

Introduction - Diago Lima

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

Introduction

:

Windows Thread Pools are an independently managed group of worker threads primarily used to make asynchronous desktop application development less of a hassle. Interacting with thread pools from a programming standpoint is simple: give the pool a “task” to complete, and it will handle the rest. Thread management is exclusively handled by the Thread Pool Manager, which is process-specific user mode code designed to handle the pool in its entirety.

Although thread pools are extremely simple in concept, their actual implementation is anything but. The first thing we need to understand is that thread pools are predominantly implemented in user mode, but handle certain operations with help from the kernel, which we’ll get to later. Each thread pool contains various critical components necessary for its operation. First is the **Worker Factory**, which facilitates the creation and deletion of new threads within the pool. Worker Factories are implemented entirely in user mode, although we can still acquire a “handle” to one, despite them not being kernel objects. This is a common theme within thread pools: non-kernel components being referenced via pseudo-handles, or “fake handles”.

Another important aspect of thread pools is their 3 distinct queues used by the worker threads. These queues consist of:

1. 1.
The work queue: a “general-purpose” queue, implemented directly within the pool itself, used to handle generic functions.
2. 2.
The I/O queue: a kernel object that signals the thread pool upon completion of certain I/O-related tasks, such as writing to files.
3. 3.
The timer queue: a queue where callbacks are executed after a delay from a timer completes. Also implemented directly in the pool, much like the work queue.

It’s important to note that tasks queued to these pools are not executed directly. Instead, each task is represented as a structure of some kind, which will contain a pointer to the actual callback to be executed. The layout of these structures changes drastically depending on the type of task, but that isn’t particularly relevant here. This is an example of what you might see with one of these “task structs”:

```
typedef struct _FULL_TP_WORK {
```

```
struct _TPP_CLEANUP_GROUP_MEMBER CleanupGroupMember;
```

```
struct _TP_TASK Task;
```

```
volatile union _TPP_WORK_STATE WorkState;
```

```
INT32 __PADDING__[1];
```

```
} FULL_TP_WORK, * PFULL_TP_WORK;
```

So why would an attacker choose to target a thread pool? There are a few reasons for this. Firstly, leveraging thread pools negates the need to create a remote thread ourselves for payload execution. This is a very common avenue of detection for EDR solutions, as the creation of a remote thread in most contexts is considered malicious. With thread pools however, we don't have to create a thread at all, and instead, we can let the pool handle the scheduling and execution of our payload, greatly increasing the chances of slipping by undetected. Also, every process in Windows is created with one thread pool by default, making the convenience of the technique another factor worth considering.

With this being said, the three main categories that a thread pool attack falls into include:

- Worker Factory attacks: attacking the pool via interaction with its worker factory.
- I/O port attacks: associating arbitrary I/O objects such as Events or Job Objects operated by the attacker's process with a remote thread pool's I/O port.
- Timer attacks: queueing a timer and wait interval associated with a payload to the target process' timer queue

Handle Hijacking

Before we can begin attacking a process' thread pool, we have a few prerequisites we need to take care of. Firstly, to directly interact with components of a process' thread pool, we'll need to get a duplicate copy of one of its handles. The handle we'll need depends on what category your attack falls into, which I just went over.

If you're attacking the worker factory, you'll need a handle that has a type of "TpWorkerFactory". If it's an I/O port attack, you'll need an "IoCompletion" handle, and if it's a timer attack, you'll need an "IRTimer" handle.

You can perform handle hijacking by using a semi-undocumented syscall, **NtQueryInformationProcess**. This syscall retrieves information about a specified process, based on what you pass in it's `ProcessInformationClass` parameter. The function prototype is as follows:

```
NtQueryInformationProcess(  
_In_ HANDLE ProcessHandle,  
_In_ PROCESSINFOCLASS ProcessInformationClass,  
_Out_writes_bytes_(ProcessInformationLength) PVOID ProcessInformation,  
_In_ ULONG ProcessInformationLength,  
_Out_opt_ PULONG ReturnLength  
);
```

The second parameter is an enumeration of type `PROCESSINFOCLASS`, which is partially documented by Microsoft. The enum value that we'll be using however has a value of 51, which is an undocumented value. When the second parameter in this call is 51, the `ProcessInformation` pointer passed must point to a `PROCESS_HANDLE_SNAPSHOT_INFORMATION` structure, which will receive information about every handle within the process' handle table.

This structure is defined as follows:

```
typedef struct _PROCESS_HANDLE_SNAPSHOT_INFORMATION  
{  
    ULONG_PTR NumberOfHandles;  
    ULONG_PTR Reserved;  
    PROCESS_HANDLE_TABLE_ENTRY_INFO Handles[ANYSIZE_ARRAY];  
} PROCESS_HANDLE_SNAPSHOT_INFORMATION, *  
PPROCESS_HANDLE_SNAPSHOT_INFORMATION;
```

Take note of the **Handles** member here. It's an array of structs, and each one is of type `PROCESS_HANDLE_TABLE_ENTRY_INFO`. Each structure within this array will contain info about a handle that the process has ownership over.

Once we have access to these handle snapshot structs, we can iterate over each of them, and check each time whether the handle's type is one of the three aforementioned ones: **TpWorkerFactory**, **IoCompletion**, or **IRTimer**. This type can be checked using the **NtQueryObject** syscall.

Once we locate the handle we want, we'll need to place it into our host process' handle table, so that we can gain ownership over it. We can do this by using the **DuplicateHandle** WinAPI function.