Zero Day Initiative — Activation Context Cache Poisoning: Exploiting CSRSS for Privilege Escalation

Published: 2023-01-23 · Archived: 2026-04-05 14:02:42 UTC

Starting in July of 2022, the Windows CSRSS process entered the consciousness of the infosec community as the source of several local privilege escalation vulnerabilities in Microsoft Windows. The first public information appeared on July 12 with the release of the patch for CVE-2022-22047, which was being actively exploited. Shortly thereafter, Microsoft published an article providing some technical details and revealing that the threat actor involved was an Austrian hack-for-hire group tracked by Microsoft as KNOTWEED. Fortuitously, these developments coincided with closely related research I had been conducting, and in that same month, I reported to Microsoft two additional vulnerabilities affecting the same component. These have now been patched as CVE-2022-37987 and CVE-2022-37989 respectively. All these bugs, which have been commonly known as the "CSRSS" bugs, are best understood as examples of a new class of privilege escalation vulnerabilities: *activation context cache poisoning*. In this article, we will describe this new bug class in depth. We will then explore the strengths and weaknesses of the code changes that Microsoft has introduced in response.

Understanding Activation Contexts

To begin, we must understand some of the basics of activation contexts.

There are various Windows APIs that a process can use to load additional components. The two most salient are `LoadLibrary` for the loading of DLLs, and `CoCreateInstance` for instantiation of COM components. For the purpose of this discussion, we will focus primarily on `LoadLibrary`, but parallel considerations apply to `CoCreateInstance`. Also, what applies for `LoadLibrary` applies equally for implicit DLL loads via linkage.

`LoadLibrary` may be called with either an absolute or a relative path, but in either case, ambiguity remains as to which version of the requested DLL is appropriate to load. For example, even if an application specifically requests `C:\Windows\System32\comctl32.dll`, there is no assurance that the `comctl32.dll` presently installed at that location on the local machine is the version intended by the author of the application. This is one basic use-case for activation contexts. An activation context is defined as state that Windows uses to resolve ("bind") a component reference to the appropriate component version. Every thread always has exactly one activation context in effect ("active") at any given time. To resolve a reference, Windows uses the currently active activation context.

To specify activation context data for a thread, code can use the Activation Context APIs. Much more commonly, though, an activation context is specified declaratively by deploying a manifest. The manifest has the form of XML data. It is generally embedded as a resource in an EXE or DLL, though Windows will alternatively look for a `.manifest` file sibling to an EXE at launch.

Let's see a representative example of a manifest. The following manifest is found as a resource in `notepad.exe` on Windows 10 22H2 19045.2251 x64:

	<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
	<!-- Copyright (c) Microsoft Corporation -->
	<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
	<assemblyIdentity
	name="Microsoft.Windows.Shell.notepad"
	processorArchitecture="amd64"
	version="5.1.0.0"

```
type="win32"/>
<description>Windows Shell</description>
<dependency>
<dependentAssembly>
<assemblyIdentity
type="win32"
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="*"
publicKeyToken="6595b64144ccf1df"
language="*"
/>
</dependentAssembly>
</dependency>
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
<security>
<requestedPrivileges>
<requestedExecutionLevel level="asInvoker" uiAccess="false"/>
</requestedPrivileges>
</security>
</trustInfo>
<application xmlns="urn:schemas-microsoft-com:asm.v3">
<windowsSettings xmlns:ws2="http://schemas.microsoft.com/SMI/2016/WindowsSettings">
<ws2:dpiAwareness>PerMonitorV2</ws2:dpiAwareness>
</windowsSettings>
</application>
</assembly>
```

Figure 1: Manifest of `notepad.exe`

This manifest defines an activation context to be activated upon launch of `notepad.exe`. Note the `<dependency>` element. It declares that the stated assembly version (`Microsoft.Windows.Common-Controls`, version `6.0.0.0`) should be incorporated. That would be found within `C:\Windows\WinSxS`, the location for side-by-side assembly installations. After probing, Windows will locate an acceptable match, having its own manifest at `C:\Windows\WinSxS\Manifests\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.19041.1110_none_60b5254171f9507e.manifest`:

	<?xml version="1.0" encoding="utf-8" standalone="yes"?>
	<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0" copyright="Copyright (c) Microsoft Corporation. All Rights Reserved." xmlns:cmiv2="urn:schemas-microsoft-com:asm.v3" cmiv2:copyright="Copyright (c) Microsoft Corporation. All Rights Reserved.">
	<noInheritable />
	<assemblyIdentity name="Microsoft.Windows.Common-Controls" version="6.0.19041.1110" processorArchitecture="amd64" publicKeyToken="6595b64144ccf1df" type="win32" />
	<file name="comctl32.dll" cmiv2:importPath="$(build.nttree)\asms\60\msft\windows\common\controls" cmiv2:sourceName="">
	<windowClass>ToolbarWindow32</windowClass>
	<windowClass>ComboBoxEx32</windowClass>
	<windowClass>msctls_trackbar32</windowClass>
	<windowClass>msctls_updown32</windowClass>
	<windowClass>msctls_progress32</windowClass>
	<windowClass>msctls_hotkey32</windowClass>
	<windowClass>msctls_statusbar32</windowClass>
	<windowClass>SysHeader32</windowClass>
	<windowClass>SysListView32</windowClass>
	<windowClass>SysTreeView32</windowClass>
	<windowClass>SysTabControl32</windowClass>
	<windowClass>SysIPAddress32</windowClass>
	<windowClass>SysPager</windowClass>
	<windowClass>NativeFontCtl</windowClass>
	<windowClass>Button</windowClass>
	<windowClass>Static</windowClass>
	<windowClass>Listbox</windowClass>
	<windowClass>ScrollBar</windowClass>
	<windowClass>SysLink</windowClass>
	<windowClass>tooltips_class32</windowClass>
	<windowClass>ButtonListBox</windowClass>
	<windowClass>SysAnimate32</windowClass>
	<windowClass>SysMonthCal32</windowClass>
	<windowClass>SysDateTimePick32</windowClass>
	<windowClass>ReBarWindow32</windowClass>

	<windowClass>Edit</windowClass>
	<windowClass>Combobox</windowClass>
	<windowClass>ComboLBox</windowClass>
	<windowClass>DropDown</windowClass>
	<signatureInfo xmlns="urn:schemas-microsoft-com:asm.v3">
	<signatureDescriptor PETrust="true" pageHash="true" />
	</signatureInfo>
	<asmv2:hash xmlns:asmv2="urn:schemas-microsoft-com:asm.v2" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
	<dsig:Transforms>
	<dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
	</dsig:Transforms>
	<dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha256" />
	<dsig:DigestValue>iXKqUejv96Ddh9vG1w798vL5l6Ox+DrmzOqVtWULo24=</dsig:DigestValue>
	</asmv2:hash>
	</file>
	<dependency optional="yes" cmiv2:discoverable="no">
	<dependentAssembly>
	<assemblyIdentity name="Microsoft.Windows.Common-Controls.Resources" version="6.0.0.0" processorArchitecture="amd64" language="*" publicKeyToken="6595b64144ccf1df" type="win32" />
	</dependentAssembly>
	</dependency>
	<memberships xmlns:cmiv2="urn:schemas-microsoft-com:asm.v3">
	<categoryMembership>
	<id name="Microsoft.Windows.Categories" version="1.0.0.0" publicKeyToken="365143bb27e7ac8b" typeName="BootRecovery" />
	</categoryMembership>
	</memberships>
	</assembly>

Figure 2: Manifest of Microsoft.Windows.Common-Controls version 6.0.19041.1110 from WinSxS

The element `<file name="comctl32.dll" ⋯` indicates that a file with that name is part of the `Microsoft.Windows.Common-Controls` assembly. Windows will find it at the corresponding location `C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.19041.1110_none_60b5254171f9507e\comctl32.dll`. (Note that `WinSxS` is a special folder whose structure is unique.)

In sum, these manifests set up an activation context for `notepad.exe`, so that when `notepad.exe` loads `comctl32.dll` (whether through linkage, or through an explicit call to `LoadLibrary`), the appropriate version of `comctl32.dll` will be located.

There is a great deal more to be discussed on the topic of activation contexts. The best all-in-one resource on the topic I have found is the article here. The details presented above, though, should be sufficient for our present purposes.

The Activation Context Cache

The procedure of locating manifests, parsing them, and probing for their dependencies which themselves must be processed recursively, is fairly intensive both in terms of computational steps and disk accesses. For this reason, Microsoft introduced a caching mechanism. Naturally, for the cache to deliver a performance benefit, it must be capable of persisting beyond the lifetime of a single process, so that cached results can be reused. It is probably for this very reason that activation context creation does not take place in-process but is instead delegated to the per-session `CSRSS.EXE` process. In the example above, the `notepad.exe` process will start by making a cross-process call into `CSRSS.EXE` to create its activation context. `CSRSS.EXE` performs all the required probing steps and places the results into an in-memory activation context structure. It passes this structure back to the caller. When the `notepad.exe` process needs to load a module such as `comctl32.dll`, it refers to this in-memory activation context structure to direct the library load correctly. In addition to returning the activation context structure to the caller, `CSRSS.EXE` keeps a copy in a cache. The cache comes into play the next time `notepad.exe` launches. Then, when calling into `CSRSS.EXE` to generate an activation context, `CSRSS.EXE` does not have to go through as many steps to probe for on-disk manifests and parse them. Instead, it retrieves the activation context it previously stored in the cache. To validate that the cached data is still valid, it needs only to check that the file modification time has not changed for `notepad.exe` (though I am intentionally omitting some less-important details for simplicity).

Within each cache entry, `CSRSS` stores the executable's file modification time, to make it possible to later determine if the cache entry is still valid, as I have just explained. It also stores the 128-bit `FILE_ID_INFORMATION` value generated by NTFS that serves as a guaranteed unique identifier for the file. This serves to prevent canonicalization bugs, in which the same textual path could resolve to two different files due to changes to symbolic links.

As happens all too often, though, and despite the above precautions, the introduction of caching is accompanied by major risks.

The Danger of Cache Poisoning

If the cache contains incorrect data, the execution of any process using that data can be compromised. For example, incorrect ("poisoned") data in the activation context cache might result in an arbitrary DLL being loaded into a privileged process, yielding privilege escalation.

How might the cache become poisoned? One problem with the architecture of activation contexts is that, while `CSRSS.EXE` performs activation context generation and caching, it does so for the benefit of the client (caller) process. Whatever activation context the client wishes to create, it is the job of `CSRSS` to create that context. I suppose it is for this reason that the interface supported by `CSRSS` is rather permissive, allowing the client process to specify the exact manifest XML that will be parsed. This makes perfect sense when `CSRSS` is thought of as a server acting on behalf of a client. But when caching comes into the picture, this arrangement becomes toxic. The activation context generated from the arbitrary manifest XML provided by the client becomes a semi-permanent part of the system's functioning and later can affect execution of a more highly privileged process.

CVE-2022-22047: KNOTWEED Exploits Cache Poisoning

This is the approach that was taken by the exploit code found in the wild, originating from the KNOTWEED hacking group. The exploit crafts a call into CSRSS. The call requests an activation context for a privileged executable and specifies a malicious manifest. The manifest makes use of an undocumented manifest XML attribute named `loadFrom`. This attribute allows unrestricted redirection of DLLs to any location on disk, including locations outside of the normal search patch (the normal search path being the executable's folder and subfolders thereof, in addition to the shared assemblies installed to `C:\Windows\WinSxS`):

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity name="SomePrivilegedExecutable" processorArchitecture="amd64" version="1.0.0.0" type="win32"/>
<description>Windows Shell</description>
<file loadFrom="c:\repro\payload.dll" name="advapi32.dll"/>
</assembly>
```

Figure 3: A malicious manifest of the sort used in the KNOTWEED exploit

The manifest shown in Figure 3 specifies that all requests to load library `advapi32.dll` should instead load `c:\repro\payload.dll`. After `CSRSS.EXE` creates the requested activation context, it enters it into the cache. Upon a subsequent launch of the targeted executable, `CSRSS.EXE` provides the targeted process with the cached activation context. Thus, when the privileged process attempts to load the `advapi32.dll` module it depends on, the attacker's code will load instead.

To conduct this attack, the attacker's only prerequisite is the ability to write the `payload.dll` file somewhere within the filesystem. To ensure the cache will be affected, the exploit code can first flood `CSRSS` with requests to clear out all existing cache entries, and conclude with one final message to create a new poisoned entry for the targeted executable.

Note that the attacker must choose a target that is highly privileged but runs in the same session as the interactive user (typically session 1, when there is only one interactive user). This is because there is one `CSRSS.EXE` process per session, and hence one activation context cache per session. The session-wide shared cache makes the attack possible.

Microsoft patched this as CVE-2022-22047 in the July 2022 patches, but the fix was extremely narrow. It addressed only the usage of the undocumented `loadFrom` attribute. After the patch, code in `sxs.dll!CDllRedir::ContributorCallback`, which handles DLL redirections specified with `loadFrom`, sets a newly defined flag in the activation context indicating that the activation context contains a DLL redirection. By checking this flag, `CSRSS` will now treat any such activation context as non-cacheable. The logic that controls which activation contexts are inserted into the cache is found in `sxssrv.dll!BaseSrvSxsCreateActivationContextFromStructEx`.

This patch was too narrow, however. Activation context cache poisoning can be accomplished in other ways besides using a `loadFrom` attribute, as we will now see.

CVE-2022-37989: Bypassing the Patch for the KNOTWEED Exploit

Upon examining the patch for the KNOTWEED exploit I realized that using `loadFrom` was probably not the only way to write a malicious manifest that injects arbitrary code into the target process. Indeed, I quickly found that I was able to write a manifest that injected code via a different technique: declaring a dependent assembly that resides in an attacker-controlled portion of the filesystem:

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity name="SomePrivilegedExecutable" processorArchitecture="amd64" version="1.0.0.0” type=”win32”/>
<description>Windows Shell</description>
<dependency cmiv2:discoverable=”no”>
<dependentAssembly>
<assemblyIdentity
name="..\..\..\..\..\..\pwn\pwn"
```

	version="1.0.0.0"
	processorArchitecture="amd64"
	language="*"
	publicKeyToken="6595b64144ccf1df"
	type="win32"/>
	</dependentAssembly>
	</dependency>
	</assembly>

Figure 4: A malicious manifest using a `dependentAssembly` element

The malicious manifest shown in Figure 4 leads `CSRSS` to load a second manifest for the named dependency, which it finds at the attacker-controlled location `C:\pwn\pwn.MANIFEST`:

	<?xml version="1.0" encoding="utf-8" standalone="yes"?>
	<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0" copyright="Copyright (c) Microsoft Corporation. All Rights Reserved." xmlns:cmiv2="urn:schemas-microsoft-com:asm.v3" cmiv2:copyright="Copyright (c) Microsoft Corporation. All Rights Reserved.">
	<assemblyIdentity name="..\..\..\..\..\..\pwn\pwn" version="1.0.0.0" language="en-US" processorArchitecture="amd64" publicKeyToken="6595b64144ccf1df" type="win32" />
	<file name="advapi32.dll">
	</file>
	</assembly>

Figure 5: Manifest for dependent assembly, stored at `C:\pwn\pwn.MANIFEST`

Here, `advapi32.dll` is the name of a DLL needed by the target process. As a result of the combination of these two manifests, when the target process loads `advapi32.dll`, the DLL will be loaded from the attacker-controlled location `C:\pwn\` instead of the legitimate location `C:\Windows\System32\`. As before, local privilege escalation results when the highly privileged target process launches within the current user's session.

Microsoft patched this variant in October 2022 as CVE-2022-37989. The patch was analogous to the patch for the original KNOTWEED exploit. This time, the main code change was in `sxs.dll!CNodeFactory::XMLParser_Element_doc_assembly_dependency_dependentAssembly_assemblyIdentity`, which handles the `assemblyIdentity` subelement of the `dependentAssembly` element. After the patch, the code in that function checks if the name of the dependency contains any forward slashes or backslashes. If so, it sets another newly defined flag, much like the new flag introduced in the patch for CVE-2022-37989. `sxssrv.dll!BaseSrvSxsCreateActivationContextFromStructEx` recognizes activation contexts that have this flag set and does not enter them into the cache.

It's dubious whether it was ever intended to allow the `dependentAssembly` element to contain directory traversal. Nevertheless, Microsoft has decided to continue supporting this behavior, perhaps for the sake of backwards compatibility. As a reasonable trade-off, they made activation contexts non-cacheable when they rely on this behavior.

CVE-2022-37987: A New Vector for Activation Context Cache Poisoning

The two attacks discussed above both work by crafting messages to `CSRSS` to create a customized, malicious activation context that differs from the legitimate activation context that would normally be created by reading the manifest XML from disk. It is surprising that `CSRSS` was architected to accept such sensitive data from untrusted clients. As I mentioned earlier, this might be thought of as a reasonable design if you view `CSRSS` as merely providing a service to a client. From that perspective, the client should be able to specify whatever activation context details it wishes. The problem arises due to caching: crafted data from an untrusted process can affect not only that client, but also other clients that consult `CSRSS` afterwards.

As it happens, though, there is a completely separate vector for injecting arbitrary activation context data into `CSRSS`, one that has nothing at all to do with the interface that `CSRSS` exposes.

The trouble arises because `CSRSS.EXE` performs many of its filesystem accesses while impersonating the caller. I believe this is unavoidable, because `CSRSS` needs to generate an accurate activation context based upon how the caller would view the filesystem.

What is not so widely appreciated, however, is how vastly different a filesystem can appear when operating under impersonation. In particular: An unprivileged process can redirect a DOS device by creating an object namespace symlink. For example, if a process creates a symlink from `\??\C:` to `\GLOBAL??\C:\evil`, then all accesses to the filesystem rooted at `C:\` are instead redirected to the root `C:\evil`. The redirection is not limited to the process that creates the symlink. Rather, it affects the entire logon session. (The concept of "logon session" is unrelated to the concept of Windows sessions. See here and here for an explanation of logon sessions.) Not only that, but if a more highly privileged process impersonates a token for which the symlink is active, the privileged process sees the bogus filesystem as well. This is true even if the privileged process is in a different Windows session, for example, session 0.

If that spooks you, it should. James Forshaw discovered back in 2015 that it creates a massive security hole. Any time a privileged process impersonates a user, and the privileged process loads a library (or can be influenced to load a library) while that impersonation is ongoing, that privileged process can be completely compromised. All an attacker needs to do is redirect `C:\` to a bogus location where the privileged process will pick up a malicious version of the requested library.

To fix this security hole, Microsoft was compelled to add a new object attribute flag recognized by `NtOpenFile` (and similar APIs) instructing the kernel to ignore any DOS device redirections originating from the impersonation token. Ultimately, Microsoft publicly documented this flag as `OBJ_IGNORE_IMPERSONATED_DEVICEMAP`. You can find its use in the loader code in this excerpt from `ntdll!LdrpMapDllNtFileName` (`ntdll.dll` 10.0.19041.2130, Oct. 2022 patch level):

```
    attributes = OBJ_CASE_INSENSITIVE;
    ObjectAttributes.Length = 48;
    if ( !LdrpUseImpersonatedDeviceMap ) // Note how Microsoft left legacy
    // vulnerable behavior available
    // via configuration
    attributes = OBJ_IGNORE_IMPERSONATED_DEVICEMAP|OBJ_CASE_INSENSITIVE;
    ObjectAttributes.RootDirectory = 0i64;
    ObjectAttributes.Attributes = attributes;
    ObjectAttributes.ObjectName = a2;
    *(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0i64;
    ...
    ntStatus = NtOpenFile(
    &FileHandle,
```

```
    SYNCHRONIZE|FILE_TRAVERSE|FILE_LIST_DIRECTORY,
    &ObjectAttributes,
    &IoStatusBlock,
    FILE_SHARE_DELETE|FILE_SHARE_READ,
    FILE_NON_DIRECTORY_FILE|FILE_SYNCHRONOUS_IO_NONALERT);
```

Figure 6: Excerpt from `ntdll!LdrpMapDllNtFileName` showing patch for CVE-2015-1644

By contrast, `CSRSS` prior to the December 2022 patch level still did not make use of the `OBJ_IGNORE_IMPERSONATED_DEVICEMAP` flag. Thus, those filesystem operations `CSRSS` performs under impersonation were still vulnerable to manipulation by an attacker who establishes a DOS device redirection.

We can exploit this behavior by establishing a redirection for the `C:\` device while `CSRSS` is performing probing operations. The redirection will lead `CSRSS` into a bogus `C:\Windows\WinSxS\` folder, where we can place crafted manifests. `CSRSS` will cache the activation context created from a crafted manifest, and local privilege escalation can then proceed in the same way as in the other attacks described above.

For example, suppose we create a fake root at `C:\ActCtX\`. (The name `ActCtX` is arbitrary.) Within `C:\ActCtX` we can place a fake `Windows\WinSxS` folder, containing a crafted copy of the side-by-side assembly `Microsoft.Windows.GdiPlus`. The malicious copy is identical to the original except for the addition of one dependency within the manifest:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0" copyright="Copyright (c) Microsoft Corporation. All Rights Reserved." xmlns:cmiv2="urn:schemas-microsoft-com:asm.v3" cmiv2:copyright="Copyright (c) Microsoft Corporation. All Rights Reserved.">
<assemblyIdentity name="Microsoft.Windows.GdiPlus" version="1.1.19041.1706" processorArchitecture="amd64" publicKeyToken="6595b64144ccf1df" type="win32" />
<dependency cmiv2:discoverable="no">
<dependentAssembly>
<assemblyIdentity name="..\..\..\..\..\..\ActCtX\ActCtX" version="1.0.0.0" processorArchitecture="amd64" language="*" publicKeyToken="6595b64144ccf1df" type="win32" />
</dependentAssembly>
</dependency>
...
```

Figure 7: Crafted manifest to be dropped to `C:\ActCtX\Windows\WinSxS\Manifests\amd64_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.19041.1706_none_919e8e54cc8d4ca1.m`

This tricks `CSRSS` into thinking that `Microsoft.Windows.GdiPlus` has a dependency on an assembly named `ActCtX`, located at the filesystem root. Since it will be looking for that assembly, we'll need to put one there:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0" copyright="Copyright (c) Microsoft Corporation. All Rights Reserved." xmlns:cmiv2="urn:schemas-microsoft-com:asm.v3" cmiv2:copyright="Copyright (c) Microsoft Corporation. All Rights Reserved.">
```

	<assemblyIdentity name="..\..\..\..\..\..\ActCtX\ActCtX" version="1.0.0.0" processorArchitecture="amd64" publicKeyToken="6595b64144ccf1df" type="win32" />
	<file name="advapi32.dll">
	</file>
	</assembly>

Figure 8: Manifest for fake dependent assembly, to be dropped to `C:\ActCtX\ActCtX\ActCtX.MANIFEST`

Note that we drop this manifest to `C:\ActCtX\ActCtX\ActCtX.MANIFEST`, with an extra folder named `ActCtX` in the path. This compensates for the fact that CSRSS will access this manifest while the DOS device redirection is in effect. Credit to Oliver Lyak (@ly4k_) for working out the details of the required folder structure.

The overall effect of these manifests is that `CSRSS` will generate an activation context specifying that any load of `advapi32.dll` should be loaded from the attacker-controlled location `C:\ActCtX\advapi32.dll`. `CSRSS` will cache this activation context. When the cached activation context is reused for a privileged process in the same session, a load of DLL `advapi32.dll` will instead load attacker code into the process, thus achieving privilege escalation to `NT AUTHORITY\SYSTEM`.

Microsoft patched this in October 2022 as CVE-2022-37987. The patch consists of a change to `sxssrv.dll!BaseSrvSxsCreateActivationContextFromMessage`. In that function, the code retrieves the `FILE_ID_INFORMATION` of the EXE or DLL file for which `CSRSS` is creating the activation context. This value becomes part of the cache key. We mentioned earlier why this is an important precaution for ensuring that an activation context created for one executable is not later retrieved from cache and applied to a different executable. Prior to the patch, the `NtQueryInformationFile` operation that retrieves the `FILE_ID_INFORMATION` value was not performed under impersonation. Therefore, when an attacker would create a cache entry for, e.g., `C:\Windows\System32\SomePrivilegedExecutable.exe`, even if the attacker had redirected `C:\` to `C:\evil`, the resulting cache entry would apply to the legitimate `C:\Windows\System32\SomePrivilegedExecutable.exe` as opposed to `C:\evil\Windows\System32\SomePrivilegedExecutable.exe`. The latter would be useless to an attacker, because `C:\evil\Windows\System32\SomePrivilegedExecutable.exe` is not an executable that would ever be launched as a privileged process. Microsoft's patch was to add impersonation during the call to retrieve the `FILE_ID_INFORMATION`. The intention of this change, apparently, was so that any activation context generated during a DOS device redirection would apply only to a spoofed file and never to the legitimate one.

As soon as I examined this patch, I was quite skeptical of its effectiveness, and after further analysis and experimentation I concluded that my skepticism was warranted. Unfortunately, the patch accomplishes nothing. An attacker can bypass it just by removing the DOS device redirection for the duration of the `NtQueryInformationFile` operation and putting the DOS device redirection back in place immediately thereafter. With this adjustment, everything can be made to work again in the exact way it worked prior to the patch. I additionally found that an attacker can reliably determine the precise time to revert and reestablish the redirection by using an oplock.

Nevertheless, there is no reason to panic. First it should be noted that in December 2022, Microsoft augmented their patch for CVE-2022-37987 by adding in use of the `OBJ_IGNORE_IMPERSONATED_DEVICEMAP` flag during manifest probing. Furthermore, although the original patch for CVE-2022-37987 was easily bypassed, that did not necessarily mean it was possible to make the attack work again. The reason is that the attack shown here relies on directory traversal within a `dependentAssembly` element, and that technique was effectively blocked by the patch for CVE-2022-37989 discussed earlier. You could instead request a DLL redirection with a `loadFrom` attribute, but then you would be blocked similarly by the CVE-2022-22047 patch. Other manifest features could be tried, but at this time I don't see any obvious paths to arbitrary code execution. One could specify a COM class redirection, via a `comClass` element, to cause an unexpected DLL to be loaded into a privileged process, but one would be limited to DLLs already existing in a secure location (e.g., `System32`). This would cause a failure within the privileged process, but exploitation for arbitrary code execution remains extremely difficult. Determining whether any potentially hazardous features of manifest files remain unpatched is an interesting open research question.

There is an additional reason the threat of local privilege escalation using activation context cache poisoning is now greatly reduced: Microsoft has introduced a general mitigation.

Mitigating the Threat from Activation Context Cache Poisoning

In addition to the specific July 2022, October and December 2022 patches described above, Microsoft wisely added a general mitigation in October 2022.

As of the October 2022 patch, an integrity level value is now stored together with each cache entry. For example, an ordinary non-privileged process running within an interactive session has an integrity level of Medium (decimal 8192, hex 0x2000). If this process calls into `CSRSS` to create an activation context, and `CSRSS` creates a new corresponding cache entry, the cache entry will be tagged with 0x2000 (Medium) as its integrity level. Subsequently, if another process running at Medium or lower integrity calls into `CSRSS` to create an activation context for the same executable, `CSRSS` will satisfy the request from the cache. By contrast, if a process running at an integrity level higher than Medium makes such a request, `CSRSS` will treat it as a cache miss. In this event, `CSRSS` will create an activation context from scratch and evict the existing cache entry, replacing it with the new one bearing the higher integrity label. This logic is found in `sxssrv!BaseSrvSxsCreateActivationContextFromStructEx`.

This mitigation is highly beneficial. It enormously reduces the range of scenarios where the cache might be a means of privilege escalation. For example, in the KNOTWORM attack and all variations described above, a process running as Medium creates a malicious cache entry, to be picked up later by a process running as `NT AUTHORITY\SYSTEM` within the user's session. As far as I am aware, in a generic install of Windows, all processes that launch as `NT AUTHORITY\SYSTEM` within an interactive session have an integrity level of System (decimal 16384, hex 0x4000). So, thanks to this mitigation, when it comes time for the cache entry to be retrieved for use by the privileged process, `CSRSS` will discard the lower-integrity entry, so that the privileged process will remain completely unaffected by it.

This mitigation is not a panacea. Activation context cache poisoning remains a viable attack technique, but only in a much-reduced set of circumstances. Recall that there is one `CSRSS.EXE` process per session, so by nature, the cache is per-session. Add to this the restriction imposed by the mitigation, that a cache entry created by a lower-integrity process will never be picked up by a higher-integrity process. There remains a small subset of privilege escalation scenarios that skirt these restrictions. I will give two examples:

• The `Dnscache` (also known as "DNS Client") service process runs as `NETWORK SERVICE`, but `SeImpersonatePrivilege` has been removed from its token. Technically, this makes `Dnscache` a low-privileged process. If an attacker compromises this process, it is ordinarily not trivial to escalate to `NT AUTHORITY\SYSTEM`. Nevertheless, since the process runs with System integrity (not to be confused with `NT AUTHORITY\SYSTEM`), and its session is session 0, malicious code running within `Dnscache` could poison the activation context cache to compromise any service running in session 0 as `NT AUTHORITY\SYSTEM`. Note, though, that to complete the privilege escalation, it would also be necessary to find an unpatched manifest feature, as discussed earlier.

• `dwm.exe` is a process that runs in each interactive session. It runs as the interactive user, but with System integrity. Therefore, an attacker who achieves code execution in `dwm.exe` would be in a position to create a cache entry with System integrity level that would be picked up by highly privileged processes that also run within the interactive session, just as in the original KNOTWEED attack. Notably, there has been a report of probable compromise of `dwm.exe` used by an exploit chain in the wild, though since that report predates the cache mitigation, the attacker in that case must have had a different aim in compromising `dwm.exe`. Again, to complete the privilege escalation, an unpatched manifest feature must be found.

Conclusion

We have taken the wraps off a new bug class in Microsoft Windows, which we call *activation context cache poisoning*. Successful exploitation generally results in privilege escalation, as achieved in the wild by the KNOTWEED group. The vulnerabilities recently disclosed as the "CSRSS" vulnerabilities all fall into this new bug class. Though Microsoft has now released a general mitigation, corner cases remain that fall outside the mitigation's scope. It remains an open research question whether Microsoft's specific patches for the known vulnerabilities are sufficient to make activation contexts safely cacheable across processes.

Follow us on Twitter, Mastodon, LinkedIn, or Instagram for the latest updates from the ZDI and any new developments we may find in this new bug class.
