Process Failure Modes

James Forshaw @tiraniddo
Recon 2016

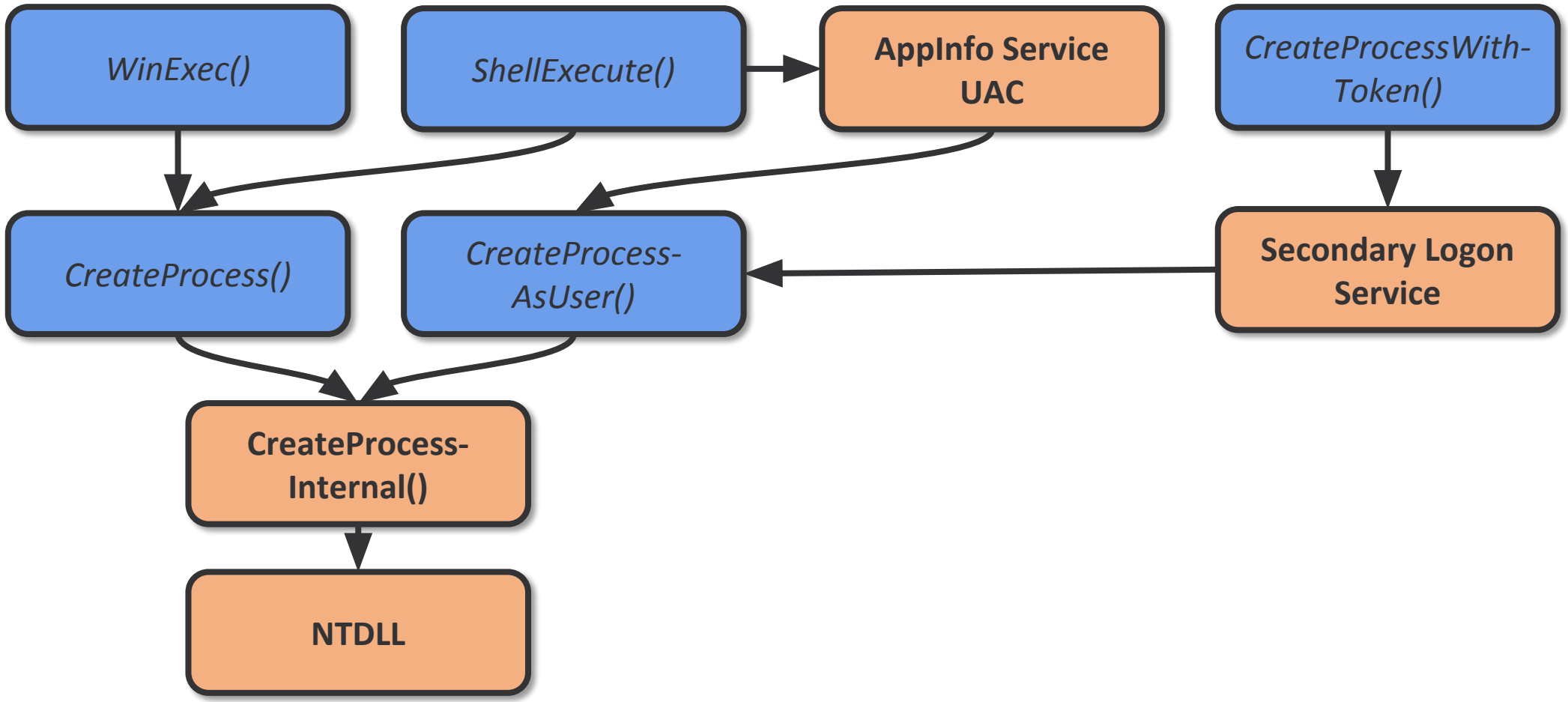
What I’m Going to Talk About

- Dig into the Windows Process Creation APIs and Low-level stuff.
- Classic bugs when creating processes, especially in privileged code
- Some silly tricks to confuse IR
- Based primarily on Windows 10 build 10586
- This’ll be a *C:\Program.exe* Free Zone!

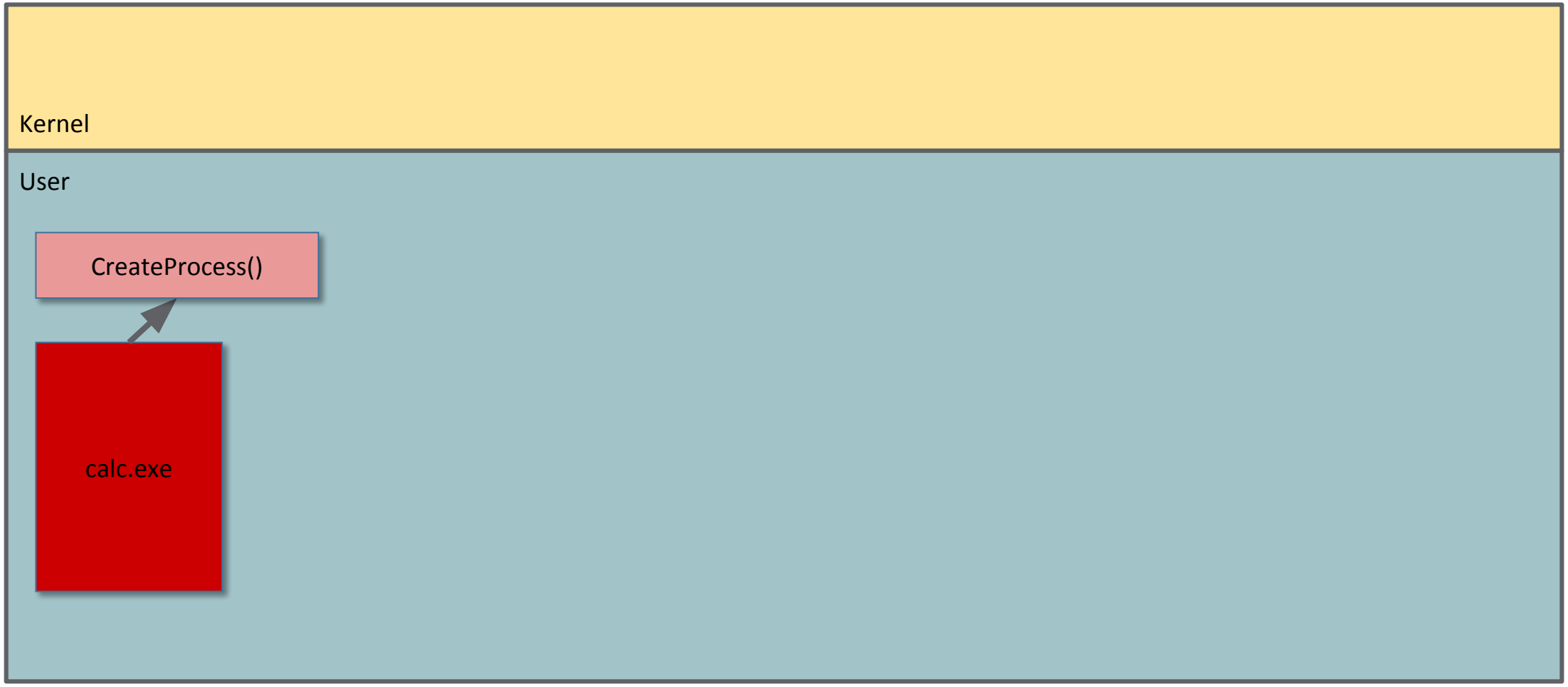
Thanks to Alex Ionescu and mj32 for various useful insights in this research.

Windows Process Creation APIs

Win32 APIs

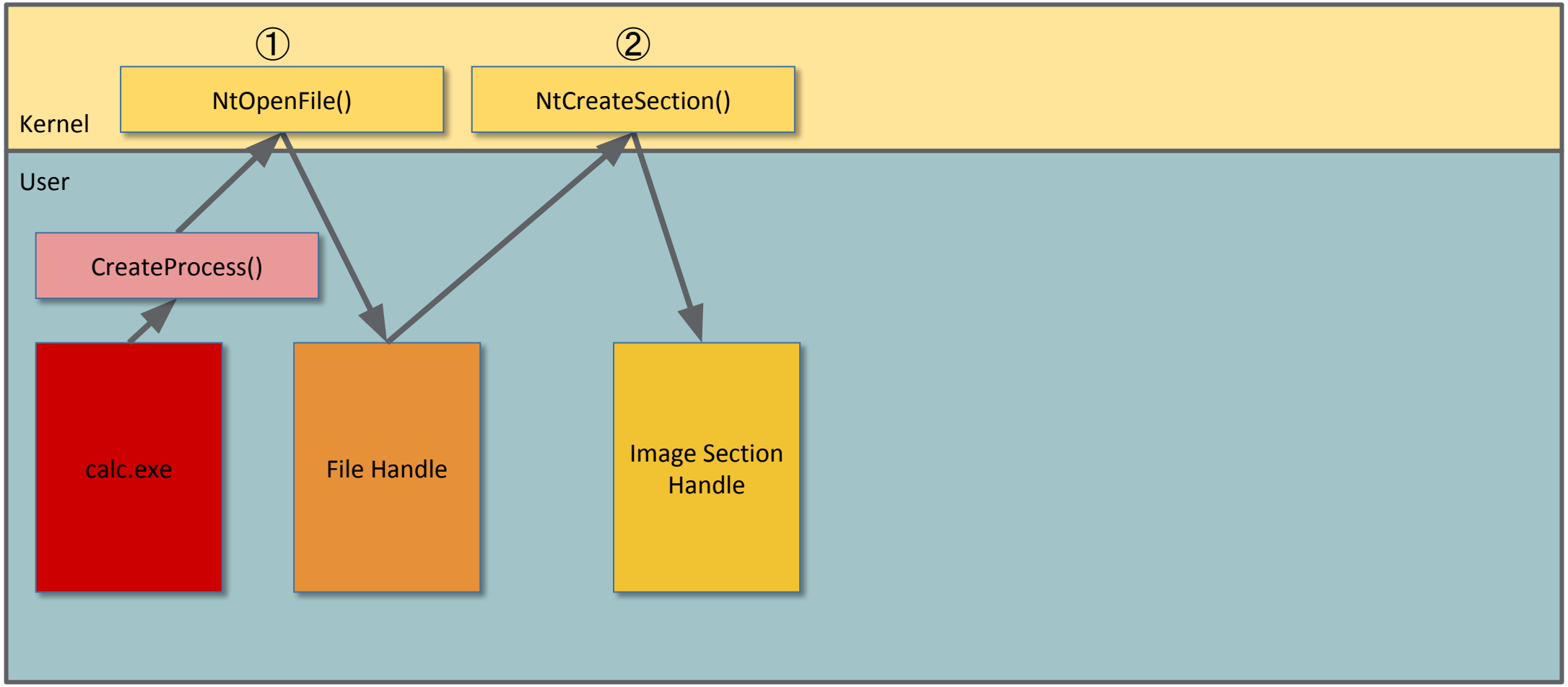

WinExec()
ShellExecute()
AppInfo Service
UAC
CreateProcessWith-
Token()
CreateProcess()
CreateProcess-
AsUser()
Secondary Logon
Service
CreateProcess-
Internal()
NTDLL

In the Beginning Was… *NtCreateProcess*

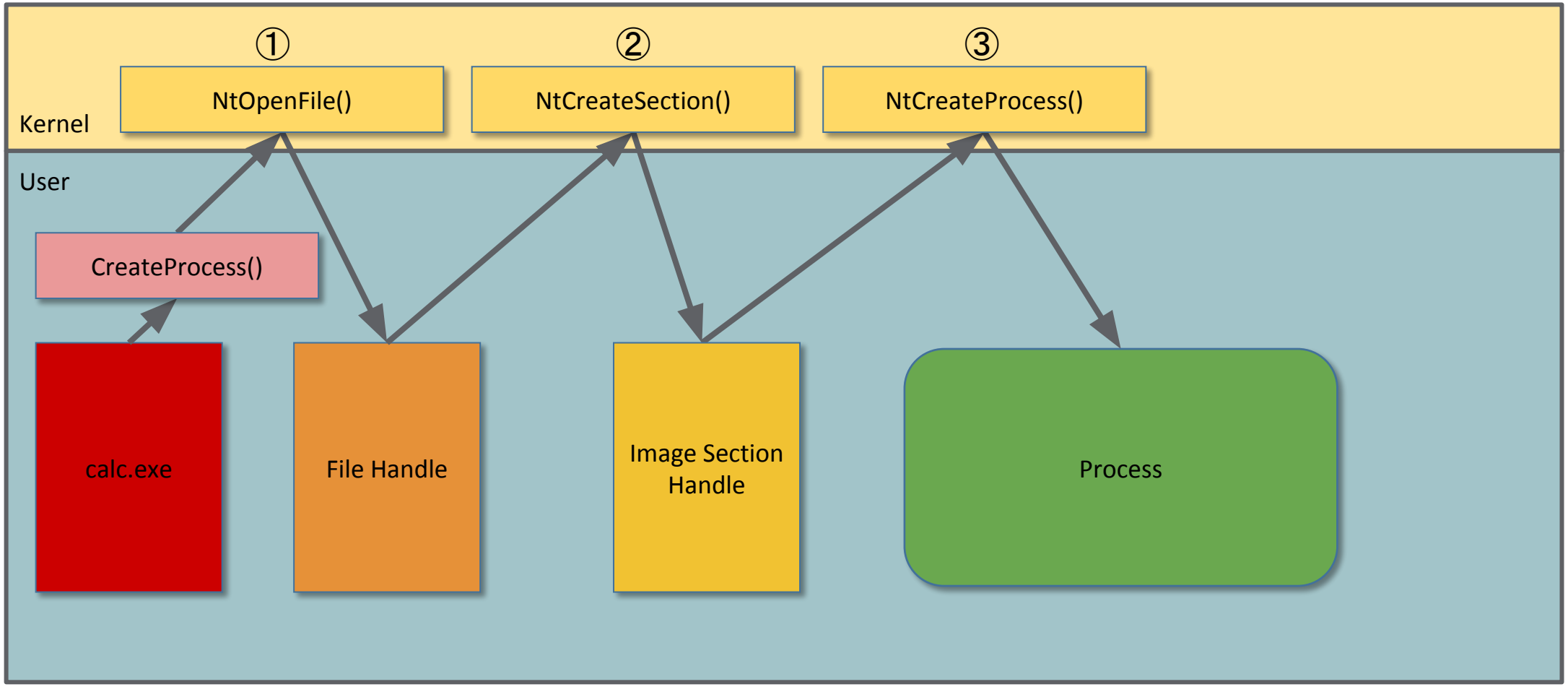


In the Beginning Was… *NtCreateProcess*

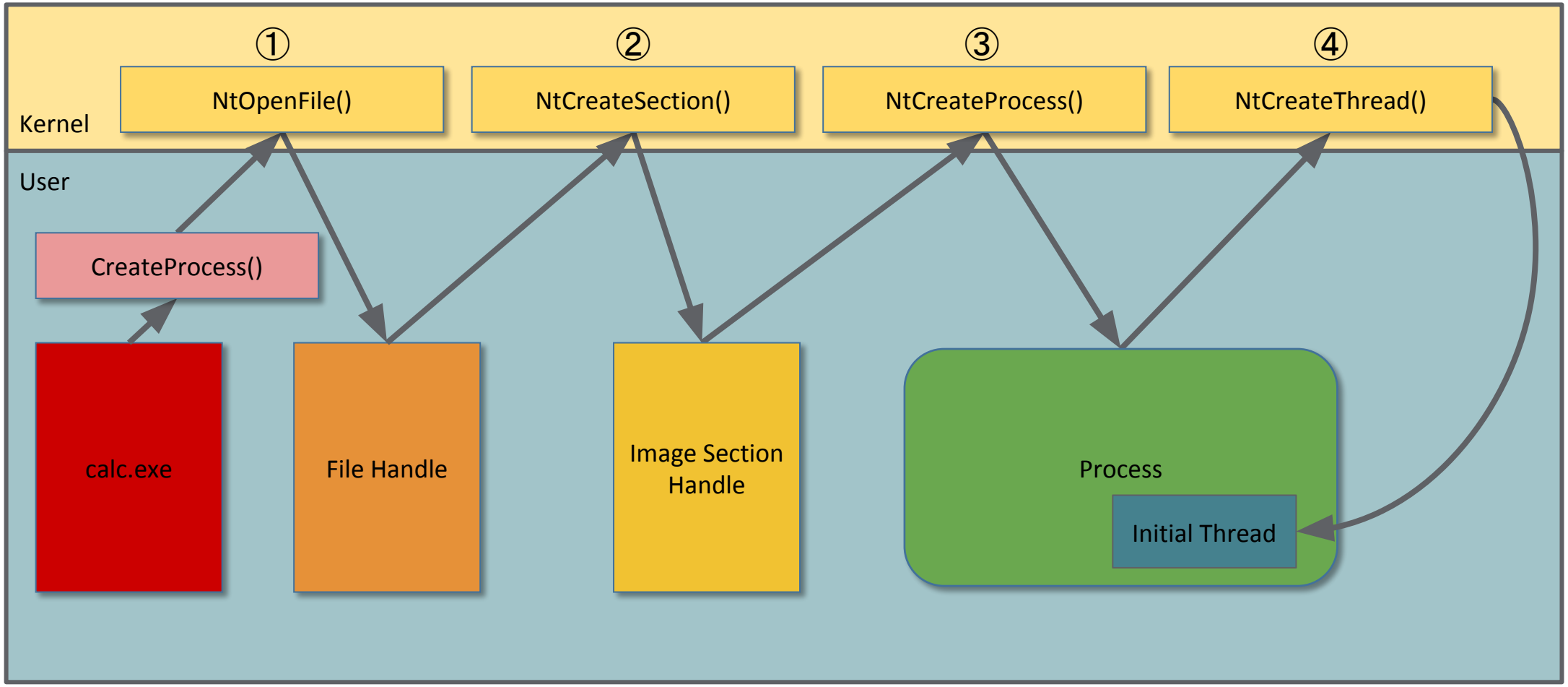
In the Beginning Was… *NtCreateProcess*



In the Beginning Was… *NtCreateProcess*



In the Beginning Was… *NtCreateProcess*



Little Tweaks to make NtCreateProcessEx

```
NTSTATUS NtCreateProcessEx(
_Out_     PHANDLE ProcessHandle,
_In_      ACCESS_MASK DesiredAccess,
_In_opt_  POBJECT_ATTRIBUTES ObjectAttributes,
_In_      HANDLE ParentProcess,
_In_      ULONG Flags,
_In_opt_  HANDLE SectionHandle,
_In_opt_  HANDLE DebugPort,
_In_opt_  HANDLE ExceptionPort,
_In_      ULONG Unused // At least in Win8+
);
```

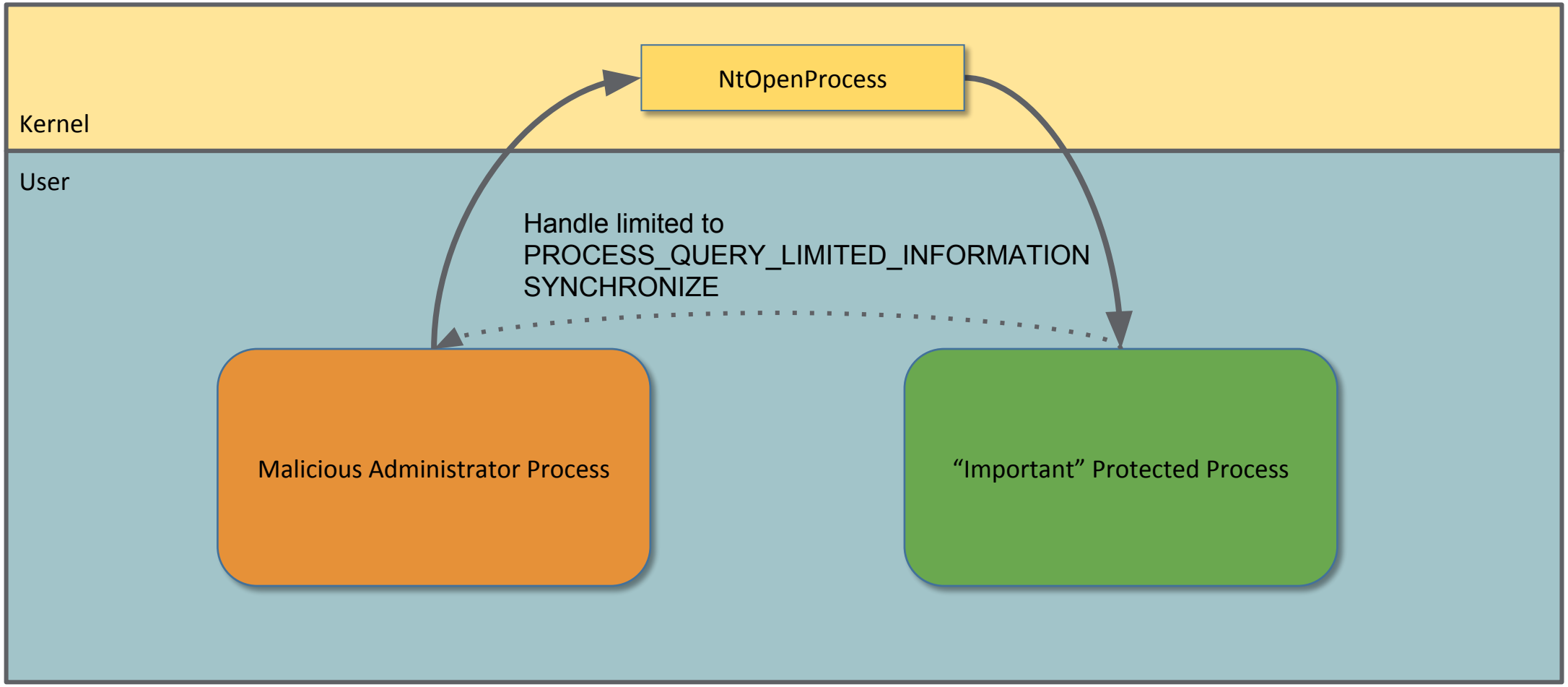
Handle to parent process to inherit from

Assorted Flags

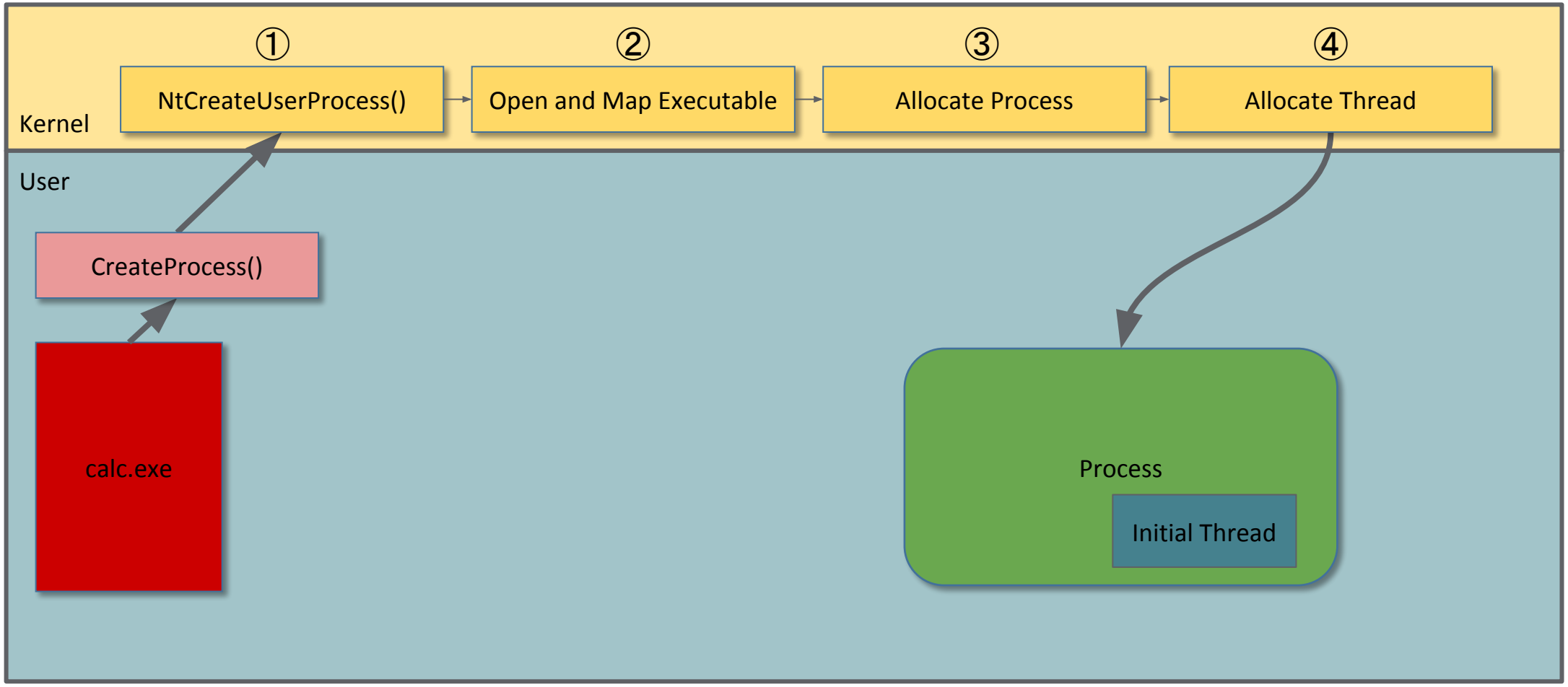
```
BreakawayJob     0x01
NoDebugInherit   0x02
InheritHandles   0x04
```

Optional Image Section Handle

Vista Protected Processes



So Then Came… *NtCreateUserProcess*



NtCreateUserProcess

```
NTSTATUS NtCreateUserProcess(
  _Out_ PHANDLE ProcessHandle,
  _Out_ PHANDLE ThreadHandle,
  _In_ ACCESS_MASK ProcessDesiredAccess,
  _In_ ACCESS_MASK ThreadDesiredAccess,
  _In_opt_ POBJECT_ATTRIBUTES ProcessObjectAttributes,
  _In_opt_ POBJECT_ATTRIBUTES ThreadObjectAttributes,
  _In_ ULONG ProcessFlags,
  _In_ ULONG ThreadFlags,
  _In_opt_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
  _Inout_ PPS_CREATE_INFO CreateInfo,
  _In_opt_ PPS_ATTRIBUTE_LIST AttributeList
);
```

User Process Parameters

```
struct RTL_USER_PROCESS_PARAMETERS
{
    // ...
    HANDLE ConsoleHandle;
    ULONG  ConsoleFlags;
    HANDLE StandardInput;
    HANDLE StandardOutput;
    HANDLE StandardError;
    CURDIR CurrentDirectory;
    UNICODE_STRING DllPath;
    UNICODE_STRING ImagePathName;
    UNICODE_STRING CommandLine;
    PVOID Environment;
    // ....
}
```

```
struct PEB
{
   UCHAR InheritedAddressSpace;
   UCHAR ReadImageFileExecOptions;
   UCHAR BeingDebugged;
   UCHAR BitField;
   // ...
   PPEB_LDR_DATA Ldr;
   PRTL_USER_PROCESS_PARAMETERS
                    ProcessParameters;
   // ...
};
```

Create Info

```
struct PS_CREATE_INFO
{
  SIZE_T Size;
  PS_CREATE_STATE State;
  union {
    PS_INIT_START InitState;
    PS_SUCCESS_STATE SuccessState;
    // ...
  };
};
```

```
struct PS_INIT_STATE
{
  USHORT Flags;
  USHORT ProhibitedImageCharacteristics : 16;
  ACCESS_MASK AdditionalFileAccess;
};
```

```
struct PS_SUCCESS_STATE
{
  UCHAR Flags;
  HANDLE FileHandle;
  HANDLE SectionHandle;
  ULONGLONG UserProcessParametersNative;
  ULONG UserProcessParametersWow64;
  ULONG CurrentParameterFlags;
  ULONGLONG PebAddressNative;
  ULONG PebAddressWow64;
  ULONGLONG ManifestAddress;
  ULONG ManifestSize;
};
```

Process and Thread Attributes

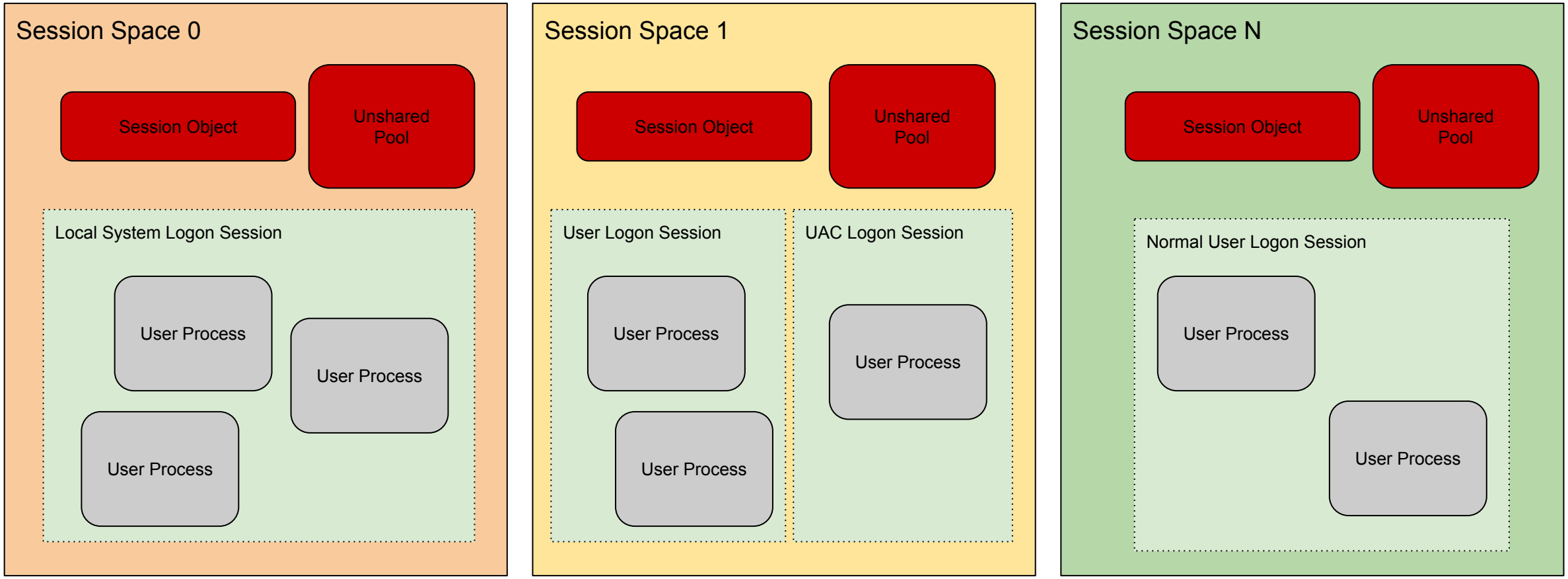
- Kernel uses a list of attributes to extend system call functionality
  - New attributes are added in newer versions of Windows

Attribute	*Data Type*
PsAttributeParentProcess	HANDLE to a process
PsAttributeToken	HANDLE to a token
PsAttributeClientId	Pointer to a CLIENT_ID structure to get PID and TID
PsAttributeImageName	Pointer to a null terminated wide character string
PsAttributeHandleList	Array of HANDLEs to inherit
PsAttributeMitigationOptions	Pointer to a INT64 containing mitigation policy bitmask
PsAttributeProtectionLevel	Pointer to a PS_PROTECTION_LEVEL structure

What has *CreateProcessInternal* Ever Done For Us?

- Registers the process manifest with CSRSS (though not the process itself)
- Check whether the process should be elevated
  - Returns error if process manifest requires elevation
- Handles IFEO *Debugger* settings (well sort of)
- Handles spawning cmd.exe in response to executing a batch file
- Looks up application compatibility settings and applies them to new process
- Startup VDM if running a Win16/DOS application (you'd assume on 32 bit only!)
- Check if executable is blocked by Software Restriction Policy (not AppLocker)

Session Spaces



Setting the Session ID

- Session ID is a field in the process token
- Set using NtSetInformationToken
  - Needs TCB privilege to set session ID
- Can create new sessions by specifying the Create Session process flag
  - Needs SeLoadDriverPrivilege to attach to a new session
- Can also be set without need TCB privilege if you have a handle to a process in a different session.
  - If you have a process and you assign it a new token using NtSetInformationProcess it will set the SessionID of the old primary token
  - If you use the process as the parent of a new process the parent's token will have the Session ID of the caller process set instead

Parent Process

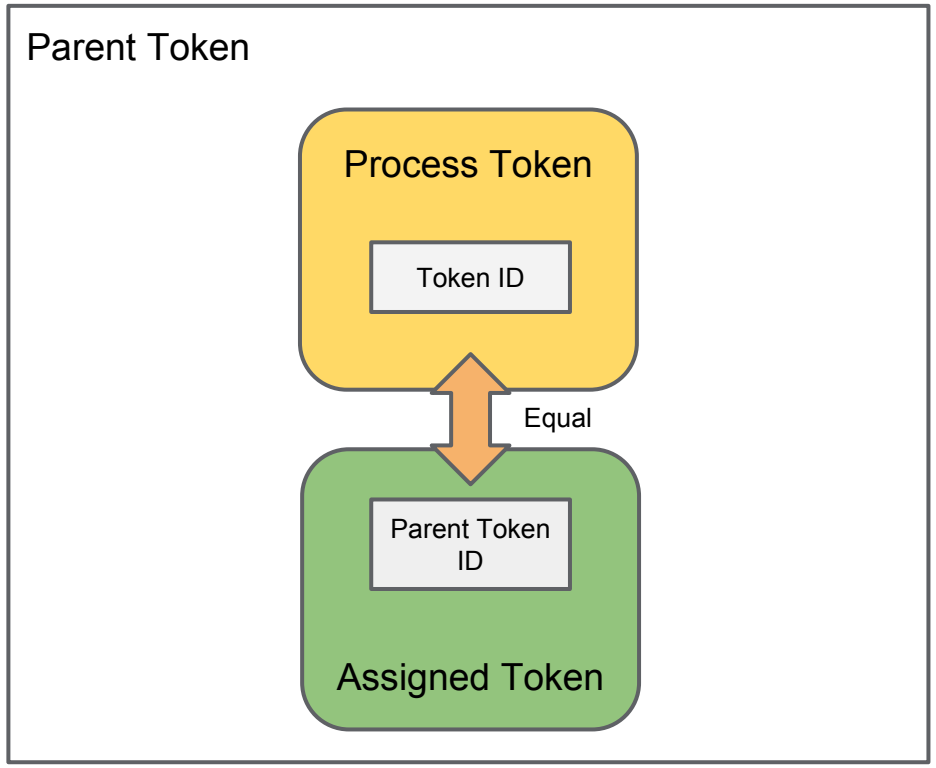
- By default the parent of a new process is the one calling NtCreateUserProcess
- Can specify a new parent process using the Parent Process Attribute
  - Handle must have PROCESS_CREATE_PROCESS access right, a GENERIC_WRITE access
- Defaults to inheriting the primary token from the parent
  - Can be used to spawn a System Process with only SeDebugPrivilege

DEMO

Parent Processes

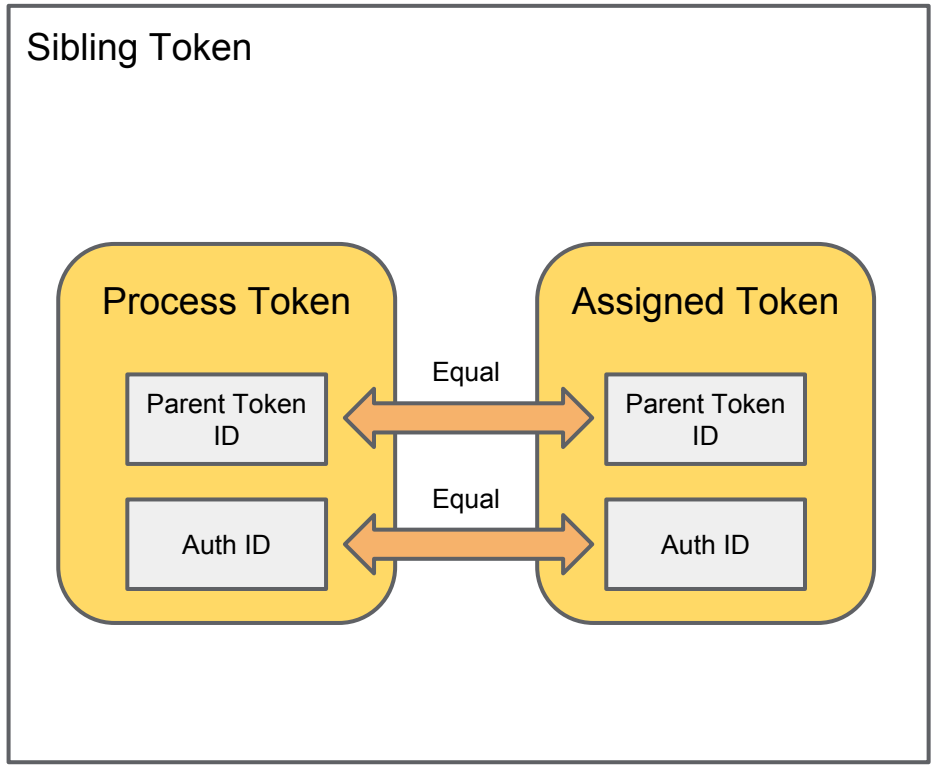
Restriction on Explicit Process Token

- Assigned through specifying Token Process Attribute, or after process creation using NtSetInformationProcess
- If caller doesn't have Assign Primary Token Privilege one of two tokens are allowed



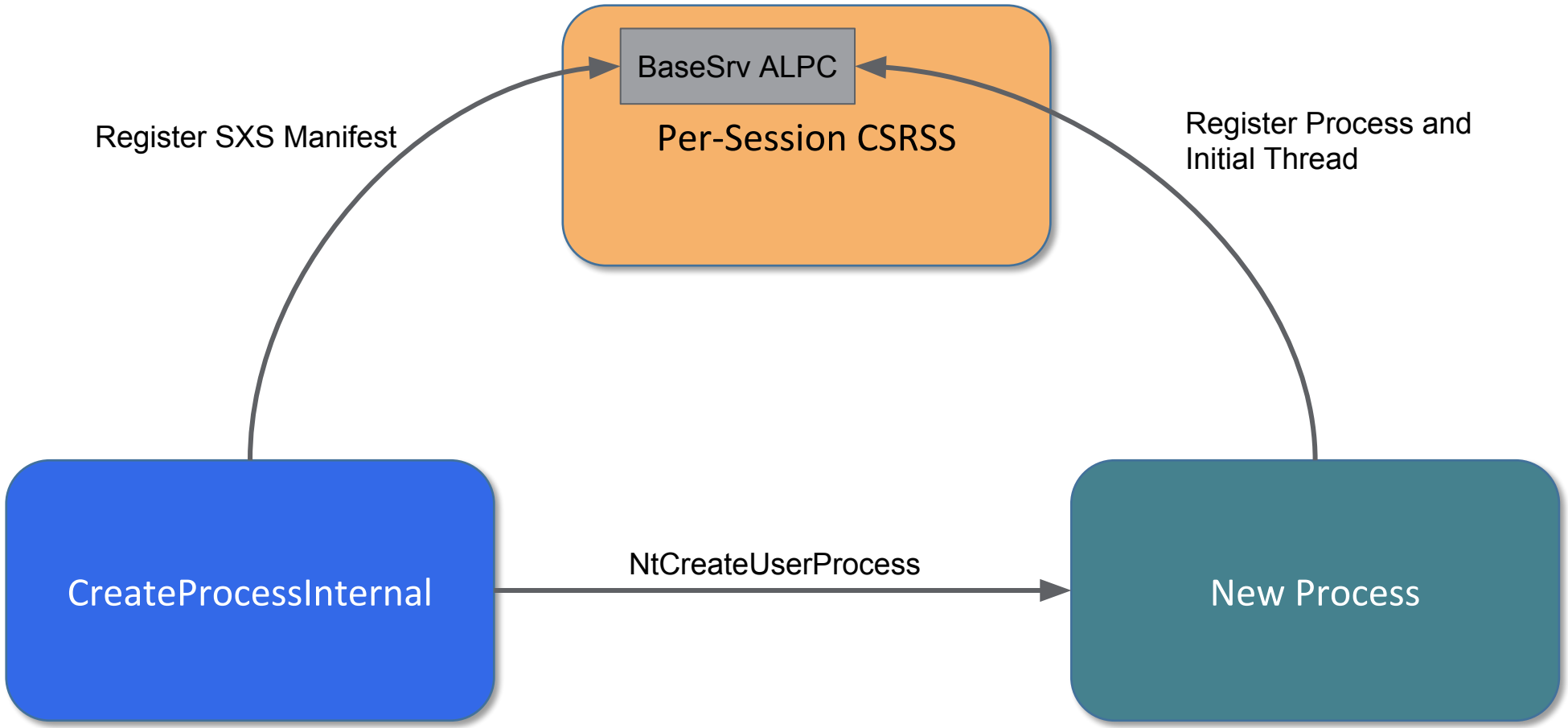
CreateRestrictedToken

OR



DuplicateToken
NtCreateLowBoxToken

CSRSS and the Pain of Subsystems

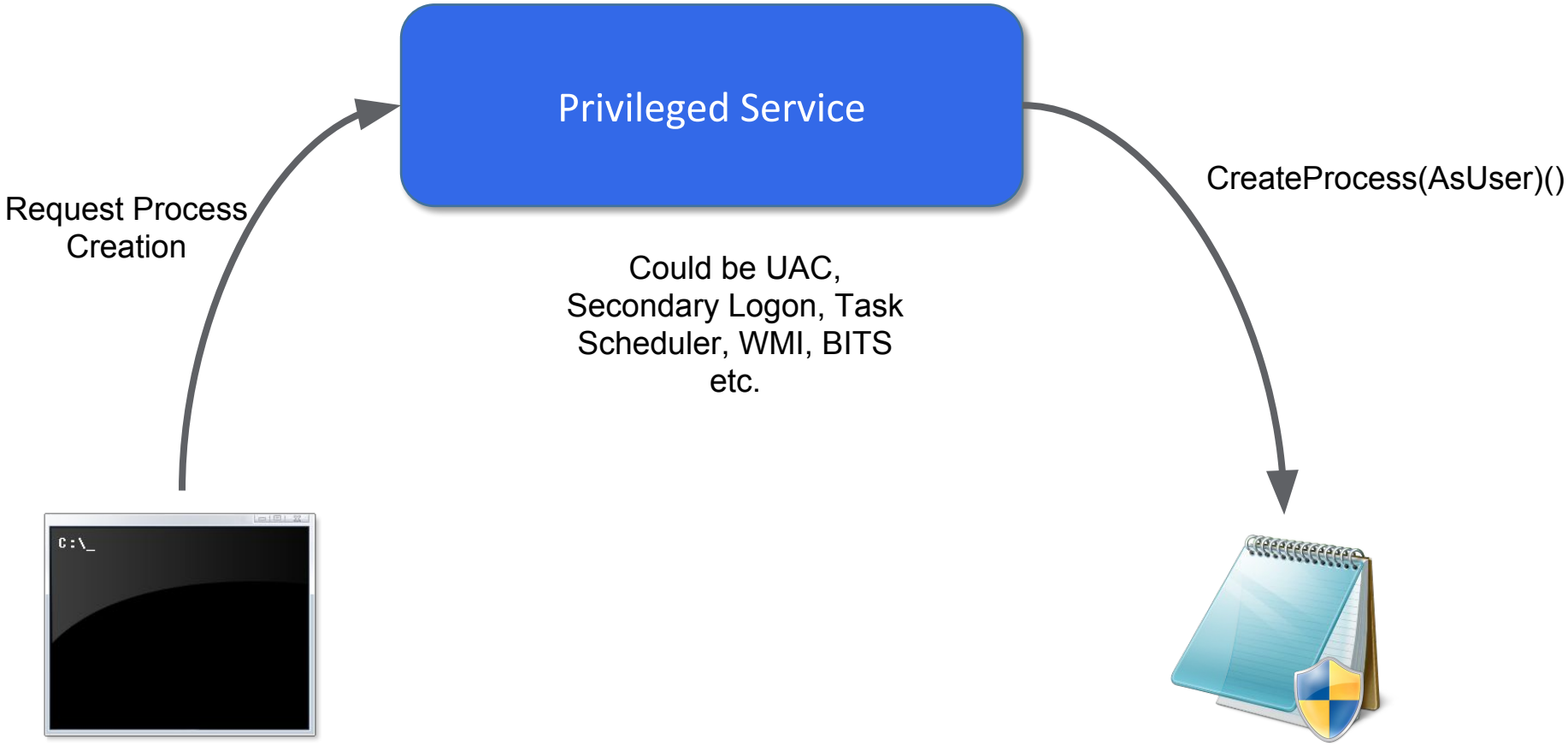


Handle Inheritance

- Unlike *nix *exec* Windows processes don't automatically inherit open handles
- Handles must be marked as inheritable
  - Creation time using API specific mechanism
  - Using Set-Handle-Information with the HANDLE_FLAG_INHERIT flag
- Must specify the *Inherit Handles* process creation flag
- All inheritable handles are passed to child from the parent process
  - Can restrict what handles are passed using the Handle List process attribute

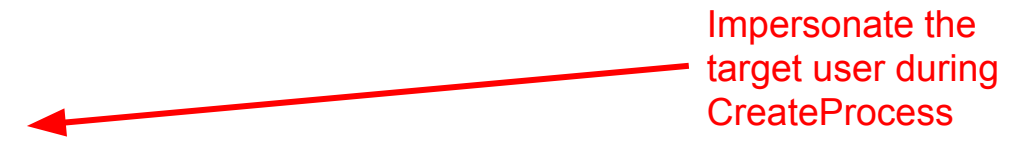
Process Handling Bug Classes

Process Creation Services



Canonical “Safe” User Process Creation

```
BOOL CreateProcessForUser(HANDLE Token,
                          LPCWSTR CommandLine)
{
  STARTUPINFO startInfo = {};
  PROCESS_INFORMATION procInfo = {};

  ImpersonateLoggedOnUser(Token);
  ret = CreateProcessAsUser(Token, NULL, CommandLine);
  RevertToSelf();
  return ret;
}
```



Let's Look at how NtCreateUserProcess Opens Files

```
RtlInitUnicodeString(&name, Attributes->ImagePath);
InitializeObjectAttributes(&obja, &name,
    OBJ_FORCE_ACCESS_CHECK | OBJ_KERNEL_HANDLE |
    OBJ_CASE_INSENSITIVE);

HANDLE hFile;
NTSTATUS status = ZwOpenFile(&hFile,
 Attributes->AdditionalFileAccess | FILE_EXECUTE,
       &obja, ...);

if (status < 0)
  status = ZwOpenFile(&hFile, FILE_EXECUTE, &obja, ...);
```

Force Access Check

Doesn't impersonate process token

Bug Class: Creating User Process No Impersonation

```
BOOL CreateProcessForUser(LPCWSTR CommandLine)
{
  STARTUPINFO startInfo = {};
  PROCESS_INFORMATION procInfo = {};

  RpcImpersonateClient();
  HANDLE Token;
  OpenThreadToken(GetCurrentThread(), &Token, ...);
  RevertToSelf();

  return CreateProcessAsUser(Token, NULL, CommandLine);
}
```

Capture the caller's token

Create Process outside of impersonation

Example Bug - PZ Issue 161

Windows: Task Scheduler Executable File Permissions Bypass

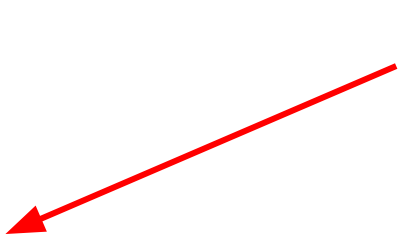
Project Member Reported by forshaw@google.com, Nov 11, 2014

```
Windows: Task Scheduler Executable File Permissions Bypass
Platform: Windows 8.1 Update 32/64 bit (7 doesn't seem to be vulnerable)
Class: Security Bypass

The Windows Task Scheduler can be abused by a low privileged user to execute files which they don't have access to because of
access control permissions. The only requirement is LocalSystem is allowed to access the executable file, and that the executable
doesn't need to access other resources from the same location (this rules out .NET applications which re-open their executable
process).

The bug comes from the task scheduler creating the process directly from the privileged service without impersonating the target
user. Even though the service is calling CreateProcessAsUser the API doesn't limit the process creation to the user's token,
instead the access check is done against the LocalSystem token. As long as the executable is reasonably standalone once it's
mapped into memory as LocalSystem it will be allowed to execute once the token is swapped for the low-privileged user.

Locking down executable files on disk is a common technique in Enterprise environments. This seems to circumvent Software
Restriction Policy (I've not tried AppLocker). Of course if you've got code executing already to setup the task it isn't clear if
that is more of a security issue. I suppose if the user's permitted to setup scheduled tasks it might be an issue.
```

Added Bonus, bypass of SRP

Bug Class: Creating Privileged Process While Impersonating User

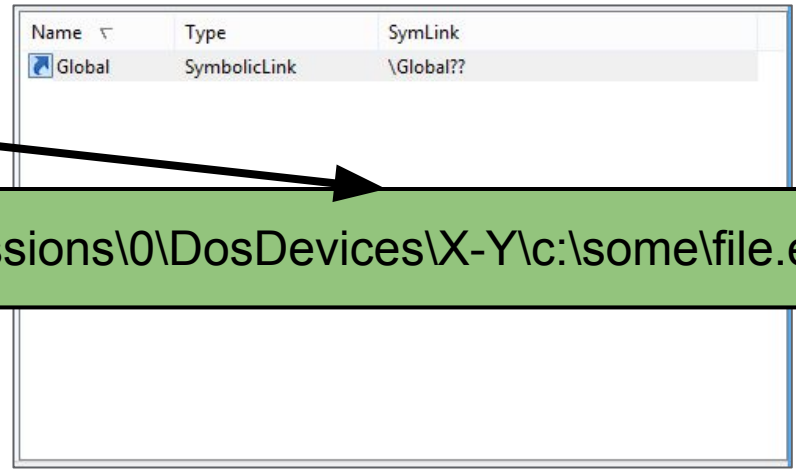
```
BOOL DoSomeThingForUser()
{
  RpcImpersonateClient();
  // Do something.
  CreateProcess(NULL, "C:\\somepath\\somefile.exe ...");
  // Do something else.
  RevertToSelf();
}
```



Important: Process Token not inherited from Impersonation Token

DosDevice Prefix During Impersonation

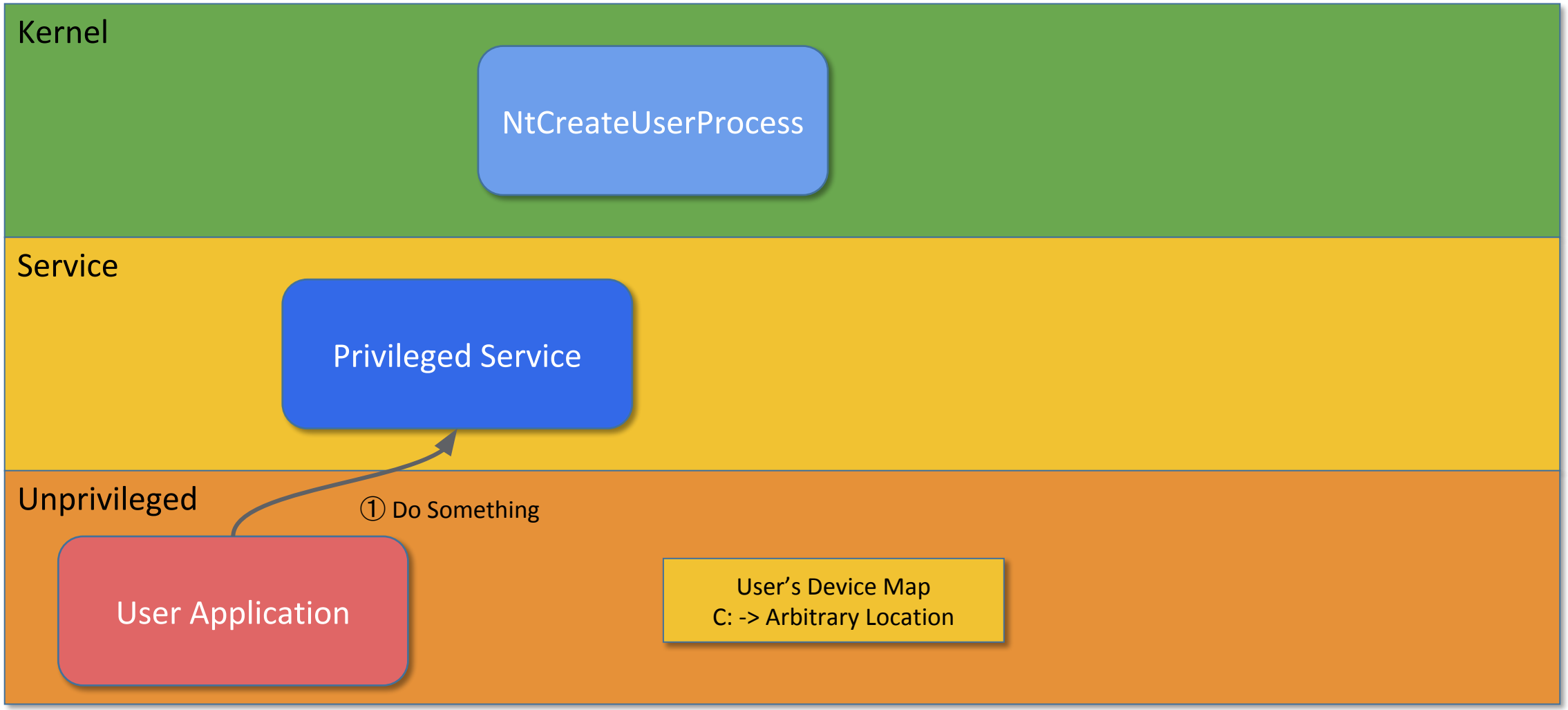
\??\c:\some\file.exe



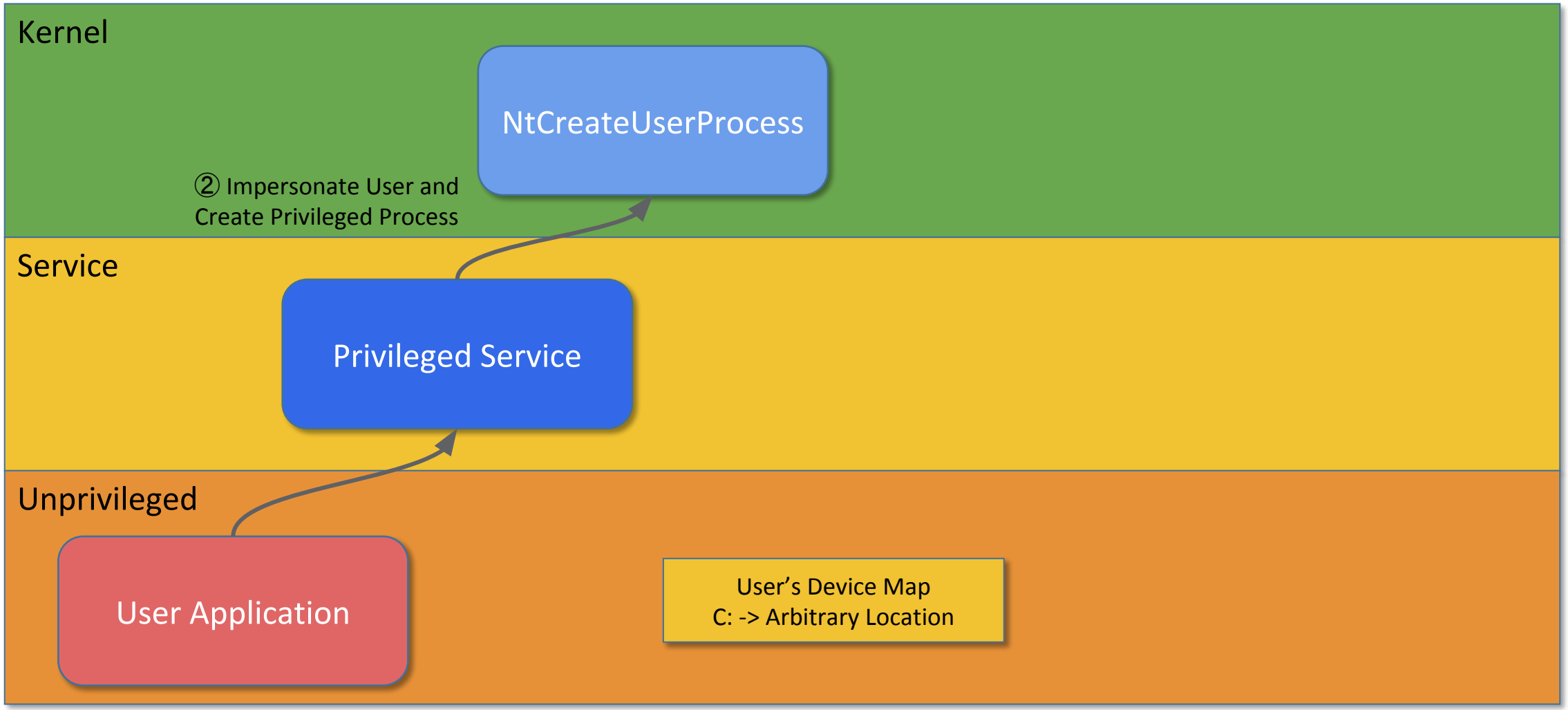
\Sessions\0\DosDevices\X-Y\c:\some\file.exe

This can be disabled using the
OBJ_IGNORE_IMPERSONTED_DEVICE_MAP flag
But NtCreateUserProcess doesn't specify the flag.

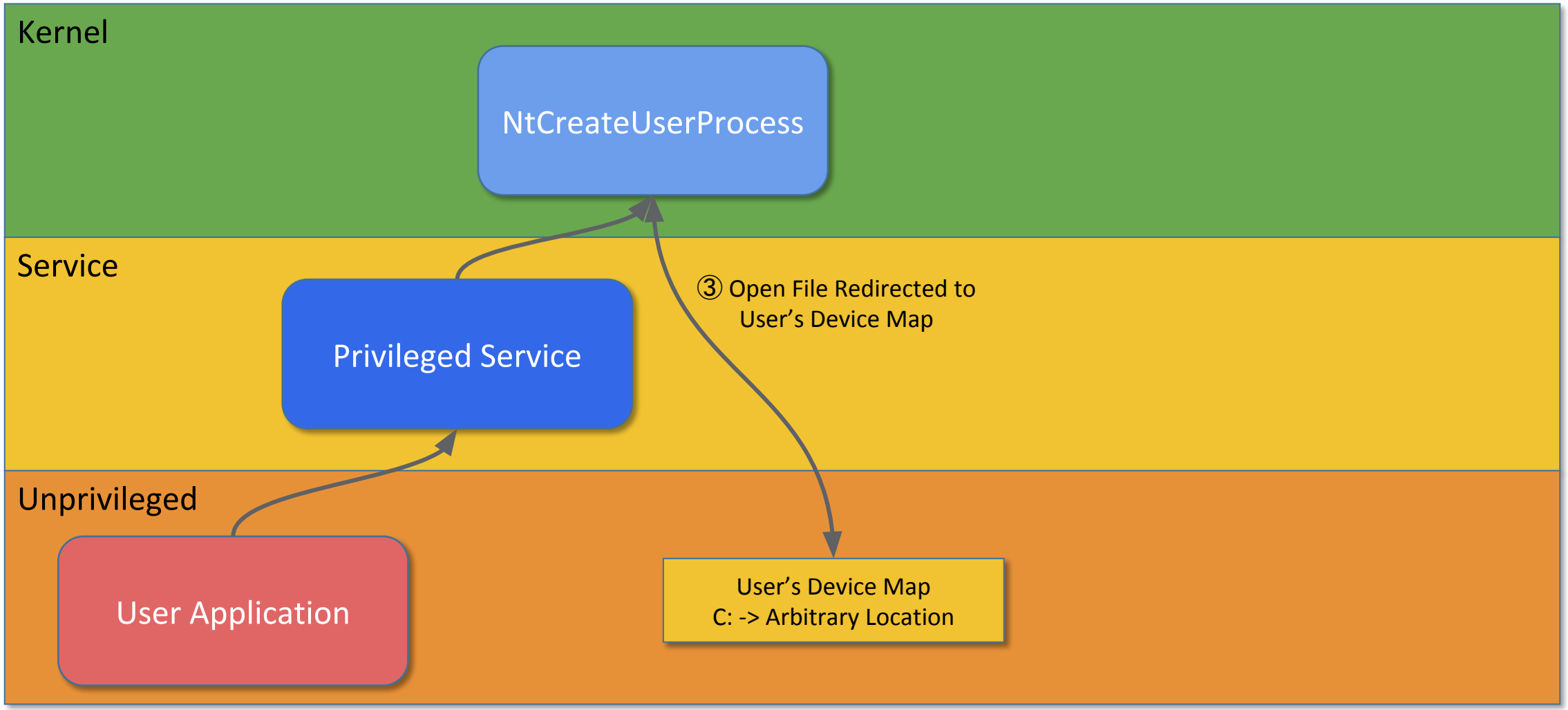
Creating Privileged Process



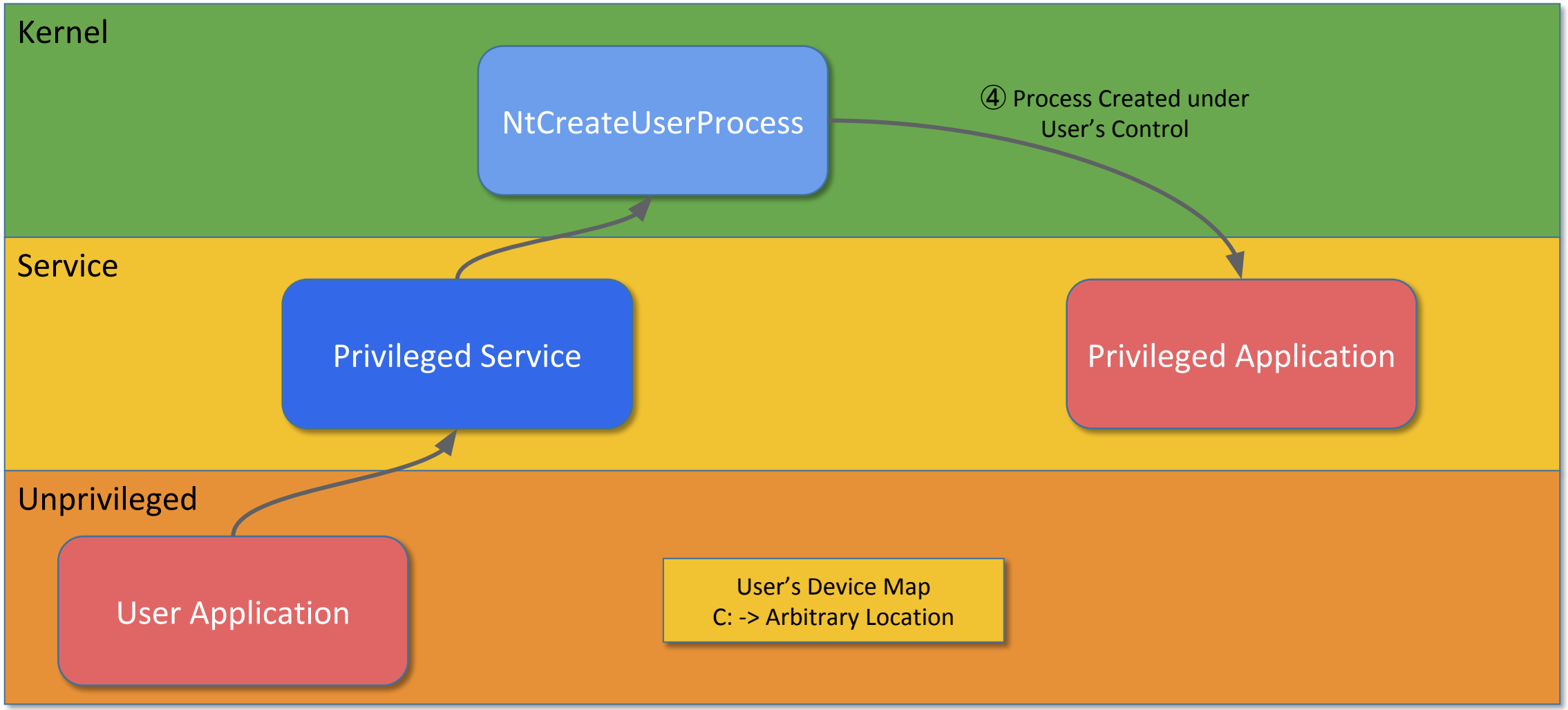
Creating Privileged Process



Creating Privileged Process



Creating Privileged Process



MS14-027

Wednesday, 21 May 2014

Impersonation and MS14-027

The recent MS14-027 patch intrigued me, a local EoP using ShellExecute. It seems it also intrigued others so I pointed out how it probably worked on Twitter but I hadn't confirmed it. This post is just a quick write up of what the patch does and doesn't fix. It turned out to be more complex than it first seemed and I'm not even sure it's correctly patched. Anyway, first a few caveats, I am fairly confident that what I'm presenting here is already known to some anyway. Also I'm not providing direct exploitation details, you'd need to find the actual mechanism to get the EoP working (at least to LocalSystem).

I theorized that the issue was due to mishandling of the registry when querying for file associations. Specifically the handling of *HKEY_CLASSES_ROOT* (*HKCR*) registry hive when under an impersonation token. When the ShellExecute function is passed a file to execute it first looks up the extension in the HKCR key. For example if you try to open a text file, it will try and open *HKCR\.txt*. If you know anything about the registry and how COM registration works you might know that *HKCR* isn't a real registry hive at all. Instead it's a merging of the keys *HKEY_CURRENT_USER\Software\Classes* and *HKEY_LOCAL_MACHINE\Software\Classes*. In most scenarios *HKCU* is taken to override *HKLM* registration as we can see in the following screenshot from Process Monitor (note PM records all access to *HKLM* classes as *HKCR* confusing the issue somewhat).

RegOpenKey	HKCU\Software\Classes\.txt	NAME NOT FOUND	Desired Access: Query Value
RegOpenKey	HKCR\.txt	SUCCESS	Desired Access: Query Value

https://tyranidslair.blogspot.co.uk/2014/05/impersonation-and-ms14-027.html

Bug Class: Mismatched Impersonation to Process Token

Process token specified explicitly

```
BOOL CreateProcessForUser(HANDLE Token,
                          LPCWSTR CommandLine)
{
  STARTUPINFO startInfo = {};
  PROCESS_INFORMATION procInfo = {};

  RpcImpersonateClient();
  ret = CreateProcessAsUser(Token, NULL, CommandLine);
  RevertToSelf();
  return ret;
}
```

Impersonate Client, might not be the same as process token

Example Bug - PZ Issue 692

Windows: CSRSS BaseSrvCheckVDM Session 0 Process Creation EoP

Project Member Reported by forshaw@google.com, Jan 5, 2016

```
Windows: CSRSS BaseSrvCheckVDM Session 0 Process Creation EoP
Platform: Windows 8.1, not tested on Windows 10 or 7
Class: Elevation of Privilege

Summary:
The CSRSS BaseSrv RPC call BaseSrvCheckVDM allows you to create a new process with the anonymous token, which results on a new
process in session 0 which can be abused to elevate privileges.

Description:

CSRSS/basesrv.dll has a RPC method, BaseSrvCheckVDM, which checks whether the Virtual DOS Machine is installed and enabled. On
Windows 8 and above the VDM is off by default (on 32 bit Windows) so if disabled CSRSS tries to be helpful and spawns a process
on the desktop which asks the user to install the VDM.  The token used for the new process comes from the impersonation token of
the caller. So by impersonating the anonymous token before the call to CsrClientCallServer we can get CSRSS to use that as the
primary token. As the anonymous token has a Session ID of 0 this means it creates a new process in session 0 (because nothing
else changes the session ID).
```

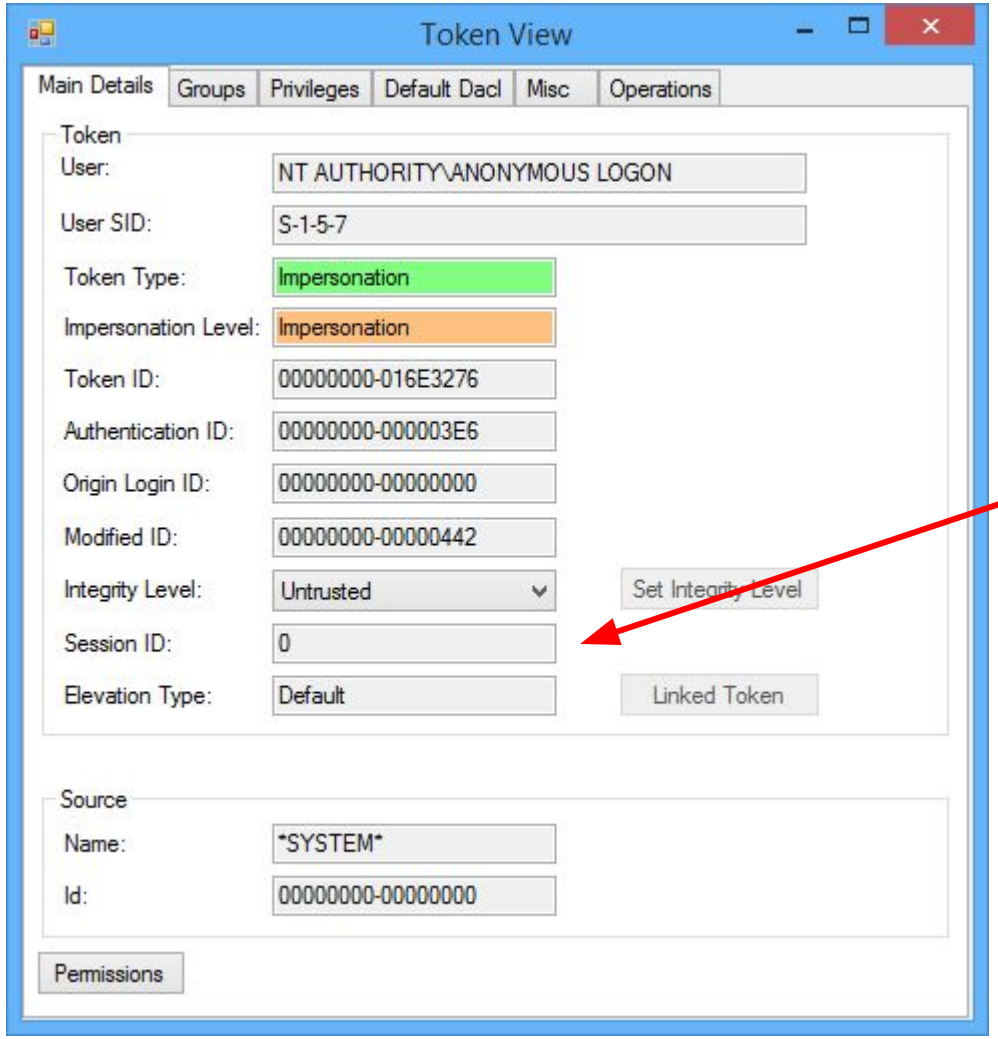
BaseSrvCheckVDM

- CSRSS call spawns a helper process on the user's desktop to enable the VDM
- Implemented even in 64 bit Windows even though no VDM available

```
void BaseSrvCheckVDM() {
  RpcImpersonateClient();
  OpenThreadToken(..., &hToken);
  HANDLE hPrimaryToken;
  DuplicateTokenEx(hToken, ...,
      TokenPrimary, &hPrimaryToken);
  RevertToSelf();

  BasepImpersonateClientProcess();
  CreateProcessAsUser(hPrimaryToken,
            L"\\SystemRoot\\...");
  CsrRevertToSelf()
}
```

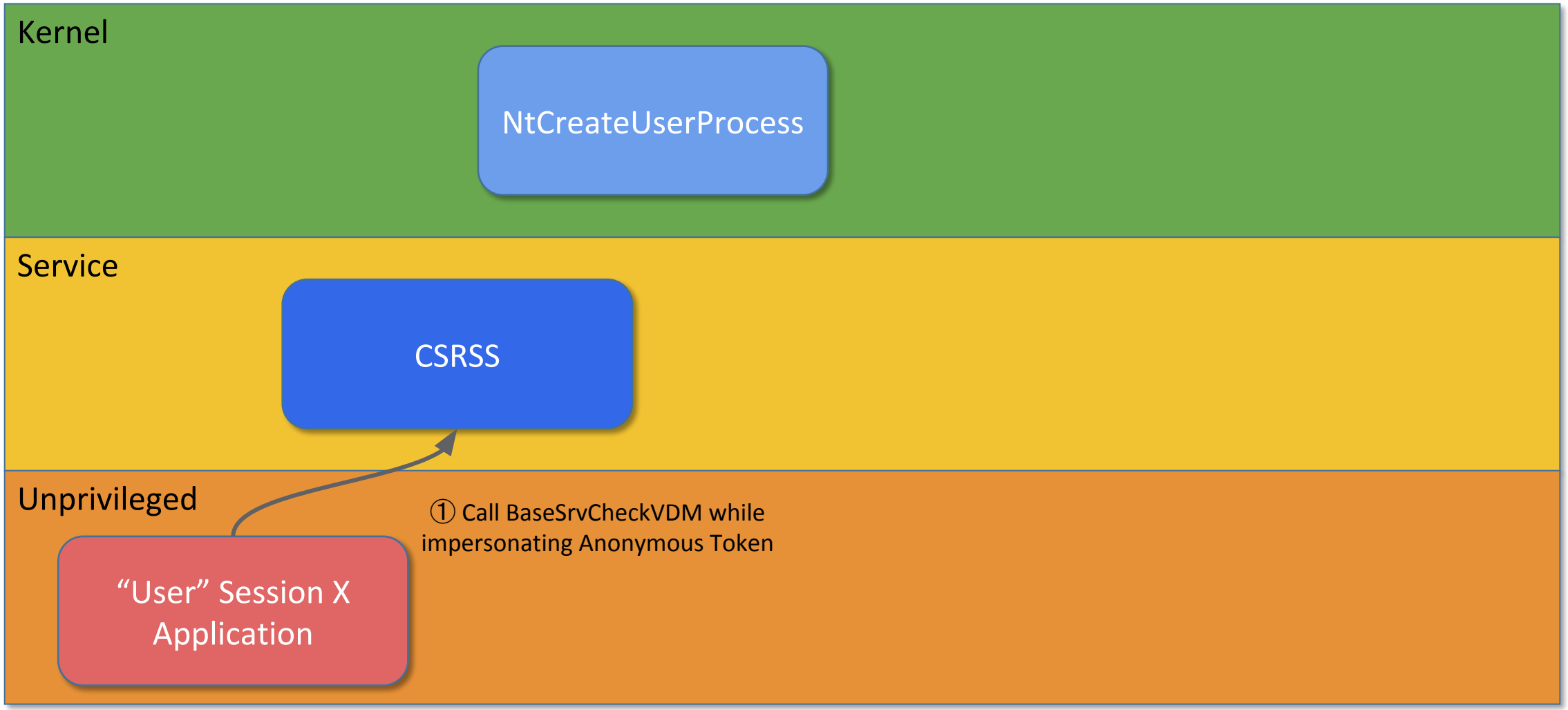
```
void BasepImpersonateClientProcess() {
  HANDLE hProcess = GetCaller();
  OpenProcessToken(hProcess, &hToken);
  DuplicateToken(hToken,
     SecurityImpersonation,
     &hImpToken);
  ImpersonateLoggedOnUser(hThread,
                          hImpToken);
}
```

Anonymous Token

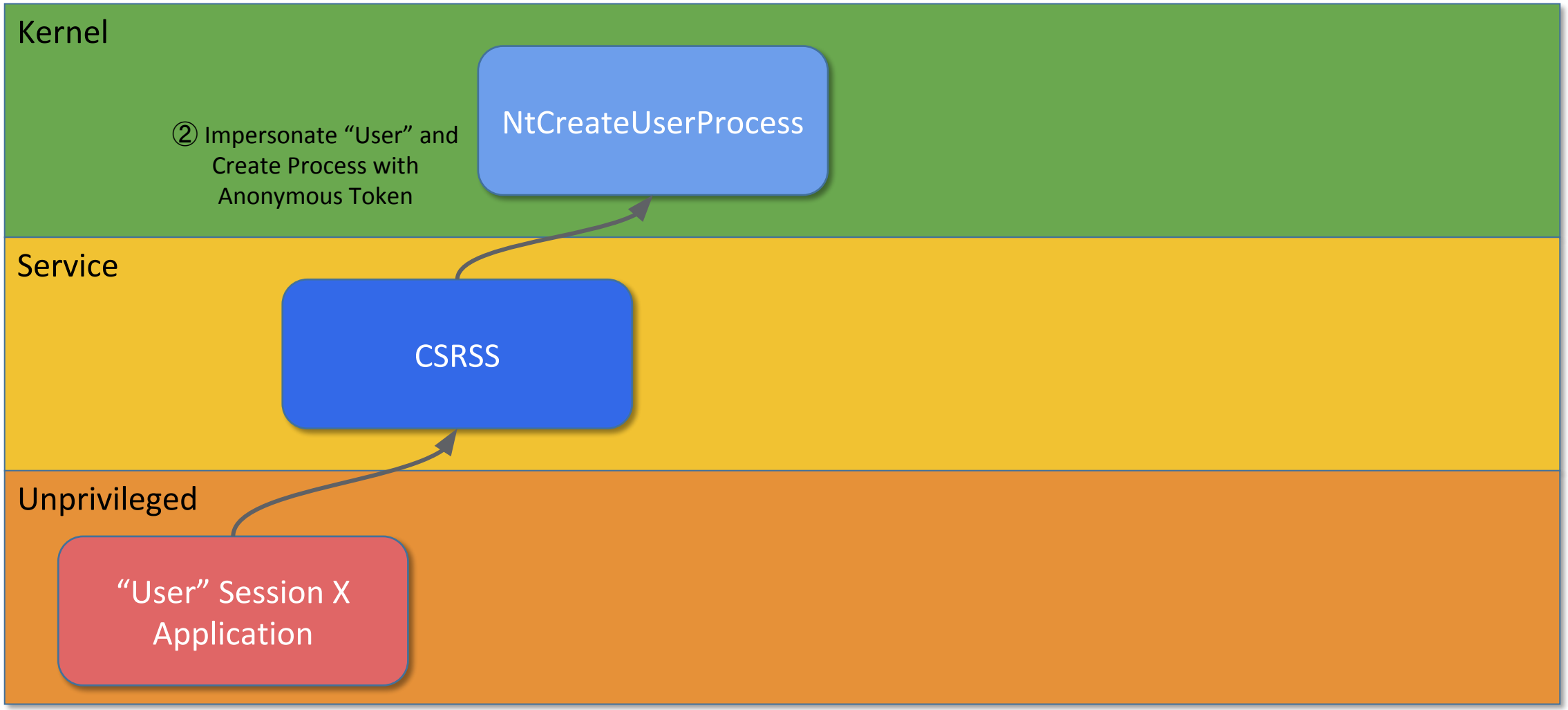


Token has a
Session ID of 0

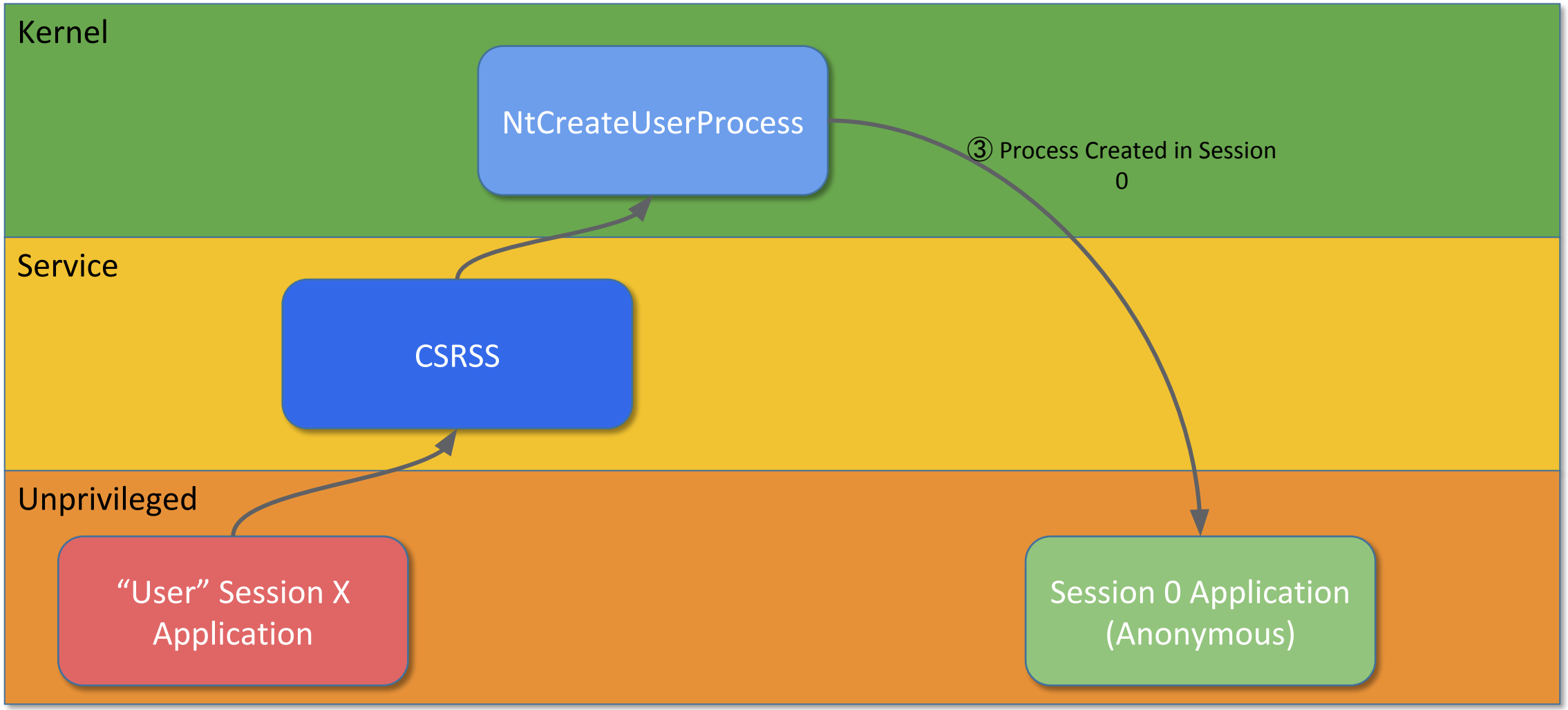
Process Creation Services



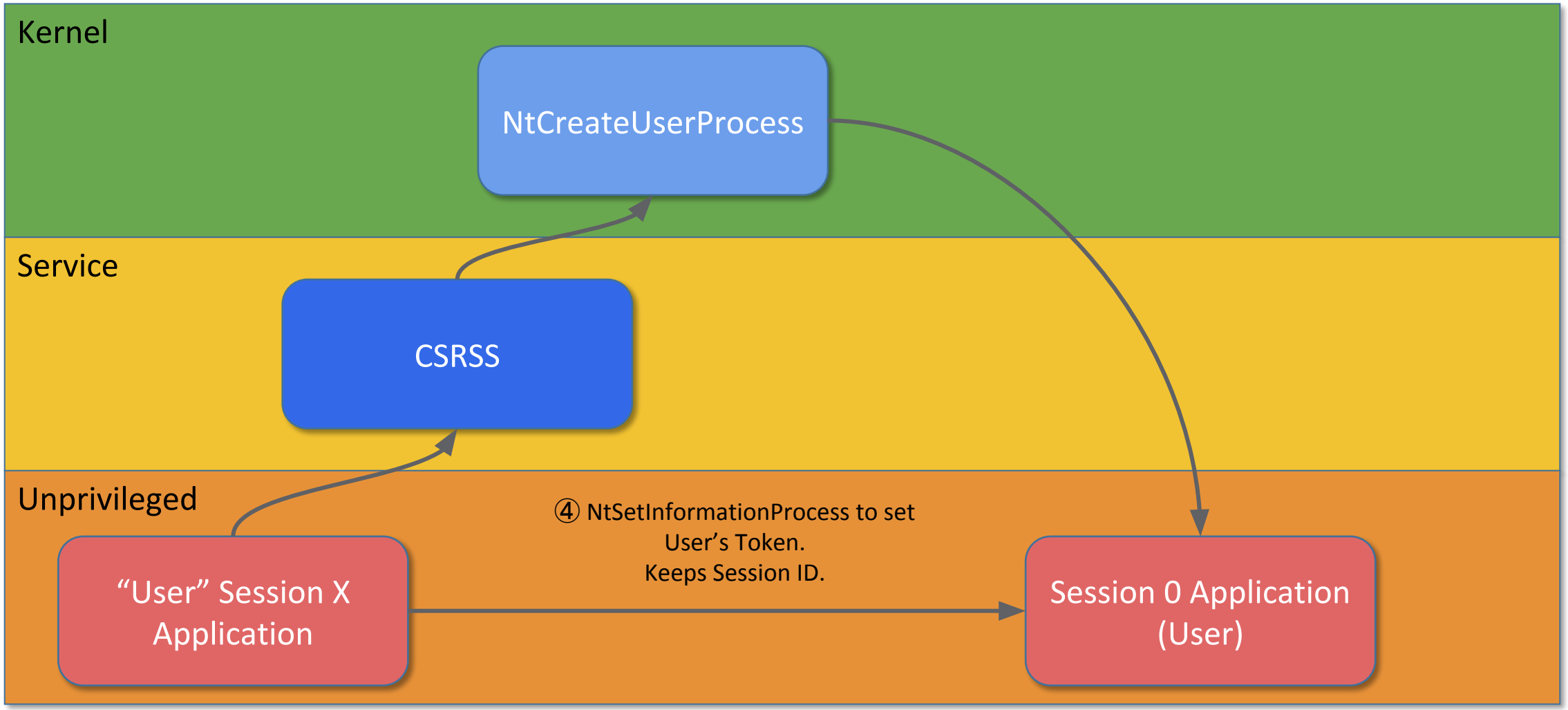
Process Creation Services



Process Creation Services



Process Creation Services

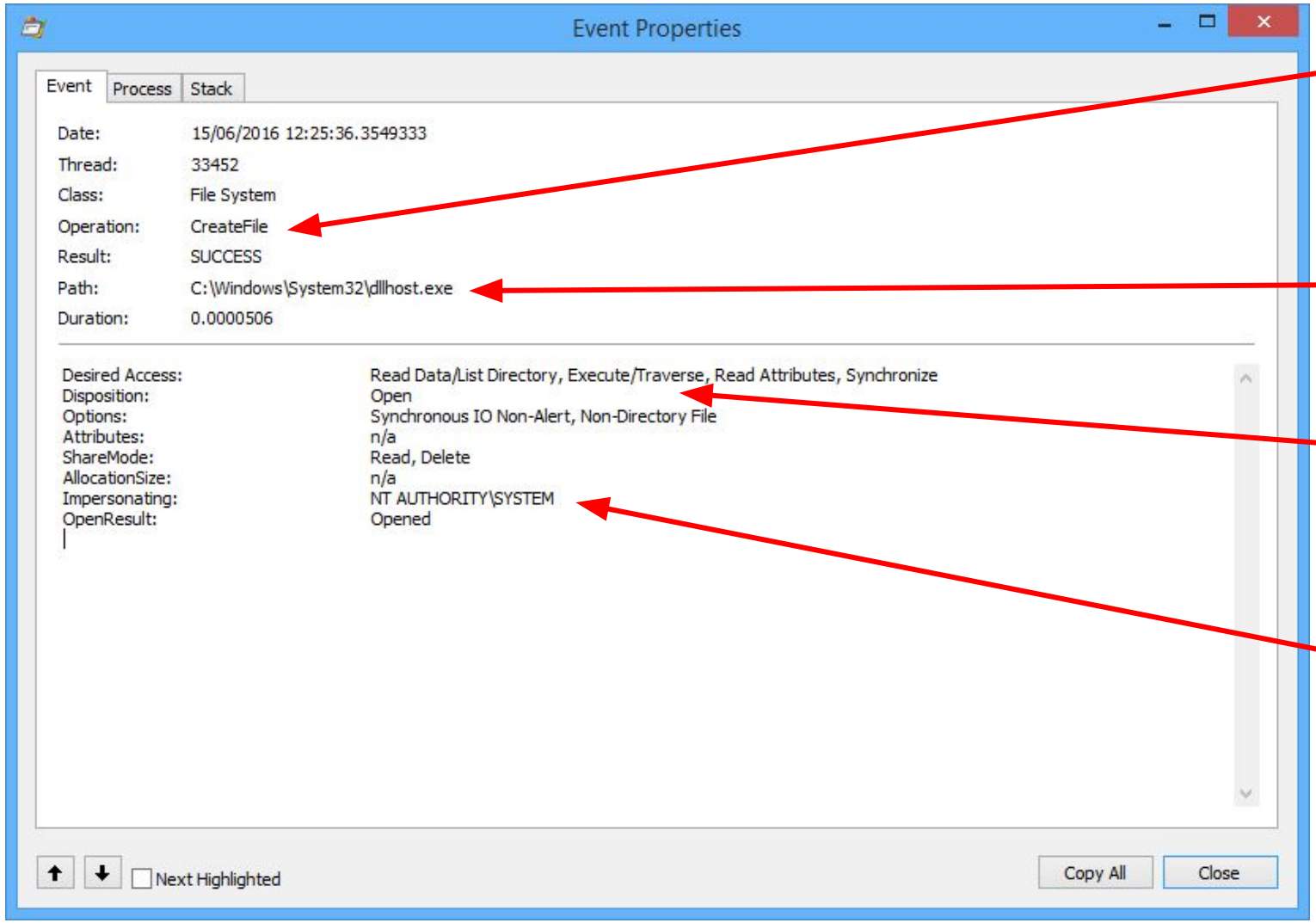


What Can You Do in Session 0?

- Session 0 since Vista is reserved only for services so being able to create a process in that session could have unexpected consequences.
- For example Sessions other than 0 can't create Sections or Symbolic Links in *BaseNamedObjects* to prevent planting attacks:

```
if (Directory->SessionId != -1 &&
    (Type == MmSectionObjectType || Type == ObpSymbolicLinkObjectType) &&
     Directory->SessionId != PsGetCurrentProcessSessionId() &&
     !SeSinglePrivilegeCheck(SeCreateGlobalPrivilege, PreviousMode) )
{
  if (!ObpIsUnsecureName(Name, CaseSensitive))
    return STATUS_ACCESS_DENIED;
}
```

Find Bugs Using Process Monitor



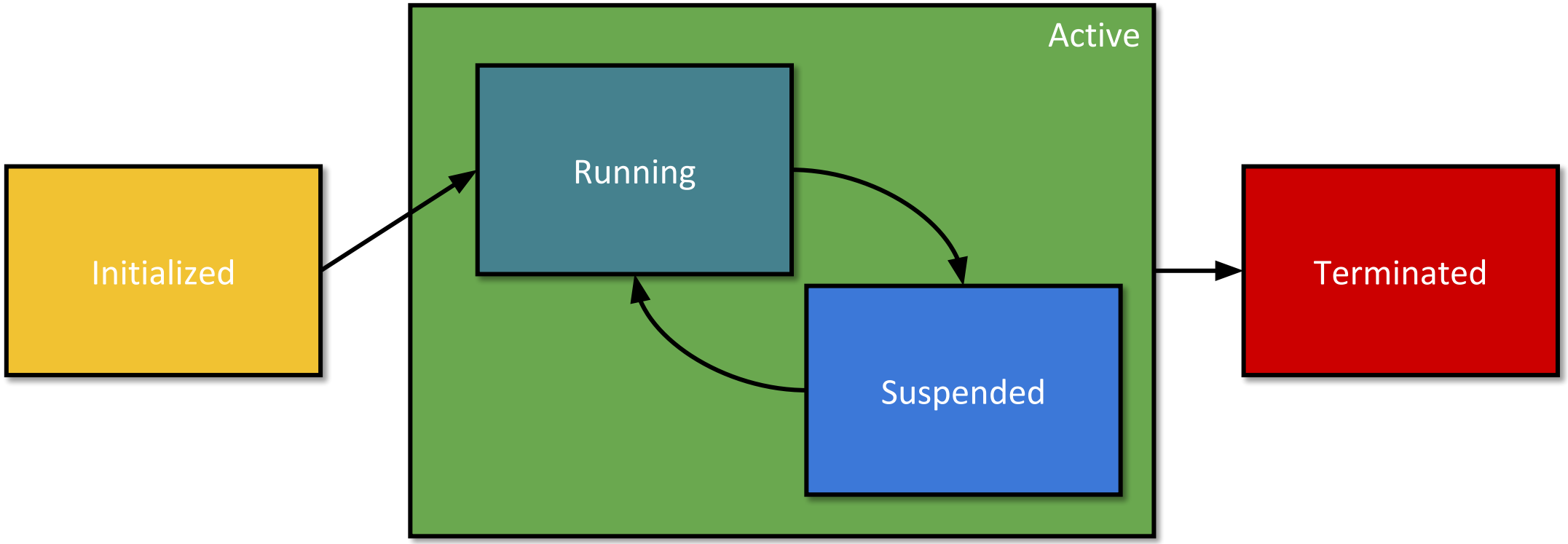
Set operation to *CreateFile*

Path ends with *.exe*

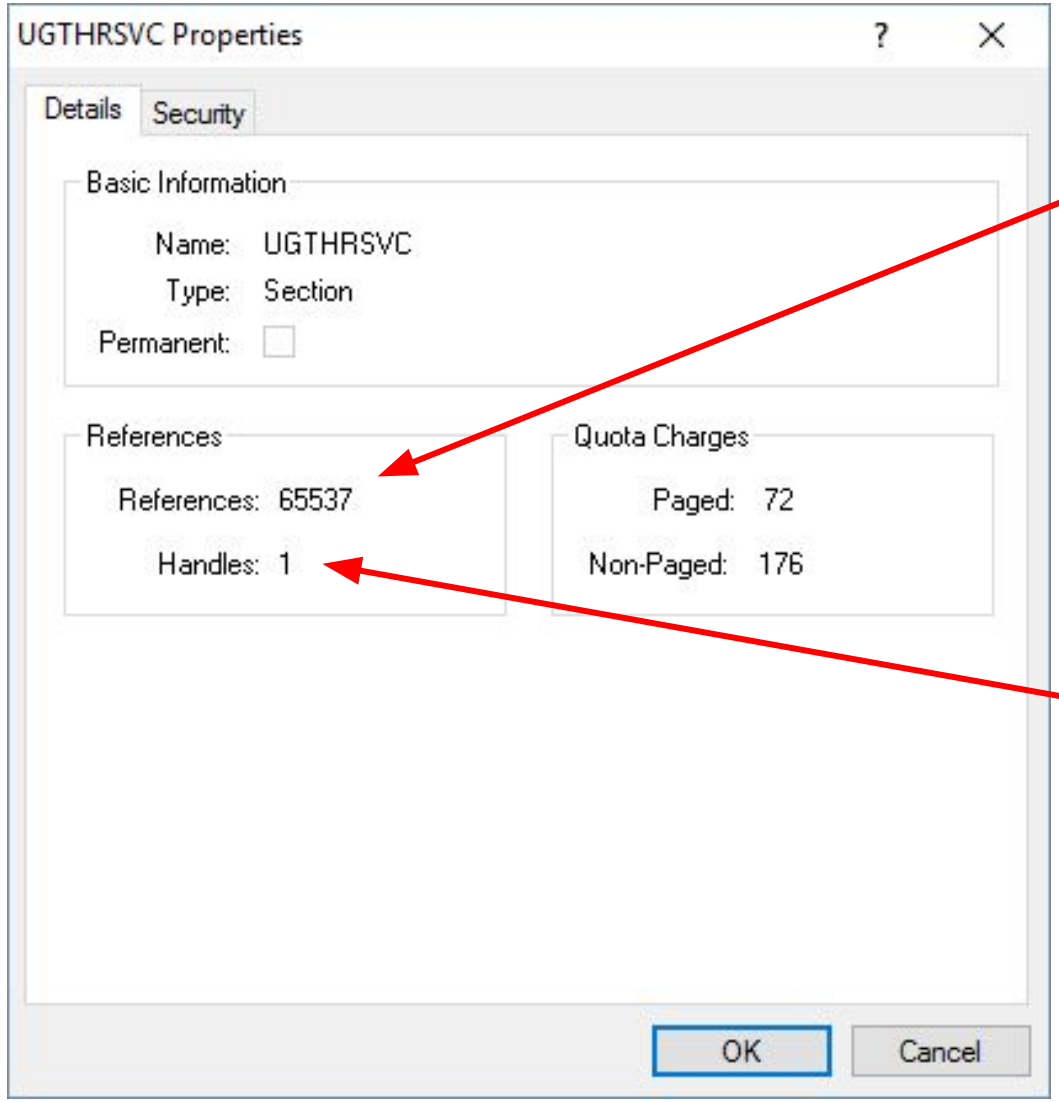
Detail contains open for Execute/Traverse Access

Detail contains Impersonating (or not)

Process Lifecycle



Windows Kernel Object Lifetimes

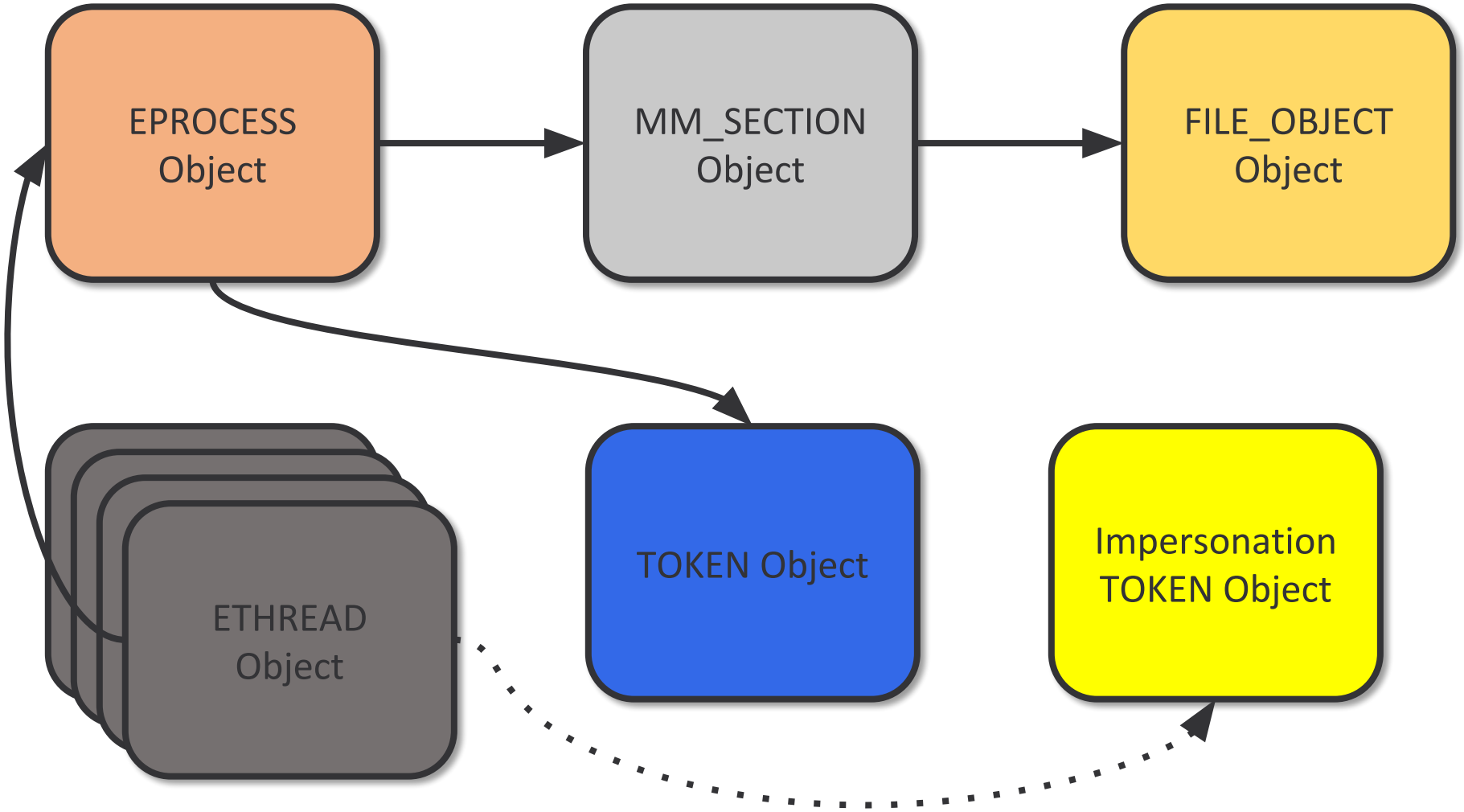


Kernel Reference Count

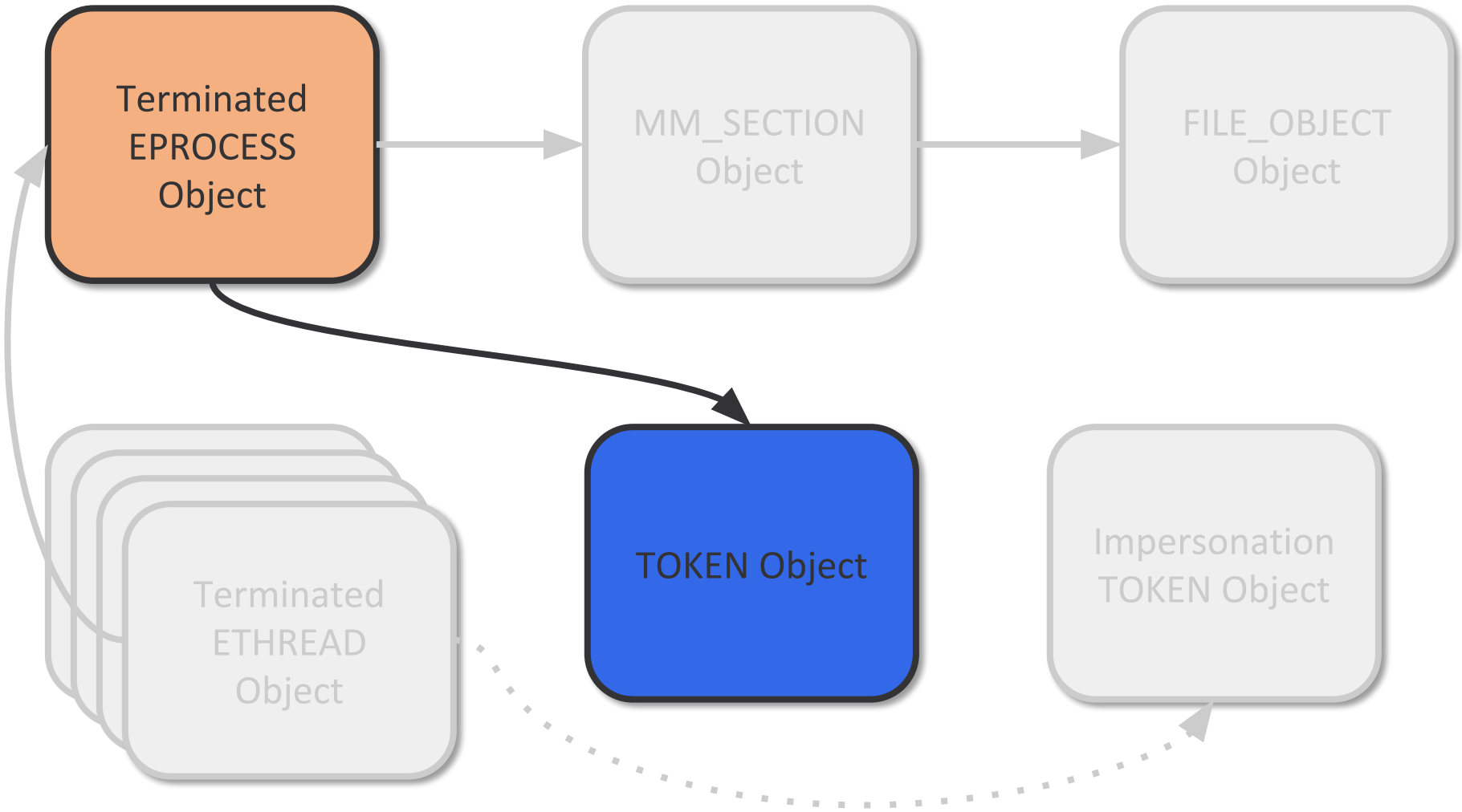
User Handle Count

EPROCESS object not deleted until both counts go to 0

Kernel Objects Reference Graph



Terminated Process



DEMO

Exposing Terminated Processes

Cross Session Terminated Processes

- When a user logs out the Session is terminated, that should also terminate the process
- All handles are removed from terminated processes
- However we've seen that the kernel objects stick around if we there's any reference to the process or thread
- Even so what use would it be?

Example Bug - PZ Issue 483

Windows: NtCreateLowBoxToken Handle Capture Local DoS/Elevation of Privilege

Project Member Reported by forshaw@google.com, Jul 16, 2015

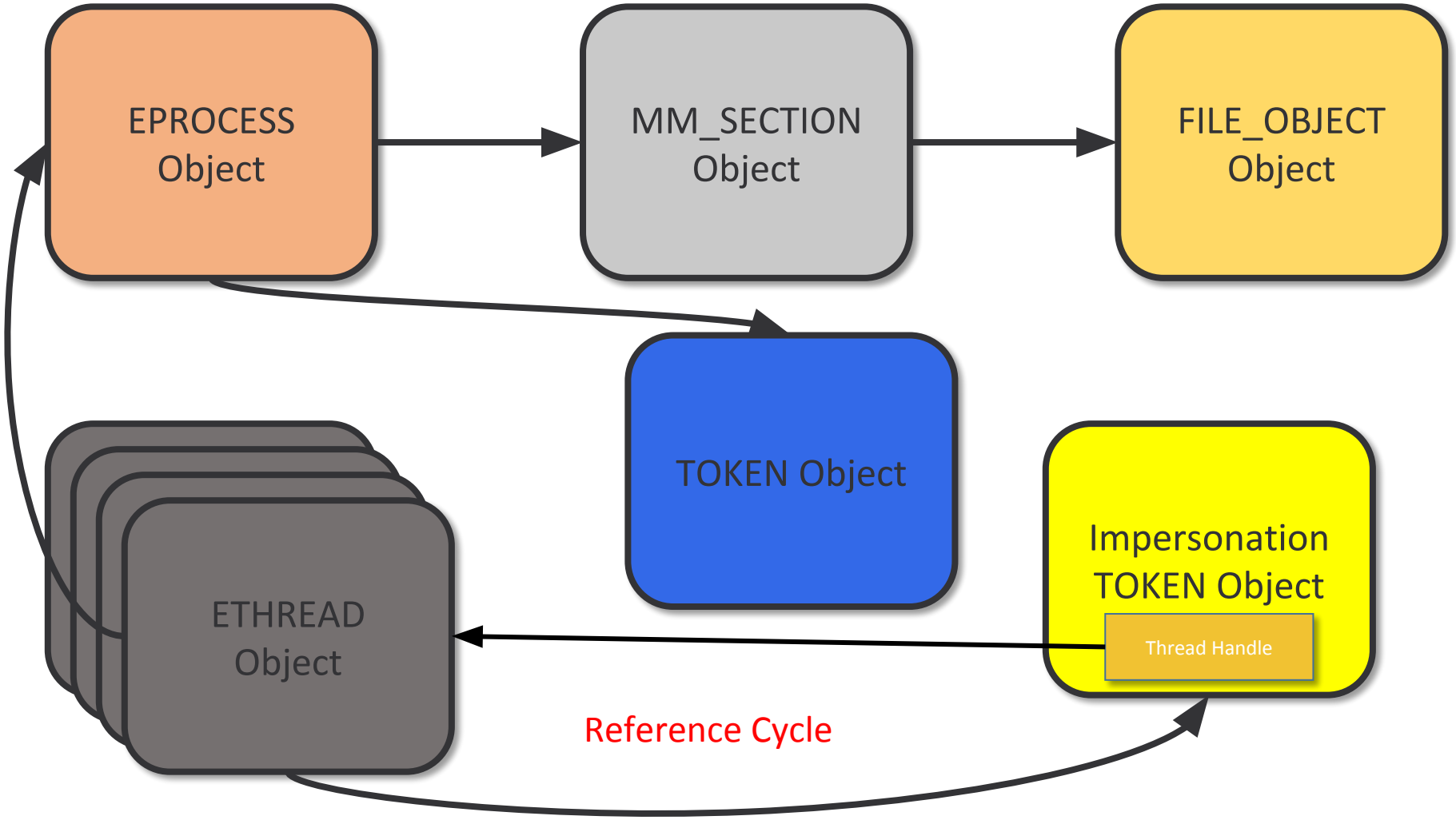
```
Windows: NtCreateLowBoxToken Handle Capture Local DoS/Elevation of Privilege
Platform: Windows 8.1 Update, Windows 10, Windows Server 2012
Class: Local Dos/Elevation of Privilege

Summary:
The NtCreateLowBoxToken API allows the capture of arbitrary handles which can lead to to local DoS or elevation of privilege.

Description:

The NtCreateLowBoxToken system call accepts an array of handles which are stored with the new token. This is presumably for
maintaining references to the appcontainer specific object directories and symbolic links so that they do not need to be
maintained anywhere else. The function, SepReferenceLowBoxObjects which captures the handles has a couple of issues which can
lead to abuse:

1) It calls ZwDuplicateObject which means the API can capture kernel handles as well as user handles.
2) No checks are made on what object types the handles represent.
```

NtCreateLowBoxToken

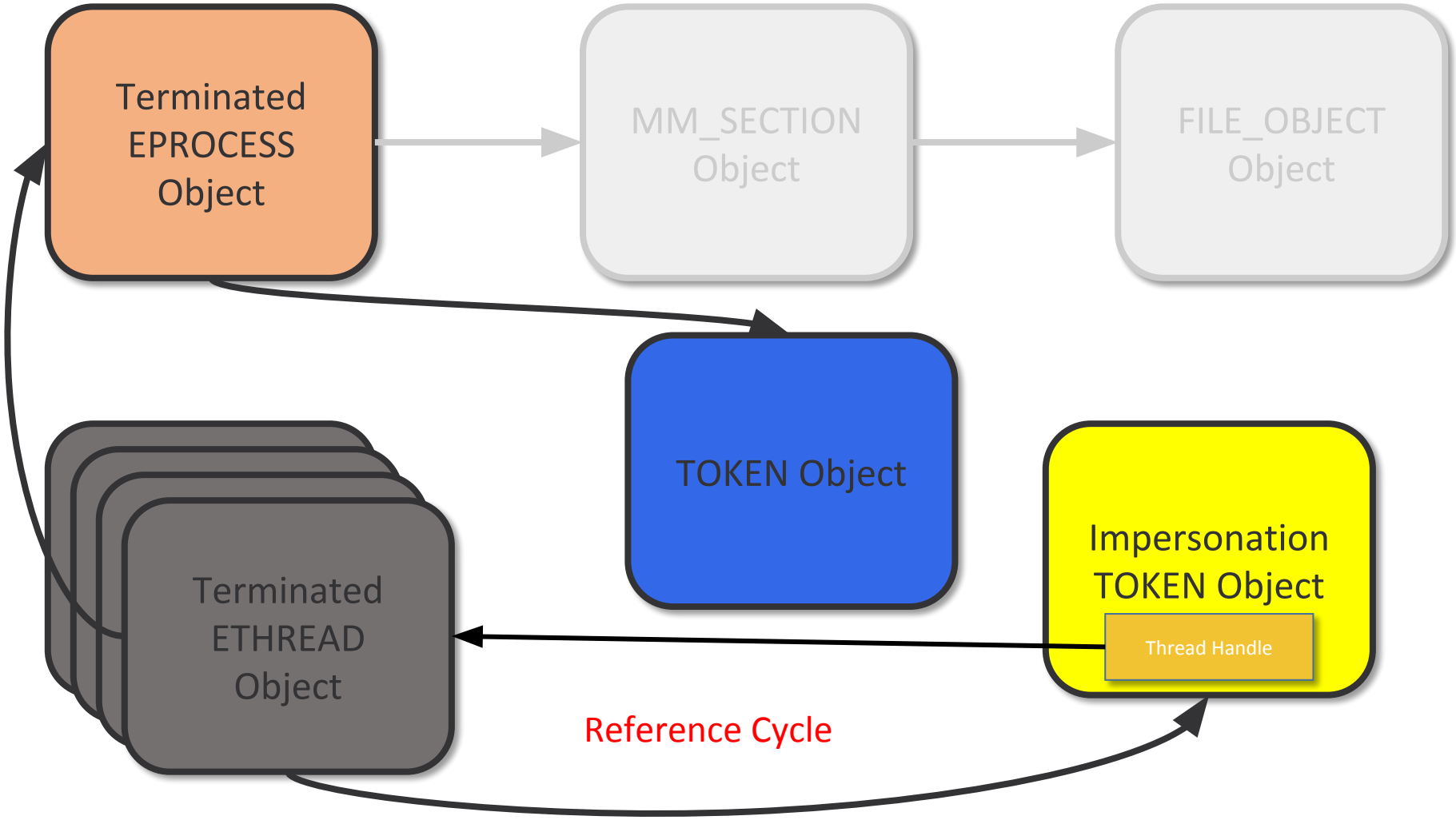
```
NTSTATUS NtCreateLowBoxToken(
  _Out_ PHANDLE TokenHandle,
  _In_ HANDLE ExistingTokenHandle,
  _In_ ACCESS_MASK DesiredAccess,
  _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
  _In_ PSID PackageSid,
  _In_ ULONG CapabilityCount,
  _In_opt_ PSID_AND_ATTRIBUTES Capabilities,
  _In_ ULONG HandleCount,
  _In_opt_ HANDLE *Handles
);
```

Arbitrary List of Handles

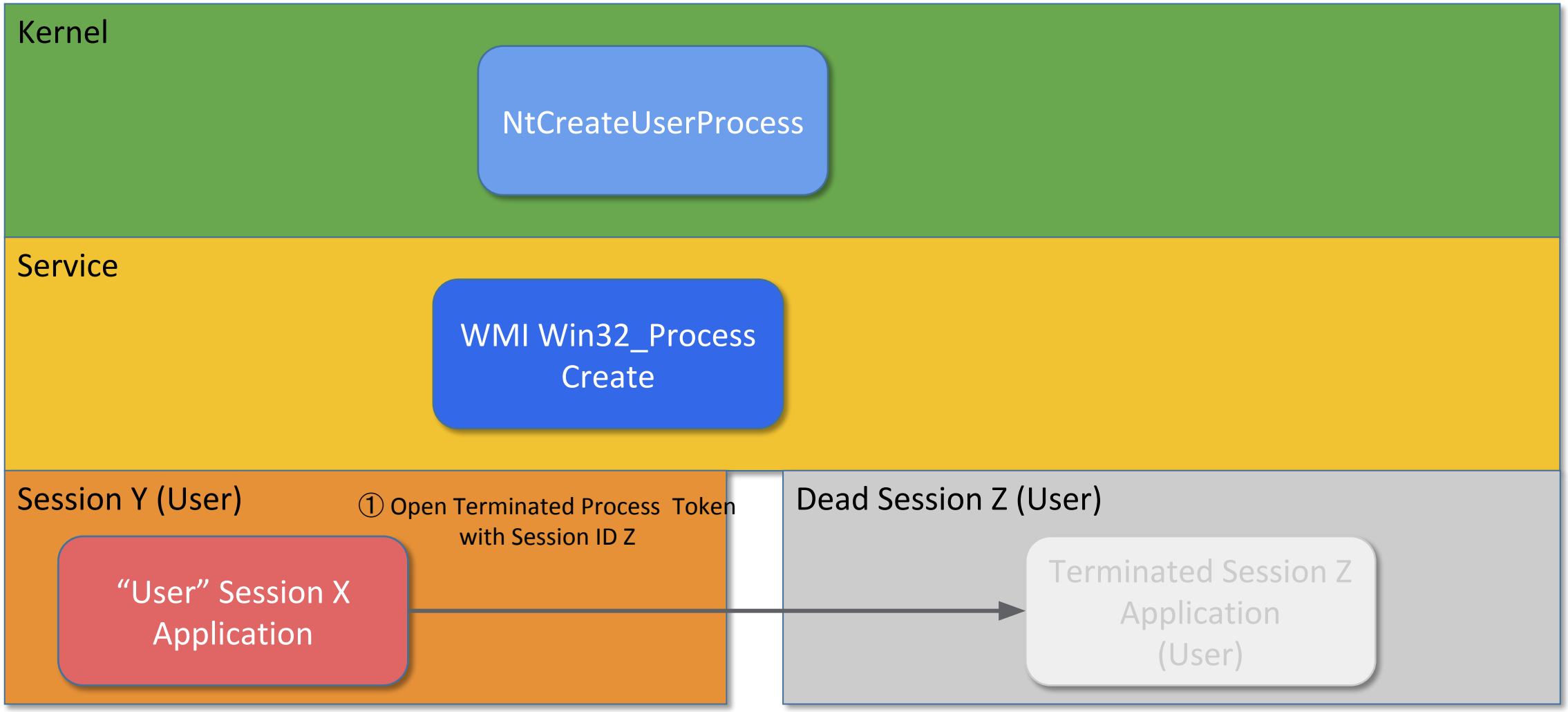
Reference Cycle



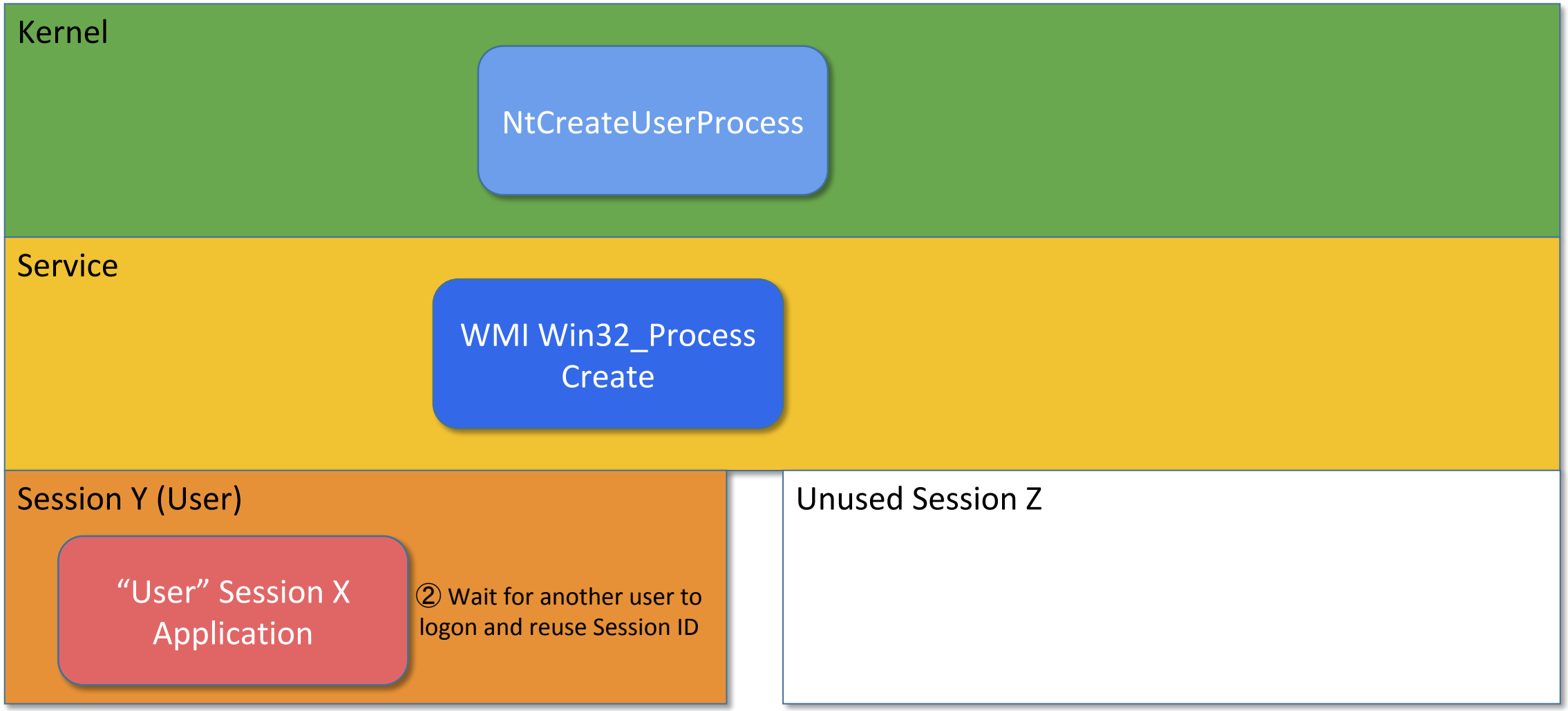
Terminated Reference Cycle



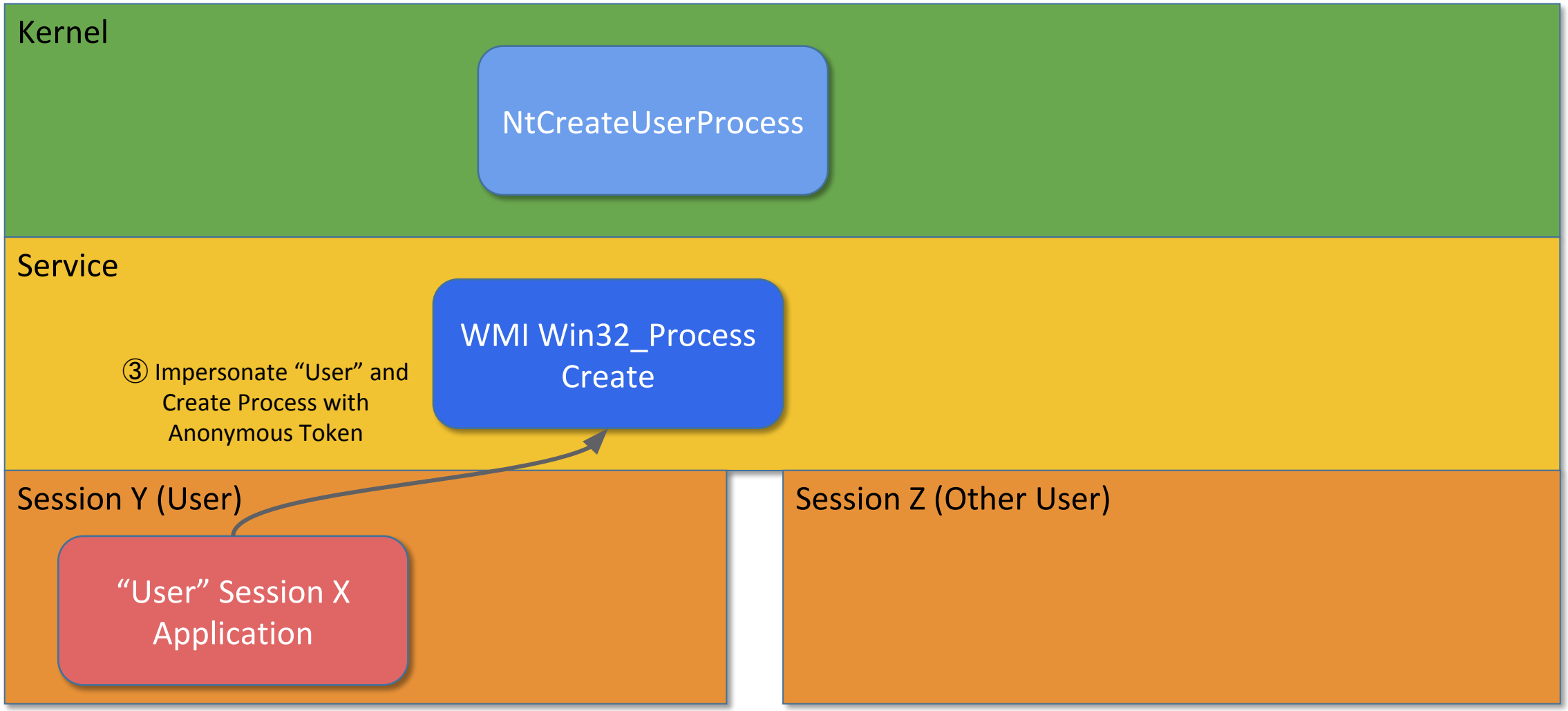
Session ID Use-After-Free



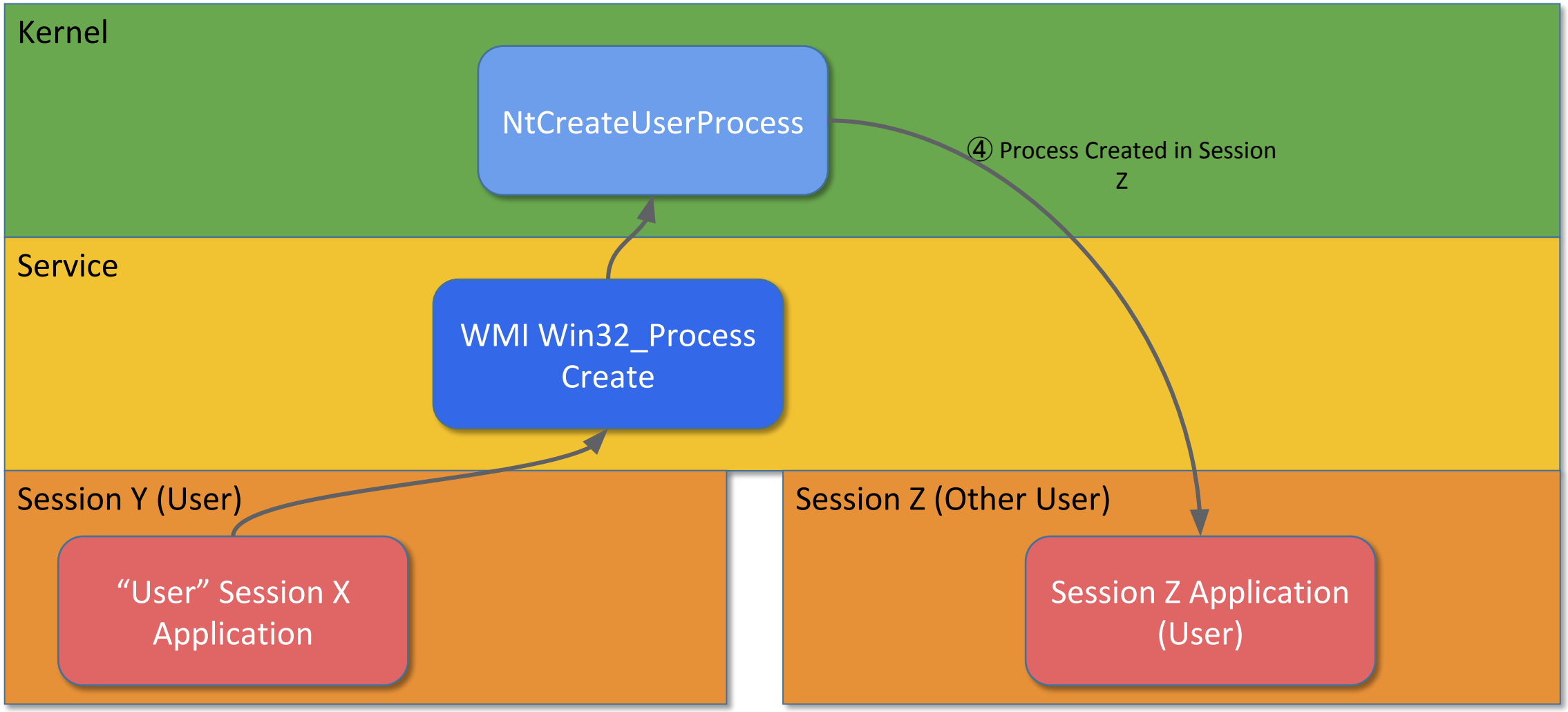
Session ID Use-After-Free



Session ID Use-After-Free



Session ID Use-After-Free



Bonus Material

Tricks to confuse the incident responders

Fooling WMI

- WMI is getting a resurgence, people using for all sorts of things including remote scanning of computers for malicious code
- Commonly enumerates *Win32_Process* instances
- Any way of hiding processes from WMI?
  - No, but, where does *Win32_Process* get the path to the process from?

Loading Executable Path

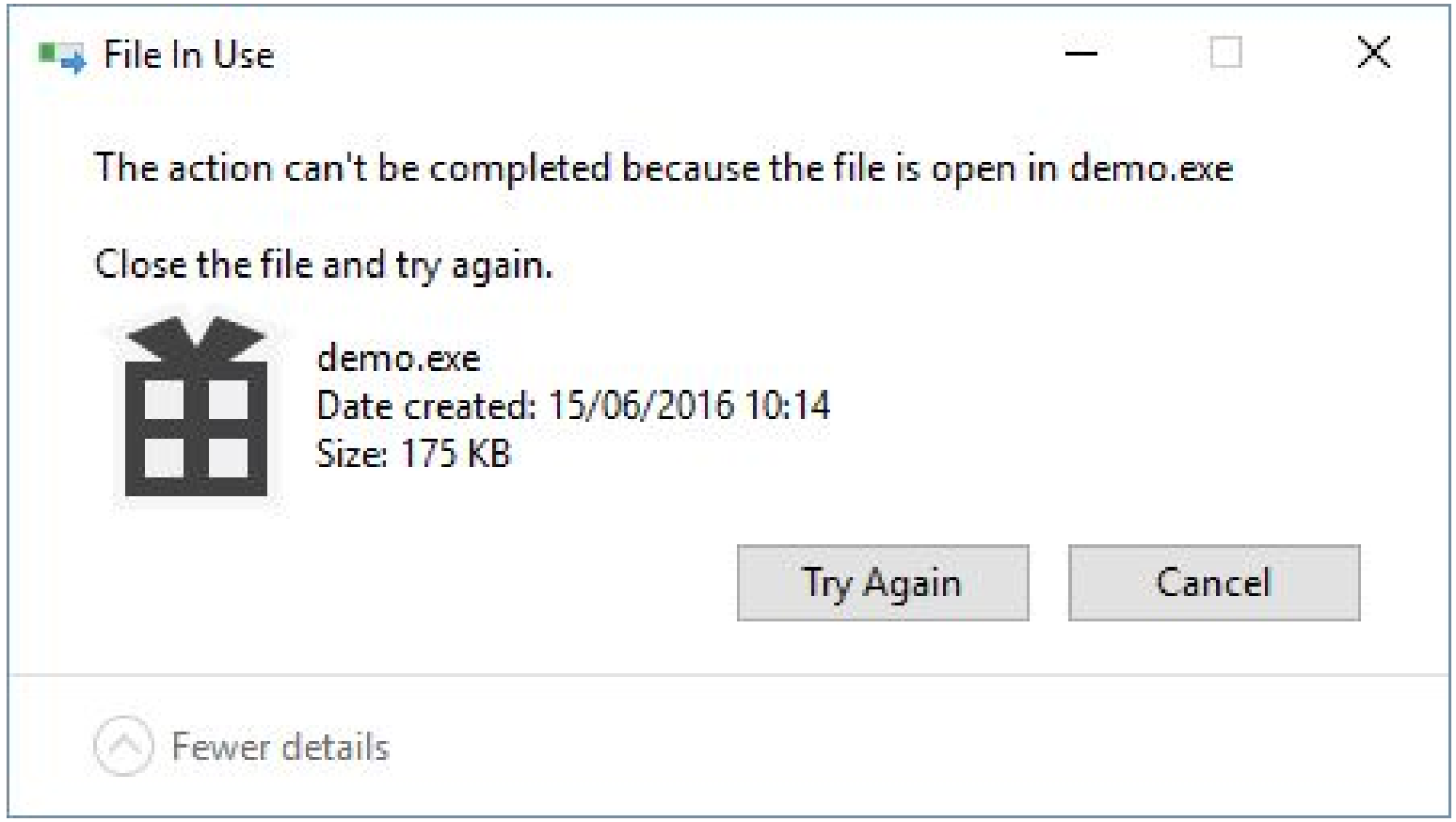
```
Process::LoadCheapPropertiesNT() {
  HANDLE h = OpenProcess(PROCESS_VM_READ |
        PROCESS_QUERY_INFORMATION, FALSE, pid);
  PEB peb;
  ReadPeb(h, &peb);
  RTL_USER_PROCESS_PARAMETERS ProcessParams;
  ReadProcessMemory(h, peb.ProcessParameters,
          &ProcessParams, sizeof(ProcessParams));
  CHString Path;
  GetModuleName(h, &ProcessParams, &Path);
  CInstance::SetWCHARSplat(this, L"ExecutablePath", Path);
  //...
}
```

DEMO

Fooling Win32_Process Enumeration

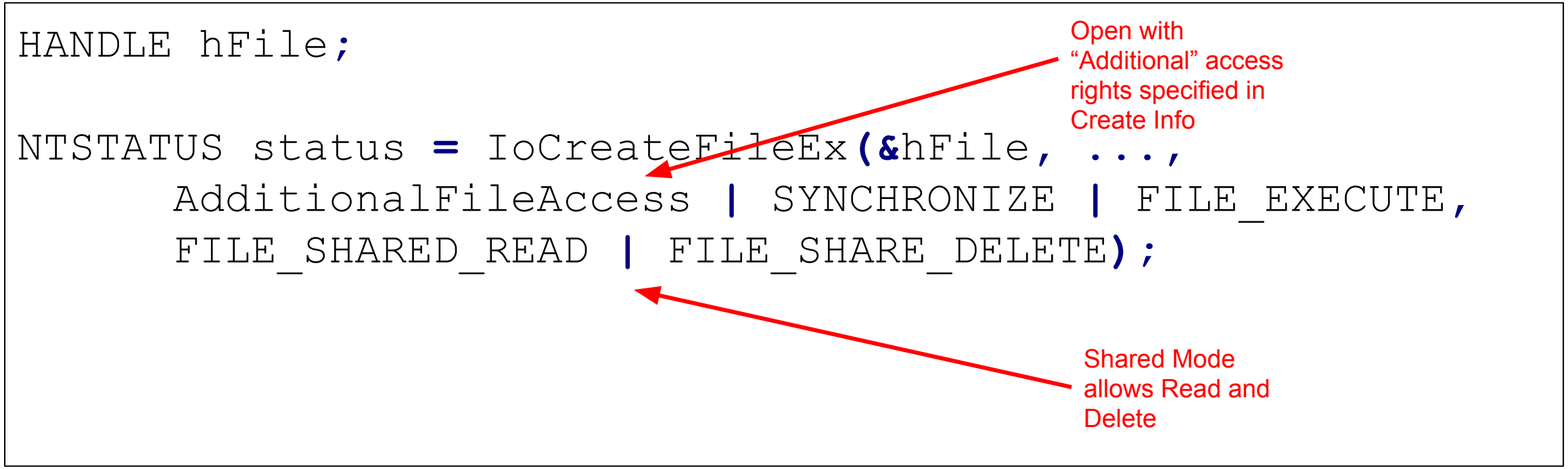
Erasing Our Tracks

- Can’t delete or write to an EXE file once the process is created



Let's Look Back at how NtCreateUserProcess Opens Files

```
HANDLE hFile;

NTSTATUS status = IoCreateFileEx(&hFile, ...,
      AdditionalFileAccess | SYNCHRONIZE | FILE_EXECUTE,
      FILE_SHARED_READ | FILE_SHARE_DELETE);
```



We can use this to open the file for write access.
NtCreateUserProcess will then return the file handle to us!

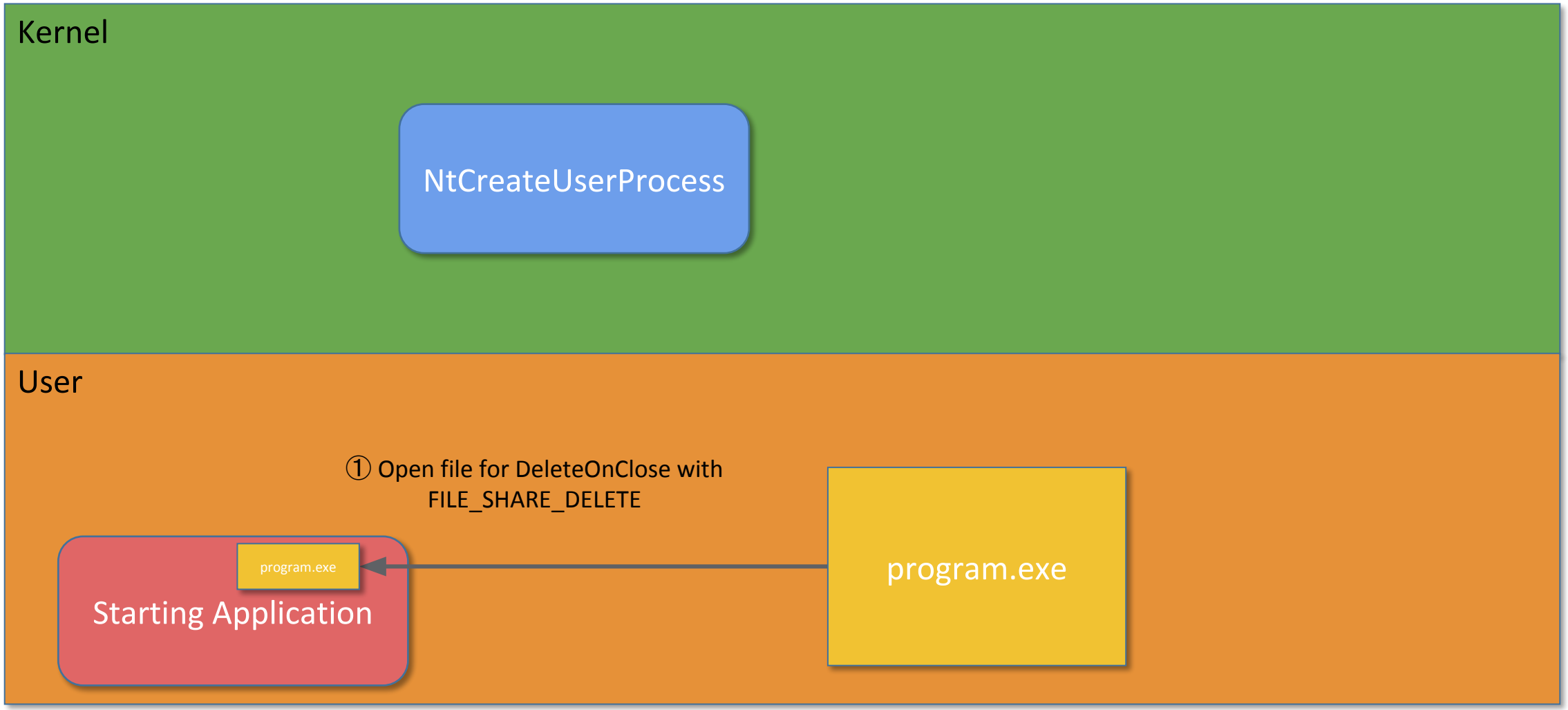
DEMO

Overwriting the Executable File

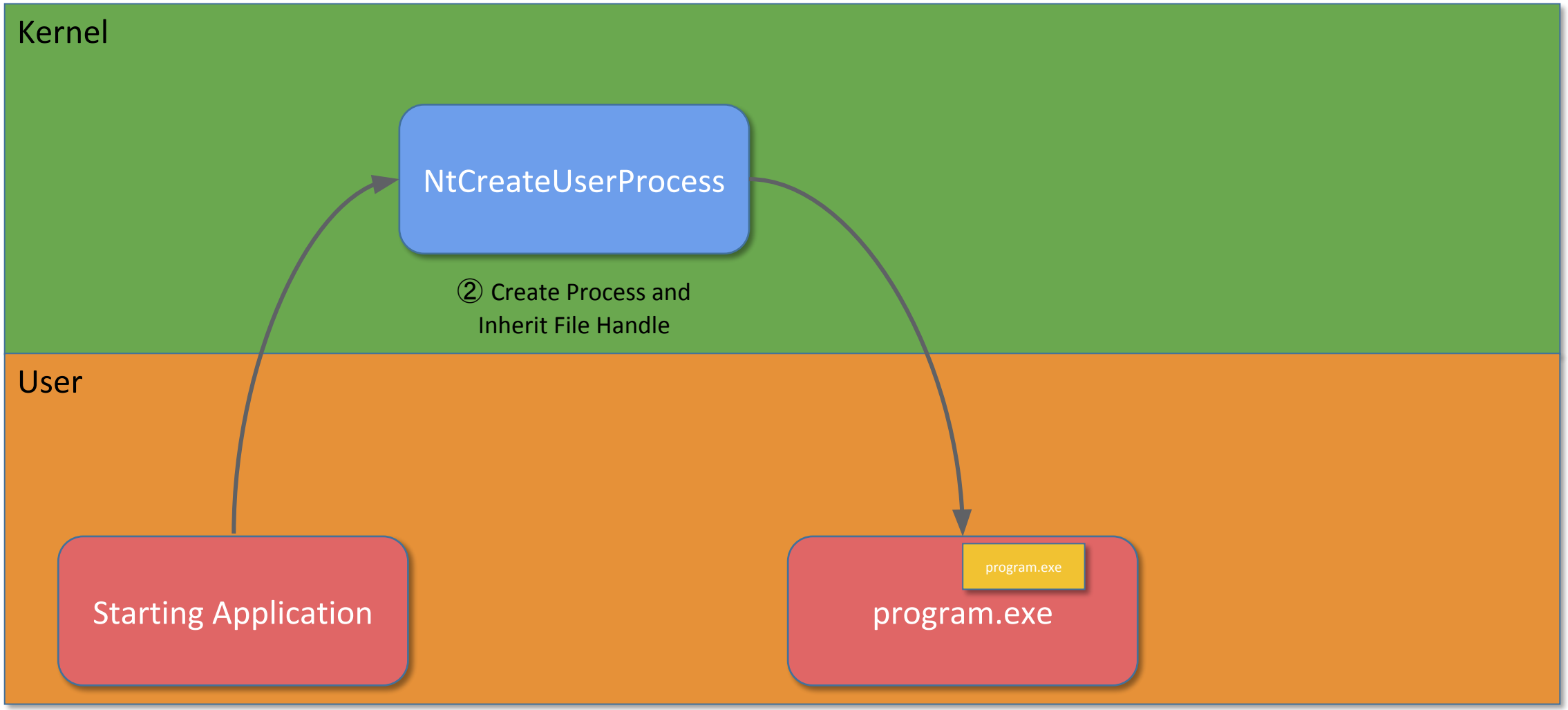
Deleting the File As Well

- File opened with FILE_SHARE_DELETE, surely this means we can delete the file by specifying DELETE for *AdditionalFileAccess*?
- When File object used in image mapped the file is locked
  - Trying to delete the file using NtSetInformationFile disposition information fails on the handle
  - You can open a new handle to the file with DELETE access but gives the same results
- We can the file first and set Delete On Close, however while the image section is valid the delete will fail
- How can we get around this?

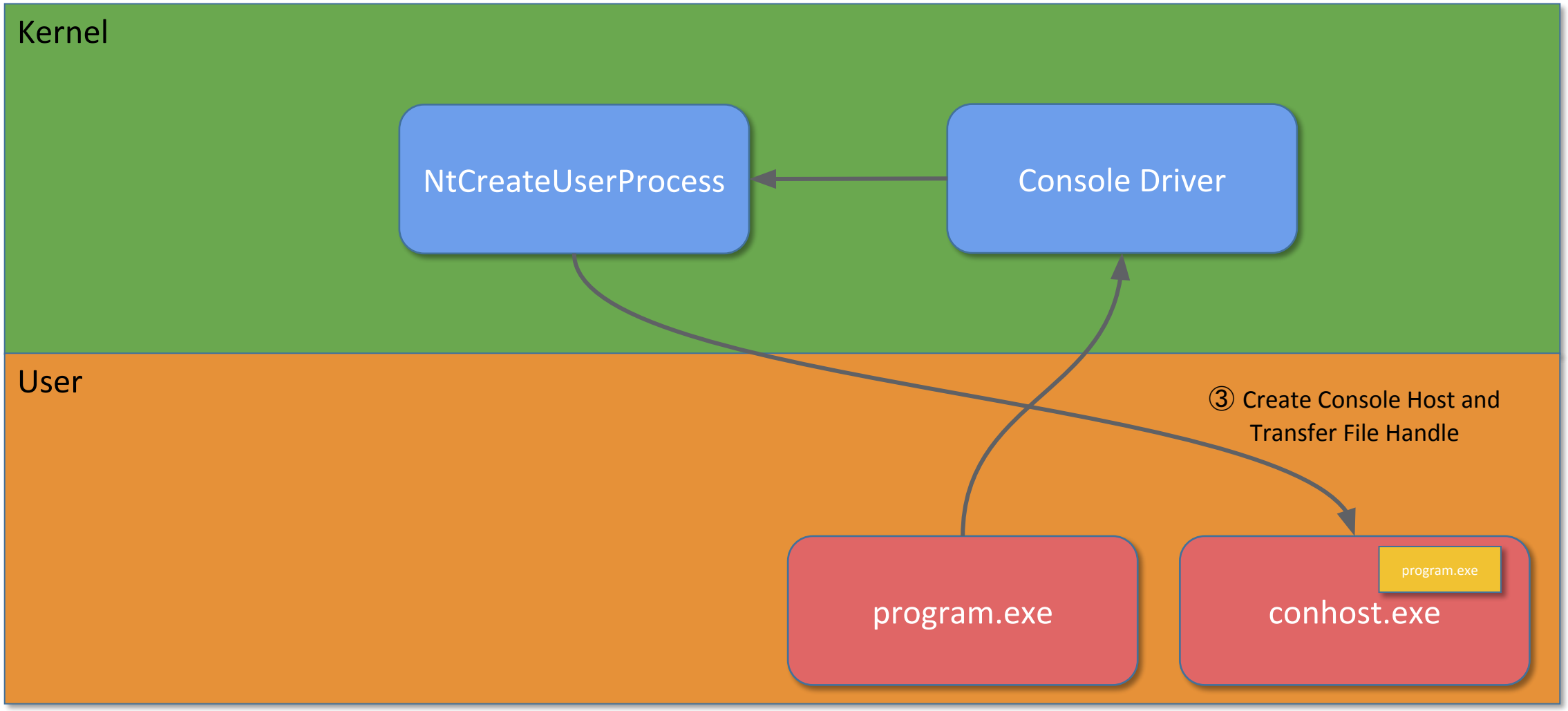
Self Deletion (Kind of anyway, ¯_(ツ)_/¯)



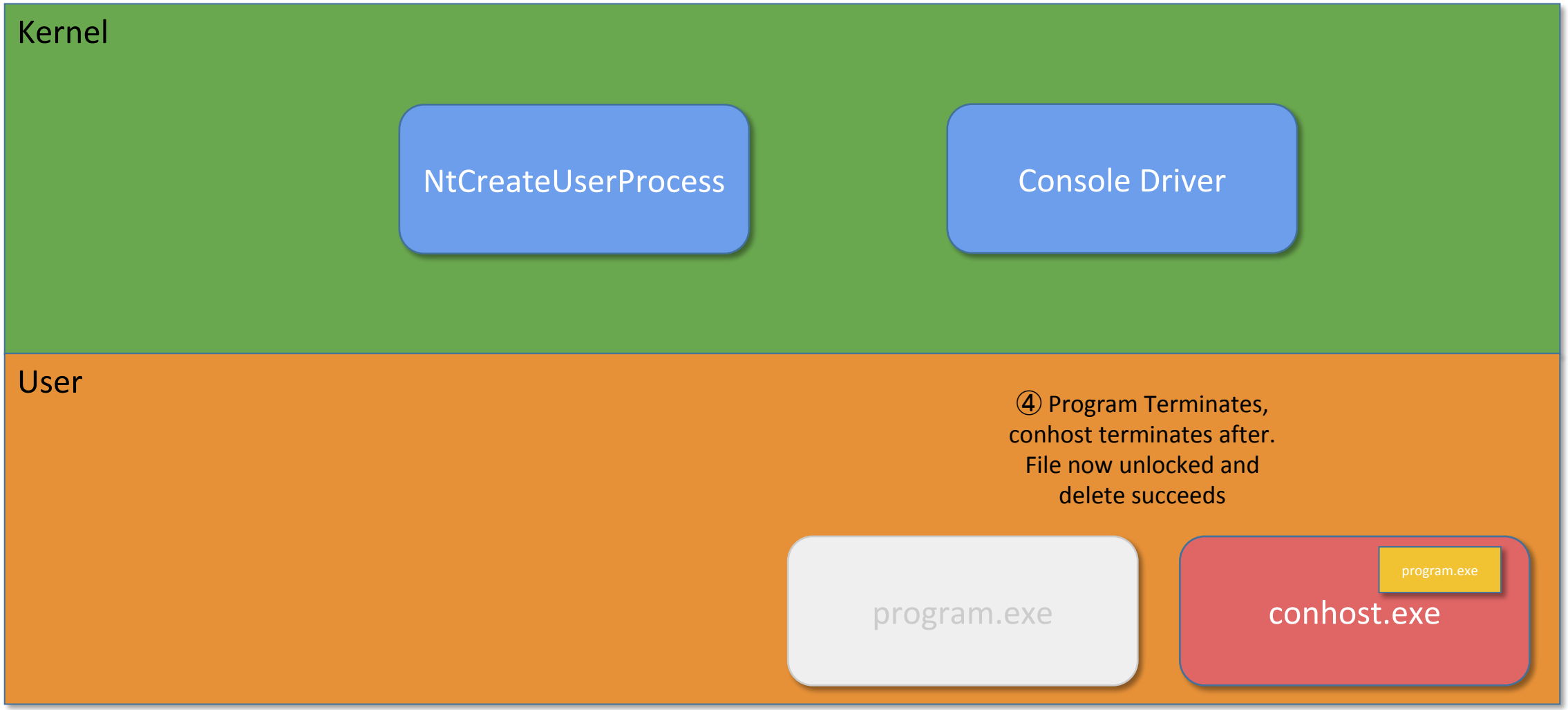
Self Deletion (Kind of anyway, ¯_(ツ)_/¯)



Self Deletion (Kind of anyway, ¯_(ツ)_/¯)



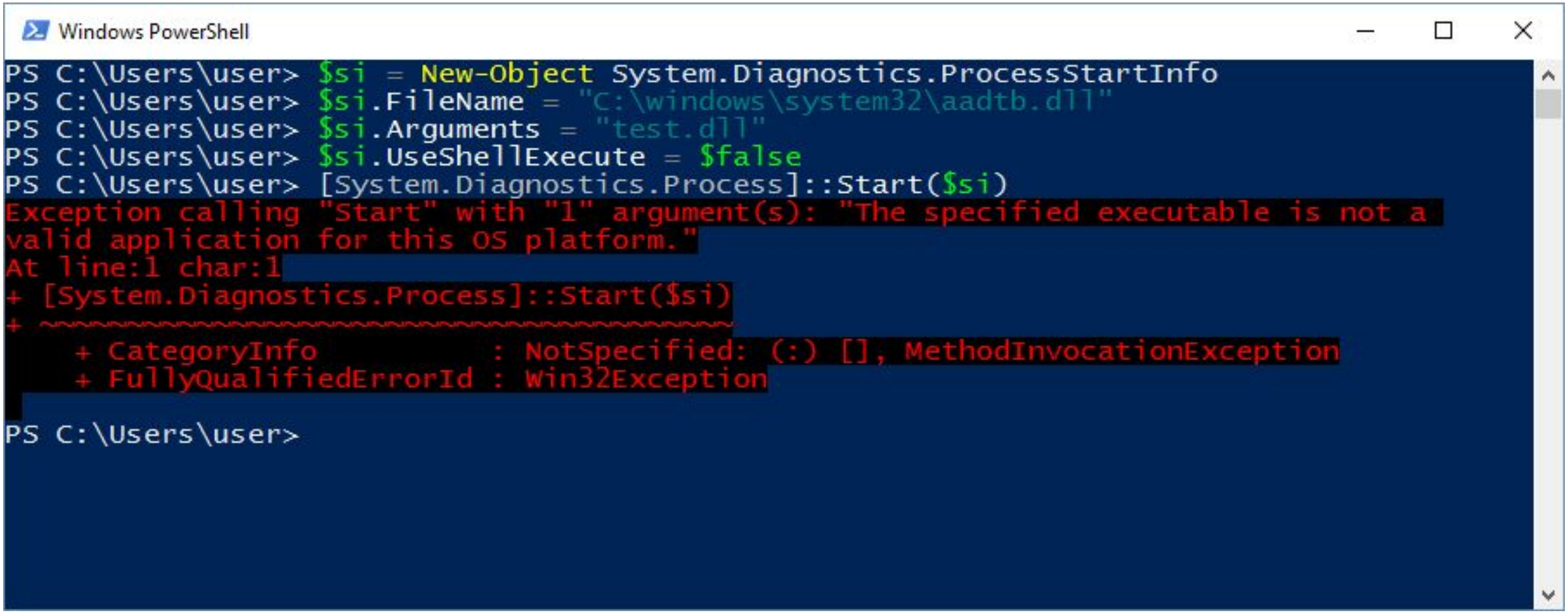
Self Deletion (Kind of anyway, ¯_(ツ)_/¯)



DEMO

Self Deletion Trick

Creating Processes Backed by DLLs



Prohibited Image Characteristics

- NtCreateUserProcess PS_CREATE_INFO supports a setting to automatically reject certain values of the Characteristics Field in the PE Image Header

`WORD SizeOfOptionalHeader`
The size of an optional header that can follow this structure. In OBJs, the field is 0. In executables, it is the size of the IMAGE_OPTIONAL_HEADER structure that follows this structure.
`WORD Characteristics`
Flags with information about the file. Some important fields:

0x0001	There are no relocations in this file
0x0002	File is an executable image (not a OBJ or LIB)
0x2000	File is a dynamic-link library, not a program

```
void CreateProcessInternal () {
  PS_CREATE_INFO CreateInfo;

  CreateInfo.InitState.
ProhibitedImageCharacteristics
                        = IMAGE_FILE_DLL;

  NtCreateUserProcess(..., &CreateInfo);
}
```

DEMO

Creating a DLL as a Process

WRAP UP

Questions?