One Year to I/O Ring: What Changed?

windows-internals.com/one-year-to-i-o-ring-what-changed

By Yarden Shafir

It's been just over a year since the first version of I/O ring was introduced into Windows. The initial version was introduced in Windows `21H2` and I did my best to document it here, with a comparison to the Linux io_uring here. Microsoft also documented the Win32 functions. Since that initial version this feature progressed and received pretty significant changes and updates, so it deserves a follow-up post documenting all of them and explaining them in more detail.

New Supported Operations

Looking at the changes, the first and most obvious thing we can see is that two new operations are now supported – write and flush:

```
switch ( Sqe->OpCode )
{
  case IORING_OP_NOP:
    ioStatusBlock.Status = 0;
    goto LABEL_22;
  case IORING_OP_READ:
    return IopIoRingDispatchRead(IoRingObject, Sqe);
  case IORING_OP_REGISTER_FILES:
    return IopIoRingDispatchRegisterFiles(IoRingObject, Sqe);
  case IORING_OP_REGISTER_BUFFERS:
    return IopIoRingDispatchRegisterBuffers(IoRingObject, Sqe);
  case IORING_OP_CANCEL:
    return IopIoRingDispatchCancel(IoRingObject, Sqe);
  case IORING_OP_WRITE:
    return IopIoRingDispatchWrite(IoRingObject, Sqe);
}
ioringObj = IoRingObject;
if ( Sqe->OpCode == IORING_OP_FLUSH )
  return IopIoRingDispatchFlush(IoRingObject, Sqe);
ioStatusBlock.Status = STATUS_NOT_IMPLEMENTED;
```

These allow using the I/O ring to perform write and flush operations. These new operations are processed and handled similarly to the read operation that's been supported since the first version of I/O rings and forwarded to the appropriate I/O functions. New wrapper functions were added to KernelBase.dll to queue requests for these operations: `BuildIoRingWriteFile` and `BuildIoRingFlushFile`, and their definitions can be found in the `ioringapi.h` header file (available in the preview SDK):

```
STDAPI
BuildIoRingWriteFile (
    _In_ HIORING ioRing,
```

```
    IORING_HANDLE_REF fileRef,
    IORING_BUFFER_REF bufferRef,
    UINT32 numberOfBytesToWrite,
    UINT64 fileOffset,
    FILE_WRITE_FLAGS writeFlags,
    UINT_PTR userData,
    IORING_SQE_FLAGS sqeFlags
);

STDAPI
BuildIoRingFlushFile (
    _In_ HIORING ioRing,
    IORING_HANDLE_REF fileRef,
    FILE_FLUSH_MODE flushMode,
    UINT_PTR userData,
    IORING_SQE_FLAGS sqeFlags
);
```

Similarly to `BuildIoRingReadFile`, both of these build the submission queue entry with the requested `OpCode` and add it to the submission queue. Obviously, there are different flags and options needed for the new operations, such as the `flushMode` for flush operations or `writeFlags` for writes. To handle that, the `NT_IORING_SQE` structure now contains a union for the input data that gets interpreted according to the requested OpCode – the new structure is available in the public symbols and also at the end of this post.

One small kernel change that was added to support write operations can be seen in `IopIoRingReferenceFileObject`:

```
NTSTATUS __fastcall IopIoRingReferenceFileObject(
        _IORING_OBJECT *IoRing,
        ULONG64 FileIndex,
        BYTE PreregisteredFileHandle,
        KPROCESSOR_MODE PreviousMode,
        BYTE WriteOperation,
        ACCESS_MASK DesiredAccess,
        _FILE_OBJECT *FileObject,
        _OBJECT_HANDLE_INFORMATION *HandleInformation)
{
  void *handle; // r10

  handle = FileIndex;
  if ( PreregisteredFileHandle )
  {
    if ( FileIndex >= IoRing->RegFilesCount )
      return STATUS_INDEX_OUT_OF_BOUNDS;
    handle = IoRing->RegFiles[FileIndex];
  }
  if ( WriteOperation )
    return ObReferenceFileObjectForWrite(handle, PreviousMode, FileObject, HandleInformation);
  else
    return IopReferenceFileObject(handle, DesiredAccess, PreviousMode, FileObject, HandleInformation);
}
```

There are a few new arguments and an additional call to `ObReferenceFileObjectForWrite`. Probing of different buffers across the various functions also changed depending on the operations type.

User Completion Event

Another interesting change that was introduced as well is the ability to register a user event to be notified for every new completed operation. Unlike the I/O Ring's `CompletionEvent`, that only gets signaled when all operations are complete, the new optional user event will be signaled for every newly completed operation, allowing the application to process the results as they are being written to the completion queue.

To support this new functionality, another system call was created: `NtSetInformationIoRing`:

```
NTSTATUS
NtSetInformationIoRing (
    HANDLE IoRingHandle,
    ULONG IoRingInformationClass,
    ULONG InformationLength,
    PVOID Information
);
```

Like other `NtSetInformation*` routines, this function receives a handle to the `IoRing` object, an information class, length and data. Only one information class is currently valid: `1`. The `IORING_INFORMATION_CLASS` structure is unfortunately not in the public symbols so we can't know what it's official name is, but I'll call it `IoRingRegisterUserCompletionEventClass`. Even though only one class is currently supported, there might be other information classes supported in the future. One interesting thing here is that the function uses a global array `IopIoRingSetOperationLength` to retrieve the expected information length for each information class:

```
if ( IoRingInformationClass >= 2 )
  goto InvalidInfoClass;
lengthForClass = IopIoRingSetOperationLength[IoRingInformationClass];
if ( !lengthForClass )
  goto InvalidInfoClass;
if ( InformationLength < lengthForClass )
{
  status = STATUS_INFO_LENGTH_MISMATCH;
  goto Exit;
}
```

The array currently only has two entries: `0`, which isn't actually a valid class and returns a length of `0`, and entry `1` which returns an expected size of `8`. This length matches the function's expectation to receive an event handle (`HANDLE`s are `8` bytes on `x64`). This

could be a hint that more information classes are planned in the future, or just a different coding choice.

After the necessary input checks, the function references the I/O ring whose handle was sent to the function. Then, if the information class is `IoRingRegisterUserCompletionEventClass`, calls `IopIoRingUpdateCompletionUserEvent` with the supplied event handle. `IopIoRingUpdateCompletionUserEvent` will reference the event and place the pointer in `IoRingObject->CompletionUserEvent`. If no event handle is supplied, the `CompletionUserEvent` field is cleared:

```
completionUserEvent = 0i64;
eventHandle = *EventHandle;
if ( !*EventHandle
  || (eventObj = 0i64,
      result = ObReferenceObjectByHandle(eventHandle, 2u, ExEventObjectType, PreviousMode, &eventObj, 0i64),
      completionUserEvent = eventObj,
      result >= 0) )
{
  v_oldIrql = KeAcquireSpinLockRaiseToDpc(&IoRingObject->CompletionLock);
  oldCompletionEvent = IoRingObject->CompletionUserEvent;
  oldIrql = v_oldIrql;
  IoRingObject->CompletionUserEvent = completionUserEvent;
  KxReleaseSpinLock(&IoRingObject->CompletionLock);
  if ( KiIrqlFlags )
  {
    CurrentIrql = KeGetCurrentIrql();
    if ( (KiIrqlFlags & 1) != 0 && CurrentIrql <= 0xFu && oldIrql <= 0xFu && CurrentIrql >= 2u )
    {
      CurrentPrcb = KeGetCurrentPrcb();
      SchedulerAssist = CurrentPrcb->SchedulerAssist;
      v13 = ~(-1i64 << (oldIrql + 1));
      v14 = (v13 & SchedulerAssist[5]) == 0;
      SchedulerAssist[5] &= v13;
      if ( v14 )
        KiRemoveSystemWorkPriorityKick(CurrentPrcb);
    }
  }
  __writecr8(oldIrql);
  if ( oldCompletionEvent )
    ObfDereferenceObjectWithTag(oldCompletionEvent, 0x746C6644u);
  return STATUS_SUCCESS;
}
return result;
```

The RE Corner

On a side note, this function might look rather large and mildly threatening, but most of it is simply synchronization code to guarantee that only one thread can edit the `CompletionUserEvent` field of the I/O ring at any point and prevent race conditions. And in fact, the compiler makes the function look larger than it actually is since it unpacks macros, so if we try to reconstruct the source code this function would look much cleaner:

```
NTSTATUS
IopIoRingUpdateCompletionUserEvent (
    PIORING_OBJECT IoRingObject,
    PHANDLE EventHandle,
    KPROCESSOR_MODE PreviousMode
```

```
    )
{
    PKEVENT completionUserEvent;
    HANDLE eventHandle;
    NTSTATUS status;
    PKEVENT oldCompletionEvent;
    PKEVENT eventObj;

    completionUserEvent = 0;
    eventHandle = *EventHandle;
    if (!eventHandle ||
        (eventObj = 0,
        status = ObReferenceObjectByHandle(
                 eventHandle, PAGE_READONLY, ExEventObjectType,
PreviousMode, &eventObj, 0),
        completionUserEvent = eventObj,
        !NT_SUCCESS(status))
    {
        KeAcquireSpinLockRaiseToDpc(&IoRingObject->CompletionLock);
        oldCompletionEvent = IoRingObject->CompletionUserEvent;
        IoRingObject->CompletionUserEvent = completionUserEvent;
        KeReleaseSpinLock(&IoRingObject->CompletionLock);
        if (oldCompletionEvent)
        {
            ObDereferenceObjectWithTag(oldCompletionEvent, 'tlfD');
        }
        return STATUS_SUCCESS;
    }
    return status;
}
```

That’s it, around six lines of actual code. But, that is not the point of this post, so let’s get back to the topic at hand: the new `CompletionUserEvent` .

Back to the User Completion Event

The next time we run into `CompletionUserEvent` is when an `IoRing` entry is completed, in `IopCompleteIoRingEntry` :

```
CompletionQueue = IoRingObject->CompletionQueue;
CompletionQueueRingMask = IoRingObject->UserInfo.CompletionQueueRingMask;
CompletionUserEvent = 0i64;
Head = CompletionQueue->Head;
oldIrql = KeAcquireSpinLockRaiseToDpc(&IoRingObject->CompletionLock);
Tail = CompletionQueue->Tail;
v_oldIrql = oldIrql;
newTail = Tail + 1;
lastIndex = CompletionQueueRingMask & Tail;
if ( newTail - Head <= IoRingObject->UserInfo.CompletionQueueSize )
{
  index = lastIndex;
  CompletionQueue->Entries[index].UserData = UserData;
  CompletionQueue->Entries[index].IoStatus = *IoStatusBlock;
  CompletionQueue->Tail = newTail;
  if ( IoRingObject->CompletionUserEvent )
  {
    _InterlockedOr(v22, 0);
    if ( (newTail - CompletionQueue->Head) % IoRingObject->UserInfo.CompletionQueueSize == 1 )
    {
      CompletionUserEvent = IoRingObject->CompletionUserEvent;
      ObfReferenceObjectWithTag(CompletionUserEvent, 'tlfD');
    }
  }
}
shouldSetEvent = 0;
++IoRingObject->CompletionCount;
if ( IoRingObject->SignalCompletionEvent && IoRingObject->CompletionCount == IoRingObject->CompletionWaitUntil )
{
  IoRingObject->SignalCompletionEvent = 0;
  shouldSetEvent = 1;
}
KxReleaseSpinLock(&IoRingObject->CompletionLock);
if ( shouldSetEvent )
  KeSetEvent(&IoRingObject->CompletionEvent, priority, 0);
if ( CompletionUserEvent )
  KeSetEvent(CompletionUserEvent, priority, 0);
```

While the normal I/O ring completion event is only signaled once all operations are complete, the `CompletionUserEvent` is signaled under different conditions. Looking at the code, we see the following check:

```
if ( (newTail - CompletionQueue->Head) % IoRingObject->UserInfo.CompletionQueueSize == 1 )
{
  CompletionUserEvent = IoRingObject->CompletionUserEvent;
  ObfReferenceObjectWithTag(CompletionUserEvent, 'tlfD');
}
```

Every time an I/O ring operation is complete and written into the completion queue, the `CompletionQueue->Tail` field gets incremented by one (referenced here as `newTail`). The `CompletionQueue->Head` field contains the index of the last completion entry that was written, and gets incremented every time the application processes another entry (If you use `PopIoRingCompletion` it'll do that internally, otherwise you need to increment it yourself). So, `(newTail - Head) % CompletionQueueSize` calculates the number of completed entries that have not yet been processed by the application. If that amount is one, that means that the application has processed all completed entries except the latest one, that is being completed now. In that case, the function will reference the `CompletionUserEvent` and then call `KeSetEvent` to signal it.

This behavior allows the application to follow along with the completion of all its submitted operations by creating a thread whise purpose is to wait on the user event and process every newly completed entry just as it's completed. This makes sure that the `Head` and `Tail` of the completion queue are always the same, so the next entry to be completed will signal the event, the entry will process the entry, and so on. This way the main thread of the application can keep doing other work, but the I/O operations all get processed as soon as possible by the worker thread.

Of course, this is not mandatory. An application might choose to not register a user event and simply wait for the completion of all events. But the two events allow different applications to choose the option that works best for them, creating an I/O completion mechanism that can be adjusted to suit different needs.

There is a function in `KernelBase.dll` to register the user completion event: `SetIoRingCompletionEvent`. We can find its signature in `ioringapi.h`:

```
STDAPI
SetIoRingCompletionEvent (
    _In_ HIORING ioRing,
    _In_ HANDLE hEvent
);
```

Using this new API and knowing how this new event operates, we can build a demo application that would look something like this:

```
HANDLE g_event;

DWORD
WaitOnEvent (
    LPVOID lpThreadParameter
    )
{
    HRESULT result;
    IORING_CQE cqe;

    WaitForSingleObject(g_event, INFINITE);
    while (TRUE)
    {
        //
        // lpThreadParameter is the handle to the ioring
        //
        result = PopIoRingCompletion((HIORING)lpThreadParameter, &cqe);
        if (result == S_OK)
        {
            /* do things */
```

```
        }
        else
        {
            WaitForSingleObject(g_event, INFINITE);
            ResetEvent(g_event);
        }
    }
    return 0;
}

int
main ()
{
    HRESULT result;
    HIORING ioring = NULL;
    IORING_CREATE_FLAGS flags;

    flags.Required = IORING_CREATE_REQUIRED_FLAGS_NONE;
    flags.Advisory = IORING_CREATE_ADVISORY_FLAGS_NONE;
    result = CreateIoRing(IORING_VERSION_3, flags, 0x10000, 0x20000,
&ioring);

    /* Queue operations to ioring... */

    //
    // Create user completion event, register it to the ioring
    // and create a thread to wait on it and process completed operations.
    // The ioring handle is sent as an argument to the thread.
    //
    g_event = CreateEvent(NULL, FALSE, FALSE, NULL);
    result = SetIoRingCompletionEvent(handle, g_event);
    thread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)WaitOnEvent,
ioring, 0, &threadId);
    result = SubmitIoRing(handle, 0, 0, &submittedEntries);

    /* Clean up... */

    return 0;
}
```

Drain Preceding Operations

The user completion event is a very cool addition, but it's not the only waiting-related improvement to I/O rings. Another one can be found by looking at the `NT_IORING_SQE_FLAGS` enum:

```
typedef enum _NT_IORING_SQE_FLAGS
{
    NT_IORING_SQE_FLAG_NONE = 0x0,
    NT_IORING_SQE_FLAG_DRAIN_PRECEDING_OPS = 0x1,
} NT_IORING_SQE_FLAGS, *PNT_IORING_SQE_FLAGS;
```

Looking through the code, we can find a check for `NT_IORING_SQE_FLAG_DRAIN_PRECEDING_OPS` right in the beginning of `IopProcessIoRingEntry`:

```
if ( (Sqe->Flags & NT_IORING_SQE_FLAG_DRAIN_PRECEDING_OPS) != 0 )
{
  // Setup a wait on all the entries that have been queued and submitted but have not been completed yet.
  // Passing -1 in EntryCount makes the function calculate how many entries have been submitted so far
  // and setup the correct CompletionWaitUntil value.
  // Then it'll wait until all entries have been completed before processing this entry.
  completionWaitSet = 0;
  IopIoRingSetupCompletionWait(IoRingObject, 0, 0xFFFFFFFF, 0, &completionWaitSet);
  if ( completionWaitSet )
  {
    IopIoRingWaitForCompletionEvent(IoRingObject, 0, 0, 0i64);
    *PrecedingOpsDrained = 1;
  }
}
```

This check happens before any processing is done, to check if the submission queue entry contains the `NT_IORING_SQE_FLAG_DRAIN_PRECEDING_OPS` flag. If so, `IopIoRingSetupCompletionWait` is called to setup the wait parameters. The function signature looks something like this:

```
NTSTATUS
IopIoRingSetupCompletionWait (
    _In_ PIORING_OBJECT IoRingObject,
    _In_ ULONG SubmittedEntries,
    _In_ ULONG WaitOperations,
    _In_ BOOL SetupCompletionWait,
    _Out_ PBYTE CompletionWait
);
```

Inside the function there are a lot of checks and calculations that are both very technical and very boring, so I'll spare myself the need to explain them and you the need to read through the exhausting explanation and skip to the good parts. Essentially, if the function receives `-1` as the `WaitOperations`, it will ignore the `SetupCompletionWait` argument and calculate the number of operations that have already been submitted and processed but not yet completed. That number gets placed in `IoRingObject->CompletionWaitUntil`. It also sets `IoRingObject->SignalCompletionEvent` to `TRUE` and returns `TRUE` in the output argument `CompletionWait`.

If the function succeeded, `IopProcessIoRingEntry` will then call `IopIoRingWaitForCompletionEvent`, which will until `IoRingObject->CompletionEvent` is signaled. Now is the time to go back to the check we've seen earlier in

`IopCompleteIoRingEntry`:

```
if ( IoRingObject->SignalCompletionEvent && IoRingObject->CompletionCount == IoRingObject->CompletionWaitUntil )
{
  IoRingObject->SignalCompletionEvent = 0;
  shouldSetEvent = 1;
}
```

If `SignalCompletionEvent` is set (which it is, because `IopIoRingSetupCompletionWait` set it) and the number of completed events is equal to `IoRingObject->CompletionWaitUntil`, `IoRingObject->CompletionEvent` will get signaled to mark that the pending events are all completed. `SignalCompletionEvent` also gets cleared to avoid signaling the event again when it's not requested.

When called from `IopProcessIoRingEntry`, `IopIoRingWaitForCompletionEvent` receives a timeout of `NULL`, meaning that it'll wait indefinitely. This is something that should be taken under consideration when using the `NT_IORING_SQE_FLAG_DRAIN_PRECEDING_OPS` flag.

So to recap, setting the `NT_IORING_SQE_FLAG_DRAIN_PRECEDING_OPS` flag in a submission queue entry will make sure all preceding operations are completed before this entry gets processed. This might be needed in certain cases where one I/O operation relies on an earlier one.

But waiting on pending operations happens in one more case: When submitting an I/O ring. In my first post about I/O rings last year, I defined the `NtSubmitIoRing` signature like this:

```
NTSTATUS
NtSubmitIoRing (
    _In_ HANDLE Handle,
    _In_ IORING_CREATE_REQUIRED_FLAGS Flags,
    _In_ ULONG EntryCount,
    _In_ PLARGE_INTEGER Timeout
    );
```

My definition ended up not being entirely accurate. The more correct name for the third argument would be `WaitOperations`, so the accurate signature is:

```
NTSTATUS
NtSubmitIoRing (
    _In_ HANDLE Handle,
    _In_ IORING_CREATE_REQUIRED_FLAGS Flags,
    _In_opt_ ULONG WaitOperations,
    _In_opt_ PLARGE_INTEGER Timeout
    );
```

Why does this matter? Because the number you pass into `WaitOperations` isn't used to process the ring entries (they are processed entirely based on `SubmissionQueue->Head` and `SubmissionQueue->Tail`), but to request the number of operations to wait on. So, if `WaitOperations` is not `0` , `NtSubmitIoRing` will call `IopIoRingSetUpCompletionWait` before doing any processing:

```
if ( waitOperations )
{
  status = IopIoRingSetupCompletionWait(ioringObj, submittedEntries, waitOperations, FALSE, &completionWaitSet);
  if ( status < 0 )
    goto Exit;
}
```

However, it calls the function with `SetupCompletionWait=FALSE` , so the function won't actually setup any of the wait parameters, but only perform the sanity checks to see if the number of wait operations is valid. For example, the number of wait operations can't be higher than the number of operations that were submitted. If the checks fail, `NtSubmitIoRing` won't process any of the entries and will return an error, usually `STATUS_INVALID_PARAMETER_3` .

Later, we see both functions again after operations have been processed:

```
precedingOpsDrained = 0;
if ( Head != submissionQueueTail )
{
  tail = submissionQueueTail;
  mask = SubmissionQueueRingMask;
  do
  {
    ioringSqe = SubmissionQueue->Entries[(Head++ & mask)];
    SubmissionQueue->Head = Head;
    IopProcessIoRingEntry(ioringObject, &ioringSqe, &precedingOpsDrained);
  }
  while ( Head != tail );
  waitOperations = WaitOperations;
}
if ( completionWaitSet )
{
  if ( !precedingOpsDrained
    || (status = IopIoRingSetupCompletionWait(ioringObject, 0, waitOperations, TRUE, &completionWaitSet), status >= 0) )
  {
    if ( completionWaitSet )
      status = IopIoRingWaitForCompletionEvent(ioringObject, previousMode, Flags & TRUE, Timeout);
  }
}
```

`IopIoRingSetupCompletionWait` is called again to recalculate the number of operations that need to be waited on, taking into consideration any operations that might have already been completed (or waited on already if any of the `SQE` s had the flag mentioned earlier). Then `IopIoRingWaitForCompletionEvent` is called to wait on `IoRingObject->CompletionEvent` until all requested events have been completed.
In most cases applications will choose to either send `0` as the `WaitOperations` argument or set it to the total number of submitted operations, but there may be cases where an application could want to only wait on part of the submitted operations, so it can choose a lower number to wait on.

Looking at Bugs

Comparing the same piece of code in different builds is a fun way of finding bugs that were fixed. Sometimes these are security vulnerabilities that got patched, sometimes just regular old bugs that can affect the stability or reliability of the code. The I/O ring code in the kernel received a lot of modifications over the past year, so this seems like a good chance to go hunting for old bugs.

One bug that I'd like to focus on here is pretty easy to spot and understand, but is a fun example for the way different parts of the system that seem entirely unrelated can clash in unexpected ways. This is a functional (not security) bug that prevented `WoW64` processes from using some of the I/O ring features.

We can find evidence of this bug when looking at `IopIoRingDispatchRegisterBuffers` and `IopIoRingispatchRegisterFiles`. When looking at the new build we can see a piece of code that wasn't there in earlier versions:

```
isWow64 = FALSE;
if ( Process->WoW64Process )
{
  Machine = Process->Machine;
  if ( Machine == IMAGE_FILE_MACHINE_I386 || Machine == IMAGE_FILE_MACHINE_ARMNT )
    isWow64 = TRUE;
}
```

This is checking whether the process that is registering the buffers or files is a `WoW64` process – a `32`-bit process running on top of a `64`-bit system. Since Windows now supports `ARM64`, this `WoW64` process can now be either a `x86` application or an `ARM32` one.

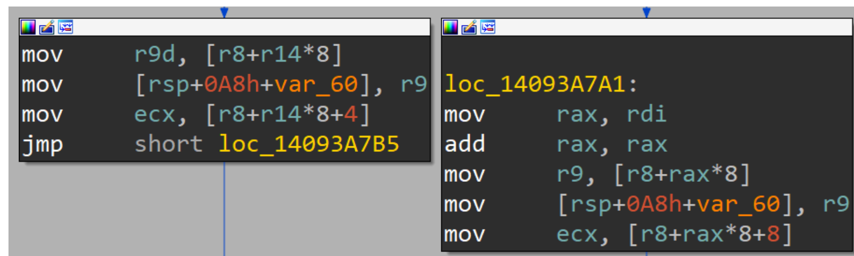
Looking further ahead can show us why this information matters here. Later on, we see two cases where `isWow64` is checked:

This first case is when the array size is being calculated to check for invalid sizes if caller is `UserMode`.

```
size = 0xFFFFFFFFi64;
if ( isWow64 )
  arraySize = 8 * bufCount;
else
  arraySize = 16 * bufCount;
if ( arraySize <= 0xFFFFFFFF )
  size = arraySize;
```

```
if ( isWow64 )
{
  address = *(&bufferInfo->Address + 2 * index);
  length = *(&bufferInfo->Address + 2 * index + 1);
}
else
{
  address = bufferInfo[index].Address;
  length = bufferInfo[index].Length;
}
```

This second case happens when iterating over the input buffer to register the buffers in the array that will be stored in the I/O ring object. In this case it's slightly harder to understand what we're looking at because of the way the structures are handled here, but if we look at the disassembly it might become a bit clearer:



The block on the left is the `WoW64` case and the block on the right is the native case. Here we can see the difference in the offset that is being accessed in the `bufferInfo` variable (`r8` in the disassembly). To get some context, `bufferInfo` is read from the submission queue entry:

```
bufferInfo = Sqe->RegisterBuffers.Buffers;
```

When registering a buffer, the `SQE` will contain a `NT_IORING_OP_REGISTER_BUFFERS` structure:

```
typedef struct _NT_IORING_OP_REGISTER_BUFFERS
{
    /* 0x0000 */ NT_IORING_OP_FLAGS CommonOpFlags;
    /* 0x0004 */ NT_IORING_REG_BUFFERS_FLAGS Flags;
    /* 0x000c */ ULONG Count;
    /* 0x0010 */ PIORING_BUFFER_INFO Buffers;
} NT_IORING_OP_REGISTER_BUFFERS, *PNT_IORING_OP_REGISTER_BUFFERS;
```

The sub-structures are all in the public symbols so I won't put them all here, but the one to focus on in this case is `IORING_BUFFER_INFO` :

```
typedef struct _IORING_BUFFER_INFO
{
    /* 0x0000 */ PVOID Address;
```

```
    /* 0x0008 */ ULONG Length;
} IORING_BUFFER_INFO, *PIORING_BUFFER_INFO; /* size: 0x0010 */
```

This structure contains an address and a length. The address is of type `PVOID`, and this is where the bug lies. A `PVOID` doesn't have a fixed size across all systems. It is a pointer, and therefore its size depends on the size of a pointer on the system. On `64`-bit systems that's `8` bytes, and on `32`-bit systems that's `4` bytes. However, `WoW64` processes aren't fully aware that they are running on a `64`-bit system. There is a whole mechanism put in place to emulate a `32`-bit system for the process to allow `32`-bit applications to execute normally on `64`-bit hardware. That means that when the application calls `BuildIoRingRegisterBuffers` to create the array of buffers, it calls the `32`-bit version of the function, which uses `32`-bit structures and `32`-bit types. So instead of using an `8`-byte pointer, it'll use a `4`-byte pointer, creating an `IORING_BUFFER_INFO` structure that looks like this:

```
typedef struct _IORING_BUFFER_INFO
{
    /* 0x0000 */ PVOID Address;
    /* 0x0004 */ ULONG Length;
} IORING_BUFFER_INFO, *PIORING_BUFFER_INFO; /* size: 0x008 */
```

This is, of course, not the only case where the kernel receives pointer-sized arguments from a user-mode caller and there is a mechanism meant to handle these cases. Since the kernel doesn't support `32`-bit execution, the `WoW64` emulation later is in charge of translating system call input arguments from the `32`-bit sizes and types to the `64`-bit types expected by the kernel. However in this case, the buffer array is not sent as an input argument to a system call. It is written into the shared section of the I/O ring that is read directly by the kernel, never going through the `WoW64` translation DLLs. This means no argument translation is done on the array, and the kernel directly reads an array that was meant for a `32`-bit kernel, where the `Length` argument is not at the expected offset. In the early versions of I/O ring this meant that the kernel always skipped the buffer length and interpreted the next entry's address as the last entry's length, leading to bugs and errors.

In newer builds, the kernel is aware of the differently shaped structure used by `WoW64` processes, and interprets it correctly: It assumes that the size of each entry is `8` bytes instead of `0x10`, and reads only the first `4` bytes as the address and the next `4` bytes as the length.

The same issue existed when pre-registering file handles, since a `HANDLE` is also the size of a pointer. `IopIoRingDispatchRegisterFiles` now has the same checks and processing to allow `WoW64` processes to successfully register file handles as well.

Other Changes

There are a couple of smaller changes that aren't large or significant enough to receive their own section of this post but still deserve an honorable mention:

- The successful creation of a new I/O ring object will generate an ETW event containing all the initialization information in the I/O ring.
- `IoringObject->CompletionEvent` received a promotion from `NotificationEvent` type to `SynchronizationEvent`.
- Current I/O ring version is `3`, so new rings created for recent builds should use this version.
- Since different versions of I/O ring support different capabilities and operations, KernelBase.dll exports a new function: `IsIoRingOpSupported`. It receives the `HIORING` handle and the operation number, and returns a boolean indicating whether the operation is supported on this version.

Data Structures

One more exciting thing happened in Windows `11` `22H2` (build `22577`): nearly all the internal I/O ring structures are available in the public symbols! This means there is no longer a need to painfully reverse engineer the structures and try to guess the field names and their purposes. Some of the structures received major changes since `21H2`, so not having to reverse engineer them all over again is great.

Since the structures are in the symbols there is no real need to add them here. However, structures from the public symbols aren't always easy to find through a simple Google search – I highly recommend trying GitHub search instead, or just directly using ntdiff. At some point people will inevitably search for some of these data structures, find the REd structures in my old post, which are no longer accurate, and complain that they are out of date. To avoid that at least temporarily, I'll only post here the updated versions of the structures that I had in the old post but will highly encourage you to get the up-to-date structures from the symbols – the ones here are bound to change soon enough *(edit: one build later, some of them already did)*. So, here are some of the structures from Windows `11` build `22598`:

```
typedef struct _NT_IORING_INFO
{
    IORING_VERSION IoRingVersion;
    NT_IORING_CREATE_FLAGS Flags;
    ULONG SubmissionQueueSize;
    ULONG SubmissionQueueRingMask;
    ULONG CompletionQueueSize;
    ULONG CompletionQueueRingMask;
    PNT_IORING_SUBMISSION_QUEUE SubmissionQueue;
    PNT_IORING_COMPLETION_QUEUE CompletionQueue;
} NT_IORING_INFO, *PNT_IORING_INFO;
```

```
typedef struct _NT_IORING_SUBMISSION_QUEUE
{
    ULONG Head;
    ULONG Tail;
    NT_IORING_SQ_FLAGS Flags;
    NT_IORING_SQE Entries[1];
} NT_IORING_SUBMISSION_QUEUE, *PNT_IORING_SUBMISSION_QUEUE;

typedef struct _NT_IORING_SQE
{
    enum IORING_OP_CODE OpCode;
    enum NT_IORING_SQE_FLAGS Flags;
    union
    {
        ULONG64 UserData;
        ULONG64 PaddingUserDataForWow;
    };
    union
    {
        NT_IORING_OP_READ Read;
        NT_IORING_OP_REGISTER_FILES RegisterFiles;
        NT_IORING_OP_REGISTER_BUFFERS RegisterBuffers;
        NT_IORING_OP_CANCEL Cancel;
        NT_IORING_OP_WRITE Write;
        NT_IORING_OP_FLUSH Flush;
        NT_IORING_OP_RESERVED ReservedMaxSizePadding;
    };
} NT_IORING_SQE, *PNT_IORING_SQE;

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;
    BYTE InSubmit;
    ULONG64 CompletionLock;
    ULONG64 SubmitCount;
    ULONG64 CompletionCount;
    ULONG64 CompletionWaitUntil;
    KEVENT CompletionEvent;
    BYTE SignalCompletionEvent;
    PKEVENT CompletionUserEvent;
```

```
    ULONG RegBuffersCount;
    PIORING_BUFFER_INFO RegBuffers;
    ULONG RegFilesCount;
    PVOID* RegFiles;
} IORING_OBJECT, *PIORING_OBJECT;
```

One structure that isn't in the symbols is the `HIORING` structure that represents the `ioring` handle in `KernelBase`. That one slightly changed since `21H2` so here is the reverse engineered `22H2` version:

```
typedef struct _HIORING
{
    HANDLE handle;
    NT_IORING_INFO Info;
    ULONG IoRingKernelAcceptedVersion;
    PVOID RegBufferArray;
    ULONG BufferArraySize;
    PVOID FileHandleArray;
    ULONG FileHandlesCount;
    ULONG SubQueueHead;
    ULONG SubQueueTail;
} HIORING, *PHIORING;
```

Conclusion

This feature barely just shipped a few months ago, but it's already receiving some very interesting additions and improvements, aiming to make it more attractive to I/O-heavy applications. It's already at version `3`, and it's likely we'll see a few more versions coming in the near future, possibly supporting new operation types or extended functionality. Still, no applications seem to use this mechanism yet, at least on Desktop systems.

This is one of the more interesting additions to Windows `11`, and just like any new piece of code it still has some bugs, like the one I showed in this post. It's worth keeping an eye on I/O rings to see how they get used (or maybe abused?) as Windows `11` becomes more widely adapted and applications begin using all the new capabilities it offers.