Symbolic Hooks Part 4: The App Container Traverse-ty

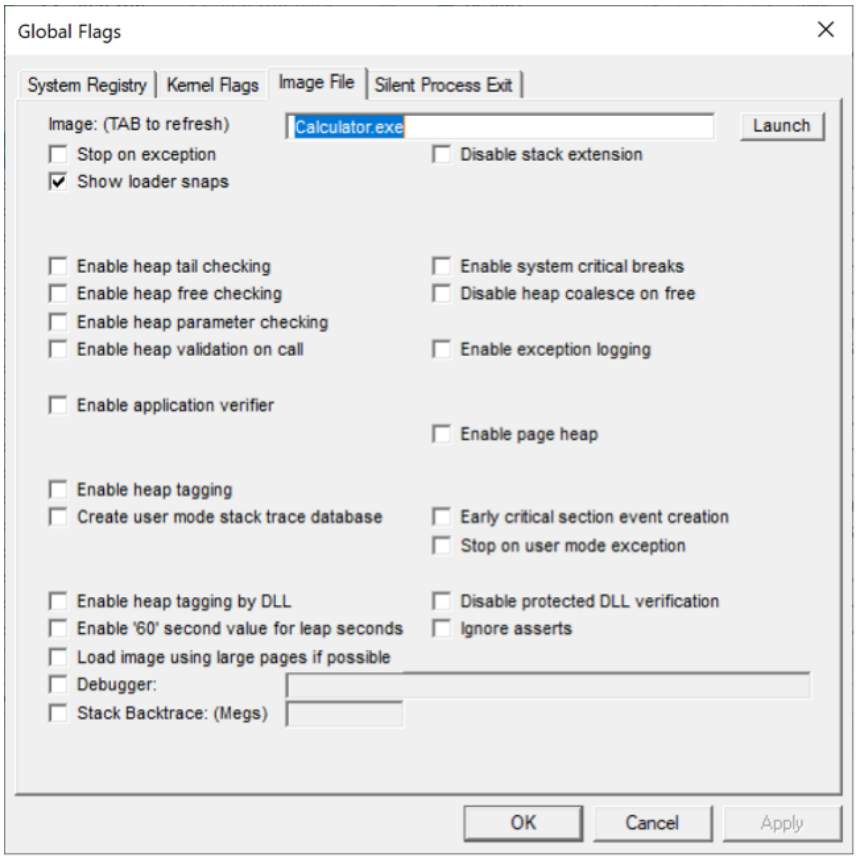
windows-internals.com/symhooks-part-four

By Yarden Shafir & Alex Ionescu

After getting the driver in Part 3 of our blog to load and adding a `DbgPrintEx` statement in our hook, we managed to get all the paths that were being opened without crashing the machine. We got really excited thinking we were done. But as soon as we clicked on the Start Menu, we noticed things had gone awry – it wasn't starting up at all, and when we launched Process Monitor from SysInternals, we could see `ShellExperienceHost.exe` crashing. We tried other applications, which ran fine but still, the machine was pretty much unusable. So, we relaunched our IDA and WinDbg and went hunting for more bugs.

As we were playing around, we noticed that another process that wasn't working was the new Windows Calculator. We then launched it in the debugger, taking advantage of the fact that the WinDbg Preview on the Microsoft Store can now easily launch Application Packages (which is needed, since `Calc.exe` is now simply a launcher for the real `Calculator.exe` — which essentially just does a `ShellExecute` of `calculator://`). Unfortunately, as soon as the debugger "attached", the process had already died. This is usually a sign of a loader issue – such as an import library not being present, failing to load, or missing some required import.

A really useful way to debug such issues is to enable "Loader Snaps", which is a Windows debugging feature that leverages "Global Flags". These flags are set either in the kernel (`nt!NtGlobalFlag` — and recently, `nt!NtGlobalFlag2`) or in user-space, in the Process Environment Block (PEB) of every process, as `Peb->NtGlobalFlag` (or again, recently, `Peb->NtGlobalFlag2` as well). You can enable system-wide global flags (either kernel or user ones) as well as per-process global flags through a handy utility that ships with the Windows Debugging Tools, unoriginally also called Global Flags (`Gflags.exe`). In the screenshot below, you can see how we enabled this debugging feature for `Calculator.exe`



Loader snaps instruct the loader to print out short debugging messages ("snaps") which trace all parts of the link-loading process: import resolution, DLL loading, manifest file parsing, SxS redirection, even down to calls of `GetProcAddress`. Our thought was to launch Calculator with snaps enabled, with and without our driver, and then do a simple diff between the two debugger outputs.

The first thing we noticed was that when running with our driver we get this interesting error, right as soon as the process starts:

```
LdrpInitializeProcess - ERROR: Initializing the current directory to
"C:\Program
Files\WindowsApps\Microsoft.WindowsCalculator_10.1910.0.0_x64__8wekyb3d8bbwe\"
failed with status 0xc0000022
```

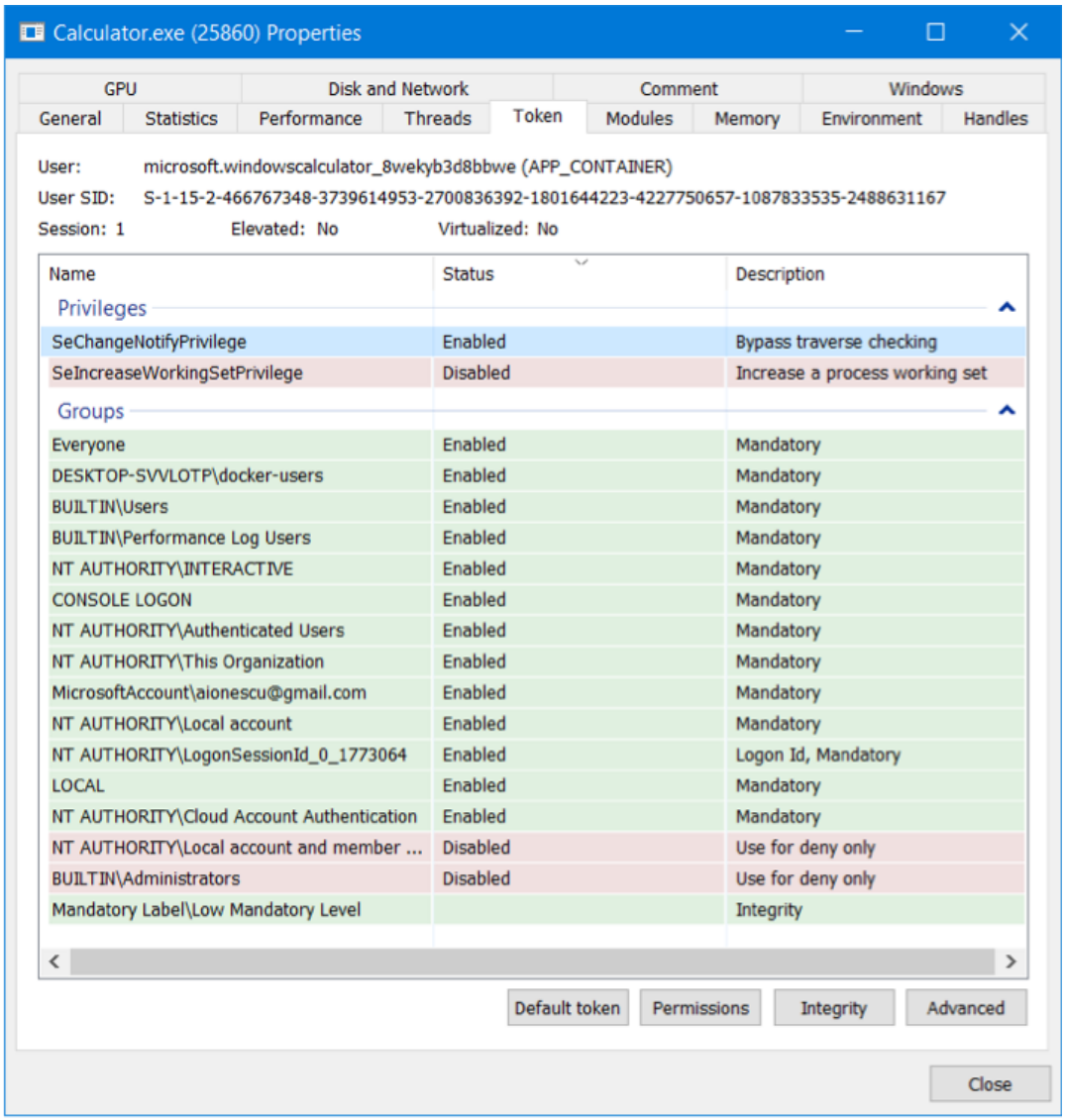
This tells us that initializing the current application's directory failed with error `0xC0000022`, which is the `NTSTATUS` code for `STATUS_ACCESS_DENIED`. Normally, you'd expect an application to have access to its own local directory, so this was already unusual. We searched for this error in the rest of the log, and compared with the log of our run without our symbolic link hook:

```
C:\logs>findstr "0xc0000022" "calc_with_symlink_hook.txt"
LdrpInitializeProcess - ERROR: Initializing the current directory to "C:\Program Files\WindowsApps\Microsoft.WindowsCalc
ulator_10.1910.0.0_x64__8wekyb3d8bbwe\" failed with status 0xc0000022
LdrpResolveDllName - RETURN: Status: 0xc0000022
LdrpResolveDllName - RETURN: Status: 0xc0000022
LdrpResolveDllName - RETURN: Status: 0xc0000022
LdrpResolveDllName - RETURN: Status: 0xc0000022
LdrpResolveDllName - RETURN: Status: 0xc0000022

C:\logs>findstr "0xc0000022" "calc_without_symlink_hook.txt"

C:\logs>
```

The log shows that we see this error a few times, both when initializing the process, as well as when loading certain DLLs, whenever our driver is hooking the `C:` volume, but we don't see it at all when running without our hook.

At first, we were also puzzled as to why only certain DLLs were failing to load – then we realized most of the libraries needed by Calculator are "known DLLs". These is a special optimization Windows does wherein `Smss.exe` pre-maps these libraries at boot and caches their section objects in the `\KnownDlls` namespace of the Object Manager. The `LoadLibrary` API (more strictly speaking, its `LdrLoadDll` implementation in the loader) has special logic to always look for DLLs in this namespace first, and only try accessing the file system if it cannot find them there.

So, it seems like we found a hint to our problem, but what is the cause? We investigated Calculator's token to try and notice anything that our hook might've affected, which could've led to this access denied error:



Although the token looked the same with, and without, our driver, we did notice something obvious in hindsight – this process is running under an app container (Windows `8` 's new sandboxing technology) – as does the Start Menu, and each one of the other applications which were failing to execute! This made a lot of sense, as Microsoft is moving more and more applications into their new sandboxing model.

At first, we thought this would affect all Microsoft Store applications, but our assumption had been broken when WinDbg Preview launched fine. Now the reason made sense: it's a "Centennial" App, meaning it's a Windows Desktop Bridge UWP Application – and runs with full, regular privileges.

Well then, what's special about an app container? Among many other security restrictions, the default "traverse" checks which are performed by the I/O Manager work a little differently. You see, one of the things most people don't think about, is that to access a path such as `c:\windows\system32\spool\drivers\colors\foo.dat`, the Windows ACL-based security model technically dictates that " `c:` ", " `windows` ", " `system32` ", " `spool` ",

“ `drivers` ”, and “ `colors` ” should all be opened one by one, and that the current application’s token should be validated for `FILE_TRAVERSE` access to the directory. This is not only expensive but would also fail for privileged paths which contain user-accessible locations (such as this very example).

To solve this, Windows, by default, grants all users (even `Guests` !) the `SeChangeNotifyPrivilege` , which is a strange name for the “bypass traverse checking” privilege. As the name suggests, this causes the I/O Manager to bypass these expensive, likely failing, checks, and directly skip to checking the ACL of only the underlying file being accessed. And since app containers do run with a regular user token, even they get this privilege, as shown in the token screenshot above.

However, this may not be the desired behaviour for other types of device objects – remember that the I/O manager doesn’t only gate access to partitions, but all sorts of other virtual devices too, such as `\Device\Afd` to bring up one example, which represents Sockets, or `\Device\NamedPipe` , which is used for their namesake. Within these devices, there are internal paths as well, such as `\Device\NamedPipe\SomePipeName` . Because app containers are meant to provide strong security boundaries, the I/O manager implements a function, `IopDoFullTraverseCheck` , which we show below, in order to enforce certain restrictions:

```
BOOLEAN
IopDoFullTraverseCheck (
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_ KPROCESSOR_MODE AccessMode,
    _In_ PSECURITY_SUBJECT_CONTEXT SubjectSecurityContext
    )
{
    BOOLEAN isAppContainer;

    //
    // Check if AppContainer Traversal is allowed, or if the caller is from kernel-mode.
    //
    if (((DeviceObject->Characteristics & (FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL |
                                           FILE_DEVICE_SECURE_OPEN)) ==
                                           FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL) ||
        (AccessMode == KernelMode))
    {
        //
        // Don't do a full traverse check
        //
        return FALSE;
    }

    //
    // Otherwise, assume the caller is not AppContainer, unless it fails to open
    // a NULL DACL object (which is what the function below will check for).
    //
    isAppContainer = FALSE;
    (VOID) SeIsAppContainerOrIdentifyLevelContext(SubjectSecurityContext,
                                                  &isAppContainer);
    return isAppContainer;
}
```

As you can see, for user-mode callers, a full traverse check will always be done for an app container unless the device object has the `FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL` flag set. This helper routine is called deep in the guts of `IopParseDevice`, a function which we already talked about in Part 3 of this blog series and which we had referenced the ReactOS source code for. Unfortunately, as app container-related logic is new to Windows `8`, ReactOS can't offer much help here, so we'll have to go back to IDA. In the same if branch where the `VolumeOpen` checks are eventually done (which caused the crash in Part 2), we can now see some additional code, which we've reversed and shown below:

```
286         //
287         // Check if this is user-level access on a file on the volume, or a direct volume open
288         //
289         if ( (AccessMode || Context->Options & IO_FORCE_ACCESS_CHECK)
290           && (!RelatedFileObject || (_BYTE)VolumeOpen)
291           && !Override )
292         {
293             //
294             // Subject Context is not locked yet
295             //
296             localFlags &= ~IOP_PARSE_CONTEXT_LOCKED;
297
298             //
299             // Check if a file/directory on the volume is being opened, not the root itself
300             //
301             if ( RemainingName->Length )
302             {
303                 //
304                 // Check if this is an AppContainer and the device does not allow traversal
305                 //
306                 SecurityContext = &AccessState->SubjectSecurityContext;
307                 if ( IopDoFullTraverseCheck(
308                           DeviceObject,
309                           AccessMode,
310                           SecurityContext) )
311                 {
312                     //
313                     // Perform a full access check for FILE_TRAVERSE rights
314                     //
315                     localFlags ^= (IopCreateSecurityCheck(
316                                           DeviceObject,
317                                           NULL,
318                                           AccessState,
319                                           DesiredAccess | FILE_TRAVERSE,
320                                           FILE_OPEN,
321                                           Privileges,
322                                           &GrantedAccess,
323                                           CompleteName,
324                                           &ObjectTypeName,
325                                           Thread,
326                                           FALSE) ^ localFlags) & IOP_PARSE_TRAVERSE_ALLOWED;
327                     goto CheckResults;
328                 }
329
330                 //
331                 // Otherwise, check if the token has SeChangeNotifyPrivilege enabled
332                 //
333                 if ( AccessState->Flags & TOKEN_HAS_TRAVERSE_PRIVILEGE )
334                 {
335                     //
336                     // It does, so we can bypass access checks
337                     //
338                     localFlags |= IOP_PARSE_TRAVERSE_ALLOWED;
339 CheckResults:
340                     //
341                     // Unlock the subject context if needed
342                     //
343                     if ( localFlags & IOP_PARSE_CONTEXT_LOCKED )
344                         SeUnlockSubjectContext(SecurityContext);
345
346                     //
347                     // Return STATUS_ACCESS_DENIED if traverse checks failed
348                     // or the token does not have the privilege.
349                     //
350                     // Otherwise, continue the parse operation.
351                     //
352                     if ( !(localFlags & IOP_PARSE_TRAVERSE_ALLOWED) )
353                         goto ReturnAccessDenied;
```

```
354                     goto DoneTraverseChecks;
355                 }
356                 KeEnterCriticalRegionThread(&Thread->Tcb);
```

The check at line `307` is what calls the helper function shown earlier, which then results in a full `SeAccessCheck` being done by `IopCreateSecurityCheck`. As a side note, if you'd like to read some great research on these checks, and some of the abuses around bypassing them, James Forshaw has a great presentation at NullCon 2019 which you should read over.

In our case, this failed, because the ACL for `\Device\HarddiskVolume0` does not give `FILE_TRAVERSE` to the Calculator Package SID (or the `ALL_APPLICATION_PACKAGES` SID). While we could certainly add this, it would amount to a hack – the correct fix, which is what `\Device\HarddiskVolume3` itself has (our original partition device object), is to add the `FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL` flag when we call `IoCreateDevice`. Note the debugger output below that compares our device with the real device:

```
lkd> !devobj \Device\HarddiskVolume3
Device object (ffffdd0602606b90) is for:
HarddiskVolume3 \Driver\volmgr DriverObject ffffdd05ffd25e30
Current Irp 00000000 RefCount 16457 Type 00000007 Flags 00001150
Vpb ffffdd0602854e00 SecurityDescriptor ffffc80824a024e0 DevExt
ffffdd0602606ce0 DevObjExt ffffdd0602606ea8 Dope ffffdd0602854620 DevNode
ffffdd0602607bd0
ExtensionFlags (0x00000800) DOE_DEFAULT_SD_PRESENT
Characteristics (0x00020000) FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL
AttachedDevice (Upper) ffffdd060285e030 \Driver\fvevol
Device queue is not busy.

lkd> !devobj \Device\HarddiskVolume0
Device object (ffffdd0602db87b0) is for:
HarddiskVolume0 \Driver\symlink DriverObject ffffdd05fd9e64e0
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00000040
SecurityDescriptor ffffc808248a1aa0 DevExt 00000000 DevObjExt
ffffdd0602db8900
ExtensionFlags (0x00000800) DOE_DEFAULT_SD_PRESENT
Characteristics (0000000000)
Device queue is not busy.
```

So, while you all had to read another six pages of ranting, the only line of code that we had to fix is our call to IoCreateDevice:

```
status = IoCreateDevice(DriverObject,
                        0,
                        &g_DeviceName,
                        FILE_DEVICE_UNKNOWN,
-                       0,
```

```
+                       FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL,
                        FALSE,
                        &g_DeviceObject);
```

Well, there you have it! With this small fix, our hook driver now perfectly works on all the systems we've tested it on (about most of you, given the `ATMFD` RCE we've been using to deploy the driver). The final version is now posted on our GitHub here. Thanks a lot for reading!

Read our other blog posts: