Thomas Betterly

8/23/2023

Introduction:

A curious interaction can be observed in the combination of how Windows deals with debugger and debugged process communication, structured exception handling, and Microsoft Detours. This interaction can work even if the debugger is running in the .NET Runtime due to the case of said runtime *still* being a Windows Process. The .NET Runtime functionally acts as a simulated machine that compilers for .NET may target. An application that targets this Runtime is known as a managed application. The structured exception handling gives a debugger that's using the Windows API to maintain communication first dibs on exceptions that happen before the debugged software. This is true regardless of if the user of the Windows Debug API is running as .NET based app or an unmanaged app. With that in mind, there exists with Microsoft Detours library a method to spawn processes with a specially written DLL to enable detouring something in the unmanaged application world. This document is going to example several components of these items.

The plan is to combine these things to form a way to look at calls for routines of interest. We'll go onto that a little later. For example, CreateFileW is the Unicode version of the Windows API routine CreateFile. It's going to serve as an example. On input, it takes a few arguments and returns a HANDLE to the open file provided by the OS and possibly sets a 4 byte sized value indicating error and retrievable by calling the win32 routine GetLastError(). Our detoured routine can still call the original and indeed probably should, but we can pack the arguments of the routine into ana array suitable to pass to RaiseException(), another routine in the Win32 API that quite unsurprisingly raises a software level exception, - something that our debugging code gets first dibs on. While it's handling the exception that contains information about the CreateFileW call, the debugged software is frozen by Windows until the debugger calls ContinueDebugEvent(), another routine in the Win32 API. With some ways to Duplicate Handles into the debugged process and read/write memory from the same process, a debugger is free to let the call go thru or swap out the returned HANDLE our replacement CreateFileW routine gives to the debugged app with something else or the bad handle value (INVALID_HANDLE_VALUE).

A C# based unmanaged application environment.

The .NET Runtime is C#'s target virtual machine. When a C# compiles, it compiles to that machine. One of the best perks about this is that should someone need to swap out the said virtual machine, there should be little issue in getting the code running on a new one. The .NET Runtime abstracts away a lot of things for the programmer to let she or he focus on the code part of their software, what it does rather than say how to allocate memory or interface with the system. A C# program using the File static class to open and deal with files on Windows is ideally going to get the same action for a C# program using the same class on a Linux or Mac System. This paired with a quick build time makes C# and Visual Studio nifty.

One vital important thing to note is that although the .NET runtime does all these things, it's still subject to running a process on the machine of choice and subject to making calls into the operating system. For example, opening a file in C# with "File.Open" may just end up making a call to the Win32 unmanaged routine CreateFileW once the .NET Runtime is done implementing the action. There's also going to be different code executing in the process depending on if the C# runtime targets x86 code or x64 code. This also includes different pointer sizes.

Most remotely modern instances of C# provide ways to directly interface with the underlying system via Platform Invoke and access to pointers via functionally boxing the pointer code into 'unsafe' context. Platform Invoke or PInvoke from here on out is a way to define imports from the native or code that that not machined by the .NET Runtime. Below is the static import of a couple of routines imported via DllImport in the .NET Runtime.

```
    internal static partial class NativeMethods
     {
         [DllImport("InsightAPI.dll",CallingConvention= CallingConvention.Winapi,
EntryPoint = "ImageHlp_GetBaseOfImage")]
         public static extern ulong ImageHlp64_GetBaseOfImage(IntPtr
ImageHlpModule64);

[DllImport("InsightApi.Dll", BestFitMapping = false, CallingConvention =
CallingConvention.Winapi, CharSet = CharSet.Unicode, EntryPoint =
"ThreadContext_SetThreadDescriptionW")]
         public static extern bool ThreadContext_SetTheadDescriptionW(IntPtr That,
[MarshalAs(UnmanagedType.LPWStr)] string NewDescription);
     }
```

There are a few points to pick up from this example. Both are decorated with DllImport tags that let the programmer dictate how the system will link to this routine.   The ThreadContext_SetThreadDescriptW also has a MashalAs tag for its string. This tells the runtime exactly how to send the string off to the unmanaged or native routine. In this case, it is sending string as an Unicode String.   Both 'That' arguments are filling in as a reference to a relevant memory block and will end up being 8 bytes on x64 bit code and 4 bytes on x86 code. The reason for this is the IntPtr type. IntPtr is one way to tell the .NET Runtime to pick an argument size to hold a pointer to an Int in memory. One thing to pick up from this is that even with the .NET Runtime doing a significant amount of heavy lifting, it is important to have a passing idea of how pointers will work in the environment the .NET Runtime will be executing with.

One may notice that I skipped the unsafe implementation. I'm still learning with it myself; however, the important aspect is that functions and code marked unsafe are given the green light to deal with pointers directly – unlike code that doesn't have that mark.   You'll see later through a couple of routines called RemotePoke4 and RemotePoke8 that dealing with pointers underneath the .NET runtime is a possibility. Pointers are a powerful tool and using them in C# is no exception.

First Component: The Windows Debugging System:

Now let's move on to the Windows System Debugger API. It's through this our debugged app is going be triggering events and communicating with our debugger about any special messages. When a debugger is monitoring a process, it sets up a message pump not unlike the usually Win32 Message pump of GetMessage() and ProcessMessage(). The workflow for the debugger app is either launch the process with a flag set to indicate to Windows it wants to be a debugger for this app or request to be attached to an already running. When a debugger does this, a link in Windows OS is associated with each process. The Debugger can call WaitForDebugEvent() or WaitForDebugEventEx() which can pause it until Windows sees a debug event. When that happens, Windows will pause all the threads in the debugged software and fill out an unmanaged data struct called DEBUG_EVENT. The Debugger can receive a lot of info about the triggered event and exactly where it came from in the debugged process. Those events range from when the debugged software starts to loading a DLL, to triggered exceptions and more. The full list of debug events is located at the DEBUG_EVENT struct link at the bottom of this article. The debugger gets to inspect this data structure presented by Windows, do what it wants to do and signs off via calling ContinueDebugEvent(). One **critical** thing of note is that for processes that are spawned to be debugged, only the thread in the debugger that spawned the process can create it.

Exceptional Exceptions:

The scope of this article is going to be focused on exceptions due to what it is leading up to. Exceptions can be triggered for all kinds of reasons and a Windows process can Raise an Exception programmatically via calling a Win32 routine named RaiseException(). According to RaiseException's documentation a debugger will functionally **always** get a first look at the exception and be able to deal with it before the debugged process. A Debugger could for example, log the event, attempt to fix it, or kill the offending process. RaiseException() also lets the process communicate with an exception handler (or debugger) via arguments too. It can get up to 15 (EXCEPTION_MAXIMUM_PARAMETERS) ULONG_PTR sized values to pass to an exception handler. For x86 code, this is an array of at most fifteen 4-byte sized values. For x64 bit code, these are going to be at most fifteen 8-byte sized values. These values will be passed to whatever tries to handle the exception. One **critical** thing to note is that while numbers (and pointers) passed this way will remain unaltered, pointers passed this way are valid **only** in the process they originated from. It's one of the perks of how Virtual Memory works. If I'm attempting to directly dereference a pointer received by my debugger from another process, it's going to not go well.

Pointers, Code Considerations and more:

Luckly, Windows acts as the middle man for this also by providing two routines of interest, ReadProcessMemory() and WriteProcessMemory(). I've included their C prototypes below as seen in their online document from Microsoft below. I'm going to break the arguments down with respect to x86 and x64 bit code.

```
BOOL ReadProcessMemory(

  [in]  HANDLE  hProcess,
  [in]  LPCVOID lpBaseAddress,

  [out] LPVOID  lpBuffer,

  [in]  SIZE_T  nSize,

  [out] SIZE_T  *lpNumberOfBytesRead

);
```

```
BOOL WriteProcessMemory(

  [in]  HANDLE  hProcess,

  [in]  LPVOID  lpBaseAddress,

  [in]  LPCVOID lpBuffer,

  [in]  SIZE_T  nSize,

  [out] SIZE_T  *lpNumberOfBytesWritten

);
```

ReadProcess will take 4 pointer sized arguments and one SIZE_T sized argument. But wait! Where is the 4th one? It's sitting in the HANDLE passed to indicate which process. HANDLEs are the opaque data values one gets from the Windows OS to interact with a resource. Once good thing about HANDLEs are that the user code need not know the details in how to access with the HANDLE. It just passes the value Windows provides after Windows gives the OK to access the resource such as ACLs and SACLs. HANDLEs are valid in the context of which process opened it unless using something like DuplicateHandle() or they were spawned to have inherited HANDLEs from the parent process.

What this means for consideration of x86 vs x64 code is that our debugger needs to know its own pointer size and the debugged software's pointer size. Promoting an x86 pointer to x64 while working with it isn't going to break stuff provided care is used to remember that when putting the promoted x86 pointer back into its original x86 size, its value is not going to be truncated. Going into a lower level to solve this could be linking to both versions of ReadProcessMemory/WriteProcessMemory from the x86 kernel32 DLL code and x64 kernel32 DLL. That's outside the scope of what I wanted to do for this project. I went with having C# target x64 bit code while promoting x86 pointers to x64 bit in the .NET runtime/C# and demoting back to x86 when moving them back. Below is an example of C routines that C# could link to put pointers back in a debugged process.

```
BOOL WINAPI RemotePokeCommon(HANDLE Process, ULONG64 Value, VOID* RemoteLocation,
SIZE_T Size) noexcept
       {
               SIZE_T BytesWrote = 0;
               if ((RemoteLocation != 0) && (Process != 0) && ((Size >= 1) && (Size
<= sizeof(ULONG64))))
               {
                       if (WriteProcessMemory(Process, RemoteLocation, &Value, Size,
&BytesWrote))
```

```
                        {
                                if (BytesWrote == Size)
                                {
                                        return TRUE;
                                }
                        }
                }
                return FALSE;
        }
```

```
        BOOL WINAPI RemotePoke8(HANDLE Process, ULONG64 Value, VOID*
RemoteLocation) noexcept
        {
                return RemotePokeCommon(Process, Value, RemoteLocation,
sizeof(ULONGLONG));
        }
```

```
        BOOL WINAPI RemotePoke4(HANDLE Process, DWORD Value, VOID* RemoteLocation)
noexcept
        {
                return RemotePokeCommon(Process, Value, RemoteLocation,
sizeof(DWORD));
                }
```

The variant routines RemotePoke4 and RemotePoke8 converge into a single routine, RemotePokeCommon. The difference between them is the size of the pointer they pass to the common routine. One thing to notice is that the remote location is a VOID*/ untyped pointer rather than something like ULONG64. This means that its upper size is dependent on if our C# app is running under x86 code or x64 code. In Practice, this means that x64 C# code calling this function can write to either an x64 or x86 as long as they know the pointer size in the target process ahead of time. However, should the C# code have x86 sized pointers, it's likely **not** going to be able to write x64 sized pointers to the debugged process. Another thing to note is that RemotePoke8 functionally takes similar arguments to the RemotePokeCommon routine, which is a potential solution talked about above in dealing with x86/x64 code.

A Little Detour and Process loading.

Microsoft Detours or just Detours from here on out is a way to insert small 'detours' in an unmanaged/native routine while preserving an execution path to the original. This small bypass can be used to get information on arguments being passed, modify and extern the original routine based on needs and more. It's very flexible too and has functionality to force a DLL that's programmed to detour select routines to be loaded into a newly spawned process. This vital tool is how we can work towards what we can do.

Combining the Tools: The Detour DLL.

Our finale, however, is going to go through a different routine. Remember how a debugger gets first dibs on any exceptions generated from earlier? Windows also provides a way to generate arbitrary exceptions via RaiseException(). Now RaiseException lets one specify exactly what will be passed to the exception handler via an array. The plan is going to pack identifying information for the routine, for example CreateFileW, into an array suitable for the RaiseException routine. We also pack identifying information to let the debugger know it is an exception generated for CreateFileW and a couple of writable pointers to let the debugger have an easy way to modify what's returned to the debugged software. Some example code is below for CreateFileW. I'll explain what the code does following it.

```
DWORD __CreateFileAW_CommmonAlert(
       LPCWSTR                lpFileName,
       DWORD                  dwDesiredAccess,
       DWORD                  dwShareMode,
       LPSECURITY_ATTRIBUTES lpSecurityAttributes,
       DWORD                  dwCreationDisposition,
       DWORD                  dwFlagsAndAttributes,
       HANDLE                 hTemplateFile,
       /* This is an argument that gets passed to debugger (if any)*/
       HANDLE*                             hReplacementPointer,
       DWORD*                       lpLastError)
{
       ULONG_PTR ExceptionArgs[EXCEPTION_MAXIMUM_PARAMETERS];
       ZeroMemory(&ExceptionArgs, sizeof(ExceptionArgs));
       int DebugDidNotSee = 0;

       ExceptionArgs[EXCEPTION_ARG_TYPE] = ARG_TYPE_CREATEFILE_NORMAL;
       ExceptionArgs[EXCEPTION_LAST_ERROR] = (ULONG_PTR)lpLastError;

       ExceptionArgs[CF_AW_FILENAME] = (ULONG_PTR)lpFileName;

       if (lpFileName != nullptr)
       {

              ExceptionArgs[CF_AW_FILENAME_CHARCOUNT] = wcslen(lpFileName);
       }
       ExceptionArgs[CF_AW_DESIREDACCESS] = dwDesiredAccess;
       ExceptionArgs[CT_AW_SHAREMODE] = dwShareMode;
       ExceptionArgs[CT_AW_SECURITYATTRIB] = (ULONG_PTR)lpSecurityAttributes;
       ExceptionArgs[CT_AW_CREATIONDISPOSITION] = dwCreationDisposition;
       ExceptionArgs[CT_AW_FLAGANDATTRIBUTES] = dwFlagsAndAttributes;
       ExceptionArgs[CT_AW_TEMPLATE_FILE] = (ULONG_PTR)hTemplateFile;
       ExceptionArgs[CT_AW_OVEERRIDE_HANDLE] = (ULONG_PTR)hReplacementPointer;


       __try
       {
              RaiseException(EXCEPTION_VALUE, 0, 15, &ExceptionArgs[0]);
       }
       __except (GetExceptionCode() == EXCEPTION_VALUE)
       {
              DebugDidNotSee = 1;
       }
       return 0;

}
```

```
HANDLE WINAPI DetouredCreateFileW(
	LPCWSTR               lpFileName,
	DWORD                 dwDesiredAccess,
	DWORD                 dwShareMode,
	LPSECURITY_ATTRIBUTES lpSecurityAttributes,
	DWORD                 dwCreationDisposition,
	DWORD                 dwFlagsAndAttributes,
	HANDLE                hTemplateFile
)
{
	HANDLE hReplacement = 0;
	BOOL Overwritten = FALSE;
	DWORD lastErrorRep = 0;
#ifdef _DEBUG
	//OutputDebugString(L"CreatedCreateFIleW for \"");
	//OutputDebugString(lpFileName);
	//OutputDebugString(L"\"\r\n");
#endif
	DWORD common_branch = __CreateFileAW_CommmonAlert(lpFileName,
dwDesiredAccess, dwShareMode, lpSecurityAttributes, dwCreationDisposition,
dwFlagsAndAttributes, hTemplateFile, &hReplacement,&lastErrorRep);


	if (hReplacement != 0)
	{
		Overwritten = TRUE;
	}

	if (Overwritten)
	{
		SetLastError(lastErrorRep);
		return hReplacement;

	}
	else
	{

		return OriginalCreateFileW(lpFileName, dwDesiredAccess, dwShareMode,
lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile);
	}

}
```

Now we’re successfully got our code written for the Detour DLL for routine of interest, CreateFileW. Let’s walk through how it will work. The debugged app makes a call to access a file via CreateFileW. When this happens, our routine DetouredCreateFileW is called instead in the example. DetouredCreateFileW then starts off by calling a common routine, __CreateFileAW_CommmonAlert. The resource for this separation is that CreateFileA also would eventually invoke CreateFileW. It’s a way to not have to duplicate the alert code between them. After that call, the CreateFileW routine checks if the replacement handle value has been changed from 0. If it has, it sets the last error for the calling thread to the provided last error value and returns the replacement HANDLE without calling CreateFileW.

The CommonAlert function packs the arguments for CreateFileW and writable pointers for the replacement last error code/HANDLE value into an array suitable for RaiseException(). It then calls RaiseException() while attempting to catch the same exception via Structured Exception Handling. If the debugger does not handle the exception – i.e. place a call to ContinueDebugEvent with the value of **DBG_EXCEPTION_NOT_HANDLED,** then this code assumes the debugger did not understand said exception and falls back to returning the value that causes the replacement CreateFileW to just call the original. If the Debugger does set the exception has handled – i.e. a call to ContinueDebugEvent **DBG_CONTINUE,** that is when skipping the CreateFileW and using the replacement HANDLE and Last Error Code come into play.

Combining the Tools: C# .NET managed runtime

We're still not done. For a C# based debugger, we also need to include code to handle both the unmanaged DEBUG_EVENT struct and extract the exception info. It's all cool to get nofiiting when but CreateFileW routine is called; however, what use is it if we can't unpack the example and look at the arguments for the CreateFileW routine. For my implementation in debug event, I went with Exporting C routines that return the various data points from an unmanaged DLL and creating a class in C# to import them. That's going to be explained a little bit later. For now, let's consider the code below to deal with the exception raised via `__CreateFileAW_CommmonAlert`.

```
public static IoDeviceTelemetyCreateFile GetCreateFileSettings(this 
DebugEventExceptionInfo that)
        {
            MachineType type;
            IoDeviceTelemetyCreateFile ret;
            var Arguments = that.ExceptionParameter64;
            IntPtr Handle = 
HelperRoutines.OpenProcessForVirtualMemory(that.ProcessID);

            try
            {
                if (that.IsEventFrom32BitProcess)
                {
                    type = MachineType.MachineI386;
                }
                else
                {
                    type = MachineType.MachineAmd64;
                }
                ret = new IoDeviceTelemetyCreateFile(that.ProcessID, 
that.ThreadID, (IntPtr)Arguments[CreateFile_OvrridePtr], 
(IntPtr)Arguments[GeneralTelemetry.LastError_Ptr], type)
                {
                    FileName = RemoteStructure.RemoteReadString(Handle, new 
IntPtr((long)Arguments[CreateFile_FilenamePtr]), 
Arguments[CreateFile_FileNameCharLen]),
                    DesiredAccess = 
(AccessMasks)Arguments[CreateFile_DesiredAccess],
                    SharedMode = (FileShare)Arguments[CreateFile_ShareMode],
                    SecurityAttrib = new 
IntPtr((long)Arguments[CreateFile_SecurityPtr]),
                    CreateDisposition = 
(CreationDisposition)Arguments[CreateFile_CreationDisposition],
                    FlagsAndAttributes =
(uint)Arguments[CreateFile_FlagsAndAttribs],
                    TemplateFile = new
IntPtr((long)Arguments[CreateFile_TemplateFile])
                };


            }
            finally
            {
                HelperRoutines.CloseHandle(Handle);
            }
            return ret;
              }
```

What this routine does is check a flag on the C# side that makes an educated guess if the event is from Windows x64 or Windows x86 bit. It then uses that to make an instance of a managed class called IoDeviceTelemetryClass that was designed to act as the way for C# code to receive information from the unmanaged part of the DEBUG_EVENT data struct for this exception, extract the arguments for use in C# and provide ways to let the C# programmer be able to interpret said arguments. That includes getting any pointer based information such as the FileName read successfully from the debugger process and into unmanaged land for example. You may notice that the arguments that are passed into creating this class to handle the event are nearly the same arguments that __CreateFileAW_CommmonAlert from earlier packs before calling RaiseException(). You're correct. Windows has passed them from the debugged process by value into the unmanaged part of *our* C# debugger process. This code here is extracting the info and putting it into a form C# and .NET are happy with before cleaning.

Back to how to force a specific handle and last error value. I'll include code in c# that shows what we do.

```
public void SetForceHandle(ulong HandleValue)
        {
            IntPtr handle =
HelperRoutines.OpenProcessForHandleDuplicating(ProcessId);
            try
            {
                if (HandleValue != ulong.MaxValue)
                {
                    IntPtr duphandle =
NativeImports.NativeMethods.DuplicateHandleIntoTarget(new
IntPtr((long)HandleValue), 0, true, handle, true);
                    MemoryNative.RemotePoke8(handle, (ulong)duphandle.ToInt64(),
ForceHandlePtr);
                }
                else
                {
                   MemoryNative.RemotePoke8(handle, ulong.MaxValue,
ForceHandlePtr);
                }
            }
            finally
            {
                HelperRoutines.CloseHandle(handle);
            }
        }
```

```
public void SetLastErrorValue(uint NewValue)
        {
            IntPtr handle =
NativeImports.NativeMethods.OpenProcessForMemoryAccess(ProcessId);
            try
            {
                MemoryNative.RemotePoke4(handle, NewValue, LastErrorPtr);
            }
            finally
            {
                HelperRoutines.CloseHandle(handle);
            }
        }
```

These two routines exist to write a DWORD sized and HANDLE sized value to a remote process. Drawing from the CreateFileW Detour example, those values written back to our process are going to become what HANDLE and Last Error Code are set by our replacement detour routine example with CreateFIleW. There's a few new routines and routines in C# that eventually resolved the C/C++ RemotePoke routine from earlier in this essay. The table below shows an idea of what they do.

`OpenProcessForHandleDuplicating`	This is a helper routine exported by a custom C/C++ part of the project. It opens the passed process with access requests suitable for duplicating a HANDLE from the caller info the target.
`DuplicateHandleIntoTarget`	This routine takes a HANDLE to a process to duplicate too, whether or not if new HANDLE will get the same access. It returns the value of the HANDLE that was duplicated info the remote target process.
`RemotePoke4`	Write a 4 byte value into the specific memory location in the process.
`CloseHandle`	A C# import of Kernel32's CloseHandle to clean up the Process HANDLE.
`SetDebugEventCallbackResponse`	This routine eventually resolves to ContinueDebugEvent. Here we set the state to Tell Windows the exception was HANDLED.

Now to finally tie this whole system together. We write the desired Win32 routine CreateFileW and setup the action in how it will notify the C# based debugger. Next, we tell our debug code to use Detours to spawn or debug target with an arbitrary DLL that will perform the detour for CreateFileW. It's important to note that *all* calls to CreateFileW in the process will eventually get send to the C# debugger that's going to deal with it via exceptions. It's best to play defensive unless you have a backup plan.   I've include code below to elaborate and show an example of customizing on the C# size while leaving the C/C++ handling alone. This is an if statement planted in a callback in C# that our C/C++ handler calls for each debug event. Specifically, it's located in the call's exception handling code. This code is also used for testing while running it as the debugger for an instance of Notepad in Windows.

```
if (Info.DesiredAccess.HasFlag(AccessMasks.GenericRead))
                                        {
                                            if
(!Info.FileName.ToLower().EndsWith(".txt"))
                                            {
                                                if
(Info.FileName.ToLower().EndsWith(".log"))
                                                {

Info.SetForceHandle(Telemetry.InvalidHandleValue64); // INVALID_HANDLE_VALUE
                                                    Info.SetLastErrorValue(5);
// 5 = ERROR_ACCESS_DENIED
                                                    // this should block opening
log files.
                                                }
                                            }
                                            else
                                            {
                                                // notepad will open this
instead.
                                                string ReplacementFile =
@"C:\Dummy\InsightApiDemo\ReDirectTextFileToThis.txt";
                                                using (var HandleToInsert =
File.Open(ReplacementFile, FileMode.Open, FileAccess.ReadWrite))
                                                {

Info.SetForceHandle(HandleToInsert.SafeFileHandle.DangerousGetHandle());
                                                    Info.SetLastErrorValue(0);
                                                }
                                            }
                                          }
      InsightProcess.SetDebugEventCallbackResponse(ContStat,
DebugContState.DebugContinueState);
```

Now what this code does is grab the exception info via GetCreateFileSettings(). From there, it can look at what it wants. In this case, the code is looking to care about attempts to open something via CreateFileW for read access.   This code snippet tests if the file ends in either a .LOG or .TXT extension. If it's a .LOG file, the snippet calls the routine to set the returned handle to the INVALID_HANDLE_VALUE for 64 bit code and routine routine to set returned Last Error Code to 5 which means access denied with Windows error codes. Once doing this, the code snippet is done.   When the exception it examined is marked as being handled returns control to Windows and by extension Notepad, Notepad is going to look at the results and think that access has been denied to the file.

The bottom aspect though is a little different. The code includes a hard coded path and will trigger when opening a file with the .TXT extension. When it happens, the debugger is essentially opening a file of its choice and making the call to ForceHandle and ForceLastErrorValue to tell Notepad to use this HANDLE. From Notepad's perspective the call worked, and it got the file it wanted. The accessed file notepad delas with is returned different from the requested file's info.

Finally, let's consider some additional applications of this mechanism with a hypothetical example. I have a piece of unknown Malware that I'm attempting to see what it does in a security setting. Specifically, I'm wanting to see what it touches in the registry and file system, I could for example use a Detour based DLL with the routines the malware in. I could prove a DLL that detours the routines said Malware links too and follow the method set out above in CreateFileW example. The Detour DLL packs the arguments and some ways to control what's returned into an array for RaiseException() and then call it. When our C# based debugger gets the exception, it can have code added to unpack the exception, process the arguments, log the call and decide whether or not to let the call work before returning control to the malware we're analyzing. The user of this mechanism would also have to consider ways that may block the communication such as calling routines to stop Windows from sending debugging events or bypassing RaiseException exception completely.

References

unsafe keyword - C# Reference | Microsoft Learn

WriteProcessMemory function (memoryapi.h) - Win32 apps | Microsoft Learn

ReadProcessMemory function (memoryapi.h) - Win32 apps | Microsoft Learn

RaiseException function (errhandlingapi.h) - Win32 apps | Microsoft Learn

WaitForDebugEvent function (debugapi.h) - Win32 apps | Microsoft Learn

ContinueDebugEvent function (debugapi.h) - Win32 apps | Microsoft Learn

DEBUG_EVENT (minwinbase.h) - Win32 apps | Microsoft Learn

microsoft/Detours: Detours is a software package for monitoring and instrumenting API calls on Windows. It is distributed in source code form. (github.com)