I/O Rings – When One I/O Operation is Not Enough

windows-internals.com/i-o-rings-when-one-i-o-operation-is-not-enough

By Yarden Shafir

Introduction

I usually write about security features or techniques on Windows. But today's blog is not directly related to any security topics, other than the usual added risk that any new system call introduces. However, it's an interesting addition to the I/O world in Windows that could be useful for developers and I thought it would be interesting to look into and write about. All this is to say – if you're looking for a new exploit or EDR bypass technique, you should save yourselves the time and look at the other posts on this website instead.

For the three of you who are still reading, let's talk about I/O rings!

I/O ring is a new feature on Windows preview builds. This is the Windows implementation of a ring buffer – a circular buffer, in this case used to queue multiple I/O operations simultaneously, to allow user-mode applications performing a lot of I/O operations to do so in one action instead of transitioning from user to kernel and back for every individual request.

This new feature adds a lot of new functions and internal data structures, so to avoid constantly breaking the flow of the blog with new data structures I will not put them as part of the post, but their definitions exist in the code sample at the end. I will only show a few internal data structures that aren't used in the code sample.

I/O Ring Usage

The current implementation of I/O rings only supports read operations and allows queuing up to `0x10000` operations at a time. For every operation the caller will need to supply a handle to the target file, an output buffer, an offset into the file and amount of memory to be read. This is all done in multiple new data structures that will be discussed later. But first, the caller needs to initialize its I/O ring.

Create and Initialize an I/O Ring

To do that, the system supplies a new system call – `NtCreateIoRing`. This function creates an instance of a new `IoRing` object type, described here as `IORING_OBJECT`:

```
typedef struct _IORING_OBJECT
{
  USHORT Type;
  USHORT Size;
```

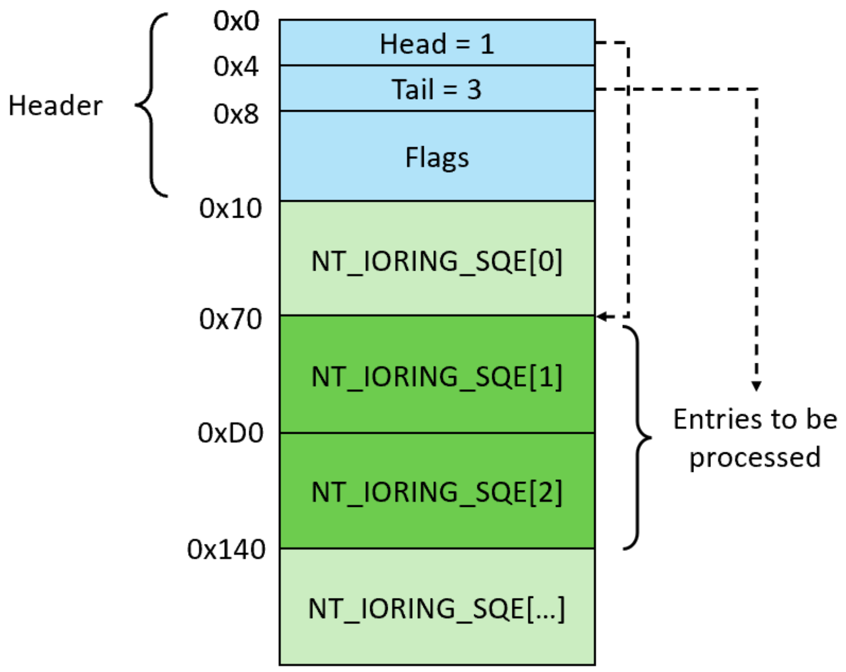
```
    NT_IORING_INFO Info;
    PSECTION SectionObject;
    PVOID KernelMappedBase;
    PMDL Mdl;
    PVOID MdlMappedBase;
    ULONG_PTR ViewSize;
    ULONG SubmitInProgress;
    PVOID IoRingEntryLock;
    PVOID EntriesCompleted;
    PVOID EntriesSubmitted;
    KEVENT RingEvent;
    PVOID EntriesPending;
    ULONG BuffersRegistered;
    PIORING_BUFFER_INFO BufferArray;
    ULONG FilesRegistered;
    PHANDLE FileHandleArray;
} IORING_OBJECT, *PIORING_OBJECT;
```

`NtCreateIoRing` receives one new structure as an input argument – `IO_RING_STRUCTV1`. This structure contains information about current version, which currently can only be `1`, required and advisory flags (both don't currently support any values other than `0`) and the requested size for the submission queue and completion queue.

The function receives this information and does the following things:

1. Validates all the input and output arguments – their addresses, size alignment, etc.
2. Checks the requested submission queue size and calculate the amount of memory needed for the submission queue based on the requested number of entries.
    1. If `SubmissionQueueSize` is over `0x10000` a new error status `STATUS_IORING_SUBMISSION_QUEUE_TOO_BIG` gets returned.
3. Checks the completions queue size and calculates the amount of memory needed for it.
    1. The completion queue is limited to `0x20000` entries and error code `STATUS_IORING_COMPLETION_QUEUE_TOO_BIG` is returned if a larger number is requested.
4. Creates a new object of type `IoRingObjectType` and initializes all fields that can be initialized at this point – flags, submission queue size and mask, event, etc.
5. Creates a section for the queues, maps it in system space and creates an `MDL` to back it. Then maps the same section in user-space. This section will contain the submission space and completion space and will be used by the application to communicate the parameters for all requested I/O operations with the kernel and receive the status codes.
6. Initializes the output structure with the submission queue address and other data to be returned to the caller.

After `NtCreateIoRing` returns successfully, the caller can write its data into the supplied submission queue. The queue will have a queue head, followed by an array of `NT_IORING_SQE` structures, each representing one requested I/O operation. The header describes which entries should be processed at this time:



The queue header describes which entries should be processed using the `Head` and `Tail` fields. `Head` specifies the index of the last unprocessed entry, and `Tail` specifies the index to stop processing at. `Tail - Head` has to be lower that total number of entries, as well as equal to or highrt than the number of entries that will be requested in the call to `NtSubmitIoRing`.

Each queue entry contains data about the requested operation: file handle, file offset, output buffer base, offset and amount of data to be read. It also contains an `OpCode` field to specify the requested operation.

I/O Ring Operation Codes

There are four possible operation types that can be requested by the caller:

1. `IORING_OP_READ` : requests that the system reads data from a file into an output buffer. The file handle will be read from the `FileRef` field in the submission queue entry. This will either be interpreted as a file handle or as an index into a pre-registered array of file handles, depending on whether the `IORING_SQE_PREREGISTERED_FILE` flag (`1`) is set in the queue entry `Flags` field. The output will be written into an output buffer supplied in the `Buffer` field of the entry. Similar to `FileRef` , this field can instead contain an index into a pre-registered array of output buffers if the `IORING_SQE_PREREGISTERED_BUFFER` flag (`2`) is set.
2. `IORING_OP_REGISTERED_FILES` : requests pre-registration of file handles to be processed later. In this case the `Buffer` field of the queue entry points to an array of file handles. The requested file handles will get duplicated and placed in a new array, in the `FileHandleArray` field of the queue entry. The `FilesRegistered` field will contain the number of file handles.
3. `IORING_OP_REGISTERED_BUFFERS` : requests pre-registration of output buffers for file data to be read into. In this case, the `Buffer` field in the entry should contain an array of `IORING_BUFFER_INFO` structures, describing addresses and sizes of buffers into which file data will be read:
   ```
   typedef struct _IORING_BUFFER_INFO
   {
       PVOID Address;
       ULONG Length;
   } IORING_BUFFER_INFO, *PIORING_BUFFER_INFO;
   ```

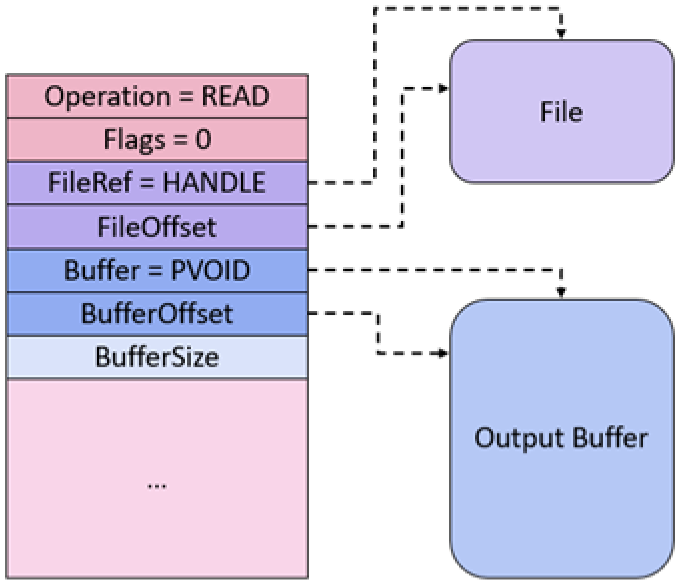
   The buffers' addresses and sizes will be copied into a new array and placed in the `BufferArray` field of the submission queue. The `BuffersRegistered` field will contain the number of buffers.

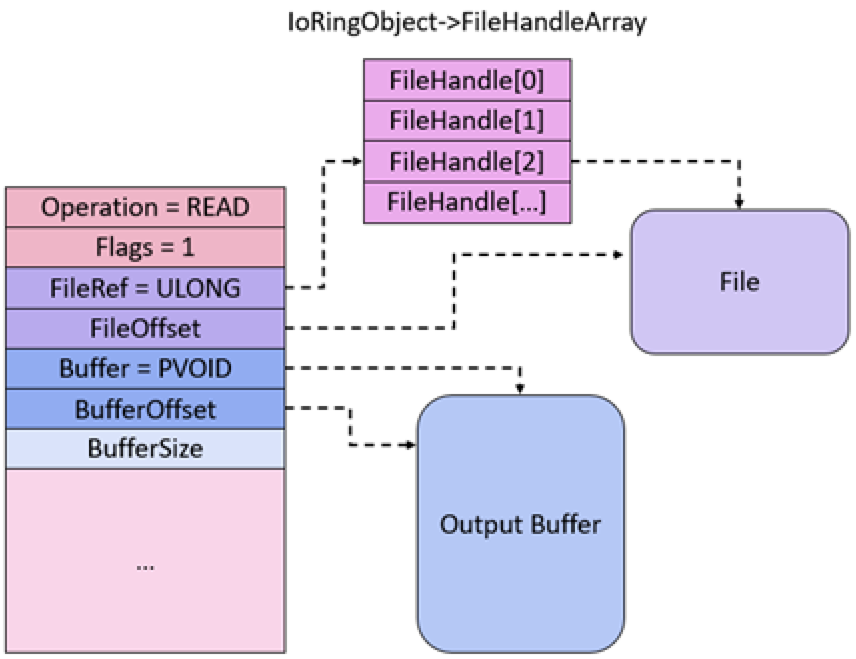
4. `IORING_OP_CANCEL` : requests the cancellation of a pending operation for a file. Just like the in `IORING_OP_READ` , the `FileRef` can be a handle or an index into the file handle array depending on the flags. In this case the Buffer field points to the `IO_STATUS_BLOCK` to be canceled for the file.

All these options can be a bit confusing so here are illustrations for the 4 different reading scenarios, based on the requested flags:

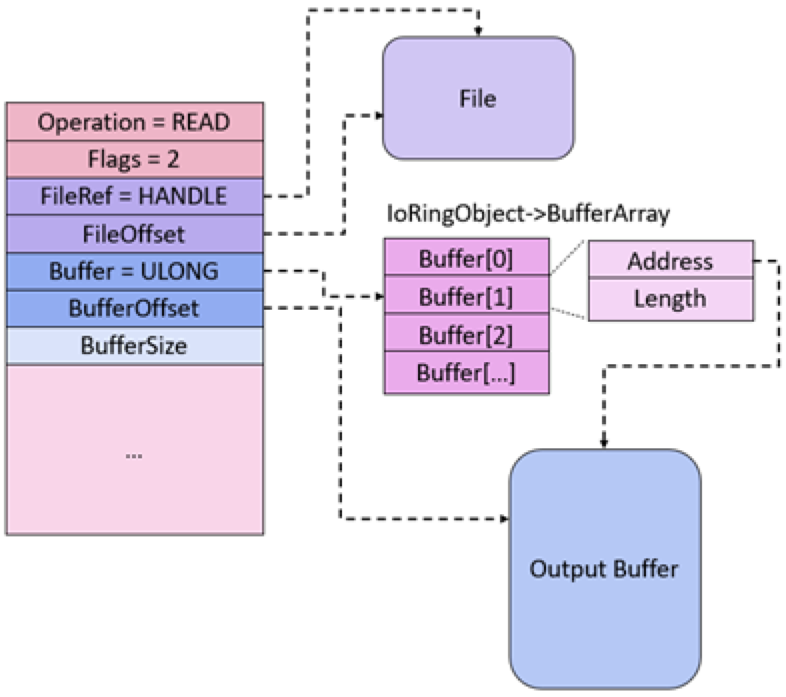
Flags are `0` , using the `FileRef` field as a file handle and the `Buffer` field as a pointer to the output buffer:

Flag `IORING_SQE_PREREGISTERED_FILE` (`1`) is requested, so `FileRef` is treated as an index into an array of pre-registered file handles and `Buffer` is a pointer to the output buffer:

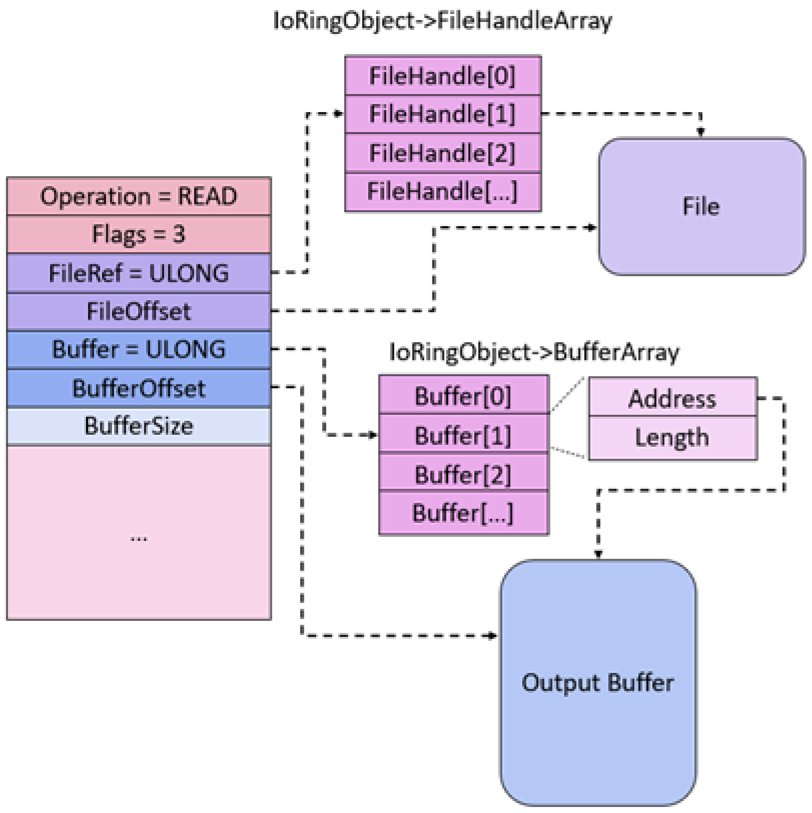




Flag `IORING_SQE_PREREGISTERED_BUFFER` (`2`) is requested, so `FileRef` is a handle to a file and Buffer is treated as an index into an array of pre-registered output buffers:



Both `IORING_SQE_PREREGISTERED_FILE` and `IORING_SQE_PREREGISTERED_BUFFER` flags are set, so `FileRef` is treated as an index into a pre-registered file handle array and `Buffer` is treated as index into a pre-registered buffers array:



Submitting and Processing I/O Ring

Once the caller set up all its submission queue entries, it can call `NtSubmitIoRing` to submit its requests to the kernel to get processed according to the requested parameters. Internally, `NtSubmitIoRing` iterates over all the entries and calls `IopProcessIoRingEntry`, sending the `IoRing` object and the current queue entry. The entry gets processed according to the specified `OpCode` and then calls `IopIoRingDispatchComplete` to fill in the completion queue. The completion queue, much like the submission queue, begins with a header, containing a `Head` and a `Tail`, followed by an array of entries. Each entry is an `IORING_CQE` structure – it has the `UserData` value from the submission queue entry and the Status and Information from the `IO_STATUS_BLOCK` for the operation:

```
typedef struct _IORING_CQE
{
    UINT_PTR UserData;
    HRESULT ResultCode;
    ULONG_PTR Information;
} IORING_CQE, *PIORING_CQE;
```

Once all requested entries are completed the system sets the event in `IoRingObject->RingEvent`. As long as not all entries are complete the system will wait on the event using the `Timeout` received from the caller and wake up when all requests are completed, causing the event to be signaled, or when the timeout expires.

Since multiple entries can be processed, the status returned to the caller will either be an error status indicating a failure to process the entries or the return value of `KeWaitForSingleObject`. Status codes for individual operations can be found in the completion queue – so don't confuse receiving a `STATUS_SUCCESS` code from `NtSubmitIoRing` with successful read operations!

Using I/O Ring – The Official Way

Like other system calls, those new IoRing functions are not documented and not meant to be used directly. Instead, `KernelBase.dll` offers convenient wrapper functions that receive easy-to-use arguments and internally handle all the undocumented functions and data structures that need to be sent to the kernel. There are functions to create, query, submit and close the `IoRing`, as well as helper functions to build queue entries for the four different operations, which were discussed earlier.

CreateIoRing

`CreateIoRing` receives information about flags and queue sizes, and internally calls `NtCreateIoRing` and returns a handle to an IoRing instance:

```
HRESULT
CreateIoRing (
    _In_ IORING_VERSION IoRingVersion,
    _In_ IORING_CREATE_FLAGS Flags,
    _In_ UINT32 SubmissionQueueSize,
    _In_ UINT32 CompletionQueueSize,
    _Out_ HIORING* Handle
);
```

This new handle type is actually a pointer to an undocumented structure containing the structure returned from `NtCreateIoRing` and other data needed to manage this `IoRing` instance:

```
typedef struct _HIORING
{
    ULONG SqePending;
    ULONG SqeCount;
    HANDLE handle;
    IORING_INFO Info;
    ULONG IoRingKernelAcceptedVersion;
} HIORING, *PHIORING;
```

All the other `IoRing` functions will receive this handle as their first argument.

After creating an `IoRing` instance, the application needs to build queue entries for all the requested I/O operations. Since the internal structure of the queues and the queue entry structures are not documented, `KernelBase.dll` exports helper functions to build those using input data supplied by the caller. There are four functions for this purpose:

1. `BuildIoRingReadFile`
2. `BuildIoRingRegisterBuffers`
3. `BuildIoRingRegisterFileHandles`
4. `BuildIoRingCancelRequest`

Each function create adds a new queue entry to the submission queue with the required opcode and data. Their names make their purposes pretty obvious but lets go over them one by one just for clarity:

BuildIoRingReadFile

```
HRESULT
BuildIoRingReadFile (
    _In_ HIORING IoRing,
    _In_ IORING_HANDLE_REF FileRef,
    _In_ IORING_BUFFER_REF DataRef,
    _In_ ULONG NumberOfBytesToRead,
    _In_ ULONG64 FileOffset,
    _In_ ULONG_PTR UserData,
    _In_ IORING_SQE_FLAGS Flags
);
```

The function receives the handle returned by `CreateIoRing` followed by two pointers to new data structures. Both of these structures have a `Kind` field, which can be either `IORING_REF_RAW`, indicating that the supplied value is a raw reference, or `IORING_REF_REGISTERED`, indicating that the value is an index into a pre-registered array. The second field is a union of a value and an index, in which the file handle or buffer will be supplied.

BuildIoRingRegisterFileHandles and BuildIoRingRegisterBuffers

```
HRESULT
BuildIoRingRegisterFileHandles (
    _In_ HIORING IoRing,
    _In_ ULONG Count,
    _In_ HANDLE const Handles[],
    _In_ PVOID UserData
);
```

```
HRESULT
BuildIoRingRegisterBuffers (
    _In_ HIORING IoRing,
    _In_ ULONG Count,
    _In_ IORING_BUFFER_INFO count Buffers[],
    _In_ PVOID UserData
);
```

These two functions create submission queue entries to pre-register file handles and output buffers. Both receive the handle returned from `CreateIoRing`, the count of pre-registered files/buffers in the array, an array of the handles or buffers to register and `UserData`.

In `BuildIoRingRegisterFileHandles`, Handles is a pointer to an array of file handles and in `BuildIoRingRegisterBuffers`, Buffers is a pointer to an array of `IORING_BUFFER_INFO` structures containing Buffer base and size.

BuildIoRingCancelRequest

```
HRESULT
BuildIoRingCancelRequest (
    _In_ HIORING IoRing,
    _In_ IORING_HANDLE_REF File,
    _In_ PVOID OpToCancel,
    _In_ PVOID UserData
);
```

Just like the other functions, `BuildIoRingCancelRequest` receives as its first argument the handle that was returned from `CreateIoRing`. The second argument is again a pointer to an `IORING_REQUEST_DATA` structure that contains the handle (or index in the file handles array) to the file whose operation should be canceled. The third and fourth arguments are the output buffer and `UserData` to be placed in the queue entry.

After all queue entries were built with those functions, the queue can be submitted:

SubmitIoRing

```
HRESULT
SubmitIoRing (
    _In_ HIORING IoRingHandle,
    _In_ ULONG WaitOperations,
    _In_ ULONG Milliseconds,
    _Out_ PULONG SubmittedEntries
);
```

The function receives the same handle as the first argument that was used to initialize the `IoRing` and submission queue. Then it receives the amount of entries to submit, time in milliseconds to wait on the completion of the operations, and a pointer to an output parameter that will receive the number of entries that were submitted.

GetIoRingInfo

```
HRESULT
GetIoRingInfo (
    _In_ HIORING IoRingHandle,
    _Out_ PIORING_INFO IoRingBasicInfo
);
```

This API returns information about the current state of the `IoRing` with a new structure:

```
typedef struct _IORING_INFO
{
  IORING_VERSION IoRingVersion;
  IORING_CREATE_FLAGS Flags;
  ULONG SubmissionQueueSize;
  ULONG CompletionQueueSize;
} IORING_INFO, *PIORING_INFO;
```

This contains the version and flags of the `IoRing` as well as the current size of the submission and completion queues.

Once all operations on the `IoRing` are done, it needs be closed using `CloseIoRing` which receives the handle as its only argument and closes the handle to the `IoRing` object and frees the memory used for the structure.

So far I couldn't find anything on the system that makes use of this feature, but once `21H2` is released I'd expect to start seeing I/O-heavy Windows applications start using it, probably mostly in server and azure environments.

Conclusion

So far, no public documentation exists for this new addition to the I/O world in Windows, but hopefully when `21H2` is released later this year we will see all of this officially documented and used by both Windows and 3rd party applications. If used wisely, this could lead to significant performance improvements for applications that have frequent read operations. Like every new feature and system call this could also have unexpected security effects. One bug was already found by hFirefox, who was the first to publicly mention this feature and managed to crash the system by sending an incorrect parameter to `NtCreateIoRing` – a bug that was fixed since then. Looking more closely into these functions will likely lead to more such discoveries and interesting side effects of this new mechanism.

Code

Here's a small PoC showing two ways to use I/O rings – either through the official `KernelBase` `API`, or through the internal ntdll `API`. For the code to compile properly make sure to link it against `onecoreuap.lib` (for the `KernelBase` functions) or `ntdll.lib` (for the `ntdll` functions):

```
#include <ntstatus.h>
#define WIN32_NO_STATUS
#include <Windows.h>
#include <cstdio>
#include <ioringapi.h>
#include <winternal.h>

typedef struct _IO_RING_STRUCTV1
{
    ULONG IoRingVersion;
    ULONG SubmissionQueueSize;
    ULONG CompletionQueueSize;
    ULONG RequiredFlags;
    ULONG AdvisoryFlags;
} IO_RING_STRUCTV1, *PIO_RING_STRUCTV1;
```

```
typedef struct _IORING_QUEUE_HEAD
{
    ULONG Head;
    ULONG Tail;
    ULONG64 Flags;
} IORING_QUEUE_HEAD, *PIORING_QUEUE_HEAD;

typedef struct _NT_IORING_INFO
{
    ULONG Version;
    IORING_CREATE_FLAGS Flags;
    ULONG SubmissionQueueSize;
    ULONG SubQueueSizeMask;
    ULONG CompletionQueueSize;
    ULONG CompQueueSizeMask;
    PIORING_QUEUE_HEAD SubQueueBase;
    PVOID CompQueueBase;
} NT_IORING_INFO, *PNT_IORING_INFO;

typedef struct _NT_IORING_SQE
{
    ULONG Opcode;
    ULONG Flags;
    HANDLE FileRef;
    LARGE_INTEGER FileOffset;
    PVOID Buffer;
    ULONG BufferSize;
    ULONG BufferOffset;
    ULONG Key;
    PVOID Unknown;
    PVOID UserData;
    PVOID stuff1;
    PVOID stuff2;
    PVOID stuff3;
    PVOID stuff4;
} NT_IORING_SQE, *PNT_IORING_SQE;

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

```
NTSTATUS
NtCreateIoRing (
    _Out_ PHANDLE pIoRingHandle,
    _In_ ULONG CreateParametersSize,
    _In_ PIO_RING_STRUCTV1 CreateParameters,
    _In_ ULONG OutputParametersSize,
    _Out_ PNT_IORING_INFO pRingInfo
    );

NTSTATUS
NtClose (
    _In_ HANDLE Handle
    );

EXTERN_C_END

void IoRingNt ()
{
    NTSTATUS status;
    IO_RING_STRUCTV1 ioringStruct;
    NT_IORING_INFO ioringInfo;
    HANDLE handle;
    PNT_IORING_SQE sqe;
    LARGE_INTEGER timeout;
    HANDLE hFile = NULL;
    ULONG sizeToRead = 0x200;
    PVOID *buffer = NULL;
    ULONG64 endOfBuffer;

    ioringStruct.IoRingVersion = 1;
    ioringStruct.SubmissionQueueSize = 1;
    ioringStruct.CompletionQueueSize = 1;
    ioringStruct.AdvisoryFlags = IORING_CREATE_ADVISORY_FLAGS_NONE;
    ioringStruct.RequiredFlags = IORING_CREATE_REQUIRED_FLAGS_NONE;

    status = NtCreateIoRing(&handle,
                            sizeof(ioringStruct),
                            &ioringStruct,
                            sizeof(ioringInfo),
                            &ioringInfo);
    if (!NT_SUCCESS(status))
    {
        printf("Failed creating IO ring handle: 0x%x\n", status);
        goto Exit;
    }
    ioringInfo.SubQueueBase->Tail = 1;
    ioringInfo.SubQueueBase->Head = 0;
    ioringInfo.SubQueueBase->Flags = 0;

    hFile = CreateFile(L"C:\\Windows\\System32\\notepad.exe",
                       GENERIC_READ,
                       0,
                       NULL,
                       OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL,
                       NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Failed opening file handle: 0x%x\n", GetLastError());
        goto Exit;
    }

    sqe = (PNT_IORING_SQE)((ULONG64)ioringInfo.SubQueueBase +
sizeof(IORING_QUEUE_HEAD));
    sqe->Opcode = 1;
    sqe->Flags = 0;
    sqe->FileRef = hFile;
    sqe->FileOffset.QuadPart = 0;
    buffer = (PVOID*)VirtualAlloc(NULL, sizeToRead, MEM_COMMIT,
PAGE_READWRITE);
    if (buffer == NULL)
    {
        printf("Failed allocating memory\n");
        goto Exit;
    }
    sqe->Buffer = buffer;
    sqe->BufferOffset = 0;
    sqe->BufferSize = sizeToRead;
    sqe->Key = 1234;
    sqe->UserData = nullptr;

    timeout.QuadPart = -10000;

    status = NtSubmitIoRing(handle, IORING_CREATE_REQUIRED_FLAGS_NONE, 1,
&timeout);
    if (!NT_SUCCESS(status))
    {
        printf("Failed submitting IO ring: 0x%x\n", status);
        goto Exit;
    }
    printf("Data from file:\n");
    endOfBuffer = (ULONG64)buffer + sizeToRead;
    for (; (ULONG64)buffer < endOfBuffer; buffer++)
    {
        printf("%p ", *buffer);
    }
    printf("\n");

Exit:
    if (handle)
    {
        NtClose(handle);
    }
    if (hFile)
    {
        NtClose(hFile);
    }
    if (buffer)
    {
        VirtualFree(buffer, NULL, MEM_RELEASE);
    }
}

void IoRingKernelBase ()
{
    HRESULT result;
    HIORING handle;
    IORING_CREATE_FLAGS flags;
    IORING_HANDLE_REF requestDataFile;
    IORING_BUFFER_REF requestDataBuffer;
    UINT32 submittedEntries;
    HANDLE hFile = NULL;
    ULONG sizeToRead = 0x200;
    PVOID *buffer = NULL;
    ULONG64 endOfBuffer;

    flags.Required = IORING_CREATE_REQUIRED_FLAGS_NONE;
    flags.Advisory = IORING_CREATE_ADVISORY_FLAGS_NONE;
    result = CreateIoRing(IORING_VERSION_1, flags, 1, 1, &handle);
    if (!SUCCEEDED(result))
    {
        printf("Failed creating IO ring handle: 0x%x\n", result);
        goto Exit;
    }

    hFile = CreateFile(L"C:\\Windows\\System32\\notepad.exe",
                        GENERIC_READ,
                        0,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Failed opening file handle: 0x%x\n", GetLastError());
        goto Exit;
    }
    requestDataFile.Kind = IORING_REF_RAW;
    requestDataFile.Handle = hFile;
    requestDataBuffer.Kind = IORING_REF_RAW;
    buffer = (PVOID*)VirtualAlloc(NULL,
                                   sizeToRead,
                                   MEM_COMMIT,
                                   PAGE_READWRITE);
    if (buffer == NULL)
    {
        printf("Failed to allocate memory\n");
        goto Exit;
    }
    requestDataBuffer.Buffer = buffer;
    result = BuildIoRingReadFile(handle,
                                  requestDataFile,
                                  requestDataBuffer,
                                  sizeToRead,
                                  0,
                                  NULL,
                                  IOSQE_FLAGS_NONE);
    if (!SUCCEEDED(result))
    {
        printf("Failed building IO ring read file structure: 0x%x\n",
result);
        goto Exit;
    }

    result = SubmitIoRing(handle, 1, 10000, &submittedEntries);
    if (!SUCCEEDED(result))
    {
        printf("Failed submitting IO ring: 0x%x\n", result);
        goto Exit;
    }
    printf("Data from file:\n");
    endOfBuffer = (ULONG64)buffer + sizeToRead;
    for (; (ULONG64)buffer < endOfBuffer; buffer++)
    {
        printf("%p ", *buffer);
    }
    printf("\n");

Exit:
    if (handle != 0)
    {
        CloseIoRing(handle);
    }
    if (hFile)
    {
        NtClose(hFile);
    }
    if (buffer)
    {
        VirtualFree(buffer, NULL, MEM_RELEASE);
    }
}

int main ()
{
    IoRingKernelBase();
    IoRingNt();
    ExitProcess(0);
}
```