KAPITEL 3

Prozesse und Jobs

In diesem Kapitel erklären wir die Datenstrukturen und Algorithmen für die Prozesse und Jobs in Windows. Als Erstes sehen wir uns dabei an, wie Prozesse erstellt werden. Anschließend untersuchen wir die internen Strukturen, die einen Prozess ausmachen. Danach werfen wir einen Blick auf geschützte Prozesse und ihre Unterschiede zu nicht geschützten. Wir schauen uns auch die erforderlichen Schritte an, um einen Prozess (und seinen Anfangsthread) zu erstellen. Zum Abschluss des Kapitels befassen wir uns mit Jobs.

Da Prozesse so viele Windows-Komponenten berühren, werden in diesem Kapitel eine Reihe von Begriffen und Datenstrukturen erwähnt (wie Arbeitssätze, Threads, Objekte, Handles, Heaps usw.), die an anderen Stellen dieses Buches ausführlich erläutert werden. Um dieses Kapitel richtig verstehen zu können, müssen Sie mit den in Kapitel 1 und 2 vorgestellten Begriffen und Grundprinzipien vertraut sein, also z. B. die Unterschiede zwischen einem Prozess und einem Thread und zwischen Benutzer- und Kernelmodus sowie den Aufbau des virtuellen Adressraums von Windows kennen.

Prozesse erstellen

Die Windows-API enthält mehrere Funktionen, um Prozesse zu erstellen. Die einfachste ist `CreateProcess`, die versucht, einen Prozess anzulegen, der dasselbe Zugriffstoken wie der Erstellerprozess hat. Wird ein anderes Token benötigt, bietet sich `CreateProcessAsUser` an, der als (erstes) zusätzliches Argument ein Handle für ein auf irgendeine Weise (z. B. durch Aufruf der Funktion `LogonUser`) erworbenes Tokenobjekt entgegennimmt.

Weitere Funktionen zum Erstellen von Prozessen sind `CreateProcessWithTokenW` und `CreateProcessWithLogonW` (die beide zu *Advapi32.dll* gehören). `CreateProcessWithTokenW` ähnelt `CreateProcessAsUser`, allerdings muss der Aufrufer über andere Rechte verfügen (siehe die Dokumentation des Windows SDK). `CreateProcessWithLogonW` ist eine praktische Möglichkeit, um sich mit den Anmeldeinformationen eines gegebenen Benutzers anzumelden und gleichzeitig einen Prozess mit dem abgerufenen Token zu erstellen. Um den Prozess tatsächlich zu erstellen, rufen beide Funktionen den Dienst *Secondary Logon* (*seclogon.dll* in einem *SvcHost.exe*-Prozess) über einen Remoteprozeduraufruf auf. `SecLogon` führt diesen Aufruf in seiner internen Funktion `SlrCreateProcessWithLogon` aus und ruft, wenn alles gut geht, schließlich `CreateProcessAsUser` auf. Der Dienst `SecLogon` ist standardmäßig für den manuellen Start eingerichtet, weshalb er erst beim ersten Aufruf von `CreateProcessWithTokenW` oder `CreateProcessWithLogonW` gestartet wird. Schlägt der Dienststart fehl (etwa weil ein Administrator den Dienst deaktiviert hat), so schlagen auch diese Funktionen fehl. Das Befehlszeilenprogramm `runas`, das Sie wahrscheinlich kennen, nutzt diese Funktionen.

Abb. 3–1 stellt die beschriebenen Aufrufe grafisch dar.

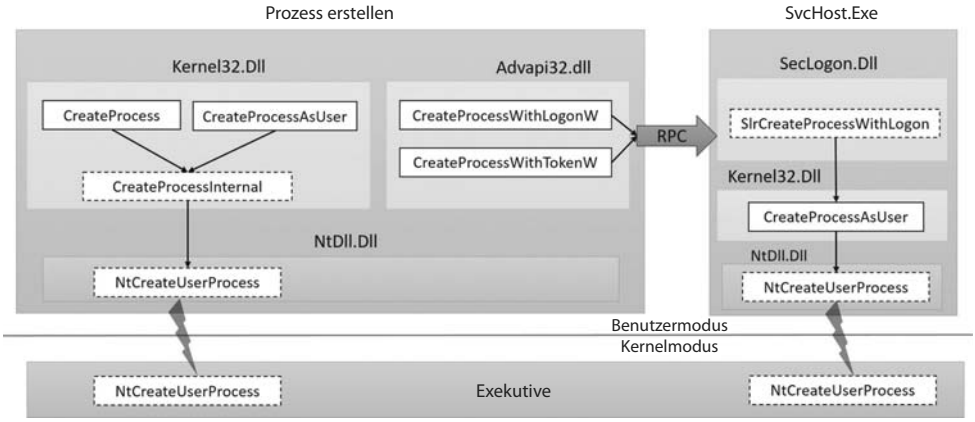


Abbildung 3–1 Funktionen zum Erstellen eines Prozesses. Interne Funktionen sind durch gestrichelte Kästen gekennzeichnet.

Alle zuvor erwähnten, dokumentierten Funktionen erwarten eine ordnungsgemäße PE-Datei (Portable Executive; wobei die Endung *.exe* jedoch nicht unbedingt erforderlich ist), eine Batchdatei oder eine 16-Bit-COM-Anwendung. Darüber hinaus jedoch wissen sie nicht, welche Zusammenhänge zwischen einzelnen Endungen (wie *.txt*) und ausführbaren Dateien (wie dem Windows-Editor) bestehen. Solche Kenntnisse werden durch die Windows-Shell in Funktionen wie `ShellExecute` und `ShellExecuteEx` bereitgestellt. Sie können beliebige Dateien entgegennehmen (nicht nur ausführbare) und versuchen dann, auf der Grundlage der Dateiendung und der Registrierungseinstellungen in `HKEY_CLASSES_ROOT` die zugehörige ausführbare Datei zu finden (siehe Kapitel 9 in Band 2). Schließlich ruft `ShellExecute(Ex)` die Funktion `CreateProcess` mit einer geeigneten ausführbaren Datei auf und hängt die entsprechenden Argumente an die Befehlszeile an, um das zu erreichen, was der Benutzer tun will (z. B. hängt sie den Dateinamen *Notepad.exe* an, wenn der Benutzer eine *.txt*-Datei bearbeiten möchte).

Letzten Endes führen all diese Ausführungspfade zu der gemeinsamen internen Funktion `CreateProcessInternal`, die mit der eigentlichen Arbeit beginnt, um einen Windows-Prozess im Benutzermodus zu erstellen. Wenn alles gut geht, ruft `CreateProcessInternal` schließlich `NtCreateUserProcess` in *Ntdll.dll* auf, um in den Kernelmodus überzugehen und mit den Aspekten der Prozesserstellung weiterzumachen, die im Kernelmodus ablaufen. Dies geschieht in einer Funktion der Exekutive mit demselben Namen (`NtCreateUserProcess`).

Argumente der CreateProcess*-Funktionen

Es lohnt sich, einen Blick auf die Argumente der `CreateProcess*`-Funktionen zu werfen. Einigen davon werden Sie in dem Abschnitt über den Ablauf von `CreateProcess` wiederbegegnen. Ein Prozess, der im Benutzermodus erstellt wird, enthält immer einen Thread, der schließlich die Hauptfunktion der ausführbaren Datei ausführen wird. Die wichtigsten Argumente für `CreateProcess*`-Funktionen lauten:

- Bei `CreateProcessAsUser` und `CreateProcessWithTokenW` das Tokenhandle, unter dem der neue Prozess ausgeführt werden soll. Bei `CreateProcessWithLogonW` sind der Benutzername, die Domäne und das Kennwort erforderlich.
- Der Pfad der ausführbaren Datei und die Befehlszeilenargumente.
- Optionale Sicherheitsattribute für den neuen Prozess und das Threadobjekt.
- Ein boolesches Flag, das angibt, ob alle als vererbbar gekennzeichneten Handles im aktuellen Prozess (dem Erstellerprozess) vom neuen Prozess geerbt (in ihn kopiert) werden sollen. (Mehr über Handles und die Handlevererbung erfahren Sie in Kapitel 8 von Band 2.)
- Mehrere Flags, die den Vorgang zum Erstellen des Prozesses steuern. Die Dokumentation des Windows SDK enthält die vollständige Liste. Hier geben wir nur einige Beispiele:
  - `CREATE_SUSPENDED` Erstellt den Anfangsthread des neuen Prozesses im angehaltenen Zustand. Der Thread wird ausgeführt, wenn Sie später `ResumeThread` aufrufen.
  - `DEBUG_PROCESS` Der Erstellerprozess deklariert sich selbst als Debugger und erstellt den neuen Prozess unter seiner Kontrolle.
  - `EXTENDED_STARTUPINFO_PRESENT` Anstelle von `STARTUPINFO` wird die erweiterte Struktur `STARTUPINFOEX` bereitgestellt (siehe weiter unten).
- Optional ein Umgebungsblock für den neuen Prozess, der die Umgebungsvariablen angibt. Wird er nicht angegeben, so wird er vom Erstellerprozess geerbt.
- Optional das aktuelle Verzeichnis für den neuen Prozess. Wird es nicht angegeben, so wird es vom Erstellerprozess geerbt. Der neue Prozess kann später `SetCurrentDirectory` aufrufen, um ein anderes Verzeichnis festzulegen. Das aktuelle Verzeichnis eines Prozesses wird in verschiedenen Suchvorgängen ohne Angabe eines vollständigen Pfades genutzt (z. B. beim Laden einer DLL nur mit dem Dateinamen).
- Eine `STARTUPINFO`- oder `STARTUPINFOEX`-Struktur, die noch mehr Konfigurationseinstellungen für den Erstellungsvorgang eines Prozesses bietet. `STARTUPINFOEX` enthält ein zusätzliches, undurchsichtiges Feld für einen Satz von Prozess- und Threadattributen, bei denen es sich im Grunde genommen um ein Array aus Schlüssel-Wert-Paaren handelt. Die Attribute werden durch den Aufruf von `UpdateProcThreadAttributes` für jedes benötigte Attribut festgelegt. Einige von ihnen sind nicht dokumentiert und werden intern verwendet, z. B. beim Erstellen von Store-Apps (siehe den nächsten Abschnitt).
- Eine `PROCESS_INFORMATION`-Struktur für die Ausgabe beim erfolgreichen Erstellen des Prozesses. Sie enthält die IDs des neuen Prozesses und des neuen Threads sowie die Handles dafür. Diese Handles kann der Erstellerprozess nutzen, wenn er den neuen Prozess oder Thread anschließend auf irgendeine Weise bearbeiten möchte.

Moderne Windows-Prozesse erstellen

In Kapitel 1 wurde die neue Art von Anwendungen beschrieben, die ab Windows 8 und Windows Server 2012 zur Verfügung stehen. Die Namen solcher Anwendungen wurden im Laufe der Zeit geändert. Wir nennen sie hier moderne Anwendungen, UWP-Anwendungen oder immersive Prozesse, um sie von den klassischen Desktopanwendungen zu unterscheiden.

Um einen Prozess für eine moderne Anwendung zu erstellen, ist mehr erforderlich, als einfach `CreateProcess` mit dem Pfad der ausführbaren Datei aufzurufen. Es sind auch einige Befehlszeilenargumente und (mithilfe von `UpdateProcThreadAttribute`) ein nicht dokumentiertes Prozessattribut mit dem Schlüssel `PROC_THREAD_ATTRIBUTE_PACKAGE_FULL_NAME` anzugeben. Dieses Attribut ist nicht dokumentiert, aber die API bietet noch andere Möglichkeiten, um eine Store-App auszuführen. Beispielsweise enthält die Windows-API die COM-Schnittstelle `IApplicationActivationManager`, die von einer COM-Klasse mit der Klassen-ID `CLSID_ApplicationActivationManager` implementiert wird. Eine der Methoden in dieser Schnittstelle ist `ActivateApplication`, und damit können Sie eine Store-App starten, nachdem Sie mit einem Aufruf von `GetPackageApplicationsIDs` die sogenannte `AppUserModelId` aus dem vollständigen Paketnamen der App abgerufen haben. (Mehr über diese APIs erfahren Sie im Windows SDK.)

Mehr über Paketnamen und darüber, wie eine Store-App erstellt wird, wenn ein Benutzer auf die Kachel einer modernen App tippt (wobei letzten Endes ebenfalls `CreateProcess` aufgerufen wird), erfahren Sie in Kapitel 9 von Band 2.

Andere Arten von Prozessen erstellen

Windows-Anwendungen starten entweder klassische oder moderne Anwendungen, doch die Exekutive bietet auch Unterstützung für andere Arten von Prozessen, die durch Umgehen der Windows-API gestartet werden müssen. Dazu gehören native, minimale und Pico-Prozesse. Beispielsweise ist der in Kapitel 2 vorgestellte Sitzungs-Manager (`Smss`) ein natives Abbild. Da er direkt vom Kernel erstellt wird, nutzt er offensichtlich nicht die API `CreateProcess`, sondern ruft direkt `NtCreateUserProcess` auf. Wenn `Smss` die Prozesse `Autochk` (das Programm zur Überprüfung der Festplatte) oder `Csrss` (den Windows-Teilsystemprozess) erstellt, steht die Windows-API gar nicht zur Verfügung, weshalb ebenfalls `NtCreateUserProcess` verwendet werden muss. Des Weiteren können Windows-Anwendungen keine nativen Prozesse erstellen, da die Funktion `CreateProcessInternal` Abbilder mit dem Typ des nativen Teilsystems ablehnt. Um all diese Schwierigkeiten abzumildern, enthält die native Bibliothek *Ntdll.dll* die exportierte Hilfsfunktion `RtlCreateUserProcess`, die einen einfachen Wrapper um `NtCreateUserProcess` darstellt.

Wie der Name schon sagt, dient `NtCreateUserProcess` dazu, Benutzermodusprozesse zu erstellen. Wie Sie in Kapitel 2 gesehen haben, enthält Windows jedoch auch eine Reihe von Kernelmodusprozessen wie den System- und den Speicherkomprimierungsprozess (bei denen es sich um minimale Prozesse handelt). Es können auch Pico-Prozesse vorhanden sein, die von einem Anbieter wie dem Windows-Teilsystem für Linux verwaltet werden. Erstellt werden solche Prozesse durch einen Systemaufruf von `NtCreateProcessEx`, wobei einige Möglichkeiten (etwa das Erstellen von minimalen Prozessen) ausschließlich für Aufrufer aus dem Kernelmodus reserviert sind.

Pico-Anbieter rufen die Hilfsfunktion `PspCreatePicoProcess` auf, die sich sowohl darum kümmert, den minimalen Prozess zu erstellen, als auch dessen Pico-Anbieterkontext zu initialisieren. Diese Funktion wird nicht exportiert und ist nur über eine besondere Schnittstelle ausschließlich für Pico-Anbieter verfügbar.

Wie Sie in dem Abschnitt über den Ablauf weiter hinten in diesem Kapitel noch sehen werden, handelt es sich bei `NtCreateProcessEx` und `NtCreateUserProcess` um verschiedene Systemaufrufe, die aber auf dieselben internen Routinen zurückgreifen, um ihre Arbeit zu erledigen,

nämlich PspAllocateProcess und PspInsertProcess. Alle Möglichkeiten, um einen Prozess zu erstellen, die Sie bisher betrachtet haben, und auch alle weiteren Möglichkeiten, die Sie sich vorstellen können – von einem WMI-PowerShell-Cmdlet bis zu einem Kerneltreiber – landen schließlich dort.

Interne Mechanismen von Prozessen

Dieser Abschnitt beschreibt die wichtigsten Datenstrukturen von Windows-Prozessen, die von verschiedenen Teilen des Systems unterhalten werden, und die Möglichkeiten und Werkzeuge zu ihrer Untersuchung.

Jeder Windows-Prozess wird durch eine Exekutivprozessstruktur (EPROCESS) dargestellt. Neben vielen Prozessattributen enthält eine EPROCESS-Struktur auch weitere zugehörige Datenstrukturen bzw. verweist darauf. Beispielsweise hat jeder Prozess einen oder mehrere Threads, die jeweils durch eine Exekutivthreadstruktur (ETHREAD) dargestellt werden. (Threaddatenstrukturen werden in Kapitel 4 erklärt.)

Die EPROCESS-Struktur und die meisten ihrer zugehörigen Datenstrukturen befinden sich im Systemadressraum. Eine Ausnahme bildet der Prozessumgebungsblock (Process Environment Block, PEB), der sich im Prozess- (Benutzer-) Adressraum befindet (da er Informationen enthält, auf die Benutzermoduscode zugreifen muss). Außerdem sind einige der Prozessdatenstrukturen für die Speicherverwaltung, z. B. die Liste der Arbeitssätze, nur im Kontext des aktuellen Prozesses gültig, da sie im prozessspezifischen Systemadressraum gespeichert sind. (Mehr über den Prozessadressraum erfahren Sie in Kapitel 5.)

Für jeden Prozess, der ein Windows-Programm ausführt, unterhält der Prozess des Windows-Teilsystems (*Csrss*) eine parallele Struktur namens CSR_PROCESS und der Kernelmodusteil des Windows-Teilsystems (*Win32k.sys*) die Datenstruktur W32PROCESS, die erstellt wird, wenn ein Thread zum ersten Mal die im Kernelmodus implementierte Windows-Funktion USER oder GDI aufruft. Das geschieht, sobald die Bibliothek *User32.dll* geladen ist. Typische Funktionen, die das Laden dieser Bibliothek veranlassen, sind CreateWindow(Ex) und GetMessage.

Da der Kernelmodusteil des Windows-Teilsystems starken Gebrauch von Grafiken mit DirectX-Hardwarebeschleunigung macht, veranlasst die GDI-Infrastruktur (Graphics Device Interface) den DirectX-Grafikkernel (*Dxgkrnl.sys*), eine eigene Struktur zu initialisieren, nämlich DXGPROCESS. Sie enthält Informationen für DirectX-Objekte (Oberflächen, Schattierer usw.) und die GPGPU-Zähler und Richtlinieneinstellungen für die Zeitplanung sowohl für Rechenvorgänge als auch für die Speicherverwaltung.

Außerdem wird beim Leerlaufprozess jede EPROCESS-Struktur vom Objekt-Manager der Exekutive als ein Prozessobjekt gekapselt (siehe Kapitel 8 von Band 2). Da Prozesse keine benannten Objekte sind, werden sie im Sysinternals-Programm WinObj nicht angezeigt. Sie können sich in WinObj jedoch das Type-Objekt Process im Verzeichnis *\ObjectTypes* ansehen. Ein Handle für einen Prozess ermöglicht durch die Nutzung der APIs für diesen Prozess den Zugriff auf Daten in der EPROCESS-Struktur und einiger ihrer zugehörigen Strukturen.

Durch die Registrierung von Prozesserstellungsbenachrichtigungen können viele andere Treiber und Systemkomponenten ihre eigenen Datenstrukturen erstellen, um die Informationen, die sie prozessweise speichern, zu verfolgen. (Das ist durch die Exekutivfunktionen

PsSetCreateProcessNotifyRoutine(Ex, Ex2) möglich, die im WDK dokumentiert sind.) Um den Prozessoverhead zu bestimmen, muss die Größe dieser Datenstrukturen oft berücksichtigt werden, wobei es jedoch praktisch unmöglich ist, genaue Zahlen zu ermitteln. Außerdem erlauben einige dieser Funktionen solchen Komponenten, das Erstellen von Prozessen zu verbieten oder zu blockieren. Dadurch erhalten Antimalware-Anbieter in der Architektur eine Möglichkeit, Sicherheitsverbesserungen am Betriebssystem vorzunehmen, z. B. durch hashgestützte Blacklists oder andere Techniken.

Sehen wir uns als Erstes das Prozessobjekt an. Abb. 3–2 zeigt die wichtigsten Felder in einer EProcess-Struktur.

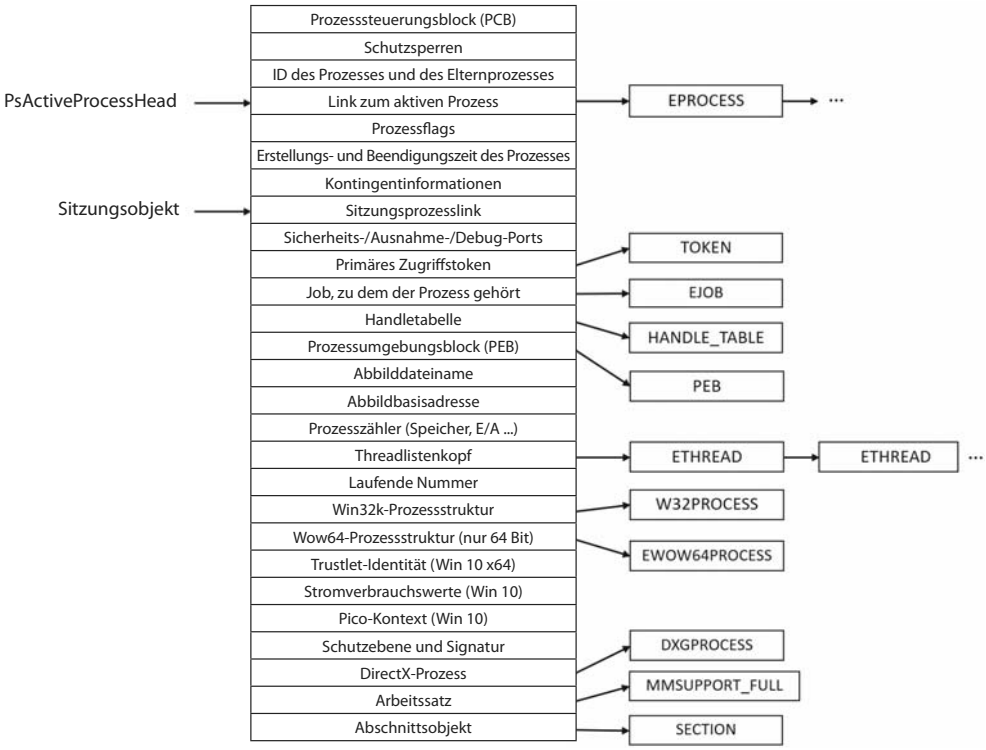


Abbildung 3–2 Wichtige Felder in der EPROCESS-Struktur

Die Datenstrukturen für einen Prozess folgen einem ähnlichen Aufbau wie die APIs und Komponenten des Kernels, die auf isolierte und geschichtete Module mit ihren eigenen Namenskonventionen verteilt sind. Wie Abb. 3–2 zeigt, ist das erste Element der EPROCESS-Struktur der *Prozesssteuerungsblock* (*Process Control Block*, *PCB*). Dabei handelt es sich um eine Struktur vom Typ KPROCESS (Kernelprozess). Routinen in der Exekutive speichern Informationen in der EPROCESS-Struktur, doch der Dispatcher, der Scheduler und der Interrupt- und Zeiterfassungscode, die zum Betriebssystemkernel gehören, verwenden stattdessen die KPROCESS-Struktur. Dies ermöglicht eine Abstraktionsebene zwischen der höheren Funktionalität der Exekutive und den zugrunde liegenden Implementierungen einzelner Funktionen und hilft, unerwünschte Abhängigkeiten zwischen den Ebenen zu vermeiden. Abb. 3–3 zeigt die wichtigsten Felder einer KPROCESS-Struktur.

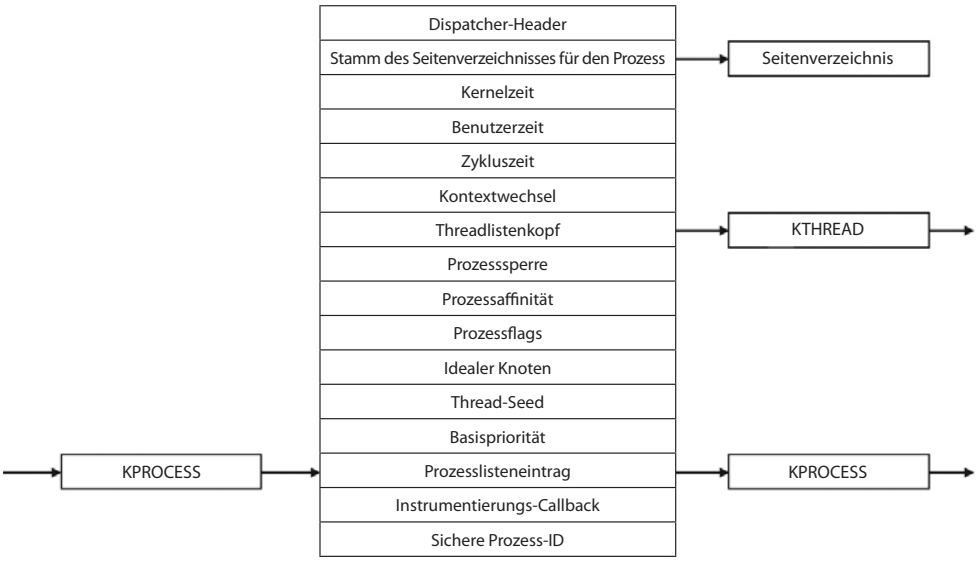


Abbildung 3–3 Wichtige Felder der Kernelprozessstruktur

Experiment: Das Format einer EPROCESS-Struktur anzeigen

Um sich eine Liste der Felder einer `EPROCESS` und ihrer Offsets im Hexadezimalformat anzusehen, geben Sie im Kerneldebugger **dt nt!_eprocess** ein. (Mehr über den Kerneldebugger und seine Verwendung für das Kerneldebugging auf dem lokalen System erfahren Sie in Kapitel 1.) Das folgende Listing zeigt eine Ausgabe auf einem 64-Bit-System mit Windows 10, die allerdings aus Platzgründen gekürzt wurde:

```
lkd> dt nt!_eprocess
   +0x000 Pcb               : _KPROCESS
   +0x2d8 ProcessLock       : _EX_PUSH_LOCK
   +0x2e0 RundownProtect    : _EX_RUNDOWN_REF
   +0x2e8 UniqueProcessId   : Ptr64 Void
   +0x2f0 ActiveProcessLinks : _LIST_ENTRY
...
   +0x3a8 Win32Process      : Ptr64 Void
   +0x3b0 Job               : Ptr64 _EJOB
...
   +0x418 ObjectTable       : Ptr64 _HANDLE_TABLE
   +0x420 DebugPort         : Ptr64 Void
   +0x428 WoW64Process      : Ptr64 _EWOW64PROCESS
...
   +0x758 SharedCommitCharge : Uint8B
   +0x760 SharedCommitLock   : _EX_PUSH_LOCK
```

→

```
   +0x768 SharedCommitLinks  : _LIST_ENTRY
   +0x778 AllowedCpuSets     : Uint8B
   +0x780 DefaultCpuSets     : Uint8B
   +0x778 AllowedCpuSetsIndirect : Ptr64 Uint8B
   +0x780 DefaultCpuSetsIndirect : Ptr64 Uint8B
```

Das erste Element dieser Struktur (`Pcb`) ist eine eingebettete `KPROCESS`-Struktur. Dort sind die Scheduling- und Zeiterfassungsdaten gespeichert. Das Format der `KPROCESS`-Struktur können Sie sich auf die gleiche Weise anzeigen lassen wie das der `EPROCESS`-Struktur:

```
lkd> dt nt!_kprocess
   +0x000 Header                   : _DISPATCHER_HEADER
   +0x018 ProfileListHead          : _LIST_ENTRY
   +0x028 DirectoryTableBase       : Uint8B
   +0x030 ThreadListHead           : _LIST_ENTRY
   +0x040 ProcessLock              : Uint4B
...
   +0x26c KernelTime               : Uint4B
   +0x270 UserTime                 : Uint4B
   +0x274 LdtFreeSelectorHint      : Uint2B
   +0x276 LdtTableLength           : Uint2B
   +0x278 LdtSystemDescriptor      : _KGDTENTRY64
   +0x288 LdtBaseAddress           : Ptr64 Void
   +0x290 LdtProcessLock           : _FAST_MUTEX
   +0x2c8 InstrumentationCallback : Ptr64 Void
   +0x2d0 SecurePid                : Uint8B
```

Mit dem Befehl `dt` können Sie sich auch den Inhalt eines oder mehrere Felder anzeigen lassen, indem Sie hinter dem Strukturnamen den Feldnamen angeben. Beispielsweise zeigt `dt nt!_eprocess UniqueProcessId` das Feld mit der eindeutigen Prozess-ID an. Wenn ein Feld wiederum für eine Struktur steht – wie das Feld `Pcb` in der `EPROCESS`-Struktur, das selbst eine `KPROCESS`-Unterstruktur enthält –, können Sie hinter dem Feldnamen einen Punkt angeben, um den Debugger anzuweisen, die Unterstruktur anzuzeigen. Wenn Sie also beispielsweise die `KPROCESS`-Struktur ansehen wollen, geben Sie `dt nt!_eprocess Pcb.` ein. Dahinter können Sie wiederum die Namen von Feldern (innerhalb der `KPROCESS`-Struktur) angeben usw. Mit dem Schalter `-r` des Befehls `dt` können Sie sämtliche Unterstrukturen durchlaufen. Wenn Sie hinter dem Schalter eine Zahl angeben, legen Sie damit die Verschachtelungstiefe vor, bis zu der sich der Befehl vorarbeitet.

In der zuvor gezeigten Verwendung zeigt der Befehl `dt` jedoch das Format der ausgewählten Struktur und nicht die Inhalte irgendwelcher Instanzen dieses Strukturtyps an. Um die Instanz eines Prozesses zu sehen, geben Sie die Adresse einer `EPROCESS`-Struktur als Argument von `dt` an. Die Adressen fast aller `EPROCESS`-Strukturen im System (außer für den Leerlaufprozess) können Sie durch den Befehl `!process 0 0` abrufen. Da die `KPROCESS`-Struktur das erste Element der `EPROCESS`-Struktur ist, können Sie bei `dt _kprocess` als Adresse der `KPROCESS`-Struktur auch die Adresse der `EPROCESS`-Struktur angeben.

Experiment: Der Befehl !process des Kerneldebuggers

Der Befehl `!process` des Kerneldebuggers zeigt einen Teil der Informationen in einem Prozessobjekt und dessen zugehörigen Strukturen an. Die Ausgabe für einen Prozess besteht jeweils aus zwei Teilen. Als Erstes sehen Sie wie im Folgenden Angaben über den Prozess. Wenn Sie keine Prozessadresse oder -ID angeben, gibt `!process` Informationen über den Prozess aus, dem der zurzeit auf CPU 0 ausgeführte Thread gehört. Auf einem Einzelprozessorsystem kann das auch der WinDbg-Prozess selbst sein (bzw. `livekd`, falls Sie diesen Debugger statt WinDbg verwenden).

```
lkd> !process
PROCESS ffffe0011c3243c0
    SessionId: 2 Cid: 0e38 Peb: 5f2f1de000 ParentCid: 0f08
    DirBase: 38b3e000 ObjectTable: ffffc000a2b22200 HandleCount: <Data Not
                                                                  Accessible>
    Image: windbg.exe
    VadRoot ffffe0011badae60 Vads 117 Clone 0 Private 3563. Modified 228.
        Locked 1.
    DeviceMap ffffc000984e4330
    Token                             ffffc000a13f39a0
    ElapsedTime                       00:00:20.772
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.015
    QuotaPoolUsage[PagedPool]         299512
    QuotaPoolUsage[NonPagedPool]      16240
    Working Set Sizes (now,min,max) (9719, 50, 345) (38876KB, 200KB, 1380KB)
    PeakWorkingSetSize                9947
    VirtualSize                       2097319 Mb
    PeakVirtualSize                   2097321 Mb
    PageFaultCount                    13603
    MemoryPriority                    FOREGROUND
    BasePriority                      8
    CommitCharge                      3994
    Job                               ffffe0011b853690
```

Auf diese Ausgabe von grundlegenden Prozessinformationen folgt eine Liste der Threads in dem Prozess. Diese Ausgabe wird im Experiment »Der Befehl !thread des Kerneldebuggers« in Kapitel 4 erläutert.

Es gibt noch weitere Befehle, die Prozessinformationen anzeigen. Dazu gehört auch `!handle`, der die Prozesshandletabelle ausgibt (siehe Kapitel 8 von Band 2). Sicherheitsstrukturen von Prozessen und Threads sind Thema von Kapitel 7.

→

In dieser Ausgabe erhalten Sie auch die Adresse des PEB. Sie können sie im Befehl `!peb` einsetzen, der im nächsten Experiment beschrieben wird, um sich den PEB eines beliebigen Prozesses auf übersichtliche Weise anzeigen zu lassen. Es ist auch möglich, den regulären Befehl `dt` mit der `_PEB`-Struktur zu verwenden. Da sich der PEB im Benutzeradressraum befindet, ist er jedoch nur im Kontext seines eigenen Prozesses gültig. Um sich den PEB eines anderen Prozesses anzusehen, müssen Sie in WinDbg erst zu diesem Prozess wechseln. Das können Sie mit dem Befehl `.process /P` gefolgt von dem `EPROCESS`-Zeiger tun.

Die Version von WinDbg im aktuellsten Windows 10 SDK zeigt unterhalb der PEB-Adresse einen Hyperlink an, über den Sie die Befehle `.process` und `!peb` automatisch ausführen lassen können.

Der PEB befindet sich im Benutzeradressraum des Prozesses, den er beschreibt. Er enthält Informationen, die der Abbildlader, der Heap-Manager und andere Windows-Komponenten benötigen, um vom Benutzermodus aus auf den Prozess zuzugreifen. Es wäre zu aufwendig, all diese Informationen über Systemaufrufe verfügbar zu machen. Die `EPROCESS`- und die `KPROCESS`-Struktur sind nur vom Kernelmodus aus zugänglich. Die wichtigsten Felder des PEB sehen Sie in Abb. 3–4 Eine ausführliche Erklärung folgt weiter hinten in diesem Kapitel.

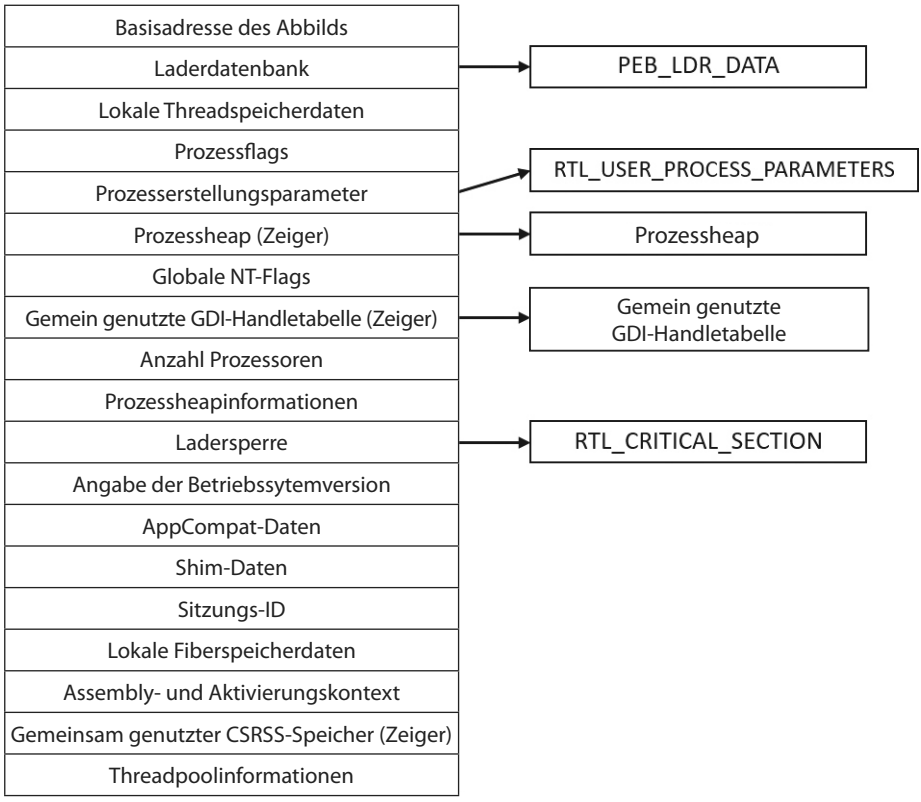


Abbildung 3–4 Wichtige Felder des Prozessumgebungsblocks

Experiment: Den PEB untersuchen

Mit dem Befehl `!peb` des Kerneldebuggers können Sie die PEB-Struktur des Prozesses ausgeben, dem der zurzeit auf CPU 0 ausgeführte Thread gehört. Sie können auch den im vorherigen Experiment gewonnenen PEB-Zeiger als Argument für den Befehl angeben.

```
lkd> .process /P ffffe0011c3243c0 ; !peb 5f2f1de000
PEB at 0000003561545000
   InheritedAddressSpace:    No
   ReadImageFileExecOptions: No
   BeingDebugged:            No
   ImageBaseAddress:         00007ff64fa70000
   Ldr                       00007ffdf52f5200
   Ldr.Initialized:          Yes
   Ldr.InInitializationOrderModuleList: 000001d3d22b3630 . 000001d3d6cddb60
   Ldr.InLoadOrderModuleList:           000001d3d22b3790 . 000001d3d6cddb40
   Ldr.InMemoryOrderModuleList:         000001d3d22b37a0 . 000001d3d6cddb50
     Base TimeStamp Module
     7ff64fa70000 56ccafdd Feb 23 21:15:41 2016 C:\dbg\x64\windbg.exe
     7ffdf51b0000 56cbf9dd Feb 23 08:19:09 2016 C:\WINDOWS\SYSTEM32\ntdll.dll
     7ffdf2c10000 5632d5aa Oct 30 04:27:54 2015 C:\WINDOWS\system32\KERNEL32.DLL
...
```

Die `CSR_PROCESS`-Struktur enthält spezifische Prozessinformationen des Windows-Teilsystems (*Csrss*). Daher ist eine solche Struktur nur mit Windows-Anwendungen verbunden. (Beispielsweise hat *Smss* keine.) Da jede Sitzung ihre eigene Instanz des Windows-Teilsystems aufweist, werden die `CSR_PROCESS`-Strukturen jeweils vom `Csrss`-Prozess der Sitzung gepflegt. Den grundlegenden Aufbau einer `SCR_PROCESS`-Struktur sehen Sie in Abb. 3–5. Eine ausführliche Erklärung folgt weiter hinten in diesem Kapitel.

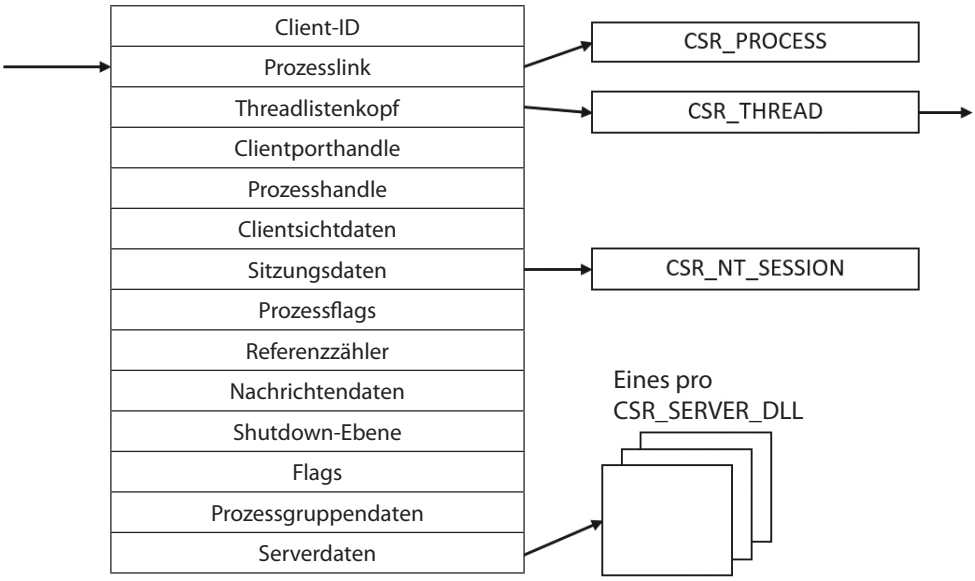


Abbildung 3–5 Felder einer CSR_PROCESS-Struktur

Experiment: Die CSR_PROCESS-Struktur untersuchen

Csrss-Prozesse sind geschützt (mehr darüber weiter hinten in diesem Kapitel), weshalb es nicht möglich ist, einen Benutzermodusdebugger daran anzuhängen (nicht einmal mit erhöhten Rechten oder auf nicht invasive Weise). Stattdessen verwenden wir den Kerneldebugger.

Als Erstes listen wir die Csrss-Prozesse auf:

```
lkd> !process 0 0 csrss.exe
PROCESS ffffe00077ddf080
    SessionId: 0 Cid: 02c0 Peb: c4e3fc0000 ParentCid: 026c
    DirBase: ObjectTable: ffffc0004d15d040 HandleCount: 543.
    Image: csrss.exe

PROCESS ffffe00078796080
    SessionId: 1 Cid: 0338 Peb: d4b4db4000 ParentCid: 0330
    DirBase: ObjectTable: ffffc0004ddff040 HandleCount: 514.
    Image: csrss.exe
```

Als Nächstes ändern wir den Debuggerkontext und lassen ihn auf einen dieser Prozesse verweisen, sodass die Benutzermodusmodule sichtbar werden:

```
lkd> .process /r /P ffffe00078796080
Implicit process is now ffffe000'78796080
Loading User Symbols
.............
```

→

Der Schalter /p ändert den Prozesskontext des Debuggers auf den des angegebenen Prozessobjekts (EPROCESS, was hauptsächlich beim Live-Debugging benötigt wird), und /r fordert, die Benutzermodussymbole zu laden. Anschließend können Sie sich die Module mit dem Befehl lm ansehen oder die CSR_PROCESS-Struktur betrachten:

```
lkd> dt csrss!_csr_process
   +0x000 ClientId                 : _CLIENT_ID
   +0x010 ListLink                 : _LIST_ENTRY
   +0x020 ThreadList               : _LIST_ENTRY
   +0x030 NtSession                : Ptr64 _CSR_NT_SESSION
   +0x038 ClientPort               : Ptr64 Void
   +0x040 ClientViewBase           : Ptr64 Char
   +0x048 ClientViewBounds         : Ptr64 Char
   +0x050 ProcessHandle            : Ptr64 Void
   +0x058 SequenceNumber           : Uint4B
   +0x05c Flags                    : Uint4B
   +0x060 DebugFlags               : Uint4B
   +0x064 ReferenceCount           : Int4B
   +0x068 ProcessGroupId           : Uint4B
   +0x06c ProcessGroupSequence     : Uint4B
   +0x070 LastMessageSequence      : Uint4B
   +0x074 NumOutstandingMessages   : Uint4B
   +0x078 ShutdownLevel            : Uint4B
   +0x07c ShutdownFlags            : Uint4B
   +0x080 Luid                     : _LUID
   +0x088 ServerDllPerProcessData : [1] Ptr64 Void
```

W32PROCESS ist die letzte Systemdatenstruktur im Zusammenhang mit Prozessen, die wir uns ansehen wollen. Sie enthält alle Informationen, die der Grafik- und Fensterverwaltungscode im Windows-Kernel (*Win32k*) benötigt, um die Statusinformationen über die GUI-Prozesse zu unterhalten (die weiter vorn als Prozesse mit mindestens einem USER/GDI-Systemaufruf definiert worden sind). Den grundlegenden Aufbau einer W32PROCESS-Struktur sehen Sie in Abb. 3–6. Da Typinformationen für *Win32k*-Strukturen in öffentlichen Symbolen nicht verfügbar sind, können wir Ihnen leider kein einfaches Experiment zeigen, mit dem Sie sich diese Informationen selbst ansehen könnten. Allerdings ginge eine Erörterung von Datenstrukturen und Funktionsprinzipien für die Grafik ohnehin über den Rahmen dieses Buches hinaus.

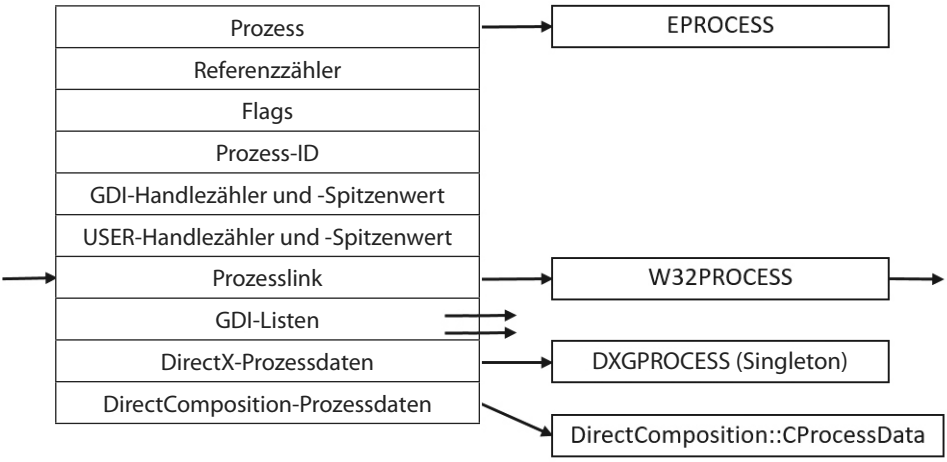


Abbildung 3–6 Felder der `W32PROCESS`-Struktur

Geschützte Prozesse

Im Windows-Sicherheitsmodell kann jeder Prozess, der mit einem Token mit Debugrechten ausgeführt wird (z. B. unter einem Administratorkonto), jegliche Zugriffsrechte für beliebige andere Prozesse auf demselben Computer anfordern. Beispielsweise kann er willkürlich im Prozessspeicher lesen und schreiben, Code injizieren, Threads anhalten und wieder aufnehmen und Informationen über andere Prozesse abfragen. Werkzeuge wie Process Explorer und der Task-Manager benötigen solche Zugriffsrechte, um ihre Aufgabe zu erfüllen.

Dieses Verhalten ist sinnvoll, da es dafür sorgt, dass Administratoren stets die volle Kontrolle über die Codeausführung in dem System haben, doch es kollidiert mit dem Systemverhalten für digitale Rechteverwaltung, das die Medienbranche für Computerbetriebssysteme zur Wiedergabe von modernen, hochwertigen digitalen Inhalten wie Blu-Ray-Medien fordert. Um eine zuverlässige und geschützte Wiedergabe solcher Inhalte zu ermöglichen, wurden in Windows Vista und Windows Server 2008 geschützte Prozesse eingeführt. Sie existieren neben den normalen Windows-Prozessen und schränken die Zugriffsrechte, die andere Prozesse auf dem System anfordern können (selbst diejenigen mit Administratorrechten), erheblich ein.

Jede Anwendung kann geschützte Prozesse erstellen, allerdings lässt das Betriebssystem den Schutz des Prozesses nur dann zu, wenn die Abbilddatei mit einem besonderen Windows-Medienzertifikat signiert wurde. Der Pfad für geschützte Medien (Protected Media Path, PMP) in Windows nutzt geschützte Prozesse, um wertvolle Medien zu schützen. Entwickler von Anwendungen wie DVD-Playern können über die Media-Foundation-API (MF) geschützte Prozesse verwenden.

Der Prozess Audio Device Graph (*Audiodg.exe*) ist geschützt, da über ihn geschützte Musikinhalte decodiert werden können. Damit verwandt ist die Media Foundation Protected Pipeline (*Mfpmp.exe*), die aus ähnlichen Gründen ein geschützter Prozess ist. (Er wird nicht standardmäßig ausgeführt.) Der Clientprozess *Werfaultsecure.exe* der Windows-Fehlerberichterstattung (Windows Error Reporting, WER; siehe Kapitel 8 in Band 2) kann ebenfalls geschützt ausgeführt

werden, da er für den Fall, dass ein geschützter Prozess ausfällt, Zugriff darauf haben muss. Auch der Systemprozess selbst ist geschützt, da einige der Entschlüsselungsinformationen vom Treiber *Ksecdd.sys* generiert und im Benutzermodusspeicher abgelegt werden. Der Schutz des Systemprozesses erfolgt auch, um die Integrität aller Kernelhandles zu wahren (da die Handletabelle des Systemprozesses alle Kernelhandles des Systems enthält). Dass andere Treiber unter Umständen Arbeitsspeicher im Benutzermodus-Adressraum des Systemprozesses zuordnen können (z. B. das Codeintegritätszertifikat und Katalogdaten), ist ein weiterer Grund, den Prozess zu schützen.

Auf Kernelebene gibt es zwei Formen der Unterstützung für geschützte Prozesse. Erstens erfolgt der Großteil der Prozesserstellung im Kernelmodul, um Injektionsangriffe zu vermeiden. (Der Ablauf der Erstellung von normalen und von geschützten Prozessen wird ausführlich im nächsten Abschnitt beschrieben.) Zweitens ist in der `EPROCESS`-Struktur von geschützten Prozessen (und ihren nächsten Verwandten, den im nächsten Abschnitt beschriebenen PPLs [Protected Processes Light]) ein besonderes Bit gesetzt, das das Verhalten von sicherheitsbezogenen Routinen im Prozess-Manager so ändert, dass bestimmte Zugriffsrechte, die Administratoren normalerweise erhalten, verweigert werden. Die einzigen Zugriffsrechte, die für geschützte Prozesse vergeben werden, sind `PROCESS_QUERY/SET_LIMITED_INFORMATION`, `PROCESS_TERMINATE` und `PROCESS_SUSPEND_RESUME`. Manche Zugriffsrechte werden auch für die Threads innerhalb der geschützten Prozesse deaktiviert. Das sehen wir uns in Kapitel 4 im Abschnitt »Interne Strukturen von Threads« genauer an.

Da das Programm Process Explorer reguläre Windows-APIs des Benutzermodus verwendet, um Informationen über Prozessinterna abzurufen, kann es bei geschützten Prozessen bestimmte Operationen nicht durchführen. Dagegen läuft WinDbg im Kerneldebuggingmodus und nutzt daher die Kernelmodus-Infrastruktur, um Informationen zu gewinnen, weshalb es damit möglich ist, sämtliche Informationen anzuzeigen. Das Verhalten von Process Explorer beim Umgang mit einem geschützten Prozess wie *Audiodg.exe* lernen Sie in dem Experiment im Abschnitt »Innere Mechanismen von Threads« von Kapitel 4 kennen.

Wie bereits in Kapitel 1 erwähnt, müssen Sie zum lokalen Kerneldebugging im Debuggingmodus starten (indem Sie `bcdedit /debug` oder die erweiterten Bootoptionen in *Msconfig* verwenden). Das dient dazu, die Gefahr von debuggergestützten Angriffen auf geschützte Prozesse und den PMP zu verringern. Beim Start im Debuggingmode können High-Definition-Inhalte nicht wiedergegeben werden.

Durch die Einschränkung der Zugriffsrechte kann der Kernel geschützte Prozesse wie in einer Sandbox zuverlässig gegen Zugriff vom Benutzermodus abschirmen. Allerdings sind geschützte Prozesse durch ein Flag in der `EPROCESS`-Struktur gekennzeichnet, sodass Administratoren immer noch in der Lage sind, einen Kernelmodustreiber zu laden, der dieses Flag ändert. Das allerdings würde eine Verletzung des PMP-Modells darstellen und als schädliches Verhalten eingestuft, weshalb das Laden eines solchen Treibers auf einem 64-Bit-System wahrscheinlich blockiert würde, da die Codesignierungsrichtlinie für den Kernelmodus eine digitale Signierung von Schadcode nicht zulässt. Außerdem würden der Kernelmoduspatchschutz (PatchGuard, siehe Kapitel 7) sowie der Treiber für geschützte Umgebung und Authentifizierung (Protected

Environment and Authentication, *Peauth.sys*) solche Versuche erkennen und melden. Selbst auf 32-Bit-Systemen muss der Treiber von der PMP-Richtlinie anerkannt werden, da die Wiedergabe sonst angehalten wird. Diese Richtlinie wird von Microsoft implementiert und nicht von irgendeiner Kernelerkennung. Dieser Block würde ein manuelles Eingreifen von Microsoft erfordern, um die Signatur als bösartig zu erkennen und den Kernel zu aktualisieren.

Protected Process Light (PPL)

Wie Sie gerade gesehen haben, ging es beim ursprünglichen Modell für geschützte Prozesse vor allem um DRM-geschützte Inhalte. Ab Windows 8.1 und Windows Server 2012 R2 wurde das Modell jedoch um Protected Process Light (PPL), also sozusagen die »Light-Version« von geschützten Prozessen erweitert.

PPLs werden auf die gleiche Weise geschützt wie herkömmliche geschützte Prozesse: Benutzermoduscode (selbst solcher mit erhöhten Rechten) kann nicht in sie eindringen und Threads injizieren oder Informationen über die geladenen DLLs gewinnen. Beim PPL-Modell kommen jedoch noch Attributwerte hinzu. Die verschiedenen Signierer haben unterschiedliche Vertrauensebenen, was dazu führt, dass manche PPLs stärker geschützt sind als andere.

Da sich die digitale Rechteverwaltung von der reinen Multimedia-DRM auch zur Windows-Lizenz-DRM und Windows-Store-DRM entwickelt hat, werden herkömmliche geschützte Prozesse jetzt auch anhand ihres Signiererwerts unterschieden. Die einzelnen anerkannten Signierer legen auch die Zugriffsrechte fest, die weniger geschützten Prozessen verweigert werden. Beispielsweise sind normalerweise nur die Zugriffsmasken `PROCESS_QUERY/SET_LIMITED_INFORMATION` und `PROCESS_SUSPEND_RESUME` zugelassen. Dagegen ist `PROCESS_TERMINATE` für bestimmte PPL-Signierer nicht zugelassen.

Tabelle 3–1 zeigt die zulässigen Werte für das Schutzflag in der `EPROCESS`-Struktur.

Internes Symbol für die Schutzebene des Prozesses	Schutztyp	Signierer
`PS_PROTECTED_SYSTEM` (0x72)	Geschützt	WinSystem
`PS_PROTECTED_WINTCB` (0x62)	Geschützt	WinTcb
`PS_PROTECTED_WINTCB_LIGHT` (0x61)	PPL	WinTcb
`PS_PROTECTED_WINDOWS` (0x52)	Geschützt	Windows
`PS_PROTECTED_WINDOWS_LIGHT` (0x51)	PPL	Windows
`PS_PROTECTED_LSA_LIGHT` (0x41)	PPL	Lsa
`PS_PROTECTED_ANTIMALWARE_LIGHT` (0x31)	PPL	Anti-malware
`PS_PROTECTED_AUTHENTICODE` (0x21)	Geschützt	Authenticode
`PS_PROTECTED_AUTHENTICODE_LIGHT` (0x11)	PPL	Authenticode
`PS_PROTECTED_NONE` (0x00)	Keine	Keine

Tabelle 3–1 Gültige Schutzwerte für Prozesse

Wie Sie in Tabelle 3–1 sehen, sind mehrere Signierer definiert, hier in der Reihenfolge absteigender Priorität angegeben. `WinSystem` ist der Signierer höchster Priorität und wird vom Systemprozess sowie von minimalen Prozessen wie dem Speicherkomprimierungsprozess ver-

wendet. Für Benutzermodusprozesse hat WinTCB (Windows Trusted Computer Base) die höchste Priorität und wird zum Schutz kritischer Prozesse genutzt, die der Kernel genau kennt und für die er seine Sicherheitsmaßnahmen reduzieren mag. Geschützte Prozesse haben stets mehr Befugnisse als PPLs, Prozesse mit einem höherwertigen Signierer haben Zugriff auf Prozesse mit einem Signierer geringerer Priorität, aber nicht umgekehrt. Tabelle 3–2 zeigt die Signierer-ebenen (je höher der Wert, umso mehr Befugnisse hat der Signierer) und einige Beispiele ihrer Nutzung. Sie können sie im Debugger mit dem Typ _PS_PROTECTED_SIGNER ausgeben.

Signierer (PS_PROTECTED_SIGNER**)**	**Ebene**	**Verwendung**
PsProtectedSignerWinSystem	7	System- und minimale Prozesse (einschließlich Pico-Prozesse)
PsProtectedSignerWinTcb	6	Kritische Windows-Komponenten. PROCESS_TERMINATE wird verweigert.
PsProtectedSignerWindows	5	Wichtige Windows-Komponenten, die mit sensiblen Daten umgehen
PsProtectedSignerLsa	4	*Lsass.exe* (falls zur geschützten Ausführung eingerichtet)
PsProtectedSignerAntimalware	3	Anti-Malware-Dienste und Prozesse, auch von Drittanbietern. PROCESS_TERMINATE wird verweigert.
PsProtectedSignerCodeGen	2	NGEN (native .NET-Codegenerierung)
PsProtectedSignerAuthenticode	1	Hosting von DRM-Inhalten und Laden von Benutzermodus-Schriftarten
PsProtectedSignerNone	0	Ungültig (kein Schutz)

Tabelle 3–2 Signierer und ihre Ebenen

Vielleicht fragen Sie sich an dieser Stelle, was einen Schadprozess daran hindert, sich als geschützten Prozess auszugeben und sich gegen Anti-Malware-Anwendungen abzuschirmen. Da für eine Ausführung als geschützter Prozess das Windows-Media-DRM-Zertifikat nicht mehr erforderlich ist, hat Microsoft sein Codeintegritätsmodul so erweitert, dass es die beiden besonderen EKU-OIDs (Enhanced Key Usage) 1.3.6.1.4.1.311.10.3.22 und 1.3.6.4.1.311.10.3.20 versteht, die in Zertifikaten zur Codesignierung verwendet werden können. Ist eine dieser EKUs vorhanden, werden die im Zertifikat hartcodierten Strings für den Signierer und den Aussteller zusammen mit weiteren möglichen EKUs mit den verschiedenen Werten für geschützte Signierer verknüpft. Beispielsweise kann der Herausgeber Microsoft Windows den geschützten Signiererwert PsProtectedSignerWindows gewähren, aber nur dann, wenn die EKU für Windows System Component Verification (1.3.6.1.4.1.311.10.3.6) ebenfalls vorhanden ist. Als Beispiel zeigt Abb. 3–7 das Zertifikat für *Smss.exe*, der als WinTcb-Light ausgeführt werden darf.

Die Schutzebene eines Prozesses bestimmt auch, welche DLLs er laden darf, denn ansonsten kann ein legitimer geschützter Prozess durch einen Bug oder eine einfache Dateiersetzung dazu gebracht werden, eine Drittanbieter- oder eine Schadbibliothek zu laden, die dann mit derselben Schutzebene ausgeführt würde wie der Prozess. Dazu erhält jeder Prozess eine »Signaturebene«, die im Feld SignatureLevel der EPROCESS-Struktur gespeichert wird. In einer internen Nachschlagetabelle wird dann die zugehörige DLL-Signaturebene gesucht, die als SectionSignatureLevel in der EPROCESS-Struktur steht. Jede in den Prozess geladene DLL wird

von der Codeintegritätskomponente auf diese Weise überprüft wie die ausführbare Hauptdatei. Beispielsweise kann ein Prozess mit dem Signierer `WinTcb` nur DLLs mit dem Signierer `Windows` oder höher laden.

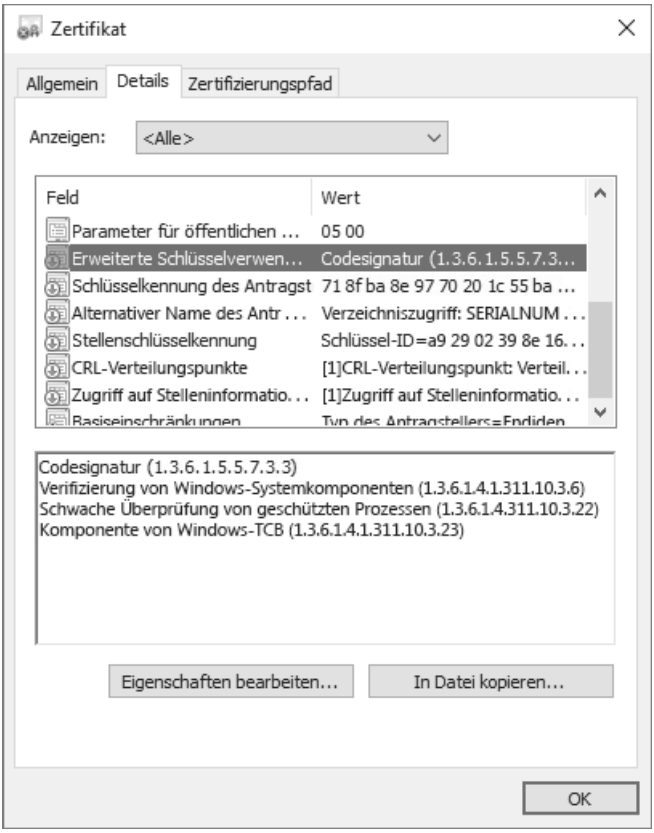


Abbildung 3–7 *Smss*-Zertifikat

In Windows 10 und Windows Server 2016 sind die Prozesse *smss.exe*, *csrss.exe*, *services.exe* und *wininit.exe* mit `WinTcb-Lite` PPL-signiert. *Lsass.exe* läuft in Windows für ARM-Geräte (z. B. Windows Mobile 10) als PPL und kann auf x86/x64 als PPL ausgeführt werden, wenn er durch eine Registrierungseinstellung oder eine Richtlinie entsprechend konfiguriert ist (siehe Kapitel 7). Außerdem sind einige Dienste so eingerichtet, dass sie als Windows-PPL oder als geschützter Prozess laufen, z. B. *sppsvc.exe* (Software Protection Platform). Auch manche Diensthostprozesse (*Svchost.exe*) werden mit dieser Schutzebene ausgeführt, da viele Dienste ebenfalls geschützt sind, z. B. der AppX-Bereitstellungsdienst und der Dienst des Windows-Teilsystems für Linux. Weitere Informationen über solche geschützten Dienste erhalten Sie in Kapitel 9.

Für die Sicherheit des Systems ist es von entscheidender Bedeutung, dass diese Kernbinärdateien als TCB ausgeführt werden. So hat beispielsweise *Csrss.exe* Zugriff auf private APIs, die vom Windows-Manager (*Win32k.sys*) implementiert werden und über die ein Angreifer mit Administratorrechten Zugriff auf sensible Teile des Kernels gewinnen könnte. *Smss.exe* und

Wininit.exe implementieren die Logik für Systemstart und -verwaltung, die unbedingt ausgeführt werden müssen, ohne dass sich ein Administrator daran zu schaffen machen kann. Windows garantiert, dass diese Binärdateien stets als `WinTcb-Lite` ausgeführt werden. Dadurch ist es beispielsweise nicht möglich, dass jemand sie startet, ohne beim Aufruf von `CreateProcess` die richtige Prozessschutzebene in den Prozessattributen anzugeben. Diese Garantie wird als *Mindest-TCB-Liste* bezeichnet und erzwingt für die Prozesse aus Tabelle 3–3, die sich in einem Systempfad befinden, eine Mindestschutz- oder -signierebene unabhängig von der Eingabe des Aufrufers.

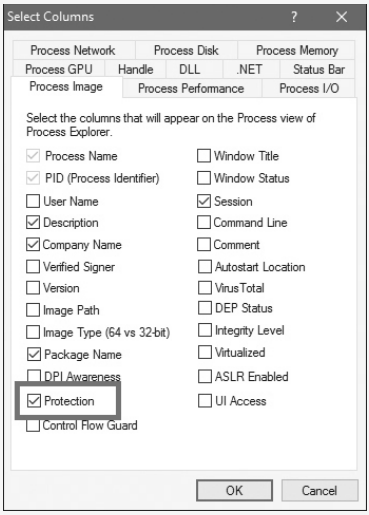
Prozess	Mindestsignierebene	Mindestschutzebene
Smss.exe	Aus der Schutzebene abgeleitet	`WinTcb-Lite`
Csrss.exe	Aus der Schutzebene abgeleitet	`WinTcb-Lite`
Wininit.exe	Aus der Schutzebene abgeleitet	`WinTcb-Lite`
Services.exe	Aus der Schutzebene abgeleitet	`WinTcb-Lite`
Werfaultsecure.exe	Aus der Schutzebene abgeleitet	`WinTcb-Full`
Sppsvc.exe	Aus der Schutzebene abgeleitet	`Windows-Full`
Genvalobj.exe	Aus der Schutzebene abgeleitet	`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`[1]	0

Tabelle 3–3 Mindest-TCB

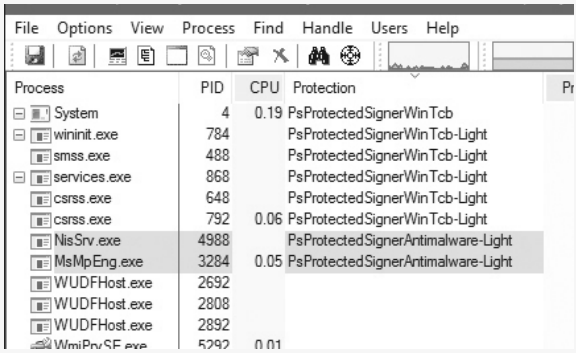
[1] Nur auf UEFI-Firmwaresystemen

Experiment: Geschützte Prozesse in Process Explorer anzeigen

In diesem Experiment sehen wir uns an, wie Process Explorer geschützte Prozesse (beider Arten) anzeigt. Starten Sie Process Explorer und aktivieren Sie auf der Registerkarte *Process Image* das Kontrollkästchen *Protection*, um die gleichnamige Spalte einzublenden:



Sortieren Sie nun die Schalter *Protection* in absteigender Reihenfolge und scrollen Sie bis ganz nach oben. Die geschützten Prozesse werden mit ihrem Schutztyp angezeigt. Der folgende Screenshot wurde auf einem x64-Computer mit Windows 10 angefertigt:



Wenn Sie einen geschützten Prozess markieren und einen Blick auf den unteren Teil werfen, sehen Sie in der DLL-Ansicht nichts. Das liegt daran, dass Process Explorer die geladenen Module mithilfe von Benutzermodus-APIs abfragt, wozu ein Zugriff erforderlich ist, der für geschützte Prozesse nicht gewährt wird. Die Ausnahme bildet der Systemprozess, der zwar geschützt ist, für den Process Explorer aber trotzdem die Liste der geladenen Kernelmodule (hauptsächlich Treiber) anzeigt, da Systemprozesse keine DLLs enthalten. Für diese Anzeige verwendet Process Explorer die System-API `EnumDeviceDrivers`, die kein Prozesshandle benötigt.

→

In der Handleansicht werden die gesamten Handleinformationen angezeigt. Dazu verwendet Process Explorer eine nicht dokumentierte API, die ebenfalls kein bestimmtes Prozesshandle benötigt. Das Programm kann die Prozesse identifizieren, da mit den Informationen über die Handles auch die PIDs der zugehörigen Prozesse zurückgegeben werden.

Unterstützung für Drittanbieter-PPLs

Der PPL-Mechanismus dehnt die Schutzmöglichkeiten für Prozesse auch auf ausführbare Dateien aus, die nicht von Microsoft entwickelt wurden. Ein Beispiel dafür bildet Anti-Malware-Software. Typische AM-Produkte bestehen aus den folgenden drei Hauptkomponenten:

- Einem Kerneltreiber, der E/A-Anforderungen zum Dateisystem und zum Netzwerk abfängt und über Objekt-, Prozess- und Thread-Callbacks Blockiermöglichkeiten einrichtet
- Einem Benutzermodusdienst (gewöhnlich unter einem privilegierten Konto), der die Treiberrichtlinien konfiguriert, Benachrichtigungen des Treibers über »interessante« Ereignisse empfängt (z. B. über eine infizierte Datei) und mit einem lokalen Server oder dem Internet kommunizieren kann
- Einem Benutzermodus-GUI-Prozess, der Informationen an den Benutzer vermittelt und ihn ggf. Entscheidungen treffen lässt

Eine Angriffsmöglichkeit für Malware besteht darin, Code in einen Prozess mit erhöhten Rechten zu injizieren – am besten noch in einen Anti-Malware-Dienst, um diesen zu modifizieren oder auszuschalten. Läuft der AM-Dienst aber als PPL, so ist weder eine Codeinjektion noch eine Beendigung des Prozesses erlaubt. Dadurch ist die AM-Software besser gegen Malware geschützt (sofern diese Exploits auf Kernelebene einsetzt).

Um das zu ermöglichen, muss der zuvor beschriebene AM-Kerneltreiber über einen zugehörigen ELAM-Treiber (Early-Launch Anti Malware) verfügen. ELAM wird in Kapitel 7 ausführlicher beschrieben. Lassen Sie uns hier nur betonen, dass solche Treiber ein besonderes Anti-Malware-Zertifikat benötigen, das von Microsoft ausgestellt wird (nach eingehender Überprüfung des Softwareherstellers). Ein solcher Treiber kann in seiner ausführbaren Hauptdatei (PE) einen benutzerdefinierten Ressourcenabschnitt namens `ELAMCERTIFICATEINFO` enthalten, der drei zusätzliche Signierer mit jeweils drei zusätzlichen EKUs beschreibt (wobei die Signierer anhand ihres öffentlichen Schlüssels und die EKUs anhand ihrer OIDs bezeichnet werden). Wenn das Codeintegritätssystem feststellt, dass eine Datei von einem dieser drei Signierer mit einem dieser drei EKUs signiert wurde, erlaubt es dem Prozess, einen PPL von `PS_PROTECTED_ANTIMALWARE_LIGHT` (`0x31`) anzufordern. Das Standardbeispiel dafür ist Windows Defender, die eigene AM-Software von Microsoft. In Windows 10 ist ihr Dienst (*MsMpEng.exe*) mit dem AM-Zertifikat signiert, um ihn besser gegen Malwareangriffe gegen ihn selbst zu schützen. Das Gleiche gilt auch für den Network Inspection Server (*NisSvc.exe*).

Minimale und Pico-Prozesse

Bei den Prozessen, die wir bisher betrachtet haben, sah es so aus, als seien sie zur Ausführung von Benutzermoduscode da und würden dazu erhebliche Mengen an Datenstrukturen im Arbeitsspeicher unterbringen. Allerdings werden nicht alle Prozesse für diesen Zweck verwendet. Wie wir bereits festgestellt haben, ist der Systemprozess lediglich ein Container für die meisten Systemthreads, damit diese bei ihrer Ausführung keine anderen Benutzermodusprozesse kontaminieren, und für die Treiberhandles (Kernelhandles), damit auch diese nicht in den Besitz irgendwelcher Anwendungen geraten.

Minimale Prozesse

Wenn die Funktion `NtCreateProcessEx` vom Kernelmodus aufgerufen wird und über ein bestimmtes Flag verfügt, verhält sie sich anders als sonst und verursacht die Ausführung der API `PsCreateMinimalProcess`. Dadurch wird ein Prozess erstellt, dem viele der zuvor betrachteten Strukturen fehlen:

- Es wird kein Benutzermodus-Adressraum eingerichtet, weshalb es auch den PEB und die zugehörigen Strukturen nicht gibt.
- Dem Prozess wird keine NTDLL und auch keine Loader- oder API-Set-Information zugeordnet.
- Es wird kein Abschnittsobjekt an den Prozess gebunden. Das heißt, mit der Ausführung oder dem Namen (der leer oder ein willkürlicher String sein kann) wird keine ausführbare Abbilddatei verknüpft.
- In den `EPROCESS`-Flags wird das Flag `Minimal` gesetzt. Dadurch werden auch alle Threads zu Minimalthreads und es werden jegliche Benutzermodus-Zuweisungen wie der TEB (siehe Kapitel 4) oder der Benutzermodusstack vermieden.

Wie Sie in Kapitel 2 gesehen haben, gibt es in Windows 10 mindestens zwei minimale Prozesse, nämlich den System- und den Speicherkomprimierungsprozess. Ist die virtualisierungsgestützte Sicherheit aktiviert, kann mit dem Secure-System-Prozess noch ein dritter hinzukommen (siehe Kapitel 2 und 7).

Die zweite Möglichkeit, um minimale Prozesse auf einem Windows 10-System ausführen zu lassen, besteht darin, das optionale Windows-Teilsystem für Linux (WSL) zu aktivieren (siehe Kapitel 2). Dadurch wird ein im Lieferumfang von Windows enthaltener Pico-Provider installiert, der aus den Treibern *Lxss.sys* und *LxCore.sys* besteht.

Pico-Prozesse

Minimale Prozesse sind nur von beschränktem Nutzen, um Kernelkomponenten Zugriff auf den virtuellen Benutzermodus-Adressraum zu geben und diesen zu schützen. Pico-Prozesse dagegen spielen eine wichtigere Rolle, da sie einer besonderen Komponente – dem *Pico-Anbieter* – ermöglichen, die meisten Aspekte ihrer Ausführung zu steuern. Damit kann ein solcher Anbieter letzten Endes das Verhalten eines völlig anderen Betriebssystemkernels emulieren, ohne dass die

zugrunde liegende Benutzermodus-Binärdatei weiß, dass sie auf einem Windows-Betriebssystem läuft. Im Grunde genommen handelt es sich dabei um eine Implementierung des Microsoft Research-Projekts Drawbridge, das auf ähnliche Weise auch SQL Server für Linux unterstützt (wenn auch mit einem Windows-Bibliotheksbetriebssystem oberhalb des Linux-Kernels).

Um auf einem System Pico-Prozesse verwenden zu können, muss zunächst ein Anbieter vorhanden sein. Er kann mit der API `PsRegisterPicoProvider` registriert werden, wobei jedoch eine besondere Regel gilt: Der Pico-Anbieter muss vor allen anderen Drittanbietertreibern geladen werden (einschließlich der Boottreiber). Nur jeweils ein einziger aus einem eingeschränkten Satz von etwa einem Dutzend Kerntreibern darf diese API aufrufen, bevor diese Funktionalität deaktiviert wird. Diese Kerntreiber müssen mit einem Microsoft-Signiererzertifikat und einer Windows-Komponenten-EKU signiert sein. Auf Windows-Systemen, auf denen die optionale WSL-Komponente aktiviert ist, handelt es sich hierbei um den Treiber *Lxss.sys*, der als Stubtreiber fungiert, bis etwas später der Treiber *LxCore.sys* geladen wird und die Verantwortung für den Pico-Anbieter übernimmt, indem er die Dispatchtabellen auf sich überträgt. Zurzeit kann sich auch nur ein einziger Kerntreiber als Pico-Anbieter registrieren.

Wenn ein Pico-Anbieter die Registrierungs-API aufruft, erhält er eine Reihe von Funktionszeigern, mit denen er Pico-Prozesse erstellen und verwalten kann:

- Zwei Funktionen, um Pico-Prozesse bzw. Pico-Threads zu erstellen
- Eine Funktion, um den Kontext (einen willkürlichen Zeiger, mit dem der Anbieter bestimmte Daten speichern kann) eines Pico-Prozesses abzurufen, eine, um ihn festzulegen, und zwei weitere Funktionen, um das Gleiche für Pico-Threads zu tun. Damit wird das Feld `PicoContext` in der `ETHREAD`- bzw. `EPROCESS`-Struktur ausgefüllt.
- Eine Funktion, um die CPU-Kontextstruktur (`CONTEXT`) eines Pico-Threads abzurufen, und eine, um ihn festzulegen
- Eine Funktion, um das `FS`- und `GS`-Segment eines Pico-Threads zu ändern. Diese Segmente werden normalerweise von Benutzermoduscode verwendet, um auf eine lokale Threadstruktur zu verweisen (wie den TEB in Windows).
- Zwei Funktionen, um Pico-Threads bzw. Pico-Prozesse zu beenden
- Eine Funktion, um einen Pico-Thread anzuhalten, und eine weitere, um ihn wieder aufzunehmen

Mithilfe dieser Funktionen kann der Pico-Anbieter maßgeschneiderte Prozesse und Threads erstellen, wobei deren ursprünglicher Startzustand, die Segmentregister und die zugehörigen Daten seiner Kontrolle unterliegen. Das allein macht es aber noch nicht möglich, ein anderes Betriebssystem zu emulieren. Es gibt noch einen zweiten Satz von Funktionszeigern, die jedoch vom Anbieter zum Kernel übertragen werden und als Callbacks fungieren, wenn ein Pico-Thread oder -Prozess bestimmte Aktivitäten ausführt.

- Ein Callback für den Fall, dass ein Pico-Thread mithilfe der Anweisung `SYSCALL` einen Systemaufruf vornimmt
- Ein Callback für den Fall, dass ein Pico-Thread eine Ausnahme auslöst
- Ein Callback für den Fall, dass in einem Pico-Thread ein Fehler bei einer Sondierungs- und Sperroperation an einer Speicherdeskriptorliste (MDL) auftritt

- Ein Callback für den Fall, dass ein Aufrufer den Namen eines Pico-Prozesses anfordert
- Ein Callback für den Fall, dass die Ereignisverfolgung für Windows (ETW) die Benutzermodus-Stackablaufverfolgung eines Pico-Prozesses anfordert
- Ein Callback für den Fall, dass eine Anwendung versucht, ein Handle für einen Pico-Prozess oder -Thread zu öffnen
- Ein Callback für den Fall, dass jemand die Beendigung eines Pico-Prozesses anfordert
- Ein Callback für den Fall, dass ein Pico-Prozess oder -Thread unerwartet beendet wird

Der Pico-Anbieter nutzt auch den in Kapitel 7 beschriebenen Kernelpatchschutz (Kernel Patch Protection, KPP), um seine Callbacks und Systemaufrufe zu schützen und um zu verhindern, dass sich gefälschte oder schädliche Pico-Anbieter oberhalb von ihm registrieren.

Angesichts dieses beispiellosen Zugriffs auf alle möglichen Benutzer-Kernel-Übergänge und sichtbaren Kernel-Benutzer-Interaktionen zwischen sich selbst und der Welt können Pico-Prozesse und -Threads offensichtlich komplett von einem Pico-Anbieter (und entsprechenden Benutzermodusbibliotheken) gekapselt werden, um als Wrapper für eine ganz andere Kernelimplementierung als die von Windows zu dienen. (Wobei es natürlich Ausnahmen gibt, da nach wie vor die Regeln für Threadplanung und Speicherverwaltung gelten, z. B. für Commits.) Ordnungsgemäß geschriebene Anwendungen sollten nicht von solchen internen Algorithmen beeinflusst werden, da diese sich sogar in dem Betriebssystem ändern können, in dem die Anwendungen normalerweise laufen.

Pico-Anbieter sind daher letzten Endes maßgeschneiderte Kernelmodule, die die notwendigen Callbacks für die Reaktion auf die von einem Pico-Prozess hervorgerufenen Ereignisse (siehe weiter vorn) implementieren. Dadurch kann das WSL unveränderte Linux-ELF-Binärdateien im Benutzermodus ausführen. Eingeschränkt wird diese Möglichkeit durch die Vollständigkeit der Systemaufrufemulation und des damit verbundenen Funktionsumfangs.

Zur Abrundung des Themas sehen Sie in Abb. 3–8 die Strukturen von NT-, minimalen und Pico-Prozessen.

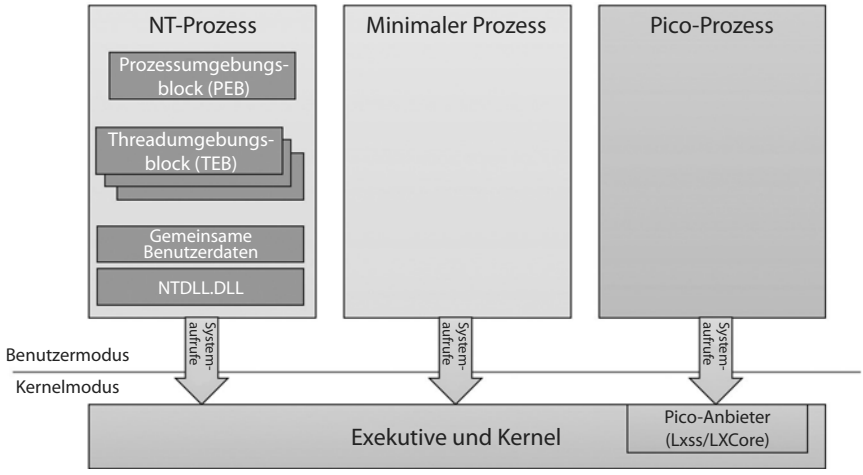


Abbildung 3–8 Arten von Prozessen

Trustlets (sichere Prozesse)

Wie bereits in Kapitel 2 erwähnt, enthält Windows neue VBS-Merkmale (Virtualization-Based Security) wie DeviceGuard und CredentialGuard, die mithilfe eines Hypervisors die Sicherheit des Betriebssystems und der Benutzerdaten verstärken. Eines dieser Merkmale, nämlich CredentialGuard (ausführlich beschrieben in Kapitel 7), läuft in der neuen Umgebung des isolierten Benutzermodus, der zwar immer noch unprivilegiert ist (Ring 3), aber eine virtuelle Vertrauensebene von 1 (VTL 1) aufweist. Dadurch ist er von der regulären Ebene VTL 0 geschützt, auf der sich sowohl der NT-Kernel (Ring 0) als auch die Anwendungen (Ring 3) befinden. Sehen wir uns nun an, wie der Kernel solche Prozesse zur Ausführung einrichtet und welche Datenstrukturen diese Prozesse verwenden.

Der Aufbau von Trustlets

Trustlets sind zwar reguläre PE-Dateien (Windows Portable Executables), weisen aber einige IUM-spezifische Eigenschaften auf:

- Aufgrund der eingeschränkten Anzahl von Systemaufrufen, die für Trustlets zur Verfügung stehen, können sie nur Funktionen aus einer eingeschränkten Menge von Windows-System-DLLs importieren (C/C++ Runtime, KernelBase, Advapi, RPC Runtime, CNG Base Crypto und NTDLL). Mathematische DLLs, die nur Datenstrukturen bearbeiten (wie NTLM, ASN.1 usw.), können jedoch ebenfalls verwendet werden, da sie keine Systemaufrufe durchführen.
- Sie können Funktionen von einer IUM-spezifischen System-DLL namens `Iumbase` importieren, die ihnen zur Verfügung gestellt wird. Diese DLL enthält die grundlegende IUM-System-API mit Unterstützung für Mailslots, Speicherboxen, Kryptografie usw. Diese Bibliothek führt letzten Endes Aufrufe in *Iumdll.dll* durch, der VTL-1-Version von *Ntdll.dll*, und enthält sichere Systemaufrufe (also solche, die vom sicheren Kernel implementiert werden und nicht an den normalen VTL-0-Kernel übergeben werden).
- Sie enthalten den PE-Abschnitt `.tPolicy` mit der exportierten globalen Variablen `s_IumPolicyMetadata`. Damit werden Metadaten für den sicheren Kernel bereitgestellt, um Richtlinieneinstellungen für den VTL-0-Zugriff des Trustlets zu implementieren (z. B. um Debugging zu erlauben, Unterstützung für Absturzabbilder bereitzustellen usw.).
- Sie werden mit einem Zertifikat signiert, dass die IUM-EKU enthält (1.3.6.1.4.311.10.3.37). Abb. 3–9 zeigt die Zertifikatdaten für *LsaIso.exe* mit dieser EKU.

Außerdem muss beim Start von Trustlets mit `CreateProcess` ein besonderes Prozessattribut verwendet werden, um die Ausführung im IUM anzufordern und um besondere Starteigenschaften anzugeben. Die Richtlinien-Metadaten und die Prozessattribute beschreiben wir in den folgenden Abschnitten.

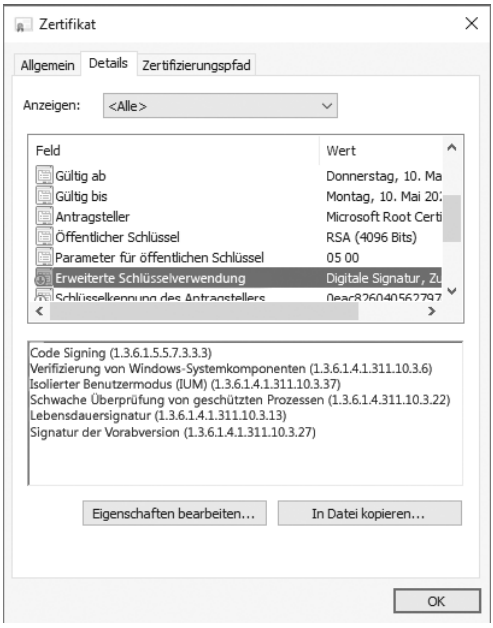


Abbildung 3–9 Trustlet-EKU im Zertifikat

Richtlinien-Metadaten für Trustlets

Zu den Richtlinien-Metadaten gehören verschiedene Optionen, um festzulegen, wie »zugänglich« das Trustlet von VLT 0 aus sein soll. Sie werden durch eine Struktur beschrieben, die sich in der zuvor erwähnten exportierten Variablen `s_IumPolicyMetadaten` befindet, und enthalten die Versionsnummer (zurzeit 1) und die Trustlet-ID, also eine eindeutige Zahl zur Bezeichnung eines bestimmten der bekannten Trustlets. Beispielsweise hat *BioIso.exe* die Trustlet-ID 4. Des Weiteren gehört zu den Metadaten ein Array mit Richtlinienoptionen. Zurzeit werden die Optionen aus Tabelle 3–4 unterstützt. Da diese Richtlinien Teil der signierten ausführbaren Daten sind, verletzt jeder Versuch, sie zu ändern, die IUM-Signatur und verhindert damit eine Ausführung.

Richtlinie	Bedeutung	Weitere Informationen
ETW	Aktiviert oder deaktiviert ETW	
Debug	Konfiguriert das Debugging	Das Debugging kann jederzeit aktiviert sein; nur bei deaktiviertem SecureBoot; oder bei Bedarf über einen Challenge/Response-Mechanismus.
Crash Dump	Aktiviert oder deaktiviert Absturzabbilder	
Crash Dump Key	Gibt den öffentlichen Schlüssel zur Verschlüsselung von Absturzabbildern an	Absturzabbilder können an das Microsoft-Produktteam übermittelt werden, das über den privaten Schlüssel für die Entschlüsselung verfügt.

→

Richtlinie	Bedeutung	Weitere Informationen
Crash Dump GUID	Gibt den eindeutigen Bezeichner für den Absturzabbildschlüssel an	Dadurch kann das Produktteam mehrere Schlüssel verwenden bzw. erkennen.
Parent Security Descriptor	SDDL-Format	Dient zur Validierung, dass der Besitzer- oder Elternprozess der erwartete ist.
Parent Security Descriptor Revision	Revisions-ID des SDDL-Formats	Dient zur Validierung, dass der Besitzer- oder Elternprozess der erwartete ist.
SVN	Sicherheitsversion	Eine eindeutige Zahl, die das Trustlet (zusammen mit seiner ID) bei der Verschlüsselung von AES256/GCM-Nachrichten nutzen kann.
Device ID	PCI-Bezeichner des sicheren Geräts	Das Trustlet kann nur mit einem sicheren Gerät kommunizieren, das den entsprechenden PCI-Bezeichner aufweist.
Capability	Aktiviert VTL-1-Fähigkeiten	Aktiviert den Zugriff auf die API Create Secure Section; DRM- und Benutzermodus-MMIO-Zugriff auf sichere Geräte; und APIs für die sichere Speicherung.
Scenario ID	Gibt die Szenario-ID für die Binärdatei an	Trustlets müssen diesen GUID angeben, wenn sie sichere Abbildabschnitte erstellen, um sicherzustellen, dass es sich um ein bekanntes Szenario handelt.

Tabelle 3–4 Trustlet-Richtlinienoptionen

Attribute von Trustlets

Der Start eines Trustlets erfordert die ordnungsgemäße Verwendung des Attributs `PS_CP_SECURE_PROCESS`. Es dient zur Bestätigung, dass der Aufrufer wirklich ein Trustlet erstellen möchte, und zur Überprüfung, dass das Trustlet, das der Aufrufer auszuführen glaubt, auch tatsächlich das Trustlet ist, das ausgeführt wird. Dazu wird eine Trustlet-ID in das Attribut eingebettet. Sie muss mit der Trustlet-ID in den Richtlinien-Metadaten übereinstimmen. Es können auch noch weitere Attribute angegeben werden, die in Tabelle 3–5 aufgeführt sind.

Attribut	Bedeutung	Weitere Informationen
Mailbox Key	Dient zum Abrufen von Postfachdaten	Über Postfächer kann ein Trustlet Daten mit VLT-0-Anwendungen gemeinsam nutzen, sofern der Trustlet-Schlüssel bekannt ist.
Collaboration ID	Legt die Kollaborations-ID fest, die zur Verwendung der IUM-API Secure Storage benötigt wird	Die Secure-Storage-API ermöglicht Trustlets mit gemeinsamer Kollaborations-ID, Daten untereinander gemeinsam zu nutzen. Ist keine Kollaborations-ID vorhanden, wird stattdessen die ID der Trustlet-Instanz verwendet.
TK Session ID	Die Sitzungs-ID, die bei kryptografischen Vorgängen verwendet wird	

Tabelle 3–5 Trustlet-Attribute

Integrierte System-Trustlets

Zurzeit enthält Windows 10 die fünf Trustlets, die in Tabelle 3–6 aufgeführt sind. Die Trustlet-ID 0 steht für den sicheren Kernel selbst.

Binärdatei mit Trustlet-ID in Klammern	Bezeichnung	Richtlinienoptionen
LsaIso.exe (1)	Credential and Key Guard Trustlet	Allow ETW, Disable Debugging, Allow Encrypted Crash Dump
Vmsp.exe (2)	Secure Virtual Machine Worker (vTPM Trustlet)	Allow ETW, Disable Debugging, Disable Crash Dump, Enable Secure Storage Capability, Verify Parent Security Descriptor is S-1-5-83-0 (NT VIRTUAL MACHINE\Virtual Machines)
Unbekannt (3)	vTPM Key Enrollment Trustlet	Unbekannt
BioIso.exe (4)	Secure Biometrics Trustlet	Allow ETW, Disable Debugging, Allow Encrypted Crash Dump
FsIso.exe (5)	Secure Frame Server Trustlet	Disable ETW, Allow Debugging, Enable Create Secure Section Capability, Use Scenario ID { AE53FC6E-8D89-4488-9D2E-4D008731C5FD}

Tabelle 3–6 Integrierte Trustlets

Trustlet-Identität

Trustlets verfügen über verschiedene Identitätsangaben:

- **Trustlet-ID** Dies ist ein Integerwert, der in den Richtlinien-Metadaten des Trustlets hartcodiert ist und in den Prozesserstellungsattributen für das Trustlet verwendet werden muss. Dadurch weiß das System, dass es nur eine Handvoll Trustlets gibt. Anhand der ID wird auch sichergestellt, dass Aufrufer tatsächlich das gewünschte Trustlet starten.
- **Trustlet-Instanz** Hierbei handelt es sich um eine kryptografisch sichere 16-Byte-Zufallszahl, die vom sicheren Kernel erstellt wird. Gibt es keine Kollaborations-ID, so wird die Trustlet-Instanz herangezogen, um dafür zu sorgen, dass Secure-Storage-APIs nur dieser einen Instanz des Trustlets erlauben, Daten in ihr Storage-BLOB zu stellen und von dort abzurufen.
- **Kollaborations-ID** Diese ID wird verwendet, wenn ein Trustlet anderen Trustlets mit derselben ID oder anderen Instanzen desselben Trustlets den gemeinsamen Zugriff auf ein Secure-Storage-BLOB erlauben möchte. Ist diese ID vorhanden, wird die Instanz-ID beim Aufruf der Get- und Put-APIs ignoriert.
- **Sicherheitsversion (SVN)** Sie wird für Trustlets verwendet, die einen starken kryptografischen Beweis für den Ursprung signierter oder verschlüsselter Daten verlangen, etwa bei der Verschlüsselung von AES256/GCM-Daten durch Credential- und KeyGuard und für den Dienst Cryptograph Report.

- **Szenario-ID** Sie wird für Trustlets verwendet, die benannte sichere Kernelobjekte erstellen, z. B. sichere Abschnitte. Diese Objekte werden im Namensraum mit diesem GUID gekennzeichnet, um zu bestätigen, dass das Trustlet sie im Rahmen eines vorherbestimmten Szenarios erstellt hat. Andere Trustlets, die diese benannten Objekte öffnen wollen, müssen über dieselbe Szenario-ID verfügen. Es ist zwar möglich, dass mehr als eine Szenario-ID vorhanden ist, doch zurzeit verwenden Trustlets nicht mehr als eine.

IUM-Dienste

Die Ausführung als Trustlet bietet nicht nur den Vorteil des Schutzes gegen Angriffe aus der normalen VLT-0-Umgebung, sondern bietet auch Zugriff auf privilegierte und geschützte sichere Systemaufrufe, die der sichere Kernel nur für Trustlets bereitstellt. Dazu gehören die folgenden Dienste:

- **Sichere Geräte (IumCreateSecureDevice, IumDmaMapMemory, IumGetDmaEnabler, IumMapSecureIo, IumProtectSecureIo, IumQuerySecureDeviceInformation, IopUnmapSecureIo, IumUpdateSecureDeviceState)** Sie bieten Zugriff auf sichere ACPI- und PCI-Geräte, die nicht von VTL 0 aus zugänglich sind und exklusiv dem sicheren Kernel (und seinen Hilfsdiensten für die sichere HAL und das sichere PCI) gehören. Trustlets mit entsprechenden Fähigkeiten (siehe den Abschnitt »*Richtlinien-Metadaten für Trustlets*« weiter vorn in diesem Kapitel) können die Register solcher Geräte auf den VTL-1-IUM abbilden und auch Übertragungen mit direktem Speicherzugriff vornehmen (Direct Memory Access, DMA). Außerdem können sie als Benutzermodus-Gerätetreiber für solche Hardware dienen, indem sie das Framework für sichere Geräte (Secure Device Framework, SDF) aus *SDFHost.dll* verwenden. Diese Funktionalität wird für biometrische Verfahren in Windows Hello verwendet, z. B. für die Verwendung von USB-Smartcards (über PCI) und von Webcams und Fingerabdrucksensoren (über ACPI).
- **Sichere Abschnitte (IumCreateSecureSection, IumFlushSecureSectionBuffers, IumGetExposedSecureSection, IumOpenSecureSection)** Über sichere Abschnitte ist es möglich, physische Seiten gemeinsam mit einem VTL-0-Treiber zu nutzen (der dazu `VslCreateSecureSection` verwenden muss). Es können auch Daten als benannte sichere Abschnitte innerhalb von VTL 1 mit anderen Trustlets oder anderen Instanzen desselben Trustlets geteilt werden (wobei der zuvor im Abschnitt »*Trustlet-Identität*« erwähnte Identitätsmechanismus eingesetzt wird). Um dies tun zu können, müssen die Trustlets über die im Abschnitt »*Richtlinien-Metadaten für Trustlets*« erwähnte Fähigkeit *Secure Sections* verfügen.
- **Postfächer (IumPostMailbox)** Damit kann sich ein Trustlet bis zu acht Slots von maximal ca. 4 KB Daten mit einer Komponente im normalen VLT-0-Kernel teilen, der dazu `VslRetrieveMailbox` aufruft und die Slot-ID und den geheimen Postfachschlüssel angibt. Dies wird unter anderem von *Vid.sys* in VTL 0 verwendet, um verschiedene Geheimnisse abzurufen, die vTPM im Trustlet *Vmsp.exe* nutzt.

- **Identitätsschlüssel (IumGetIdk)** Damit kann das Trustlet einen eindeutig kennzeichnenden Entschlüsselungsschlüssel oder einen Signierschlüssel abrufen. Das Schlüsselmaterial bezeichnet eindeutig den Computer und kann nur von einem Trustlet bezogen werden. Dies ist ein entscheidender Aspekt von CredentialGuard, um den Computer zu authentifizieren und zu bestätigen, dass die Anmeldeinformationen tatsächlich aus dem IUM kommen.
- **Kryptografiedienste (IumCrypto)** Sie erlauben einem Trustlet, Daten mit einem lokalen oder Bootsitzungsschlüssel zu verschlüsseln und zu entschlüsseln. Dieser Schlüssel wird vom sicheren Kernel erstellt und steht nur im IUM zur Verfügung. Des Weiteren kann ein Trustlet damit ein TPM-Bindungshandle erlangen; den FIPS-Modus des sicheren Kernels abrufen; einen Seed des Zufallszahlengenerators gewinnen, der ausschließlich vom sicheren Kernel für IUM generiert wird; einen Bericht mit IDK-Signatur, SHA-2-Hash, Zeitstempel und der Identität und SVN des Trustlets anlegen; eine Abbilddatei seiner Richtlinien-Metadaten unabhängig davon erstellen, ob sie jemals mit einem Debugger verknüpft wurden oder nicht; und jegliche andere angeforderten Daten generieren, die seiner Kontrolle unterliegen. Das lässt sich als eine Art von TPM-Maßnahme nutzen, mit der das Trustlet beweisen kann, dass es nicht manipuliert worden ist.
- **Sicherer Speicher (IumSecureStorageGet, IumSecureStoragePut)** Dadurch bekommt das Trustlet die Fähigkeit *Secure Storage* (wie zuvor im Abschnitt »*Richtlinien-Metadaten für Trustlets*« beschrieben), um BLOBs willkürlicher Größe zu speichern und abzurufen. Der Zugriff auf das BLOB kann auf die jeweilige Trustlet-Instanz beschränkt sein oder anderen Trustlets mit derselben Kollaborations-ID offenstehen.

Für Trustlets zugängliche Systemaufrufe

In dem Bemühen, seine Angriffsfläche zu reduzieren, stellt der sichere Kernel nur eine Teilmenge von weniger als fünfzig der Hunderte von Systemaufrufen zur Verfügung, die eine normale VLT-0-Anwendung nutzen kann. Diese Aufrufe bilden das absolute Minimum für eine Kompatibilität mit den DLLs, die von Trustlets verwendet werden können (welche das sind, erfahren Sie im Abschnitt »*Der Aufbau von Trustlets*«), und den Diensten, die zur Unterstützung der RPC-Laufzeit (*Rpcrt4.dll*) und der ETW-Verfolgung erforderlich sind.

- **Worker-Factory- und Thread-APIs** Sie unterstützen die von RPCs verwendete Threadpool-API und die vom Lader verwendeten TLS-Slots.
- **Prozessinformations-API** Sie unterstützt TLS-Slots und die Threadstackzuweisung.
- **Ereignis-, Semaphoren-, Warte- und Vervollständigungs-APIs** Sie unterstützten Threadpools und Synchronisierung.
- **ALPC-APIs (Advanced Local Procedure Calls)** Sie unterstützen lokale RPCs über den Transport `ncalrpc`.
- **Systeminformations-API** Sie ermöglicht das Lesen von Informationen über den sicheren Start, grundlegenden und NUMA-Systeminformationen für *Kernel32.dll* und die Threadpoolskalierung, Leistungsinformationen und einigen Zeitinformationen.
- **Token-API** Sie bietet minimale Unterstützung für RPC-Identitätswechsel.

- **APIs für virtuelle Speicherzuweisung** Sie unterstützen Zuweisungen durch den Heap-Manager des Benutzermodus.
- **Abschnitts-APIs** Sie unterstützen den Lader (für DLL-Abbilder) und die Funktionalität sicherer Abschnitte (sobald sie durch die zuvor angegebenen sicheren Systemaufrufe erstellt bzw. verfügbar gemacht worden sind).
- **Steuer-API für die Ablaufverfolgung** Sie unterstützt ETW.
- **Ausnahme- und Fortsetzungs-API** Sie unterstützt die strukturierte Ereignisbehandlung (Structured Exception Handling, SEH).

Diese Liste zeigt deutlich, dass es keine Unterstützung für Operationen wie die folgenden gibt: Geräte-E/A, unabhängig davon, ob es sich um Dateien oder physische Geräte handelt (es gibt schon einmal keine `CreateFile`-API); Registrierungs-E/A; Erstellen anderer Prozesse; jegliche Verwendung von Grafik-APIs (es gibt keinen *Win32k.sys*-Treiber in VTL 1). Trustlets sind als isolierte Arbeits-Back-Ends in VTL 1 für ihre komplexen Front-Ends in VTL 0 gedacht. Als einzige Kommunikationsmechanismen stehen ihnen ALPC und bereitgestellte sichere Abschnitte zur Verfügung (wobei ihnen die Handles dieser sicheren Abschnitte über ALPC vermittelt werden müssen). In Kapitel 7 schauen wir uns die Implementierung des Trustlets *LsaIso.exe* genauer an, das für CredentialGuard und KeyGuard zuständig ist.

Experiment: Sichere Prozesse identifizieren

Abgesehen von ihrem Namen können Sie sichere Prozesse im Kerneldebugger auf zwei verschiedene Weisen erkennen. Erstens verfügt jeder sichere Prozess über eine sichere PID, die in der Handletabelle des sicheren Kernels für sein Handle steht. Sie wird vom normalen VLT-0-Kernel verwendet, wenn er Threads in dem Prozess erstellt oder dessen Beendigung anfordert. Zweitens ist mit den Threads ein Threadcookie verknüpft, das ihren Index in der Threadtabelle des sicheren Kernels angibt.

Probieren Sie Folgendes im Kerneldebugger aus:

```
lkd> !for_each_process .if @@(((nt!_EPROCESS*)${@#Process})->Pcb.SecurePid) {
.printf "Trustlet: %ma (%p)\n", @@(((nt!_EPROCESS*)${@#Process})
   ->ImageFileName), @#Process }
Trustlet: Secure System (ffff9b09d8c79080)
Trustlet: LsaIso.exe (ffff9b09e2ba9640)
Trustlet: BioIso.exe (ffff9b09e61c4640)
lkd> dt nt!_EPROCESS ffff9b09d8c79080 Pcb.SecurePid
   +0x000 Pcb :
      +0x2d0 SecurePid : 0x00000001'40000004
lkd> dt nt!_EPROCESS ffff9b09e2ba9640 Pcb.SecurePid
   +0x000 Pcb :
      +0x2d0 SecurePid : 0x00000001'40000030
```

→

```
lkd> dt nt!_EPROCESS ffff9b09e61c4640 Pcb.SecurePid
   +0x000 Pcb :
      +0x2d0 SecurePid : 0x00000001'40000080
lkd> !process ffff9b09e2ba9640 4
PROCESS ffff9b09e2ba9640
   SessionId: 0 Cid: 0388 Peb: 6cdc62b000 ParentCid: 0328
   DirBase: 2f254000 ObjectTable: ffffc607b59b1040 HandleCount: 44.
   Image: LsaIso.exe
      THREAD ffff9b09e2ba2080 Cid 0388.038c Teb: 0000006cdc62c000 Win32Thread:
0000000000000000 WAIT
lkd> dt nt!_ETHREAD ffff9b09e2ba2080 Tcb.SecureThreadCookie
   +0x000 Tcb
      +0x31c SecureThreadCookie : 9
```

Der Ablauf von CreateProcess

Wir haben uns bereits die verschiedenen Datenstrukturen für die Bearbeitung und Verwaltung des Prozesszustands und die Werkzeuge und Debuggerbefehle angesehen, mit denen sie sich untersuchen lassen. In diesem Abschnitt beschäftigen wir uns damit, wie diese Datenstrukturen erstellt und ausgefüllt und wie Prozesse erstellt und beendet werden. Wie bereits erwähnt, rufen alle Funktionen zur Prozesserstellung letzten Endes `CreateProcessInternalW` auf, weshalb wir damit beginnen.

Um einen Windows-Prozess zu erstellen, werden mehrere Schritte in drei Teilen des Betriebssystems ausgeführt: in der clientseitigen Windows-Bibliothek *Kernel32.dll* (die eigentliche Arbeit beginnt mit `CreateProcessInternalW`), in der Windows-Exekutive und im Windows-Teilsystemprozess (*Csrss*). Aufgrund der Teilsystemarchitektur für mehrere Umgebungen wird die Erzeugung eines Exekutivprozessobjekts (das auch andere Teilsysteme nutzen können) von den Arbeiten zum Erstellen eines Windows-Teilsystemprozesses getrennt. Die folgende Beschreibung des Ablaufs der Windows-Funktion `CreateProcess` ist zwar kompliziert, doch denken Sie daran, dass dieser Teil der Arbeit nur für die zusätzliche Semantik des Windows-Teilsystems erforderlich ist und nicht zu der grundlegenden Arbeit zum Erstellen eines Exekutivprozessobjekts gehört.

Die folgende Übersicht zeigt die Hauptphasen beim Erstellen eines Prozesses mit den `CreateProcess*`-Funktionen von Windows. Die in den jeweiligen Phasen ausgeführten Operationen werden in den anschließenden Abschnitten ausführlich erläutert.

Viele Schritte von `CreateProcess` hängen mit der Einrichtung des virtuellen Adressraums zusammen, weshalb dabei viele Begriffe und Strukturen der Speicherverwaltung erwähnt werden, deren Definition Sie in Kapitel 5 finden.

1. Parameter validieren; Windows-Teilsystemflags und -optionen in ihre nativen Gegenstücke umwandeln; Attributliste analysieren, validieren und in ihr natives Gegenstück umwandeln.
2. Die Abbilddatei (*.exe*) öffnen, die in dem Prozess ausgeführt werden soll.
3. Das Windows-Exekutivprozessobjekt erstellen.
4. Den ursprünglichen Thread erstellen (Stack, Kontext und Windows-Exekutivthreadobjekt).
5. Spezifische Prozessinitialisierung für das Windows-Teilsystem durchführen.
6. Ausführung des ursprünglichen Threads starten (sofern nicht das Flag `CREATE_SUSPENDED` angegeben war).
7. Initialisierung des Adressraums im Kontext des neuen Prozesses und Threads abschließen (z. B. durch Laden der erforderlichen DLLs) und die Ausführung des Programmeintrittspunkts beginnen.

Abb. 3–10 gibt einen Überblick über die Phasen, die Windows beim Erstellen eines Prozesses durchläuft.

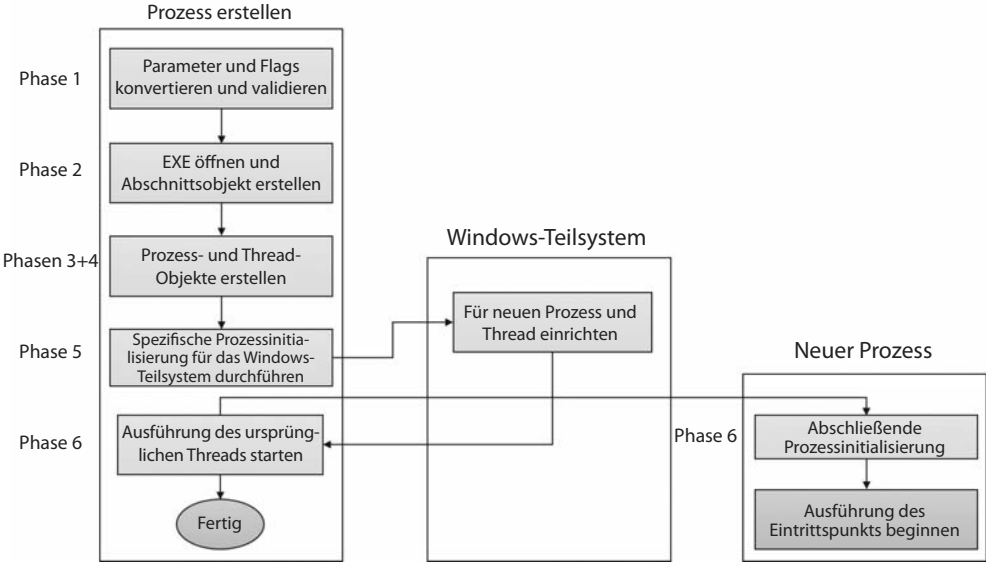


Abbildung 3–10 Die Hauptphasen der Prozesserstellung

Phase 1: Parameter und Flags konvertieren und validieren

Bevor die Funktion `CreateProcessInternalW` das auszuführende Image öffnet, führt sie folgende Schritte aus:

1. Die Prioritätsklasse für den neuen Prozess wird in Form unabhängiger Bits im Parameter `CreationFlags` der `CreateProcess*`-Funktionen angegeben. Daher ist es möglich, bei einem `CreateProcess*`-Aufruf mehrere Prioritätsklassen anzugeben. Windows weist dem Prozess die Klasse mit der niedrigsten Priorität zu.

Es sind sechs Prioritätsklassen definiert, denen jeweils eine Zahl zugeordnet ist:

- Leerlauf oder niedrig (4)
- Niedriger als normal (6)
- Normal (8)
- Höher als normal (10)
- Hoch (13)
- Echtzeit (24)

Die Prioritätsklasse wird als Basispriorität für die in dem Prozess erstellten Threads verwendet. Dieser Wert wirkt sich nicht direkt auf den Thread aus, sondern nur auf die enthaltenen Threads. Eine Erläuterung der Prioritätsklassen und ihrer Auswirkung auf die Threadplanung erhalten Sie in Kapitel 4.

2. Wurde für den neuen Prozess keine Prioritätsklasse angegeben, wird *Normal* verwendet. Wurde die Prioritätsklasse *Echtzeit* angegeben und hat der Aufrufer des Prozesses nicht das Recht *Anheben der Zeitplanungspriorität* (`SE_INC_BASE_PRIORITY`), dann wird stattdessen die Klasse *Hoch* verwendet. Die Prozesserstellung schlägt also nicht fehl, nur weil der Aufrufer unzureichende Berechtigungen hat, um einen Prozess mit Echtzeitpriorität anzulegen. Stattdessen erhält der neue Prozess einfach eine niedrigere Priorität.
3. Verlangen die Erstellungsflags ein Debugging des Prozesses, so stellt *Kernel32* eine Verbindung zum nativen Debuggingcode in *Ntdll.dll* her, indem er `DbgUiConnectToDbg` aufruft und ein Handle für das Debugobjekt vom aktuellen Threadumgebungsblock (TEB) abruft.
4. *Kernel32.dll* richtet den standardmäßigen harten Fehlermodus ein, wenn dies in den Erstellungsflags angegeben ist.
5. Die vom Benutzer definierte Attributliste wird vom Format des Windows-Teilsystems in das native Format umgewandelt und um interne Attribute ergänzt. Die möglichen zusätzlichen Attribute finden Sie in Tabelle 3–7 zusammen mit ihren dokumentierten Windows-API-Gegenstücken, falls vorhanden.

Die an `CreateProcess*`-Aufrufe übergebene Attributliste ermöglicht es, an den Aufrufer Informationen zurückzugeben, die über einen einfachen Statuscode hinausgehen, z. B. die TEB-Adresse des ursprünglichen Threads oder Informationen über den Abbildabschnitt. Das ist für geschützte Prozesse notwendig, da der Elternprozess diese Information nach dem Erstellen des Kindprozesses nicht abfragen kann.

Natives Attribut	Entsprechendes W32-Attribut	Typ	Beschreibung
`PS_CP_PARENT_PROCESS`	`PROC_THREAD_ATTRIBUTE_PARENT_PROCESS`. Auch bei Rechteerhöhung verwendet.	Eingabe	Handle für den Elternprozess
`PS_CP_DEBUG_OBJECT`	Nicht verfügbar. Wird bei der Verwendung von `DEBUG_PROCESS` als Flag genutzt.	Eingabe	Debugobjekt, wenn der Prozess im Debuggingmodus gestartet wird

→

Natives Attribut	Entsprechendes W32-Attribut	Typ	Beschreibung
PS_CP_PRIMARY_TOKEN	Nicht verfügbar. Wird bei der Verwendung von CreateProcessAsUser/WithTokenW genutzt.	Eingabe	Prozesstoken, wenn CreateProcessAsUser verwendet wurde
PS_CP_CLIENT_ID	Nicht verfügbar. Wird von der Win32-API als Parameter zurückgegeben (PROCESS_INFORMATION).	Ausgabe	Gibt die TID und PID des ursprünglichen Threads bzw. Prozesses zurück.
PS_CP_TEB_ADDRESS	Nicht verfügbar. Intern verwendet und nicht verfügbar gemacht.	Ausgabe	Gibt die Adresse des TEB für den ursprünglichen Thread zurück.
PS_CP_FILENAME	Nicht verfügbar. Wird als Parameter in CreateProcess-APIs verwendet.	Eingabe	Der Name des zu erstellenden Prozesses
PS_CP_IMAGE_INFO	Nicht verfügbar. Intern verwendet und nicht verfügbar gemacht.	Ausgabe	Gibt SECTION_IMAGE_INFORMATION mit Informationen über die Version, die Flags und das Teilsystem der ausführbaren Datei sowie die Stackgröße und den Eintrittspunkt zurück.
PS_CP_MEM_RESERVE	Nicht verfügbar. Intern von SMSS und CSRSS verwendet.	Eingabe	Ein Array von Reservierungen für virtuellen Speicher, die beim ursprünglichen Erstellen des Prozessadressraums durchgeführt werden sollen. Die Verfügbarkeit wird garantiert, da noch keine anderen Zuweisungen stattgefunden haben.
PS_CP_PRIORITY_CLASS	Nicht verfügbar. Wird als Parameter an die CreateProcess-API übergeben.	Eingabe	Die Prioritätsklasse, die der Prozess haben soll
PS_CP_ERROR_MODE	Nicht verfügbar. Wird über das Flag CREATE_DEFAULT_ERROR_MODE übergeben.	Eingabe	Verarbeitungsmodus des Prozesses für harte Fehler
PS_CP_STD_HANDLE_INFO	Keines. Wird intern verwendet.	Eingabe	Gibt an, ob Standardhandles dupliziert oder ob neue Handles erstellt werden sollen.
PS_CP_HANDLE_LIST	PROC_THREAD_ATTRIBUTE_HANDLE_LIST	Eingabe	Die Handles des Elternprozesses, die der neue Prozess erben soll
PS_CP_GROUP_AFFINITY	PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY	Eingabe	Die Prozessorgruppen, auf denen der Thread laufen darf
PS_CP_PREFERRED_NODE	PROC_THREAD_ATTRIBUTES_PREFERRED_NODE	Eingabe	Der bevorzugte (ideale) NUMA-Knoten, der mit dem Prozess verknüpft werden soll. Dies legt den Knoten fest, auf dem der ursprüngliche Prozessheap und der ursprüngliche Threadstack erstellt werden (siehe Kapitel 5).

→

Natives Attribut	Entsprechendes W32-Attribut	Typ	Beschreibung
PS_CP_IDEAL_PROCESSOR	PROC_THREAD_ATTTRIBUTE_IDEAL_PROCESSOR	Eingabe	Der bevorzugte (ideale) Prozessor, auf dem der Thread eingeplant werden soll
PS_CP_UMS_THREAD	PROC_THREAD_ATTRIBUTE_UMS_THREAD	Eingabe	Enthält die UMS-Attribute, die Vervollständigungsliste und den Kontext.
PS_CP_MITIGATION_OPTIONS	PROC_THREAD_MITIGATION_POLICY	Eingabe	Enthält Informationen über die vorbeugenden Maßnahmen (SEHOP, ATL-Emulation, NX), die für den Prozess aktiviert oder deaktiviert werden sollen.
PS_CP_PROTECTION_LEVEL	PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL	Eingabe	Muss auf einen der zulässigen Prozessschutzwerte aus Tabelle 3–1 oder den Wert PROTECT_LEVEL_SAME zeigen. In letzterem Fall erhält der Prozess dieselbe Schutzebene wie sein Elternprozess.
PS_CP_SECURE_PROCESS	Keines. Wird intern verwendet.	Eingabe	Gibt an, dass der Prozess als IUM-Trustlet (Isolated User Mode) ausgeführt werden soll (siehe Kapitel 8 in Band 2).
PS_CP_JOB_LIST	Keines. Wird intern verwendet.	Eingabe	Weist den Prozess einer Jobliste zu.
PS_CP_CHILD_PROCESS_POLICY	PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY	Eingabe	Gibt an, ob der neue Prozess direkt oder indirekt Kindprozesse erstellen darf (z. B. durch WMI).
PS_CP_ALL_APPLICATION_PACKAGES_POLICY	PROC_THREAD_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY	Eingabe	Gibt an, ob das AppContainer-Token von Überprüfungen von Zugriffssteuerungslisten ausgenommen werden soll, die die Gruppe ALL APPLICATION PACKAGES enthalten. Stattdessen wird die Gruppe ALL RESTRICTED APPLICATION PACKAGES verwendet.
PS_CP_WIN32K_FILTER	PROC_THREAD_ATTRIBUTE_WIN32K_FILTER	Eingabe	Gibt an, ob viele GDI/USER-Systemaufrufe des Prozesses an *Win32k.sys* gefiltert (blockiert) oder ob sie zugelassen, aber überwacht werden sollen. Wird vom Browser Microsoft Edge verwendet, um die Angriffsfläche zu verringern.
PS_CP_SAFE_OPEN_PROMPT_ORIGIN_CLAIM	Keines. Wird intern verwendet.	Eingabe	Wird von »Mark of the Web« verwendet, um anzuzeigen, dass die Datei aus einer nicht vertrauenswürdigen Quelle stammt.

→

Natives Attribut	Entsprechendes W32-Attribut	Typ	Beschreibung
PS_CP_BNO_ISOLATION	PROC_THREAD_ATTRIBUTE_BNO_ISOLATION	Eingabe	Sorgt dafür, dass das Primärtoken des Prozesses mit einem isolierten BaseNamedObjects-Verzeichnis verknüpft wird (siehe Kapitel 8 in Band 2).
PS_CP_DESKTOP_APP_POLICY	PROC_THREAD_ATTRIBUTE_DESKTOP_APP_POLICY	Eingabe	Gibt an, ob die moderne Anwendung ältere Desktopanwendungen starten darf und wenn ja, in welcher Weise.
PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES	Keines. Wird intern verwendet.	Eingabe	Gibt einen Zeiger auf eine SECURITY_CAPABILITIES-Struktur an, die dazu dient, ein AppContainer-Token für den Prozess zu erstellen, bevor NtCreateUserProcess aufgerufen wird.

Tabelle 3–7 Prozessattribute

6. Fordert das Erstellungsflag eine getrennte virtuelle DOS-Maschine (VDM) an, obwohl der Prozess Teil eines Jobobjekts ist, so wird das Flag ignoriert.
7. Die an die CreateProcess-Funktion übergebenen Sicherheitsattribute für den Prozess und den ursprünglichen Thread werden in ihre interne Darstellung umgewandelt (OBJECT_ATTRIBUTES-Strukturen; dokumentiert im WDK).
8. CreateProcessInternalW prüft, ob der Prozess als moderner Prozess erstellt werden soll. Das ist der Fall, wenn es in dem Attribut PROC_THREAD_ATTRIBUTE_PACKAGE_FULL_NAME mit dem vollständigen Paketnamen so verlangt wird oder wenn der Ersteller selbst ein moderner Prozess ist (und mit dem Attribut PROC_THREAD_ATTRIBUTE_PARENT_PROCESS nicht ausdrücklich ein Elternprozess angegeben wurde). In diesem Fall wird die interne Funktion BasepAppXExtension aufgerufen, um mehr Kontextinformationen über die Parameter der modernen App zu ermitteln. Sie befinden sich in der Struktur APPX_PROCESS_CONTEXT, die Informationen wie den Paketnamen (intern als »package moniker« bezeichnet), die Fähigkeiten der Anwendung, das aktuelle Verzeichnis für den Prozess und die Angabe enthält, ob der App volles Vertrauen geschenkt werden soll. Die Möglichkeit, eine voll vertrauenswürdige moderne App zu erstellen, ist nicht öffentlich verfügbar, sondern für Anwendungen reserviert, die zwar über ein modernes Erscheinungsbild verfügen, aber Operationen auf Systemebene durchführen. Ein wichtiges Beispiel ist die App *Einstellungen* in Windows 10 (*SystemSettings.exe*).
9. Wird ein moderner Prozess erstellt, so wird die interne Funktion BasepCreateLowBox aufgerufen, um die Sicherheitsfähigkeiten (sofern in PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES bereitgestellt) für die Erstellung des ursprünglichen Tokens aufzuzeichnen. Die Bezeichnung LowBox bezieht sich auf die Sandbox (den Anwendungscontainer), in der der Prozess ausgeführt wird. Es ist zwar nicht möglich, moderne Prozesse direkt durch einen Aufruf von CreateProcess zu erstellen (stattdessen müssen die zuvor beschriebenen COM-Schnittstellen verwendet werden), doch im Windows SDK und auf MSDN wird die

Möglichkeit beschrieben, ältere AppContainer-Desktopanwendungen durch die Übergabe dieses Attributs zu erstellen.

10. Soll ein moderner Prozess erstellt werden, so wird ein Flag gesetzt, um anzuzeigen, dass der Kernel die Erkennung des eingebetteten Manifests überspringen soll. Moderne Prozesse sollten kein eingebettetes Manifest haben, da es nicht benötigt wird. (Eine moderne App hat ihr eigenes Manifest, das aber nichts mit dem hier gemeinten eingebetteten Manifest zu tun hat.)
11. Wurde das Debugflag `DEBUG_PROCESS` angegeben, wird der `Debugger`-Wert im Registrierungsschlüssel `Image File Execution Options` für die ausführbare Datei (siehe den nächsten Abschnitt) so markiert, dass er übersprungen wird. Anderenfalls könnte ein Debugger niemals den zu debuggenden Prozess erstellen, da der Vorgang in eine Endlosschleife eintritt (es wird immer und immer wieder versucht, den Debuggerprozess anzulegen).
12. Alle Fenster werden mit Desktops verknüpft, den grafischen Darstellungen eines Arbeitsbereichs. Ist in der `STARTUPINFO`-Struktur kein Desktop angegeben, so wird der Prozess mit dem aktuellen Desktop des Aufrufers verknüpft.

Die Windows 10-Funktion der virtuellen Desktops verwendet in Wirklichkeit nicht mehrere Desktopobjekte (im Sinne von Kernelobjekten). Es gibt immer noch lediglich einen Desktop, auf dem aber die Fenster nach Bedarf angezeigt bzw. ausgeblendet werden. Anders ist es beim Sysinternals-Tools *Desktops.exe*, das wirklich bis zu vier Desktopobjekte erstellt. Den Unterschied können Sie erkennen, wenn Sie versuchen, ein Fenster von einem Desktop zu einem anderen zu verschieben. Bei *Desktops.exe* geht das nicht, da eine solche Operation in Windows nicht unterstützt wird. In den virtuellen Desktops von Windows 10 ist es jedoch möglich, da dabei in Wirklichkeit gar nichts verschoben wird.

13. Die an `CreateProcessInternalW` übergebene Anwendung und die Befehlszeilenargumente werden analysiert. Der Pfad der ausführbaren Datei wird in den internen NT-Namen umgewandelt (beispielsweise wird aus *c:\temp\a.exe* etwas wie *\device\harddiskvolume1\temp\a.exe*), da einige Funktionen auf dieses Format angewiesen sind.
14. Die meisten erfassten Informationen werden in eine einzige große Struktur vom Typ `RTL_USER_PROCESS_PARAMETERS` umgewandelt.

Nach Abschluss dieser Schritte ruft `CreateProcessInternalW` die Funktion `NtCreateUserProcess` auf, um zu versuchen, den Prozess zu erstellen. Da *Kernel32.dll* zu diesem Zeitpunkt noch nicht weiß, ob der Name des Abbilds zu einer echten Windows-Anwendung oder zu einer Batchdatei (*.bat* oder *.cmd*), einer 16-Bit- oder DOS-Anwendung gehört, kann dieser Aufruf fehlschlagen. In diesem Fall schaut sich `CreateProcessInternalW` den Grund für den Fehler an und versucht, ihn zu korrigieren.