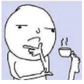
Life is Pane: Persistence via Preview Handlers

posts.specterops.io/life-is-pane-persistence-via-preview-handlers-3c0216c5ef9e-b73a9515c9a8

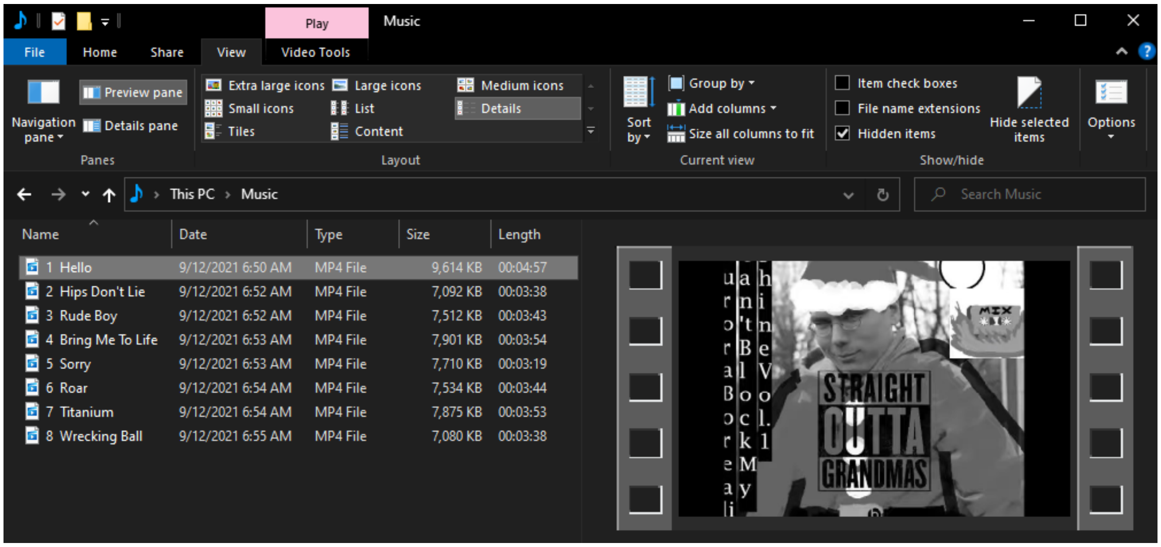
October 21, 2021



Matt Hand
Oct 21, 2021

People can have some strong preferences about how their files are laid out in Explorer. Some like the compact Details view. Others like the descriptive Content view with the Details pane. Some insane people even use Small Icons 😱. Explorer offers dozens of customizations to how Windows users can view the contents of the filesystem, but a feature which became particularly interesting to us was the Preview Pane.

The preview pane allows users to have a quick peek at the content of a selected file without actually having to open it. This feature is disabled on default Windows 10 builds, but can be enabled in the Explorer menu under View→Preview pane.



☐ Item check boxes
☐ File name extensions
☑ Hidden items

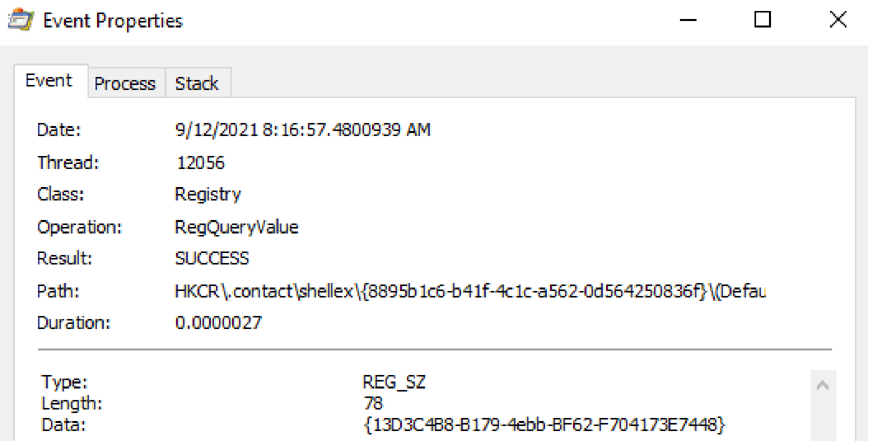
Name	Date	Type	Size	Length
1 Hello	9/12/2021 6:50 AM	MP4 File	9,614 KB	00:04:57
2 Hips Don't Lie	9/12/2021 6:52 AM	MP4 File	7,092 KB	00:03:38
3 Rude Boy	9/12/2021 6:52 AM	MP4 File	7,512 KB	00:03:43
4 Bring Me To Life	9/12/2021 6:53 AM	MP4 File	7,901 KB	00:03:54
5 Sorry	9/12/2021 6:53 AM	MP4 File	7,710 KB	00:03:19
6 Roar	9/12/2021 6:54 AM	MP4 File	7,534 KB	00:03:44
7 Titanium	9/12/2021 6:54 AM	MP4 File	7,875 KB	00:03:53
8 Wrecking Ball	9/12/2021 6:55 AM	MP4 File	7,080 KB	00:03:38

While this seems relatively simple at face value, it is anything but under the hood. For example, how does Windows know how to display the contents of certain filetypes but not others? Are the previews controlled by Explorer or is it done in another process? Are these handlers abusable? We spent a few days exploring preview handlers to gain a deeper understanding of how they work and answer these questions.

Behind the Pane

The first step in our research was to figure out exactly what was going on when Explorer wanted to present a preview of a file to the user. To start, we enabled the preview pane, navigated to a folder with filetypes which are known to display previews (we used .CONTACT filetypes as it's installed on Windows by default, but there are many more that could be used), launched Procmon and Process Hacker, and observed the system's behavior. While our findings aren't as complete as they could be, the general gist is as follows:

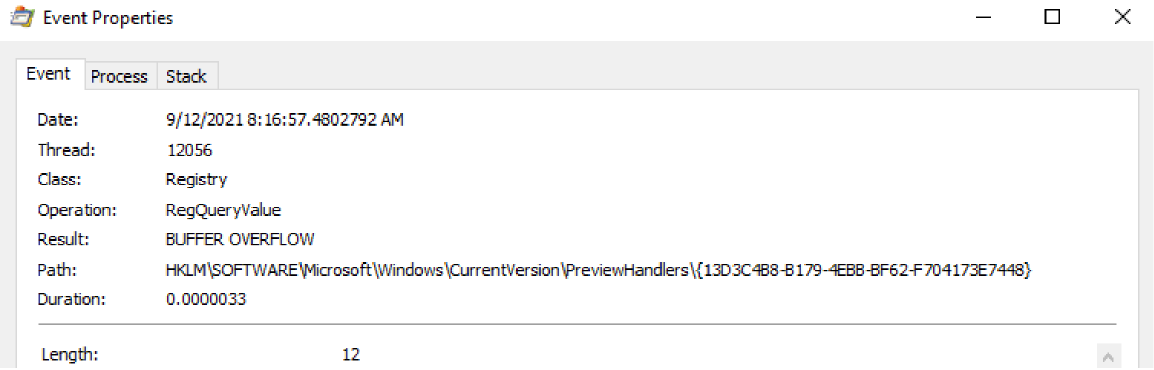
1. Explorer queries the preview handlers for the associated filetype identified by the subkey `{8895b1c6-b41f-4c1c-a562-0d564250836f}`, first in HKCU and then in HKCR, and takes its default value.



```
Date:        9/12/2021 8:16:57.4800939 AM
Thread:      12056
Class:       Registry
Operation:   RegQueryValue
Result:      SUCCESS
Path:        HKCR\.contact\shellex\{8895b1c6-b41f-4c1c-a562-0d564250836f}\(Defau
Duration:    0.0000027

Type:                      REG_SZ
Length:                    78
Data:                      {13D3C4B8-B179-4ebb-BF62-F704173E7448}
```

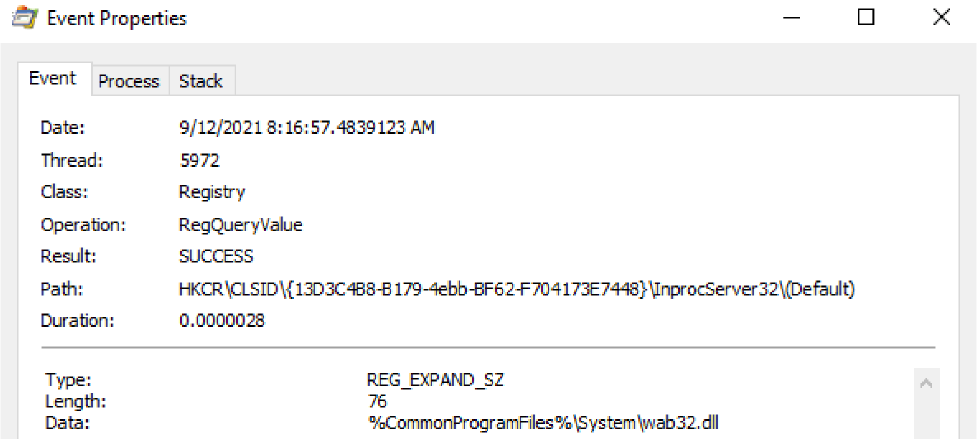
2. Explorer queries the value associated with the CLSID collected from the extension (`{13D3C4B8-B179-4ebb-BF62-F704173E7448}` for .CONTACT files) in the list of registered preview handlers. This list resides in `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers\` and is used as an optimization by the OS according to Microsoft.



```
Date:        9/12/2021 8:16:57.4802792 AM
Thread:      12056
Class:       Registry
Operation:   RegQueryValue
Result:      BUFFER OVERFLOW
Path:        HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers\{13D3C4B8-B179-4EBB-BF62-F704173E7448}
Duration:    0.0000033

Length:                    12
```

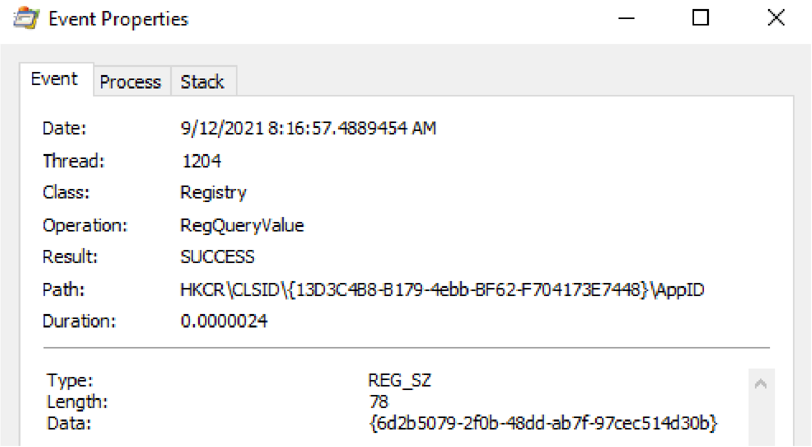
3. Explorer then queries the `InProcServer32` value of the CLSID.



```
Date:        9/12/2021 8:16:57.4839123 AM
Thread:      5972
Class:       Registry
Operation:   RegQueryValue
Result:      SUCCESS
Path:        HKCR\CLSID\{13D3C4B8-B179-4ebb-BF62-F704173E7448}\InprocServer32\(Default)
Duration:    0.0000028

Type:                      REG_EXPAND_SZ
Length:                    76
Data:                      %CommonProgramFiles%\System\wab32.dll
```

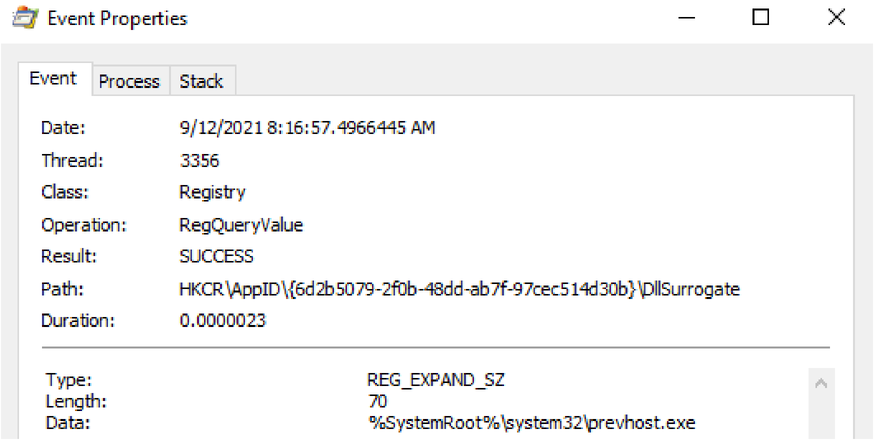
4. Explorer finally hands things off to the DCOM Server Process Launcher service (DcomLaunch) which collects the AppID associated with the CLSID.



Date: 9/12/2021 8:16:57.4889454 AM
Thread: 1204
Class: Registry
Operation: RegQueryValue
Result: SUCCESS
Path: HKCR\CLSID\{13D3C4B8-B179-4ebb-BF62-F704173E7448}\AppID
Duration: 0.0000024

Type: REG_SZ
Length: 78
Data: {6d2b5079-2f0b-48dd-ab7f-97cec514d30b}

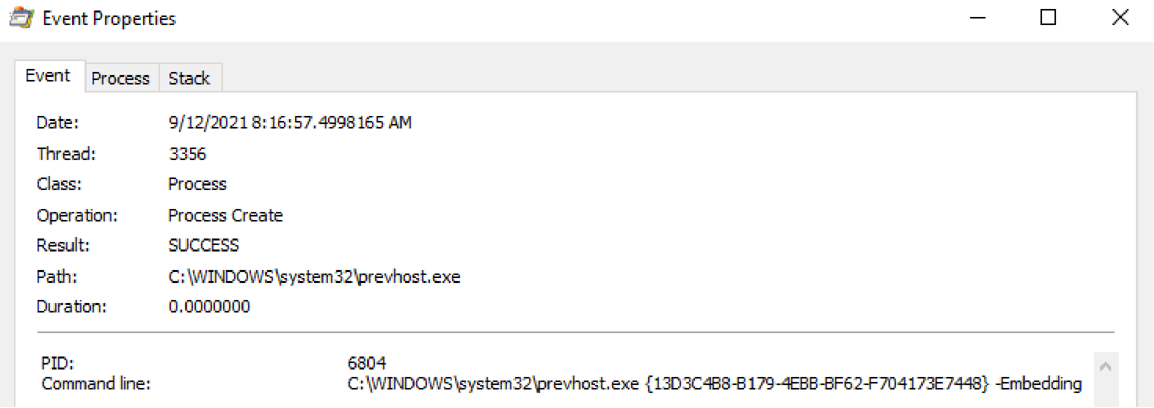
5. DcomLaunch references the `DllSurrogate` value of the associated AppID, located in `HKCR\AppID\`. Note that `{6d2b5079-2f0b-48dd-ab7f-97cec514d30b}` is the default for native x64 preview handlers. WOW64 handlers use `{534A1E02-D58F-44f0-B58B-36CBED287C7C}`.



Date: 9/12/2021 8:16:57.4966445 AM
Thread: 3356
Class: Registry
Operation: RegQueryValue
Result: SUCCESS
Path: HKCR\AppID\{6d2b5079-2f0b-48dd-ab7f-97cec514d30b}\DllSurrogate
Duration: 0.0000023

Type: REG_EXPAND_SZ
Length: 70
Data: %SystemRoot%\system32\prevhost.exe

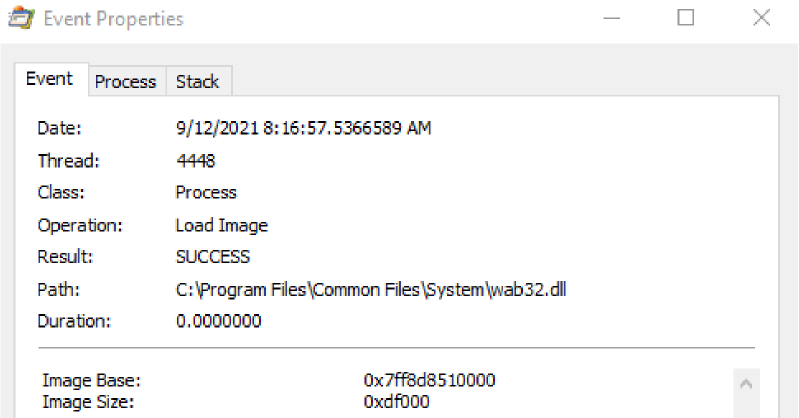
6. DcomLaunch then launches the surrogate process, `PREVHOST.EXE`, passing the command line arguments `{HANDLER-INPROCSERVER32-CLSID} -Embedding`.



Date: 9/12/2021 8:16:57.4998165 AM
Thread: 3356
Class: Process
Operation: Process Create
Result: SUCCESS
Path: C:\WINDOWS\system32\prevhost.exe
Duration: 0.0000000

PID: 6804
Command line: C:\WINDOWS\system32\prevhost.exe {13D3C4B8-B179-4EBB-BF62-F704173E7448} -Embedding

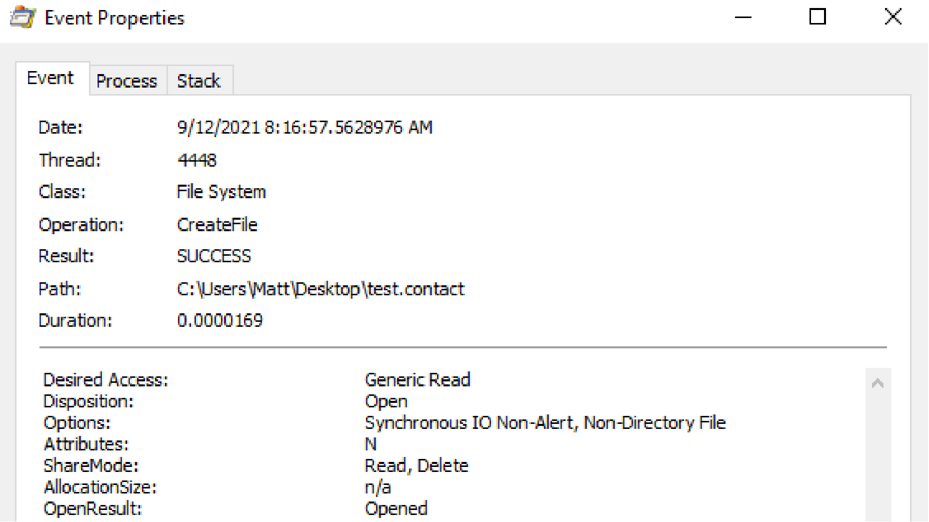
7. `PREVHOST.EXE` loads the in-process COM server referenced by the CLSID.



Date: 9/12/2021 8:16:57.5366589 AM
Thread: 4448
Class: Process
Operation: Load Image
Result: SUCCESS
Path: C:\Program Files\Common Files\System\wab32.dll
Duration: 0.0000000

Image Base: 0x7ff8d8510000
Image Size: 0xdf000

8. `PREVHOST.EXE` opens the file to be previewed.



Date: 9/12/2021 8:16:57.5628976 AM
Thread: 4448
Class: File System
Operation: CreateFile
Result: SUCCESS
Path: C:\Users\Matt\Desktop\test.contact
Duration: 0.0000169

Desired Access: Generic Read
Disposition: Open
Options: Synchronous IO Non-Alert, Non-Directory File
Attributes: N
ShareMode: Read, Delete
AllocationSize: n/a
OpenResult: Opened

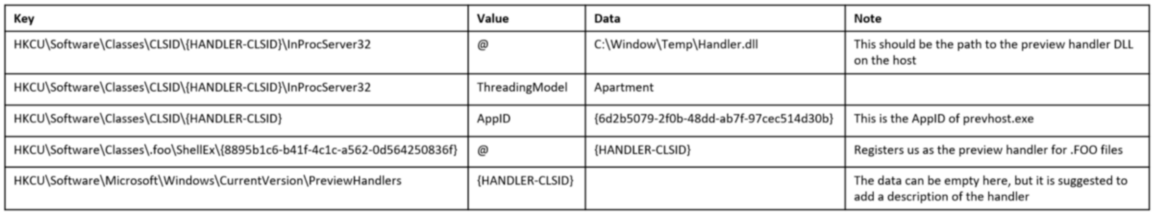
At this point, the preview handler DLL is mapped into the surrogate process, `PREVHOST.EXE` , and the file can be processed and passed back to Explorer's preview pane. As mentioned before, there are many minor details not covered both during and after loading the handle, but by this point we had a good idea how this could be abused.

Building Our Handler

Now that we had the general flow worked out, we could set out on building our own preview handler. Thankfully, Microsoft published some pretty robust documentation and sample code which we could reference. While the resources provided are extremely helpful, they are geared toward a developer writing production-ready preview handlers and contain a lot of bloat required to make them work properly (e.g. adaptively resizing the preview based on the preview pane's size).

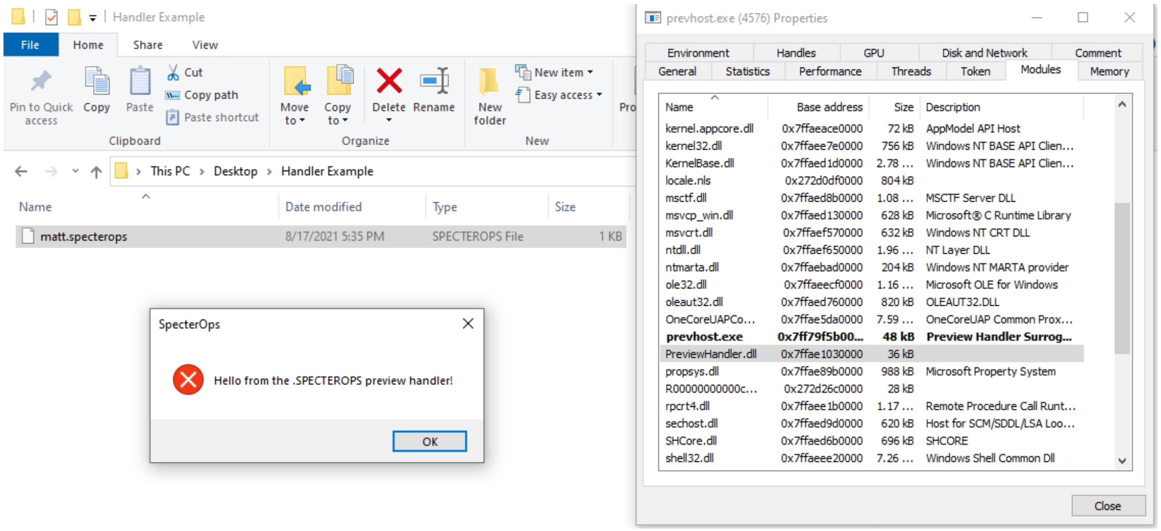
We really only needed a minimal example to test our theory, so we wrote a basic in-process COM server and implemented the `IPreviewHandler` and `IInitializeWithStream` interfaces as described in Microsoft's documentation. While Microsoft states that the `IObjectWithSite` , `IOleWindow` , and `IPreviewHandlerVisuals` interfaces also need to be implemented, we found that this is not the case when only code execution inside of the handler is required and the author doesn't care about rendering a full preview in the pane. To test our handler, the renderer function called by `IPreviewHandler::DoPreview()` simply spawns a message box.

As with all things COM-related, we were then off to the registry to build out all of the keys needed to get our handler running on the host. Again, Microsoft's documentation helped quite a bit here, but we weren't sure what was an actual requirement versus a best practice. What we found was that the following registry keys and values were required in order for our test message box to pop up:



Key	Value	Data	Note
HKCU\Software\Classes\CLSID\{HANDLER-CLSID}\InProcServer32	@	C:\Window\Temp\Handler.dll	This should be the path to the preview handler DLL on the host
HKCU\Software\Classes\CLSID\{HANDLER-CLSID}\InProcServer32	ThreadingModel	Apartment	
HKCU\Software\Classes\CLSID\{HANDLER-CLSID}	AppID	{6d2b5079-2f0b-48dd-ab7f-97cec514d30b}	This is the AppID of prevhost.exe
HKCU\Software\Classes\.foo\ShellEx\{8895b1c6-b41f-4c1c-a562-0d564250836f}	@	{HANDLER-CLSID}	Registers us as the preview handler for .FOO files
HKCU\Software\Microsoft\Windows\CurrentVersion\PreviewHandlers	{HANDLER-CLSID}		The data can be empty here, but it is suggested to add a description of the handler

At this point, we had a functioning minimal POC of the preview handler targeting .SPECTEROPS files from which we could build our capability.



Name	Date modified	Type	Size
matt.specterops	8/17/2021 5:35 PM	SPECTEROPS File	1 KB

SpecterOps

Hello from the .SPECTEROPS preview handler!

prevhost.exe (4576) Properties

Name	Base address	Size	Description
kernel.appcore.dll	0x7ffaeace0000	72 kB	AppModel API Host
kernel32.dll	0x7ffaee7e0000	756 kB	Windows NT BASE API Clien...
KernelBase.dll	0x7ffaed1d0000	2.78 ...	Windows NT BASE API Clien...
locale.nls	0x272d0df0000	804 kB	
msctf.dll	0x7ffaed8b0000	1.08 ...	MSCTF Server DLL
msvcp_win.dll	0x7ffaed130000	628 kB	Microsoft® C Runtime Library
msvcrt.dll	0x7ffaef570000	632 kB	Windows NT CRT DLL
ntdll.dll	0x7ffaef650000	1.96 ...	NT Layer DLL
ntmarta.dll	0x7ffaebad0000	204 kB	Windows NT MARTA provider
ole32.dll	0x7ffaeecf0000	1.16 ...	Microsoft OLE for Windows
oleaut32.dll	0x7ffaed760000	820 kB	OLEAUT32.DLL
OneCoreUAPCo...	0x7ffae5da0000	7.59 ...	OneCoreUAP Common Prox...
prevhost.exe	**0x7ff79f5b00...**	**48 kB**	**Preview Handler Surrog...**
PreviewHandler.dll	0x7ffae1030000	36 kB	
propsys.dll	0x7ffae89b0000	988 kB	Microsoft Property System
R00000000000c...	0x272d26c0000	28 kB	
rpcrt4.dll	0x7ffaee1b0000	1.17 ...	Remote Procedure Call Runt...
sechost.dll	0x7ffaed9d0000	620 kB	Host for SCM/SDDL/LSA Loo...
SHCore.dll	0x7ffaed6b0000	696 kB	SHCORE
shell32.dll	0x7ffaeee20000	7.26 ...	Windows Shell Common Dll

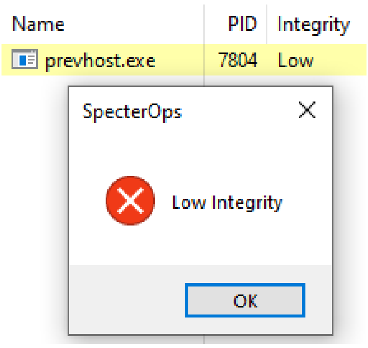
Leaving Low IL

The biggest hurdle with this technique is that, by default, preview handlers are run in a low integrity instance of `PREVHOST.EXE` . This means that even though we can get code execution, our token's integrity level (IL) will limit us from accessing important parts of the operating system in the context of post-exploitation activities.

Thankfully for us, Microsoft realized that there are plenty of cases where running in low IL just won't work for some developers (e.g. needing to save a file to a directory marked with medium integrity label). To support these use cases, developers are allowed to opt out of the low IL isolation behavior and instead be hosted in a medium IL instance of the `PREVHOST.EXE` surrogate process. In order to opt out, Microsoft instructs developers to create a new value, `DisableLowILProcessIsolation` , under `HKCR\CLSID\{PREVIEW-HANDLER-CLSID}` and set the value to `1` .

Because HKCR is really just a combination of `HKCU\Software\Classes` and `HKLM\Software\Classes` , the developer should theoretically be able to register the preview handler under the context of the current user by creating the required registry keys and setting the values under `HKCU` . Then their handler will be executed in a medium IL surrogate whenever the user previews their chosen file type. We tested this assumption by adding the `DisableLowILProcessIsolation` value to the preview handler we had previously registered. After setting the value and refreshing the preview pane, we found that we were still running in low IL.

To try to figure out what was going on, we opened up Procmon and set a filter for registry operations whose path ended with `DisableLowILProcessIsolation` . We refreshed the preview pane but didn't see anything. After trying a few other file types, Procmon eventually caught `EXPLORER.EXE` querying the value of this key. The call stack for this event can be seen below.

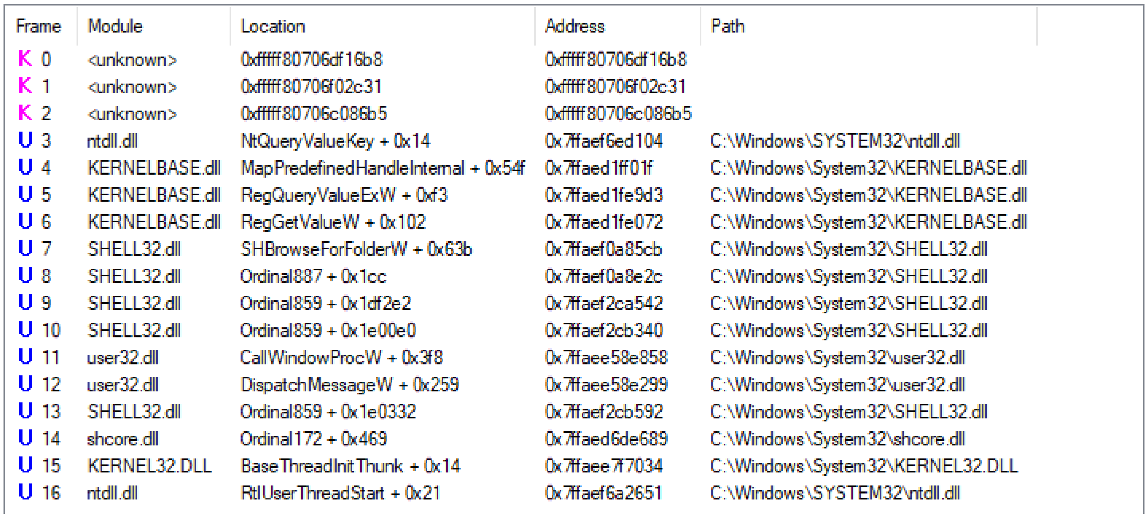


Name	PID	Integrity
prevhost.exe	7804	Low

SpecterOps

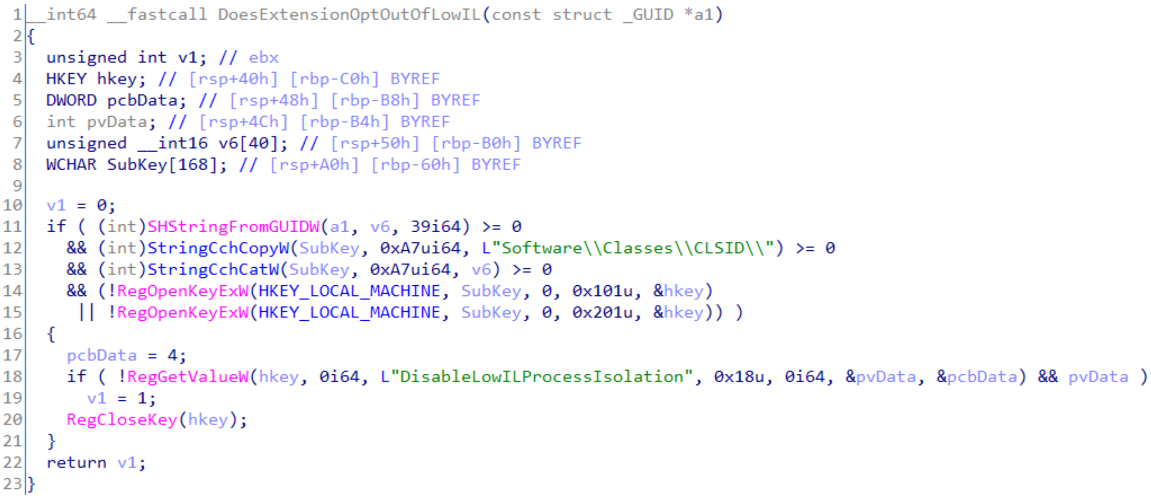
Low Integrity

We replaced the original message box with one which shows the token's integrity level



Frame	Module	Location	Address	Path
K 0	<unknown>	0xfffff80706df16b8	0xfffff80706df16b8	
K 1	<unknown>	0xfffff80706f02c31	0xfffff80706f02c31	
K 2	<unknown>	0xfffff80706c086b5	0xfffff80706c086b5	
U 3	ntdll.dll	NtQueryValueKey + 0x14	0x7ffaef6ed104	C:\Windows\SYSTEM32\ntdll.dll
U 4	KERNELBASE.dll	MapPredefinedHandleInternal + 0x54f	0x7ffaed1ff01f	C:\Windows\System32\KERNELBASE.dll
U 5	KERNELBASE.dll	RegQueryValueExW + 0xf3	0x7ffaed1fe9d3	C:\Windows\System32\KERNELBASE.dll
U 6	KERNELBASE.dll	RegGetValueW + 0x102	0x7ffaed1fe072	C:\Windows\System32\KERNELBASE.dll
U 7	SHELL32.dll	SHBrowseForFolderW + 0x63b	0x7ffaef0a85cb	C:\Windows\System32\SHELL32.dll
U 8	SHELL32.dll	Ordinal887 + 0x1cc	0x7ffaef0a8e2c	C:\Windows\System32\SHELL32.dll
U 9	SHELL32.dll	Ordinal859 + 0x1df2e2	0x7ffaef2ca542	C:\Windows\System32\SHELL32.dll
U 10	SHELL32.dll	Ordinal859 + 0x1e00e0	0x7ffaef2cb340	C:\Windows\System32\SHELL32.dll
U 11	user32.dll	CallWindowProcW + 0x3f8	0x7ffaee58e858	C:\Windows\System32\user32.dll
U 12	user32.dll	DispatchMessageW + 0x259	0x7ffaee58e299	C:\Windows\System32\user32.dll
U 13	SHELL32.dll	Ordinal859 + 0x1e0332	0x7ffaef2cb592	C:\Windows\System32\SHELL32.dll
U 14	shcore.dll	Ordinal172 + 0x469	0x7ffaed6de689	C:\Windows\System32\shcore.dll
U 15	KERNEL32.DLL	BaseThreadInitThunk + 0x14	0x7ffaee7f7034	C:\Windows\System32\KERNEL32.DLL
U 16	ntdll.dll	RtlUserThreadStart + 0x21	0x7ffaef6a2651	C:\Windows\SYSTEM32\ntdll.dll

Procmon's symbol resolution is a little off here. Frame 7 (`SHELL32!SHBrowseForFolder+0x63b`) is the most interesting for us as it resolves to an address inside of the function `SHELL32!DoesExtensionOptOutOfLowIL` . The disassembly of this function can be seen below.



```
__int64 __fastcall DoesExtensionOptOutOfLowIL(const struct _GUID *a1)
{
  unsigned int v1; // ebx
  HKEY hkey; // [rsp+40h] [rbp-C0h] BYREF
  DWORD pcbData; // [rsp+48h] [rbp-B8h] BYREF
  int pvData; // [rsp+4Ch] [rbp-B4h] BYREF
  unsigned __int16 v6[40]; // [rsp+50h] [rbp-B0h] BYREF
  WCHAR SubKey[168]; // [rsp+A0h] [rbp-60h] BYREF

  v1 = 0;
  if ( (int)SHStringFromGUIDW(a1, v6, 39i64) >= 0
    && (int)StringCchCopyW(SubKey, 0xA7ui64, L"Software\\Classes\\CLSID\\") >= 0
    && (int)StringCchCatW(SubKey, 0xA7ui64, v6) >= 0
    && (!RegOpenKeyExW(HKEY_LOCAL_MACHINE, SubKey, 0, 0x101u, &hkey)
     || !RegOpenKeyExW(HKEY_LOCAL_MACHINE, SubKey, 0, 0x201u, &hkey)) )
  {
    pcbData = 4;
    if ( !RegGetValueW(hkey, 0i64, L"DisableLowILProcessIsolation", 0x18u, 0i64, &pvData, &pcbData) && pvData )
      v1 = 1;
    RegCloseKey(hkey);
  }
  return v1;
}
```

Looking at the disassembly, it immediately became clear what was going wrong — only values in keys under `HKLM` are checked. This means that not only are our hopes of dropping per-user persistence dead, but because `HKLM` is only writable by administrators, we can't even get around the low IL isolation as a normal user. We explored Microsoft's directions for providing a separate surrogate process to host our handler, but ultimately that effort failed as those processes also spawns as low IL.

Although this wasn't the ideal outcome, we still had a persistence mechanism with which we could host our code inside of a Microsoft-signed executable. Because of the privileges required, this will rarely see any use on initial compromise and instead will be used later in the attack chain as we gain more privileged footholds in the environment. Additionally, because we're targeting `HKLM`, all users of the system will be affected and not just the current user.

Operationalization

In order to take full advantage of this technique, our tooling is broken up into three distinct components — a payload, a target file, and a dropper.

- The handler DLL which will be loaded in the surrogate and begin executing our malicious code. This is dropped to disk in a user-defined location.
- Any file with an extension matching the one we've set up our handler for. This is also dropped to disk in an arbitrary location, but one where the user will likely browse with Explorer.

The runner function inside of the handler was swapped from the testing message box to a shellcode runner. While we'll leave this as an exercise to the reader, there are a few hang-ups with this technique that are worth covering.

1. Explorer must restart after programmatically enabling the preview pane or the handler won't fire
2. Some type of mutex is needed as multiple instances of the handler can spawn unexpectedly and you'll be flooded with agents
3. Existing handler can be hijacked relatively seamlessly, but reverting the change isn't always as simple as it appears due to differences in implementations (e.g. Word uses different ProgIDs based on the file extension instead of CLSIDs)

Proof of Concept

Detection

Detection of this technique relies heavily on monitoring changes to the registry. During our development of this technique, we identified the base conditions for implementing a handler for persistence. While there are a good amount of constants that we can monitor, this technique provides the actor with many chances to subvert detection logic (e.g. using ProgIDs instead of CLSIDs). We'll first highlight the conditions from which a basic detection can be built and then discuss some of the qualifying conditions that can be used to make the detection more robust.

An important observation identified is that actors are not required to implement a preview handler for new file extensions and can just as easily hijack existing handlers following roughly the same methodology.

Base Conditions

In order for this technique to both function and be useful for attackers (i.e. not executing in low IL), we can focus our attention on a few specific registry keys while building out the base detection.

The first and most important event to monitor for the base detection is the value `DisableLowILProcessIsolation` being set to `1` in any key in `HKLM\Software\Classes\CLSID\*`. This key must be set in HKLM in order for the surrogate process to be launched in medium IL, allowing the actor to interact with the compromised host as a normal user. While the scope of this event is relatively large, we found that instances of this value being set are exceedingly rare during our testing.

The second registry key to target as part of the detection is `HKCR\Software\Classes\*\ShellEx\{8895b1c6-b41f-4c1c-a562-0d564250836f}` . The creation of this key is a base condition of installing *any* preview handler on the system. This can be for any filetype, including both existing or new extensions, but it must be set. Note that it is important to use the wildcard filter as written and not to scope it as `.*` in an attempt to restrict detections to only file extensions. This because filetypes (e.g. `.foo`) in the registry can be associated with ProgIDs (e.g. `foo`) which function the same in the context of this technique.

The final base condition is adding the CLSID of the preview handler as a value to the `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers` key for per-user installation or `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers` for all users. This value will specify the friendly name of the preview handler for debugging but it is not required to be set.

Classifying Conditions

While the base conditions provide a minimal amount of events which can be used to identify the installation of preview handlers, there are a number of others that can provide supplemental context to the base detection, are conditional, or may aid in an investigation.

The first of these conditions is that Explorer's preview pane must be enabled. This is a requirement for this technique to work. This should be considered an optional event because the victim may already have enabled this feature, meaning that a registry event won't occur. Normally disabled by default, enabling the preview pane can provide a consistent choke point where we can observe an actor enabling it manually. To do this, we can monitor `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Modules\GlobalSettings\DetailsContainer` for the `DetailsContainer` value being set to `02 00 00 00 01 00 00 00` and `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Modules\GlobalSettings\Sizer` for the value of `DetailsContainerSizer` being set to `15 01 00 00 01 00 00 00 00 00 00 00 6d 02 00 00` . These keys should only be set by `EXPLORER.EXE` under normal conditions.

Another event which can provide additional context is the surrogate process being set. This process is `PREVHOST.EXE` under normal conditions, but the actor may register their own on the system to allow for a custom application to handle the preview such as in the instance of a managed handler which needs to load a specific version of the Common Language Runtime (CLR). This provides a valuable way for an actor to evade process or image-based detections. This AppID should be set in `HKLM\Software\Classes\CLSID\*` as the value `AppID` . If the default `PREVHOST.EXE` is to be used, this should be `{6d2b5079–2f0b-48dd-ab7f-97cec514d30b}` for x64 handlers or `{534a1e02-d58f-44f0-b58b-36cbed287c7c}` for x86 handlers running on an x64 host. If the actor opts to use their own surrogate application, a key `HKLM\Software\Classes\AppId\*` must be created with the value `DllSurrogate` set to the path of their custom application. Regardless of the surrogate process, it will always be launched with the command line arguments `{MALICOUS-HANDLER-CLSID} -Embedded` where the CLSID is that of the registered handler DLL.

The final piece which could provide value, especially during an investigation, is the registration of the preview handler DLL itself. This will be set as the default value in `HKLM\Software\Classes\CLSID\*\InProcServer32` and will point to a path on disk. It is worth noting that this event can be relatively noisy compared to the other events generated by usage of this persistence technique. Additionally, this file doesn't have to exist at the time of installation and can be dropped whenever the actor is ready to operationalize the persistence mechanism.

Detection Operationalization

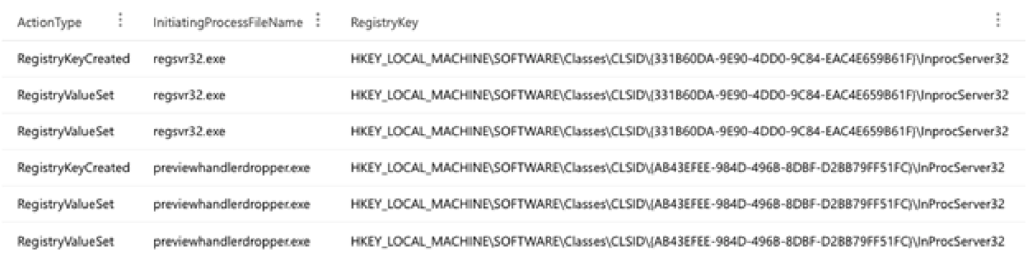
In testing the detection piece of this persistence technique, we stood up a Microsoft Defender instance in which we ran a number of Kusto queries containing the base condition syntax listed above. We wanted to highlight the following blindspots that we believe are inherent to Microsoft Defender's default filtering configurations, and are worth considering when building and tuning a MDE detection in your environment:

- —This prevented us from detecting the specific `DisableLowILProcessIsolation` registry value set to 1 as part of the first base detection. We were able to detect automatic changes in this registry key, such as benign changes from `MSIEXEC.EXE` .
- — Since HKCR is a combination of the HKLM and HKCU hives, which MDE does log, the root cause of this visibility gap is unknown at the time of this post. This gap prevents the second base condition key, `HKCR\Software\Classes\*\ShellEx\{8895b1c6-b41f-4c1c-a562-0d564250836f}` , from being detected in MDE.
- — This exists in either registry key `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers` or `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers` . This gap prevents the third base condition from being detected as written in MDE.

With Microsoft Defender's current log forwarding gaps, detection engineers will have the most success corroborating high-level detections and pivoting to investigate suspicious activity. Below is a starting query that detects one of the classifying conditions, the registration of the preview handler DLL itself in the `HKLM\Software\Classes\CLSID\*\InProcServer32` key (**Note:** the exclusions included were specific to our test environment and would need further tuning depending on your organization's environment):

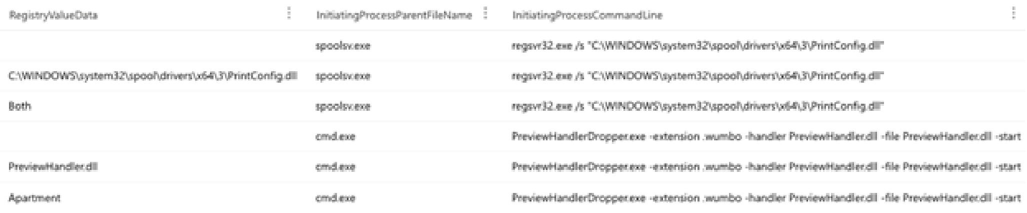
```
DeviceRegistryEvents| where RegistryKey has @"HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\" and RegistryKey
endswith "InProcServer32"| where InitiatingProcessFileName !in ("setup.exe", "wzpreviewer65.exe",
"winzip64.exe",
"msiexec.exe","wzpreviewer64.exe","wzbgtcomserver64.exe","msmpeng.exe","microsoftedgeupdatecomregistershe
 project Timestamp, DeviceName, ActionType, InitiatingProcessFileName, RegistryKey, RegistryValueType,
RegistryValueName, RegistryValueData, InitiatingProcessParentFileName, InitiatingProcessCommandLine
```

This detection example will detect the following Registry Keys:



ActionType	InitiatingProcessFileName	RegistryKey
RegistryKeyCreated	regsvr32.exe	HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{331B60DA-9E90-4DD0-9C84-EAC4E659B61F}\InprocServer32
RegistryValueSet	regsvr32.exe	HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{331B60DA-9E90-4DD0-9C84-EAC4E659B61F}\InprocServer32
RegistryValueSet	regsvr32.exe	HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{331B60DA-9E90-4DD0-9C84-EAC4E659B61F}\InprocServer32
RegistryKeyCreated	previewhandlerdropper.exe	HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{AB43EFEE-984D-4968-8DBF-D2BB79FF51FC}\InProcServer32
RegistryValueSet	previewhandlerdropper.exe	HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{AB43EFEE-984D-4968-8DBF-D2BB79FF51FC}\InProcServer32
RegistryValueSet	previewhandlerdropper.exe	HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{AB43EFEE-984D-4968-8DBF-D2BB79FF51FC}\InProcServer32

And the following Registry Value Data and Command Line activity:



RegistryValueData	InitiatingProcessParentFileName	InitiatingProcessCommandLine
	spoolsv.exe	regsvr32.exe /s "C:\WINDOWS\system32\spool\drivers\x64\3\PrintConfig.dll"
C:\WINDOWS\system32\spool\drivers\x64\3\PrintConfig.dll	spoolsv.exe	regsvr32.exe /s "C:\WINDOWS\system32\spool\drivers\x64\3\PrintConfig.dll"
Both	spoolsv.exe	regsvr32.exe /s "C:\WINDOWS\system32\spool\drivers\x64\3\PrintConfig.dll"
	cmd.exe	PreviewHandlerDropper.exe -extension .wumbo -handler PreviewHandler.dll -file PreviewHandler.dll -start
PreviewHandler.dll	cmd.exe	PreviewHandlerDropper.exe -extension .wumbo -handler PreviewHandler.dll -file PreviewHandler.dll -start
Apartment	cmd.exe	PreviewHandlerDropper.exe -extension .wumbo -handler PreviewHandler.dll -file PreviewHandler.dll -start

Further research is needed to determine whether or not a viable detection could be built querying Sysmon logs or Windows Registry events.

Acknowledgements

Thank you to Casey Smith for giving us the tip to explore preview handlers in the first place. Thank you to Dwight Hohnstein for collaborating with us to turn the handler into something usable.