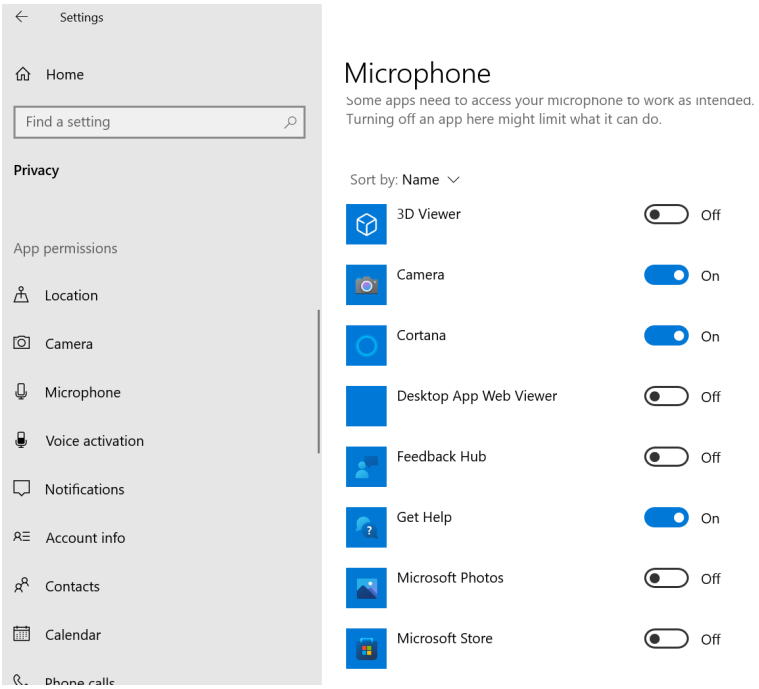
RuntimeBroker.exe

“RuntimeBroker.exe” is an executable which that is located at “%windir%\System32\RuntimeBroker.exe” (On 64 bit systems there is only a 64-bit version with no 32-bit version—in contrast to other executables such as cmd.exe).

“RuntimeBroker.exe” is running the permissions of the user (from whom the session was created). “RuntimeBroker.exe” is triggered from execution if the Windows Store is opened or any installed UWP app is started. By the way UWP apps are also known as Windows App/Windows Store App/Metro App[39].

Overall, “RuntimeBroker.exe” is responsible for managing the permissions for “Windows Store App”. We can think about it as a middleman between the application and operating system capabilities[40].

Thus, when an UWP application tries to access a specific OS resource “RuntimeBroker.exe” checks if the application has the appropriate permissions for that. In case it does not, “RuntimeBroker.exe” can ask the user to grant the permissions. We can modify the permissions for different applications using the “Settings” screen (Privacy->App permissions) - as shown in the screenshot below.



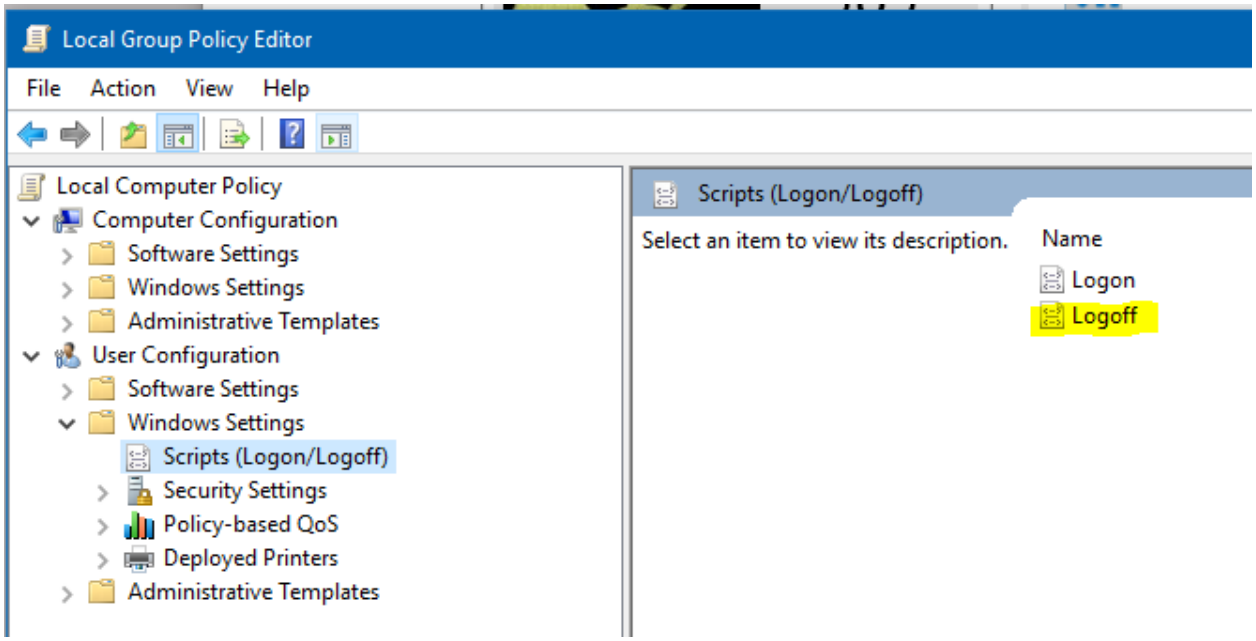
[39] https://www.file.net/process/runtimebroker.exe.html

[40] https://support.microsoft.com/en-us/windows/runtime-broker-is-using-too-much-memory-ca6ed4e3-2a36-964c-4d2e-8c93980d8a98

logoff.exe (Session Logoff Utility)

"logoff.exe" (Session Logoff Utility) is a command line tool that allows logging off a user from a session. The session could be the current session in which the command is executed, a specific session identified by a number or a remote session on a different server[41]. The executable file is located at "%windir%\System32\logoff.exe".

Moreover, an administrator can set a script/executable to be executed when the user is logging off. This setting can be configured using a local policy/group policy and is called "Logoff script). Alos, this configuration is part of the "User Configuration -> Windows Settings -> Scripts" - as shown in the screenshot below[42]. Lastly, we can also go over a reference code for "logoff.exe" from ReactOS[43].



[41] https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/logoff
[42] https://social.technet.microsoft.com/Forums/en-US/f9f011e2-59fc-42d3-a1a4-251536ce8287/i-need-to-automatically-run-an-app-at-logoff?forum=win10itprosetup
[43] https://github.com/reactos/reactos/tree/3fa57b8ff7fcee47b8e2ed869aecaf4515603f3f/base/applications/logoff

cscript.exe (Microsoft ® Console Based Script Host)

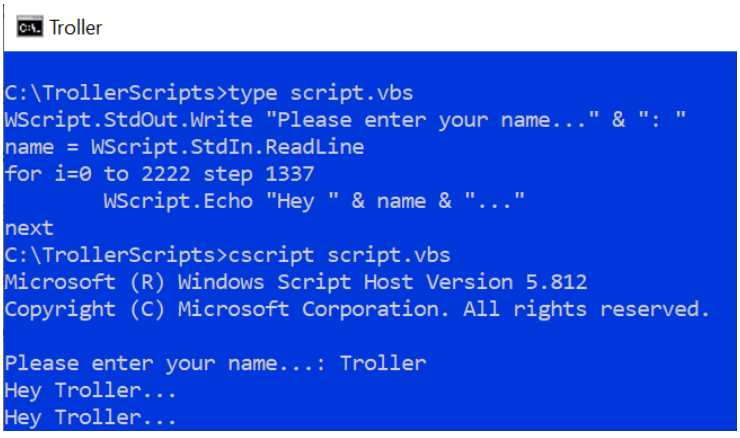
“cscript.exe” is the “Microsoft ® Console Based Script Host” which is a command line version of the “Windows Script Host”. It also allows setting script properties using command line options[44].

Also, “cscript.exe” is a PE binary file located at “%windir%\System32\cscript.exe”. On a 64-bit system (with a 64-bit OS installed) there is also a 32-bit based version located at “%windir%\SysWOW64\cscript.exe”.

Overall, the “Windows Script Host” (WSH) is an automation technology that enables scripting which was first introduced in Windows 95 (after build 950a) and became a standard component since Windows 98 (build 1111). It has support for different language engines, by default it supports JScript (*.js/*.jse) and VBScript (*.vbs/*.vbe) out of the box[45].

Moreover, users can also install other scripting engines for WSH like Perl and Python . By using WSH we can also leverage COM (). In VBScript we can do so by calling CreateObject() and in JSCript we can use an ActivexObject or call WSCript.CreateObject()[46].

When using “cscript.exe” to run a script to run in a command-line environment we don’t have to use administrator permissions. Alos, “cscript.exe” has multiple command line options for different usages like: interactive mode, debugging mode, passing arguments to the script and more[47]. Lastly, in order to demonstrate the usage of “cscript.exe” I have created a simple script and executed it - as shown in the screenshot below. We can also go over a reference implementation of “cscript.exe” for RactOS[48].



[44] https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb490887(v=technet.10)?redirectedfrom=MSDN
[45] https://en.wikipedia.org/wiki/Windows_Script_Host
[46] https://learn.microsoft.com/vi-vn/windows/win32/com/using-com-objects-in-windows-script-host
[47] https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cscript
[48] https://github.com/reactos/reactos/tree/3fa57b8ff7fcee47b8e2ed869aecaf4515603f3f/base/applications/cmdutils/cscript

wscript.exe (Microsoft ® Windows Based Script Host)

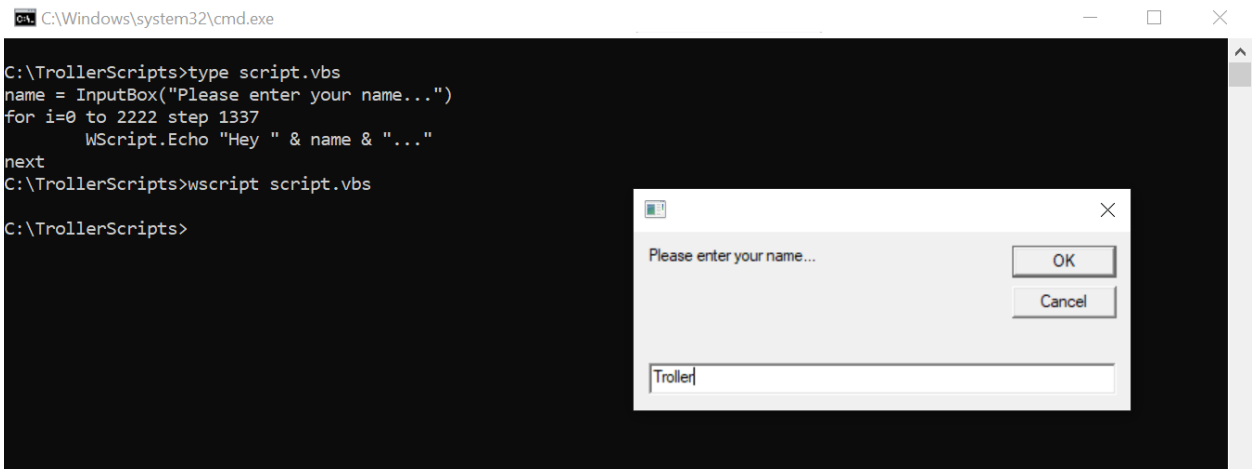
“wscript.exe” is the “Microsoft ® Windows Based Script Host” which provides an environment for executing scripts in a variety of languages[49]. It also allows setting script properties using command line options[50].

Overall, the “Windows Script Host” (WSH) is an automation technology that enables scripting which was first introduced in Windows 95 (after build 950a) and became a standard component since Windows 98 (build 1111). It has support for different language engines, by default it supports JScript (*.js/*.jse) and VBScript (*.vbs/*.vbe) out of the box[51].

Also, “wscript.exe” is a PE binary file located at “%windir%\System32\wscript.exe”. On a 64-bit system (with a 64-bit OS installed) there is also a 32-bit based version located at “%windir%\SysWOW64\wscript.exe”.

“wscript.exe” allows running the scripts in GUI mode in contrast to “cscript” which is CLI mode[52]. Gui mode means that graphical components could be displayed as the script is being executed - as shown in the screenshot below.

Lastly, in case you want to see a reference implementation of “wscript.exe” I suggest going over the implementation which is part of ReactOS[53].



[49] https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh875526(v=ws.11)
[50] https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/wscript
[51] https://en.wikipedia.org/wiki/Windows_Script_Host
[52] https://medium.com/@boutnaru/the-windows-process-journey-cscript-exe-microsoft-console-based-script-host-5878ba9354a0
[53] https://github.com/reactos/reactos/tree/3fa57b8ff7fcee47b8e2ed869aecaf4515603f3f/base/applications/cmdutils/wscript

utilman.exe (Utility Manager)

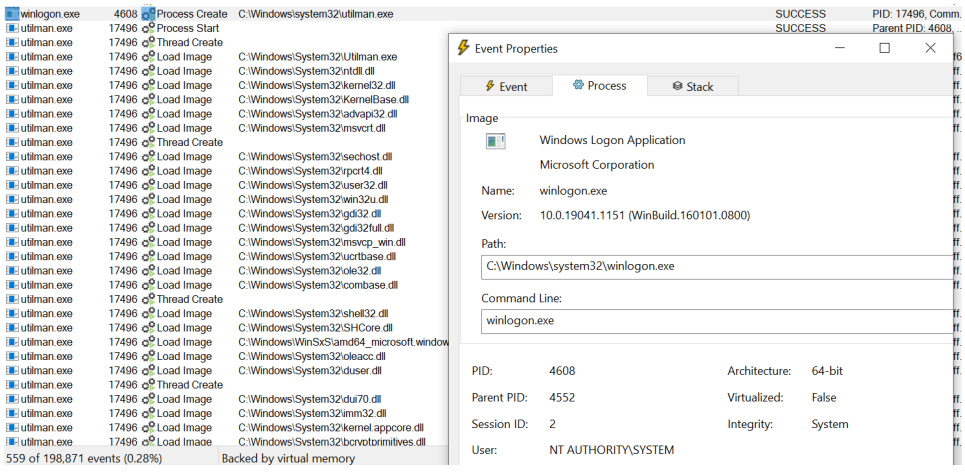
“utilman.exe” is the “Utility Manager” which is a PE binary file located at “%windir%\System32\utilman.exe”. On 64-bit systems there is also a 32-bit version located on “%windir%\SysWOW64\utilman.exe”.

Overall, “utilman.exe” can be started by clicking the icon of “Ease of Access” or by using the keyboard shortcut “WinKey+U”. When using one of those methods while the computer is locked, “utilman.exe” is started by “winlogon.exe” with the permissions of the “LocalSystem” - as shown in the screenshot below. By the way, due to the high level of permissions in use replacing “utilman.exe” is a common trick in order to reset the administrator password in Windows[54].

Moreover, “utilman.exe” allows accessing the following capabilities: narrator, magnifier, onscreen keyboard, high contrast, sticky keys and filter keys. Narrator is the screen reading application made for blind/visually impaired users[55]. Magnifier is an application that allows users to enlarge the screen content[56].

Also, sticky keys allows users to use modifier keys (like Ctrl, Shift, Alt and WinKey) without the need of pressing them constantly[57].. Filter keys is a feature that adjusts the keyboard response and ignores repeated keystrokes caused by inaccurate or slow finger movements[58].

Lastly, in case you want to see a reference implementation of “osk.exe” I suggest going over the implementation which is part of ReactOS[59].



[54] https://learn.microsoft.com/en-us/answers/questions/187973/windows-recovery-cmd
[55] https://support.microsoft.com/en-us/windows/complete-guide-to-narrator-e4397a0d-ef4f-b386-d8ae-c172f109bdb1
[56] https://support.microsoft.com/en-us/windows/use-magnifier-to-make-things-on-the-screen-easier-to-see-414948ba-8b1c-d3bd-8615-0e5e32204198
[57] https://geekflare.com/using-sticky-keys-in-windows/
[58] https://helpdeskgeek.com/how-to/what-are-filter-keys-and-how-to-turn-them-off-in-windows/
[59] https://github.com/reactos/reactos/tree/3fa57b8ff7fcee47b8e2ed869aecaf4515603f3f/base/applications/utilman

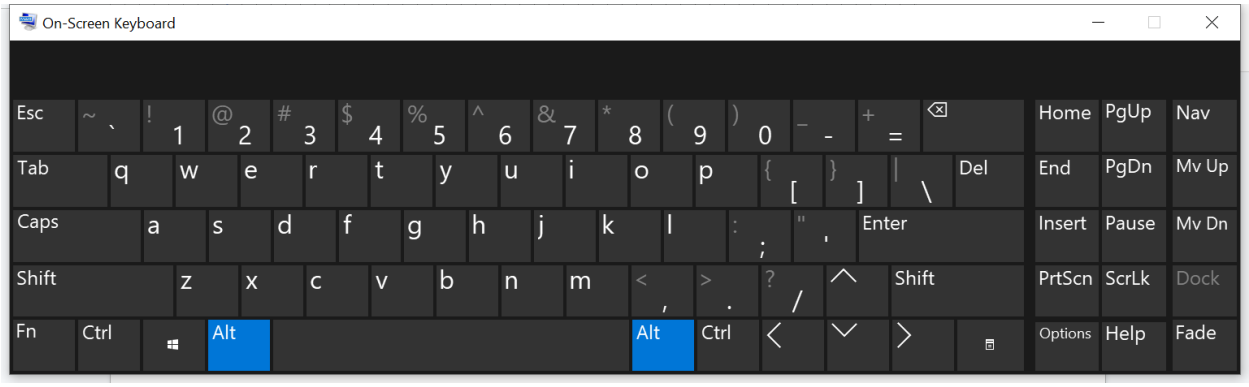
osk.exe (Accessibility On-Screen Keyboard)

“osk.exe” is the “Accessibility On-Screen Keyboard” which presents a virtual keyboard layout inside a resizable window - as shown in the screenshot below. The virtual keyboards enable the user clicking/hovering/scanning using a mouse/joystick in order to select/activate keys[60].

Moreover, “osk.exe” has a 101/102/106 key layout. “osk.exe” is a PE binary located at “%windir%\System32\osk.exe”. It is bundled with Windows and can provide some features for users with limited mobility[61].

Thus, we don’t need a touch screen in order to interact with “osk.exe”[62]. By the way, “osk.exe” is not the only virtual keyboard available as part of Windows, there is also “TabTip.exe” - but more on there is a separate writeup.

Lastly, in case you want to see a reference implementation of “osk.exe” I suggest going over the implementation which is part of ReactOS[63].



[60] https://www.file.net/process/osk.exe.html
[61] https://www.processlibrary.com/en/directory/files/osk/21965/
[62] https://support.microsoft.com/en-us/windows/use-the-on-screen-keyboard-osk-to-type-ecbb5e08-5b4e-d8c8-f794-81dbf896267a
[63] https://github.com/reactos/reactos/tree/47f3a4e144b897da0e0e8cb08c2909645061dec9/base/applications/osk