# One I/O Ring to Rule Them All: A Full Read/Write Exploit Primitive on Windows 11

windows-internals.com/one-i-o-ring-to-rule-them-all-a-full-read-write-exploit-primitive-on-windows-11

By Yarden Shafir

This blog post will cover the post-exploitation technique I presented at TyphoonCon 2022. For anyone interested in the talk itself, I'll link the recording here when it becomes available. This technique is a post exploitation primitive unique to Windows `11` `22H2+` – there are no `0` -days here. Instead, there's a method to turn an arbitrary write, or even arbitrary increment bug in the Windows kernel into a full read/write of kernel memory.

## Background

Kernel exploitation (and exploitation in general) on Windows is becoming harder with every new version. Driver Signature Enforcement made it harder for an attacker to load unsigned drivers, and later `HVCI` made it entirely impossible – with the added difficulty of a driver block list, preventing attackers from loading signed vulnerable drivers. `SMEP` and `KCFG` mitigate against code redirection through function pointer overwrites, and `KCET` makes `ROP` impossible as well. Other `VBS` features such as `KDP` protect kernel data, so common targets such as `g_CiOptions` can no longer be modified by an attacker. And on top of those, there are Patch Guard and Secure Kernel Patch Guard which validate the integrity of the kernel and many of its components.

With all the existing mitigations, just finding a user->kernel bug no longer guarantees successful exploitation. In Windows `11` with all mitigations enabled, it's nearly impossible to achieve Ring `0` code execution. However, data-based attacks are still a viable solution

A known technique for a data-only attack is to create a fake kernel-mode structure in user mode, then tricking the kernel to use it through a write-what-where bug (or any other bug type that can achieve that). The kernel will treat this structure like valid kernel data, allowing the attacker to achieve privilege escalation by manipulating the data in the structure, thus manipulating kernel actions that are done based on that data. There are numerous examples for this technique, which was used in different ways. For example, this blog post by Jooru demonstrates using a fake token table to turn an off-by-one bug into an arbitrary write, and later using that to run shellcode in ring `0` . Many other examples take advantage of different Win32k objects to achieve arbitrary read, write or both. Some of these techniques have already been mitigated by Microsoft, other are already known and hunted for by security products, and others are still usable and most likely used in the wild.

In this post I'd like to add one more technique to the pile – using I/O ring preregistered buffers to create a read/write primitive, using 1-2 arbitrary kernel writes (or increments). This technique uses a new object type that currently has very limited visibility to security products and is likely to be ignored for a while. The method is very simple to use – once you understand the underlying mechanism of I/O ring.

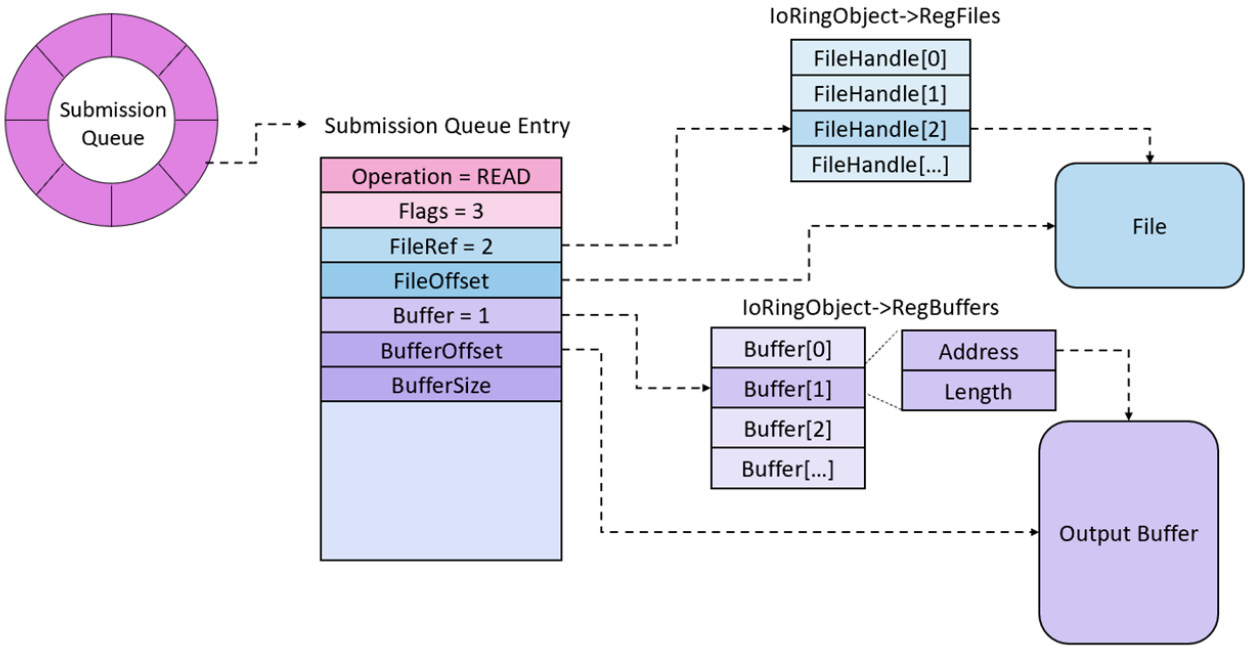
## I/O Ring

I already wrote several blog posts (and a talk) about I/O rings so I'll just present the basic idea and the parts relevant to this technique. Anyone interested in learning more about it can read the previous posts on the topic or watch the talk from P99 Conf.

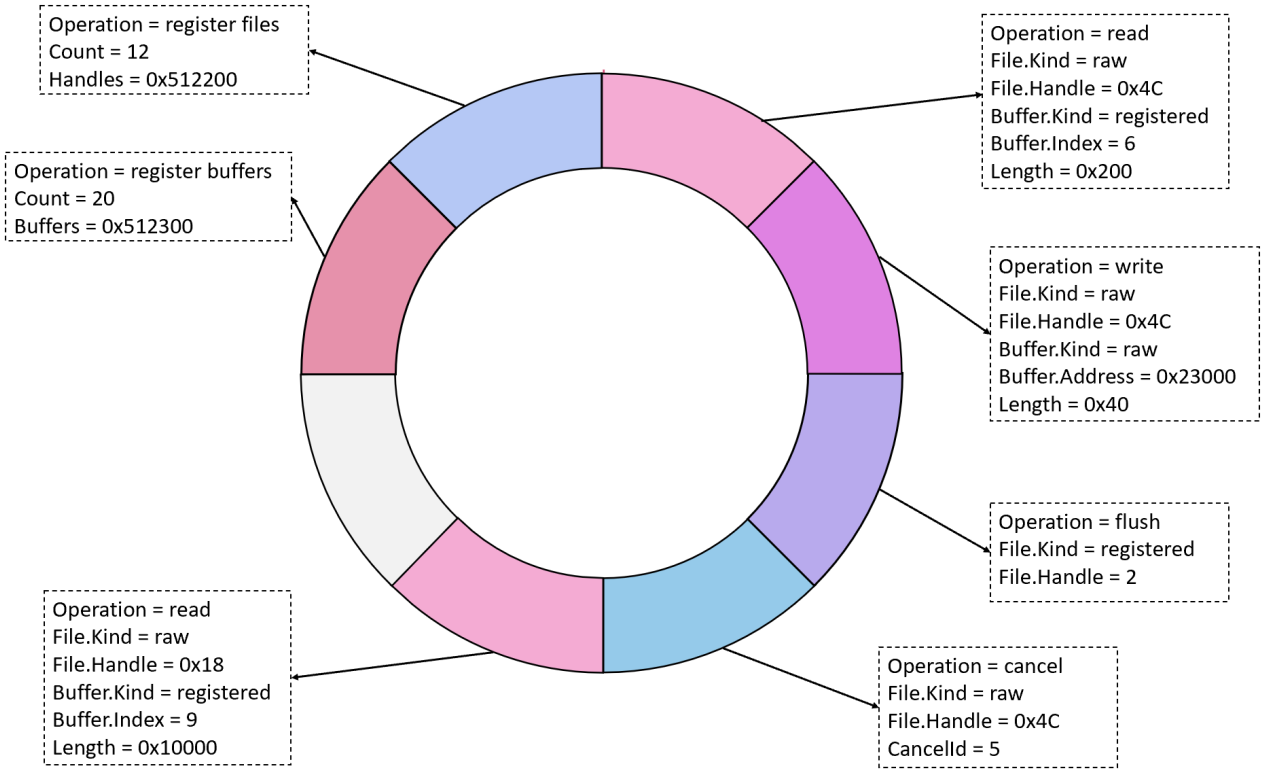
In short, I/O ring is a new asynchronous I/O mechanism that allows an application to queue as many as `0x10000` I/O operations and submit them all at once, using a single `API` call. The mechanism was modeled after the Linux `io_uring`, so the design of the two is very similar. For now, I/O rings don't support every possible I/O operation yet. The available operations in Windows `11 22H2` are read, write, flush and cancel. The requested operations are written into a Submission Queue, and then submitted all together. The kernel processes the requests and writes the status codes into a Completion Queue – both queues are in a shared memory region accessible to both user mode and kernel mode, allowing sharing of data without the overhead of multiple system calls.

In addition to the available I/O operations, the application can queue two more types of operations unique to I/O ring: preregister buffers and preregister files. These options allow an application to open all the file handles or create all the input/output buffers ahead of time, register them and later reference them by index in I/O operations queued through the I/O ring. When the kernel processes an entry that uses a preregistered file handle or buffer, it fetches the requested handle/buffer from the preregistered array and passes it on to the I/O manager where it is handled normally.

For the visual learners, here's an example of a queue entry using a preregistered file handle and buffer:




A submission queue that's ready to be submitted to the kernel could look something like this:




The exploitation technique discussed here takes advantage of the preregistered buffers array, so let's go into a bit more detail there:

## Registered Buffers

As I mentioned, one of the operations an application can do is allocate all the buffers for its future I/O operations, then register them with the I/O ring. The preregistered buffers are referenced through the I/O ring object:

```
typedef struct _IORING_OBJECT
{
    USHORT Type;
    USHORT Size;
    NT_IORING_INFO UserInfo;
    PVOID Section;
    PNT_IORING_SUBMISSION_QUEUE SubmissionQueue;
    PMDL CompletionQueueMdl;
    PNT_IORING_COMPLETION_QUEUE CompletionQueue;
    ULONG64 ViewSize;
    ULONG InSubmit;
    ULONG64 CompletionLock;
    ULONG64 SubmitCount;
    ULONG64 CompletionCount;
    ULONG64 CompletionWaitUntil;
    KEVENT CompletionEvent;
    UCHAR SignalCompletionEvent;
    PKEVENT CompletionUserEvent;
    ULONG RegBuffersCount;
    PVOID RegBuffers;
    ULONG RegFilesCount;
    PVOID* RegFiles;
} IORING_OBJECT, *PIORING_OBJECT;
```

When the request gets processed, the following things happen:

1. `IoRing->RegBuffers` and `IoRing->RegBuffersCount` get set to zero.
2. The kernel validates that `Sqe->RegisterBuffers.Buffers` and `Sqe->RegisterBuffers.Count` are both not zero.
3. If the request came from user mode, the array is probed to validate that it's fully in the user mode address space. Array size can be up to `sizeof(ULONG)` .
4. If the ring previously had a preregistered buffers array and the size of the new buffer is the same as the size of the old buffer, the old buffer array is placed back in the ring and the new buffer is ignored.
5. If the previous checks pass and the new buffer array is to be used, a new paged pool allocation is made – this will be used to copy the data from the user mode array and will be pointed to by `IoRing->RegBuffers` .
6. If there's previously been a registered buffers array pointed to by the I/O ring, it gets copied into the new kernel array. Any new buffers will be added in the same allocation, after the old buffers.

7. Every entry in the array sent from user mode is probed to validate that the requested buffer is fully in user mode, then gets copied to the kernel array.
8. The old kernel array (if one existed) is freed, and the operation is completed.

This whole process is safe – the data is only read from user mode once, probed and validated correctly to avoid overflows and accidental reads or writes of kernel addresses. Any future use of these buffers will fetch them from the kernel buffer.

But what if we already have an arbitrary kernel write bug?

In that case, we can overwrite a single pointer – `IoRing->RegBuffers`, to point it to a fake buffer that is fully under our control. We can populate it with kernel mode addresses and use those as buffers in I/O operations. When the buffers are referenced by index they don't get probed – the kernel assumes that if the buffers were safe when they where registered, then copied to a kernel allocation, they would still be safe when they're referenced as part of an operation.

This means that with a single arbitrary write and a fake buffer array we can get full control of the kernel address space through read and write operations.

## The Primitive

Once `IoRing->RegBuffers` points to the fake, user controlled array, we can use normal I/O ring operations to generate kernel reads and writes into whichever addresses we want by specifying an index into our fake array to use as a buffer:

1. Read operation + kernel address: The kernel will "read" from a file of our choice into the specified kernel address, leading to arbitrary write.
2. Write operation + kernel address: The kernel will "write" the data in the specified address into a file of our choice, leading to arbitrary read.

Initially my primitive relied on files to read and write to, but Alex suggested the use of named pipes instead which is way cooler and a lot less visible, leaving no traces on disk. So, the rest of the post + the exploit code will be using named pipes.

As you can see, technique itself is pretty simple. So simple, in fact, it doesn't even require the use of any (well, almost) undocumented `API`s or secret data structures. It uses Win32 `API` and structures that are available in the public symbols of `ntoskrnl.exe`. The exploit primitive involves the following steps:

1. Create two named pipes with `CreateNamedPipe` : one will be used for input for arbitrary kernel writes and the other for output for arbitrary kernel reads. At least the pipe that'll be used as input should be created with flag `PIPE_ACCESS_DUPLEX` to allow both reading and writing. I chose to create both with `PIPE_ACCESS_DUPLEX` for convenience.
2. Open client handles for both pipes with `CreateFile` , both with read and write permissions.
3. Create an I/O ring: this can be done through `CreateIoRing` API.
4. Allocate a fake buffers array in the heap: Starting from the official `22H2` release, the registered buffers array is no longer a flat array, but an array of `IOP_MC_BUFFER_ENTRY` structures, so this gets slightly more tricky.
5. Find the address of the newly created I/O ring object: since I/O rings use a new object type, `IORING_OBJECT` , we can leak its address through a well-known `KASLR` bypass technique. `NtQuerySystemInformation` with `SystemHandleInformation` leaks the kernel addresses of objects, including our new I/O ring object. Fortunately, the internal structure of `IORING_OBJECT` is in the public symbols so there's no need to reverse engineer the structure to find the offset of `RegBuffers` . We add the two together to get the target for our arbitrary write.
   Unfortunately, this `API` as well as many other `KASLR` bypasses can only be used by processes with Medium IL or higher, so Low IL processes, sandboxed processes and browsers can't use it and will have to find a different method.
6. Use your preferred arbitrary write bug to overwrite `IoRing->RegBuffers` with the address of the fake user-mode array. Notice that if you haven't previously registered a valid buffers array you'll also have to overwrite `IoRing->RegBuffersCount` to have a non-zero value.
7. Populate the fake buffers array with kernel pointers to read or write to: to do this you might need other `KASLR` bypasses in order to find your target addresses. You could use `NtQuerySystemInformation` with `SystemModuleInformation` class to find the base addresses of kernel modules, use the same technique as earlier to find kernel addresses of objects, or use the pointers available inside the I/O ring itself, which point to data structures in the paged pool.
8. Queue read and write operations in the I/O ring through `BuildIoRingReadFile` and `BuildIoRingWriteFile` .

With this method, arbitrary reads and writes aren't limited to a pointer size, like many other methods, but can be as large as `sizeof(ULONG)` , reading or writing many pages of kernel data simultaneously.

## Cleanup

This technique requires minimal cleanup: all that's required it to set `IoRing->RegBuffers` to zero before closing the handle to the I/O ring object. As long as the pointer is zero, the kernel won't try to free anything even if `IoRing->RegBuffersCount` is non-zero.

Cleanup gets slightly more complicated if you choose to first register a valid buffer array and then overwrite the existing pointer in the I/O ring object – in that case there is already an allocated kernel buffer, which also adds a reference count in the `EPROCESS` object. In that case, the `EPROCESS RefCount` will need to be decremented before the process exits to avoid leaving a stale process around. Luckily that is easy to do with one more arbitrary read + write using our existing technique.

## Arbitrary Increment

A couple years ago I published a series of blogs discussing `CVE-2020-1034` – an arbitrary increment vulnerability in `EtwpNotifyGuid` . Back then, I focused on the challenges of exploiting this bug and used it to increment the process' token privileges – a very well known privilege escalation technique. This method works, though it's possible to detect in real time or retroactively using different tools. Security vendors are well aware of this technique and many already detect it.

That project made me interested in other ways to exploit that specific bug class – an arbitrary increment of a kernel address, so I was very happy to find a post exploitation technique that finally fit. With the method I presented here, you can use an arbitrary increment to increment `IoRing->RegBuffers` from `0` to a user-mode address such as `0x1000000` (no need for `0x1000000` increments, just increment the `3` rd byte by one) and increment `IoRing->RegBuffersCount` from `0` to `1` or `0x100` (or more). This does require you to trigger the bug twice in order to create the technique, but I recommend doing that anyway to avoid the extra cleanup required when overwriting an existing pointer.

## Forensics and Detection

This post exploitation technique has very little visibility and leaves few forensic traces: I/O rings have nearly no visibility through `ETW` except on creation, and the technique leaves no forensic traces in memory. The only part of this technique that is visible to security products are the named pipes operations, visible to security products who use a filesystem filter driver (and most do). However, these pipes are local and aren't used for anything that looks too suspicious — they read and write small amounts of data with no specific format, so they're not likely to be flagged as suspicious

## Portable Features = Portable Exploits?

I/O rings on Windows were modeled after the Linux `io_uring` and share many of the same features, and this one is no different. The Linux `io_uring` also allows registering buffers or file handles, and the registered buffers are handled very similarly and stored in the `user_bufs` field of the ring. This means that the same exploitation technique should also work on Linux (though I haven't personally tested it).

The main difference between the two systems in this case is mitigation: while on Windows it's difficult to mitigate against this technique, Linux has a mitigation that makes blocking this technique (at least in its current form) trivial: `SMAP`. This mitigation prevents access to user-mode addresses with kernel-mode privileges, blocking any exploitation technique that involves faking a kernel structure in user-mode. Unfortunately due to the basic design of the Windows system it's unlikely `SMAP` will ever be a usable mitigation there, but it's been available and used on Linux since `2012`.

Of course there are still ways to bypass `SMAP`, such as shaping a kernel pool allocation to be used as the fake buffers array instead of a user-mode address or editing the PTE of the user-mode page that contains the fake array, but the basic exploitation primitive won't work on systems that support `SMAP`.

## 22H2 Changes

The official `22H2` release introduced a change that affects this technique, but only slightly. Since Windows `11` build `22610` (so a couple of builds before the official `22H2` release) the buffer array in the kernel is no longer a flat array of addresses and lengths, but instead an array of pointers to a new data structure: `IOP_MC_BUFFER_ENTRY`:

```
typedef struct _IOP_MC_BUFFER_ENTRY
{
    USHORT Type;
    USHORT Reserved;
    ULONG Size;
    ULONG ReferenceCount;
    ULONG Flags;
    LIST_ENTRY GlobalDataLink;
    PVOID Address;
    ULONG Length;
    CHAR AccessMode;
    ULONG MdlRef;
    PMDL Mdl;
    KEVENT MdlRundownEvent;
    PULONG64 PfnArray;
    IOP_MC_BE_PAGE_NODE PageNodes[1];
} IOP_MC_BUFFER_ENTRY, *PIOP_MC_BUFFER_ENTRY;
```

This data structure is used as part of the `MDL` cache capability that was added in the same build. It looks complex and scary, but in our use-case most of these fields are never used and can be ignored. We still have the same `Address` and `Length` fields that we need for our technique to work, and to be compatible with the requirements of the new feature we also need to hardcode a few values in the fields `Type`, `Size`, `AccessMode` and `ReferenceCount`.

To adapt our technique to this new addition, here are the changes needed in our code:

1. Allocate a fake buffers array, sized `sizeof(PVOID) * NumberOfEntries`.
2. Allocate a `IOP_MC_BUFFER_ENTRY` structure for each fake buffer and place the pointer into the fake buffers array. Zero out the structure, then set the following fields:

```
mcBufferEntry->Address = TargetAddress;
mcBufferEntry->Length = Length;
mcBufferEntry->Type = 0xc02;
mcBufferEntry->Size = 0x80; // 0x20 * (numberOfPagesInBuffer + 3)
mcBufferEntry->AccessMode = 1;
mcBufferEntry->ReferenceCount = 1;
```

## The PoC

I uploaded my PoC here. It works starting `22H2` preview builds (minimal supported version – before this build I/O rings didn't yet support write operations) and up to the latest Windows Preview build (`25415` as of today). For my arbitrary write/increment bugs I used the HEVD driver, recompiled to support arbitrary increments. The PoC supports both options, but if you use the latest HEVD release only the arbitrary write option will work.

For the arbitrary read target, I used a page from the `ntoskrnl.exe` data section – the offset of the section is hardcoded due to laziness, so it might break spontaneously when that offset changes.