ERNW
providing security.

ERNW WHITE PAPER 73

ANALYZING WINPMEM DRIVER VULNERABILITIES

Version:	1.0
Date:	October 1, 2025
Classification:	Public

Julian Suleder

Digital unterschrieben von Julian Suleder
Datum: 2025.10.01 16:25:55 +02'00'

Table of Content

1 Handling

The present document is classified as *Public*. Any distribution or disclosure of this document REQUIRES the permission of the document owner as referred in Section *Document Status and Owner*.

1.1 Document Status and Owner

As the owner of this report, the document owner has exclusive authority to decide on the dissemination of this document and responsibility for the distribution of the applicable version in each case to the places.

The possible entries for the status of the document are *Initial Draft*, *Draft*, *Effective* and *Obsolete*.

Report Information	
Title:	ERNW White Paper 73 - Analyzing WinpMem Driver Vulnerabilities
Document Owner:	ERNW Enno Rey Netzwerke GmbH
Version:	1.0
Status:	Effective
Classification:	Public
Project Number:	-
Author(s):	Baptiste David, bdavid@ernw.de

Table 2: Document Status and Owner

1.2 Document Version History

Version	Date	Details
1.0	October 1, 2025	Initial version after quality assurance.

Table 3: Document Version History

ERNW
providing security.

2 Introduction

In the landscape of digital forensics, where accuracy and reliability are critical, even well-intentioned tools can present unexpected challenges. Recently, attention has been focused on WinpMem[1], an open-source driver designed for forensic memory analysis.

WinpMem is an open-source driver utilized to capture the complete memory contents of a system. While its functionality resembles that of a crash dump, the methodology differs significantly. Traditional crash dumps, such as those generated by Windows, involve halting system operation (i.e. to a Blue Screen of Death) to write the entire RAM content to disk, after which the system must be restarted. In contrast, WinpMem operates dynamically, capturing the full memory contents in real time, enabling analysis tools to process the data without requiring a system reboot.

It can be used in conjunction with many other forensic tools (although WinpMem is totally independent of them), such as Volatility or Velociraptor. In fact, the memory captured by WinpMem (in a raw format file, which is a bit old-fashioned nowadays) can be directly analyzed *off-line* by tools that come and parse the memory. But there is more. WinpMem proposes by design a "read device interface" to "run analysis on the live system (e.g., can be run directly on the device)". Said otherwise, it is possible to use the driver to get access to any portion of memory in the system.

WinpMem can be utilized alongside various other forensic tools (such as Volatility or Velociraptor) though it is developed entirely independently of them. The memory acquired by WinpMem is typically stored in a raw format file (a somewhat outdated approach today) called a dump file. This dump file can be analyzed offline using tools designed to parse and interpret the memory written inside. However, its capabilities extend further. WinpMem is designed to expose a "read device interface," enabling live system analysis directly on the target machine. In other words, it provides access to any portion of system memory in real time via the driver.

WinpMem is a forensic memory tool, developed by experts due to the complexity involved in creating a driver that interacts with advanced kernel structures to manage memory. It is primarily used by computer security professionals, as manipulating memory at this level is a highly specialized task. What could possibly go wrong? Quite a lot, as it turns out. In this discussion, we highlight several critical issues within this driver, which we presented at Recon 2025 in Montreal. Let's spoil.

[1] *https://github.com/Velocidex/WinPmem/blob/master/versioninfo.md*

3 Memory Reminder

Before diving into the driver, itself, a short refresh about how the memory works on Windows. If you are already familiar with this concept, please feel free to skip this section and directly read the next one. Considerations presented here are valid since Windows 7 (and even before in some cases) and they may apply the same way for other operating system compatible with Intel x86-x64 CPUs, despite few practical adjustments.

The memory in the machine is first divided in two entities. On the first side, the physical memory, sometime called Random Access Memory (RAM), by an abuse of language. This is literally the memory plugged to the mother board of the computer. It physically contains the memory of the system, and it has physicals limits for capacity (32 or 64 Gb, depending on the content plugged in the machine). On the other hand, the virtual memory. As its name suggests it, it is virtual, which means it is managed by the CPU and the operating system. This memory ultimately relies on the physical memory, but it is an artifice used to extend the total memory capacity and managing the parameters associated to that memory (read, write, or executable, among other things).

When a process uses memory to store or read information to/from it, this one access virtual memory, not directly the physical one. This is the responsibility of the CPU (managed by the operating system) to make the translation between the virtual address accessed in a process and the physical memory, where the real content is supposed to be accessed. This system is usually called Memory Management Unit (MMU) in computer science. What are the consequences of such a system?

Firstly, it allows every process to *use* its own set of memory called the *virtual address space of a process*[2] (abbreviated *address space*, for the sake of conciseness). In practice, the virtual address space for each process is private and cannot be accessed by other processes unless it is shared. More directly, it means that if process A stored at the address `0x401000` the `0x53` value, the process B (which is different from A) will not (or with a deep low probability due to hazard) get at the address `0x401000` the `0x53` value (but another value). This explains why different processes can see the same *address* of memory but not the same content at the same address. The reason is that they have the same virtual address, which is referenced, behind the stage, by the CPU/OS with different locations in physical memory. As a side note, the *shared memory* is nothing but two different (and sometimes the same) virtual addresses from two different processes but pointing to the same physical address in memory.

In practice, the memory is not managed byte per byte, but memory page of bytes per memory page of bytes. A page of memory[3] is usually composed of 4096 bytes, i.e. 4Kb. This value is defined by Intel, meaning that all operating system interfacing with such a CPU (but also with AMD) needs to manage memory per pages of that size.

[2] *https://learn.microsoft.com/en-us/windows/win32/memory/virtual-address-space*
[3] *https://learn.microsoft.com/en-us/windows/win32/memory/virtual-address-space-and-physical-storage*

This translation from virtual to physical memory is performed with a specific structure defined by Intel and managed by the kernel of the operating system, the memory Page Table Entry (PTE). This mechanism is not so complex to describe, but it would make that post longer that it is already. For short, each process has its own set of pages tables, used to perform the translation from virtual memory to physical ones (among other things, since this is also that structure which is used to define read/write/execute rights in the memory). In a way, the PTE mechanism allows making the conversion from virtual address to physical address. Internally, there is a special *kernel mode register* in the CPU called CR3 which is used to manage the PTEs of each process. This design has two consequences: at first, it means that changing the CR4 value in the CPU is *sufficient* (of course, more operations are involved to do it correctly, but for the today's post, this is sufficient) to *switch* from one virtual address space of a process to another one. Then, it means that only the kernel of the operating system (running in ring 0) is allowed to switch process virtual memory space context, since reading and writing to CR3 register is a privileged ring-0 instruction for the CPU.

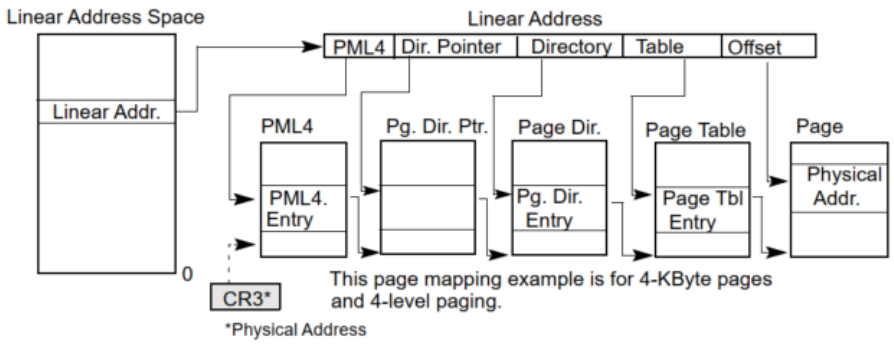


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging

Figure 1: Illustration from Intel Documentation relative to the Page Table model used to translate virtual address to physical address.

From a conceptual point of view, this is enough. Of course, the interested reader can go deeper with the concept of working set[4] which represents the virtual memory which is present in physical memory, introducing the notion of page fault and paged memory. For short, virtual memory can belong to the disk instead of physical memory for the sake of performances. This one will be loaded once accessed first in memory, via a mechanism of *page fault*. This is a relevant topic for the present discussion, especially talking about the design of WinPmem.

[4] *https://learn.microsoft.com/en-us/windows/win32/memory/working-set*

Finally, we need to introduce the notion of kernel- and user-mode memory space. On the first hand, the kernel memory is shared for the whole system, in a *pool* of memory. It means that every code running in kernel-mode shares the same *memory space* with all other codes running in kernel-mode. Of course, this memory is virtual memory and there is a centralized memory manager used to allocate and manage the kernel memory. On the other hand, and due to the design of the virtual memory, the virtual memory is specific to a process. This user-mode memory is split between different subtype of memories (stack, heap, ...) and the address space is always bellowing a limit (defined by the `MmHighestUserAddress` value in the kernel of Windows). In practice, the user-mode addresses are accessible to the process running in user-mode but also to kernel-mode code, if there is no Supervisor Mode Access Prevention (SMAP).

4 What Matters for Drivers?

Writing a driver is writing a program like any other, but it's not the same as writing a user-mode program. Not only does the API change, but there are other specificities. IRQL management, paged rather than paged memory, the slightest buffer overflow resulting in a BSOD… But a profound change in all aspects of security. It's important to understand that in the core, the notion of classic security is no longer relative. In a way, everything is allowed, the forbidden is the exception. And therein lies the difficulty.

What can we look for in a vulnerable driver? Mainly: *read/write/execute*-what-where. Behind this neologism, it is important to see the concept. What's interesting for an attacker is to gain access to the kernel's very special privileges:

- The ability to read any part of the memory without restriction is a major advantage, to bypass certain security features such as ASLR or credential/encryption key recovery.
- The ability to write any part of the memory allows unlimited modification of system, in particular regarding the security or kernel-mode code execution.
- The ability to execute any section of the memory allows a full control of the system.

With WinpMem, read access is immediate and direct by design. Indeed, the diver is designed to give unlimited access to all the memory present on the machine. This is typical of this kind of "security driver", whose purpose is for various reasons to *give access to information*, breaks the security of the system. There's no claim to make a read-what-where here, since this is a feature of the driver – even if the security issue is no less important.

However, there is an ambiguous limitation here. As explaining previously, the memory of a process is independent of another and, in practice, it directly depends on the CR3 register. By design, WinpMem captures the whole memory based on physical address, or on virtual address in the context of the calling process. There is no direct mechanism to switch from a process to another, allowing to capture the specific memory of each process. It means that even if it is possible to proceed with physical memory and the page file, the main purpose of a tool like WinpMem is to capture the memory of the kernel and optionally the memory of a process.

5 Diving Into WinpMem Drivers

Since WinpMem is open source, we propose to base our analysis on its source code directly.

The architecture of drivers is driven by the choice of one of the main technologies used to write a driver. From the good-old WDM to the WDF drivers, there are several possibilities. WinpMem is a WDM driver, meaning it uses relative classic and simple mechanisms to interface with the system.

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath)
{
    UNICODE_STRING DeviceName, DeviceLink;
    NTSTATUS ntstatus;
    PDEVICE_OBJECT DeviceObject = NULL;
    PDEVICE_EXTENSION extension;
    ULONG FastioTag = 0x54505346;
        UINT64 cr4 = 0;  // for level 5 check.

    UNREFERENCED_PARAMETER(RegistryPath);

    WinDbgPrint("WinPMEM - " PMEM_DRIVER_VERSION " \n");

    #if PMEM_WRITE_ENABLED == 1
    WinDbgPrint("WinPMEM write support available!\n");
    #endif

    RtlInitUnicodeString (&DeviceName, L"\\Device\\" PMEM_DEVICE_NAME);

    // We create our secure device.
    // http://msdn.microsoft.com/en-us/library/aa490540.aspx
    ntstatus = IoCreateDeviceSecure(DriverObject,
                                    sizeof(DEVICE_EXTENSION),
                                    &DeviceName,
                                    FILE_DEVICE_UNKNOWN,
                                    FILE_DEVICE_SECURE_OPEN,
                                    FALSE,
                                    &SDDL_DEVOBJ_SYS_ALL_ADM_ALL,
                                    &GUID_DEVCLASS_PMEM_DUMPER,
                                    &DeviceObject);

    if (!NT_SUCCESS(ntstatus))
    {
        DbgPrint ("IoCreateDevice failed. => %08X\n", ntstatus);
        // nothing to free until here.
        return ntstatus;
    }

    DriverObject->MajorFunction[IRP_MJ_CREATE] = wddCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = wddCreateClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = wddDispatchDeviceControl;
    DriverObject->MajorFunction[IRP_MJ_READ] = PmemRead; // copies tons of data, always in the range of Gigabytes.
```

Figure 2: Entry point function for the WinpMem driver.

As any driver, this has an entry point called `DriverEntry`[5]. Once the local variables have been initialized, there is a call to the `IoCreateDeviceSecure`[6] function to create a named device object used as a communication interface. This object enables subsequently a user-mode application to communicate with the driver. Interestingly, this communication interface is setup with the `DefaultSDDLString` parameter set to `SDDL_DEVOBJ_SYS_ALL_ADM_ALL`[7], meaning the access to the driver's communication interface is only restricted to administrators or system users. That way, the driver is not vulnerable, since the attacker must be running with administrator rights to get access to it. As Raymond Chen would say, it rather involved being on the other side of this airtight hatchway[8]. That being said, WinpMem is not an ordinary driver. This is often used in forensics or incident response contexts. Which means the machine may already have been compromised, for instance by malware. The malware may be already executed as an administrator (which remains a strong assumption) or have used a UAC-bypass[9] to do so. This particular context does not make the attack a sure thing, but it at least makes it plausible.

Then comes the initialization of the *I/O manager*[10] for the driver. The communication between an application and the driver is performed through the use of I/O request packets (IRPs) defined by IRP structures[11]. To handle IRPs[12] (we also say "*completing IRPs*"[13]), the driver registers some IRP handlers, also called *dispatch routines*[14,15]. The purpose (and thus the definition) of a dispatch routine depends on the I/O function code it handles. The I/O function code (IOCTLs) is also defined as the *IRP Major Function code*[16]. These are predefined by Microsoft (usually starting with `IRP_MJ` prefix) but they can be driver defined[17]. In the case of WinpMem, one can see that the dispatch routines for `IRP_MJ_CREATE`[18], `IRP_MJ_CLOSE`[19], `IRP_MJ_READ`[20], and `IRP_MJ_DEVICE_CONTROL`[21] codes are provided.

---

[5] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/writing-a-driverentry-routine*
[6] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdmsec/nf-wdmsec-wdmlibiocreatedevicesecure*
[7] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/sddl-for-device-objects*
[8] *https://devblogs.microsoft.com/oldnewthing/20150923-00/?p=91531*
[9] *According to the announcement made by Microsoft regarding the new* Administrator Protection *(see: https://techcommunity.microsoft.com/blog/microsoft-security-blog/evolving-the-windows-user-model-%E2%80%93-introducing-administrator-protection/4370453) feature, Microsoft claimed to have mitigated many UAC bypass techniques, except for the issue related to token manipulation attacks*
[10] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-i-o-manager*
[11] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_irp*
[12] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/different-ways-of-handling-irps-cheat-sheet*
[13] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/completing-irps*
[14] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/dispatch-routine-functionality*
[15] *As a side note, the driver purists talk about* routine *in kernel-mode and "function" in user-mode. This is a convenient way to make the distinction between the application field of a given code. Both terms are similar since they designate a function written in the end in a program.*
[16] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-major-function-codes*
[17] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/defining-i-o-control-codes*
[18] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-create*
[19] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-close*
[20] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-read*
[21] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-device-control*

The case of the `IRP_MJ_CREATE`[22] and `IRP_MJ_CLOSE`[23] codes are both handled by the same dispatch routine called `wddCreateClose`. This one is just a simple pass-through IRP[24] handler, whose lone purpose is to do the minimal actions to do nothing in particular. The `IRP_MJ_READ`[25] operation is the one which is supposed to provide the read-what-where capability to the calling application. The use case is simple. The calling application uses the `ReadFile`[26] function to provide to the driver an input buffer with the address to read (either in user-mode or kernel-mode), the size to be read, and an output buffer, allocated in user-mode, to retrieve the memory copied from the driver. More could be said about the way the memory is read, may be in another blog post if there is an interest about.

The most important I/O code when analyzing a driver is usually the `IRP_MJ_DEVICE_CONTROL`[27]. This handles the driver's defined actions implemented by the driver. Said otherwise, the driver can define its own *I/O code*s to execute driver's specific operations. Usually, for a user-mode application, the communication is performed via the `DeviceIoControl`[28] function, where it is possible to provide the I/O code, an input buffer, and an output buffer. For WinpMem, the dispatcher routine for the `IRP_MJ_DEVICE_CONTROL`[29] code is the `wddDispatchDeviceControl` function.

The beginning of the `wddDispatchDeviceControl` aims to retrieve the buffer provided by the `DeviceIoControl`[30] function call. This part is particularly tricky for driver developers. Why? Because the calling application, running in user-mode, provides its own buffers to the driver. It means the provided buffers are in user-mode (in the memory space of the calling application) but also under the total control of the application. That way, the application could provide invalid buffer, associated with invalid size, and so on. It is the driver's responsibility to take care of the provided buffer, with the appropriate safety procedures. The way to do it depends on the communication method used by the driver.

[22] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-create*
[23] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-close*
[24] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/how-to-complete-an-irp-in-a-dispatch-routine*
[25] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-read*
[26] *https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile*
[27] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-device-control*
[28] *https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol*
[29] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-device-control*
[30] *https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol*

```
NTSTATUS wddDispatchDeviceControl(_In_ PDEVICE_OBJECT DeviceObject, _Inout_ PIRP Irp)
{
    PIO_STACK_LOCATION IrpStack;
    NTSTATUS status = STATUS_SUCCESS;
    ULONG IoControlCode;
    PVOID inBuffer;
    PVOID outBuffer;
    PDEVICE_EXTENSION ext;
    ULONG InputLen, OutputLen;

    PAGED_CODE();

    Irp->IoStatus.Information = 0;

    if(KeGetCurrentIrql() != PASSIVE_LEVEL)
    {
        status = STATUS_SUCCESS;
        goto exit;
    }

    ext = (PDEVICE_EXTENSION)DeviceObject->DeviceExtension;

    IrpStack = IoGetCurrentIrpStackLocation(Irp);

    inBuffer = IrpStack->Parameters.DeviceIoControl.Type3InputBuffer;
    outBuffer = Irp->UserBuffer;


    OutputLen = IrpStack->Parameters.DeviceIoControl.OutputBufferLength;
    InputLen = IrpStack->Parameters.DeviceIoControl.InputBufferLength;
    IoControlCode = IrpStack->Parameters.DeviceIoControl.IoControlCode;

    switch (IoControlCode)
    {

    // Return information about memory layout etc through this ioctrl.
    case IOCTL_GET_INFO:
```

Figure 3: Beginning of the IOCTL dispatcher function from WinpMem driver.

Indeed, for WDM drivers, there are *three communication methods*[31] to exchange IRP buffers. For short:

- The `METHOD_BUFFERED`: the kernel is responsible to make the transition from the user-mode buffer to a kernel-mode buffer specifically allocated for that purpose. That way, a part of the security is handled by Windows which handles a large part of the buffer's security.
- The `METHOD_IN_DIRECT` or `METHOD_OUT_DIRECT`: either for input buffer or output buffer, access to the buffer's content is ensured by *Memory Descriptor List (MDL)*[32]. This structure is originally used to describe the physical page layout for a virtual memory buffer, especially in the case where contiguous virtual memory pages can be spread over several discontinuous physical pages. In practice, to access the pages safely regardless of process context,

[31] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/buffer-descriptions-for-i-o-control-codes*
[32] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-mdls*

drivers call calling the `MmProbeAndLockPages`[33] function and the `MmGetSystemAddressForMdlSafe`[34] macro to map an MDL safely. That way, the MDL mechanism ensures safe access to the buffer.

- The `METHOD_NEITHER` method: the user-mode buffers provided for input and output are directly accessible. More directly, the kernel does not provide any support to validate or map them the user-mode virtual addresses in the IRP. It is up to the driver to take care of the whole security. The name comes from the fact this method does neither use intermediate buffers nor MDL to handle the communication.

Managing the buffers securely with can be challenging, even following Microsoft's documentation[35]. But the most challenging is by far the `METHOD_NEITHER`. As a general rule, this method should be avoided except in the very few cases where it could be relevant (considering it could have relevant cases, which is far to be obvious). The specific management of the neither method, as described by Microsoft[36], is not rocket science, but it requires being particularly rigorous on every buffer's access. For the sake of simplicity (and avoiding edge-cases that need to be considered anyway), the user-mode buffers must be evaluated within driver-supplied exception handler (i.e. `__try/__except` blocs), evaluated with the `ProbeForRead`[37] (for input buffer) and `ProbeForWrite`[38] (for the output buffer) support routines. Ideally and as recommended by Microsoft, buffers should be copied into kernel-mode buffer specifically allocated for this purpose. Said otherwise, the best solution is to mimic the buffered method (hence the reason why preferring the buffered method to the neither method).

Because neither method should be avoided (*irony*), WinpMem decided to use the `METHOD_NEITHER`. I personally cannot explain why the `METHOD_NEITHER` has been so much used in the field of driver development, but I can guess the apparent simplicity (which is in practice the total opposite) of the method and the magic-power of copy and past from legacy source-code could be a good start. In most of the cases, observing the `METHOD_NEITHER` in a driver is a surefire mark of vulnerability.

[33] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmprobeandlockpages*
[34] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmgetsystemaddressformdlsafe*
[35] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/security-issues-for-i-o-control-codes*
[36] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-neither-buffered-nor-direct-i-o*
[37] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforread*
[38] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforwrite*

6 WinpMem Vulnerability I: TOCTOU

Following the code given in Figure 4 shows the way the WinpMem driver shares memory layout content with the calling application. The way the memory layout and other information shared is not what matters here, even if we could discuss the quality of a method using the undocumented `MmGetPhysicalMemoryRanges` routine. For the sake of reading, we propose to split the code in Figure 4 in two parts. On the one hand, in the blue box, there is the buffer's verification procedure, which follows the mainlines of the Microsoft documentation[39] regarding the buffer management in the `METHOD_NEITHER` context. The verification on the buffers is correctly performed with the `ProbeForRead`[40] and the `ProbeForWrite`[41] routines in `__try/__except` blocks. On the other hand, in the red box, the output buffer is casted into a `WINPMEM_MEMORY_INFO` structure, defined for WinpMem. This structure is first initialized to zero (with the help of the `RltZeroMemory`[42] macro) before being filed as expected in the `AddMemoryRange` routine.

The problem of this code is its two-step approach. Doing the verification first and subsequently using the buffer outside the safe-verification environment exposes the code to a time-to-check to time-to-use attack (*TOCTOU*[43] attack).

Please remember, with the `METHOD_NEITHER`, the output buffer is under the exclusive control of the user-mode application interfacing with the driver. This buffer is verified once but no more thereafter. The idea of the attack is to let the verification happens as expected and then update the buffer once the verification has been performed, during the initialization of the buffer, where the buffer is vulnerable.

[39] *https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-neither-buffered-nor-direct-i-o*
[40] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforread*
[41] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforwrite*
[42] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-rtlzeromemory*
[43] *https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use*

```
switch (IoControlCode)
{

// Return information about memory layout etc through this ioctrl.
case IOCTL_GET_INFO:
{
    PWINPMEM_MEMORY_INFO pInfo = NULL;

    if (!outBuffer)
    {
        DbgPrint("Error: outbuffer invalid in device io dispatch.\n");
        status = STATUS_INVALID_PARAMETER;
        goto exit;
    }

    if (OutputLen < sizeof(WINPMEM_MEMORY_INFO))
    {
        DbgPrint("Error: outbuffer too small for the info struct!\n");
        status = STATUS_INFO_LENGTH_MISMATCH;
        goto exit;
    }

    try
    {
        ProbeForRead( outBuffer, sizeof(WINPMEM_MEMORY_INFO), sizeof( UCHAR ) );
        ProbeForWrite( outBuffer, sizeof(WINPMEM_MEMORY_INFO), sizeof( UCHAR ) );
    }
    except(EXCEPTION_EXECUTE_HANDLER)
    {
        status = GetExceptionCode();
        DbgPrint("Error: 0x%08x, probe in Device io dispatch, outbuffer. A naughty process sent us a bad/nonexisting buffer.\n", status);

        status = STATUS_INVALID_PARAMETER; // Naughty usermode process, this is your fault.
        goto exit;
    }

    pInfo = (PWINPMEM_MEMORY_INFO) outBuffer;

    // Ensure we clear the buffer first.
    RtlZeroMemory(pInfo, sizeof(WINPMEM_MEMORY_INFO));

    status = AddMemoryRanges(pInfo);

    if (status != STATUS_SUCCESS)
    {
        DbgPrint("Error: AddMemoryRanges returned %08x. output buffer size: 0x%x\n",status,OutputLen);
        goto exit;
    }
```

Figure 4: Incorrect management of the output buffer in the context of the `IOCTL_GET_INFO` code.

How is it possible? The simple way is to change user-mode buffer rights on the flight. With the help of the `VirtualProtect`[44] function, it is possible to change memory's rights. For instance, from read/write to read-only. That way, any further write access to the user-mode buffer would result in an exception. And such an exception in kernel-mode inevitably results in a Blue Screen Of Death[45,46]. Note other strategies are possible, like releasing memory during use, and so on…

[44] *https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect*
[45] *https://devblogs.microsoft.com/oldnewthing/20240730-00/?p=110062*
[46] *https://www.crowdstrike.com/en-us/blog/falcon-content-update-preliminary-post-incident-report/*

To be able to proceed, the user-mode application creates two threads[47]. One to send the I/O code `IOCTL_GET_INFO` with the `DeviceIoControl`[48] function to the driver and another one to repeatedly change the memory rights of the user-mode buffer. The operation performed by the `DeviceIoControl`[49] function is indefinitely looped.

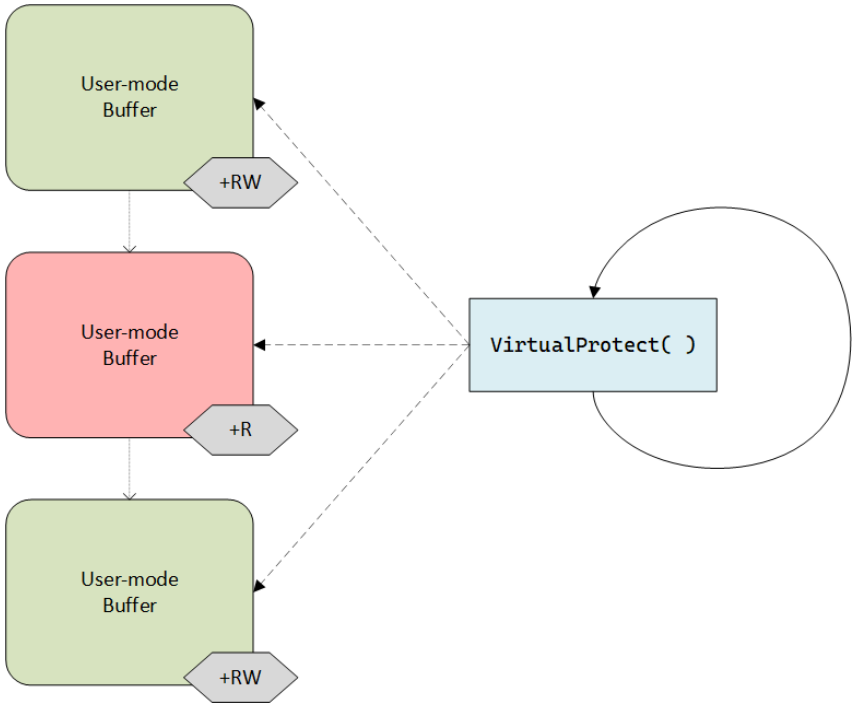


Figure 5: Illustration of the buffer's right change exploitation.

The goal is to perform the operation many times, so that the verification has a chance to be validated, but the following access to the buffer will trigger an unhandled exception. In that case, it means `VirtualProtect`[50] function call should let the user-mode buffer to be writable in the blue box of the Figure YY and to be read-only in at least one of the routines called in the red box.

[47] *https://learn.microsoft.com/en-us/windows/win32/procthread/creating-threads*
[48] *https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol*
[49] *https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol*
[50] *https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect*

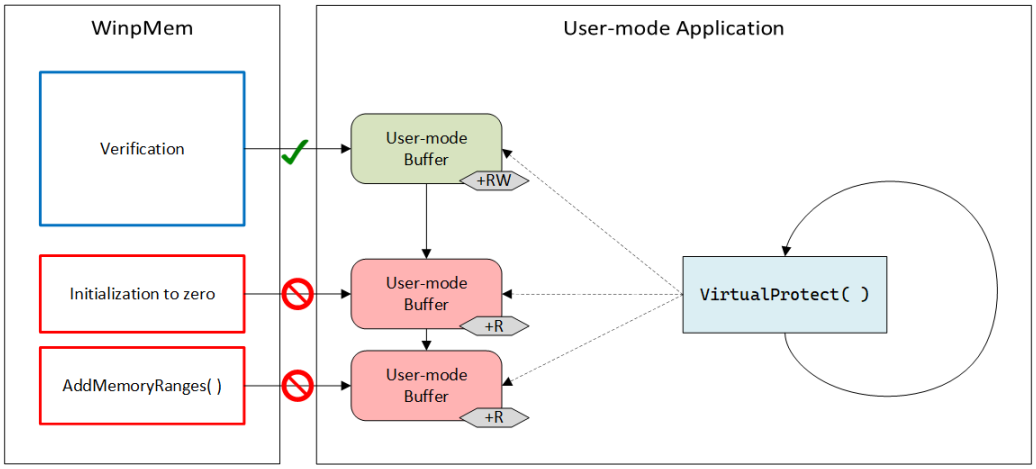


Figure 6: Illustration of the method to exploit the time-to-check-time-to-use vulnerability.

7 WinpMem Vulnerability II: write-zero-where

To be exploited, this requires a redacted version of the driver, since the issue is on an old version, but this one is not directly exploitable due to other limitations of the old version of the driver. It is provided to show different exploitation techniques.

A second issue involves another IOCTL provided by WinpMem: `IOCTL_REVERSE_SEARCH_QUERY`. This IOCTL converts a virtual address (valid only within a specific process) into a physical address (referring to actual hardware memory). As with `IOCTL_GET_INFO`, the virtual address is passed via a buffer fully controlled by the calling application. However, in this case, the buffer is properly handled: it is probed and copied into a local variable within a `__try/__except` block, and the original buffer is not accessed thereafter. The same applies to the output buffer that stores the physical address, ensuring no TOCTOU vulnerability is present.

```
try
{
    ProbeForRead( inBuffer, sizeof(UINT64), sizeof( UCHAR ) );
    ProbeForWrite( outBuffer, sizeof(UINT64), sizeof( UCHAR ) );

    In_VA.value = *(PUINT64) inBuffer;

}
except(EXCEPTION_EXECUTE_HANDLER) { ... }
```

Figure 7: Part of the code which correctly handles the input and the output buffers in a search query operation. Older version did not implement such operation.

Internally, the resolution of the physical address is handled by the `virt_find_pte` function, which uses `MmGetVirtualForPhysical`[51] to obtain the PTE associated with the given virtual address. Upon completion, three outcomes are possible:

1. the function fails (i.e., the return value is not `PTE_SUCCESS`), in which case the local variable `Out_PhysAddr` is set to zero by default;
2. the function succeeds, but the virtual address does not map to a resident memory page, so no physical address exists and `Out_PhysAddr` is also set to zero;
3. the PTE is valid, and `Out_PhysAddr` holds the corresponding physical address.

In all cases, the value of `Out_PhysAddr` is copied into the output buffer following a successful probing operation. But since the `METHOD_NEITHER` is used, the address supplied by the user-mode application is passed directly to the driver. This means it can reference either user-mode or kernel-mode address for memory buffer. As the driver accesses the buffer directly, there is no built-in restriction preventing writes to a kernel-mode address.

[51] *https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-mmgetphysicaladdress*

With the event of the lack of proper validation, by supplying an invalid address (for instance, a user-mode address referencing unallocated memory) to the `IOCTL_REVERSE_SEARCH_QUERY` request, it is possible to write a zero value (i.e., the default content of the output buffer) to an arbitrary location in memory if the `virt_find_pte` function fails. This is a classic vulnerability in driver development, resulting from improper handling of user-supplied buffers. It is worth noting that, if the `virt_find_pte` function succeeds, the physical address is written directly to the output buffer address provided by user mode.

8 Exploitation of the WinpMem Vulnerability II

When exploiting a vulnerable driver, the primary objective is often to achieve both *read-what-where* and *write-what-where* capabilities. This is because arbitrary read access allows an attacker to extract sensitive information from kernel memory, effectively bypassing protections such as Address Space Layout Randomization (ASLR). For example, an attacker could retrieve BitLocker-related components (including encryption keys), access tokens of critical processes, or the addresses of sensitive kernel structures. With this knowledge, arbitrary write access can then be used to modify these structures. For instance, it becomes possible of altering security callbacks to disable integrity checks or modifying process tokens to escalate privileges. Ultimately, unrestricted read and write capabilities in kernel mode grant the attacker control equivalent to the highest privilege level within the operating system.

The concept of *read-what-where* refers to the ability to read arbitrary locations in physical or kernel memory. With a driver like WinpMem, this capability is trivial to achieve, as it is inherently supported by the driver's design.

The term *write-what-where* refers to the ability to write arbitrary data to any location in memory, particularly within kernel memory. In our case, however, the capability is more accurately described as *write-zero-where*, since it is limited to writing zero values at arbitrary addresses rather than arbitrary data. Although subtle, this distinction significantly impacts the potential for exploitation.

8.1 Historical Exploitation: gCiOptions

A traditional method of exploiting a *write-zero-where* vulnerability involves modifying the `gCiOptions` global variable in kernel memory. This variable controls the enforcement of Driver Signature Enforcement (DSE)[52]. On 64-bit versions of Windows[53], drivers must be digitally signed. The signature verification is performed in kernel mode, specifically by the Code Integrity[54] module (`CI.dll`).

To enforce the DSE policy, `CI.dll` first checks the value of the `gCiOptions`[55] variable. If this value is set to anything other than `0x00` (disabled[56]) or `0x08` (test mode[57]), driver signature enforcement is active. By leveraging a *write-zero-where* vulnerability, an attacker can set `gCiOptions` to zero, effectively disabling[58] the signature check. This allows the loading of unsigned drivers using only standard administrator privileges.

[52] *https://learn.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing*
[53] *Starting from Windows XP 64-bit and in all subsequent versions.*
[54] *Code Integrity also takes care of other security checks, such as WDAC (see: <https://learn.microsoft.com/en-us/hololens/windows-defender-application-control-wdac) or Smart App Control, among others.*
[55] *Prior to Windows 8, it was the Boolean `g_CiEnabled` value, stored in `ntoskrnl`, that was checked.*
[56] *https://v1k1ngfr.github.io/loading-windows-unsigned-driver/*
[57] *https://learn.microsoft.com/en-us/windows-hardware/drivers/install/the-testsigning-boot-configuration-option*
[58] *https://v1k1ngfr.github.io/loading-windows-unsigned-driver/*

Beginning with Windows 8.1, Microsoft introduced a mitigation known as Kernel Patch Protection[59] (KPP), also referred to as PatchGuard, which periodically verifies the integrity of critical kernel variables. The `gCiOptions` variable is among those monitored to prevent unauthorized modifications. However, the response time of this mechanism is non-deterministic, creating a period (for a few seconds) during which an attacker could modify `gCiOptions` and load an unsigned driver before detection occurs. To address this limitation, Windows 10 introduced Hypervisor-Protected Code Integrity (HVCI)[60], also known as HyperGuard. Built on Hyper-V[61] and part of the Virtualization-Based Security (VBS) framework[62], HVCI leverages the hypervisor to enforce access control over memory pages marked as highly sensitive, including the one containing `gCiOptions`. Unlike PatchGuard, this approach provides immediate protection, eliminating any exploitable period and ensuring robust enforcement of code integrity.

8.2 Historical Exploitation: The Swan Song for DSE Tampering

If HVCI can check for read-only values, it cannot ensure the protection of dynamic structures, which are evolving over time. In particular, in the context of DSE, the `iValidateImageHeader` function[63] is called part of CI.dll. This routine is set up, in the `SeCiCallbacks` structure (in `ntoskrnel`) at the end of the `SepInitializeCodeIntegrity` routine. The idea of the attack is to replace the `CiValidateImageHeader` routine by another routine in the kernel returning always `STATUS_SUCCES` (`0x00`).

While HVCI effectively protects read-only values, it cannot guarantee the integrity of dynamic structures that change over time. In the context of DSE, the `CiValidateImageHeader` function[64] (which belongs in `CI.dll`) is responsible for validating image signatures. This function is registered within the `SeCiCallbacks` structure in `ntoskrnl`, specifically during the final steps of the `SepInitializeCodeIntegrity` routine. The core idea of the attack is to overwrite the `CiValidateImageHeader` pointer with the address of any kernel routine that always returns `STATUS_SUCCESS` (`0x00`), thereby bypassing signature checks since the return value is usually the only element considered validating a signature check.

In this case, the limitation of the attack[65] lies in its requirement[66] for a *write-what-where* capability. Simply zeroing the entry corresponding to the `CiValidateImageHeader` routine within the `SeCiCallbacks` structure would result in a system crash.

[59] *https://en.wikipedia.org/wiki/Kernel_Patch_Protection*
[60] *https://learn.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard*
[61] *https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-overview?pivots=windows*
[62] *https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs*
[63] *https://www.cybereason.com/blog/code-integrity-in-the-kernel-a-look-into-cidll*
[64] *https://www.cybereason.com/blog/code-integrity-in-the-kernel-a-look-into-cidll*
[65] *https://www.fortinet.com/blog/threat-research/driver-signature-enforcement-tampering*
[66] *https://blog.cryptoplague.net/main/research/windows-research/the-dusk-of-g_cioptions-circumventing-dse-with-vbs-enabled*

8.3 Historical Exploitation: Pimp My PID

Another possible attack using a vulnerable driver consists of replacing the access token[67] of the attacker's process with the access token of the system process. This grants the attacker full system-level privileges. The effectiveness of this technique[68] lies in the fact that access tokens are not monitored by HVCI or KPP. However, the attack requires the ability to write an arbitrary value specifically at the address of the system process's access token. As such, it depends on a *write-what-where* capability, which cannot be achieved with a *write-zero-where* capability alone.

8.4 Historical Exploitation: Nullify the Security Descriptor

Like the previous attack[69], a *write-zero-where* primitive can be used to achieve nearly the same effect. The approach consists of removing the security descriptor[70] from the object representing the system process, thereby bypassing any security checks when accessing it. For an object without a security descriptor, Windows grants unrestricted access to all users. Applying this to the system process allows the attacker's process to read from, write to, or execute code within the system process's context, effectively enabling indirect access to kernel memory.

Although this attack meets all our criteria, it has been mitigated[71] in Windows 10 version 1607 (Build 14393). Attempting to access an object with a nullified security descriptor now triggers a Blue Screen of Death with a `BAD_OBJECT_HEADER` error. However, the blog post[72] describing this mitigation suggests that nullifying the Discretionary Access Control List (DACL)[73] within the security descriptor may still allow the original behavior to be reproduced. This alternative approach has not yet been tested by us but warrants further investigation on the latest Windows versions.

[67] *https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens*
[68] *https://v1k1ngfr.github.io/pimp-my-pid/*
[69] *https://v1k1ngfr.github.io/pimp-my-pid/*
[70] *https://learn.microsoft.com/en-us/windows/win32/secauthz/security-descriptors*
[71] *https://www.lrqa.com/en/cyber-labs/analysing-the-null-securitydescriptor-kernel-exploitation-mitigation-in-the-latest-windows-10-v1607-build-14393/*
[72] *https://www.lrqa.com/en/cyber-labs/analysing-the-null-securitydescriptor-kernel-exploitation-mitigation-in-the-latest-windows-10-v1607-build-14393/*
[73] *https://learn.microsoft.com/en-us/windows/win32/secauthz/dacls-and-aces*

8.5 New Exploitation: write-anything-where

Another potential approach using the *write-zero-where* primitive would be to clear specific flags within a protected process[74] (or a protected process light[75]) such as an antivirus service[76] like Windows Defender[77]. However, this method would not enable actions such as loading an unsigned driver.

To be frank, the capabilities provided by the *write-zero-where* primitive are quite limited. Due to existing Windows mitigations and the restricted set of targetable objects, exploitation opportunities are increasingly constrained. However, the second issue in WinpMem offers more than just a *write-zero-where* capability. In addition to writing zeros, the `IOCTL_REVERSE_SEARCH_QUERY` IOCTL allows writing a physical address, a non-null 64-bit value representing a memory page. But this address is random and difficult to predict from user mode. And as anyone knows, manipulating random values in kernel-mode is not a good idea. But this is exactly what we are going to do, with a *write-anything-where* capability.

Given the guarantee that the physical address cannot be null[78], this value can be used to overwrite certain fields in the system with a non-zero value. In the context of Boolean logic, any value not explicitly defined as false is interpreted as true. Consequently, if specific values are set to false for security purposes, overwriting them with a non-zero value may effectively enable or bypass restricted functionality.

In practice, possessing both *read-what-where* and *write-what-where* capabilities is functionally equivalent to having access to a kernel debugger[79]. When analysing how WinDbg operates in kernel mode, it becomes clear that it heavily relies on the undocumented `KdSystemDebugControl` routine within `ntoskrnl.exe`. This routine follows a philosophy close to the one of `DeviceIoControl`[80] in user mode: it accepts a command code specifying the desired debugging operation (such as reading or writing memory, accessing the system bus, or interacting with MSRs), along with an input buffer (typically for write operations) and an output buffer (typically for read operations).

Crucially, the effectiveness of `KdSystemDebugControl` is determined by a set of initial condition checks. As illustrated in the figure below, the routine verifies several global flags: `KdpBootedNodebug`, `KdPitchDebugger`, `KdDebuggerEnabled`

[74] *https://www.crowdstrike.com/en-us/blog/evolution-protected-processes-part-1-pass-hash-mitigations-windows-81/*
[75] *https://medium.com/@boutnaru/the-windows-security-journey-ppl-protected-processes-light-831d5f371004*
[76] *https://learn.microsoft.com/en-us/windows/win32/services/protecting-anti-malware-services-*
[77] *https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Cyber-Sicherheit/SiSyPHus/Microsoft_Antivirus.pdf?__blob=publicationFile&v=2*
[78] *Although highly unlikely in practice, if a memory page would have a physical address set to zero, the returned address can easily be verified. In such a case, a new physical address can be obtained by providing the WinpMem driver with the virtual address of any other valid memory page, different from the one previously associated with the zero physical address.*
[79] *https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-kernel-mode-debugging-in-windbg--cdb--or-ntsd*
[80] *https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol*

, and `KdLocalDebugEnabled`. In practice, if either `KdDebuggerEnabled` or `KdLocalDebugEnabled` is set to zero, all debugging operations through the Windows kernel are disabled.

The `KdDebuggerEnabled` flag is initialized by the `KdEnableDebuggerWithLock` function, which is called during kernel initialization based on the system's boot configuration. Specifically, this is the `/DEBUG` flag[81] set via `bcdedit`[82]. The `KdLocalDebugEnabled` variable is set by the `KdInitSystem` routine, also during initialization, after parsing the `dbgsettings`[83] configuration to check for the `/LOCAL` flag[84]. This flag enables local kernel debugging, where the debugger operates on the same system being debugged.

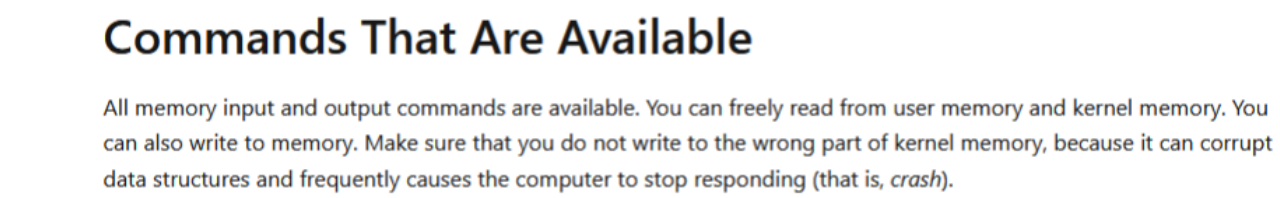
While local debugging is more restricted than remote debugging (e.g., breakpoints in kernel-mode processes are not feasible due to potential system-wide freezes), it still allows reading from and writing to memory. This capability is confirmed in Microsoft's own documentation[85] on local kernel-mode debugging.

```
NTSTATUS __stdcall KdSystemDebugControl(
        ULONG Command,
        PVOID InputBuffer,
        ULONG InputBufferLength,
        PVOID OutputBuffer,
        ULONG OutputBufferLength,
        ULONG *ReturnLength,
        char PreviousMode)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v31 = 0;
  v32 = 0LL;
  P = 0LL;
  if ( (KdpBootedNodebug || KdPitchDebugger || !KdDebuggerEnabled) && !KdLocalDebugEnabled )
    return STATUS_ACCESS_DENIED;
  if ( (int)Command > 14 )
  {
    Command_m15 = Command - 15;
    if ( !Command_m15 )
    {
      if ( InputBufferLength != 32 )
        return STATUS_INFO_LENGTH_MISMATCH;
      v44 = *(_OWORD *)InputBuffer;
      v45 = *((__m128i *)InputBuffer + 1);
      v30 = _mm_cvtsi128_si32(v45);
      if ( v30 )
      {
        LOBYTE(InputBufferLength) = PreviousMode;
        v15 = ExLockUserBuffer(*((_QWORD *)&v44 + 1), v30, InputBufferLength, 0LL, &v32, &P);
        if ( v15 >= 0 )
        {
          Msr = KdpSysWriteIoSpace(v45.m128i_i32[1], v45.m128i_i32[2], v45.m128i_i32[3], v44, v32, v30, (__int64)&v31);
```

Figure 8: Reverse engineering of the `KdSystemDebugControl` function responsible to handle kernel-mode debugging operations.

[81] *https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit--debug*
[82] *https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/bcdedit*
[83] *https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit--dbgsettings*
[84] *https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit--dbgsettings*
[85] *https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/performing-local-kernel-debugging*



Commands That Are Available

All memory input and output commands are available. You can freely read from user memory and kernel memory. You can also write to memory. Make sure that you do not write to the wrong part of kernel memory, because it can corrupt data structures and frequently causes the computer to stop responding (that is, *crash*).

Figure 9: Part of Microsoft's documentation related to the possibilities provided by local kernel debugger.

Returning to the `KdSystemDebugControl` routine, it requires that both the `KdDebuggerEnabled` and `KdLocalDebugEnabled` variables be non-zero to grant access to kernel-debugging operations. With the *write-anything-where* capability, it is possible to safely modify these values. Specifically, the `KdLocalDebugEnabled` flag is stored in `ntoskrnl` as a 32-bit value, immediately followed in memory by the `KdPrintRolloverCount` value, which is used only in non-critical contexts within the `KdSetDbgPrintBufferSize` and `KdLogDbgPrint` routines.

```
140C31C6C KdDebuggerEnteredCount dd ?              ; DA
140C31C70 KdDebuggerEnteredWithoutLock dd ?        ; DA
140C31C74 KdpContextSent  db ?                     ; DA
140C31C74                                          ; Kd
140C31C75                 align 8
140C31C78 KdIgnoredSavingSupervisorXStateFeatures dq ?
140C31C78                                          ; DA
140C31C80 KdLocalDebugEnabled db ?                 ; DA
140C31C80                                          ; Nt
140C31C81                 align 4
140C31C84 KdPrintRolloverCount dd ?                ; DA
140C31C84                                          ; Kd
```

Figure 10: Display of the `KdLocalDebugEnabled` global value in `ntoskrnl.exe`, from IDA software.

The case of the `KdDebuggerEnabled` value is more complex. This value is an 8-bit field, likely representing a BOOLEAN, and is preceded in memory by the `KdEventLoggingEnabled` 8-bit value. By default, on a debugged system, `KdDebuggerEnabled` is set to zero, and it may be desirable to maintain this state to avoid disrupting the overall system. To achieve this, it may be necessary to obtain multiple physical addresses: one whose least significant byte is zero, followed by another whose least significant byte is non-zero. In practice, locating such physical addresses is not particularly difficult since many follow this pattern.

The addresses of the `KdDebuggerEnabled` and `KdLocalDebugEnabled` global variables can be indirectly obtained by a user-mode application through the `NtQuerySystemInformation` function. By design[86], this function allows leak-

[86] *https://recon.cx/2013/slides/Recon2013-Alex%20Ionescu-I%20got%2099%20problems%20but%20a%20kernel%20pointer%20ain%27t%20one.pdf*

ing the base address of `ntoskrnl.exe`. Once the base address is acquired, the user-mode application can map[87] the `ntoskrnl.exe` file in user-mode memory and locate the global variables either by disassembling the initial opcodes of the `KdSystemDebugControl` routine or by using debugging symbols[88]. This process yields the offset of the global variables relative to the mapped base address. Adding this offset to the leaked base address provides the virtual addresses of the global values.

```
140C44378 ; Exported entry 1122. KdEventLoggingEnabled
140C44378                 public KdEventLoggingEnabled
140C44378 KdEventLoggingEnabled db ?              ; DATA
140C44378                                         ; KdChe
140C44379 ; Exported entry 1116. KdDebuggerEnabled
140C44379                 public KdDebuggerEnabled
140C44379 ; PBOOLEAN KdDebuggerEnabled
140C44379 KdDebuggerEnabled dq ?                  ; DATA
140C44379                                         ; MiQue
140C44381                 align 8
140C44388 qword_140C44388 dq ?                    ; DATA
140C44388                                         ; PipDg
140C44390                 align 20h
```

Figure 11: Display of the `KdDebuggerEnabled` global value in `ntoskrnl.exe`, from IDA software.

The next step is to translate the virtual address of the memory page containing these global variables into its corresponding physical addresses. This can be achieved using the `IOCTL_REVERSE_SEARCH_QUERY` operation. With the physical address in hand, it becomes possible to modify these global values via the same IOCTL by setting both flags to one. In this scenario, the output address supplied to the IOCTL corresponds to the virtual address of the global variable to be modified.

Consequently, a small program can be developed to emulate a kernel debugger, enabling direct interaction with the Windows kernel from user mode.

Is this solution stable? Frankly, it may lack stability. Modifying these values on a multicore CPU system is inherently challenging due to potential cache synchronization issues. Although our tests did not reveal any specific protections, such as Kernel Patch Protection (KPP), their implementation in future versions cannot be ruled out. Testing on Windows 10 demonstrated reasonable success, albeit with occasional sporadic crashes. Such exploits are inherently difficult to stabilize across all Windows versions. Therefore, this method is better regarded as a proof of concept rather than a fully reliable exploit.

[87] *https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile*
[88] *https://learn.microsoft.com/en-us/windows/win32/dxtecharts/debugging-with-symbols*

Is this solution stable? Frankly, this solution can lack stability. Indeed, changing these values on a machine with a multicore CPU is always challenging since cache synchronization issues could happen. Also, it is not impossible that on some version of Windows, a protection with KPP could have been (or could be) implemented. Tested on Windows 10, it worked quite well despite some sporadic crashes sometimes (but not always, some due to WinpMem's instability), but these kinds of exploits are always complex to stabilize on every version of Windows. Definitively more a proof of concept than a reliable exploit.

9 Conclusion

This type of vulnerability is relatively common in driver development. In the case of WinpMem, CVE-2024-10972 and CVE-2024-12668 have been assigned and addressed in the latest release. Although TOCTOU attacks are a well-established concept, they remain effective against a significant number of drivers, particularly those employing the *METHOD_NEITHER* approach to handle user-mode buffers. The second issue, related to the "write-zero-where" primitive, is especially notable due to its potential to demonstrate a "write-anything-where" scenario. This exploitation capability is unprecedented, to the best of our knowledge. This opens new opportunities for exploitation in situations where a conventional "write-what-where" primitive is not feasible.

Remediating vulnerabilities in a driver is never a trivial matter. Beyond the implementation of the actual bug fix, it also necessitates the deployment of an updated driver version. In the case of WinpMem, this process was further complicated by two major challenges.

The first challenge is not specific to WinpMem but applies broadly to all drivers: once a driver is vulnerable, it remains exploitable indefinitely. This is because, to run, a driver must be digitally signed[89] and installed with administrative privileges. From an attacker's perspective, any driver that has already been signed and contains a vulnerability can be reused on any system. This concept is known as *'Bring Your Own Vulnerable Driver'* (BYOVD), as demonstrated by the extensive list of such drivers publicly available on platforms like loldrivers.io[90].

The second issue pertains to the challenges of maintaining and updating open-source projects. WinpMem is maintained by a small yet dedicated community, moreover, there is no built-in update mechanism in the projects. Furthermore, the presence of unofficial forks (often based on partial or direct code copies) complexifies the issue. As a result, users must proactively verify and apply updates themselves, a task that many are unlikely to perform regularly. Additionally, driver signing requires a valid code-signing certificate, with a signing private key of which must be kept secure. WinpMem has publicly stated[91] that it no longer signs new driver versions, instead recommending the use of test signing mode[92]. While this mode is unsuitable for production or professional environments, it implies there is no operational, securely signed, and patched version of WinpMem readily available. This presents a significant concern: either someone else must sign the driver for broader distribution (potentially perpetuating the use of an inherently vulnerable design) or users must self-sign it for private use. Neither scenario provides a robust or scalable solution.

As explained, the fact that WinpMem device's access is limited to administrators restrict drastically the possibilities of exploitation. Nonetheless, the fact that WinpMem can be used in an already compromised environment by design still

[89] *https://learn.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing*
[90] *https://www.loldrivers.io/*
[91] *https://github.com/Velocidex/WinPmem/commit/afc9e0633b2bfe21267db47278dfa6d3ef3547f3#diff-06572a96a58dc510037d5efa622f9bec8519bc1beab13c9f251e97e657a9d4ed*
[92] *https://learn.microsoft.com/en-us/windows-hardware/drivers/install/test-signing*

make the exploitation context relevant. But may be the most relevant point regarding WinpMem is its ability to provide by design a read-what-where, no matter the security of the system. This kind of driver (and it is not the only one) is by design a vulnerability, since it breaks one of the most fundamental security rules applying in any operating system: the memory separation between the user-mode and the kernel-mode.

As previously discussed, access to the WinpMem device is restricted to administrators significantly limits the potential for exploitation. Nevertheless, its intended use within already compromised environments continues to make it relevant in exploitation scenarios. Perhaps the most critical concern regarding WinpMem, however, lies in its inherent ability to provide unrestricted "read-what-where" access, regardless of the system's security posture. Drivers of this kind (and WinpMem is not unique in this regard) constitute a vulnerability by design, as they violate one of the most fundamental principles of operating system security: the strict separation between user-mode and kernel-mode memory.

We presented everything in the recon-2025[93] edition. We detail the vulnerability and exploitation in the slides of the talk and in this blog post (or may be a series of blog post related to driver's vulnerabilities may follow), illustrating that "the cobbler always wears the worst shoes", as the old saying goes.

[93] *https://cfp.recon.cx/recon-2025/featured/*