Chapter 3. Processes and jobs

Overview:

- Process creation

- Process internal structures

- Protected processes vs Non-protected

- Process creation specifics

Contents

Creating a process

CreateProcess – new process has same access token as creating process

CreateProcessAsUser – takes handle to token object of a user (possibly obtained by LogonUser function)

CreateProcessWithLogonW – log on with a given user's credentials and create a process

CreateProcessWithTokenW/CreateProcessWithLogonW: Calls Secondary Logon service (seclogon.dll) and makes RPC to create process

SecLogon service calls internal SlrCreateProcessWithLogon which calls CreateProcessAsUser

Fail if SecLogon fails to start

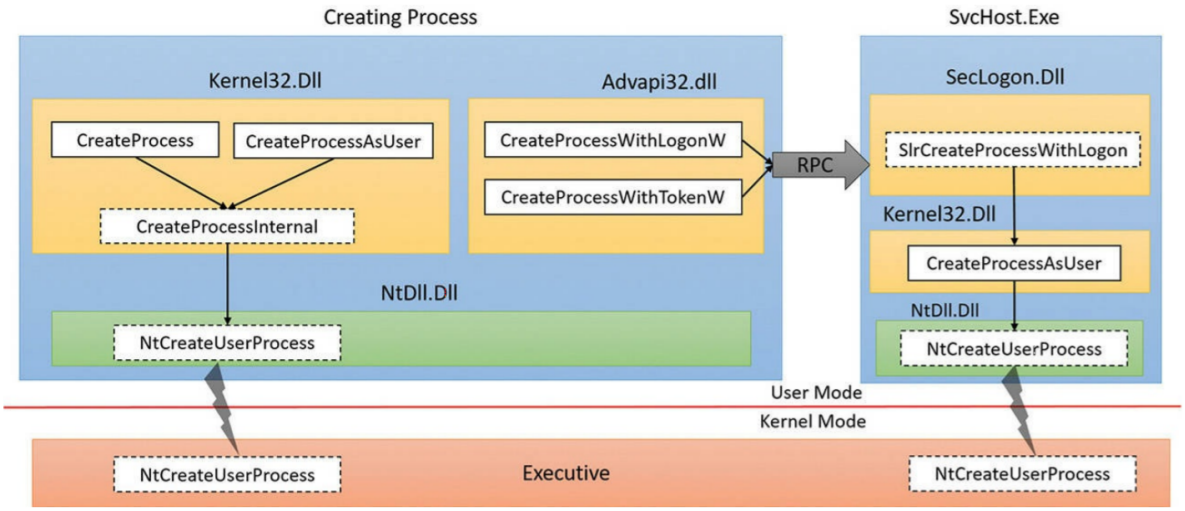

FIGURE 3-1 Process creation functions. Functions marked with dotted boxes are internal.

Windows Shell – connect files with certain extensions to an executable (.txt → Notepad)

ShellExecute, ShellExecuteEx – accepts a file and locates executable to run based on HKEY_CLASSES_ROOT

Calls CreateProcess and appends arguments

All process creation leads to CreateProcessInternal, which calls NtCreateUserProcess (Ntdll.dll) to transition to kernel mode and call NtCreateUserProcess (kernel mode version)

CreateProcess* functions arguments

CreateProcessAsUser and CreateProcessWithTokenW – needs user token handle

CreateProcessWithLogonW – need username, domain, and password

All need executable path and command line args

Security attributes (optional) for the new process and thread object

Boolean flag for inheriting handles or not (inherited handles are copied to the new process)

Process creation flags

CREATE_SUSPENDED: create new proc in suspended state, call ResumeThread to start

DEBUG_PROCESS: creating process is declaring that it is a debugger and it plans to debug the new process

EXTENDED_STARTUPINFO_PRESENT: when you provide STARTUPINFOEX instead of STARTUPINFO

Environment block (optional, otherwise inherited from parent)

Current directory for new process (optional, otherwise uses parent's current directory)

STARTUPINFO, STARTUPINFOEX process creation config options, STARTUPINFOEX has key/value fields that can be updated via UpdateProcThreadAttributes

PROCESS_INFORMATION structure, output of successful process creation, holds the new pid, thread id, handle to new proc, and handle to new thread

Creating Windows modern processes

Modern apps - UWP apps, immersive processes (as oppose to classic desktop applications)

To create modern app process - need to use UpdateProcThreadAttribute with the key PROC_THREAD_ATTRIBUTE_PACKAGE_FULL_NAME and value of the full store app package name

Can also call ActivateApplication in interface, IApplicationActivationManager (implemented by CLSID_ApplicationActivationManager)

Launches store app after calling GetPackageApplicationIds to retrieve AppUserModelId from store app full package

Creating other kinds of processes

Native processes, minimal processes, Pico processes cannot be started with Windows API

Ex: Session Manager (Smss) calls NtCreateUserProcess (since it's created by kernel)

Windows API not available when Smss calls Autochk or Csrss

Cannot create native processes from Windows applications because CreateProcessInternal rejects native subsystem image types

Native processes:

RtlCreateUserProcess (Native library, Ntdll.dll) wraps NtCreateUserProcess

Kernel-mode processes:

NtCreateProcessEx system call (has special capabilities for kernel-mode callers only such as creating minimal processes)

Pico providers:

PspCreatePicoProcess – creates minimal process and initializes Pico provider context

NtCreateProcessEx and NtCreateUserProcess – diff system calls, same internal routines:

PspAllocateProcess, PspInsertProcess

Process internals

For each Windows process…

EPROCESS (executive process) structure: represents a Windows process

Points to ETHREAD structure

Resides in system address space

Exception: Process Environment Block (PEB) exists in user address space

Encapsulated as process object by executive object manager

Access data in this structure via process handle

CSR_PROCESS: parallel structure maintained by Windows subsystem process (Csrss)

W32PROCESS: per-process data structure maintained by Win32k.sys (kernel mode of Windows subsystem)

Created when thread calls Windows USER or GDI functions implemented in kernel mode

Happens when User32.dll is loaded (CreateWindows(Ex), GetMessage)

DXGPROCESS: Direct X Graphics Kernel structure

DirectX objects and GPGPU counters and policy settings

PsSetCreateProcessNotifyRoutine(Ex, Ex2) allows system components and drivers to register process-creation notifications to track information on per-process basis

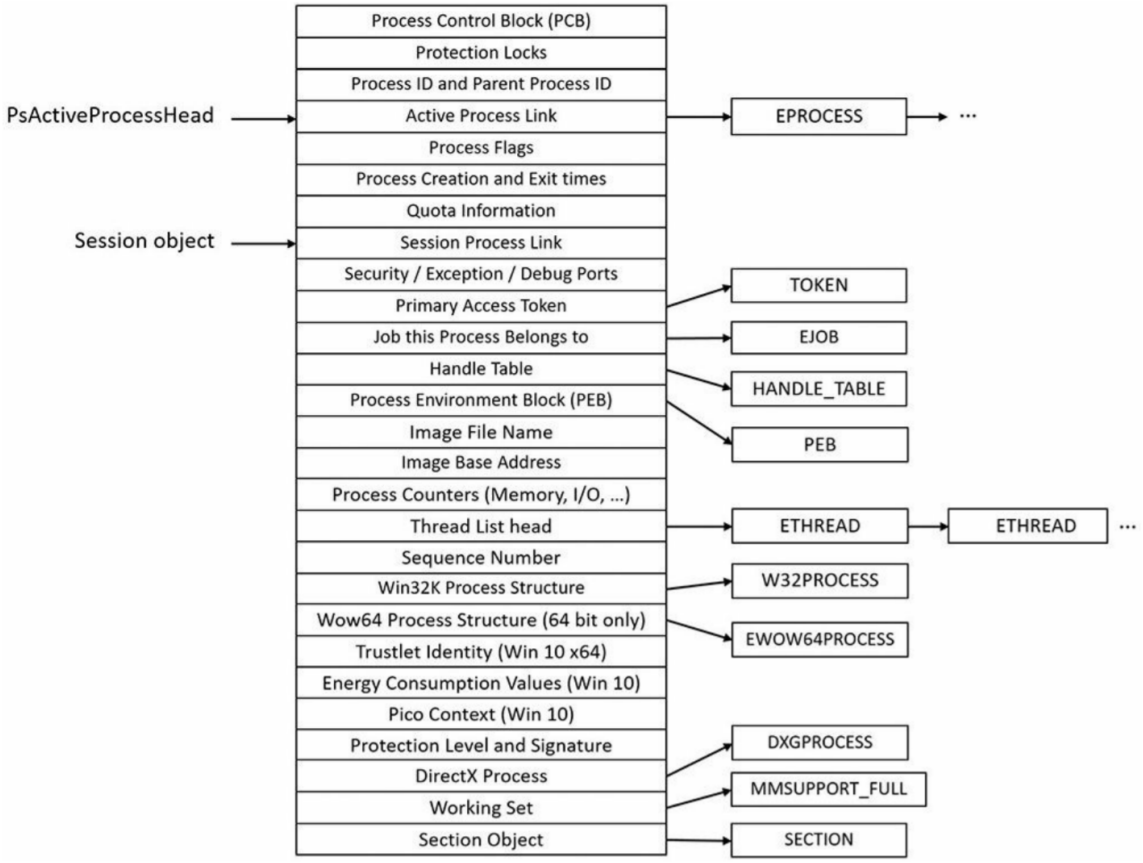

FIGURE 3-2 Important fields of the executive process structure.

Process Control Block (Pcb): first member of executive process structure

KPROCESS (kernel process) type

Dispatcher, scheduler, interrupt/time accounting use KPROCESS

Executive routines use EPROCESS

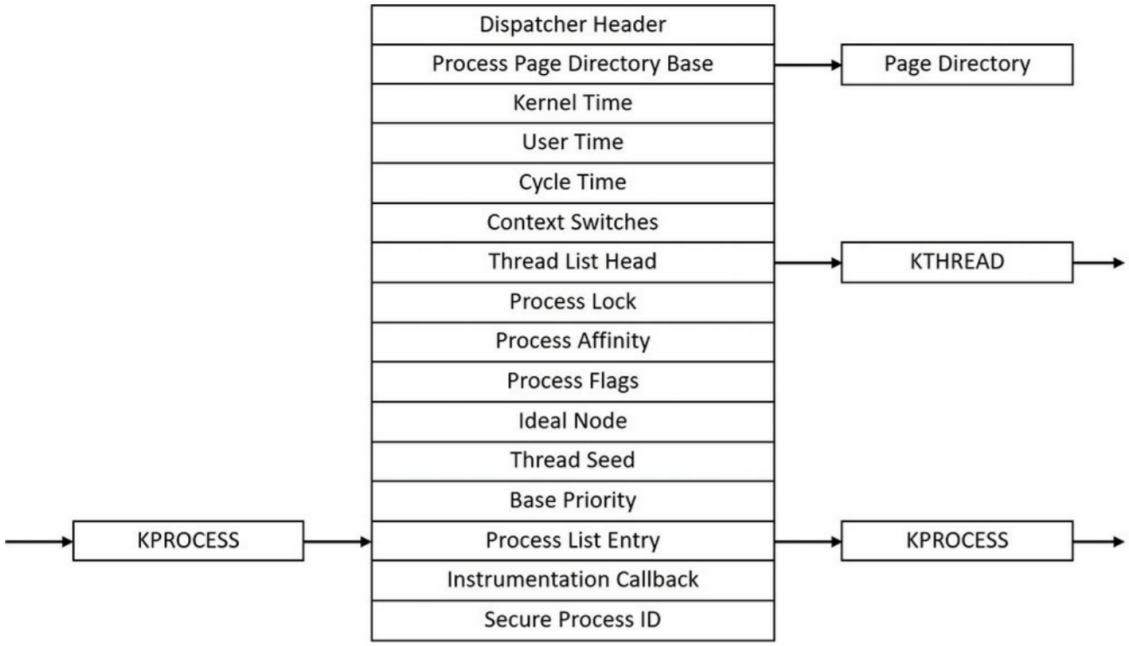

FIGURE 3-3 Important fields of the kernel process structure.

View EPROCESS in kernel debugger with dt nt!_eprocess

Process Environment Block (PEB):

User-mode address space of the process it describes

Needed by image loader, heap manager, Windows components

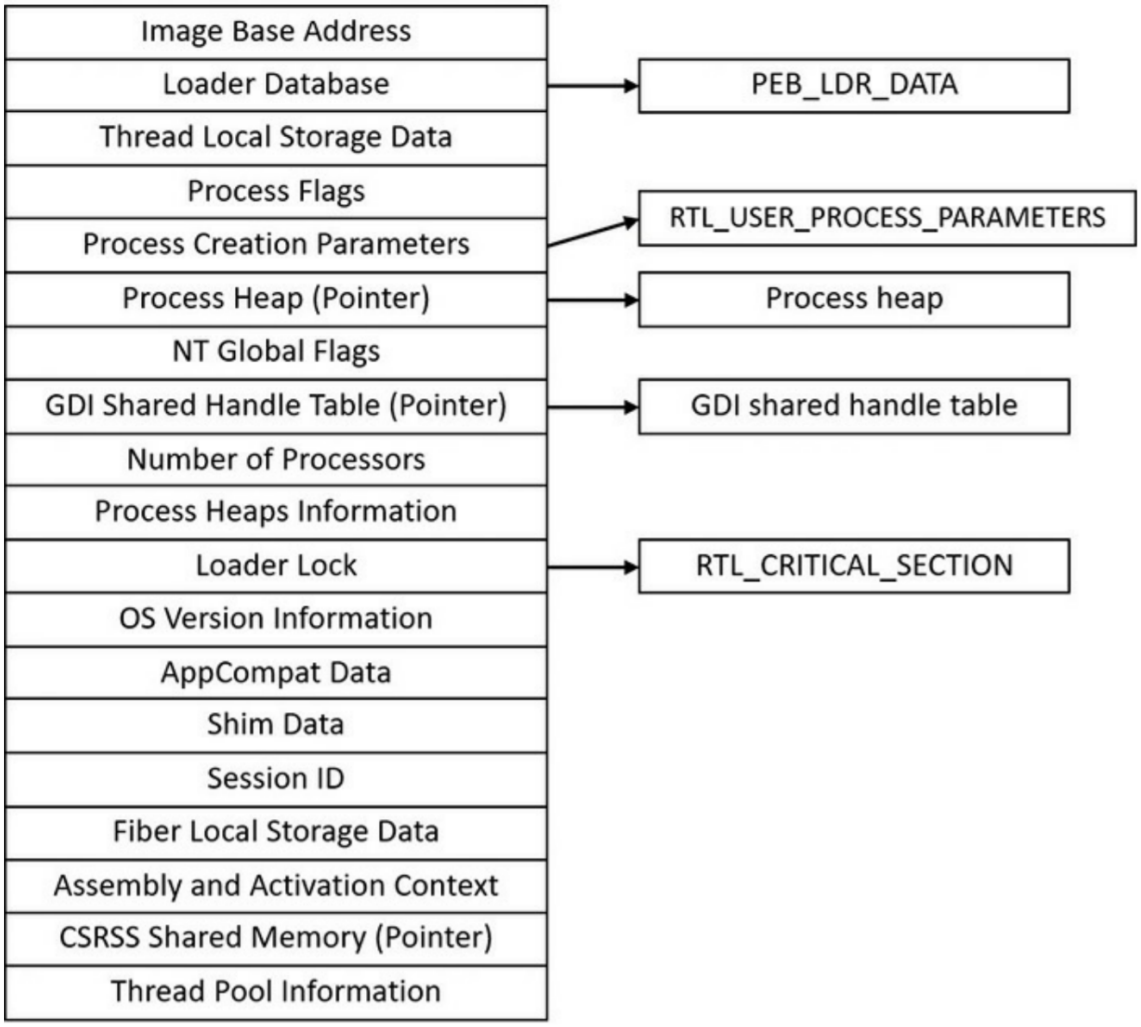

FIGURE 3-4 Important fields of the Process Environment Block.

CSR_PROCESS structure: information for Windows subsystem

Only Windows apps have CSR_PROCESS structure

Maintained by the Csrss process within individual sessions

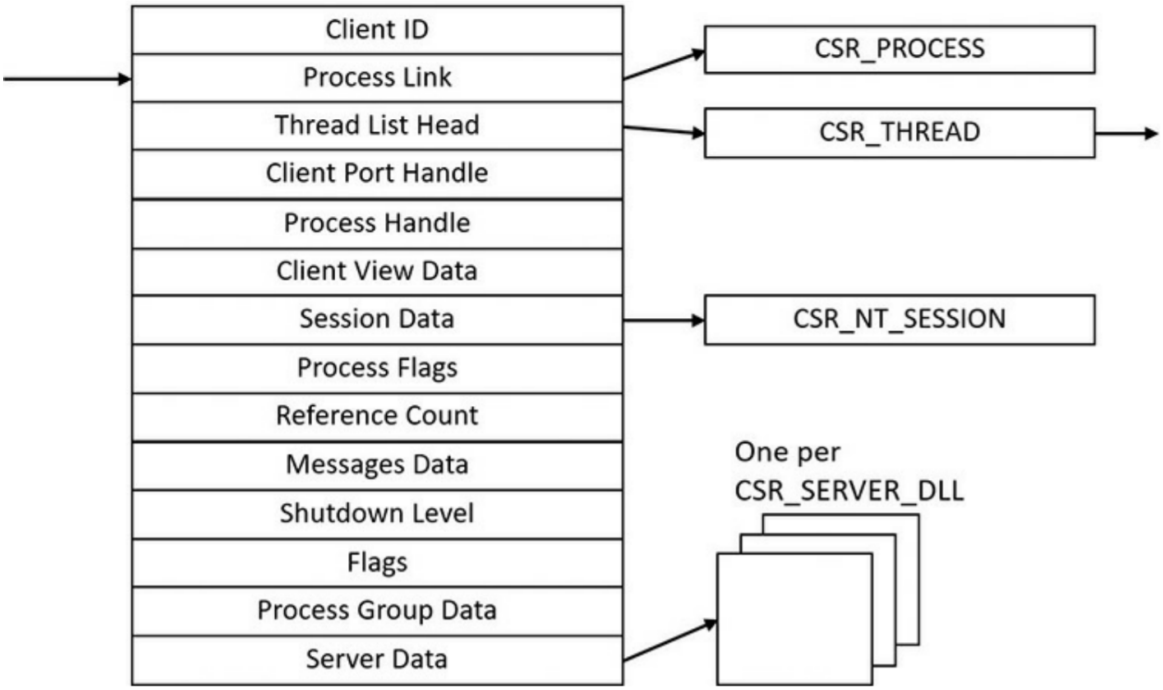

FIGURE 3-5 Fields of the CSR process structure.

Cannot attach user mode debugger to Csrss process because they are protected

W32PROCESS: information for Windows graphics and windows management code in Win32k (kernel)

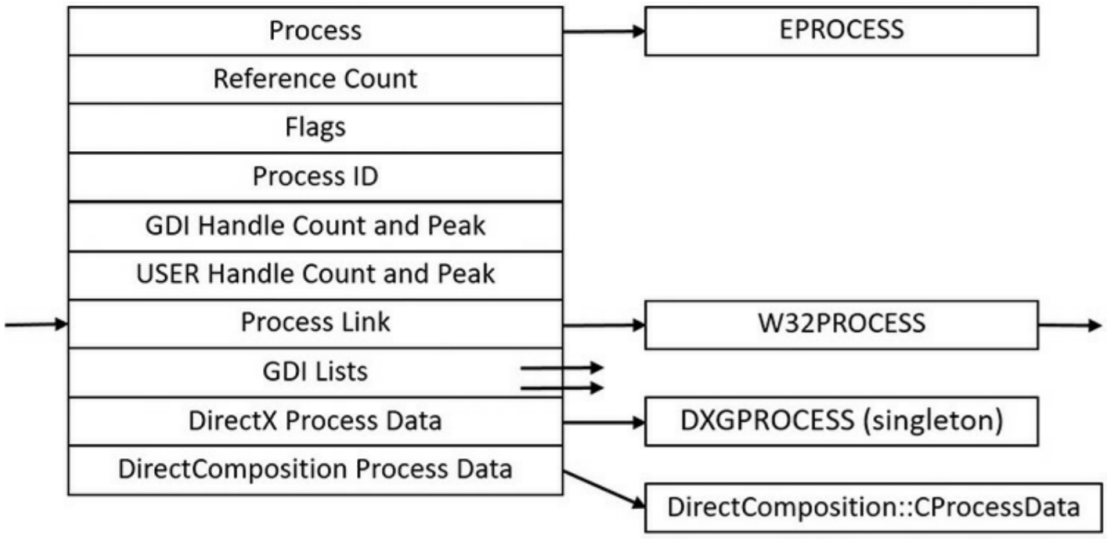

FIGURE 3-6 Fields of the Win32k Process structure.

Protected processes

Debug privilege – any proc with this privilege can request any access right to any other process running on the machine

Process Explorer, Task Manager

Clashes with digital rights management requirements

Protected processes – constrains access right requests (even for admin privilege processes)

OS requires that the image file of the protected process has Windows Media Certificate

Protected Media Path (PMP) provides protection for high-value media

User-mode code (even with elevated privs) cannot inject threads or obtained detailed info about loaded DLLs

Audio Device Graph process (Audiodg.exe) is protected since can decode protected music content

System process is protected b/c:

Decryption info generated by Ksecdd.sys is stored in its user-mode memory

Also needs to protect integrity of kernel handles (System process' handle table contains all kernel handles on the system)

Other drivers map memory inside its user-mode address space

Kernel support for protected processes:

Process creation occurs in kernel mode to avoid injection attacks

Protected processes set bits in EPROCESS to modify security-related routine behaviors

Access rights for protected processes limited to:

PROCESS_QUERY/ SET_LIMITED_INFORMATION

PROCESS_TERMINATE

PROCESS_SUSPEND_RESUME

Admin can load kernel-mode driver that modifies the EPROCESS structure flag to un-protect protected processes

Violates PMP model, driver blocked from loading on 64-bit systems

Will likely be reported by kernel-mode patch protection (PatchGuard) and ProtectedEnvironment and Authentication Driver (Peauth.sys)

Protected Process Light (PPL)

Extends protected process model since Windows 8.1 and Windows Server 2012 R2

Adds attribute values – signers with different trust level, values, access rights

WinSystem is highest-priority signer: used for System process and minimal processes (ex Memory Compression process)

WinTCB (Windows Trusted Computer Base) is highest-priority signer for user-mode processes

Protected processes always trump PPLs

Signer Name (PS_PROTECTED_SIGNER)	Level	Used For
PsProtectedSignerWinSystem	7	System and minimal processes (including Pico processes).
PsProtectedSignerWinTcb	6	Critical Windows components. PROCESS_TERMINATE is denied.
PsProtectedSignerWindows	5	Important Windows components handling sensitive data.
PsProtectedSignerLsa	4	Lsass.exe (if configured to run protected).
PsProtectedSignerAntimalware	3	Anti-malware services and processes, including third party. PROCESS_TERMINATE is denied.
PsProtectedSignerCodeGen	2	NGEN (.NET native code generation).
PsProtectedSignerAuthenticode	1	Hosting DRM content or loading user-mode fonts.
PsProtectedSignerNone	0	Not valid (no protection).

TABLE 3-2 Signers and levels

Windows Media DRM Certificate no longer necessary to run as protected process

MSFT Code Integrity module:

Enhanced Key Usage (EKU) OIDs used as digital code signing certificate

Associates Protected Signer values with EKUs, hardcoded Signer and Issuer strings in the cert

MSFT Windows Issuer can only grant PsProtectedSignerWindows if EKU for Windows System Component Verification is present (1.3.6.1.4.1.311.10.3.6)

Process protection impacts which DLLs allowed to load

If process loads malicious DLL, that DLL runs with the same protection level as the process

Process checks its Signature level (stored in SignatureLevel of EPROCESS) and uses internal lookup table to find corresponding DLL Signature Level (SectionSignatureLevel of EPROCESS)

Ex: process with WinTcp as executable signer can only load Windows or higher signed DLLs

WinTcb-Lite signed processes: smss.exe, csrss.exe, services.exe, wininit.exe

Can configure Lsass.exe to run as PPL on x86/64 via registry or policy setting

Lsass.exe runs PPL on ARM-based Windows

Minimum TCB list: forces processes that are in System path to have minimum protection level and/or signing level regardless of caller's input

Process Name	Minimum Signature Level	Minimum Protection Level
Smss.exe	Inferred from protection level	WinTcb-Lite
Csrss.exe	Inferred from protection level	WinTcb-Lite
Wininit.exe	Inferred from protection level	WinTcb-Lite
Services.exe	Inferred from protection level	WinTcb-Lite
Werfaultsecure.exe	Inferred from protection level	WinTcb-Full
Sppsvc.exe	Inferred from protection level	Windows-Full
Genvalobj.exe	Inferred from protection level	Windows-Full
Lsass.exe	SE_SIGNING_LEVEL_WINDOWS	0
Userinit.exe	SE_SIGNING_LEVEL_WINDOWS	0
Winlogon.exe	SE_SIGNING_LEVEL_WINDOWS	0
Autochk.exe	SE_SIGNING_LEVEL_WINDOWS*	0

*Only on UEFI firmware systems

TABLE 3-3 Minimum TCB

Process Explorer uses user-mode API to query loaded modules – requires access not granted for accessing protected processes

EnumDeviceDrivers API does not require process handle – retrieve list of loaded kernel modules

Uses undocumented API to return all handles on the system (does not require specific process handle)

Identify the process using PID associated with each handle

Third-party PPL support

Anti-malware (AM) main components:

1. Kernel driver that intercepts I/O requests to file system and/or network
   (Implements blocking capabilities using objects, process, thread callbacks)
2. User-mode service that configures driver's policies
   Also receives notifications from drivers regarding interesting events (may communicate with Internet)
3. User-mode GUI that communicates information to user

Cannot inject code or terminate AM service running as PPL

Bypass with kernel-level exploit

Early-Launch Anti Malware (ELAM) driver – requires anti-malware cert provided by MSFT

Custom resource section in PE file called ELAMCERTIFICATEINFO

Contains 3 additional signers, each with up to 3 additional EKUs

Code Integrity recognizes files using any of the Signers containing one of the EKUs it permits

Then allows process to request PPL of PS_PROTECTED_ANTIMALWARE_LIGHT (0x31)

Example: Windows Defender (MsMpEng.exe) and Network Inspection Server (NisSvc.exe) signed with anti-malware cert

Minimal and Pico processes

System process is mostly used as a container to hold system threads and kernel handles (drivers' handles)

Minimal Process

Created when set flag in NtCreateProcessEx and call from kernel-mode → executes PsCreateMinimalProcess API → creates proc without many structures

- No user-mode address space → No PEB
- No NTDLL, no loader/API Set info
- No executable image file associated with execution/name (section object not died to proc)
- Sets Minimal flag in EPROCESS → turns threads into minimal threads → avoids user-mode allocations for threads (TEB, user-mode stack)

Minimal processes on Windows 10

System process

Memory Compression process

Secure system process – if Virtualization-Based Security is enabled

Pico Provider (Lxss.sys and LxCore.sys) – if Windows Subsystem for Linux enabled

Pico Processes

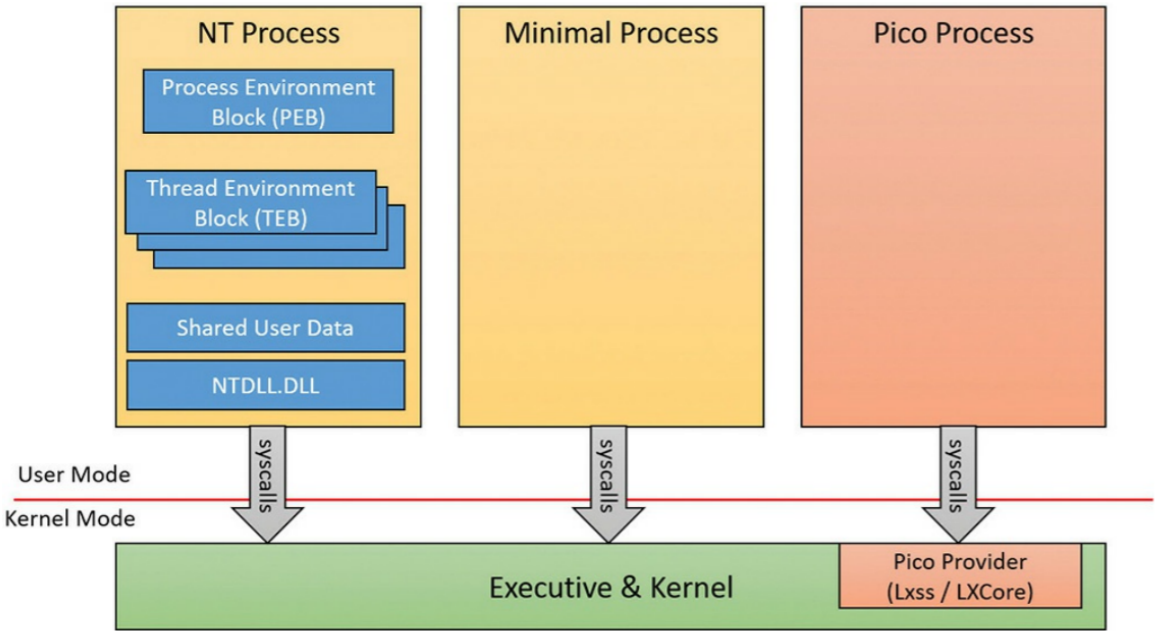

FIGURE 3-8 Process types.

Pico Provider component controls execution of pico processes → allows emulation of diff OS

Implementation of Drawbridge project from Microsoft Research

Essentially custom-written kernel modules that implement callbacks to respond to list of possible events that Pico proc can cause

Register provider with PsRegisterPicoProvider API

Pico provider must be loaded before any other third-party driver

Core driver must be signed with Microsoft Signer Certificate and Windows Component EKU

WSL core driver: Lxss.sys, which serves as stub drive for LxCore.sys:

Loaded later and transfers dispatch tables to itself

Pico registration API returns set of function pointers that can:

- Create Pico process, Create Pico thread
- Get/Set context of Pico process and Pico threads → populates PicoContext field in ETHREADS/EPROCESS
- Get/Set CPU context structure (CONTEXT) of a Pico thread
- Change FS and/or GS segments of Pico thread (used to point to thread local struct i.e. TEB)
- Terminate Pico thread/process
- Suspend/Resume Pico thread

Also returns set of callback function pointers that get call when:

- Pico thread makes a system call using SYSCALL
- Pico thread raises exception
- Fault during probe and lock operation on memory descript list (MDL) occurs inside Pico thread
- Caller requests name of Pico proc
- Event Tracing for Windows (ETW) requests user-mode stack trace of Pico proc
- App attempts to open handle to Pico proc/thread
- Request to terminate Pico proc
- Pico thread/proc terminates unexpectedly

Kernel Patch Protection (KPP) protects callbacks and system calls and prevents fraud/malicious Pico providers registering on top of it

Trustlets

VBS - Device Guard and Credential Guard

Trustlets run in Isolated User Mode which is unprivileged (ring 3) but protected from VTL 0 (NT kernel and regular user mode apps)

Trustlet structure

Have IUM-specific properties:

- Restricted number of system calls (C/C++ Runtime, KernelBase, Advapi, RPC Runtime, CNG Base Crypto, NTDLL)
- Imports Iumbase – provides Base IUM System API: mailslots, storage boxes, crypto, and more
    - Iumbase calls Iumdll.dll (VTL 1 version of Ntdll.dll):
        - contains secure system calls (implemented by Secure Kernel, not passed to VTL 0 kernel)
- .tPolicy section in PE and exported global var s_IumPolicyMetadata
- Signed with cert that contains Isolated User Mode EKU and
- Must be launch with specific process attribute in CreateProcess to request IUM execution

Trustlet policy metadata (s_IumPolicyMetadata)

Controls level of access to Trustlet from VTL 0

Trustlet ID identifies specific Trustlet among the ones known to exist

Contains signed policy options (cannot be modified without invalidating signature)

ETW – enable/disable

Debug – configures debugging – If SecureBoot disabled, Debug can be enabled at all times

Crash Dump – enable/disable

Crash Dump Key – specified public key for encrypting crash dump

Crash Dump GUID – specifies crash dump key – allows multiple keys

Parent Security Descriptor – SSDL format – validates owner/parent process

SVN – Security Version – Used to encrypt AES256/GCM messages

Device ID – secure device PCI identifier – Trustlet can only communicate with the Secure Device whose PCI ID matches this

Capability – enables VTL 1 capabilities – Access to Create Secure Section API, DMA, user-mode MMIO access to Secure Devices, Secure Storage APIs

Scenario ID – GUID specified by Trustlet when creating image sections

Trustlet attributes

Launching trustlet requires PS_CP_SECURE_PROCESS attribute

Authenticates the caller creating the Trustlet

Verifies the Trustlet being executed

Trustlet identifier in attribute must match Trustlet ID in the policy metadata

Mailbox Key – retrieves mailbox data – allows Trustlet to share data with VTL 0 world if Trustlet key is known

Collaboration ID – sets collab ID for Secure Storage IUM API – allows sharing data between trustlets with same collab ID

TK Session ID – session ID during Crypto

System built-in Trustlets

Windows 10 Trustlets

Secure Kernel (0)

Lsalso.exe (1) – Credential and Key Guard

Vmsp.exe(2) – Secure Virtual Machine Worker

Unknown (3) – vTPM Key Enrollment

BioIso.exe (4) – Secure Biometrics

Fslso.exe (5) Secure Frame Server

Trustlet identity

Trustlet identifier (Trustlet ID) – hard-coded into policy metadata

Trustlet instance – 16-byte random number generated by Secure Kernel – guarantees that Secure Storage API only allows one instance of the Trustlet to get/put data

Collaboration ID – Allows other Trustlets with the same ID shared access to Secure Storage blob

Security version (SVN) – For encrypting AES256/GCM data by Credential and Key Guard

Scenario ID – GUID that Trustlet uses to create named secure kernel objects (ex. secure sections)

Validates that Trustlet is creating predetermined object by tagging namespace with GUID

Other Trustlets opening the same named object need to have matching Scenario ID

Isolated user-mode services

Privileged and protected secure system calls only offered by Secure Kernel to Trustlets

- Secure Devices
  - IumCreateSecureDevice, IumDmaMapMemory, IumGetDmaEnabler, IumMapSecureIo, IumProtectSecureIo, IumQuerySecureDeviceInformation, IopUnmapSecureIo, IumUpdateSecureDeviceState
  - Access to secure ACPI/PCI devices (not accessible from VTL 0) owned by Secure Kernel
  - Trustlets with capabilities can map registers of these devices and perform Direct Memory Access (DMA) transfers
  - Can serve as user-mode device drivers for hardware (Secure Device Framework in SDFHost.dll)
    - Secure Biometrics for Windows Hello
    - Secure USB Smartcard (over PCI)
    - Webcam/Fingerprint Sensors (over ACPI)
- Secure Sections
  - IumCreateSecureSection, IumFlushSecureSectionBuffers, IumGetExposed-, SecureSection, IumOpenSecureSection
    - Share physical pages with VTL 0 driver (VslCreateSecureSection from VTL 0) as exposed secure sections
    - Share data solely within VTL 1 a named secure sections
    - Requires Secure Section capability
- Mailboxes
  - IumPostMailbox
  - Share up to eight slots of up to 4KB data with VTL 0 kernel component (VslRetrieveMailbox with slot identifier and secret mailbox key)
- Identity Keys
  - IumGetIdk
  - Obtain unique id decryption key or signing key
  - Unique to machine, only obtainable from a Trustlet
  - Essential part of Credential Guard to uniquely authenticate machine and verify that credentials come from IUM
- Cryptographic Services
  - IumCrypto
  - Encrypt/Decrypt data with local/per-boot session key generated by Secure Kernel
  - Obtain TPM binding handle

        - Trusted Platform Module (TPM) on a chip is an endpoint device that stores RSA encryption keys specific to the hardware for authentication
    - Get FIPS mode of Secure kernel
    - Obtain random number generator (RNG) seed generated by Secure Kernel
    - Generate IDK-signed, SHA-2 hashed, timestamped report of:
        - identity and SVN of Trustlet
        - Policy metadata dump
        - Attach to debugger or not
        - Other Trustlet-controlled data
        - Can be used as TPM-like measurement to prove Trustlet integrity
- Secure Storage
    - IumSecureStorageGet, IumSecureStoragePut
    - Allows Trustlets with Secure Storage capability to store arbitrarily sized storage blobs based on unique Trustlet instance or collaboration ID

Trustlet-accessible system calls

Less than 50 system calls in entire Secure Kernel

- Worker Factory and Thread API – support Thread Pool API (used by RPC) and TLS Slots (used by Loader)
- Process Information API – supports TLS Slots, Thread Stack Allocation
- Event, Semaphore, Wait, Completion APIs – supports Thread Pool and Synchronization
- Advanced Local Procedure Call (ALPC) APIs – supports Local RPC over ncalrpc transport
- System Information API – supports read Secure Boot info, basic and NUMA sys info for Kernel32.dll, Thread Pool scaling, performance, subsets of time info
- Token API – supports RPC impersonation
- Virtual Memory Allocation APIs – supports User-Mode Heap Manager allocations
- Section API – supports Secure Section functionality, Loader for DLL Images
- Trace Control API – supports ETW
- Exception and Continue API – supports Structured Exception Handling (SHE)

No support for Device I/O (file or physical), Registry I/O

No process creation, graphics API usage

Can only communicate through ALPC (exposed secure sections use ALPC)

Flow of CreateProcess

All documented process-creation functions call CreateProcessInternalW

Three stages of creating Windows process:

1. Client-side Kernel32.dll (CreateProcessInternalW)
2. Windows executive
3. Windows subsystem process (Csrss)

Creating an executive subsystem process diff than creating Windows subsystem process

Main stages of Windows CreateProcess* functions

1. Validate parameters, attribute lists, convert flags to native flags
2. Open image file (.exe)
3. Create Windows executive process object
4. Create initial thread (stack, context, Windows executive thread object)
5. Windows subsystem-specific post-creation procedures
6. Start execution of initial thread (unless CREATE_SUSPENDED)
7. Finish initialization of address space and begin execution at entry point

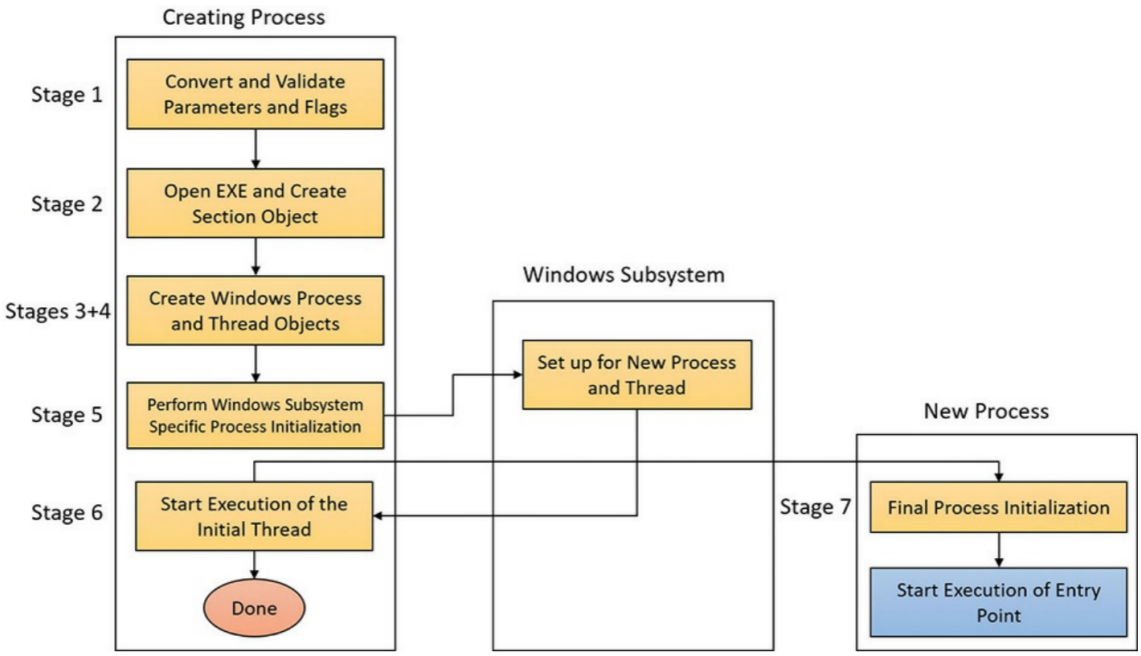

FIGURE 3-10 The main stages of process creation.

Stage 1: Converting and validating parameters and flags

1. Specify priority class via CreationFlags parameter
    a. Idle/Low (4), Below Normal (6), Normal (8), Above Normal (10), High (13), Real-time (24)
    b. Windows resolves by assigning lowest-priority class set
2. Default priority is Normal. If process caller specifies Real-time but does not have SE_INC_BASE_PRIORITY_NAME (Increase Scheduling Priority privilege) then High priority instead
3. If debug, Kernel32 calls DbgUiConnectToDbg in Ntdll.dll and gets handle to debug object from Thread Environment Block (TEB)
4. Default hard error mode – set by Kernel32.dll
5. Convert user-specified attribute list into native format. Some examples:
    a. PS_CP_SAFE_OPEN_PROMPT_ORIGIN_CLAIM – internal attribute used to indicate that file came from untrusted source
    b. PS_CP_JOB_LIST – assigns process to list of jobs
    c. PS_CP_MITIGATION_OPTIONS (PROC_THREAD_MITIGATION_POLICY) – which mitigations (SEHOP, ATL Emulation, NX) should be enabled/disabled for the process

    d. PS_CP_STD_HANDLE_LIST (PROC_THREAD_ATTRIBUTE_HANDLE_LIST) – list of handles belonging to parent process that should be inherited by new process
6. If process part of job object but creation flags request separate virtual DOS machine, flag is ignored
7. Convert Security Attributes to internal representation (OBJECT_ATTRIBUTES structures)
8. CreateProcessInternalW checks if process should be created as modern
9. BasepCreateLowBox records initial token creation if process is to be created as modern
    a. Pass the PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES to create an AppContainer legacy desktop application
10. If creating a modern proc, sets flag to tell kernel to skip embedded manifest detection
11. If debug, marks the Debugger value under Image File Execution Options registry key
12. If no specified desktop in STARTUPINFO, then use caller's current desktop
13. Analyze command line arguments passed to CreateProcessInternalW
14. Convert gathered info into RTL_USER_PROCESS_PARAMETERS

Now, CreateProcessInternalW calls NtCreateUserProcess.

Stage 2: Opening the image to be executed

Creating thread has switched to kernel mode and continues the work within NtCreateUserProcess

1. Validates arguments to make sure call to executive not from spoofed Ntdll.dll transition to kernel mode behavior
2. Find appropriate Windows image and create section object to map image into address space of the new process
    a. DOS .bat or .cmd → CMD.exe, Win16 → NtVdm.exe, Windows →Run directly, DOS .exe, .com, .pif → NtVdm.exe
    b. If failed, returns with Create Status Code or restarts stage 1 if Ntvdm.exe or Cmd.exe
3. Check signing policy if created process is protected
4. Check license if created process is modern
5. If Trustlet, set flag to allow secure kernel to create section object
6. If Windows EXE, try open file and create section object. If not a valid EXE, then call fails. If DLL, then CreateProcessInternalW fails
7. Looks under HKLM\SOFTWARE|Microsoft\Windows NT\Current Version\Image File Execution Options to see if subkey with filename and extension exists. Then calls PsAllocate-Process to get key. If key found, convert image to string and restart CreateProcessInternalW from stage 1
8. If image is not Windows EXE, CreateProcessInternalW looks for Windows support image to run it
    a. Non-Windows apps aren't run directly
    b. If support process already created, use that to run the MS-DOS app
    c. If not already created, change image to be run as Ntvdm.exe and restart from stage 1
    d. CREATE_SEPARATE_WOW_VDM and CREATE_SHARE_WOW_VDM controls whether to create new or shared VDM proc (x86 only)

Stage 3: Create the Windows executive process object

NtCreateUserProcess has successfully created section object to map executable to new process space

Now calls PspAllocateProcess to create Windows executive process object which will run the image

Parent process always required to provide security context for new process (except during system init when the System proc is created)

Stages of creating executive process object:

Stage 3A: Setting up the EPROCESS object

1. Inherit parent process affinity (unless explicitly set in attribute list)
2. Choose NUMA node if any
3. Inherit I/O (default Normal) and page priority (default 5) from parent process
4. Set process exit status to STATUS_PENDING
5. Select hard error processing mode if specified in attribute list if given (else inherit parent's processing mode)
6. Store parent process ID in InheritedFromUniqueProcessId field
7. Query Image File Execution Options (IFEO) to check if process should use large pages (UseLargePages value) and check if NTDLL should be mapped using large pages
8. Query performance options key in IEFO, PerfOptions, if it exists
9. If Wow64 process, allocate Wow64 aux structure (EWOW64PROCESS)
10. Validate LowBox if process to be created inside an AppContainer
11. Acquire privileges required for creating process
12. Create process's primary access token (duplicate of parent's primary token)
    a. Can specify different access token via CreateProcessAsUser
    b. Token might be changed unless parent has SeAssignPrimaryToken privs
13. Check session ID of new process to see if it is a cross-session create, if so, parent temporarily attaches to target session
14. Set child's quota block to parent's quota block and increment ref count for parent quota block (unless process created through CreateProcessAsUser)
15. Default working set limits PspMinimumWorkingSet, PspMaximumWorkingSet unless specified in PerfOptions
16. Initialize address space of the proc. Detach from target session (if applicable)
17. Choose group affinity if not inherited from parent
18. Initialize KPROCESS
19. Set process token
20. Set priority to Normal unless parent Below Normal or Idle
21. Init process handle table. If inherit flag set, then inheritable handles (or a subset of) copied from parent's object handle table into the new process)
22. Apply any performance options specified through PerfOptions
23. Compute and set final process priority and default quantum for threads
24. Set mitigation options provided in IFEO key. Add TreatAsAppContainer if process is under AppContainer
25. Set all other mitigation flags

Stage 3B: Creating the initial process address space

Create the Page directory, Hyperspace page, VAD bitmap page, and Working set list

1. Create page table entries to map initial pages
2. Deduct number of pages from MmTotalCommittedPages and add to MmProcessCommit

3. Deduct system-wide default proc min working set size (PsMinimumWorkingSet) from MmResidentAvailablePages
4. Create page table pages for the global system space

Stage 3C: Creating the kernel process structure

KeInitializeProcess responsible for init KPROCESS structure

1. Initialize doubly linked list (connects all threads that are part of the process)
2. Hard-code initial (reset) value of the process default quantum to 6 until it is init lated by PspComputeQuantumAndPriority
3. Set process base priority
4. Set default proc affinity for threads in the process and the calculated group affinity
5. Set process-swapping state to resident
6. Update thread seed in KeNodeBlock (initial NUMA node block) so that new proc gets random and diff ideal processor seed
7. If process is secure proc, create the secureID with HvlCreateSecureProcess

3D: Concluding the setup of the process address space

MmInitializeProcessAddressSpace

1. Virtual Memory manager sets the process's last trim time to current time
2. Mem Man init process' working set lists
3. Map section to new process' address space
4. Create and init PEB
5. Map Ntdll.dll to proc or 32-bit Ntdll.dll if Wow64 proc
6. Create new session if it was requested
7. Duplicate standard handles and write new values to proc param structure
8. Process memory reservations listed in attribute list
9. Write user process parameters (convert from absolute to relative form)
10. Write affinity info to PEB
    Map MinWin API redirection set
11. Determine process unique ID. Kernel does not distinguish between unique process and thread IDs and handles. Proc and Thread IDs (handles) stored in global handle table (PspCidTable) not associated with any process
12. Init secure process if process is secure and associate it with kernel process object

Stage 3E: Setting up the PEB

NtCreateUserProcess calls MmCreatePeb:

Map National Language Support tables to proc address space

Call MiCreatePebOrTeb: allocates page for PEB and init fields

Fields based on internal vars config through registry (MmHeap, MmCriticalSectionTimeout, MmMinimumStackCommitInBytes)

If IMAGE_FILE_UP_SYSTEM_ONLY flag set in image header, choose single CPU for all threads in this new proc to run on. Each time image is run, use next processor

Stage EF: Completing the setup of the executive process object

PspInsertProcess

1. Write process creation to Security event log if system-wide auditing enabled (local policy or group policy settings from domain controller)
2. If parent in a job, recover job level set of parent and bound it to session of the newly created proc. Then add new proc as a job
3. Add process to Windows list of active processes (PsActiveProcessHead). Makes proc accessible via EnumProcesses and OpenProcess
4. Copy debug port of parent unless NoDebugInherit
5. Make sure group affinity with new proc does not violate group affinity associated with job. Job might have permissions to modify process's affinity permissions since lesser-privileged job might interfere with affinity requirements of a more privileged process
6. Create handle for new process with ObOpenObjectByPointer and return handle to caller
   Process-creation callback sent when first thread within proc is created
   Code always sends process callbacks before object managed-based callbacks

Stage 4: Creating the initial thread stack and its stack and context

Initial thread is created internally by kernel without user-mode input.

PspCreateThread called by NtCreateThread. Uses two helpers:

PspAllocateThread handles actual creation and initialization of executive thread object

PspInsertThread handles creation of thread handle and security attributes then calls KeStartThread to turn executive object into schedulable thread

Thread created in suspended state and only resumed when process completely initialized

Can specify address of PEB in CreateThread parameter but not in CreateProcess

PspAllocateThread

1. Prevent user-mode scheduling (UMS) thread and user-mode callers from creating threads in the system process (nor in Wow64 for UMS)
2. Create and init executive thread object
3. Allocate THREAD_ENERGY_VALUES structure pointed to by ETHREAD
4. Init LPC, I/O Management, Executive
5. Set thread creation time and create thread ID
6. Setup stack and context. Stack sizes taken from the image
7. Allocate Thread Environment Block for the new thread
8. User-mode thread start address stored in ETHREAD StartAddress. Used by RtlUserThreadStart. Windows start address stored in ETHREAD Win32StartAddress
9. KeInitThread sets up KTHREAD struct
    a. Set thread initial current base priority, affinity, and quantum to the process'
    b. Allocate kernel stack for thread and init machine-dependent hardware context for thread (context, trap, exception frames)
    c. Thread starts in kernel mode in KiThreadStartup

    d. Set thread state to Initialized and return to PspAllocateThread
10. If this is UMS thread, call PspUmsInitThread to init UMS state

NtCreateUserProcess now calls **PspInsertThread**

1. If specified, init thread ideal proc, group affinity
2. Check does not violate job limitations
3. Check that proc and thread not terminated and thread has not start running
4. Create secure thread object if thread part of secure process (IUM)
5. KestartThread creates KTHREAD part of thread object and then inserts thread in the process list maintained by KPROCESS
6. Thread frozen if process in deep freeze (no threads allowed to run)
7. Thread inserted into system-wide list of procs maintained by KiProcessListHead if non-x86
8. Increment thread count and inherit owner's I/O priority and page priority.
    a. If this is second thread in the process, primary token is frozen and can no longer be changed
9. Insert into process's thread list
10. Insert thread object into process handle table
11. If this is first thread of the process, call all registered callbacks for process creation. Then call registered thread callbacks
12. Assign process to job list if job list was supplied and this is first thread
13. Set thread in defer ready state with KeReadyThread to ready thread for execution

Stage 5: Performing Windows subsystem-specific initialization

1. Check whether Windows should allow executable to run
    a. Validate image version in the header
    b. Check if Windows application certification has blocked the process
    c. Check if application imports any disallowed APIs (Win Server 2012 R2, Win Storage Server 2012 R2)
2. Create restricted token for new process if software restriction policy said to
3. Call internal functions to get SxS information (manifests, DLL redirection, media) if process not protected
4. Send message to Windows subsystem to be sent to Csrss
    a. Path and SxS name, Process and Thread handles, Section handle, Access token handle, Media info, AppCompat and shim data, Immersive proc info, PEB address, Flags for protected or require UAC, flag for whether proc belongs to Windows app, UI language info, DLL redir and .local flags, Manifest file info

Windows subsystem receives message:

1. CsrCreateProcess duplicates handle for process and thread
2. Allocate CSR_PROCESS structure
3. Set exception port to general function port for Windows subsystem
4. Calls CSR_PROCESS if new proc group is to be created with new proc as root
5. Alloc and init Csrss thread struct (CSR_THREAD)
6. CsrCreateThread insert thread into list of threads for the process

7. Increment process count in current session
8. Set process shutdown level to 0x280 (default)
9. Add new Csrss process struct into Windows subsystem-wide processes

CreateProcessInternalW checks whether proc was run elevated (executed via ShellExecute and elevated by AppInfo service)

If process was a setup program, open process's token and turn on virtualization flag

If app contains elevation shims or requested elevation level in its manifest, destroy the process and send elevation request to AppInfo service

These checks not performed for protected processes

Stage 6: Starting execution of the initial thread

Unless caller specified CREATE_SUSPENDED, resume the initial thread and perform remainder of process-init work in context of the new process

Stage 7: Performing process initialization in the context of the new process

Thread starts running kernel-mode thread startuproutine KiStartUserThread, which lower thread IRQL level from deferred procedure call (DPC) to APC and calls system initial thread routine, PspUserThreadStartup:

1. If x86 arch, install exception chain
2. Lower IRQL to PASSIVE_LEVEL (0, only IRQL user code allowed to run at)
3. Disable ability to swap primary process token at runtime
4. Terminates thread if thread was killed on startup
5. Setup locale ID and ideal proc in TEB and checks if thread creation failed
6. Calls DbgkCreateThread – checks if image notifications sent for new proc. If no notifications and notifications enabled, sends image notification for the process and for image load of Ntdll.dll
7. If is debugee and no debugger notifications yet, then send create process message through debug object
8. If prefetching enabled, call the prefetcher (and Superfetch) to process prefetch instruction file and prefetch pages referenced during first 10 seconds the last time the proc ran
9. Checks if system-wide cookie SharedUserData struct has been setup, if not, generates it
10. If process is IUM, call HvlStartSecureThread to transfer control to secure kernel to start thread execution
11. Setup initial thunk context to run image-loader init routine (LdrInitializeThunk in Ntdll.dll) and system-wide thread startup stub (RtlUserThreadStart in Ntdll.dll)
    a. Edit the context of the thread in place
    b. Issue exit from system service operations → loads crafted user context
    c. LdrInitializeThunk initializes loader, heap man, NLS tables, thread-local storage, fiber-local storage arrays, critical section structures
    d. Load required DLLs and call DLL entry point with DLL_PROCESS_ATTACH

When function returns, NtContinue restores new user context and returns to user mode

RtlUserThreadStart calls the application's entry point. Kernel already pushed parameters: actual image entry point address, start parameter)

- Allows image loader in Ntdll.dll to setup proc internally and behind scenes
- Having threads begin in common routine allows them to be wrapped in exception handling so that Ntdll.dll is aware if they crash and can call unhandled exception filter inside Kernel32.dll
- Coordinate thread exit and return from thread start routine

Terminating a process

Process needs PROCESS_VM_WRITE to use WriteProcessMemory

Process calls ExitProcess function to gracefully exit

DLLs loaded into proc can get notified and call DllMain with DLL_PROCESS_DETACH

Process startup code for first thread calls ExitProcess on behalf of thread returning from main function

TerminateProcess called from outside is ungraceful and requires that process is open with PROCESS_TERMINATE access

Process memory auto freed by kernel, destroy address space, close all kernel object handles.

EPROCESS destroyed when all handles closed

It is the responsibility of third party drivers who make kernel memory allocation is responsible for freeing this memory. IRP_MJ_CLOSE or IRP_MJ_CLEANUP tells driver that handle to device object clsed through process termination

Image Loader