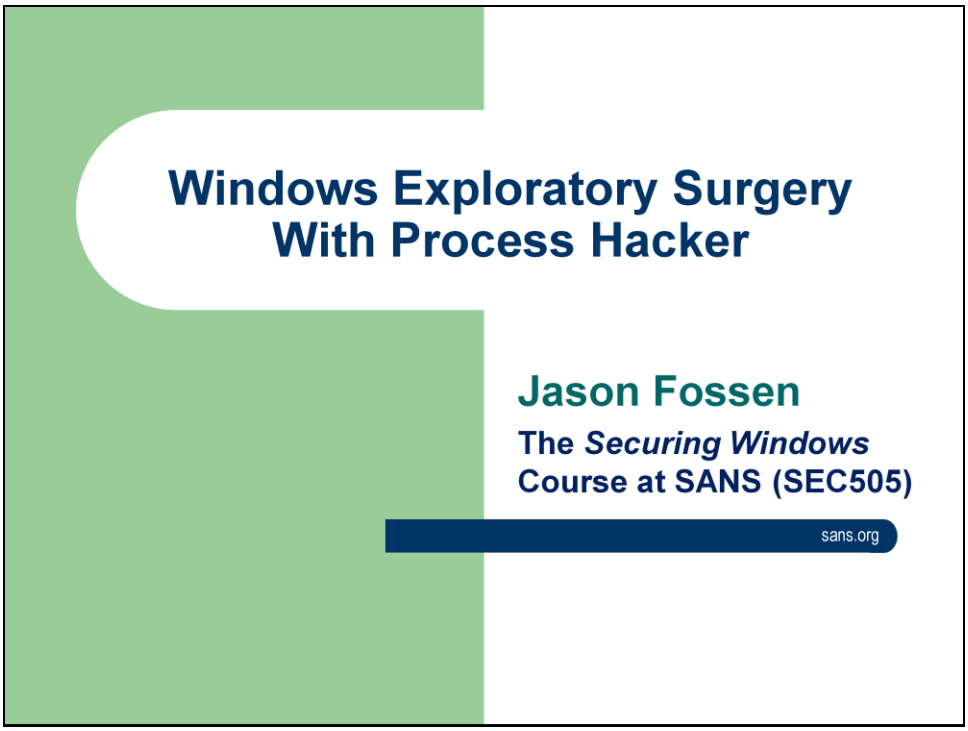


Welcome!

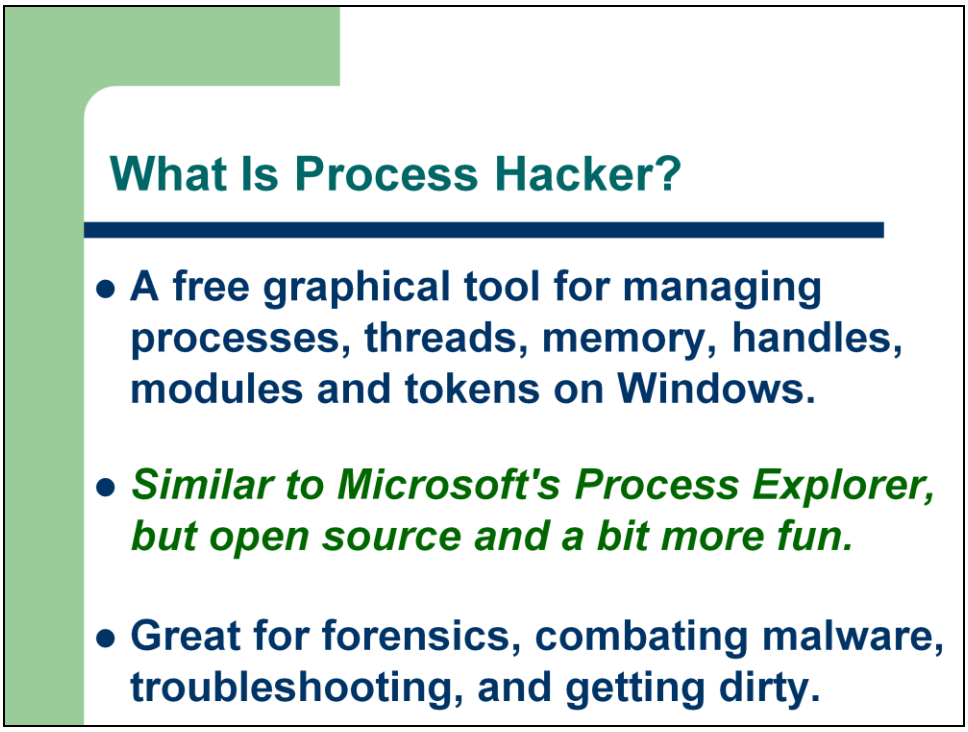
This talk will use the free, open source tool named "Process Hacker" to explore Windows processes, threads, handles and other operating system internals. You can get Process Hacker from **http://processhacker.sourceforge.net**. Having at least a basic understanding of these OS internals is important for combating malware, doing forensics, configuration hardening, troubleshooting and many other Windows tasks which require getting your hands dirty inside the OS. Please note that this presentation is intended to only apply to 64-bit Windows 7 and Windows Server 2008-R2, though many details will apply to other operating systems too. Also, if you prefer Sysinternals Process Explorer, that will work almost as well for this talk.

Windows Security at SANS

The six-day Securing Windows course at SANS (course number SEC505) is intended for those specializing in Microsoft Windows security. It covers Windows 7 and Windows Server 2008-R2, and will soon cover Server 2012 and Windows 8 too. The course author's blog is at **http://www.sans.org/windows-security/**, where you can also download the PowerShell scripts related to the course.

Recommended Reading

By far the most useful and authoritative book related to this talk is the latest edition of *Windows Internals* (Microsoft Press) by Mark Russinovich, David Solomon and Alex Ionescu. If you want to dig deeper into Windows, it is highly recommended that you get the latest edition of this book. After that, visit http://www.microsoft.com/sysinternals and download everything.



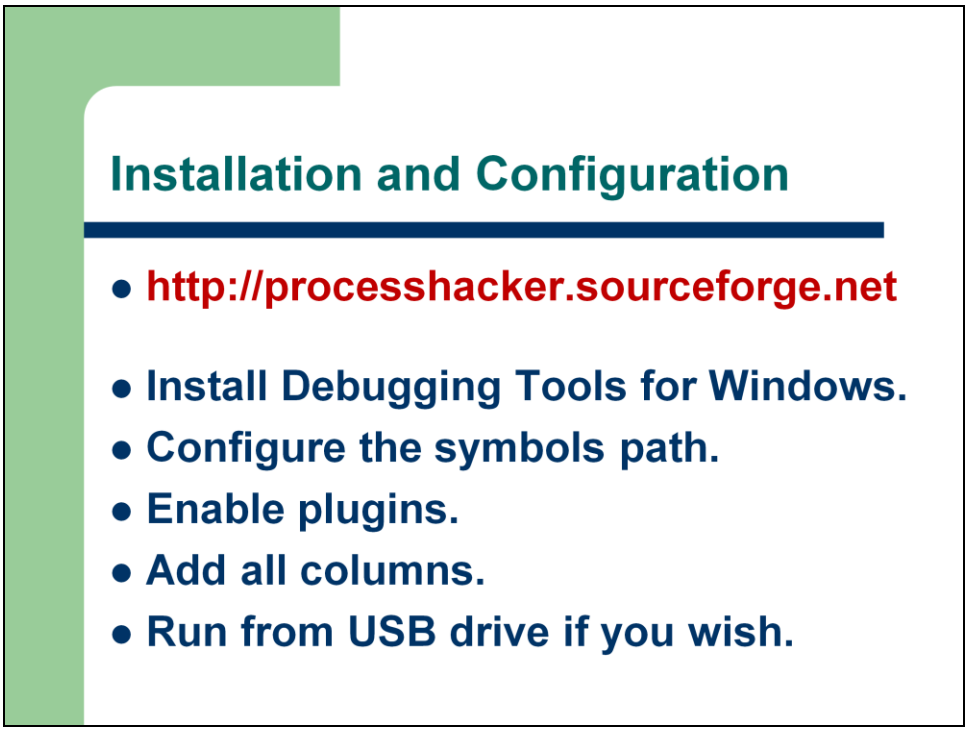
What Is Process Hacker?

Process Hacker is a free, open source, graphical tool for managing 32-bit and 64-bit Microsoft Windows processes, services, threads, memory, handles, modules, Security Access Tokens (SATs) and network connections. It is a wonderful tool for analyzing and combating malware, understanding low-level details of the Windows operating system, troubleshooting, and experimenting with Windows in ways which Microsoft never intended.

Process Hacker is similar to the famous Sysinternals Process Explorer tool from Microsoft, but open source and a bit more fun (http://www.microsoft.com/sysinternals). Now that Process Explorer is the property of Microsoft Corporation, Process Explorer cannot be enhanced with features which might be used to circumvent security restrictions or otherwise embarrass Microsoft. There are also no legal hassles when redistributing Process Hacker or its source code (no Microsoft lawyers = good thing). Examining the source code of Process Hacker is an interesting way to learn more about Windows internals, and Process Hacker itself is an actively maintained project.

Fortunately, if you prefer Process Explorer, almost all of this presentation applies to it as well. So please feel free to use Process Hacker or Process Explorer as you wish. Both tools are great.

And if you have questions, don't forget about the discussion forums for Process Hacker (http://processhacker.sourceforge.net/forums/) and Sysinternals Process Explorer (http://forum.sysinternals.com).



Installation and Configuration

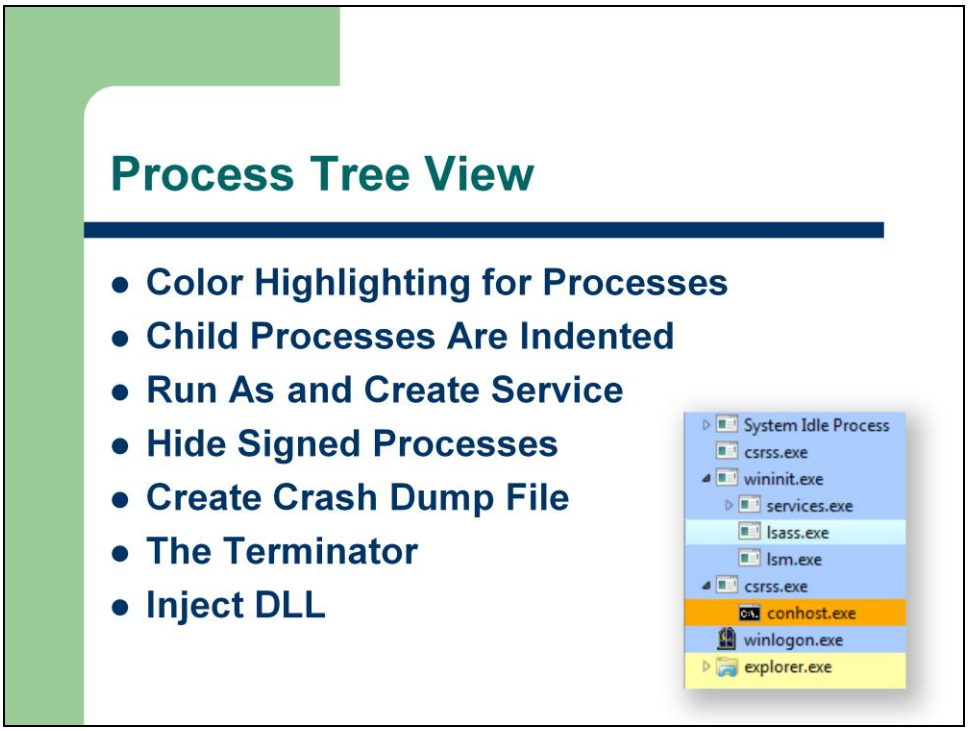
Process Hacker installs on 32-bit or 64-bit Windows XP-SP2 and later. Download binaries and source from http://processhacker.sourceforge.net. If you want to run Process Hacker from a USB drive, just copy the download zip file's contents to the USB drive and launch processhacker.exe with the -settings option with the path to your preferences file.

Make sure to install the latest version of Microsoft's "Debugging Tools for Windows" (http://msdn.microsoft.com/windows/hardware/), and then in Process Hacker, pull down the Hacker menu > Options > Symbols tab > confirm that you have a valid path to DBGHELP.DLL.

Next, in order to see the names of functions in stack traces, create a folder named "C:\Symbols" on your hard drive. Then, in Process Hacker, pull down the Hacker menu > Options > Symbols tab > and enter the following string into the Search Path text box exactly as shown, but with no double-quotes: "SRV*C:\Symbols*http://msdl.microsoft.com/download/symbols".

Next, in Process Hacker, pull down the Hacker menu > Options > General tab > check Enable Plugins, if it is not checked already. You can manage the plugins (Hacker menu > Plugins) and some plugins have their own configurable options, e.g., Extended Notifications (which are shown by right-clicking the icon for Process Hacker in the taskbar's notification area).

Finally, in almost every window with a table, you can right-click on the columns header and Choose Columns. When in doubt, add them all.



Process Tree View

A "process" is an environment in which a thread can run, providing a virtual memory address space, handles to objects, a security context, and other resources. A running process can spawn new processes, and thus there can exist parent-child relationships among processes. These parent-child relationships are shown as indentations in the "tree view" of Process Hacker. Every process contains at least one thread, and may contain many. A "thread" is that which has at least one stack and actually performs computing work within a process. A "job" is a collection of one or more processes, though a given process might not be a member of a job (look for a Job tab). A job allows a group of processes to be managed as a unit, such as placing restrictions on what those processes can do.

In Process Hacker, you can perform many tasks related to processes, including:

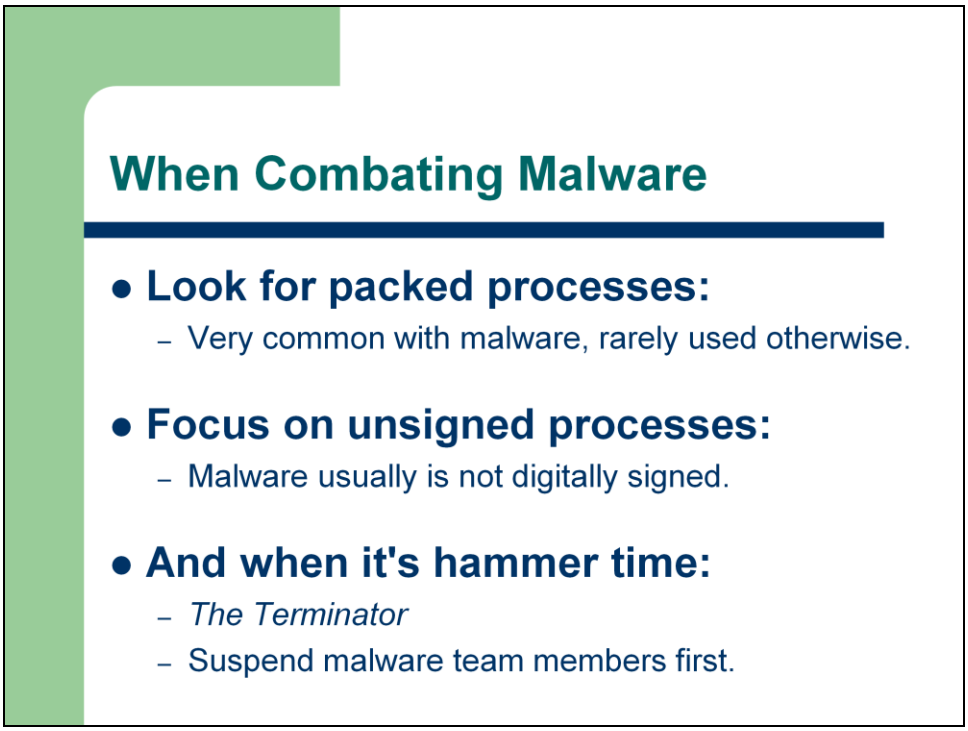
- Show different types of processes in different colors (Hacker menu > Options > Highlighting tab).
- Run commands as Local System in the desired session and desktop (Hacker menu > Run As).
- Create a new driver or interactive service (Tools menu > Create Service).
- Hide processes with verified digital signatures (View menu > Hide Signed Processes).

- Create a crash dump file of a process for analysis, but doesn't crash the process (right-click process > Create Dump File).

- Terminate almost any process, even if protected by rootkits or other security software (right-click process > Terminator).

- Inject a DLL or EXE into the memory of a running process (right-click process > Miscellaneous > Inject DLL).

Tip: You can run Process Hacker itself with Local System privileges (Hacker menu > Run As). You can also create a shortcut to do the same using the Sysinternals psexec.exe tool, like so "psexec.exe -s -d -i 1 processhacker.exe".



When Combating Malware

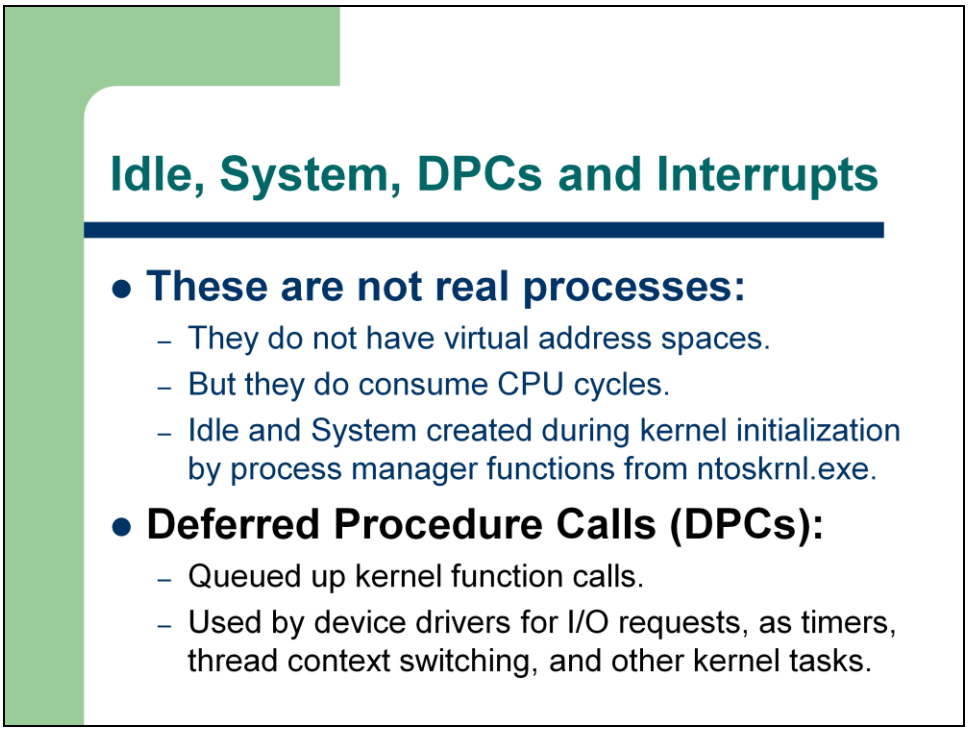
It is very common for malware executables to be packed. A "packed" process is compressed and/or encrypted, perhaps multiple times, along with a tiny bit of decompression/decryption code which is executed first in order to reconstruct (in memory, not on the hard drive) the working form of the malware. Some legitimate programs use packing to thwart reverse engineering, but this rare. Hence, enable highlighting of packed processes and focus on them first when combating malware. To scan the file system for packed binaries, use free tools like Mandiant's Red Curtain (www.mandiant.com).

Malware binaries are rarely digitally signed, so add the "Verification Status" and "Verified Signer" columns, then hide signed processes (View menu) to quickly identify those processes which lack a verified signature from a trusted code-signing certificate. Keep in mind, however, that it is still very common for legitimate programs to not have signatures. Also, a driver for the Stuxnet worm was digitally signed by a trusted issuer, so the presence of a signature does not automatically rule out that binary as being malicious.

Malware will sometimes try to hide its processes from tools like Task Manager and tasklist.exe. There are a variety of rootkit techniques for hiding processes, but some techniques affect only a few Windows API functions and user-mode data structures, while others hook kernel-mode native functions and modify deep kernel-mode objects too, which makes these cloaking techniques much harder to defeat. To try to find hidden processes on 32-bit systems (Tools menu > Hidden Processes), Process Hacker avoids simply asking the Windows API for a list of running processes and instead tries to open every possible process ID number individually using

a Windows API function which hopefully is not being hooked by the rootkit (this is the "brute force" method). Process Hacker can also examine every open handle to see what process it is associated with (the "CSR handle" method). If a process is visible by brute-forcing PIDs or following handles, but it's not visible when simply asking the Windows API for all running processes, then that process is considered hidden. Keep in mind, though, that these detection techniques can be beaten, so you'll need to use additional tools like GMER (www.gmer.net) and offline file system analysis to improve your odds of detecting any rootkits. Also, the hidden processes feature is not available on 64-bit Windows.

If you try to kill a malware process and it stubbornly refuses to die, right-click that process and select Terminator. The Terminator will try a variety of creative and aggressive methods of disrupting that process until it goes away. If you terminate a process and it simply reappears again, look for other malware processes or device drivers working as a team. You may need to right-click and Suspend each team member process before terminating all of them. If the malware process reappears after rebooting, you'll to investigate how it is persisting in the registry or file system using tools like Sysinternals AutoRuns.



Idle, System, DPCs and Interrupts

System Idle Process, System, Deferred Procedure Calls (DPCs) and Interrupts are not actually full processes in that they do not have virtual address spaces of their own, but they do consume CPU cycles.

There is one idle thread per CPU core. Idle threads aren't actually idle, they perform useful functions such as enabling/disabling interrupts, dispatching software interrupts for DPCs, dispatching some of the other threads waiting for a CPU, and granting kernel debuggers access to the OS. Idle threads run in a continuous loop as they check for work to perform. Any other thread can preempt an idle thread.

During kernel initialization, the first quasi-process created is the Idle "process" (it's not a real full-blown process) with PID = 0. Kernel bootstrapping proceeds later to create the System "process" (also not a full real process) and an Idle thread dispatches the first System thread. Idle is shown as the parent of the System process, but this is a fictional artifact of the kernel initialization sequence (winload.exe copies ntoskrnl.exe into memory and hands control to it, ntoskrnl.exe contains the functions collectively called the "Process Manager", and it's the Process Manager which creates both Idle and System).

Deferred Procedure Calls (DPCs) are requests for kernel functions to be executed on behalf of device drivers and other kernel components. Each CPU core has a queue of DPCs which are not executed immediately (they are "deferred") but only after the CPU has finished its current

higher-priority task. Device drivers use their Interrupt Service Routines (ISRs) to create DPCs to manage I/O requests. Kernel components use DPCs to handle things like timer expirations and performing context switches from one thread to another. DPCs execute at IRQ Level 2, so a DPC can interrupt any normal thread (which run at IRQL 0), but another hardware interrupt (IRQL 3-15) can interrupt a DPC.

Hal.dll imposes an IRQ Level scheme: IRQL 0 for user-mode threads ("passive", "low" or "normal" priority), IRQL 1 for Asynchronous Procedure Calls (APCs), IRQL 2 for DPCs ("dispatch" priority), IRQL 3+ for hardware interrupts. Only non-paged memory can be accessed at IRQL 2 or higher because page faults themselves are handled at IRQL 2. Threads have their own priorities (0-31), but a higher IRQL thread can always preempt a lower-IRQL thread, no matter what their relative thread priorities are.

Interrupts includes the CPU time spent processing hardware and software Interrupt Service Routines (ISRs). Often, an ISR will simply create a DPC to do the real work, but at a lower IRQL. CPU utilization for Interupts should be very low on average.

SMSS.EXE - The Session Manager

The Session Manager (smss.exe) is the first user-mode process created after kernel initialization. It is responsible for loading the rest of the registry, building the list of environment variables to be inherited by other processes, performing post-boot delayed file delete/rename changes, opening additional swap files, creating symbolic links for DOS device names like LPT1 and COM1, loads known DLLs into shared memory sections, and other tasks. More importantly, it also launches csrss.exe, wininit.exe and winlogon.exe.

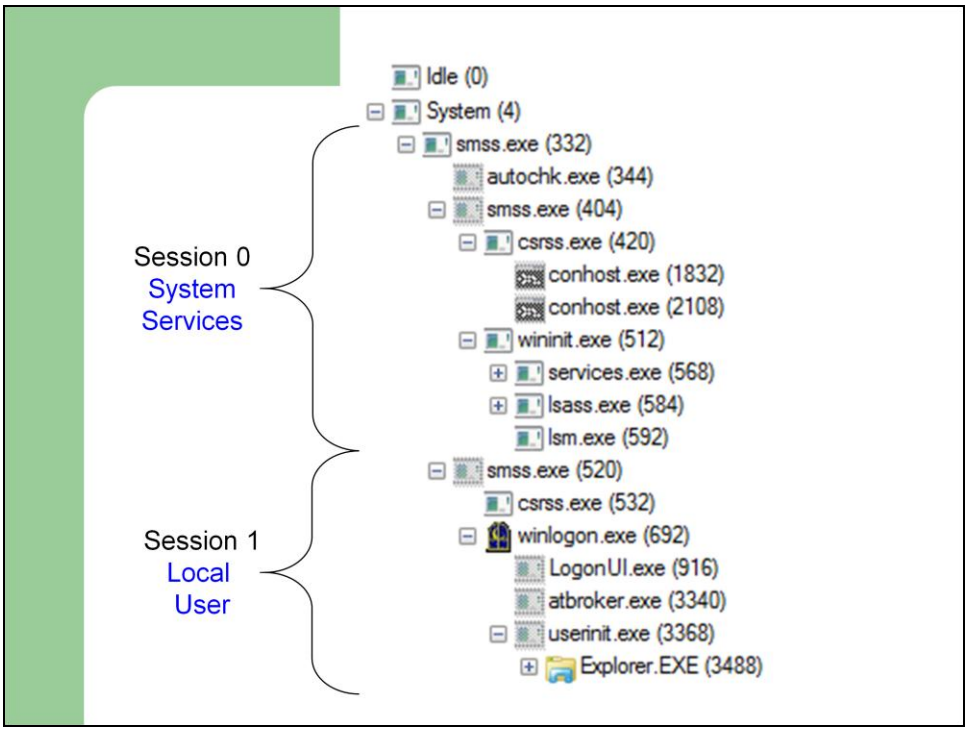
The first session created by smss.exe is for operating system services (session 0), not for human users. When session 0 is created, smss.exe copies itself into that new session, starts csrss.exe and wininit.exe (plus other tasks), then smss.exe exits (the original instance under System still runs). This is why in tree view you see csrss.exe and wininit.exe in session 0 without any parent processes, that is to say, why the session 0 csrss.exe and wininit.exe processes are left-aligned without any indentation. New sessions are created by smss.exe by its calling certain functions in the kernel's memory manager.

When a new user session is created (session 1 and greater), smss.exe copies itself into that new session, starts csrss.exe and winlogon.exe in that session (plus other tasks), then the smss.exe in the new session exits. This is why csrss.exe and winlogon.exe in session 1 are shown in the process tree without any parents. Remember, wininit.exe for session 0, winlogon.exe for user sessions greater than 0.

What Is A "Session" Anyway?

Loosely speaking, your session is all of the processes you own, plus your desktop.

Strictly speaking, a session is a small area of kernel address space which is not mapped into the virtual address spaces of all processes, but only into some, namely, into the processes originally created for the OS (session 0) or a user (session 1 or greater) by smss.exe and the child processes of these parents. This area of kernel memory is called "session space" and is shared by all the processes in the same session. Each session contains its own copy of win32k.sys and associated video drivers, a per-session directory in the object manager's namespace (\Sessions*), and a per-session paged pool of memory. The win32k.sys driver is that part of the Windows subsystem which manages the output of GUI windows on the desktop (USER, GDI and DirectX), gathers input from the keyboard and mouse, and sends input messages from the mouse or keyboard to the correct process. The processes in a session share the same station and desktop(s).

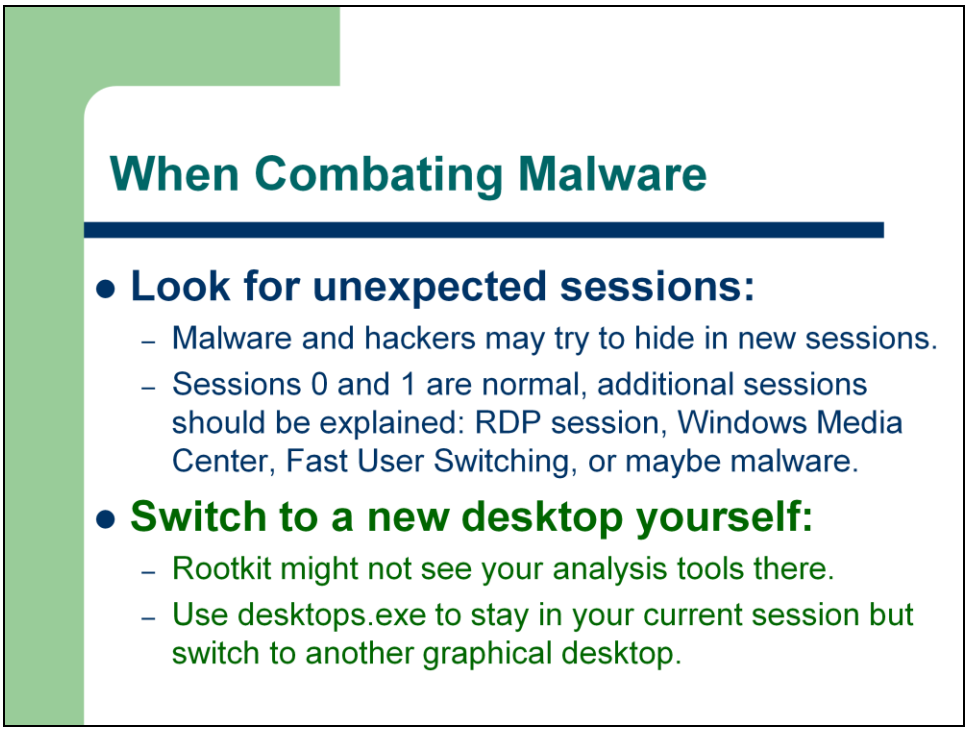


Sysinternals Process Monitor Tree

This screenshot was made with Sysinternals Process Monitor after configuring that tool for boot logging and then rebooting the computer (http://www.microsoft.com/sysinternals). The screenshot shows parent processes even if those processes are no longer running at the time the screenshot was made. It is an historical snapshot of parent-child process relationships. The processes no longer running when the screenshot was made are shown with greyed-out icons. The numbers in parentheses are their Process ID numbers (PIDs). The screenshot is from a computer running 64-bit Windows 7 Ultimate.

To see this yourself, in Process Monitor pull down the Options menu > Enable Boot Logging. Reboot the computer. Open Process Monitor again, and when prompted, save the boot log to a file. Then pull down the File menu > Open > load the boot log > Tools menu > Process Tree.

The screenshot shows how smss.exe creates session 0 for csrss.exe and wininit.exe, then smss.exe exits. In session 0, wininit.exe then runs services.exe, lsass.exe and lsm.exe. For session 1, smss.exe runs csrss.exe and winlogon.exe, then winlogon.exe spawns logonui.exe and userinit.exe



When Combating Malware

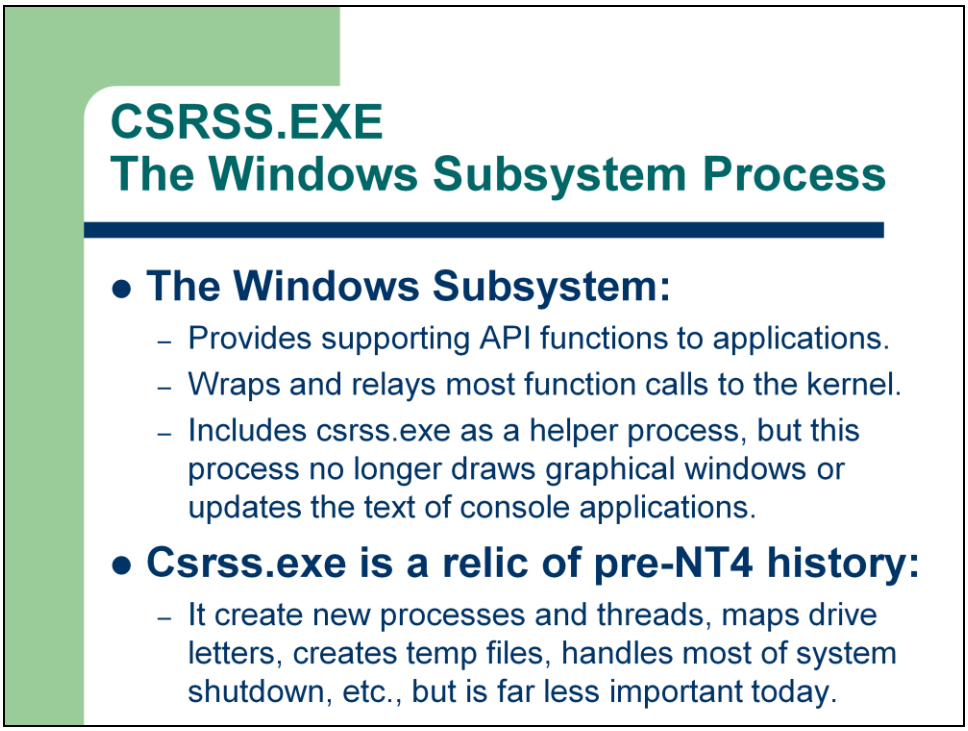
Sessions 0 and 1 are normal, but additional sessions should be explained. Additional sessions can come from RDP connections, remote Windows Media Center sessions with an extender (like an Xbox), and Fast User Switching on shared computers, but beyond these examples any unexpected sessions should be investigated. Because window stations and desktop objects have Access Control Lists (ACLs), it' possible that a rootkit might try to harden these ACLs and then hide the entire station.

A window station (or "winstation") contains a clipboard and one or more desktops. Each window station is a securable object, hence, each has an access control list. When a window station is created, it is associated with the calling process and assigned to the session of that process. The interactive window station, named "Winsta0", is the only window station which can display a graphical user interface and receive user input through one of its desktops. Each RDP remote desktop session has its own Winsta0.

A window station can have one or more desktop objects associated with it. The desktops associated with the interactive window station, Winsta0, can be made to display a user interface and receive user input, but only one of these desktops can be seen by the user at a time (this is called the "active desktop"). Windows messages can be sent only between processes that are on the same desktop. Each process which depends on the Windows subsystem is associated with a desktop, station and session (smss.exe and csrss.exe help implement the Windows subsystem, so they do not depend on it of course, so wininit.exe is the first).

Get the Sysinternals desktops.exe tool to play around with multiple desktops in your interactive window station. When you run this tool, you'll get a new icon next to your clock in the status area of the taskbar. Clicking this icon shows four different desktops, including the current one, to which you can switch. Each desktop has its own taskbar, program windows and "desktop" as we normally use that word in everyday conversation. Only one desktop can be active at any given time, but because they all share the same window station, there is one shared clipboard. If you initiate logoff in one of the desktops, you'll log off from all of them because you are really exiting your entire session.

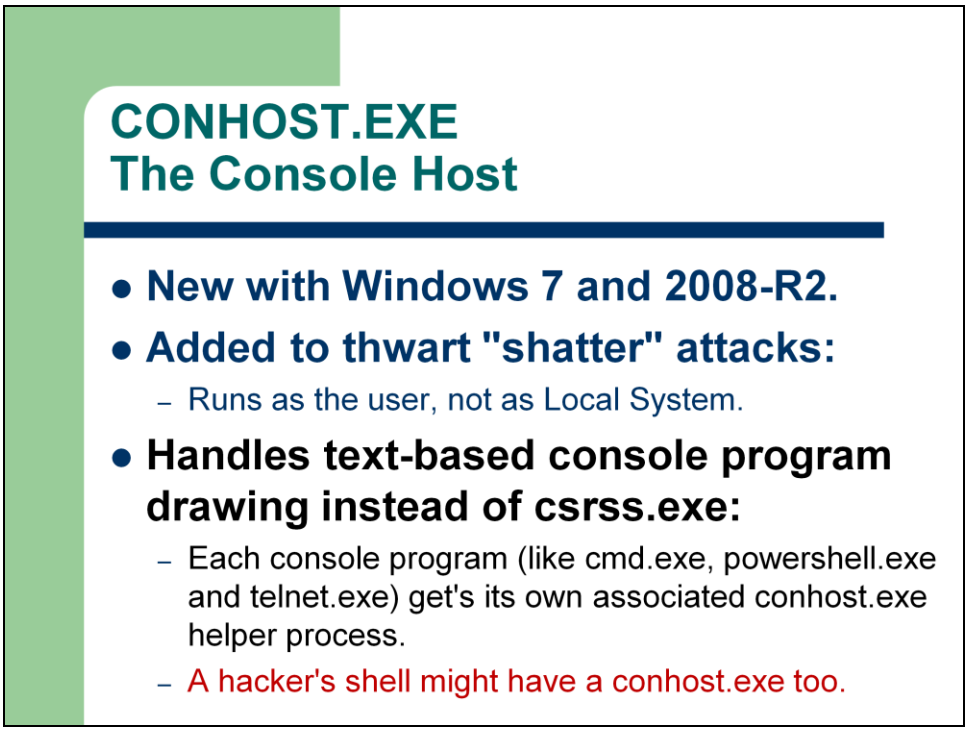
If you are combating malware which is blocking your forensics tools, you might try switching to another session or desktop in order to try to hide *from it*. For example, if you run the Sysinternals desktops.exe utility, you'll stay in your current session and window station, but you'll be able to switch among four desktops. Each desktop has its own graphical programs and Windows GUI messages can only be sent (or intercepted) between processes on the same desktop. Or with RDP or Fast User Switching, you could log on with a new session entirely, though still with the default desktop. So if you can't run your favorite AV or forensics tools in the default desktop because of rootkit monitoring, try running those tools in a new desktop instead.



CSRSS.EXE - The Windows Subsystem Process

A "subsystem" provides part of the environment in which processes can run. A subsystem makes various operating system functions available for processes to call as needed, such as for accessing the file system, using the network, or drawing graphical windows. The Windows subsystem loads DLLs into the address space of a process which can provide these functions themselves or relay these function calls down to the kernel or to other supporting processes. There are many DLLs that implement the Windows subsystem (kernel32.dll, advapi32.dll, user32.dll, gdi32.dll) and an additional one for the POSIX subsystem (psxdll.dll).

The Windows subsystem also includes csrss.exe as a helper process. Prior to Windows NT 4.0, csrss.exe was very important because it handled drawing graphical windows, but since then this functionality has been moved into the win32k.sys kernel-mode driver for performance reasons. Drawing text-based consoles used to be handled by csrss.exe too, but with Windows 7 and Server 2008-R2, even this functionality has been moved into a different process (conhost.exe) for security reasons. Spawning new processes and threads, mapping drive letters, handling much of the shutdown procedure, loading win32k.sys into the kernel at boot-up, and creating temp files is still managed by csrss.exe, but not much else of importance anymore, so csrss.exe is mainly an historical relic maintained for backwards compatibility. A future Windows version may eliminate it entirely…

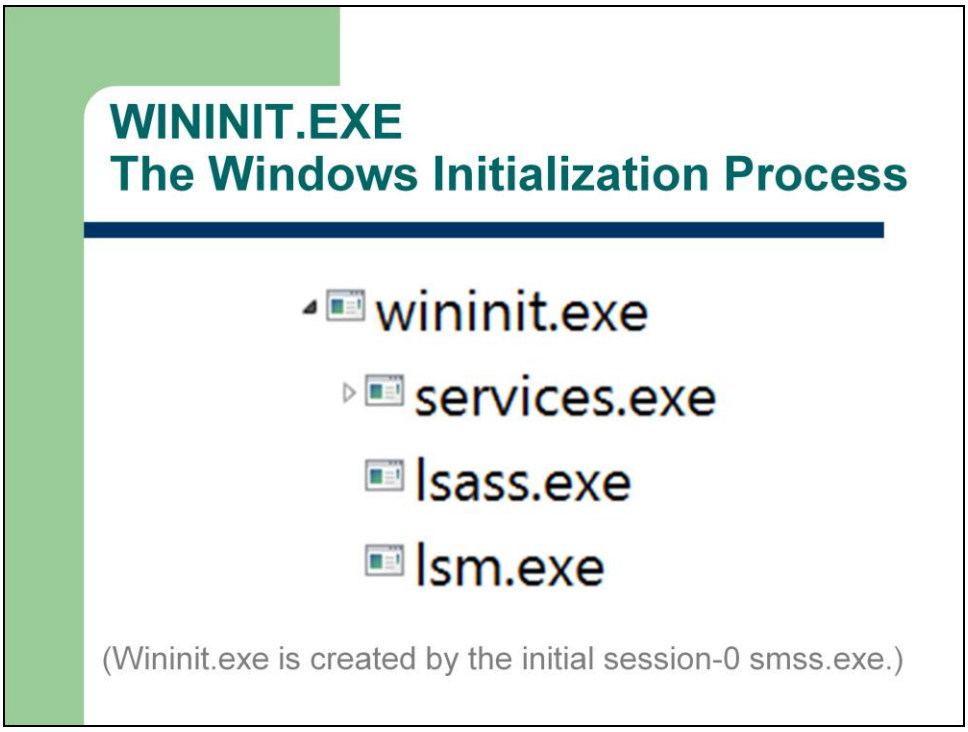


CONHOST.EXE - The Console Host

A console application does not use graphical windows, it is text-only; examples of console applications include cmd.exe, powershell.exe and telnet.exe.

In response to Chris Paget's "shatter" attack, Microsoft added conhost.exe to Windows 7 and Server 2008-R2. A shatter attack exploits a flaw in Windows' message-passing system; this system is not related to e-mail, it links user input and desktop processes together. A user's console application and its associated conhost.exe both run under the user's (limited) security context, while csrss.exe runs as Local System. A shatter attack to elevate privileges fails because conhost.exe is already running as the attacker and no Windows messages are sent through to csrss.exe anymore.

Every console application launched causes another conhost.exe to be spawned by csrss.exe in that user's session. Terminating a console application automatically terminates its associated conhost.exe. In the handles of a conhost.exe, you'll find a handle to the process object of the console application to which it is linked.



WININIT.EXE - The Windows Initialization Process

After the kernel is initialized and smss.exe is running, smss.exe creates session 0, copies itself into session 0, launches csrss.exe, launches wininit.exe, then that session-0 copy of smss.exe exits. At this point, csrss.exe and wininit.exe are still running in session 0.

The main job of wininit.exe is to 1) launch services.exe to start services and more device drivers, 2) launch lsass.exe to handle security, and 3) launch lsm.exe to do some of the session management. (Note: In Windows XP and Server 2003, lsass.exe was launched by winlogon.exe instead.)

LSM.EXE - The Local Session Manager

The Local Session Manager (lsm.exe) creates, destroys and manipulates sessions, mainly by directing smss.exe to do so. There is session 0 for services and session 1 is typically for the interactive user logon, but additional sessions may exist for the sake of Fast User Switching, Windows Media Center Extenders, and Remote Desktop Protocol (RDP) connections, such as RDP connections to a Server 2008-R2 machine running Remote Desktop Services.

When an RDP user connects, lsm.exe directs smss.exe to create a new session (including csrss.exe, winlogon.exe and logonui.exe). If there is already a desktop for that user, then the RDP connection is linked to that desktop and its currently-running processes in its current session. If a new desktop is required, then userinit.exe and explorer.exe run like normal, but in another new session. An exception to this is when an RDP connection is made for Remote

Assistance, in which case the RDP connection goes to the Remote Assistance Server (raserver.exe) and a new session is not created by smss.exe; instead, the RDP stream is simply linked to the desktop of the user being helped, either for passive viewing or for shared control of the mouse and keyboard.

The lsm.exe process never has any child processes of its own. Because multiple smss.exe processes can be spawned concurrently, it improves the performance of servers running Remote Desktop Services (formally known as "Terminal Services") when responding to many simultaneous logon requests.

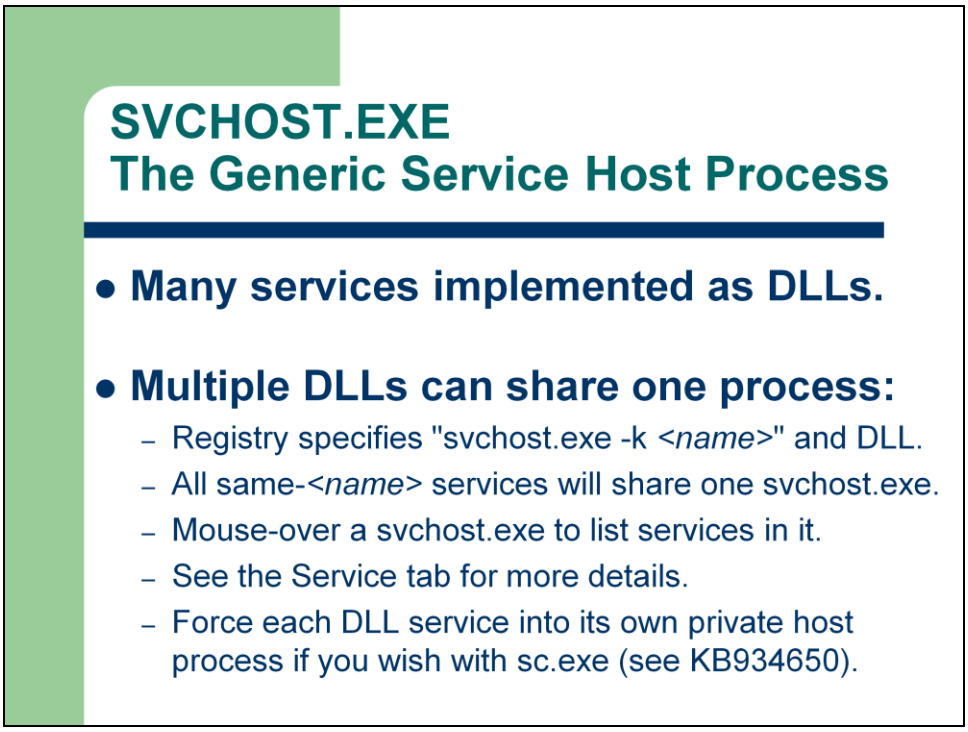
SERVICES.EXE - The Service Control Manager

The job of services.exe is to read keys in the registry representing services and device drivers (under HKLM\SYSTEM\CurrentControlSet\Services), then run these processes and load these drivers according to their dependency groups (as defined in HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List). Boot and system drivers are not loaded at this time, they are already loaded. When a service does not have a "Group" value to identify its dependency group, it is run afterwards. Triggered services are not executed by services.exe until after it has been notified by the event tracing facilities that the trigger has triggered.

By default, non-interactive services running as Local System are attached to an invisible window station named "Service-0x0-3e7$" in session 0; when these services are run interactively, they still run in session 0, but are attached to the Winsta0 station instead.

Windows services are not just fire-and-forget processes: services.exe maintains an in-memory database of service information, such as error state and recovery actions, and provides functions for querying, reconfiguring, starting, pausing and stopping services by name. Use the built-in sc.exe tool to interact with services.exe.

After a successful interactive user logon, services.exe will also make a backup copy of the keys under HKLM\SYSTEM\CurrentControlSet\Services known as the "last known good configuration", which can be restored at the next reboot with the F8 key.

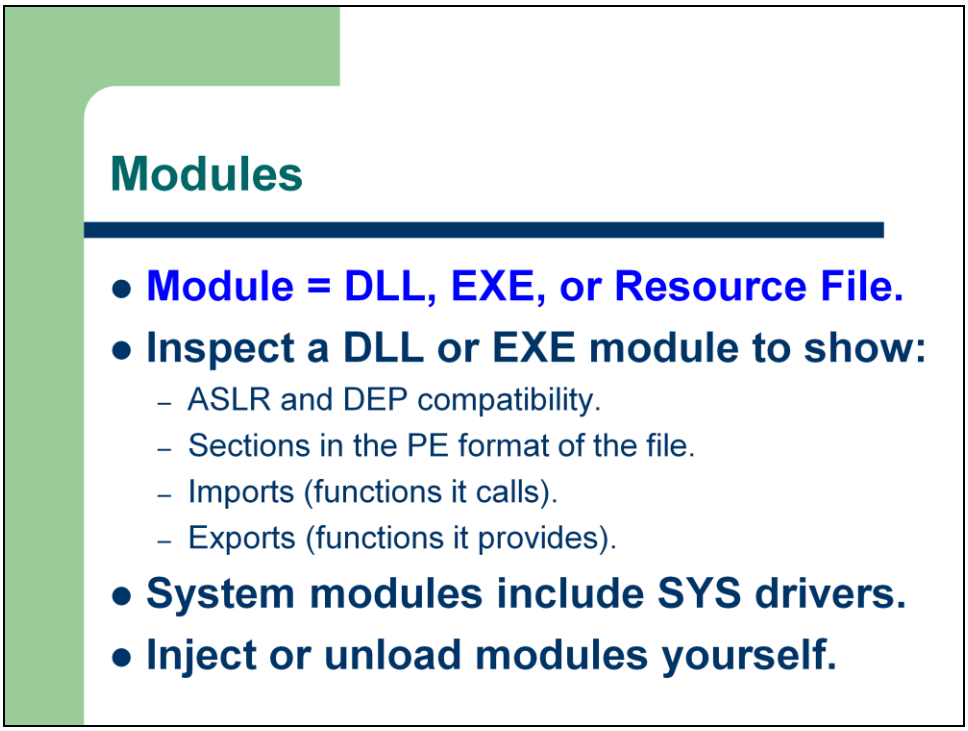


SVCHOST.EXE - The Generic Service Host Process

Many services are implemented as DLLs which can be loaded together into a generic shared svchost.exe process launched by services.exe. The registry entry for the service will include a ServiceDll value to identify the DLL to be used and a path to "svchost.exe -k <*name*>" to load it. All DLL-based services with the same "<*name*>" will share the same svchost.exe process; for example, both the DHCP Client and Windows Event Log services are DLL-based, and they share a single svchost.exe because both are launched with the "svchost.exe -k LocalServiceNetworkRestricted" command.

Hold your mouse pointer over a svchost.exe process to see a pop-up list of the services contained within it.

Sharing a single host process is nice for conserving OS resources, but not great for troubleshooting or analyzing infected services. You can force a DLL-based service into its own svchost.exe process with the sc.exe command (see KB251192 and KB934650).



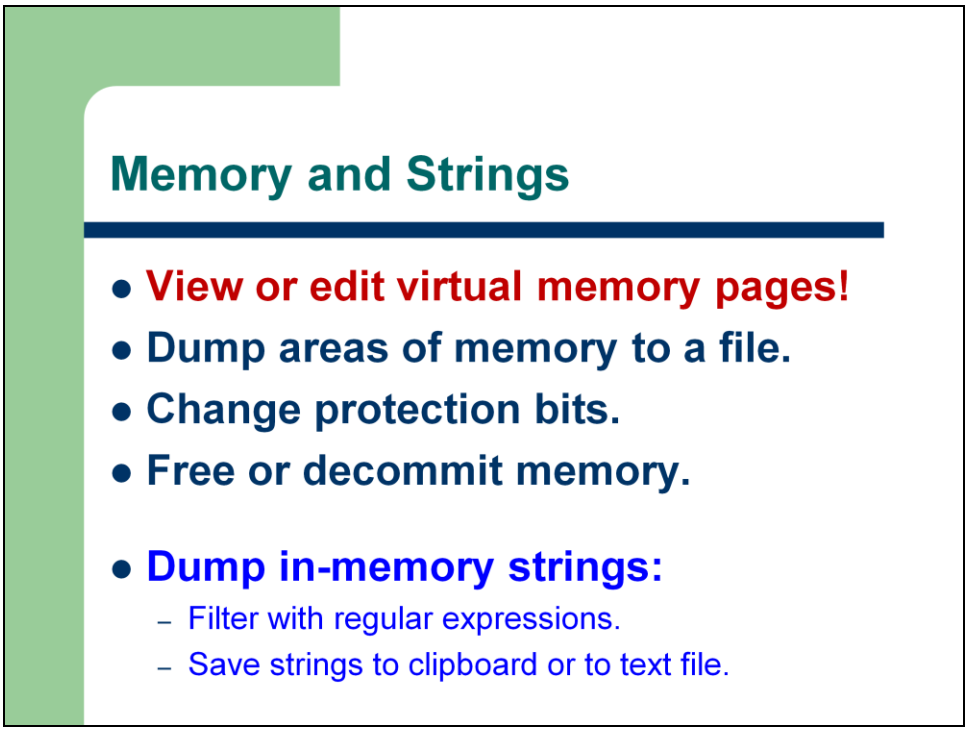
Modules And Digital Signatures

The modules of a process are the Dynamic Linked Libraries (DLLs), resource files and the original executable image (EXE) loaded into the memory address space of that process. The modules loaded into the System "process" at the top includes drivers (*.sys) and ntoskrnl.exe. Remember that System is not really a process.

In the Modules tab of a process, right-click the column headers, select Choose Columns, and add every possible column. Make sure to add the "Verification Status" and "Verified Signer" columns in particular, perhaps moving them near the top of the list. When combating malware, modules which fail their digital signature tests, or which have no signature, deserve additional analysis. You can add the same columns in the process tree view and then hide processes with verified signatures (View menu > Hide Signed Processes).

If you right-click a DLL or EXE module, you can select Inspect to show its Portable Executable (PE) sections, ASLR compatibility (Dynamic Base), DEP compatibility (NX Support), the functions it calls from other modules (Imports tab), and the functions it makes available to other modules (Exports tab). You can also add ASLR and DEP as columns.

You can also unload a module when you right-click it (which usually kills the process) or load a DLL or EXE into the process (go back to tree view, right-click the process > Miscellaneous > Inject DLL). When you inject a DLL, the operating system runs the DllMain() function within it automatically.



Memory and Strings

The 4GB virtual address space of an x86 32-bit process is typically divided into two regions: user space (0x00000000-0x7FFFFFFF) and system space (0x80000000-0xFFFFFFFF). The 16EB virtual address space of an x64 64-bit process is also divided into user space (0x0000000000000000-0x7FFFFFFFFFFFFFFF) and system space (0x8000000000000000-0xFFFFFFFFFFFFFFFF), though not all of this is addressable today due to software and hardware limitations.

Virtual-to-Physical Address Mapping

The user space is private to that process, except for shared memory sections; and the system space is common to all processes, except for a small area of system space called "session space", which is only common across processes in the same smss.exe session. An address in virtual memory can be common across processes because each committed page of virtual memory is backed by or "mapped to" a page of real physical memory, and two virtual memory addresses (like in two processes) can be mapped to the same physical address. Hence, a DLL can be loaded into physical memory once, then mapped into the system space of all processes using virtual addresses, saving a great deal of physical memory.

Kernel Mode vs. User Mode

The CPU can be in "kernel mode" or "user mode". The CPU must be in kernel mode in order to read/write memory in system space. The CPU can be in either kernel mode or user mode to read/write memory in user space. When in kernel mode, the CPU can execute any instruction it

supports and can read/write any memory anywhere. Malware running in kernel mode can do anything. The CPU and OS tightly control the switching of the CPU's mode in order to try to maintain security.

Reserved vs. Committed Memory

A process can "reserve" an area of virtual memory, which means those addresses have been set aside for possible future use, or the process can "commit" an area of memory, which means those virtual addresses have actually been mapped to physical memory. Reserved virtual memory does not consume any physical memory, but committed virtual memory does consume physical memory.

Free, Mapped, Image or Private Memory

"Free" virtual memory is neither reserved nor committed, but can be so as needed. "Mapped" memory is either part of a heap, a section of memory shareable with other processes, or all/part of the contents of a file. "Image" memory contains the contents of file, usually a binary, cached in memory. "Private" memory is for the private use of the process, such as for a thread stack. (See the Sysinternals vmmap.exe and rammap.exe tools for a better view of memory use.)

Protection Bits

The page tables in system space, which do all the virtual-to-physical address mappings, also contain per-page protection bits, similar to permissions. An area of memory can be marked as read-only (R), write (W), execute (X), some combination of these, plus a few other markings used internally for housekeeping. Finding mapped, committed memory marked as RWX could be interesting, especially if a dump of the region began with "MZ" and "This program cannot be run in DOS mode" (Stuxnet did this, for example).

Memory Tab in Process Hacker

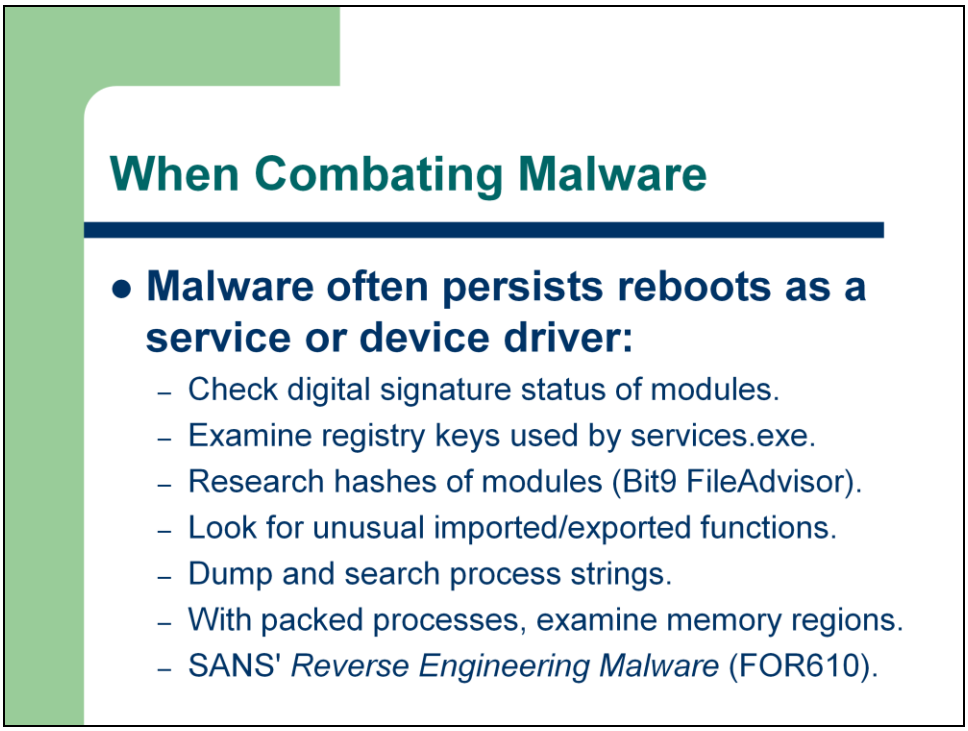
The Memory tab in Process Hacker permits several useful actions:

- View or edit areas of virtual memory (right-click > Read/Write Memory).
- Save an area of memory to a binary dump file (right-click > Save).
- Change the protection bits, with limitations (right-click > Change Protection).

- Free or decommit memory (right-click > Free or Decommit). Decommitted memory is still reserved.

- Dump the ASCII and/or Unicode strings found in the process virtual address space, then filter these strings using a regular expression pattern (Strings button > OK > Filter button). Strings can be copied to the clipboard or saved as a TXT file.

For combating malware, the ability to dump and search strings from in-memory data is very important. The strings of a process are helpful in identifying malware types and analyzing their purposes and capabilities. If you wish to script the analysis of strings from the command line, get the Sysinternals strings.exe tool.



When Combating Malware

If you are a piece of malware, you've got a persistence problem: how to survive reboots? You'll have to somehow modify the hard drive, BIOS or device firmware, because main memory is reset each time the power button is pressed. A common way to survive reboots (and get loaded automatically with elevated privileges too) is for the malware to get loaded by or as a service or device driver. By modifying an existing binary which is already loaded by services.exe, the malware can survive reboots without changing the registry. Or by adding a new binary and editing the registry, the malware can again be run by services.exe automatically.

So, when combating malware, pay special attention to the child processes of services.exe and the associated registry keys. In particular, it's common for malware to load a DLL into a svchost.exe because there are so many of these processes. An infected machine might also have a malicious device driver visible in the Modules tab of the System "process", like Stuxnet, or rootkit tricks might be in use to hide them.

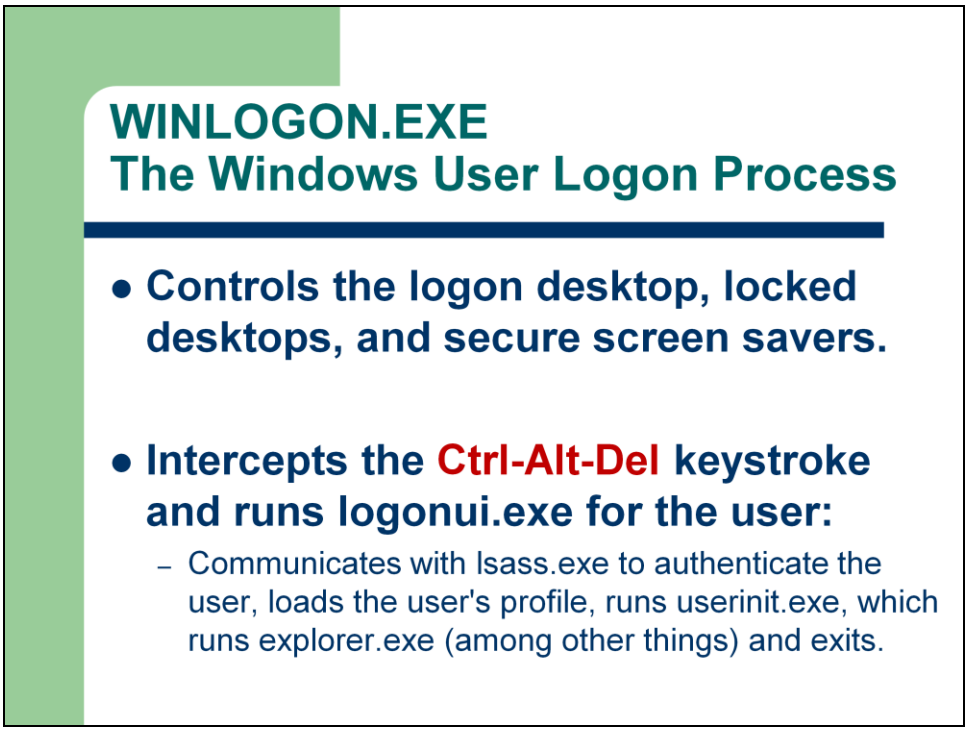
Make sure to check the digital signature status of modules and device drivers, confirm that the hashes of these binaries are known to be good (like with http://fileadvisor.bit9.com), examine the imported and exported functions of suspicious modules (look up the function names in Google/Bing), examine the strings of a process (look for strange URLs, file system paths, embedded passwords, or other "hinky" data), and with packed processes in particular you may need to examine raw memory regions (these regions can be dumped to .bin files also). If these tasks seem difficult (because they are) you might consider taking Lenny Zeltser's *Reverse Engineering Malware* course (FOR610) or Rob Lee's *Advanced Forensics* (FOR508).

LSASS.EXE - The Local Security Authority Subsystem

Another process launched by wininit.exe is lsass.exe, which is a special shared-host process for security-related services like Active Directory on a domain controller (ntdsa.dll, kdcsvc.dll), Netlogon (netlogon.dll), Security Accounts Manager for local accounts (samsrv.dll), and Crypto Next Generation Key Isolation (keyiso.dll).

Many critical operations are handled by lsass.exe, such as user authentication, enforcing password policies, creating handles to new Security Access Tokens (SATs), writing to the Security event log, and maintaining the Netlogon RPC secure channel with a domain controller. Not surprisingly, this process is the target for many attacks, such as DLL-injections to dump password hashes or to perform pass-the-hash impersonation attacks.

Incidentally, the Stuxnet worm created multiple lsass.exe processes at various times during its lifecycle, precisely because this process name is well-known and trusted.



WINLOGON.EXE - The Windows User Logon Process

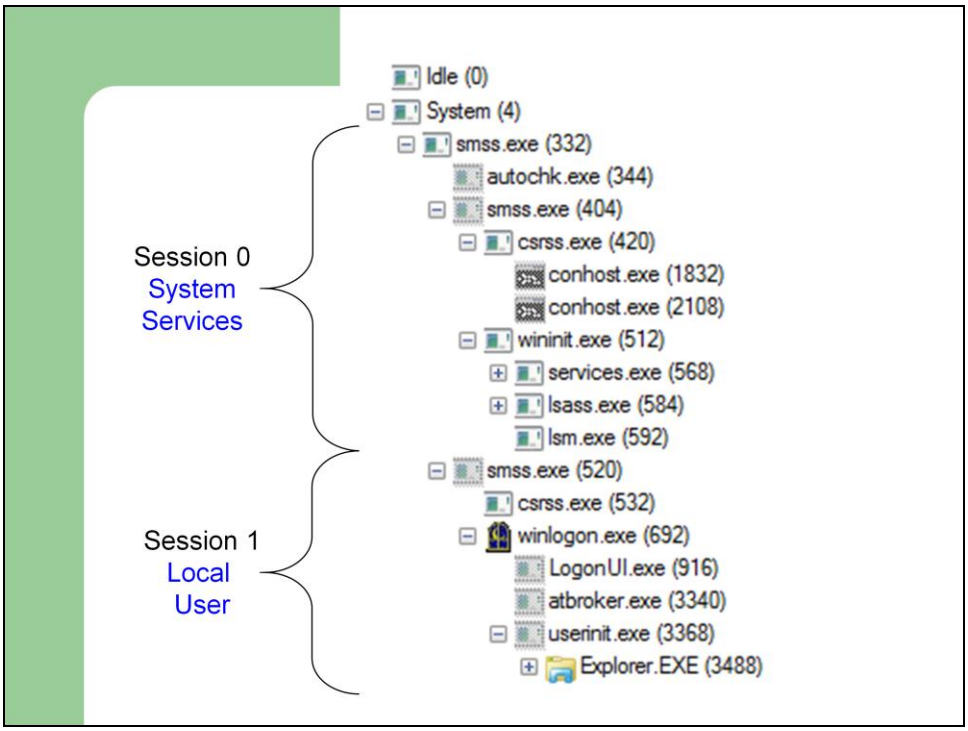
Right after boot-up, the original smss.exe creates session 1, copies itself into session 1 as a new process, launches csrss.exe and winlogon.exe, then that smss.exe exits, leaving csrss.exe and winlogon.exe still running. Next, winlogon.exe creates a window station (Winsta0) and displays the interactive desktop, which means winlogon.exe is now in control of the local keyboard, mouse and monitor (and running as Local System).

Whenever a user hits Ctrl-Alt-Del, winlogon.exe switches to another desktop and launches a special program, logonui.exe, to interact with the user. The user may be logging on initially, (un)locking the desktop, changing her password or some other task, but the user is interacting with logonui.exe on a special desktop, not winlogon.exe on the default desktop. This special desktop in which logonui.exe runs is called the "secure desktop" because its permissions only allow winlogon.exe to manage it, thus hopefully preventing trojans from capturing keystrokes. When a user locks her desktop, it's actually winlogon.exe doing the locking.

When authenticating, logonui.exe loads DLLs called "credential providers" which can handle the password, smart card or, with a third-party provider, biometric information, to authenticate against the local SAM database, Active Directory, or some other third-party authentication service. The resulting credentials are given to winlogon.exe, which calls functions within lsass.exe to do the actual authentication checking. If the authentication succeeds, lsass.exe gives winlogon.exe a handle to a Security Access Token (SAT) for the user, then winlogon.exe uses that token to set permissions on the interactive desktop (not the secure desktop) for the user's benefit, loads the user's profile, launches userinit.exe with that user's token, and then

userinit.exe typically runs explorer.exe. Once userinit.exe finishes its other tasks, such as running logon scripts, it exits, leaving explorer.exe still running (which is why explorer.exe doesn't have a running parent in tree view).

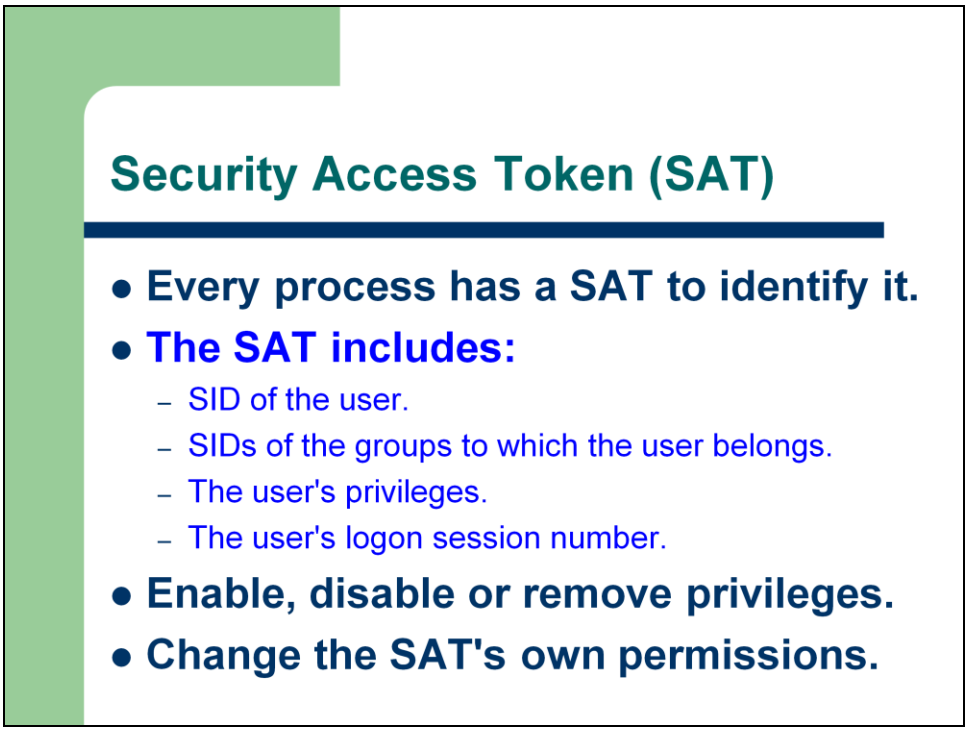
Incidentally, when user environment debug logging is enabled (KB221833), it is mainly winlogon.exe writing to the userenv.log file. And while userinit.exe and explorer.exe are the typical initial processes, registry edits permit different or additional processes to be launched for the user instead (for example, when an Xbox opens a Windows Media Center session on a remote computer as an extender, userinit.exe runs mcrmgr.exe instead). Finally, atbroker.exe helps users with disabilities navigate winlogon.exe's desktops and logonui.exe's prompts.



Sysinternals Process Monitor Tree

This screenshot was made with Sysinternals Process Monitor after configuring that tool for boot logging and then rebooting the computer (http://www.microsoft.com/sysinternals). To see this yourself, in Process Monitor pull down the Options menu > Enable Boot Logging. Reboot the computer. Open Process Monitor again, and when prompted, save the boot log to a file. Then pull down the File menu > Open > load the boot log > Tools menu > Process Tree.

The screenshot shows how smss.exe creates session 1 for csrss.exe and winlogon.exe, smss.exe exits, then winlogon.exe spawns logonui.exe and userinit.exe. Credentials are collected by logonui.exe, given to winlogon.exe, which checks them with lsass.exe, then winlogon.exe runs userinit.exe to create the user's desktop environment for explorer.exe to create the Start menu, taskbar and desktop icons. Explorer.exe is run by default as the user's initial process, but actually any process could be launched first instead.

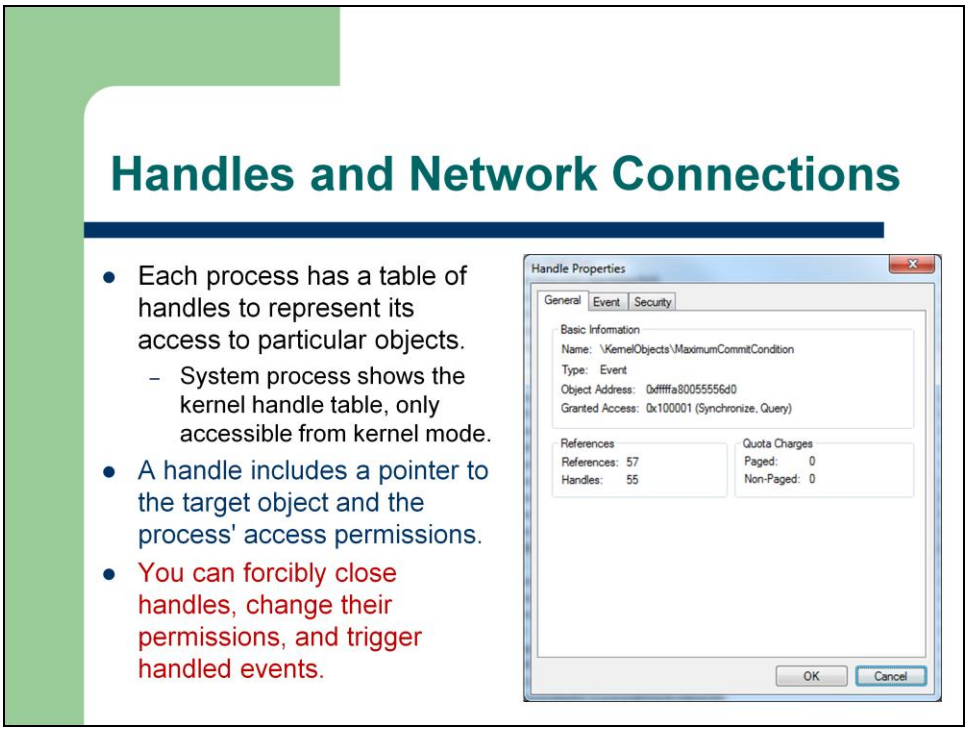


Security Access Token (SAT)

Every process and thread has an associated Security Access Token (SAT) to represent the identity under which that process or thread is running. The SAT includes the user's Security ID (SID) number, the SIDs of the groups to which the user belongs, the user's privileges, the session number, and other information. The SAT is created by lsass.exe (and the kernel's security reference monitor) when a user logs on, then it is attached to every process the user launches.

Not only can you view SAT data on the Token tab of a process, you can also enable/disable/remove privileges (but not add them) and modify the permission on the token object itself.

When User Account Control (UAC) modifies the SAT in an un-elevated process, you can also see the original elevated SAT (the Linked Token).

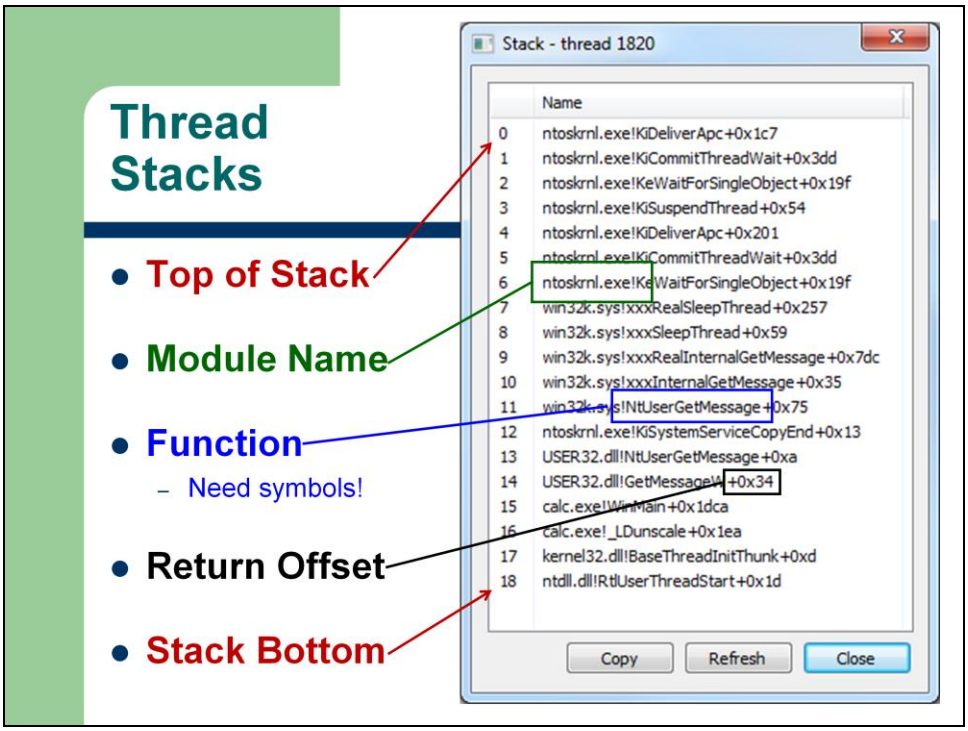


Handles

Every process has an associated table of handles. A "handle" represents the access of a process to another object. Some of these objects exist only in the kernel, such as threads, tokens, events, mutexes, timers, window stations, sessions, ports and processes; while other objects are created by the kernel to represent "real" resources to which the OS controls access, such as files, directories, shared memory sections, and registry keys. A handle includes a pointer to the target object, as well as a list of access permissions. The permissions differ depending on the type of the object. Processes can acquire handles by creating new objects, requesting the object by name, inheriting the handle from parent processes, and other methods. Before a thread in a process can interact with an object, the process must obtain a handle. In the kernel, only the object manager can create handles, and it does so only after checking permissions with the security reference monitor.

On the Handles tab of a process, not only can you view handles, you can forcibly close them, change their permissions, and change their "Inheritable" and "Protect from Close" flags. If the handle is to an event, you can also trigger that event by hand and see what the process does -- it's fun! If you right-click a process in tree view and run the Terminator, you can try to close all of its handles in one shot (double-click CH1).

When you examine the handles of the System process, you are seeing a different handle table: the kernel handle table. Kernel handles are only accessible in kernel mode, but from within *any* process in kernel mode.



Thread Stacks

Each process must have one or more threads. A thread has a stack (the process does not) and the thread is what actually performs work (the process is just a supporting environment for threads). The Threads tab of a process shows the current threads in that process. Each thread has a unique ID number (TID), a multitasking priority for scheduling by the dispatcher, and a starting address for a function in a module.

If you have symbols configured, the start address of a thread will be formatted like "*module* **!** *function*", where *module* is the DLL or EXE which asked the OS to create the thread and *function* is what was given to the OS as the first thing the thread should do. If you don't have symbols configured, the start address will simply show "*module +number*" where *number* is a hex offset into the module, the offset location of a function.

Manipulate Threads

If you right-click a thread, you can terminate, suspend or resume it. You can also cancel any synchronous input/output procedures it is carrying out, change its CPU affinity, multitasking priority, input/output priority, and permissions. If the thread manipulates a graphical desktop window, it will usually be shown with yellow highlighting, and you can right-click and select Windows to see which window objects these are.

Inspect Stack

If you right-click a thread and select Inspect (or just double-click the thread), you will see the

thread's stack. The bottom of the stack is historically the first function call, then later function calls appear as you move up the stack in the dialog box, each function calling the one above it, from bottom to top. When you get to the top of the stack, you'll see the most recent function call.

Strictly speaking, most threads begin life as a call to "ntdll.dll **!** RtlUserThreadStart", so that is what's shown at the bottom of the stack, but that wouldn't be very informative if shown on the Threads tab; the thread's interesting starting function (as shown on the Threads tab) is typically the second or third function call up from the bottom of the stack.

Line 14 in the screenshot looks like "USER32.DLL **!** GetMessageW+0x34". The module is "USER32.DLL", the called function is "GetMessageW", and the return address offset is "0x34". What is the return address offset used for? Notice that line 14 shows that GetMessageW called another function above it named "NtUserGetMessage" in line 13. Later, when NtUserGetMessage finishes (line 13), control should return to the address of GetMessageW (line 14) plus 0x34 bytes in the stack. The return address offset indicates the address where control should resume when a called function is finished. Remember, the stack is a data structure with addresses and data, not the executable bodies of the functions themselves, so the return address is somewhere in the stack.

Misc Notes

Sometimes in a stack trace you'll see strange "_", "@" and "?" characters. These come from the "decorations" of the symbols in the symbol files. The decorations indicate the function's calling convention, the function's number of argument bytes, whether it is a symbol from a precompiled header (pch), an imported symbol from another module, linker-generated strings, or other low-level details too numerous to describe here.

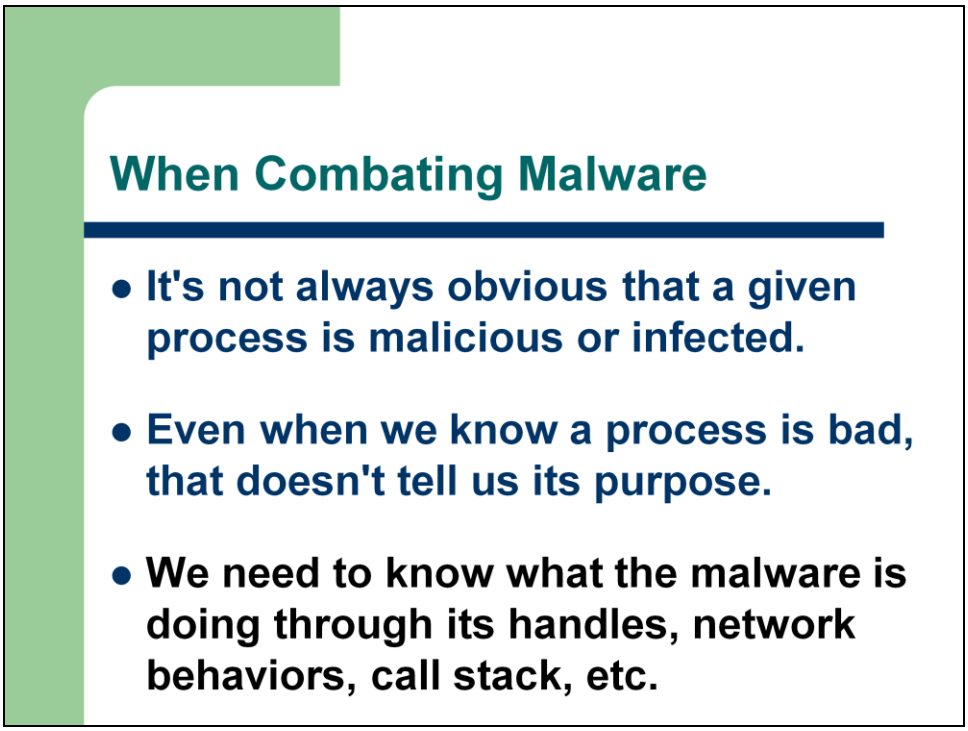
If the *function* name has two colons in it, the function is a method of a class ("*class* :: *function*"), probably a C++ class.

You will often see function calls in the stack with a name like "*Wait*" since most threads spend most of their time just sitting and waiting for other events to occur. And when an event occurs which does kick a thread into motion, it acts literally faster than a human can blink, so you can't really see the stack evolve in real-time with your eyes anyway (unless you're using a debugger and going step-by-step). Please don't be

disappointed.

Each user-mode process has two stacks per thread: one user-mode, one kernel-mode. Additionally, each CPU core has its own Deferred Procedure Call (DPC) stack too.

When you see a thread referred to as "2c5.4ba", the "2c5" is the process ID number in hex and the "4ba" is the thread ID number in hex, hence, "2c5.4ba" refers to PID 709 with TID 1210.



When Combating Malware

When you are not certain whether a process is malicious or not, examining the SAT, handles, listening ports, network connections, and thread stacks of that process can help.

Often, malware requires elevated privileges in order for the initial infection to succeed (such as having the Debug Programs privilege) or will seek to raise the privileges of a victim process; in either case, viewing the Security Access Token (SAT) of the process helps to understand what it can and cannot do.

The handles and networking characteristics of a process reveals what that process is interested in and what it may be doing. If a process appears to be one thing (a game, for example), but has handles, listening ports and live network connections which are very unusual for that type of program, it's a sign that the process may be infected or malicious.

The imported and exported functions of a process can be very revealing, and so can the stack traces of the threads in a process for the same reasons. When viewing stack traces, we are getting a glimpse into what a process is doing at a moment in time. When a process appears to be one thing, again, but we see strange function calls for that supposed type of process, it's another sign that the process may be infected or malicious.

Thank You for attending!

Please fill out the evaluation form. All written comments are read and appreciated!

For the SANS Windows Security blog, please visit:

http://www.sans.org/windows-security/

For the SANS Institute's *Securing Windows* course (SEC505), please visit:

https://www.sans.org/security-training/courses.php

(v.2.0)