# Detecting Manual Syscalls from User Mode

winternl.com/detecting-manual-syscalls-from-user-mode

February 10, 2021

By now direct system calls are ubiquitous in offensive tooling. Manual system calls remain effective for evading userland based EDRs. From within userland, there has been little answer to this powerful technique. Such syscalls can be effectively mitigated from kernel mode, but for many reasons, most EDRs will continue to operate exclusively from usermode. This post will present a novel method for detecting manual syscalls from usermode.

## Previous Work

In 2015, Alex Ionescu presented a talk at RECON entitled, *Hooking Nirvana: Stealthy Instrumentation Hooks*, where, among other techniques, he described an instrumentation callback engine which is used internally by Microsoft. You can watch his talk here and read his presentation slides here.

The techniques discussed here have already been weaponized for offensive code injection, but as far as I can tell, have not been applied defensively.

My research is also based off of previous work done by @qaz_qaz and his PoC here. This article also served as a primary point of reference.

And finally, a user by the name of *esoterik* on the game-hacking forum unknowncheats provided an example of a thread safe implementation of the instrumentation hook. Full thread.

## Hooking Nirvana Revisted

There exists an internal instrumentation engine, known as Nirvana, used by Microsoft which has been present since Windows Vista.

> *Nirvana is a lightweight, dynamic translation framework that can be used to monitor and control the (user mode) execution of a running process without needing to recompile or rebuild any code in that process. This is sometimes also referred to as program shepherding, sandboxing, emulation, or virtualization. Dynamic translation is a powerful complement to existing static analysis and instrumentation techniques.*
>
> *– Microsoft*

To understand how this technique will ultimately work, it is necessary to first understand kernel to user mode callbacks. *Ntdll* maintains a set of exported functions which are used by the kernel to invoke specific functionality in usermode. There are a number of these callbacks

which are well documented. These functions are called when the kernel transitions back to user mode. The location (i.e. exported function) will vary based upon intended functionality.

- *LdrInitializeThunk* – Thread and initial process thread creation starting point.
- *KiUserExceptionDispatcher* – Kernel exception dispatcher will IRET here on 1 of 2 conditions.
    1. the process has no debug port.
    2. the process has a debug port, but the debugger chose not to handle the exception.
- *KiRaiseUserExceptionDispatcher* – Control flow will land here in certain instances during a system service when instead of returning a bad status code, it can simply invoke the user exception chain. For instance: CloseHandle() with an invalid handle value.
- *KiUserCallbackDispatcher* – Control flow will land here for Win32K window and thread message based operations. It then calls into function table contained in the process PEB
- *KiUserApcDispatcher* – This is where user queued apc's are dispatched.

The above list was taken from this article. There are many such callbacks, and if you'd like to explore more you can visit Nynaeve's blog.

Each time the kernel encounters a scenario in which it returns to user mode code, it will check if the *KPROCESS!InstrumentationCallback* member is not *NULL*. If it is not *NULL* and it points to valid memory, the kernel will swap out the *RIP* on the trap frame and replace it with the value stored in the *InstrumentationCallback* field.

```
0: kd> dt _kprocess
nt!_KPROCESS
// ...
+0x3d8 InstrumentationCallback : Ptr64 Void
```

view raw kprocess.cpp hosted with by GitHub
But remember, this is the KPROCESS structure, which resides in kernel memory. Official documentation on the *InstrumentationCallback* field is sparse to non, but serendipitously, Microsoft may have inadvertently leaked a clue we can utilize in their SDK. Referencing a specific version of the Windows 7 SDK, there exists a *PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION* structure.

```
typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION
{
ULONG Version;
```

```
    ULONG Reserved;
    PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION,
*PPROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;
```

view raw typedef.cpp hosted with by GitHub

The *KPROCESS!InstrumentationCallback* field can be set from usermode by calling *NtSetInformationProcess* with an undocumented *PROCESSINFOCLASS* value and a pointer to a *PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION* structure.

It is worth noting that process instrumentation behavior and capabilities change between most Windows versions, and certain functionality only exists in later Windows versions. For this post, all research and development is done on a 64-bit Windows 10 machine.

## Nirvana — Now What?

To recap, there exists internal functionality on Windows machines to instrument (read: hook) all kernel to usermode callbacks. In order to detect evasive syscall behavior, there must be a defensive thesis on what makes a syscall malicious. Ideally, a defensive actor would like to allow all syscalls which originate from a legitimate source and block execution when syscalls originate from a malicious source. Manual syscalls may function exactly as legitimate ones but often originate well outside of where they "should be". And the as saying goes, what goes up must come down. Well, for this context, what transitions to the kernel, must transition back to usermode. And this is exactly the defensive thesis used.

*All syscalls which do not transition from the kernel back to usermode at a known valid location, are in fact crafted for evasive purposes*.

The plan now becomes clear. Find out if the syscall returns back to usermode at a known location. This address could be an exported function in *ntdll.dll* or *win32u.dll* (I'm sure there are more callbacks). It may not be a memory page in the *.text* section an unknown module.

## Plan of Defense

Because Nirvana's instrumentation engine hooks transitions *from* the kernel, we are tasked with determining *where* the transition originated from. An auxiliary task, which increases instrumentation robustness, is determining whether the transition was in fact a syscall or another type of transition, such as an APC which would return to *ntdll!KiUserApcDispatcher*. Still, these addresses should *always* return to a known module.

After a syscall is issued, R10 will contain the address of the first instruction to be executed back in userland. This is almost always a return instruction. Validating the integrity of this address can detect the presence of manual syscall invocations.

## Instrumenting from User Mode

Setting the *KPROCESS!InstrumentationCallback* field is easy. It can be done in about 20 lines of code and only a single function call.

```
#define PROCESS_INFO_CLASS_INSTRUMENTATION 40

typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION
{
ULONG Version;
ULONG Reserved;
PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION, *
PPROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;

PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION nirvana;
nirvana.Callback = (PVOID)(ULONG_PTR)InstrumentationCallbackThunk;
nirvana.Reserved = 0; /* Always 0 */
nirvana.Version = 0; /* x64 -> 0 | x86 -> 1 */

NtSetInformationProcess(
GetCurrentProcess(),
(PROCESS_INFORMATION_CLASS)PROCESS_INFO_CLASS_INSTRUMENTATION,
&nirvana,
sizeof(nirvana));
```

view raw Instrumentation.cpp hosted with by GitHub

Now that we have the *InstrumentationCallback* field updated, we must implement the hook. The hook has to be cognizant of all non-volatile registers, proper stack alignment, unintended recursion, and thread safety. The hook is implemented in two separate files, in part because the 64-bit MSVC compiler does not support inline assembly. The first part of the instrumentation hook is coded in assembly. This procedure will be pointed to by the *KPROCESS!InstrumentationCallback* field. It is responsible for preserving registers (which

cannot easily be accomplished without inline assembly) and subsequently calling the next part of the hooking routine. The second function is written in C/C++ and will contain the logic needed to verify the integrity of the syscall.

Prior to Windows 10, the instrumentation functionality used by this project was only available for 64-bit Windows versions. To support x86 and WoW64, four new fields were added to the *TEB* structure.

```
_TEB_64
+0x02D0 ULONG_PTR InstrumentationCallbackSp
+0x02D8 ULONG_PTR InstrumentationCallbackPreviousPc
+0x02E0 ULONG_PTR InstrumentationCallbackPreviousSp
+0x02EC BOOLEAN InstrumentationCallbackDisabled
```

view raw teb.cpp hosted with by GitHub

In x64 Windows, I believe, but am not certain, these fields are unused when implementing instrumentation callbacks. However, because they present a thread safe location to store information regarding the callback, the hook can use these addresses for reading and writing information. The following code is originally from *esoterik*, found under the previous research section.

```
InstrumentationCallbackThunk proc
mov gs:[2e0h], rsp ; _TEB_64 InstrumentationCallbackPreviousSp
mov gs:[2d8h], r10 ; _TEB_64 InstrumentationCallbackPreviousPc
mov r10, rcx ; Save original RCX
sub rsp, 4d0h ; Alloc stack space for CONTEXT structure
and rsp, -10h ; RSP must be 16 byte aligned before calls
mov rcx, rsp
call __imp_RtlCaptureContext ; Save the current register state. RtlCaptureContext does not require shadow space
sub rsp, 20h ; Shadow space
call InstrumentationCallback
InstrumentationCallbackThunk endp
```

view raw thunk64.asm hosted with by GitHub

Because Rtl* functions are implemented entirely in usermode, there is no need to worry about recursion here.

The second, and main part of the instrumentation routine is responsible for analyzing the execution context at the point of kernel to usermode return. The routine is only a PoC and performs a very cursory bounds check to determine whether *RIP* is pointing to a memory location within *ntdll.dll* or *win32u.dll*. If not, the program will warn of a potential manual syscall and break execution.

Here's my version of the instrumentation hook which implements the minimal required code for a PoC. Optionally it performs a reverse lookup if the executable is built with debug information.

```
#define RIP_SANITY_CHECK(Rip,BaseAddress,ModuleSize) (Rip > BaseAddress) &&
(Rip < (BaseAddress + ModuleSize))

VOID InstrumentationCallback(PCONTEXT ctx)
{
BOOLEAN bInstrumentationCallbackDisabled;
ULONG_PTR NtdllBase;
ULONG_PTR W32UBase;
DWORD NtdllSize;
DWORD W32USize;

#if _DEBUG
BOOLEAN SymbolLookupResult;
DWORD64 Displacement;
PSYMBOL_INFO SymbolInfo;
PCHAR SymbolBuffer[sizeof(SYMBOL_INFO) + 1024];
#endif

ULONG_PTR pTEB = (ULONG_PTR)NtCurrentTeb();

//
```

```
// https://www.geoffchappell.com/studies/windows/win32/ntdll/structs/teb/index.htm
//
ctx->Rip = *((ULONG_PTR*)(pTEB + 0x02D8)); // TEB->InstrumentationCallbackPreviousPc
ctx->Rsp = *((ULONG_PTR*)(pTEB + 0x02E0)); // TEB->InstrumentationCallbackPreviousSp
ctx->Rcx = ctx->R10;

//
// Prevent recursion. TEB->InstrumentationCallbackDisabled
//
bInstrumentationCallbackDisabled = *((BOOLEAN*)pTEB + 0x1b8);

if (!bInstrumentationCallbackDisabled) {

//
// Disabling for no recursion
//
*((BOOLEAN*)pTEB + 0x1b8) = TRUE;

#if _DEBUG
SymbolInfo = (PSYMBOL_INFO)SymbolBuffer;
RtlSecureZeroMemory(SymbolInfo, sizeof(SYMBOL_INFO) + 1024);

SymbolInfo->SizeOfStruct = sizeof(SYMBOL_INFO);
SymbolInfo->MaxNameLen = 1024;

SymbolLookupResult = SymFromAddr(
GetCurrentProcess(),
ctx->Rip,
&Displacement,
```

```
    SymbolInfo
    );
    #endif

    #if _DEBUG
    if (SymbolLookupResult) {
    #endif
    NtdllBase = (ULONG_PTR)InterlockedCompareExchangePointer(
    (PVOID*)&g_NtdllBase,
    NULL,
    NULL
    );

    W32UBase = (ULONG_PTR)InterlockedCompareExchangePointer(
    (PVOID*)&g_W32UBase,
    NULL,
    NULL
    );

    NtdllSize = InterlockedCompareExchange(
    (DWORD*)&g_NtdllSize,
    NULL,
    NULL
    );

    W32USize = InterlockedCompareExchange(
    (DWORD*)&g_W32USize,
    NULL,
    NULL
```

```
);

if (RIP_SANITY_CHECK(ctx->Rip, NtdllBase, NtdllSize)) {

if (NtdllBase) {

#if _DEBUG
//
// See if we can look up by name
//
PVOID pFunction = GetProcAddress((HMODULE)NtdllBase, SymbolInfo->Name);

if (!pFunction) {
printf("[-] Reverse lookup failed for function: %s.\n", SymbolInfo->Name);
}
else {
printf("[+] Reverse lookup successful for function %s.\n", SymbolInfo->Name);
}
#endif
}
else {
printf("[-] ntdll.dll not found.\n");
}
}
else if (RIP_SANITY_CHECK(ctx->Rip, W32UBase, W32USize)) {

if (W32UBase) {

#if _DEBUG
//
```

```
// See if we can look up by name
//
PVOID pFunction = GetProcAddress((HMODULE)W32UBase, SymbolInfo->Name);

if (!pFunction) {
printf("[-] Reverse lookup failed for function: %s.\n", SymbolInfo->Name);
}
else {
printf("[+] Reverse lookup successful for function %s.\n", SymbolInfo->Name);
}
#endif
}
else {
printf("[-] win32u.dll not found.\n");
}
}
else {

printf("[SYSCALL-DETECT] Kernel returns to unverified module, preventing further execution!\n");
#if _DEBUG
printf("[SYSCALL-DETECT] Function: %s\n", SymbolInfo->Name);
#endif
DebugBreak();
}

#if _DEBUG
}
else {
```

```
    //
    // SymFromAddr failed
    //
    printf("SymFromAddr failed.\n");
    // DebugBreak();
    }
    #endif
    //
    // Enabling so we can catch next callback.
    //
    * ((BOOLEAN*)pTEB + 0x1b8) = FALSE;
    }

    RtlRestoreContext(ctx, NULL);
    }
```

Ideally, there should be *much more* verification done to ensure the integrity of the syscall. Ultimately this will be left as an exercise to the reader. Here are some of my own ideas (I'd love to hear yours):

- If running an instrumentation routine on an executable with a *pdb* symbol file store, one can use the set of symbol handler functions located within *dbghelp.dll* to perform reverse lookups. The symbol handler functions can resolve *RIP* to a function name using the function ***SymFromAddr***. If the function does not resolve, the syscall was most likely issued in an evasive way.
- An immediate bypass to this technique which comes to mind is to simply overwrite a legitimate, but seldom used exported function in *ntdll.dll*. One could simply overwrite the syscall number with your desired index and call the function as normal. A resolution to this bypass might be to implement an anti-tamper routine on *ntdl.dll's* address space. Perhaps hash and cross-reference each of it's Nt* routines.

- Reverse disassembly seems feasible in providing further analysis of the origin of the syscall. Syscalls will (always?) be followed by a *ret* instruction, which is the location pointed to by *RIP* upon transition back to usermode. One can assume the previous instruction will be a syscall (x64 Windows 10). Following the syscall stub structure present in x64 Windows 10, the instruction preceding the syscall would move the syscall service index into eax. I wonder if it's possible to retrieve the syscall index from the information available when the kernel returns to usermode? It would be a very powerful defensive technique to reverse disassemble *RIP* until the *Nt** procedure base is identified (*mov r10, rcx*). Then cross-referencing the syscall index found via reverse disassembly to the corresponding syscall index and address pair found by performing a sort on the set of {*Zw* U Nt**} function addresses (as described by odzhan). If the base addresses and syscall indeces do not match, then the syscall was likely manual.

## Final Remarks

Of course, this is just another tool in the proverbial toolkit, and does not represent a significant change in the dynamic of the userland threat landscape. I do however, think this a powerful technique that has been overlooked by the blue team. Most userland unhookers do not account for this instrumentation callback. Conversely, I see lots of potential for misuse and offensive tooling — as I hope you do too.

Full PoC available on my GitHub.

Notepad functionality is allowed through the instrumentation callback. Functions are being resolved correctly via *SymFromAddr*. There is a noticeable performance impact due to console logging. Additionally, notepad will crash when the dll is injected before full process initialization. The hook needs a lot more work!