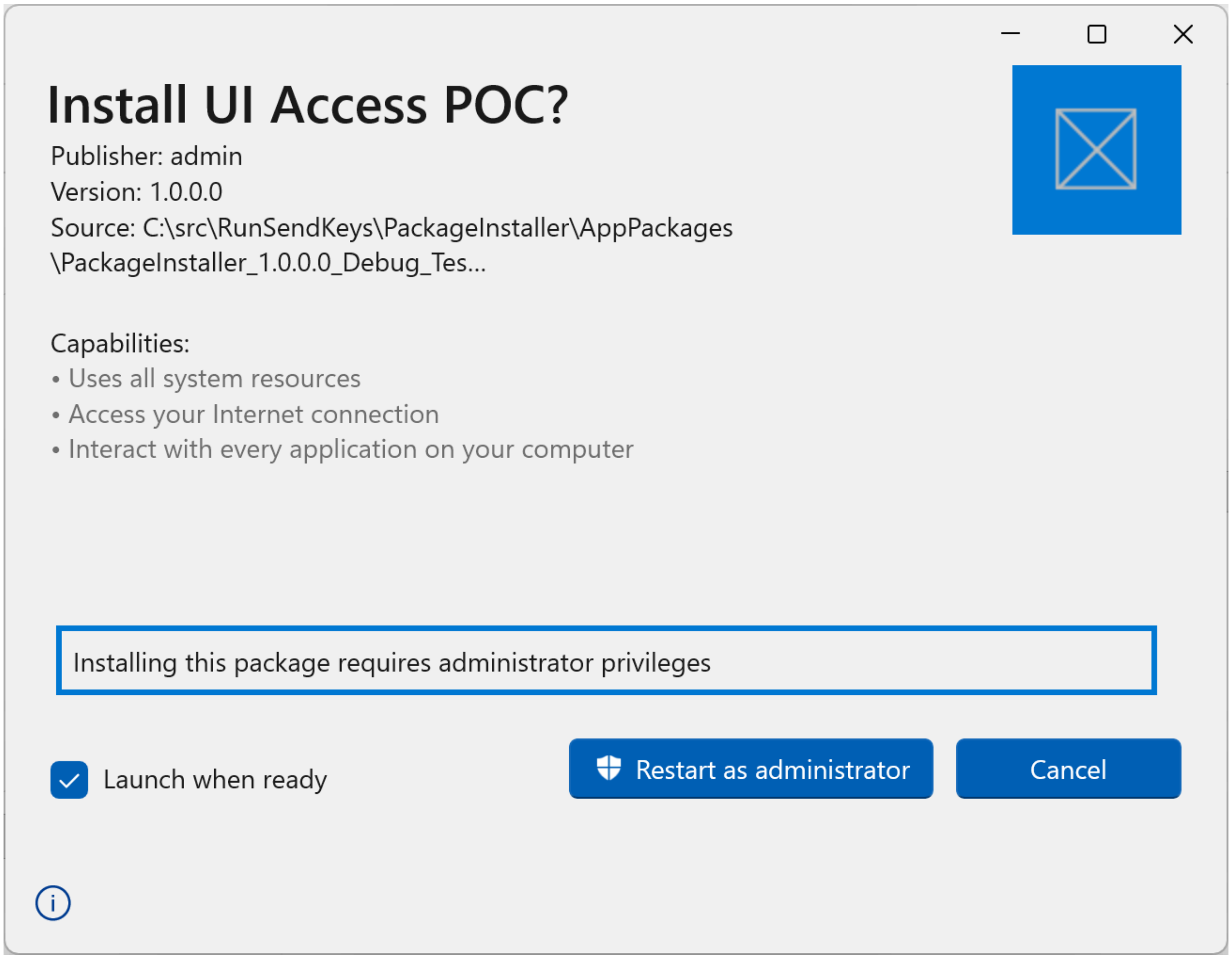
Bypassing Administrator Protection by Abusing UI Access

projectzero.google/2026/02/windows-administrator-protection.html



Install UI Access POC?

Publisher: admin

Version: 1.0.0.0

Source: C:\src\RunSendKeys\PackageInstaller\AppPackages \PackageInstaller_1.0.0.0_Debug_Tes...

Capabilities:

- Uses all system resources
- Access your Internet connection
- Interact with every application on your computer

Installing this package requires administrator privileges

In my last blog post I introduced the new Windows feature, Administrator Protection and how it aimed to create a secure boundary for UAC where one didn't exist. I described one of the ways I was able to bypass the feature before it was released. In total I found 9 bypasses during my research that have now all been fixed.

In this blog post I wanted to describe the root cause of 5 of those 9 issues, specifically the implementation of UI Access, how this has been a long standing problem with UAC that's been under-appreciated, and how it's being fixed now.

A Question of Accessibility

Prior to Windows Vista any process running on a user's desktop could control any window created by another, such as by sending window messages. This behavior could be abused if a privileged user, such as SYSTEM, displayed a user interface on the desktop. A limited user could control the UI and potentially elevate privileges. This was referred to as a Shatter Attack, and was usually fixed by removing user interface components from privileged code.

As UAC encouraged running processes at different privilege levels on the same desktop, Microsoft introduced an additional feature, User Interface Privacy Isolation (UIPI). This used the Mandatory Integrity Control feature in UAC to limit what windows a process could interact with. If the integrity level of a process was lower than the process which created a window then it would be blocked from operations such as sending messages to that window. As an additional protection, Vista no longer ran user processes on the "service" desktop so that even if UIPI was inadequate a user interface exposed by a service process was not accessible to limited processes.

To take an example, a limited user process has an assigned integrity level of "Medium" while a UAC administrator process is "High". In this case UIPI would block the limited user process sending messages to any window created by the administrator process, excluding a small set of explicitly permitted messages. It would also block other UI functionality such as windows hooks.

This introduced a problem for any user who relied on accessibility technology, such as screen readers. If the accessibility process was running as the limited user it could no longer interact with administrator processes created on the desktop. It would be blocked from both reading the contents of windows as well as performing operations such as clicking a button. This was not an acceptable compromise, so Vista needed a way to allow these applications to continue to work.

The solution Microsoft chose was to allocate a flag for the access token of a process called UI Access. If the process' access token had this flag set when it initialized its connection to the Win32 subsystem, the process would be granted special permissions to bypass many of the restrictions imposed by UIPI. Enabling this flag through a call to `NtSetInformationToken` with the `TokenUIAccess` information class was gated behind a check for SE_TCB_NAME privilege, and so it couldn't be performed by a limited user. Therefore in order to create a UI Access capable process a system service was necessary to enable the flag and create the new process.

UAC already needed a system service, so creating a UI Access process was made part of the same flow that was used for launching an administrator process through the `RAiLaunchAdminProcess` RPC call. When a UI Access process is created through this RPC call it does not show the consent prompt unlike administrator elevation. This is important as otherwise there was a risk that a user couldn't create the accessibility application needed by them to click the consent prompt for elevation.

In order to prevent malware just claiming to be an accessibility application the service imposed some additional checks on the executable file which must be met to enable the UI access flag on the new process:

- It must have an embedded manifest with the `uiAccess` attribute set to `true`.
- It must be signed by a code signing certificate that's trusted by the local machine root certificate store. There was no special requirement for the certificate outside of this, for example it doesn't need to have a special EKU or cross signing by Microsoft.
- It must be stored in an administrator only location on the system drive, such as:
  - The `Program Files` directory
  - The `Windows` directory (excluding some known writable locations)
  - The `System32` directory (excluding some known writable locations)

If all the criteria are met then when the process is launched via `RAiLaunchAdminProcess` the service will take a copy of the caller's access token, enable the UI Access flag and increase the integrity level as follows based on the caller:

- If the caller is a limited user of an UAC administrator it will set the integrity level to High.
- If the caller is an administrator then it will set the integrity level to High (normally a no-op).
- If the caller is a normal user, the integrity level is set to the caller's integrity plus 16 up to a maximum of High.

A High integrity level is the absolute maximum allowed to be set, although there exists a higher level, "System" that's reserved for service processes. Also note that the integrity level of the token is not changed if the caller already has the UI Access flag enabled, this is only important for normal users who don't automatically get set to High integrity. One benefit of setting an elevated integrity level is the created process cannot be opened for read or write access by a lower integrity process, preventing a limited user from injecting code into the new process and by extension getting access to the UI Access flag.

As an aside, you can disable the UI Access flag on the token without TCB privilege. A valid UI Access process running as a normal user can "ratchet" itself up to High integrity by clearing the flag on its own token then respawning another copy of itself via the UAC service. As there's 4096 levels between Medium and High that would require calling the UAC service 255 times which is a little on the noisy side but it does work.

Importantly, the UI Access flag only permits bypassing a limited set of operations such as sending window messages to other higher integrity processes. It doesn't permit using things like windows hooks which allow for code injection into a process. Therefore for a UI Access process running as a normal user with integrity level less than High it can interact with a spawned administrator process through messages, but it couldn't do something more invasive like hooking the window message queue.

However, if a limited user creates a UI access process, it would run with a High integrity level and could take over any administrator process that contains a window. A service process with a System integrity level could only be interacted with using windows messages. But there’s no security boundary between an administrator and a system service so this is meaningless in practice.

The end result is having the UI Access flag without having a High integrity level isn’t sufficient to trivially compromise an administrator process, you would need the process to expose a user interface that could be automated to get privileged code to run. For example, an administrator command prompt can be sent key strokes to run an arbitrary command.

However, if you have the same integrity level as the target process, the UI Access flag becomes irrelevant and you can directly compromise the process with at least one window by using windows hooks to inject a DLL. This window doesn’t need to present a user interface, in fact technologies such as COM use message-only windows under the hood that can be used to compromise the process without ever showing anything to the user.

Of course this is how things worked in UAC, but what about the new and improved Administrator Protection? Exactly the same as for the existing admin-approval UAC. The UI Access process will run under the caller’s token, which in this case will be the limited user, not the shadow administrator. The process will have the UI Access flag enabled as well as the integrity level set to High.

This is a problem, having a process running at High integrity level allows you to compromise any other process running at that level on the same desktop even if that process runs as a different user. As the UI Access process is running as the limited user there’s no profile separation which is one of the key improvements in Administrator Protection.

The elevation to High integrity is also silent, so at a minimum it’d be possible to break the security boundary without prompting the user, assuming there was a suitable administrator process to exploit. All we now need is a way of getting arbitrary code execution in a High integrity UI Access process. Fortunately, there are plenty of ways of doing this.

Achieving Arbitrary UI Access Execution

Over the years there’s been a number of ways to get arbitrary code executing as a High integrity level UI Access process. While Microsoft had made it clear that they were not a priority to be fixed, they sometimes were. Let’s break it down in a series of categories with some historical detail as well as my more recent research.

Bypassing the Secure Application Directory Check

One approach to getting arbitrary code to run is by bypassing the secure directory location check in the UAC service. If you could bypass this check, you could place either your own signed executable, or an existing executable that can be hijacked, say through DLL planting.

One approach is to find a bug in the `AiCheckSecureApplicationDirectory` method in `appinfo.dll` which implements the check. This method first opens the file path provided via `RAiLaunchAdminProcess`, then calls `GetFinalPathNameByHandle` on the handle to ensure the path isn't redirected to an insecure location. It then does a simple string check on the path looking for included and excluded directories. I found a bypass for the check back in 2017 and reported it to MSRC. The check didn't take into account that it's possible to write an NTFS named stream to a directory if the limited user has write access to that directory.

For example the directory `C:\Windows\tracing` is writable by a limited user but the `tracing` subdirectory is explicitly excluded in the check so that `C:\Windows\tracing\file.exe` would not be considered secure. However, with the same access you could write a named stream on the directory so that `C:\Windows\tracing:file.exe` would be considered inside the `C:\Windows` directory and thus secure. This bug wasn't fixed as a security bulletin but it did eventually get resolved in a later version of Windows and is not applicable to Windows 11.

Another approach is to find a writable file or directory in a secure location that is not explicitly excluded in the check. If you find a writable file then you could overwrite it with the executable file as the `CreateProcessAsUser` API used by the UAC service doesn't need a specific file extension for the executable file to be used. If you find a directory then you can just copy the executable file into that location.

On a default installation there doesn't seem to be any location that's not covered. One location I did find during the research is that sometimes on major Windows updates the `Tasks` directory is copied to the `Tasks_Migrated` directory as a backup. This backup directory is writable like the original `Tasks` and was not included in the list of excluded directories. However, you have no known way of forcing it to be created, and since I pointed it out Microsoft have added it to the list of directories to exclude.

Note: Microsoft did forget to add a check for named streams on `Tasks_Migrated`, however due to the access control on the directory it's not possible to exploit as a normal user.

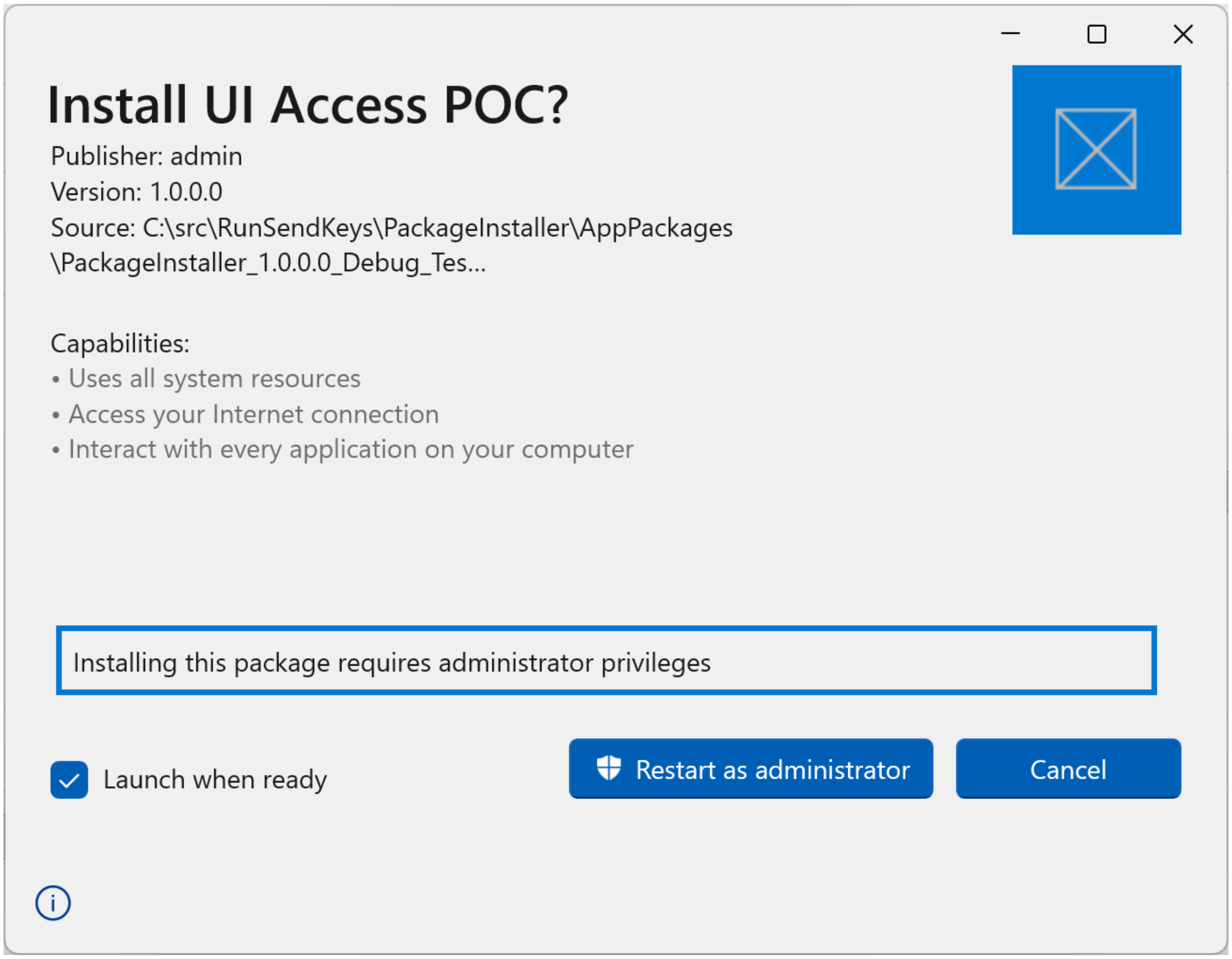
You can use my PowerShell tools to find potential candidates using the following command. For best results run it as an administrator and replace `<PID>` with a process ID of a limited user process. It doesn't filter out excluded directories, so you'd have to check yourself.

```
PS> $paths = "C:\Windows","C:\Program Files","C:\Program Files (x86)"
PS> Get-AccessibleFile -Win32Path $paths -Access Execute,WriteData `
       -DirectoryAccess AddFile -Recurse -ProcessId <PID>
```

A final approach is finding a way to write a file to an existing secure location though a separate mechanism that doesn't require bypassing the access control of that directory. I found just such an issue in my recent research. The Windows installer will install MSIX files into the `C:\Program Files\WindowsApps` directory, which is not excluded by the check. Windows 11 is configured by default to permit installing signed MSIX files without needing administrator privileges.

Therefore you can package up a UI Access executable into an MSIX installer, sign the installer with an arbitrary certificate then when installed the executable will be in a secure location. Of course to do this you'd need a code signing certificate but that isn't as big of a challenge as it seems. You might even be able to slip the signed UI Access executable file into a store application if you were so inclined. But this is now fixed as the `WindowsApps` directory is also excluded.

Interestingly there is a `uiAccess` restricted capability you can add to the manifest when building the MSIX which will elevate the packaged executable to High integrity UI Access. However, when you do that, installing the package requires administrator privileges as shown below and so it's not a bypass.



Install UI Access POC?

Publisher: admin
Version: 1.0.0.0
Source: C:\src\RunSendKeys\PackageInstaller\AppPackages
\PackageInstaller_1.0.0.0_Debug_Tes...

Capabilities:
• Uses all system resources
• Access your Internet connection
• Interact with every application on your computer

Installing this package requires administrator privileges

[✓] Launch when ready

Repurposing an Existing Secure UI Access Executable

A second category is finding functionality inside an UI Access capable executable file that's already in a secure location that can be abused. You have full control over the UI Access process' command line, perhaps there's an option to load an arbitrary DLL?

Before you can find exploitable behavior you need to find candidate executables that you can reverse engineer. You can use my PowerShell tools to find executable files which have the `uiAccess` manifest option set to true.

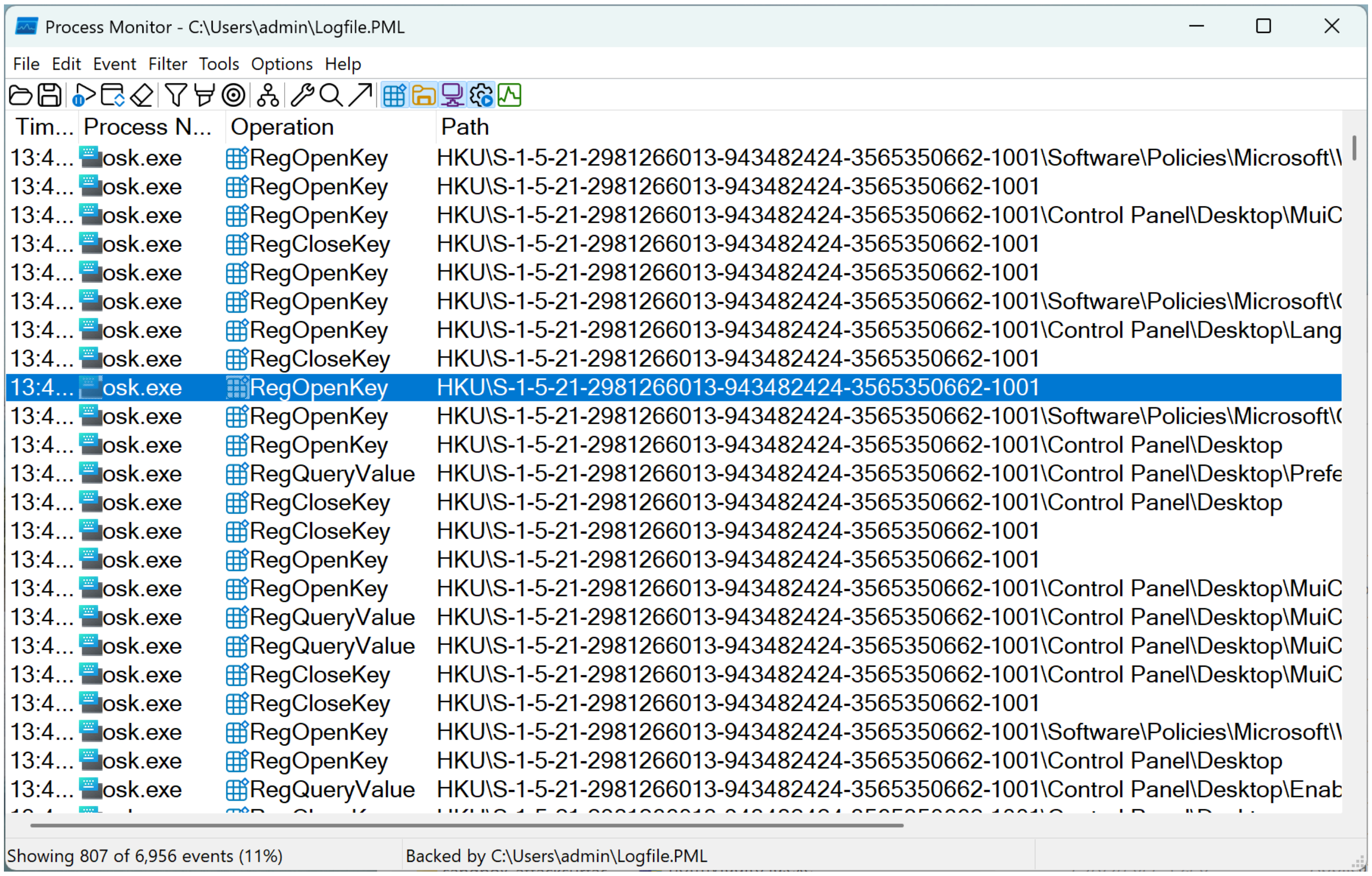
```
PS> $paths = "C:\Windows","C:\Program Files","C:\Program Files (x86)"
PS> Get-ChildItem -Path $paths -Include *.exe -Recurse |
  % {
       Get-Win32ModuleManifest $_.FullName
  } | Where-Object UiAccess | Select-Object -ExpandProperty FullPath
```

With the list of candidates you'll need to do some reverse engineering. I'll leave that up to you.

Shared Profile and Environment

One of the big changes in Administrator Protection was the separation of the shared profile between the limited user and the administrator. The goal was to prevent privilege escalation by modifying the user's profile on disk or the registry. Unfortunately as UI Access processes are created based on the limited user, the same as it was with UAC, this separation doesn't apply and you can find ways of exploiting this behavior.

A simple way of inspecting for potential exploitable behavior is to run Process Monitor and capture events accessing the limited user's registry hive or profile directory. It's also possible to hijack things like the user's `C:` drive mapping as the logon session is the same between the limited user and its UI Access processes.


Process Monitor - C:\Users\admin\Logfile.PML
File Edit Event Filter Tools Options Help

```
Tim...  Process N...  Operation       Path
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Software\Policies\Microsoft\\
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\MuiC
13:4... osk.exe       RegCloseKey     HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Software\Policies\Microsoft\(
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\Lang
13:4... osk.exe       RegCloseKey     HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Software\Policies\Microsoft\(
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop
13:4... osk.exe       RegQueryValue   HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\Prefe
13:4... osk.exe       RegCloseKey     HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop
13:4... osk.exe       RegCloseKey     HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\MuiC
13:4... osk.exe       RegQueryValue   HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\MuiC
13:4... osk.exe       RegQueryValue   HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\MuiC
13:4... osk.exe       RegCloseKey     HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\MuiC
13:4... osk.exe       RegCloseKey     HKU\S-1-5-21-2981266013-943482424-3565350662-1001
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Software\Policies\Microsoft\\
13:4... osk.exe       RegOpenKey      HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop
13:4... osk.exe       RegQueryValue   HKU\S-1-5-21-2981266013-943482424-3565350662-1001\Control Panel\Desktop\Enab

Showing 807 of 6,956 events (11%)     Backed by C:\Users\admin\Logfile.PML
```

This is a well known issue with UI Access and UAC so when I found it in Administrator Protection I didn't really need to report it, but felt I should. To ensure it got handled appropriately I found a specific exploitable condition and sent an accompanying proof-of-concept. In this case I found that the On-Screen Keyboard loaded a DLL from a path based on the `CommonProgramFiles` environment variable. By overriding this variable in the user's registry hive I could redirect the DLL load and get arbitrary code execution in the UI Access process.

During my research I stumbled upon a public bypass, originally for UAC but it still worked with Administrator Protection. This bypass was in the Quick Assist application, which seems to be an optional component but is installed by default on Windows 11. It abused the fact that the Quick Assist application would load the WebView2 APIs to display HTML content. WebView2 would look in the user's hive for an overridable installation location to load its library, by overriding this to a location under the user's control it's possible to force a DLL to be loaded into the UI Access process.

One of the most interesting aspects of this bypass is it uses an API I didn't know existed, GetProcessHandleFromHwnd, to get a kernel handle to the process which created a window to get arbitrary code execution in the UI Access process.

Exploit RAiLaunchAdminProcess

To launch a UI Access process, the shell calls the `RAiLaunchAdminProcess` RPC method in the UAC service. As is all too common with APIs that are not directly exposed or documented they can hide functionality that can result in exploitable behavior.

I reported two issues that allowed me to get arbitrary code execution in a UI Access process, one was a publicly known bypass while the other was a TOCTOU in the handling of the path to the executable file. The public bypass was described by myself in a blog post about using my PowerShell tooling to call local RPC methods. The example I gave was of calling `RAiLaunchAdminProcess` and abusing the fact the service doesn't sanitize the process creation flags.

You could pass the `DEBUG_PROCESS` flag, and from that get full control over the created process. The blog post described this in the context of a UAC bypass, but of course it applied equally well to UI Access processes as I detailed in the report I sent to MSRC. This was one of those bugs that I was concerned hadn't been remediated during the development of Administrator Protection, but as it was *just* a UAC bypass it'd clearly slipped through the cracks.

The second issue is in the handling of the path to the executable file, which allows us to compromise a UI Access process. The `RAiLaunchAdminProcess` RPC method has a very similar set of parameters to the `CreateProcessAsUser` API that's ultimately called to create the new process. This includes having a separate string representing the path to the executable to create and the command line to pass to the new process.

As I already described in the section on the secure application directory check, the validation is not done using the untrusted path string provided by the user but instead the file is opened and the final resolved path extracted to do the comparison. However, this resolved path is only used during the check, when it comes to create the process the original untrusted path is passed to `CreateProcessAsUser`'s `lpApplicationName` parameter.

For example, if you passed the path `Z:\osk.exe` as the executable file the service would try to open that path then resolve the final name. If the `Z:` drive was mapped to the `C:\Windows\system32` directory it would find the executable located at `C:\Windows\system32\osk.exe` which would be a permitted secure directory. However, `Z:\osk.exe` would then be passed as `lpApplicationName` to `CreateProcessAsUser`.

What use is this? The new process needs a base directory from where it'll check for local DLL loads, and the `CreateProcessAsUser` API uses the `lpApplicationName` parameter, with the executable filename removed, for this base directory. This means that you can start a UI Access process using the `Z:\osk.exe` path; it will try to load unknown DLLs first from the `Z:\` directory. If you remap the `Z:` drive to an untrusted location between the process being created and when it tries to load DLLs you can force an untrusted DLL to be loaded into the process and get arbitrary code execution. This is easy to do, as the UI Access process can be created suspended by passing the `CREATE_SUSPENDED` when calling `RAiLaunchAdminProcess`, remapping the drive, then resuming the process.

Access Token Stealing

The final category I'll mention is access token stealing. This is somewhat different from the others as you commonly can't get High integrity level from it, instead you get a process with the UI Access flag enabled which can be used to control higher integrity UI, just not abuse things like windows hooks.

As I described in an old blog post, if you create a UI Access process, you can open the process' access token, duplicate it, reduce the integrity level to Medium and finally create a new process using that token. No step in that process disabled the UI Access flag on the token.

Subsequent to the blog post Microsoft made a change. Now if the integrity level of a token is reduced via the `NtSetInformationToken` system call, it will also disable the UI Access flag. If you can't reduce the integrity level to Medium it's not possible to impersonate the token or use it for a new process, thereby mitigating the issue.

However, I noticed that there are some places in the kernel that lower the integrity level of the token which do not go through `NtSetInformationToken` and thus do not end up disabling the UI Access flag. One option was the creation of an App Container token via the `NtCreateLowBoxToken` system call. This will set the integrity level to Low which will allow the new token to be used to create a process. Even though the process would then run in an App Container sandbox it was still sufficient to send arbitrary window messages to more privileged processes.

Silently Bypass Windows Administrator Protection

Let's assume we now have code execution in a High integrity level UI Access process. How do we exploit that to bypass Administrator Protection? We need a process to be created silently as the shadow administrator that creates a window during its execution. From that we can use the SetWindowsHookEx, to force an arbitrary DLL to be loaded into the new process.

The best vector I found was using the fact that scheduled tasks can be configured to run with administrator privileges when executed. This still works when Administrator Protection is enabled, just the task process runs as the shadow administrator. We need a task that is enabled, can be started by the limited user and runs with administrator privileges. We can use my PowerShell tools `Get-AccessibleScheduledTask` command to find one:

```
PS> Get-AccessibleScheduledTask -Access Execute |
   ? { $_.AllowDemandStart -and $_.Enabled -and ($_.RunLevel -eq "Highest") } |
   Select-Object Name

Name
----
\Microsoft\Windows\DiskCleanup\SilentCleanup
\Microsoft\Windows\Input\LocalUserSyncDataAvailable
\Microsoft\Windows\Input\MouseSyncDataAvailable
...
```

The first task in the list `SilentCleanup` is well known to me. It's been used multiple times to bypass UAC, such as by abusing the fact it uses environment variables to find the executable file which a limited user can override. Unexpectedly the issue with environment variables wasn't fixed in the version of Administrator Protection I tested, so I reported that as a separate issue.

If we ignore the issue in handling environment variables, we can abuse this task as it'll create a window when the process runs, so we just setup a hook, start the task, wait for your hook DLL to be loaded by the `cleanmgr.exe` process and you've bypassed administrator protection. You can find a full PoC using this approach here.

Note: If you want to use the `GetProcessHandleFromHwnd` API, like the QuickAssist public bypass does, you'll probably need to win a race between the process creating a window and it terminating. For example, QuickAssist uses a file oplock to cause the task process to hang when it opens a specific file. If you use the windows hooks approach you don't need to worry about this.

Conclusions

All of the issues I reported were fixed, but that doesn't mean there's nothing left to find. Hopefully it's a little bit harder than before. However, it's still the case that if you can get code execution in a High integrity level process, with UI Access enabled or not, then you can leverage that to bypass Administrator Protection. Hopefully anything which is now found allowing code execution in a UI Access process is a serviceable security vulnerability and will be fixed.

One big change to UI Access processes over the original design of Administrator Protection is they now no longer run as the limited user. This change was introduced to fix issue 437868751. Instead they are created with a filtered copy of the shadow administrator token. This eliminates the shared profile issues, introducing much clearer separation between the administrator processes and the limited user.

Time will tell whether Administrator Protection is successful as a security boundary or not. Microsoft is taking it seriously, but more rigorous testing during development would have prevented many pre existing UAC bypasses from being missed. I'd recommend anyone interested in this feature to take a look now it's released and the previously known bugs have been fixed.