Investigating Blocked Device Exposure Paths in Legacy Windows Drivers Using Controlled RIP Redirection

Introduction

When hunting for vulnerable Windows drivers, one of the first practical questions is simple: does the driver expose a reachable interface that user mode can talk to?

At a static level, this can look straightforward. If a driver references APIs such as IoCreateDevice and IoCreateSymbolicLink, it may appear to have the ingredients needed to expose a device object and a symbolic link for user-mode access. In practice, however, those references are only weak signals. Device exposure can depend on registry configuration, hardware presence, helper drivers, initialization order, or vendor-specific runtime checks. A driver can contain the relevant code and still never expose a usable interface in a normal lab environment.

This became a bottleneck during my driver-hunting pipeline. Across a corpus of 8,650 unique driver samples, only a very small number produced unique reachable device interfaces during dynamic testing. That raised a useful research question: if device-creation logic exists inside a driver but is not reached naturally, can controlled kernel-mode execution redirection be used to reach those hidden paths?

This post explores that question through a case study on RUSB3XHC.sys. Using a controlled lab primitive for RIP redirection, I attempted to redirect execution toward suspected device-creation code and then validate whether a new user-accessible interface appeared. The result was negative, but useful: simply redirecting RIP into a suspected function was not enough. The driver still expected specific calling context, register state, initialized structures, and dispatch-path behavior.

The main takeaway is that forced device exposure is not a generic “jump to IoCreateDevice” problem. It is a driver-specific reachability and state-reconstruction problem. This post walks through the corpus observation, the hypothesis, the failed attempts, and the lessons learned for future driver-hunting automation.

Corpus Collection & Results

Before trying to force any hidden device-creation paths manually, we first wanted to understand how often drivers in our collection actually exposed reachable user-mode interfaces under normal testing.

We tested the drivers in bulk using our automated driver collection pipeline and a Python-based device enumeration script. In total, we collected 8,650 unique driver samples, deduplicated by SHA-256 hash.

From there, we looked for drivers that referenced the usual device-creation APIs, mainly IoCreateDevice and IoCreateSymbolicLink, and then compared those static signals against what actually became reachable after attempting to load the drivers in our lab environment.

The results looked like this:

Stage	Count
1. Unique driver samples, deduplicated by SHA-256	8,650
2. Drivers containing both IoCreateDevice and IoCreateSymbolicLink	1,226
3. Signed drivers containing at least one of the target APIs	1,536
4. Drivers that were loadable in the lab	509
5. Drivers that exposed any device interface	63
6. Devices likely exposed because we accidentally met test/prerequisite conditions	36
7. Unique device interfaces useful for testing	10

This shows the core problem clearly: out of 8,650 unique driver samples, only 10 unique device interfaces were useful for testing. That gives us a final yield of roughly 0.115%.

The low yield suggests that static references to IoCreateDevice and IoCreateSymbolicLink are not enough by themselves. A driver can contain the right APIs, be signed, and even be loadable, while still failing to expose a usable device interface at runtime.

Our assumption is that many of these drivers are gated behind some kind of prerequisite check. This could be a registry read (such as the ones made during installation with a .inf file), hardware presence check, dependency on another driver, initialization order requirement, or some other vendor-specific condition that prevents the device-creation path from being reached in a normal lab setup.

That gap between “the device-creation code exists” and “the device is actually reachable from user mode” is what motivated the next part of the research: can we use controlled kernel-mode RIP redirection to reach device-creation paths that are present in the driver, but not naturally reached during normal loading?

```
Summary:
Files found: 11092
Unique files copied: 8650
Files skipped (duplicates/errors): 2442
Output folder: C:\Users\kali\Desktop\Drvtest\sys_files_collection
Total unique drivers collected: 8650
```

Figure 1 — Driver collection summary.

Problem Statement

The corpus results show the core problem: even when a driver is signed, loadable, and contains references to device-creation APIs, it still may not expose a reachable user-mode device interface.

For driver vulnerability research, this creates a major bottleneck. If the device interface is never exposed, then the IOCTL surface cannot be reached through normal user-mode testing. This means that a large number of drivers may appear interesting during static analysis, but still produce no usable testing surface during dynamic analysis.

The important gap is between “the device-creation code exists” and “the device is actually reachable from user mode.” The rest of this post explores whether that gap can be tested manually by redirecting execution toward suspected device-creation paths.

Research Hypothesis

The hypothesis is simple: if a driver contains a device-creation path that is present in the binary but not naturally reached during normal loading, controlled kernel-mode RIP redirection may allow us to reach that path manually.

The goal is not to claim that this technique works generically across every driver. The goal is to test whether RIP redirection can be used as a research primitive for device-exposure testing. If we can redirect execution toward a suspected IoCreateDevice / IoCreateSymbolicLink path and satisfy the expected runtime state, we may be able to expose device interfaces that would otherwise remain unreachable through normal user-mode testing.

The more important question is where this approach breaks. Is the problem simply that the target code path is not being reached, or does the driver require specific register values, initialized structures, hardware state, registry values, or helper-driver interaction before the device-creation path can execute successfully?

Lab Setup and Constraints

For this experiment, we had two possible primitives available for controlling RIP in our lab environment.

The first was an MSR read/write primitive which, with HVCI disabled, allowed us to redirect execution through the IA32_LSTAR pointer. The second was a separate control-flow hijack primitive caused by a signed AMD driver passing a user-controlled IOCTL buffer into a function pointer call. For this writeup, we focus only on the MSR read/write path.

The idea was to use this primitive to redirect RIP toward a suspected device-creation path inside the target driver and then check whether a new device interface appeared through WinObj or our device enumeration script.

There are important limitations here. With the current primitive, user-controlled ROP chaining is not practical. SMEP prevents supervisor-mode execution from user pages, and in this setup we do not have clean control over a trusted kernel stack or a reliable stack pivot target. Because of that, this experiment focuses on direct RIP redirection into individual target locations rather than a full kernel ROP chain.

This limitation matters because reaching the target address is only one part of the problem. The target function may still expect a valid calling context. This could include specific register values, initialized structures, object pointers, or previous driver state. If those conditions are not satisfied, then the result is likely to be a crash rather than successful device exposure.

Experimental Plan

The manual approach for testing this was straightforward:

1. List the drivers in the signed, loadable, device-API overlap set.
2. Use Ghidra to find functions that reference `IoCreateDevice` and, ideally, `IoCreateSymbolicLink`.
3. Calculate the relevant RVA for the suspected device-creation path.
4. Use WinDbg to find the loaded base address of the driver and resolve the runtime address.
5. Redirect RIP toward the suspected path inside the lab environment.
6. Use WinObj and our device enumeration script to check whether a new device interface appeared.
7. If the attempt failed, debug whether the issue was reachability, register state, initialized structures, or some other driver-specific precondition.

At this stage, the approach was still theoretical. The main purpose of the first test was to see whether direct RIP redirection was enough to expose a hidden device interface, or whether the driver would require additional state reconstruction before the target path could execute correctly.

Case Study: RUSB3XHC.sys

For the first manual test, we picked `RUSB3XHC.sys` as the target driver. The goal was to start with a simple case study: find the function that contains the `IoCreateDevice` call, resolve the runtime address, redirect RIP there, and then check whether the driver exposes a new device interface.

This section walks through the attempts, the crashes, the breakpoint analysis, and the final reason this specific driver did not expose a usable interface through our approach.

Attempt 1: Direct RIP Redirection

The first attempt was supposed to be simple. We found the function in Ghidra that contained the `IoCreateDevice` call.

We then looked at the offset inside of Ghidra:

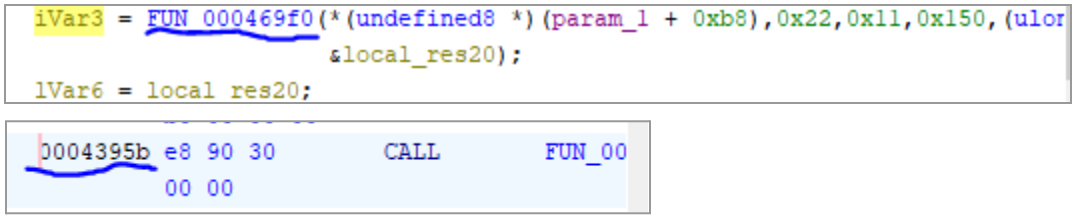

Figure 2 — `IoCreateDevice` reference in Ghidra.

From there, we calculated the relevant RVA for the target location. In this case, the offset came out to `0x3395b`. To get the runtime address, we used WinDbg's `lmDvm` command to get the loaded base address of the driver.

We confirmed that this was the correct address by using the following command and comparing the result with the Ghidra output:

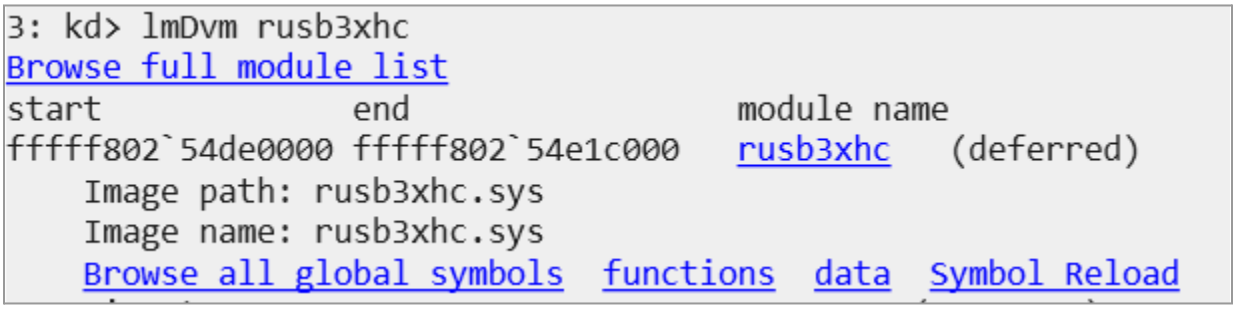

Figure 3 — Runtime address validation in WinDbg.

The output matched. So, for the first test, we set RIP to `fffff802`54e1395b` so execution would land at the suspected target location. We used the following command for that:

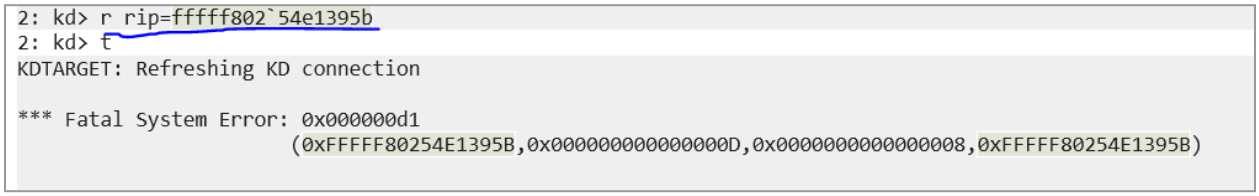

Figure 4 — RIP redirected to the suspected path.

As expected, this crashed the VM:

```
Fatal System Error: 0x000000d1 (0xFFFFF80254E1395B,
0x000000000000000D, 0x0000000000000008, 0xFFFFF80254E1395B)
```

This was not too surprising. This was the first attempt, and the goal was mainly to see what would happen if we directly redirected execution into the suspected device-creation path. Since the expected register values and calling context were not properly configured, the function did not have the state it needed to execute safely.

The bugcheck was `0xD1`, also known as `DRIVER_IRQL_NOT_LESS_OR_EQUAL`. In this case, the likely issue was not that the target address was wrong, but that directly landing inside the function without reconstructing the expected context caused the driver to access invalid state.

Result 1: Calling Context Matters

The crash gave us the first useful result: reaching the target address is not enough.

If we redirect RIP into a function that expects specific register values, initialized structures, object pointers, or previous driver state, then the function may immediately fail. This means direct RIP redirection is only useful if the target path can tolerate the current context, or if we can reconstruct enough of the expected state before calling it.

So instead of continuing to blindly redirect RIP, the next step was to observe what the driver was naturally doing. Specifically, we wanted to know whether the normal load path ever reached `IoCreateDevice`, and if it did, what values were being passed around at runtime.

Tracing the Natural IoCreateDevice Path

We found the function where the API call was present:

```
int CreatesDevice(undefined8 param_1,undefined8 param_2,undefined1 param_3,ushort param_4,
                 ulonglong param_5,undefined8 *param_6)

{
  ulonglong *puVar1;
  ulonglong *puVar2;
  int Bool_IoCreateDevice;
  undefined4 uVar3;
  ulonglong local_18 [2];

  Bool_IoCreateDevice = IoCreateDevice(param_1,param_4,0,0x22,0x180,0,local_18);
  if (-1 < Bool_IoCreateDevice) {
    puVar2 = *(ulonglong **)(local_18[0] + 0x40);
    FUN_0002a5d4(puVar2,param_3,param_4,0x32c38,param_5);
    uVar3 = (undefined4)(param_5 >> 0x20);
    puVar2[6] = local_18[0];
    *(undefined4 *)(puVar2 + 7) = 0;
    *(undefined4 *)((longlong)puVar2 + 0x3c) = 0;
    KeInitializeEvent(puVar2 + 8,1);
    IoInitializeRemoveLockEx(puVar2 + 0xb,0x56454442,0,0,CONCAT44(uVar3,0x20));
```

Figure 5 — Suspected `IoCreateDevice` callsite.

From Ghidra, we also found the relevant offset to be `0x36a2e`. We then set a breakpoint there and restarted the driver:

```
rusb3xhc+0x36a2e:
fffff803`13326a2e ff15a4f8feff    call    qword ptr [rusb3xhc+0x262d8 (fffff803`133162d8)]
fffff803`13326a34 8bf8            mov     edi,eax
fffff803`13326a36 85c0            test    eax,eax
fffff803`13326a38 0f889f000000    js      rusb3xhc+0x36add (fffff803`13326add)
fffff803`13326a3e 488b4c2440      mov     rcx,qword ptr [rsp+40h]
fffff803`13326a43 488b5940        mov     rbx,qword ptr [rcx+40h]
fffff803`13326a47 488b8c2480000000 mov     rcx,qword ptr [rsp+80h]
fffff803`13326a4f 48894c2420      mov     qword ptr [rsp+20h],rcx
2: kd> bp fffff803`13326a2e
2: kd> g
```

Figure 6 — Breakpoint on the suspected callsite.

The breakpoint was never hit. This told us that the function was not being called during the normal driver load path.

So the next question became: why is this device-creation path not being reached?

Finding the Suspected IRP_MJ_CREATE Path

Looking further, we found a function involved in setting up what looked like the IRP dispatch flow:

```
if (-1 < iVar3) {
  if (((*param_3 == 0x94) && ((char)param_3[1] == '\x03')) &&
     (*(char *)((longlong)param_3 + 3) == '\0')) {
    FUN_00032ea0((undefined8 *)&DAT_0003a1d0,(undefined8 *)param_3,0x94);
    _DAT_0003a1c2 = *param_2 + 2;
    _DAT_0003a1c0 = *param_2;
    DAT_0003a1c8 = ExAllocatePoolWithTag(1,_DAT_0003a1c2,0x6233756e);
    if (DAT_0003a1c8 == 0) {
      iVar3 = -0x3fffff66;
    }
    else {
      RtlCopyUnicodeString(&DAT_0003a1c0,param_2);
      *(code **)(param_1 + 0x148) = FUN_00042f28;
      *(code **)(param_1 + 0x120) = FUN_00044b3c;
      *(code **)(param_1 + 0x128) = FUN_00044eb0;
      *(code **)(param_1 + 0x70) = FUN_00044f38;
      *(code **)(param_1 + 0x80) = FUN_00044f8c;
      *(code **)(param_1 + 0xe0) = FUN_0002c260;
      *(code **)(param_1 + 0xe8) = FUN_0002c260;
      *(code **)(param_1 + 0x68) = FUN_00042c1c;
      *(code **)(*(longlong *)(param_1 + 0x30) + 8) = IRP_CreateDevice;
    }
  }
```

Figure 7 — Dispatch setup routine.

At this point, we wanted to understand what `param_1 + 0x30` was actually pointing to. From WinDbg, we reached the relevant breakpoint and started inspecting the runtime state.

R12 was holding the `param_1` value, so we dumped the registers to see what `param_1` actually was:

```
1: kd> u fffff805`8ace0000+38409h
Unable to load image rusb3xhc.sys, Win32 error 0n2
rusb3xhc+0x38409:
fffff805`8ad18409 498b442430      mov     rax,qword ptr [r12+30h]
fffff805`8ad1840e 488d0d3ba8ffff  lea     rcx,[rusb3xhc+0x32c50 (fffff805`8ad12c50)]
fffff805`8ad18415 48894808        mov     qword ptr [rax+8],rcx
fffff805`8ad18419 eb14            jmp     rusb3xhc+0x3842f (fffff805`8ad1842f)
fffff805`8ad1841b bf9a0000c0      mov     edi,0C000009Ah
fffff805`8ad18420 897c2428        mov     dword ptr [rsp+28h],edi
fffff805`8ad18424 eb09            jmp     rusb3xhc+0x3842f (fffff805`8ad1842f)
fffff805`8ad18426 bf0d0000c0      mov     edi,0C000000Dh
```

```
0: kd> r r12
r12=ffff8c87e756f800
```

Figure 8 — Register dump for `param_1`.

Then we looked at the address at `param_1 + 0x30`:

```
0: kd> dq ffff8c87e756f800+30
ffff8c87`e756f830  ffff8c87`e756f950 00000000`00180018
ffff8c87`e756f840  ffff8c87`e57e97d0 fffff805`67f2e990
ffff8c87`e756f850  00000000`00000000 fffff805`8ad184b8
ffff8c87`e756f860  00000000`00000000 fffff805`8ad12c1c
ffff8c87`e756f870  fffff805`8ad14f38 fffff805`675159c0
ffff8c87`e756f880  fffff805`8ad14f8c fffff805`675159c0
ffff8c87`e756f890  fffff805`675159c0 fffff805`675159c0
ffff8c87`e756f8a0  fffff805`675159c0 fffff805`675159c0
```

Figure 9 — Inspecting `param_1 + 0x30`.

This did not particularly look like a normal dispatch table at first. However, we also knew that `ffff8c87`e756f950` would be stored inside RAX, and its +8 offset would be used next:

```
0: kd> dq ffff8c87`e756f950 L2
ffff8c87`e756f950  ffff8c87`e756f800 00000000`00000000
```

Figure 10 — Pointer assignment target.

From the assembly, we can see that the code stores the pointer to the `IRP_CreateDevice` function into `ffff8c87'e756f958`.

This made the path more interesting. The driver was not simply calling the device-creation function directly during initialization. Instead, it looked like `IRP_CreateDevice` was being placed into a dispatch-related structure, potentially as a create routine.

From here, we inspected the `IRP_CreateDevice` function itself. This looked like it could be an induced device-creation routine. If that was true, then we could try to trigger it through a user-mode open attempt using `CreateFileW`.

The immediate problem was that we did not know the correct device name. So, we went back into Ghidra and searched for hardcoded device-related strings.

```
FUN_000135ec(local_2f0,L"\\Parameters\\%04X%04X%04X",(ulonglong)*(ushort *)(param_2 + 0x20),
    iVar5 = FUN_0002a380(lVar2 + 0x80,L"\\DosDevices\\",*(longlong *)(lVar2 + 0x38),4,
RtlAppendUnicodeToString(&local_58,L"\\DosDevices\\");
uVar1 = FUN_0002a380(local_res10,L"\\DosDevices\\",param_1,0x10000,
FUN_00032ea0(local_88,(undefined8 *)L"\\DosDevices\\HCD",0x20);
for (local_40 = L"\\Parameters"; ((local_38 != 0 && (local_28 != 0)) && (*local_40 != L'\0'));
        L"RENESAS_USB3\\ROOT_HUB30&VID_%04X&PID_%04X&REV_%04X&SID_%04X%04X",0x13e);
            L"RENESAS_USB3\\ROOT_HUB30&VID_%04X&PID_%04X&REV_%04X&SID_%04X%04X",
FUN_00032ea0(puVar8,(undefined8 *)L"RENESAS_USB3\\ROOT_HUB30",0x34);
```

Figure 11 — Device strings in Ghidra.

Here, we saw references to `DosDevices` and `DosDevices\\HCD`. Interestingly enough, we also saw similar-looking entries inside WinObj:

HCD0	SymbolicLink	\Device\USBFDO-0
USB#ROOT_HUB30#4&24054718&0&0#{f18a0e88-c30c-11d...	SymbolicLink	\Device\USBPDO-0
USB#VID_80EE&PID_0021#5&12c8f4c0&0&1#{a5dcbf10-653...	SymbolicLink	\Device\USBPDO-1

Figure 12 & 13 — Similar names in WinObj.

However, these did not appear to be devices created by this driver. More importantly, we saw references to `RENESAS` inside the driver. There were potential device names such as `RENESAS_USB3\\ROOT_HUB30&VID...`, which suggested that the device name may start with something like `Global\\RENESAS_USB3`. We also saw a reference to `\\DosDevices\\HCD`.

Attempt 2: Triggering the Path from User Mode

With these strings in mind, we attempted to trigger the suspected path from user mode.

Before:

Figure 14 — Device namespace before the trigger.

After:

Figure 15 — Device namespace after the trigger.

This was not successful. No new device interface appeared.

At this point, we needed to understand whether the failure was because of the device name, a missing prerequisite, or the wrong execution path entirely.

Checking the Magic-Value Condition

Looking back at the code, we saw a conditional check that guarded the section where the `IRP_CreateDevice` function lived:

```
if (((*param_3 == 0x94) && ((char)param_3[1] == '\x03')) &&
   (*(char *)((longlong)param_3 + 3) == '\0')) {
```

Figure 16 — Conditional check before setup.

From the decompiler output, it looked like the function expected a magic value inside `param_3`. Specifically, it appeared to check whether `param_3` started with the bytes `0x94 0x03 0x00`.

So the next question was: what is `param_3`, and is it actually under our control?

```
iVar1 = Step_routine_IRP_MJ(DeviceExtention,IRP,(short *)local_b8,2_DeviceExtention);
```

```
local_b8._0_2_ = 0x94;
local_54 = &LAB_00011008;
local_b8[2] = 3;
local_4c = FUN_0001la1c;
local_b8[3] = 0;
```

Figure 17 — Inspecting `param_3`.

We can see that `param_3` is locally initialized and does not appear to be directly controlled from user mode. We also see that `local_b8` contains the expected magic bytes hardcoded inside the function. Therefore, the conditional check itself may not actually be the barrier.

To confirm this, we set a breakpoint on the conditional check and observed what happened at runtime.

We used `sxe ld:rusb3xhc` to break when the driver loaded. Then we used `lmDvm rusb3xhc` to find the loaded base address and set a breakpoint at the target location. In this case, the resolved address was `fffff802\76c9830d`.

The breakpoint hit, so we inspected the runtime state.

Inside WinDbg, we successfully hit the following location:

```
rusb3xhc+0x38349:
fffff802`76c98349 e852abfeff      call    rusb3xhc+0x22ea0 (fffff802`76c82ea0)
```

Figure 18 — Breakpoint hit before the check.

Which corresponds to the following location in Ghidra:

```
                        IRP_CreateDevice

00042c50 4c 8b dc        MOV        R11,RSP
00042c53 49 89 5b 08     MOV        qword ptr [R11 + local_res8
00042c57 49 89 73 10     MOV        qword ptr [R11 + local_res1
00042c5b 57              PUSH       RDI
00042c5c 41 54           PUSH       R12
00042c5e 41 55           PUSH       R13
00042c60 48 83 ec 40     SUB        RSP,0x40
00042c64 4c 8b e2        MOV        R12,param_2
00042c67 4c 8b e9        MOV        R13,param_1
00042c6a 33 ff           XOR        EDI,EDI
00042c6c 49 89 7b 18     MOV        qword ptr [R11 + local_res1
```

```
if ((*param_3 == 0x94) && ((char)param_3[1] == '\x03')) &&
   (*(char *)((longlong)param_3 + 3) == '\0')) {
  FUN_00032ea0((undefined8 *)&DAT_0003a1d0,(undefined8 *)param_3,0x94);
```

Figure 19 — Matching Ghidra location.

Therefore, we can be certain that `param_3`, or more specifically the locally initialized `local_b8`, passes this check.

In fact, after setting a breakpoint at the exact line where `IRP_CreateDevice` was being initialized:

```
*(code **)(*(longlong *)(param_1 + 0x30) + 8) = IRP_CreateDevice;
```

We still saw the breakpoint being hit.

So the magic values were not the issue. The more likely issue was our approach to getting the driver to expose the device interface.

Attempt 3: Breaking on IRP_CreateDevice

From the code, it looked like the function responsible for device creation, `IRP_CreateDevice`, was being attached to a dispatch path that appeared related to `IRP_MJ_CREATE`. In a normal user-mode flow, this kind of path would usually be triggered when a correct request is made through `CreateFileW`.

To test that, we set a breakpoint directly on the `IRP_CreateDevice` function and then ran our Python script to try to open the suspected device name.

We calculated the relevant offset as `0x32c50`.

Then we set the breakpoint at the resolved runtime address:

```
2: kd> bp fffff805`98c32c50
2: kd> g
```

Figure 20 — Breakpoint on `IRP_CreateDevice`.

After that, we ran the Python script using `CreateFileW` to try to trigger the `IRP_MJ_CREATE` path:

```
PS C:\Users\kali\Desktop> python3 .\Force_drvr_dev_exposure.txt
[*] Attempting to open candidate device names...

[+] Success: \\.\HCD (handle 18446744073709551615)
[+] Success: \\.\Devices\HCD0 (handle 18446744073709551615)
[+] Success: \\.\Global\RENESAS_USB3 (handle 18446744073709551615)
[+] Success: \\.\Global\RENESAS_USB3\ROOT_HUB30 (handle 18446744073709551615)
[+] Success: \\.\Global\RENESAS_USB3\ROOT_HUB30&VID (handle 18446744073709551615)
[+] Success: \\.\RENESAS_USB3 (handle 18446744073709551615)
PS C:\Users\kali\Desktop>
```

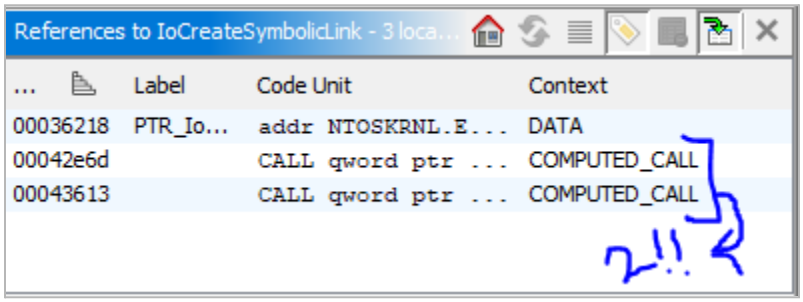

Figure 21 — User-mode trigger attempt.

Although some candidate names returned valid handles, the breakpoint on `IRP_CreateDevice` was not hit. This suggests that these opens were not reaching the target routine we were trying to trigger.

The Device Exposure Deadlock

At this point, we looked again at the symbolic link creation path.

There were only two instances of `IoCreateSymbolicLink` present. One of them was inside the `IRP_CreateDevice` function itself. This creates a chicken-and-egg problem: to reach the device from user mode, we need a symbolic link, but the symbolic link appears to be created inside a path that we cannot reach from user mode unless the device is already exposed.

In other words, this does not look like a simple user-mode `CreateFileW` trigger path.

The more likely explanation is that this routine is triggered by another kernel-mode component, possibly a helper driver or some kernel-mode equivalent create/open path. Since we do have RIP control, we could still attempt to redirect execution into this code manually. However, the first crash already showed the main problem: we would need the proper register values and expected runtime structures for the function to execute safely.

At this stage, we were unable to recover the full calling context required for a successful call into `IRP_CreateDevice`. So, for this driver, direct RIP redirection was not enough to expose the device interface.

Takeaway from this Case Study

The important takeaway from this attempt is that the barrier was not simply the presence of a magic value or the existence of the device-creation function.

The target path existed. The magic-value check passed. The function pointer assignment happened. However, the path still was not reachable through our user-mode trigger attempt, and direct RIP redirection crashed without the correct runtime state.

So for `RUSB3XHC.sys`, the real problem was reachability and state reconstruction. Getting the driver to expose the device interface required more than just landing RIP at the right address. The driver expected a specific calling context, and without that context, the approach failed.

This does not invalidate the broader idea, but it does narrow it. Forced device exposure is not a generic “jump to `IoCreateDevice`” problem. It is a driver-specific problem involving control flow, initialized state, dispatch behavior, and the exact context expected by the target routine.

Lessons Learned

The main lesson from this case study is that RIP redirection can get us to interesting code, but it does not automatically give us a valid execution context.

In the case of `RUSB3XHC.sys`, the device-creation path did exist. We found the relevant `IoCreateDevice` / `IoCreateSymbolicLink` related logic, we identified the suspected `IRP_CreateDevice` routine, and we confirmed that some of the conditional checks were not the actual barrier. However, directly redirecting RIP into the suspected path still crashed the VM because the expected register values and runtime structures were not properly configured.

This means the problem is not just “can we reach the code?” The more important problem is “can we reach the code with the state it expects?”

That distinction matters. A driver may contain the right device-creation code, but that code may depend on a very specific calling path. It may expect initialized structures, specific object pointers, helper-driver interaction, hardware state, registry values, or some other driver-specific setup. Without that context, the function may not behave correctly even if we land RIP at the right address.

So, for this driver, forced device exposure was not a simple “jump to `IoCreateDevice`” problem. It was a reachability and state-reconstruction problem.

Limitations

This was a single-driver case study, not a generic proof that this method works across all drivers.

The test case, `RUSB3XHC.sys`, did not allow us to successfully expose a new user-mode device interface through direct RIP redirection. We were able to identify interesting device-creation logic and understand more about why the path was not naturally reachable, but we were not able to recover the full register and structure state needed for a successful call into `IRP_CreateDevice`.

The lab setup also used a controlled kernel-mode redirection primitive with HVCI disabled. This makes the experiment useful for research and methodology development, but it should not be treated as a claim that the same approach applies unchanged to fully hardened production environments.

Another limitation is automation. Even if this approach works against some drivers, the methodology is likely to be highly driver-dependent. Each target may require different register values, object pointers, initialization state, and triggering conditions. That makes generic automation difficult unless the device-creation patterns can first be classified.

Future Work

The next step is to test this methodology against more drivers from the signed, loadable, device-API overlap set.

The `RUSB3XHC.sys` case showed that direct RIP redirection is not enough when the target function expects a specific calling context. However, other drivers may have simpler device-creation paths with fewer prerequisites. Those would be better candidates for a working proof of concept.

Future work will focus on:

1. Testing more drivers that contain both `IoCreateDevice` and `IoCreateSymbolicLink`.
2. Classifying the different patterns used for device creation.
3. Identifying drivers where the device-creation function requires minimal external state.
4. Automating the mapping between Ghidra-discovered callsites and WinDbg runtime breakpoints.
5. Recovering the expected register and structure state for candidate device-creation routines.
6. Comparing drivers that expose devices naturally against drivers that contain the APIs but fail to expose anything during normal loading.

The main goal is to understand which drivers are actually good candidates for forced device-exposure testing, and which ones are too dependent on driver-specific runtime state to be useful.

Security Disclosure

All work was performed inside an isolated lab environment on systems under our control.

The goal of this research is to improve Windows driver research methodology and better understand why some drivers expose reachable device interfaces while others do not. This post does not target any third-party system, and the techniques discussed here should only be used in controlled environments where you have explicit permission to perform this kind of testing.

Please follow proper security etiquette while performing similar research.

Conclusion

This research started with a simple problem: our driver-hunting pipeline had an extremely low yield. Even though many drivers contained references to `IoCreateDevice` and `IoCreateSymbolicLink`, very few actually exposed reachable user-mode device interfaces during normal testing.

To explore whether that yield could be improved, we tested whether controlled kernel-mode RIP redirection could be used to reach hidden or normally unreachable device-creation paths.

For `RUSB3XHC.sys`, the result was negative, but still useful. The target path existed, and parts of it were reachable during initialization, but directly redirecting execution into the suspected device-creation routine was not enough. The function expected a specific runtime context that we were not able to fully reconstruct.

The final takeaway is that forced device exposure is not a generic “jump to the right function” technique. It is a driver-specific reachability problem. RIP control may help us reach interesting code, but successful execution still depends on the correct register state, initialized structures, dispatch flow, and prerequisite conditions expected by the driver.

That makes the approach more limited, but also more interesting. If we can identify drivers with simpler creation paths and fewer state requirements, this technique may still help expand the reachable IOCTL surface available for future driver vulnerability research.