SPLINEUSER
CLIENT CASE STUDY

WINDOWS KERNEL DRIVER EXPLOITABILITY RESEARCH

Out-of-Bounds Kernel Pool Write → Exploitability Assessment

PREPARED FOR
Private Client

PREPARED BY
Priyan S.

DATE
21st Feb 2026

This document is a sanitized case-study version of the original D02.E01 report. Client identifiers, target identifiers, ROP addresses, offsets, and exploit-operational materials have been intentionally removed, generalized, or redacted.

Windows Kernel Driver Exploitability Research

Case Study Overview

This case study summarizes a Windows kernel driver exploitability research engagement, this was part of a bigger full kernel code execution (KCE) chain development task. For this specific driver, the core question was simple: given a reachable driver bug with an out-of-bounds write into kernel pool memory, could that bug be shaped into a practical path toward kernel-mode code execution?

The goal was to understand the vulnerable driver path, characterize the primitives gained from the bug, test how practical those primitives were, and map the remaining engineering work needed to move from “bug exists” to a usable kernel exploitation path.

The research confirmed a meaningful kernel exploitation surface. The vulnerable IOCTL path was identified, the out-of-bounds kernel pool write was analyzed, and the available primitives were characterized. A major part of the work focused on turning this bug from an initially unreliable kernel read primitive into something repeatable enough to support further exploit development. This required debugging why requests failed, identifying the blocker in the driver’s execution path, and adjusting the input strategy so the driver would complete the relevant code path more reliably.

The research also explored a possible path toward controlled kernel execution through pool layout manipulation and target structure corruption. The work validated that pool adjacency could be influenced in the tested environment and that a candidate kernel structure contained a control-flow-relevant field worth targeting. At the same time, the research did not overclaim completion: the final controlled-execution trigger was not completed during this phase.

The main remaining blocker was synchronization. The target allocation was short-lived, which made it difficult to both verify the expected memory layout and safely perform the final corruption before the allocation was released, reused, or caused instability. In other words, the project moved past the “is there a real bug?” stage and into the harder exploit-engineering stage: reliability, timing, safe corruption, and crash-free execution.

The outcome of this research phase was a clear exploitability assessment: the driver exposed a serious kernel attack surface, useful primitives were validated, a plausible route toward kernel-mode code execution was mapped, and the remaining work was narrowed down to specific engineering blockers rather than open-ended discovery. A full KCE chain was later developed through a separate vulnerability chain, which is outside the scope of this public case study.

This public version is a sanitized portfolio case study derived from private client-approved research. Client identifiers, target-specific details, memory addresses, offsets, ROP material, and exploit-operational steps have been removed, generalized, or redacted. The goal of this document is to show the research process, technical reasoning, and deliverable quality without exposing sensitive materials or identifiers.

Research Objective and Final Answer

The research objective was to answer one main question: given the out-of-bounds kernel pool write exposed by the driver, could this bug realistically be developed into a path toward kernel-mode code execution?

The goal was to understand what the bug actually gives us in practice. More specifically, the research focused on identifying the vulnerable driver path, understanding the primitive gained from the bug, testing whether the primitive could be made reliable, and mapping what would still be required to turn the vulnerability into a stable kernel exploitation path.

From the work, the answer turned out to be a yes.

The research confirmed that the driver exposed a meaningful kernel attack surface. The vulnerable IOCTL path was reachable, the out-of-bounds kernel pool write was identified, and the bug could be reasoned about as more than a simple crash condition. From there, the research showed that the driver's copy behavior could also be shaped into a kernel read primitive. This was important because the kernel read primitive gave us a way to inspect kernel memory and verify whether our expected pool layout was actually present before attempting corruption.

The first version of the kernel read primitive was not reliable enough. Through testing, most requests failed, and a simple retry loop was not enough to solve the issue. The useful part was identifying why the request was failing. After debugging the driver path, the blocker was narrowed down to object-handling behavior inside the execution flow. Once that was understood, the input strategy was adjusted so that the driver would complete the relevant path much more consistently. This moved the primitive from working inconsistently (<2% of the time, required reboot to retry) to something that was consistent (~100% of the time).

The research also identified a plausible path toward controlled kernel execution. The approach was to influence pool layout, place a controlled allocation near a useful kernel structure, and target a control-flow-relevant field inside that structure. This path was not just theoretical: pool adjacency was observed in the tested environment, and a candidate structure with a useful execution field was identified and manually validated.

However, the final controlled-execution trigger was not completed during this phase. The main blocker was not the original vulnerability itself. The blocker was reliability around a short-lived allocation. In order to safely move toward kernel code execution, the exploit path needed to verify the nearby structure, preserve important fields (alongside windows 11 pool structure protection), apply the corruption, and then trigger execution without causing a bugcheck. The problem was that the allocation being used for this layout was volatile, which made synchronization the next real engineering hurdle.

The vulnerability is real, the kernel attack surface is meaningful, useful primitives were validated, and a plausible kernel-mode code execution path was mapped. However, full reliable KCE was not completed in *this* research phase. The remaining work was later done and required another vulnerability chain for a reliable KCE exploit.

Scope & Assumptions

The scope of this research was focused on exploitability assessment and exploit-path development for the Windows kernel driver vulnerability.

In-Scope:

1. **vulnerable driver analysis (Root-Cause Analysis)**
2. **IOCTL (func: IRP_MJ_DEVICE_CONTROL) review**
3. **Out-of-bounds (OOB) kernel pool write vulnerability analysis**
4. **Kernel read primitive development**
5. **Reliability Testing & Development process**
6. **pool grooming and layout analysis**
7. **candidate target structure selection & verification**
8. **Path forward and future work recommendation**

Out of Scope:

This research was not a general infrastructure audit, not a full product security assessment, and not a broad review of the client's security posture.

The following areas were outside the scope of this phase:

1. **production deployment**
2. **stealth or EDR bypass**
3. **persistence**
4. **post-exploitation payload development**
5. **cross-version reliability testing**

Full reliable kernel-mode code execution was also not completed during *this* phase. The work reached the point where useful primitives were validated and a plausible path toward controlled execution was mapped, but the final trigger still required additional exploit engineering.

Assumptions:

The research was conducted in a controlled lab environment where the driver could be loaded, tested, debugged, and crashed without affecting production systems.

The analysis also assumes that the attacker or operator has a way to interact with the vulnerable driver interface.

The findings are based on the tested environment and build. Kernel exploitability is heavily dependent on version, configuration and ROP gadget offsets. Therefore, results from one environment should not be treated as automatically universal across every Windows build or deployment configuration.

Limitations:

The main limitation of this phase was that full reliable kernel-mode code execution was not completed through the OOB bug alone. The research validated the vulnerable path, characterized the out-of-bounds kernel pool write, improved the kernel read primitive, and mapped a plausible route toward controlled execution. However, the final trigger still required additional engineering work around synchronization, target structure preservation, and crash-free execution.

The exploitability results were also dependent on the tested environment. Kernel pool behavior can vary based on Windows version, system activity, allocation timing, mitigation state, and target configuration.

Additional limitation in the context of BYOVD-type attacks is the requirement of High Integrity and the process being extremely noisy against security tools.

Intended Audience:

Originally, this document was meant to be read by the engineering team to verify the accuracy of the exploit, to assess the practicality for operational usage and to ensure the integration in the platform.

TECHNICAL DETAILS - Driver

The Vulnerability Explained

The vulnerable driver exposes an IOCTL-accessible code path that results in an out-of-bounds write into non-paged kernel pool memory. The write destination is derived from a fixed offset inside the driver-managed buffer, while the source pointer and copy size are influenced by attacker-controlled input. This creates a kernel pool corruption primitive that could potentially be developed into kernel-mode code execution.

The relevant logic is present inside the *IRP_MJ_DEVICE_CONTROL* dispatch path. The decompiled view of the function is shown below:

```
undefined4 IRP_MJ_DEVICE_CONTROL(undefined8 DevEXT,longlong IRP)

{
  undefined8 uVar1;
  ulonglong uVar2;
  undefined4 uVar3;
  uint local_res10 [6];
  char *IRP_0xb8;

  IRP_0xb8 = *(char **)(IRP + 0xb8);
  uVar3 = 0;
  local_res10[0] = 0;
  uVar2 = 0;
  if ((*IRP_0xb8 != '\0') && (uVar2 = 0, *IRP_0xb8 != '\x02')) {
    if ((*IRP_0xb8 == '\x0e') && (*(int *)(IRP_0xb8 + 0x18) == 0x[redacted]) {
      uVar1 = returns_string_by_handle((longlong)IRP_0xb8,IRP,local_res10);
      uVar3 = (undefined4)uVar1;
      uVar2 = (ulonglong)local_res10[0];
    }
    else {
      uVar3 = 0xc0000002;
    }
  }
  *(undefined4 *)(IRP + 0x30) = uVar3;
  *(ulonglong *)(IRP + 0x38) = uVar2;
  IofCompleteRequest(IRP,0);
  return uVar3;
}
```

Figure 1 – Showcasing the IRP_MJ_DEVICE_CONTROL Function of the vulnerable driver

The driver exposes a limited IOCTL interface, with the relevant request being routed into the *returns_string_by_handle* function. The underlying vulnerability is present inside this function, after the driver resolves the caller-supplied input into an internal buffer used by the rest of the operation.

From there, the driver begins using data derived from the request buffer as part of its copy logic. This is the point where the vulnerable behavior becomes reachable.

```
system_buffer = *(ulonglong **)(IRP + 0x18);
usermode_buffer = (short *)*system_buffer;
```

Figure 2 – Showcasing the system and usermode buffer and its relation with the IRP (I/O Request Packet)

The driver uses the normal IRP buffer (as shown as system_buffer) and uses an address present there to store another buffer (as shown as usermode_buffer). Since we can control the IRP + 0x18 pointer, we can control the entire content of the usermode_buffer.

```
if ((*(int *)(IRP_0xb8 + 0x10) == 8) && (*(int *)(IRP_0xb8 + 8) == 0x414)) {
  if (*usermode_buffer == 5) {
    if (**(short **)(usermode_buffer + 4) == 3) {
      *(undefined4 *)(system_buffer + 0x82) =
           *(undefined4 *)(*(short **)(usermode_buffer + 4) + 0x24);
      optimal_memset(system_buffer + 0x41,0,0x208);
      optimal_memset(system_buffer,0,0x208);
      memcpy(system_buffer + 0x41,*(void **)(usermode_buffer + 0x30),
             (ulonglong)(ushort)usermode_buffer[0x2c]);
```

Figure 3 – Showcasing the usermode_buffer buffer validation check

The next part of the code conducts some checks to ensure the usermode_buffer follows a certain pattern; however, since we control the entire usermode_buffer, we can easily forge the buffer such that it passes these checks.

It uses "*optimal_memset*" to clear out a section of the system_buffer.

The vulnerability arises here:

```
memcpy(system_buffer + 0x41,*(void **)(usermode_buffer + 0x30),
       (ulonglong)(ushort)usermode_buffer[0x2c]);
```

Figure 4 – Showcasing the vulnerable memcpy.

This system_buffer + 0x41 offset is the destination which lies at the **Non-Paged Kernel Pool**, both the source and the size are taken from the usermode_buffer **i.e. the buffer we have full control over.**

TECHNICAL DETAILS - Exploit Development

Available primitives and goals

As mentioned earlier, the vulnerable driver allows for an OOB Write in the kernel non-paged pool, additionally, the driver memcpy can be leveraged for an arbitrary kernel read primitive. The goal is to cause a CPU execution (i.e. control the RIP) with ring 0 privilege (i.e. CPL 0).

Strategy for KCE

At a high abstraction layer, the strategy is to somehow overwrite a critical function pointer of a nearby structure such that we can control where the execution will flow towards next. To achieve that – we will focus on conducting “pool grooming” to reliably put our vulnerable buffer before the carefully placed buffer to overwrite.

Reliable Kernel Read

Since the output of the `memcpy` eventually lands back in the driver’s `system_buffer`, and the source pointer is influenced by our controlled input, the bug can be shaped into a kernel read primitive. In practice, this means that if we can control the source pointer used by the copy, we can point it at kernel memory we want to inspect and have the copied data returned through the user-visible output buffer.

The first version of this primitive was not reliable enough to be useful for exploit development. During testing, the read succeeded in less than 2% of boots, and most attempts failed with `DeviceIoControl` returning an error. This also meant that a normal retry loop was not enough; the failure was not random noise that could be brute-forced away, but a blocker inside the driver’s execution path.

The next step was therefore to debug why the request was failing before the vulnerable path could complete. By tracing the driver flow and observing where execution stopped, we were able to narrow the issue down to the object-handling behavior that occurred before the copy completed. This became the first real exploitability hurdle: not proving that the primitive existed, but making the driver complete the relevant path consistently enough for the primitive to become useful.

```
memcpy(system_buffer + 0x41,*(void **)(usermode_buffer + 0x30),
       (ulonglong)(ushort)usermode_buffer[0x2c]);
str_name = (ushort *)ExAllocatePoolWithTag(1,0x210,0x64727672);
str_name[1] = 0x208;
*str_name = 0x104;
iVar1 = ObQueryNameString(*(undefined8 *)(usermode_buffer + 4),str_name,0x208,local_res20);
```

Figure 5 – Showcasing ObQueryNameString Call.

During debugging, we identified that the driver was calling `ObQueryNameString` on a value derived from `usermode_buffer + 4`. This mattered because that field was user-influenced, but the driver still expected it to behave like a valid kernel object path. The same buffer also had to satisfy the earlier validation requirement where the first DWORD was expected to contain 3.

In WinDbg, we confirmed that execution reached the vulnerable `memcpy` path, but once the flow entered `ObQueryNameString`, it failed to return cleanly back into the driver's code. This made the failure mode much clearer: the primitive itself was reachable, but the request was being killed by object-handling behavior before the driver could complete the IOCTL path reliably.

In other words, the main issue was not that the read primitive was impossible. The issue was that our input caused `ObQueryNameString` to operate on something it could not treat as a valid object. That API call became the reliability blocker that had to be solved before the kernel read primitive could become repeatable.

```
nt!ObQueryNameStringMode+0xa0:
fffff805`c5769450 48899c2498000000 mov     qword ptr [rsp+98h],rbx
0: kd> t
nt!ObQueryNameStringMode+0xa8:
fffff805`c5769458 488b81a0000000  mov     rax,qword ptr [rcx+0A0h]
2: kd> t

*BUSY*  Debuggee is running...
```

Figure 6 – winDBG assembly instruction

From the debugger output, we could see ObQueryNameString trying to interpret the supplied value as a valid object and read the related structure/header information from it. The failure made sense: we were passing a controlled value, but not something that the API could safely treat as a valid object. As a result, the driver would fail at this point instead of returning cleanly back into its own code path.

The strategy was therefore to stop fighting this API call and instead satisfy it. Rather than passing an invalid or partially controlled value, we needed to provide something that ObQueryNameString would accept as a valid object, while still satisfying the driver's earlier validation requirement where the relevant field needed to contain 3.

```
if (**(short **)(usermode_buffer + 4) == 3) {
```

Figure 7 – Usermode buffer offset value requirement

To solve this, we used a semaphore object created through CreateSemaphoreW, with the initial count set to 3. This gave us a valid object that the kernel API could handle properly, while also matching the value pattern expected by the driver's validation logic. Once this was in place, ObQueryNameString could return normally, allowing the driver to complete the IOCTL path instead of failing mid-execution.

This also changed the reliability profile of the primitive. If a specific semaphore instance did not produce the expected layout or value ordering, we could simply create another one and retry without rebooting the system. With this valid-object strategy and fail-safe retry loop in place, the kernel read primitive became effectively repeatable in the tested environment, moving from a less-than-2% per-boot success rate to approximately ~100% reliability.

```
Finding PsInitialSystemProcess...
PsInitialSystemProcess @ 0xfffff805c5dc5ab0
SYSTEM EPROCESS @ 0xffffe78d51699040

Finding EPROCESS offsets...
Scanning for EPROCESS offsets...
  UniqueProcessId @ 0x1d0 (value=4)
  ActiveProcessLinks @ 0x1d8
  Token @ 0x248

Walking process list...
Looking for our process (PID 1868)...
  Found our EPROCESS @ 0xffffe78d5a165080

Reading tokens...
SYSTEM Token:  0xffffd40c9222d04c
Our Token:     0xffffd40ca0dcb063
Our Token @:   0xffffe78d5a1652c8
```

Figure 8 – Showcasing Successful and reliable kernel read.

Choosing the structure

The choice of target structure was important because it would heavily influence the difficulty, reliability, and overall exploitability of the path moving forward. The ideal structure needed to be groomable, reasonably predictable in the kernel pool, and contain a field that could be abused for controlled execution if corrupted safely.

One possible candidate was the _DATA_QUEUE structure, which is commonly discussed in Windows kernel pool exploitation material because its allocation size can be influenced, making it attractive for grooming. However, on modern Windows 11 builds, this approach becomes more complicated because fake _DATA_QUEUE_ENTRY structures are subject to integrity checks. From my current understanding, this still looked like a viable strategy, but it also introduced additional validation hurdles that would need to be handled correctly.

Through further analysis in WinDbg, including pool inspection with !poolfind, we identified another promising target: the IRP structure. In the tested allocation pattern, the generated IRP was approximately 0x500 bytes in size including pool metadata, which made it relevant to the pool layout we were trying to shape. More importantly, the structure contained a CancelRoutine function pointer, giving us a control-flow-relevant field worth investigating.

To validate that this field was actually useful and not just interesting on paper, we manually modified the CancelRoutine pointer in the debugger and then triggered the associated

cleanup/cancellation path by killing the related process. This confirmed that execution could be redirected through that pointer under the right conditions.

After that, we tested whether these structures could be placed near the vulnerable allocation. By spraying IRP-like structures and then triggering the vulnerable DeviceIoControl path, we observed adjacency between the relevant allocations in the tested environment. This confirmed that pool grooming against this target was practical enough to continue exploring, and made the IRP structure a stronger candidate for the remaining exploitability work.

```
0: kd> dt nt!_IRP
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 AllocationProcessorNumber : Uint2B
   +0x006 Reserved1        : Uint2B
   +0x008 MdlAddress       : Ptr64 _MDL
   +0x010 Flags            : Uint4B
   +0x014 Reserved2        : Uint4B
   +0x018 AssociatedIrp    : <unnamed-tag>
   +0x020 ThreadListEntry  : _LIST_ENTRY
   +0x030 IoStatus         : _IO_STATUS_BLOCK
   +0x040 RequestorMode    : Char
   +0x041 PendingReturned  : UChar
   +0x042 StackCount       : Char
   +0x043 CurrentLocation  : Char
   +0x044 Cancel           : UChar
   +0x045 CancelIrql       : UChar
   +0x046 ApcEnvironment   : Char
   +0x047 AllocationFlags  : UChar
   +0x048 UserIosb         : Ptr64 _IO_STATUS_BLOCK
   +0x048 IoRingContext    : Ptr64 Void
   +0x050 UserEvent        : Ptr64 _KEVENT
   +0x058 Overlay          : <unnamed-tag>
   +0x068 CancelRoutine    : Ptr64     void
   +0x070 UserBuffer       : Ptr64 Void
   +0x078 Tail             : <unnamed-tag>
```

Figure 9 – Showcasing IRP structure and emphasis on CancelRoutine Func PTR.

Conducting the pool grooming

Since the target structure was the IRP, the minimum useful layout was [SystemBuffer][IRP], where the SystemBuffer allocation is created by the vulnerable driver during the IOCTL request. However, for the exploitation path we were exploring, the better layout was [IRP][SystemBuffer][IRP]. This gave us more room to verify the surrounding pool layout before attempting corruption, instead of blindly assuming that the next allocation was the target we wanted.

This verification step mattered because the goal was not just to cause a crash. The goal was to move toward a KCE path that could work under modern Windows constraints, where safe corruption and structure preservation are just as important as adjacency. Because of that, the grooming did not need to be perfect on every single attempt. It needed to produce the desired

layout often enough that we could detect the correct case, verify it with the kernel read primitive, and only then proceed.

The grooming strategy started by filling relevant kernel pool regions with named-pipe allocations, specifically NpFr-like structures. The purpose of this step was to occupy existing chunks in the target size range so that the next driver-controlled allocation had a better chance of landing in a useful region instead of being placed in already-noisy pool areas.

After that, we sprayed a large “wall” of IRP allocations. In the tested environment, these IRPs were generated through the CreateEvent path and were approximately 0x500 bytes in size with pool metadata included. The idea was to create a dense region of candidate target structures in memory.

The next step was to selectively free some of those IRPs to create holes inside the wall. When the vulnerable IOCTL was then triggered, the driver’s SystemBuffer allocation had a higher chance of landing inside one of those holes, producing adjacency between the controlled allocation and the IRP structures around it.

At a high level, the intended result was to move from a noisy kernel pool state into a layout where the vulnerable allocation was positioned between useful target structures. Once that layout appeared, the kernel read primitive could be used as a verification step before attempting any corruption.

```
1: kd>
Pool page ffff9088b050cd08 region is Nonpaged pool
 ffff9088b050c000 size:  5c0 previous size:    0  (Allocated)  FMic
 ffff9088b050c5d0 size:  500 previous size:    0  (Allocated)  Irp
*ffff9088b050cae0 size:  480 previous size:    0  (Allocated) *IoSB Process: ffff9088b05e1080
        Pooltag IoSB : Io system buffer, Binary : nt!io
 ffff9088b050cf60 size:   80 previous size:    0  (Free)       2...
```

Figure 10 – Showcasing adjacent IRP-IoSB structure.

To improve the odds of landing the driver-controlled SystemBuffer allocation into one of the prepared holes, we also used threaded triggering around the grooming stage. The idea was to fire the IOCTL requests shortly after the hole-generation step, once the pool layout had stabilized enough, but before the freed regions could be reused by unrelated kernel activity. This helped reduce the chance of a different allocation hijacking the hole before our target allocation could land there.

Another possible layout strategy was a sandwich-style grooming sequence. At a high level, this meant first creating a dense wall of IRPs, then freeing selected allocations to create holes, placing the driver-controlled SystemBuffer allocation into one of those holes, and finally placing another IRP after it.

The rough intended transition looked like this:

[IRP][IRP][IRP] → [IRP][Free][Free] → [IRP][SystemBuffer][Free] → [IRP][SystemBuffer][IRP]

This layout was useful because it gave us the desired vulnerable allocation in the middle, with candidate IRP structures around it. From there, the kernel read primitive could be used to verify that the surrounding structures were actually the expected ones before attempting any

out-of-bounds corruption. The goal was not blind pool corruption, but verified corruption against a layout that was known to be useful in the tested environment.

Verifying the structural integrity

The next crucial step was verification. At this point, we could not blindly assume that the grooming had worked, but to confirm that the vulnerable allocation had actually landed next to the intended target structure in kernel pool memory.

Since we had already developed a reliable kernel read primitive, we could use it as a layout-verification tool before attempting corruption. Instead of immediately writing out-of-bounds and risking a bugcheck, we could inspect the surrounding allocations and check whether the expected IRP structures were present.

The useful part here was that the current thread's ETHREAD structure maintains an IrpList, which gives us a way to walk through the IRPs associated with that thread. By using the kernel read primitive to traverse this list, we could identify the IRPs we had generated and then inspect nearby pool memory to see what structures were adjacent to them. This turned the exploitation path from a blind corruption attempt into a verified layout-dependent strategy.

```
0: kd> dt nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x4c0 CreateTime       : _LARGE_INTEGER
   +0x4c8 ExitTime         : _LARGE_INTEGER
   +0x4c8 KeyedWaitChain   : _LIST_ENTRY
   +0x4d8 PostBlockList    : _LIST_ENTRY
   +0x4d8 ForwardLinkShadow : Ptr64 Void
   +0x4e0 StartAddress     : Ptr64 Void
   +0x4e8 TerminationPort  : Ptr64 _TERMINATION_PORT
   +0x4e8 ReaperLink       : Ptr64 _ETHREAD
   +0x4e8 KeyedWaitValue   : Ptr64 Void
   +0x4f0 ActiveTimerListLock : Uint8B
   +0x4f8 ActiveTimerListHead : _LIST_ENTRY
   +0x508 Cid              : _CLIENT_ID
   +0x518 KeyedWaitSemaphore : _KSEMAPHORE
   +0x518 AlpcWaitSemaphore : _KSEMAPHORE
   +0x538 ClientSecurity   : _PS_CLIENT_SECURITY_CONTEXT
   +0x540 IrpList          : _LIST_ENTRY
   +0x550 TopLevelIrp      : Uint8B
```

Figure 11 – Showcasing adjacent IrpList offset in nt!_ETHREAD.

The IrpList points into the Tail.Overlay.ListEntry field of each individual IRP. That list entry contains the usual Flink and Blink pointers, which can be used to move across the IRPs associated with the current thread.

With the kernel read primitive in place, we could use this linked-list relationship to walk through the IRPs we had generated during grooming. Since the IRP allocation size was known from pool metadata in the tested environment, we could calculate where the neighboring allocation should begin and then read the adjacent pool tag to verify what structure was placed next to it.

This is also why the [IRP][SystemBuffer][IRP] layout was more useful than a simpler one-sided adjacency pattern. If the vulnerable SystemBuffer allocation was positioned between two IRPs, we had a safer and more reliable way to verify the surrounding layout before corruption. Reading forward from a known IRP into the next allocation was preferable to blindly reading behind an IRP, which could risk touching unmapped or unstable memory and causing a bugcheck.

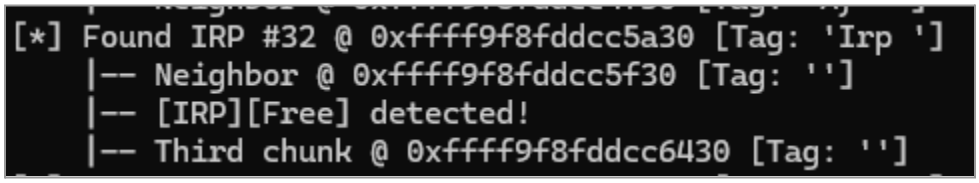

Figure 12 – Showcasing script detecting [IRP][Free] adjacent structures.

The next major hurdle was synchronization. The driver-controlled SystemBuffer allocation was short-lived, which meant we could not simply trigger DeviceIoControl, expect the allocation to appear in the desired pool location, and then take unlimited time to walk the IRP list and verify the neighboring structure.

This mattered because the exploitation path depended on safe, verified corruption rather than blind overwriting. Before using the out-of-bounds write, we needed to confirm that the adjacent allocation was actually the expected IRP-like structure, verify its pool metadata, and preserve the original fields required for stability, including the values needed to avoid immediate corruption-related bugchecks.

At the time of this research phase, this synchronization problem was the main remaining engineering blocker. The ideal solution would be to delay or pause the driver-controlled allocation long enough to inspect the pool layout, preserve the required target fields, update the controlled input, and then let the vulnerable path continue. Another possible direction was to slow the relevant path enough to create a usable race window where verification and corruption could happen safely.

In other words, the remaining challenge was no longer initial vulnerability discovery or primitive development. The vulnerable path, kernel read primitive, target structure, and grooming strategy had already been validated. The hard part was making the final corruption step safe and repeatable against a volatile allocation lifetime.

This issue was later overcome during follow-up exploit engineering, but the specific synchronization solution is outside the scope of this public case study.

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffff9f8fddc2e430, memory referenced.
Arg2: 0000000000000000, X64: bit 0 set if the fault was due to a not-present PTE.
    bit 1 is set if the fault was due to a write, clear if a read.
    bit 3 is set if the processor decided the fault was due to a corrupted PTE.
    bit 4 is set if the fault was due to attempted execute of a no-execute PTE.
    - ARM64: bit 1 is set if the fault was due to a write, clear if a read.
    bit 3 is set if the fault was due to attempted execute of a no-execute PTE.
Arg3: fffff806419d14d0, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 0000000000000002, (reserved)

Debugging Details:
------------------
```

Figure 13 – 0x50 PageFault error illustration

One possible direction was to slow the vulnerable path down enough to create a usable race window. In that model, the exploit could verify the neighboring IRP structures, confirm the expected pool layout, and then use a TOCTOU-style update or another controlled timing technique to modify the input before the final vulnerable copy occurred. The goal was not simply to win a race, but to win it safely enough to avoid corrupting the wrong structure and causing a bugcheck.

As a hypothesis, we also tested whether the source memory could be backed by a disk-based .bin mapping and then evicted to force a page fault during the driver's read path. This part worked: the access produced the expected page-fault behavior. The next idea was to combine this with an oplock so that the driver would be forced to wait on user-mode interaction before the read completed.

In practice, that specific oplock approach did not give us the pause we wanted. The oplock was effectively ignored in this path, and the operation still completed successfully, with the return state indicating that the read itself had succeeded. This made the experiment useful, even though it did not solve the synchronization problem: it showed that the memory-access behavior could be influenced, but that this particular method was not enough to create a controllable delay window.

Remaining Exploitability Work

At this stage, the required pool layout had been observed in the tested environment: a vulnerable allocation positioned between two IRP-like structures. This layout made the next exploitation step clearer: use the out-of-bounds write to modify the adjacent target structure while preserving enough of its original state to avoid immediate system instability.

```
   +0x058 Overlay          : <unnamed-tag>
   +0x068 CancelRoutine    : 0xfffff803`c6c256d0     void  nt!DbgPrint+0
   +0x070 UserBuffer       : 0x00000182`73b60000 Void
   +0x078 Tail             : <unnamed-tag>
0: kd> bp fffff803`c6c256d0
breakpoint 0 redefined
0: kd> g
Breakpoint 0 hit
nt!DbgPrint:
fffff803`c6c256d0 488bc4          mov     rax,rsp
```

Figure 14 – Showcasing manual CancelRoutine ptr change trigger (kernel control flow hijacking)

The intended objective was to modify only the control-flow-relevant field while keeping the rest of the structure intact. In this case, the target field was the routine pointer that could be reached through a normal cleanup or cancellation path. If performed safely, this would provide a path toward controlled execution in kernel context.

The remaining work was focused on reliability and safe triggering rather than initial vulnerability discovery. The key engineering tasks were:

- delay or synchronize the vulnerable driver path long enough to inspect the target layout
- identify and verify the desired adjacent structure before corruption
- preserve the original target structure fields required for stability
- apply the minimal required modification through the out-of-bounds write
- trigger the relevant execution path in a controlled way
- return or recover cleanly without causing a bugcheck
- repeat the process across multiple runs to measure reliability

References

1. vp777, **Windows Non-Paged Pool Overflow Exploitation**, GitHub repository.
   This research documents techniques for exploiting Windows non-paged pool overflows, including named-pipe based pool shaping and primitive development.
2. rootkit, **Windows Kernel Exploitation Tutorial Part 4: Pool Feng-Shui → Pool Overflow**, rootkits.xyz, November 2017.
   Used as background material for Windows kernel pool overflow exploitation and pool feng-shui concepts.
3. Connor McGarr, **Exploit Development: Swimming In The Kernel Pool — Leveraging Pool Vulnerabilities From Low-Integrity Exploits, Part 1**, Connor McGarr's Blog.
   Used as supporting background for kernel pool exploitation strategy and low-integrity pool vulnerability research.