Shadow Boxing Or How To Fight The New Windows Security Boundary

Mouad Abouhali
(@_m00dy_)

Agenda

>_ UAC

- *>_ Back to the UAC origins : Pre vista Era*
- *>_ Some internals about UAC mechanism*
  - *>_Windows Token*
  - *>_ The Split Token model*
- *>_ Why UAC is not a security boundary*
- *>_ Demo: breaking the UAC*

>_ Windows Administrator Protection

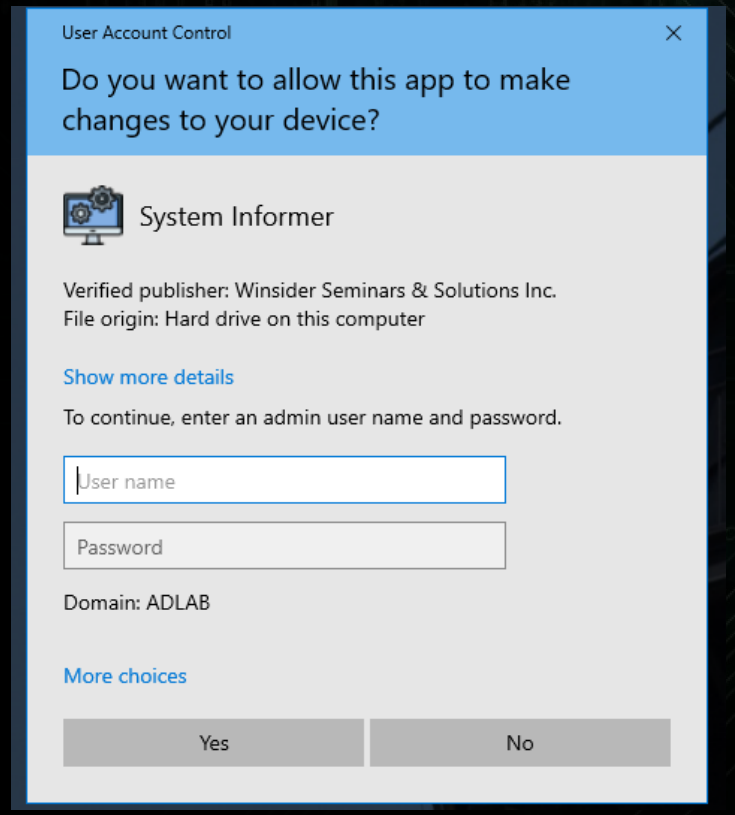
- *>_Windows Administrator Protection: new design*
- *>_ SMAA and its impacts*
  - *>_WAP elevation flow*
  - *>_ Taxonomy a of a WAP Bypass*

THE PRE-VISTA ERA

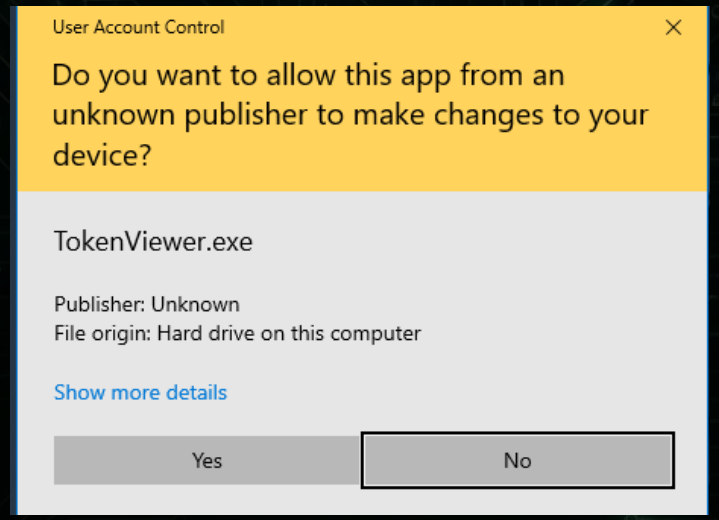
Default High Privileges = High Risk.

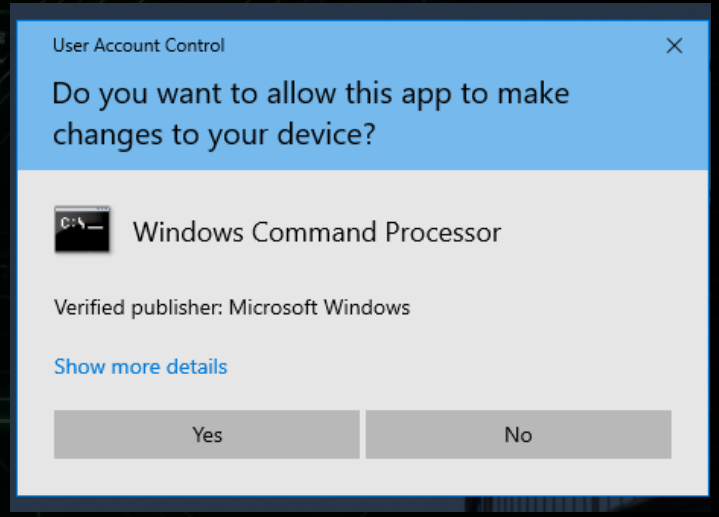
#include "Windows UAC"

</> UAC Elevation: Mechanisms and Trust Models



Over the shoulder (OTS) consent dialog box

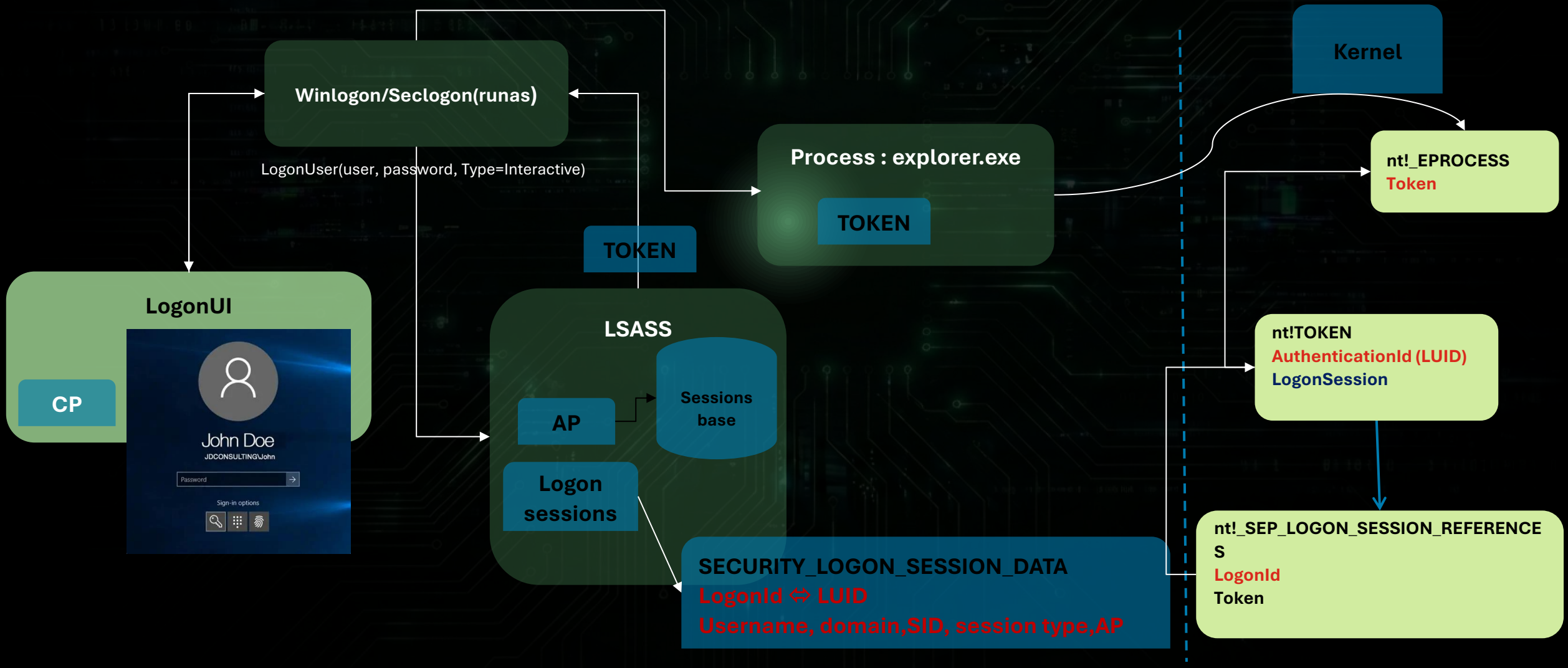




AAC UAC elevation dialog boxes based on image signature

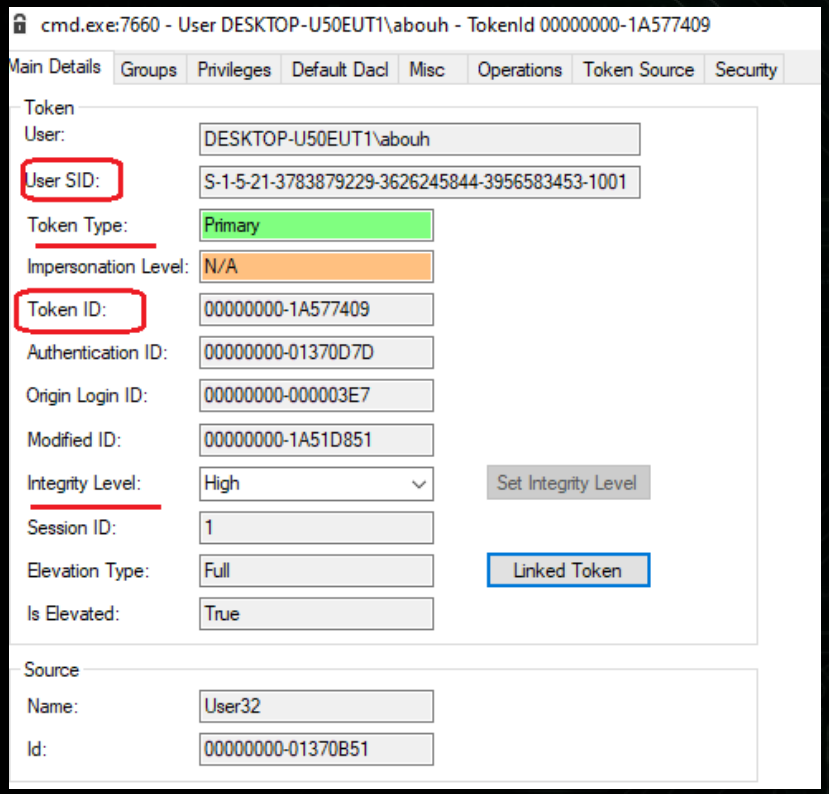
Windows Security Model : TOKEN 1/2

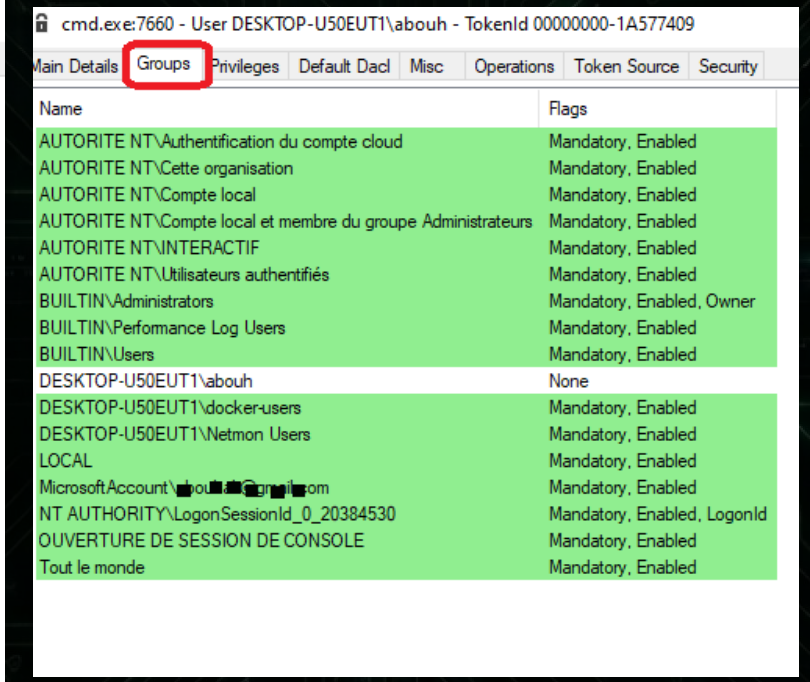
</> Windows Tokens : standard authentication process

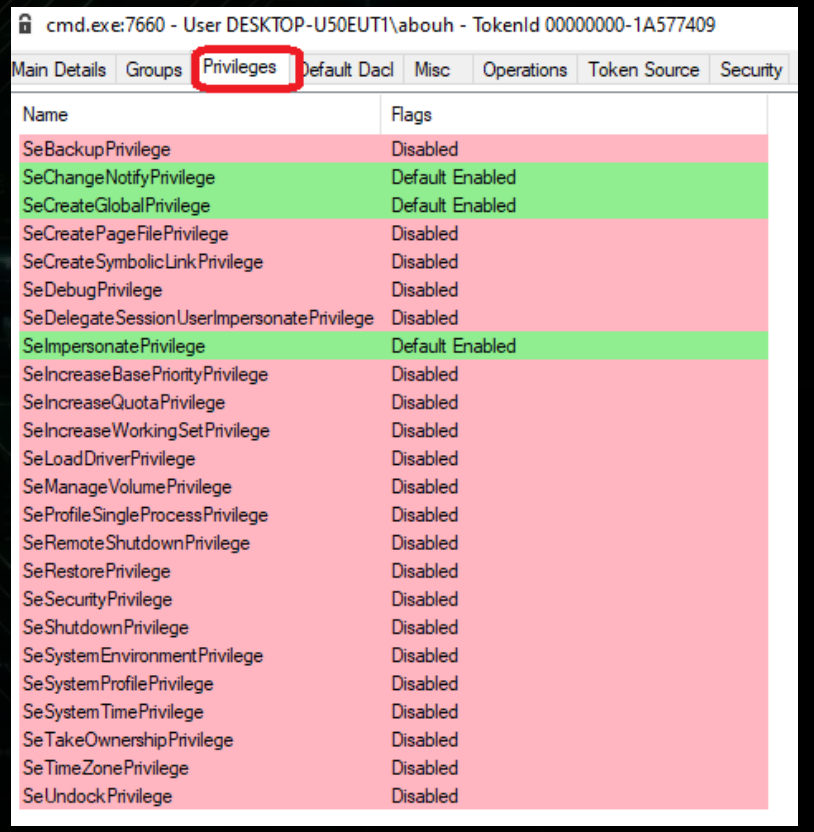


Windows Security Model : TOKEN 2/2

</> Token = Security identity

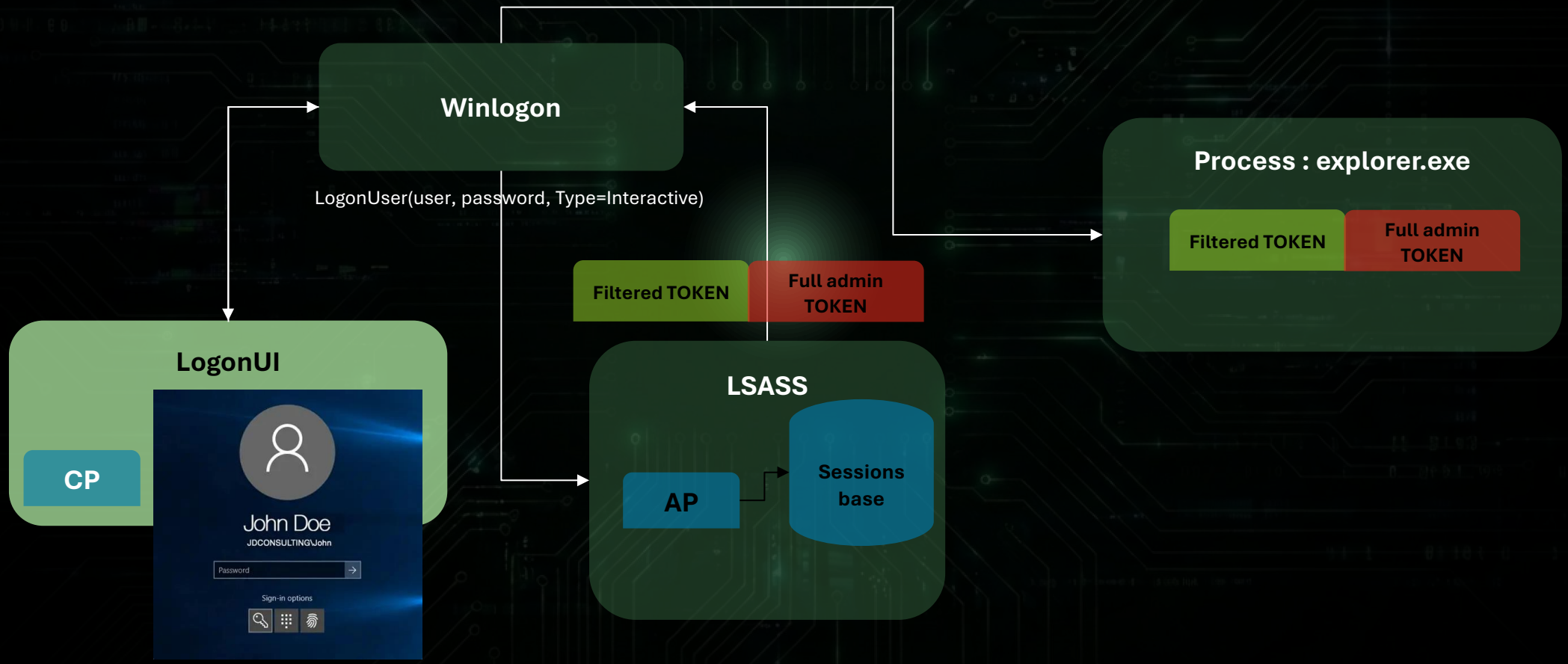






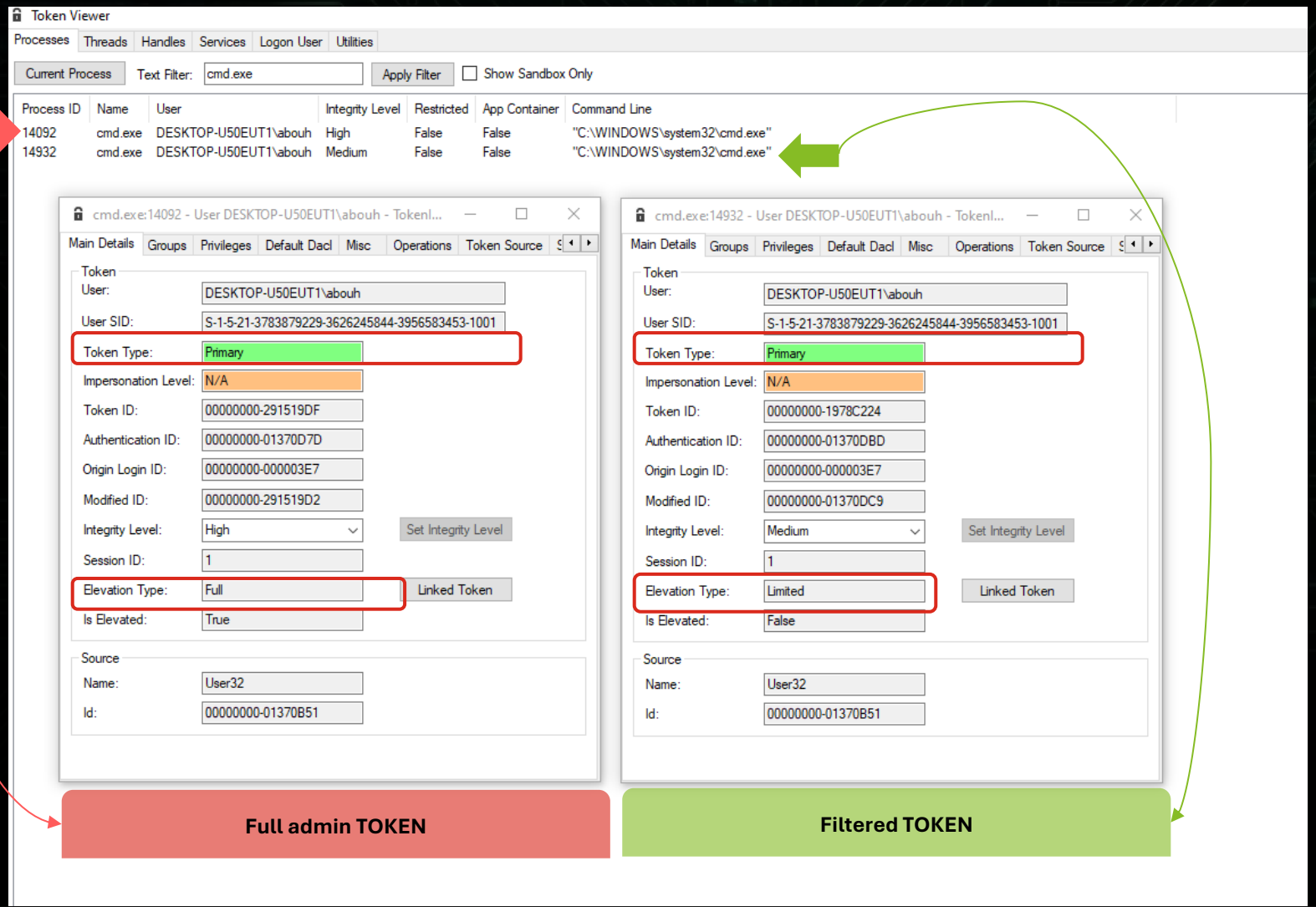
UAC Design and Principles 1/7

</> Windows Tokens : Split-model Token



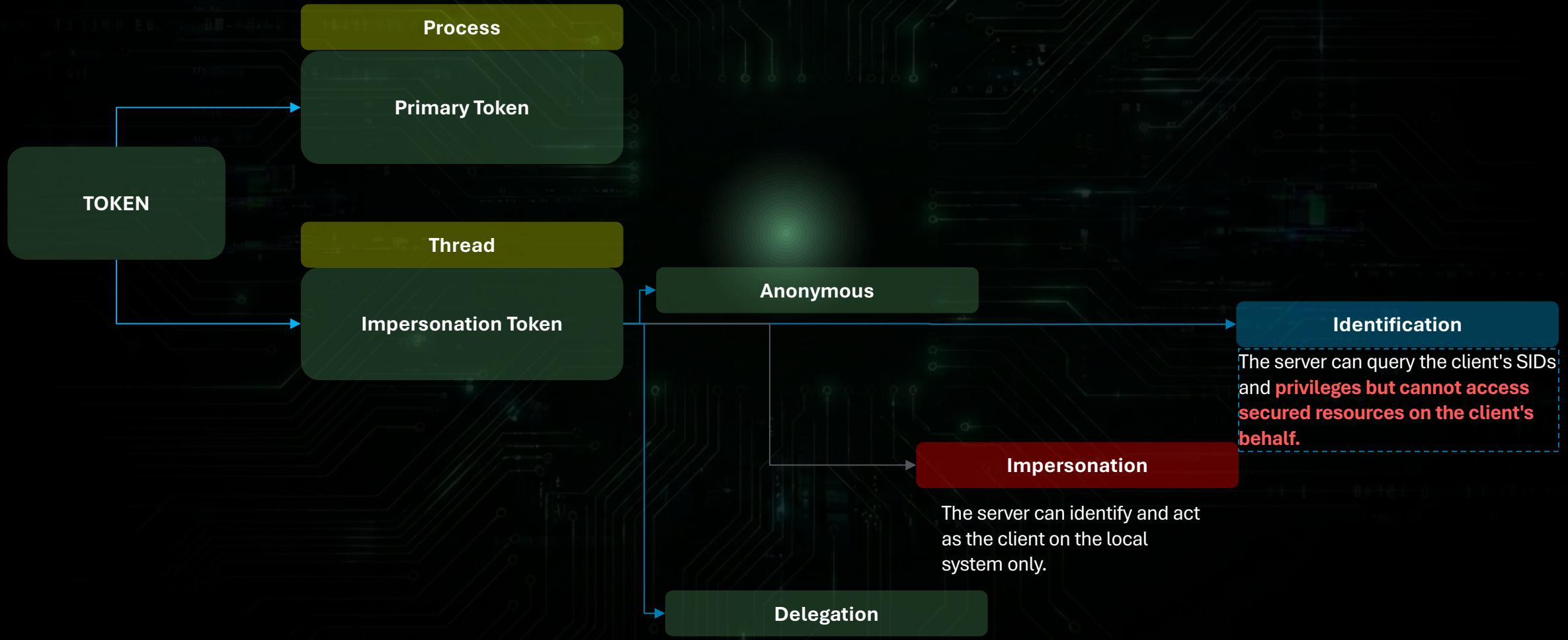
UAC Design and Principles 2/7

</> The Split-Token Administration model: Full vs Limited



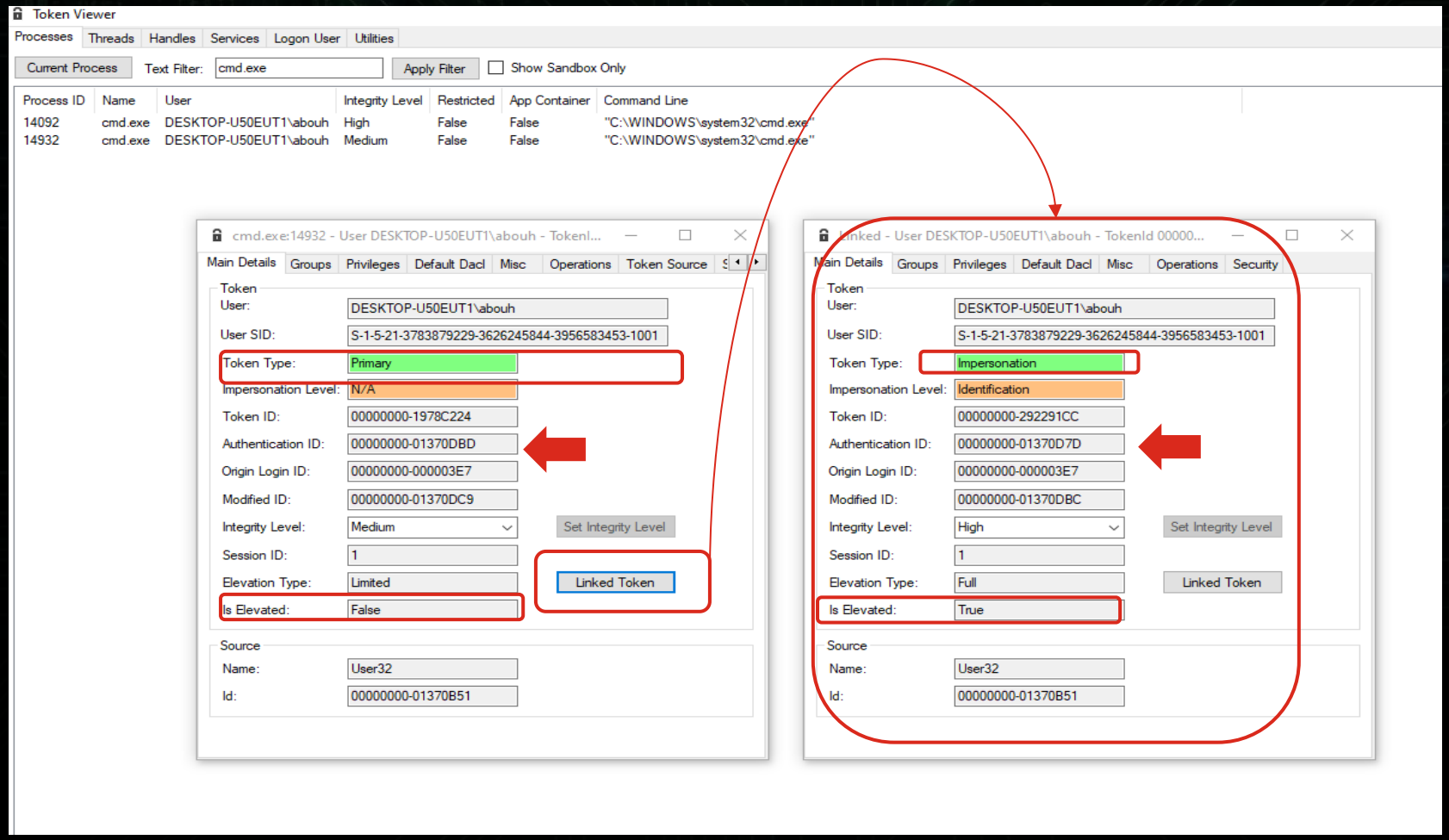
UAC Design and Principles 3/7

</> Tokens type: Primary vs Impersonation



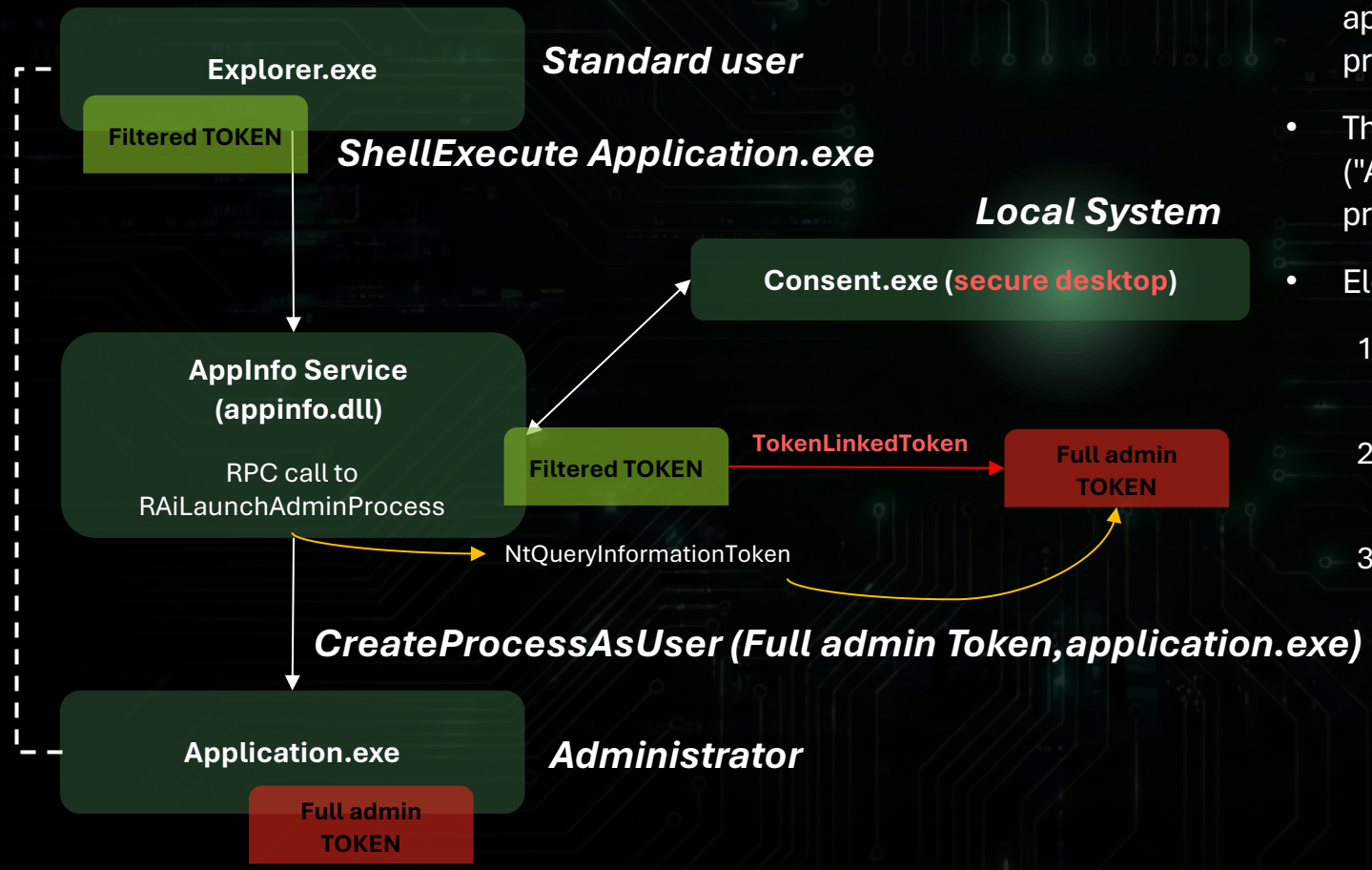
UAC Design and Principles 4/7

</> The Split-Token Administration model: LinkedToken



UAC Design and Principles 5/7

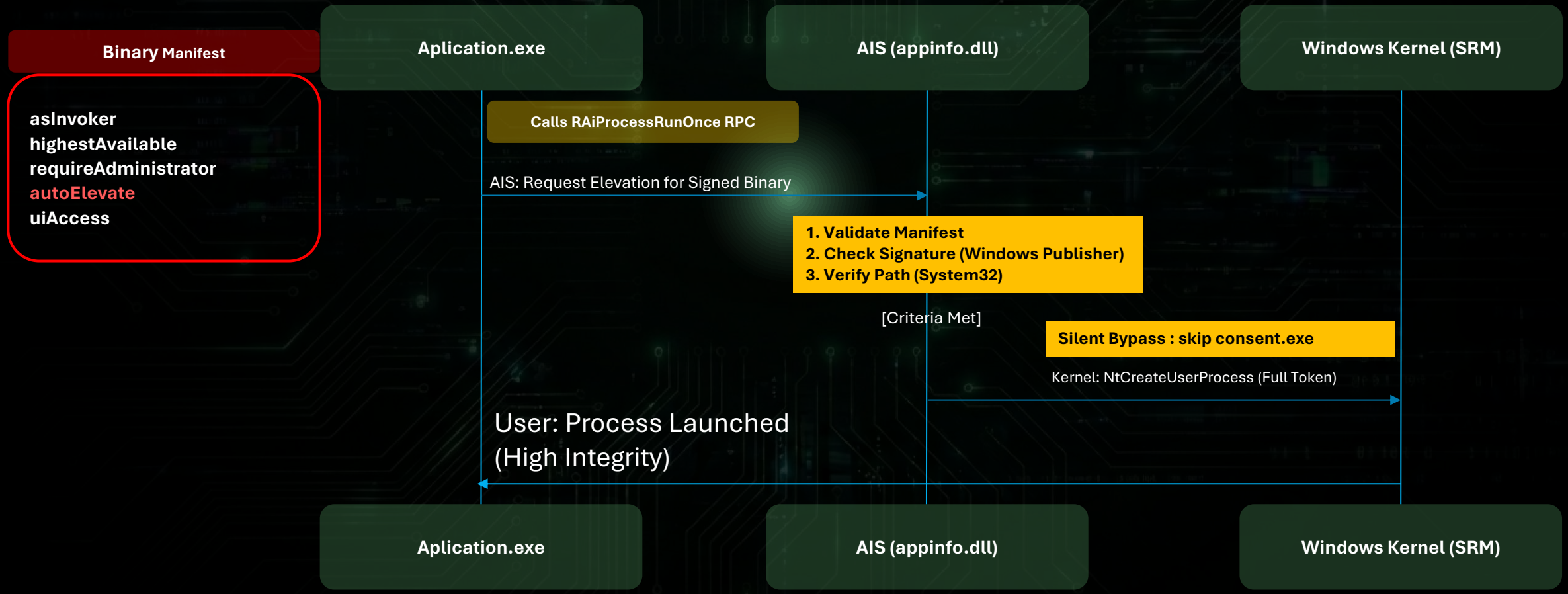
</> UAC Elevation: AIS flow



- The Service: The Application Information Service (AIS / appinfo.dll) runs as SYSTEM inside a svchost.exe process.
- The "Super Power": AIS possesses the **SeTcbPrivilege** ("Act as part of the operating system"), the most powerful privilege in Windows.
- Elevation Orchestration:
  1. When elevation is requested, AIS triggers consent.exe on the Secure Desktop.
  2. Upon approval, AIS uses its **SeTcbPrivilege** to "activate" the linked Full token.
  3. AIS then calls **CreateProcessAsUser** to launch the elevated application and "reparents" it to the original caller (e.g., Explorer)

UAC Design and Principles 6/7

</> Application Manifests and UAC auto elevation



UAC Design and Principles 6/7

</> Application Manifests and UAC elevation

```
C:\SysinternalsSuite>sigcheck.exe -m c:\Windows\System32\taskmgr.exe

Sigcheck v2.90 - File version and signature viewer
Copyright (C) 2004-2022 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\windows\system32\Taskmgr.exe:
        Verified:       Signed
        Signing date:   2:22 PM 8/18/2025
        Publisher:      Microsoft Windows
        Company:        Microsoft Corporation
        Description:    Task Manager
        Product:        Microsoft« Windows« Operating System
        Prod version:   10.0.19041.6280
        File version:   10.0.19041.6280 (WinBuild.160101.0800)
        MachineType:    64-bit
        Manifest:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Copyright (c) Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" xmlns:asmv3="urn:schemas-microsoft-com:asm.v3" manifestVersion="1.0">
<assemblyIdentity
    processorArchitecture="amd64"
    version="5.1.0.0"
    name="Microsoft.Windows.Diagnosis.AdvancedTaskManager"
    type="win32"
/>
<description>Task Manager</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="amd64"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel
                level="highestAvailable"
            />
        </requestedPrivileges>
    </security>
</trustInfo>
<asmv3:application>
    <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
        <dpiAware>true</dpiAware>
        <autoElevate>true</autoElevate>
    </asmv3:windowsSettings>
</asmv3:application>
</assembly>


C:\SysinternalsSuite>
```

UAC Design and Principles : the end !

</> The UAC Design Flaw: Shared Resource Environment

Same-Account Model

- Traditional UAC (Admin Approval Mode) runs both limited and elevated processes under the same user account
- Unlike a true security boundary, there is no isolation between the environment of a "Standard" user and an "Administrator"

Common Profile Resources

- Registry Hive: Both tokens share access to HKEY_CURRENT_USER (HKCU)
- File System: Both tokens share the same User Profile directories (%AppData%, %TEMP%, Documents)

The "Pollution" Attack Vector

- A Medium Integrity process can modify configuration files or registry keys that a High Integrity process will later read and trust
- DLL Planting: Malware can place a malicious DLL in a user-writable directory (like %TEMP%) that a system binary might load upon elevation

Bypassing UAC for fun and profit !

</> DLL planting demo : SilentCleanup

- **Target: The SilentCleanup Scheduled Task:**
  - Location: \Microsoft\Windows\DiskCleanup\SilentCleanup.
  - Privilege Level: Configured to "Run with highest privileges" (High Integrity).
  - The "Silent" Advantage: Because it is a trusted Windows component, it can be triggered by a standard user without a UAC prompt (Auto-elevation).
  - The Vulnerability: Path Hijacking & Shared Resources
  - Shared Environment: Traditional UAC allows Limited and Elevated tokens to share the same User Profile (Registry HKCU and directories like %LocalAppData%).
  - Search Order Exploitation: When triggered, the task's process (e.g., cleanmgr.exe or dism.exe) attempts to load specific DLLs.
  - Environment Overloading: Attackers can manipulate environment variables in HKCU or place a malicious DLL in a user-writable directory that the elevated process trusts and searches first.

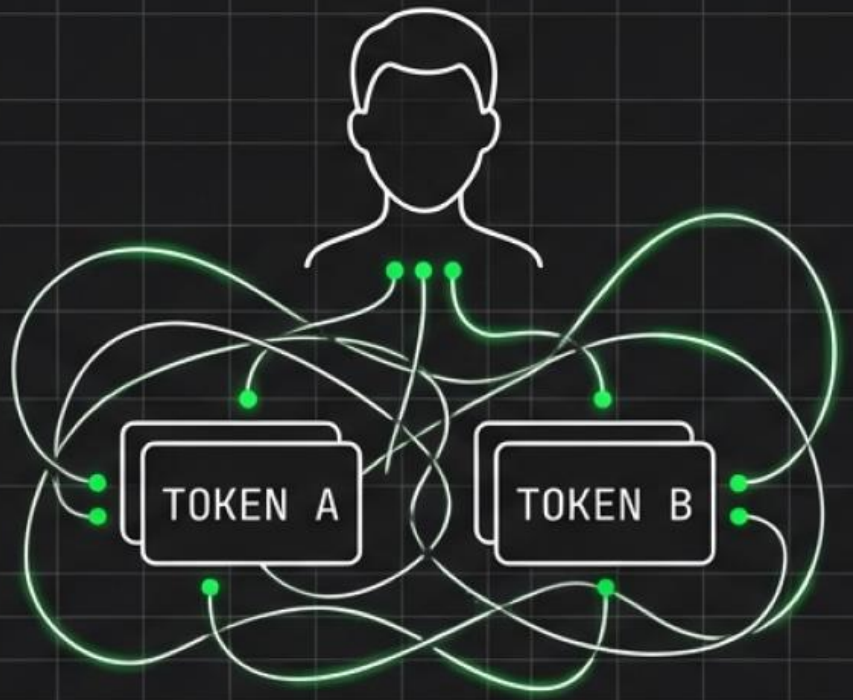
01000100011001010110011001100011011011101101110001000000100011101110010011011110111010101110000001000000100001101100001011100110110000101100010011011000110000101101110011000110100001

DEMO

0100010001100101011001100110001101101111011011100010000001000111011100100110111101110101011100000010000001000011011000010111001101100001011000100110110001100001011011100110001101100001

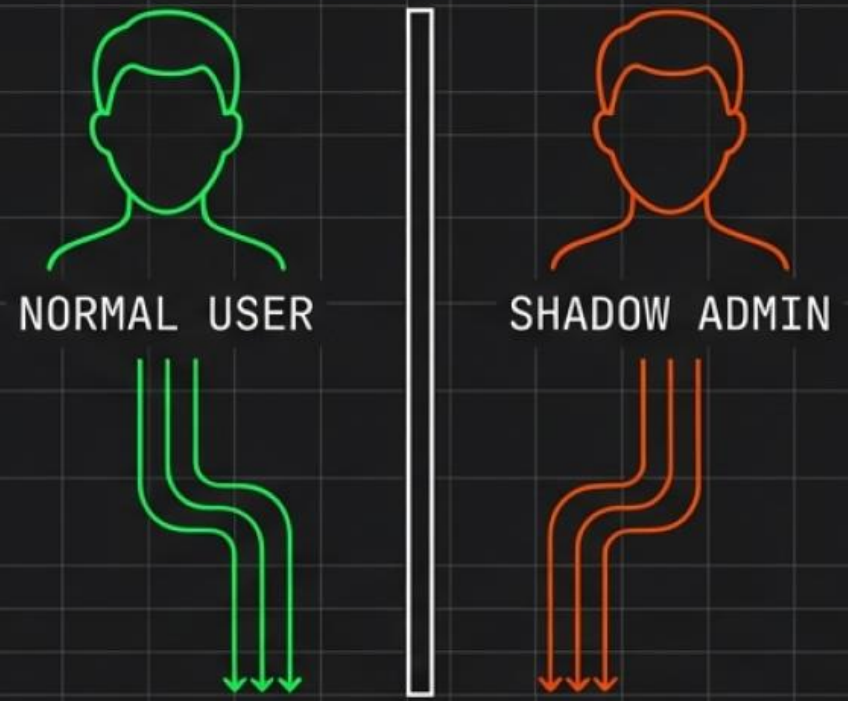
THE PARADIGM SHIFT

UAC (Legacy)



Shared Session

WAP (Modern)



Isolated Session

Windows Administration Protection : new design 1/3

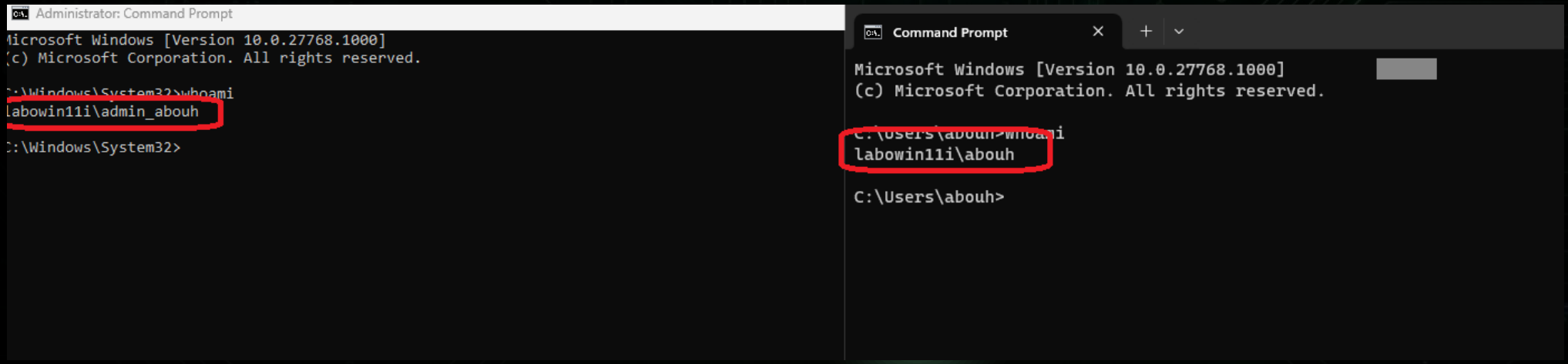
</> Introduction to Windows Administrator Protection (WAP)

- **From Convenience to Security Boundary:**
  - ✓ Traditional UAC is a convenience feature where limited and elevated tokens share the same user account and environment.
  - ✓ WAP transforms elevation into a **hard security boundary**, isolating administrative tasks from the standard user environment.

- **Three main principles :**
  - ➢ **The SMAA Mechanism (System Managed Administrator Account)**
  - ➢ **Isolated User Profile**
  - ➢ **Mandatory Windows Hello Confirmation**
    - – WAP removes "silent" auto-elevations.
    - – Every elevation request requires an interactive confirmation via Windows Hello (PIN, Biometrics),

Windows Administration Protection : let me duplicate everything!

</> Technical Anatomy of the Shadow Admin Account



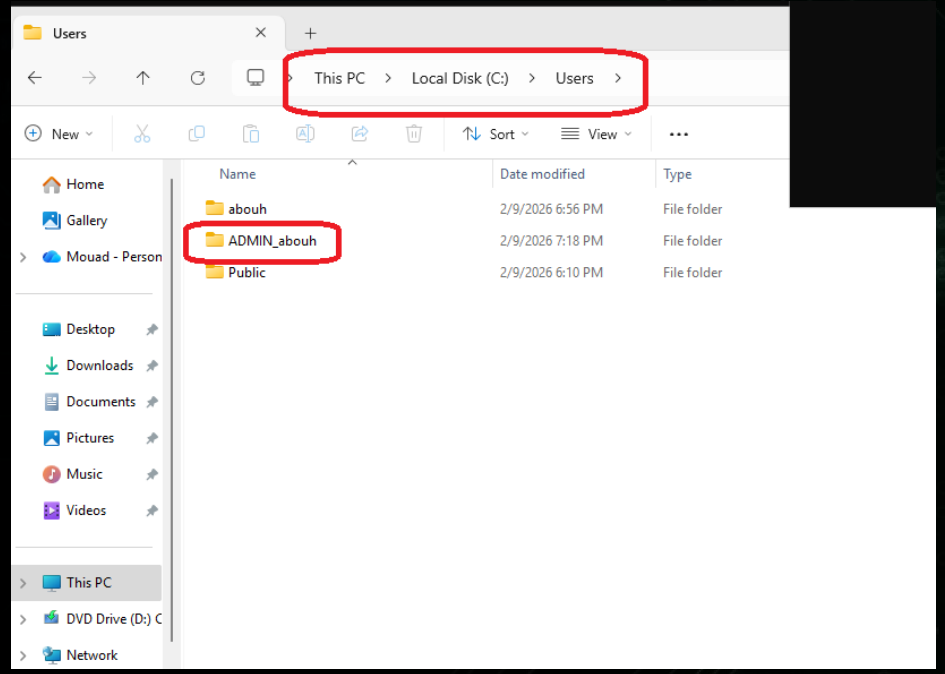
The SMAA Mechanism (System Managed Administrator Account):

Elevated processes no longer run under the user's identity.

They are executed by a System Managed Administrator Account (SMAA), a dedicated virtual account created for the administrative task.

Windows Administration Protection : let me duplicate everything!

</> Technical Anatomy of the Shadow Admin Account



Isolated User Profile:

- **Strict Separation**: The SMAA uses an isolated profile (visible as ADMIN_ folders) instead of the standard user profile.
- **No Resource Sharing**: Registry keys (HKCU) and file system paths (AppData, Temp) of the standard user are invisible to the elevated process, preventing "environment pollution" attacks like DLL planting or registry hijacking [History].

Windows Administration Protection : new design 1/3

</> Introduction to Windows Administrator Protection (WAP)

Consent.exe triggers token creation via **LogonUserExExW** using a blank password

Phase 1: Request

Elevation Request

Windows Hello validation

approved

Phase 2: JIT Creation

LSASS Validation: lsasrv.dll (function **LsapCanLogonShadowAdmin**) enforces that the caller must have SYSTEM integrity and match an internal allowlist (currently only Consent.exe).

AIS Requests LSA → **Authenticate New SMAA Instance** → **Create New LOGON SESSION –Unique LUID** → **Generate JIT TOKEN – High Integrity Admin** → **Phase 3**

Orchestrated via samsrv.dll!SamrFindOrCreateShadowAdminAccount.

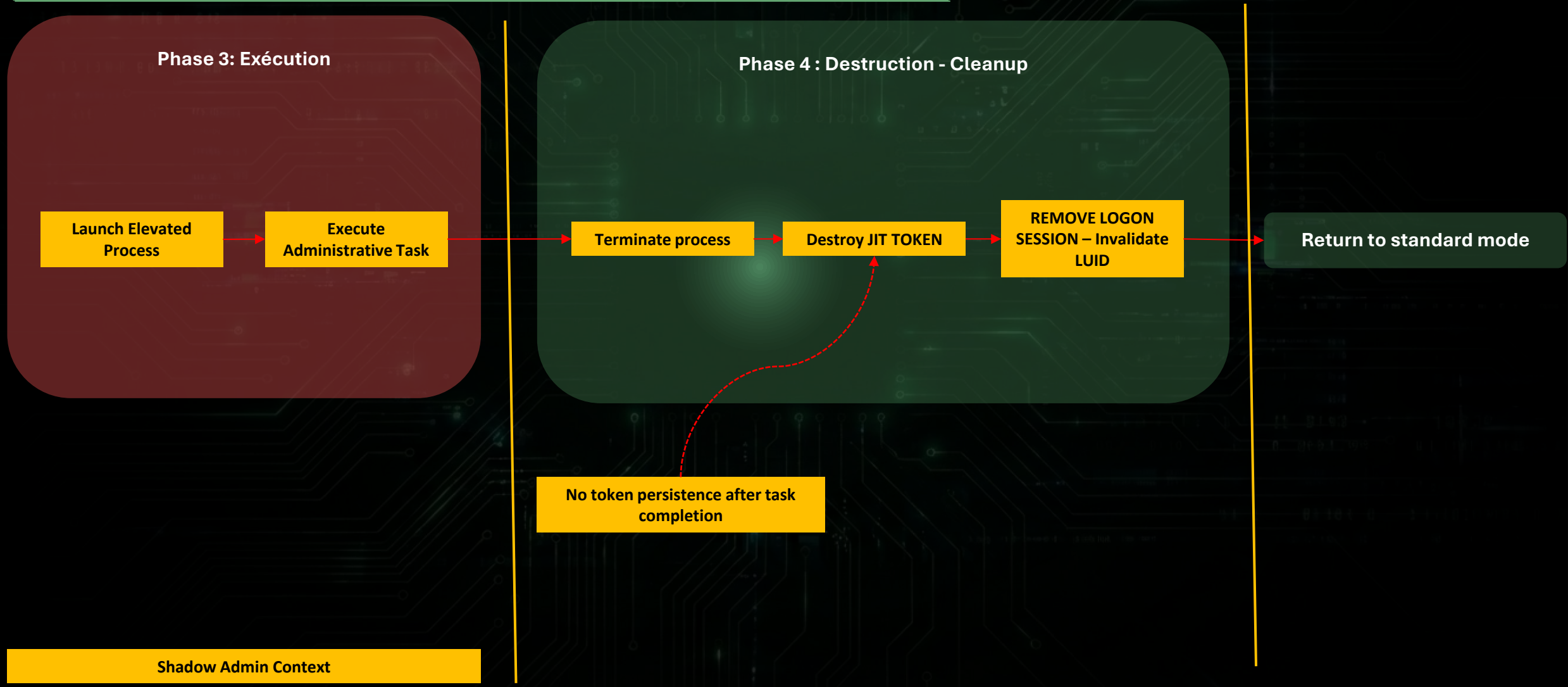
Each New Elevation = New unique LUID

User Context

SYSTEM/LSA context

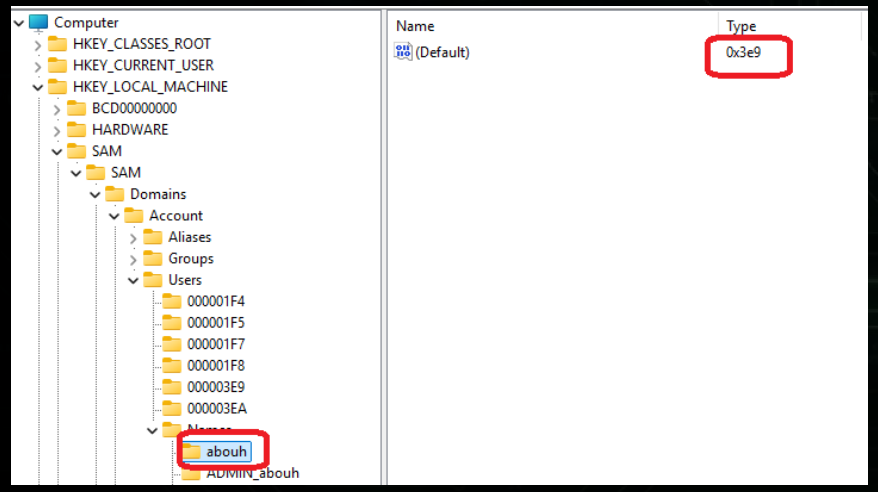
Windows Administration Protection : new design 1/3

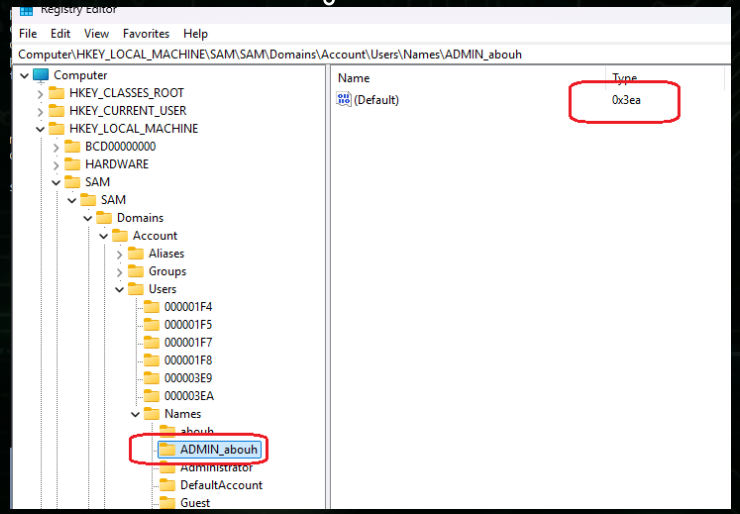
</> Introduction to Windows Administrator Protection (WAP)



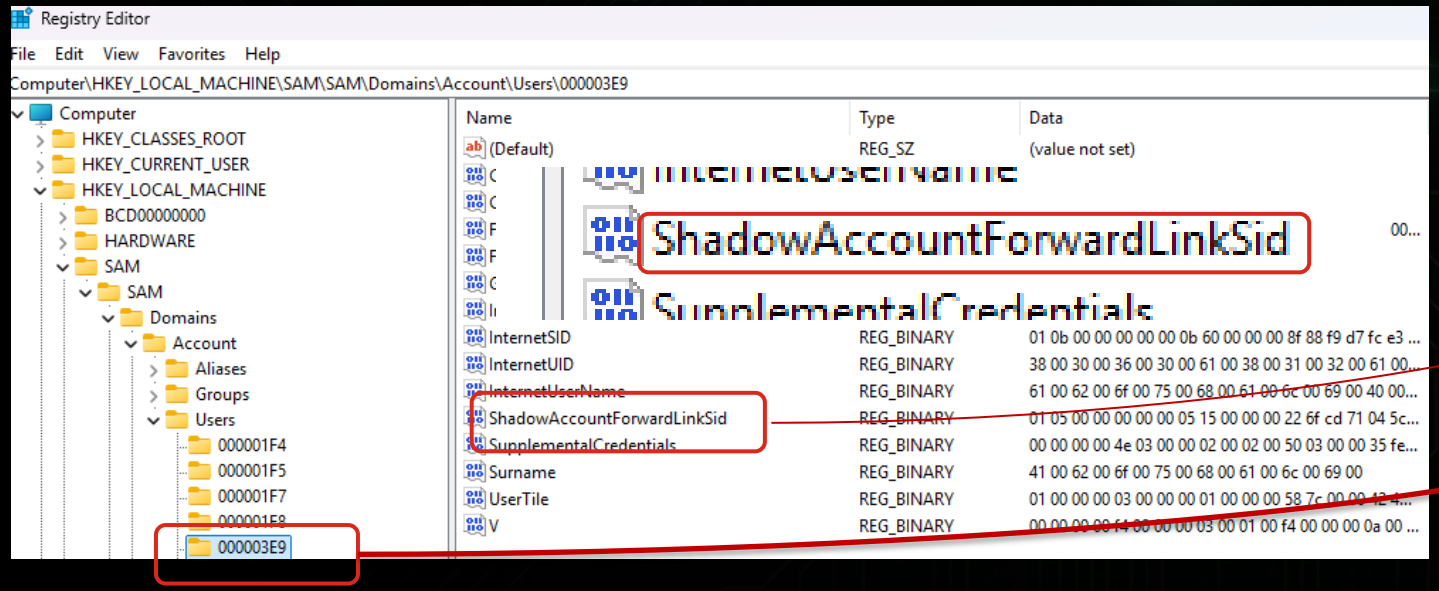
Windows Administrator Protection: Registry artefacts

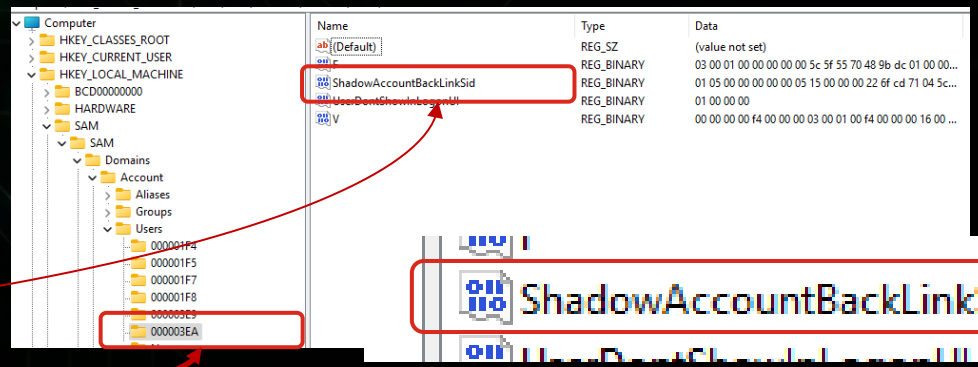
</> Technical Anatomy of the Shadow Admin Account





Registry Linkage: Trust is maintained through reciprocal links in the SAM: ShadowAccountForwardLinkSid (on the user account) and ShadowAccountBackLinkSid (on the shadow account).

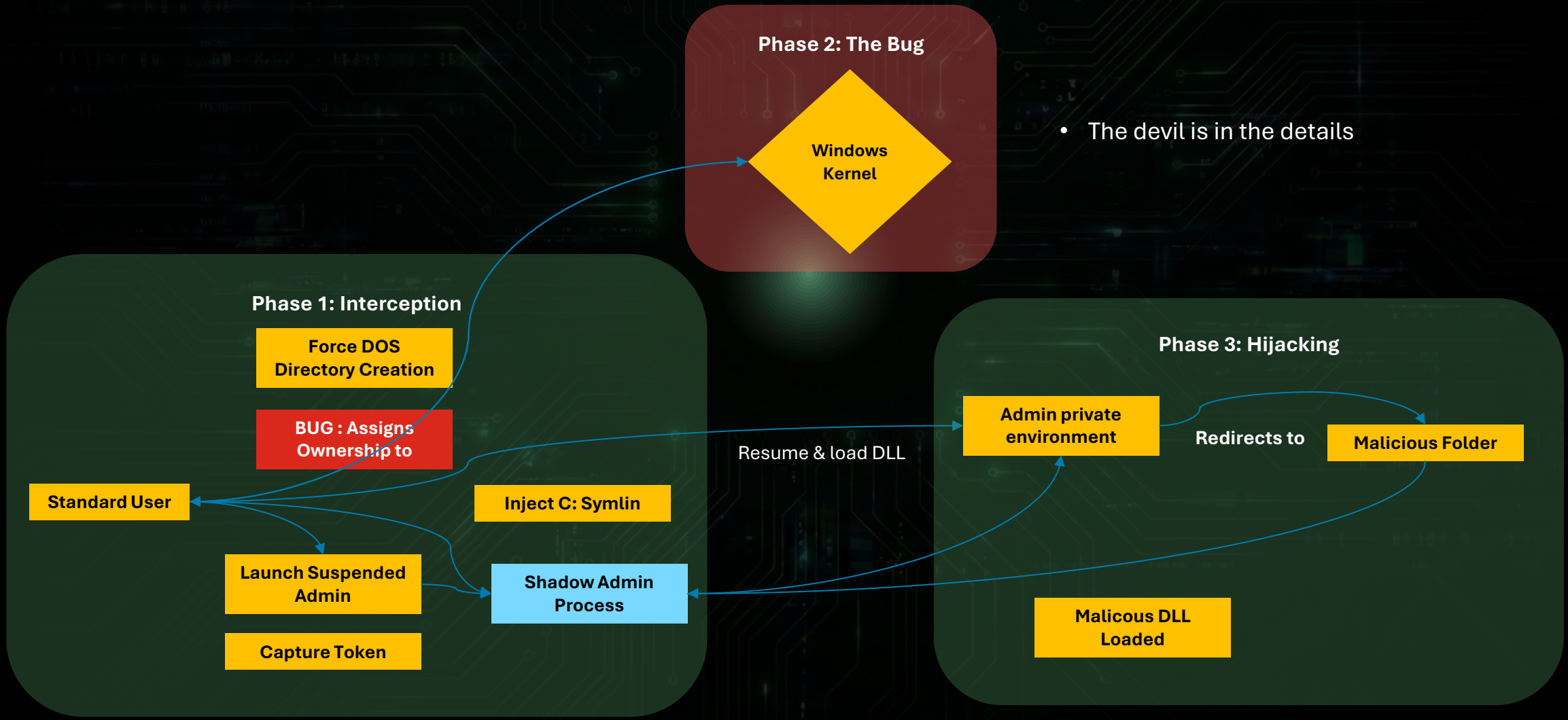




Manipulating these links can potentially redirect elevation to a different shadow account

Windows Administration Protection : Still bypassable ?

</> Taxonomy of a Windows Administrator bypass



- The devil is in the details

01000100011001010110011001100011011011101101110001000000100011101110010011011110111010101110000001000000100001101100001011100110110000101100010011011000110000101101110011000110100001

DEMO