

Process Injection

 safebreach.com/blog/process-injection-using-windows-thread-pools

This website stores cookies on your computer. These cookies are used to improve your website experience and provide more personalized services to you, both on this website and through other media. To find out more about the cookies we use, see our Privacy Policy.

We won't track your information when you visit our site. But in order to comply with your preferences, we'll have to use just one tiny cookie so that you're not asked to make this choice again.

Threat Coverage | Research

Dec 6, 2023

The Pool Party You Will Never Forget: New Process Injection Techniques Using Windows Thread Pools

See how SafeBreach Labs Researchers developed a brand new set of highly flexible process injection techniques that are able to completely bypass leading endpoint detection and response (EDR) solutions.

Authors: Alon Leviev

During a cyber attack, malicious actors often breach an organization's perimeter security with tactics like vulnerability exploitation and phishing. Once inside, they attempt to navigate the organization's network to escalate their privileges and steal or encrypt data—but here they often face sophisticated endpoint detection and response (EDR) systems designed to identify and prevent this type of activity. To evade detection, threat actors have adopted process injection techniques that allow them to inject malicious code into a computer system's legitimate processes. The code is then executed by the target process—rather than the attacker—making it extremely difficult for organizations to identify and track from a forensics perspective.

While process injection techniques used to be more prevalent, most operating system (OS) and EDR vendors have tightened security measures to either block known techniques completely or severely limit their impact. As a result, fewer techniques have been seen in recent years and those still seen in the wild only work on specific process states—until now.

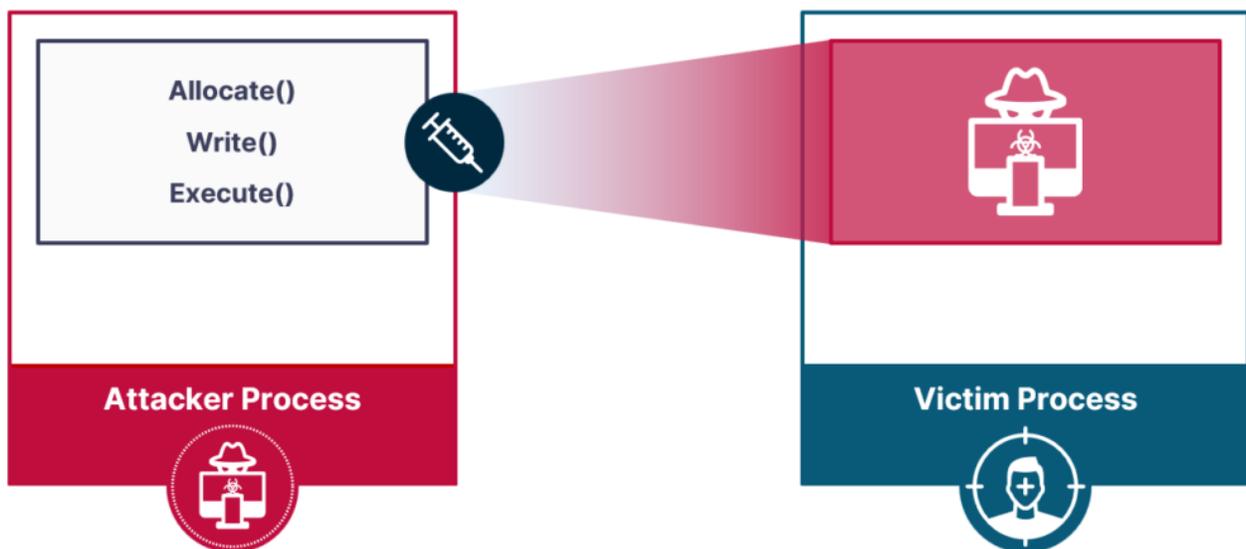
The SafeBreach Labs team set out to explore the viability of using Windows thread pools—an under-analyzed area of the Microsoft Windows OS—as a novel attack vector for process injection. In the process, we discovered eight new process injection techniques we dubbed Pool Party variants that were able to trigger malicious execution as a result of a completely legitimate action. The techniques were capable of working across all processes without any limitations, making them more flexible than existing process injection techniques. And, more importantly, the techniques were proven to be fully undetectable when tested against five leading EDR solutions.

Below we will share the details behind our research, which was first presented at [Black Hat Europe 2023](#). We will begin first with a high-level overview about how process injection works and how endpoint security controls detect current known techniques. We will then explain the architecture and relevant components of Windows thread pools and discuss the research process that led us to successfully exploit them to develop eight unique process injection techniques. Finally, we will highlight the EDR solutions we tested against and identify how SafeBreach is sharing this information with the broader security community to help organizations protect themselves.

Background

As an evasion technique used to execute arbitrary code in a target process, process injection usually consists of a chain of three primitives:

1. Allocation primitive: Used to allocate memory on the target process
2. Writing primitive: Used to write malicious code to the allocated memory
3. Execution primitive: used to execute the malicious code written

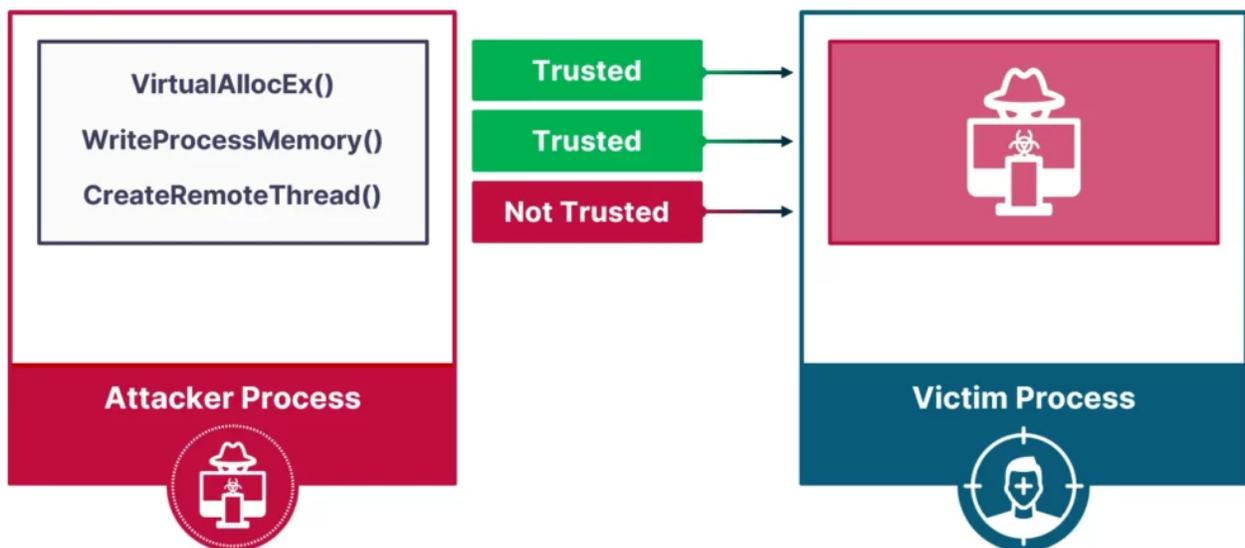


The most basic injection technique would use `VirtualAllocEx()` for allocation, `WriteProcessMemory()` for writing, and `CreateRemoteThread()` for execution. This injection technique, publicly known as `CreateRemoteThread` injection, is very simple and powerful, but there is one downside: it is detectable by all modern EDRs. Our research sought to discover if it was possible to create process injection techniques that were fully undetectable.

Through this process, we sought to understand if EDRs could effectively distinguish the legitimate versus malicious use of a feature. We also wanted to find out if the current detection approach used by EDRs was generic enough to detect new and never-before-seen process injections.

EDR Detection Approach

To answer these questions, we needed to review the current detection approach employed by EDRs against process injections. Experimenting with the different primitives led us to the conclusion that EDRs base their detection mainly on the execution primitive. On top of that, write and allocate primitives—in their most basic forms—are not detected.



Based on this finding, what would happen if we created an execution primitive based only on allocation and writing primitives? Furthermore, what if the execution was triggered by a legitimate action—writing to an innocent file, for example—and could trigger shellcode on a victim process? Such capabilities would make the process injection even harder to detect.

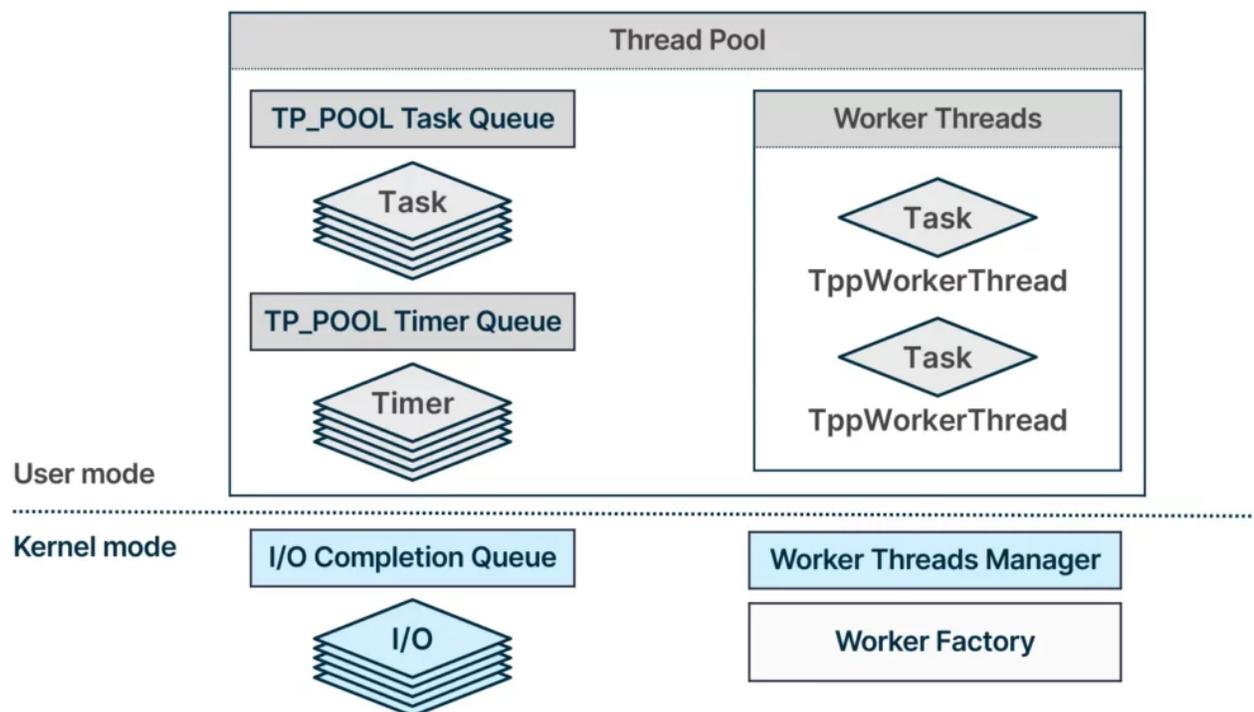
Windows User-Mode Thread Pools

While searching for a suitable component that would help achieve the research goals, we came across the Windows user-mode thread pool. This ended up being the perfect target, because:

1. All Windows processes have a thread pool by default, which means that abusing the thread pool would be applicable against all Windows processes.
2. Work items and thread pools are represented by structures, which increases the chances of having an execution primitive based on the allocation and writing primitives.
3. Multiple work item types are supported, which means more opportunities.
4. The thread pool is a considerably complex component, with both kernel and user-mode code, which widens the attack surface.

Architecture

The thread pool comprises three distinct work queues, each dedicated to a different type of work item. The worker threads are operating on the different queues to dequeue work items and execute them. In addition, the thread pool contains a worker factory object, which is responsible for managing the worker threads.



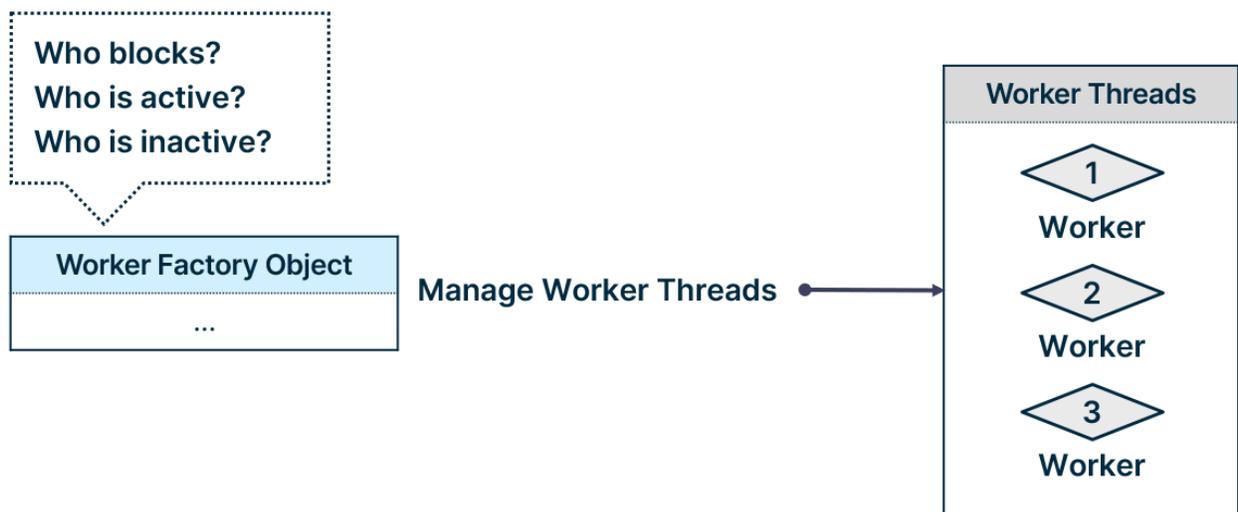
Based on this architecture, there are few potential areas in the thread pool that could be abused for process injections:

1. Worker factory
2. Task queue
3. I/O completion queue
4. Timer queue

We know that a valid work item insertion into one of these queues would be executed by the worker threads. Other than the queues, the worker factory that serves as the worker threads manager may be used to take over the worker threads.

Attacking Worker Factories

The worker factory is a Windows object responsible for managing thread pool worker threads. It manages the worker threads by monitoring active or blocking worker threads and, based on the monitoring results, it creates or terminates worker threads. The worker factory does not perform any scheduling or execution of work items; it is there to make sure that the number of worker threads is sufficient.



The kernel exposes seven system calls to interact with worker factory objects:

- NtCreateWorkerFactory
- NtShutdownWorkerFactory
- NtQueryInformationWorkerFactory
- NtSetInformationWorkerFactory
- NtWorkerFactoryWorkerReady
- NtWaitForWorkViaWorkerFactory
- NtReleaseWorkerFactoryWorker

With the goal of taking over worker threads, the relevant target would be the start routine. The start routine is basically the entry point of the worker threads—usually this routine serves as the thread pool scheduler, responsible for dequeuing and executing work items.

The start routine can be controlled in the worker factory creation system call and, more interestingly, the system call accepts a handle to the process for which the worker factory to

be created:

```
NTSTATUS NTAPI NtCreateWorkerFactory(
    _Out_ PHANDLE WorkerFactoryHandleReturn,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE CompletionPortHandle,
    _In_ HANDLE WorkerProcessHandle,
    _In_ PVOID StartRoutine,
    _In_opt_ PVOID StartParameter,
    _In_opt_ ULONG MaxThreadCount,
    _In_opt_ SIZE_T StackReserve,
    _In_opt_ SIZE_T StackCommit
);
```

Looking at the implementation of the system call in the kernel, we noticed that there is a validation that makes sure no worker factories are created for processes other than the current process:

Ntoskrnl:: NtCreateWorkerFactory

```
NTSTATUS NTAPI NtCreateWorkerFactory(..., HANDLE WorkerProcessHandle, ...)
{
    [snip]

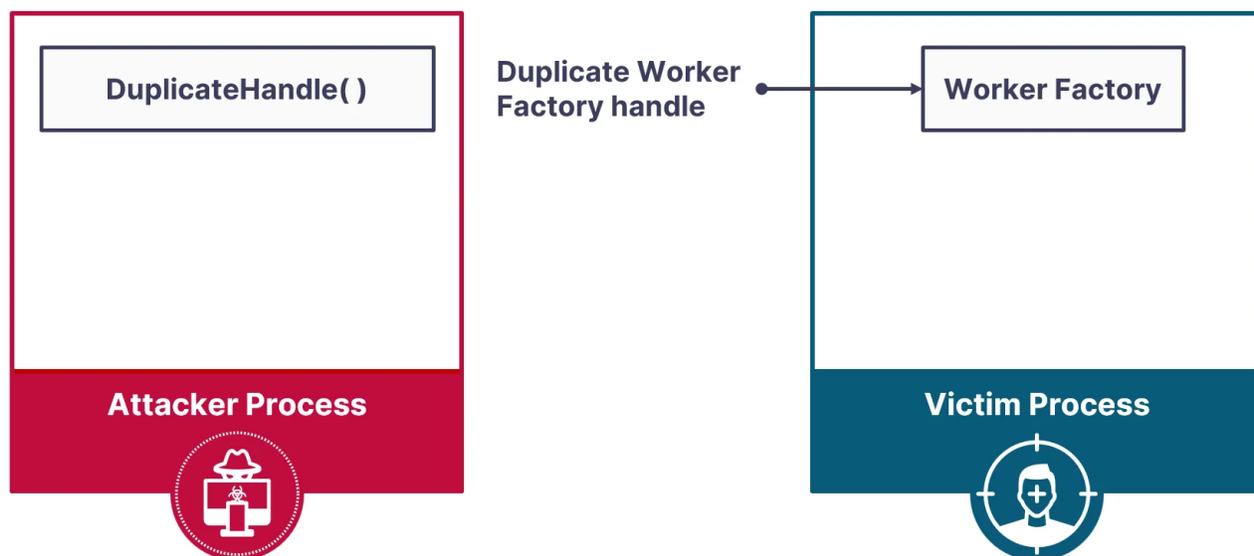
    KPROCESS * pWorkerProcessObject;
    ObpReferenceObjectByHandleWithTag(WorkerProcessHandle, ..., &pWorkerProcessObject);

    if ( KeGetCurrentThread()->ApcState.Process != pWorkerProcessObject)
    {
        return STATUS_INVALID_PARAMETER;
    }

    [snip]
}
```

Generally speaking, it is a bit odd that the system call gets a parameter with only one possible value. All processes have a thread pool by default, and consequently, a worker factory by default.

Instead of going through the trouble of creating a worker factory, we can simply utilize the DuplicateHandle() API to gain access to a worker factory belonging to the target process.



Having access to an existing worker factory did not let us control the start routine value, as this value is constant and could not be naturally changed after the object was initialized. With that said, if we could determine the start routine value, we could overwrite the routine code with a malicious shellcode.

To get worker factory information, the NtQueryWorkerFactoryInformation system call could be used:

```
NTSTATUS NTAPI NtQueryInformationWorkerFactory(
    _In_ HANDLE WorkerFactoryHandle,
    _In_ QUERY_WORKERFACTORYINFOCLASS WorkerFactoryInformationClass,
    _In_reads_bytes_(WorkerFactoryInformationLength) PVOID WorkerFactoryInformation,
    _In_ ULONG WorkerFactoryInformationLength,
    _Out_opt_ PULONG ReturnLength
);
```

The only supported information class that the query system call can retrieve is basic worker factory information:

```
typedef enum _QUERY_WORKERFACTORYINFOCLASS
{
    WorkerFactoryBasicInformation = 7,
} QUERY_WORKERFACTORYINFOCLASS, * PQUERY_WORKERFACTORYINFOCLASS;
```

In this case this is enough, as the basic worker factory information includes the start routine value:

```

typedef struct _WORKER_FACTORY_BASIC_INFORMATION
{
    [snip]

    PVOID StartRoutine;

    [snip]
} WORKER_FACTORY_BASIC_INFORMATION, * PWORKER_FACTORY_BASIC_INFORMATION;

```

Given the start routine value, we could overwrite the start routine content with malicious shellcode.

The start routine is guaranteed to run at some point, but it would be even better if we could also trigger its execution instead of waiting for it. To accomplish this, we looked at the `NtSetInformationWorkerFactory` system call:

```

NTSTATUS NTAPI NtSetInformationWorkerFactory(
    _In_ HANDLE WorkerFactoryHandle,
    _In_ SET_WORKERFACTORYINFOCLASS WorkerFactoryInformationClass,
    _In_reads_bytes_(WorkerFactoryInformationLength) PVOID WorkerFactoryInformation,
    _In_ ULONG WorkerFactoryInformationLength
);

```

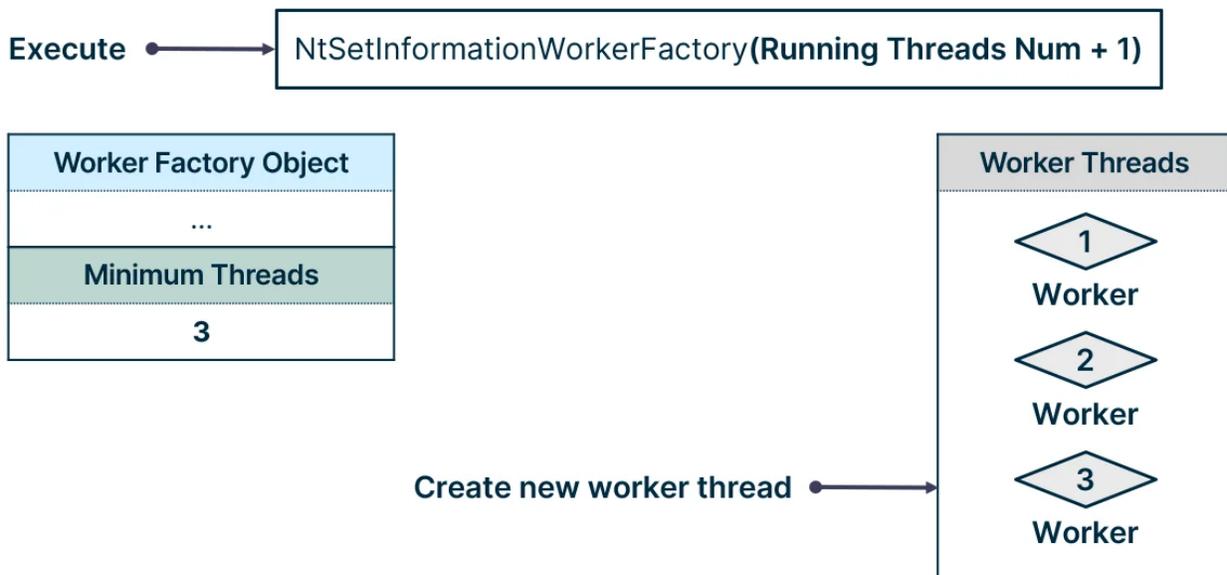
The set system call supports more information classes than the query system call, and the one that suited our needs the best was the `WorkerFactoryThreadMinimum` information class:

```

typedef enum _SET_WORKERFACTORYINFOCLASS
{
    WorkerFactoryTimeout = 0,
    WorkerFactoryRetryTimeout = 1,
    WorkerFactoryIdleTimeout = 2,
    WorkerFactoryBindingCount = 3,
    WorkerFactoryThreadMinimum = 4,
    WorkerFactoryThreadMaximum = 5,
    WorkerFactoryPaused = 6,
    WorkerFactoryAdjustThreadGoal = 8,
    WorkerFactoryCallbackType = 9,
    WorkerFactoryStackInformation = 10,
    WorkerFactoryThreadBasePriority = 11,
    WorkerFactoryTimeoutWaiters = 12,
    WorkerFactoryFlags = 13,
    WorkerFactoryThreadSoftMaximum = 14
} SET_WORKERFACTORYINFOCLASS, * PSET_WORKERFACTORYINFOCLASS;

```

Setting the minimum worker threads number to be the current running threads number + 1 resulted in a new worker thread being created, meaning the start routine was executed:



And with that, we successfully developed our first Pool Party variant:

Pool Party Variant 1: Worker Factory Start Routine Overwrite

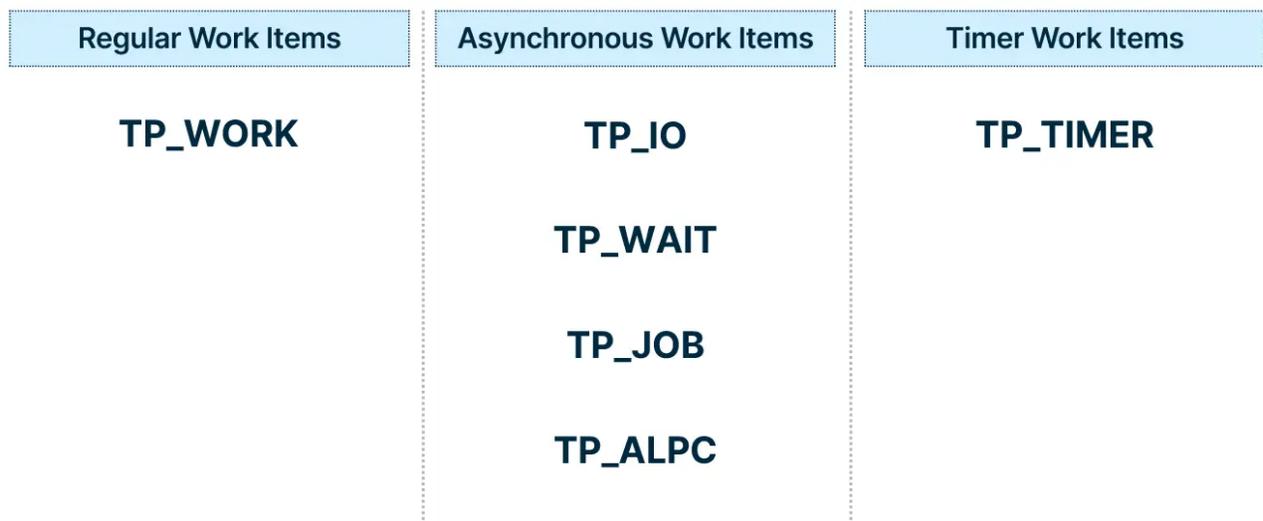
Attacking Thread Pools

When attacking the thread pool, our goal was to insert a work item to a target process, so we focused on how work items are inserted into the thread pool. We know that if we insert a work item correctly, it will be executed by the worker threads. We will assume that we already have access to the worker factory of the target thread pool, as we proved in the previous section that such access can be granted by duplicating the worker factory handle.

Work Item Types

The supported work items can be divided into three types:

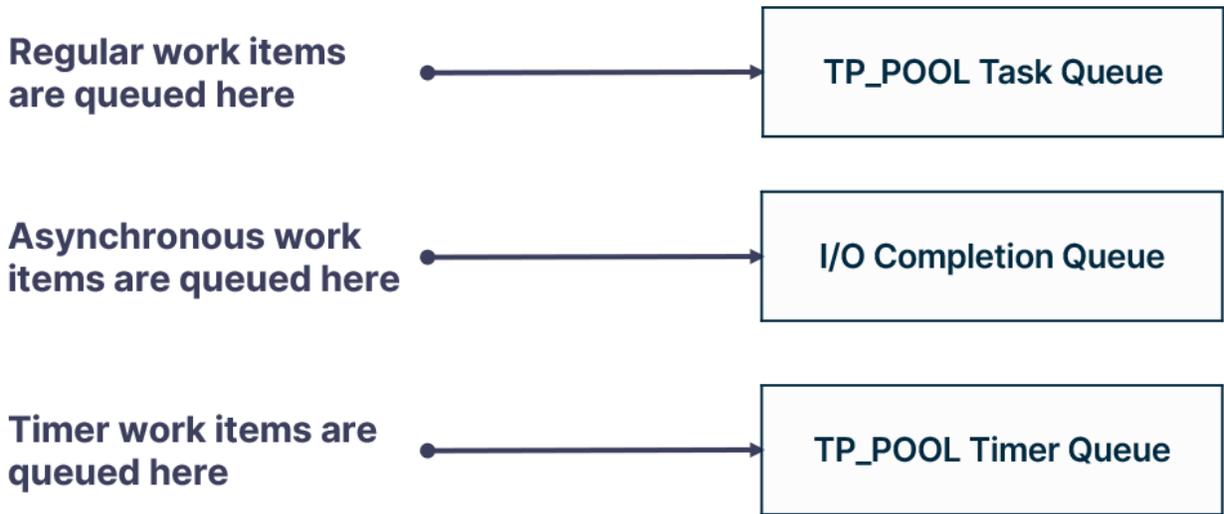
- The regular work items, which are queued right away by the queueing API call.
- The asynchronous work items, which are queued on operation completion, for example, when a write file operation is completed.
- The timer work items, which are queued right away by the queueing API call, but are executed when a timer expires.



Queue Types

As for the three types of work items, there are also three queues:

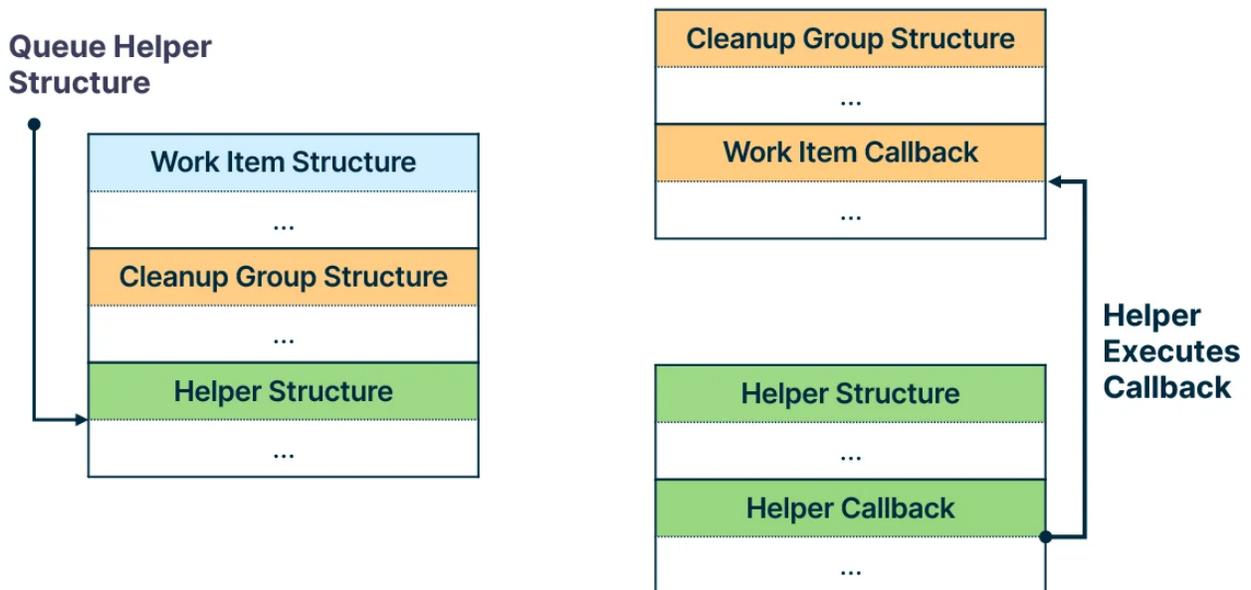
- The regular work items are queued to the task queue, residing in the main thread pool structure, the TP_POOL.
- The asynchronous work items are queued to the I/O completion queue, which is a Windows object.
- And timer work items are queued to the timer queue, also residing in the main thread pool structure.



The main thread pool structure resides in user-mode in the process memory address space, so modifications to its queues can be done through memory writing primitives. The I/O completion queue is a Windows object, so the queue resides in the kernel and can be manipulated by its exposed system calls.

Helper Structures

Before we dive into the queueing mechanism of each work item type, it is important to note that work item callbacks are not executed directly by the worker threads. Instead, each work item has a helper callback that is used to execute the work item callback. The structure that is queued is the helper structure.



Attacking Thread Pools: TP_WORK

By looking at the TP_WORK work item structure, we found that its helper structure is the TP_TASK structure. We know that the task structure is what gets inserted into the task queue within the thread pool structure.

```
typedef struct _TP_WORK
{
    TPP_CLEANUP_GROUP_MEMBER CleanupGroupMember;
    TP_TASK Task; ← Helper Structure
    TPP_WORK_STATE WorkState;
    INT32 __PADDING__[1];
} TP_WORK, * PTP_WORK;
```

The API that is responsible for submitting the TP_WORK work item is named SubmitThreadpoolWork. Going down the call chain of SubmitThreadpoolWork, we reached the queueing API named TpPostTask.

The TpPostTask API is responsible for inserting a task to the task queue, which is represented by a doubly linked list. It retrieves the corresponding task queue by priority and inserts the task to the tail of the task queue.

Ntdll:: TpPostTask

```
NTSTATUS NTAPI TpPostTask(TP_TASK* TpTask, TP_POOL* TpPool, int CallbackPriority, ...)
{
    [snip]

    TPP_QUEUE* TaskQueue = &TpPool->TaskQueue[CallbackPriority];

    InsertTailList(&TaskQueue->Queue, &TpTask->ListEntry);

    [snip]
}
```

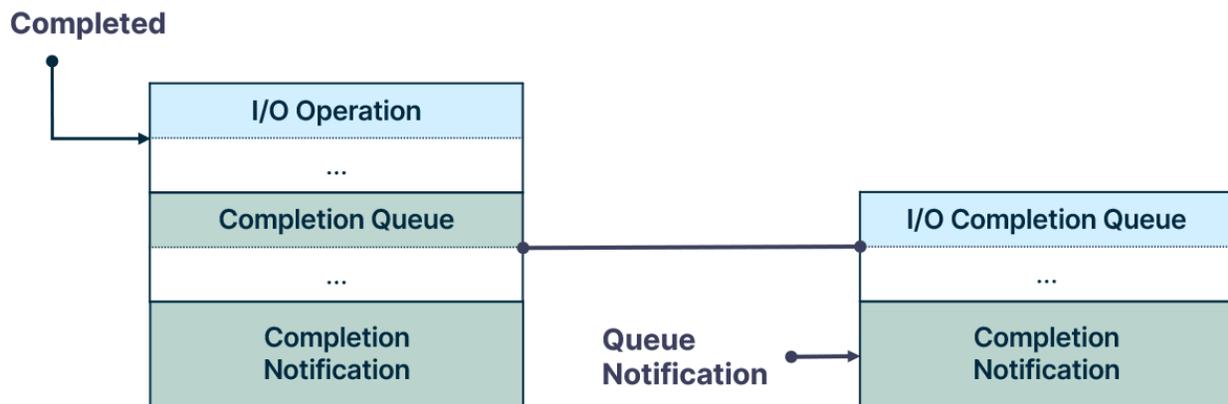
Given the thread pool structure of the target process, we could tamper with its task queue to inject a malicious task into it. To get the thread pool structure of the target process, the NtQueryInformationWorkerFactory could be used. The basic worker factory information

included the start parameter of the start routine, and this start parameter was essentially a pointer to the TP_POOL structure. We had our second Pool Party variant:

Pool Party Variant 2: Remote TP_WORK Work Item Insertion

Attacking Thread Pools: TP_IO

Recalling the queue types, asynchronous work items are queued to the I/O completion queue. The I/O completion queue is a Windows object that serves as a queue for completed I/O operations. Notifications are inserted into the queue once an I/O operation completes



The thread pool relies on the I/O completion queue to receive notifications when an asynchronous work item's operation is completed.

NOTE: Microsoft refers to I/O completion queues as I/O completion ports. This object is essentially a kernel queue (KQUEUE), so to avoid confusion we refer to it as I/O completion queue.

The kernel exposes eight system calls to interact with I/O completion queues:

- NtCreateIoCompletion
- NtOpenIoCompletion
- NtQueryIoCompletion
- NtQueryIoCompletionEx
- NtSetIoCompletion
- NtSetIoCompletionEx
- NtRemoveIoCompletion
- NtRemoveIoCompletionEx

Keep in mind the NtSetIoCompletion system call is used to queue a notification to the queue. We will get back to this system call later on.

Equipped with some I/O completion background, we can jump right into the queuing mechanism of the asynchronous work items. We will use the TP_IO work item as an example, but note that the same concepts apply to the other asynchronous work items.

The TP_IO work item is a work item intended to execute on completion of file operations such as read and write. The helper structure of the TP_IO work item is the TP_DIRECT structure, so we expect this structure to be queued to the completion queue.

```
typedef struct _TP_IO
{
    _TPP_CLEANUP_GROUP_MEMBER CleanupGroupMember;
    _TP_DIRECT Direct; ← ● Helper Structure
    HANDLE File;
    INT32 PendingIrpCount;
    INT32 __PADDING__[1];
} TP_IO, * PTP_IO;
```

As asynchronous work items were queued to the I/O completion queue, we looked for the function that associated the work item to the thread pool's I/O completion queue. Looking at the call chain of CreateThreadpoolIo, we reached the function of interest: the TpBindFileToDirect function. This function sets the file completion queue to be the thread pool's I/O completion queue, and the file completion key to be the direct structure:

Ntdll:: TpBindFileToDirect

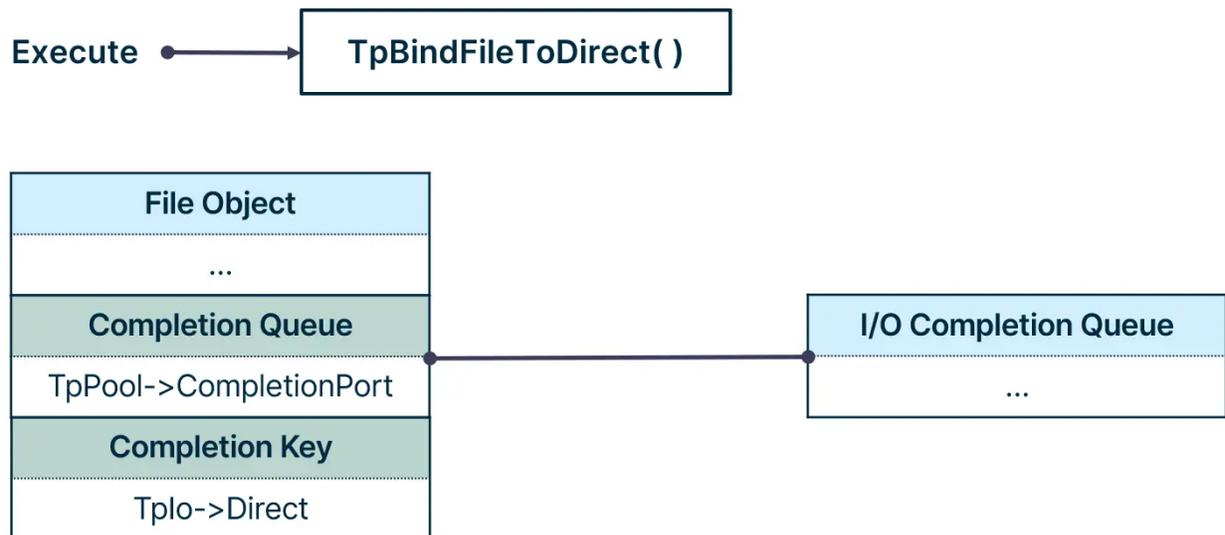
```
NTSTATUS NTAPI TpBindFileToDirect(HANDLE hFile, TP_DIRECT* TpDirect, TP_POOL* TpPool)
{
    [snip]

    FILE_COMPLETION_INFORMATION FileCompletionInfo{ 0 };
    FileCompletionInfo.Key = TpDirect;
    FileCompletionInfo.Port = TpPool->CompletionPort;

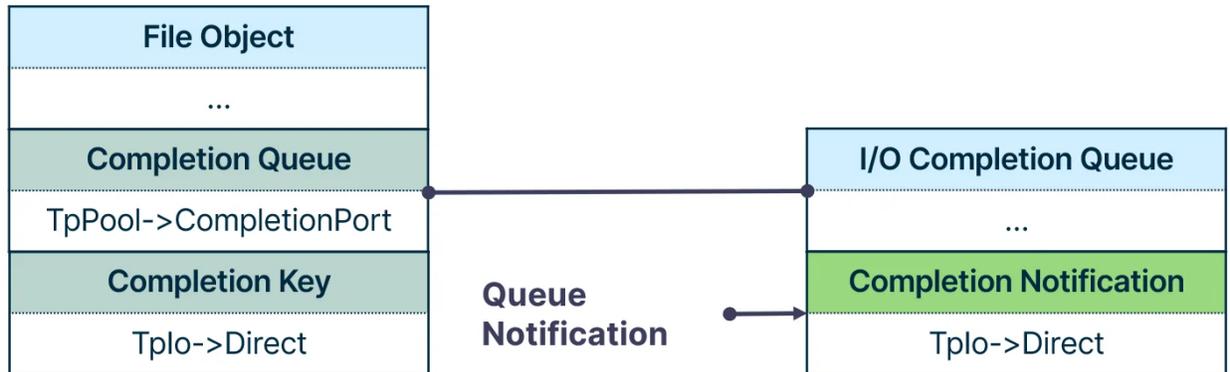
    NtSetInformationFile(
        hFile,
        &IoStatusBlock,
        &FileCompletionInfo,
        sizeof(FILE_COMPLETION_INFORMATION),
        FileCompletionInformation);

    [snip]
}
```

Calling `TpBindFileToDirect` on a file object results in the completion queue of the file object pointing to the thread pool's I/O completion queue, and the completion key pointing to the direct structure.



At that point, the I/O completion queue was still empty, as no operation on the file occurred. Any operation on the file following the function call—for example, `WriteFile`—would cause the completion key to be queued to the I/O completion queue.



To conclude, asynchronous work items are queued to the I/O completion queue and the direct structure is the field that is queued. Having a handle to the I/O completion queue of the target process gave us the ability to queue notifications to it. This handle could be duplicated using the DuplicateHandle API, similarly to how we duplicated the worker factory handle. And with that, we had our third Pool Party variant:

Pool Party Variant 3: Remote TP_IO Work Item Insertion

How did we also insert ALPC, JOB and WAIT work items? Any valid TP_DIRECT structure queued to the I/O completion queue will get executed. It is all a matter of how we queue the TP_DIRECT structure to the I/O completion queue.

Queuing can be done in one of the following ways:

1. Utilizing Windows objects, similar to the TP_IO abuse. This will involve associating the object with the I/O completion queue of the target process, and then any operation completion on this object will queue a notification.
2. Utilizing NtSetIoCompletion to queue a notification directly into the completion queue.

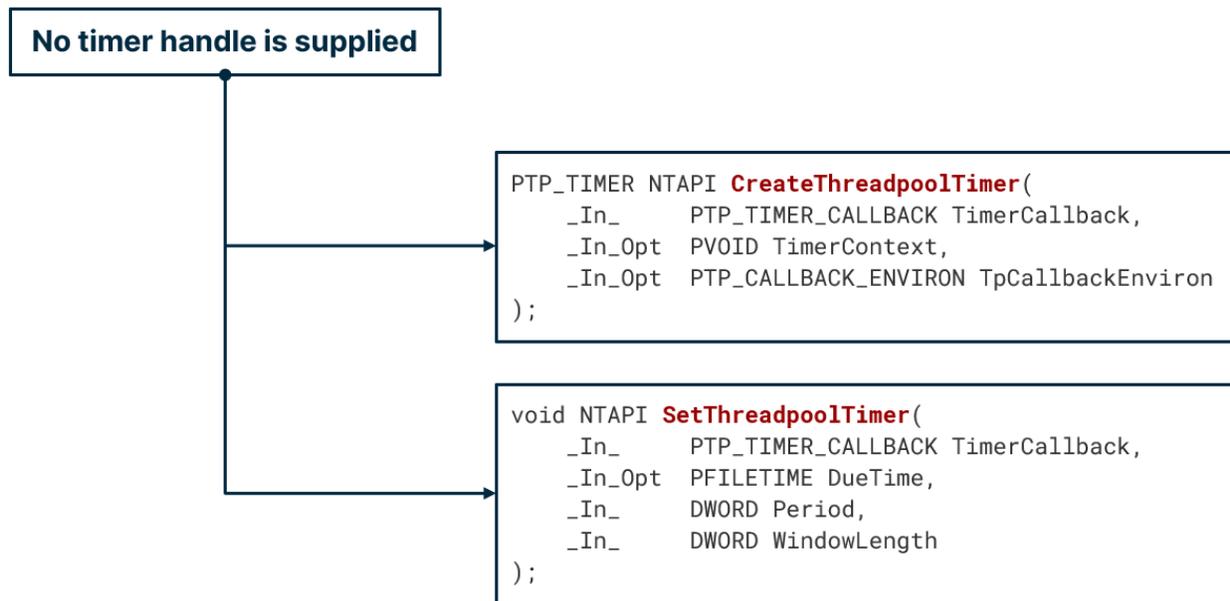
With that in mind, we can inject the rest of the asynchronous work items, the TP_WAIT, TP_ALPC and TP_JOB, by associating the underlying Windows object with the target thread pool's I/O completion queue, and setting its completion key to point to the malicious work item. On top of that, we can inject a malicious TP_DIRECT structure directly without proxying it through a Windows object, which involves using the NtSetIoCompletion system call. This allowed us to create four more Pool Party variants:

- PoolParty Variant 4 – Remote TP_WAIT Work Item Insertion
- PoolParty Variant 5 – Remote TP_ALPC Work Item Insertion
- PoolParty Variant 6 – Remote TP_JOB Work Item Insertion
- PoolParty Variant 7 – Remote TP_DIRECT Insertion

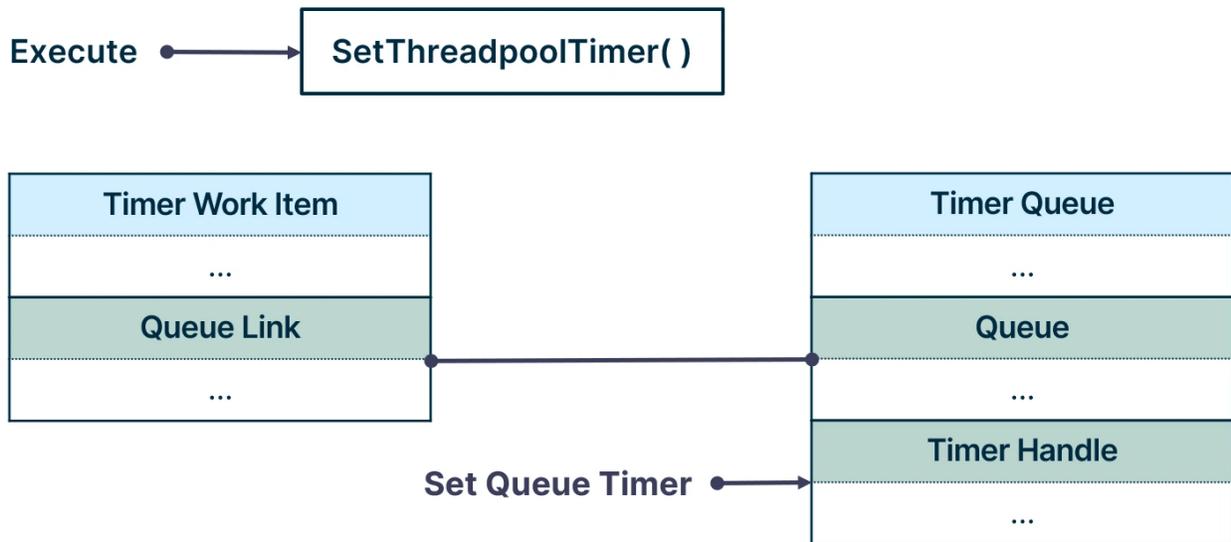
These variants are special as the execution is triggered by a completely legitimate action, such as writing to a file, connecting to an ALPC port, assigning a process to a job object, and setting an event.

Attacking Thread Pools: TP_TIMER

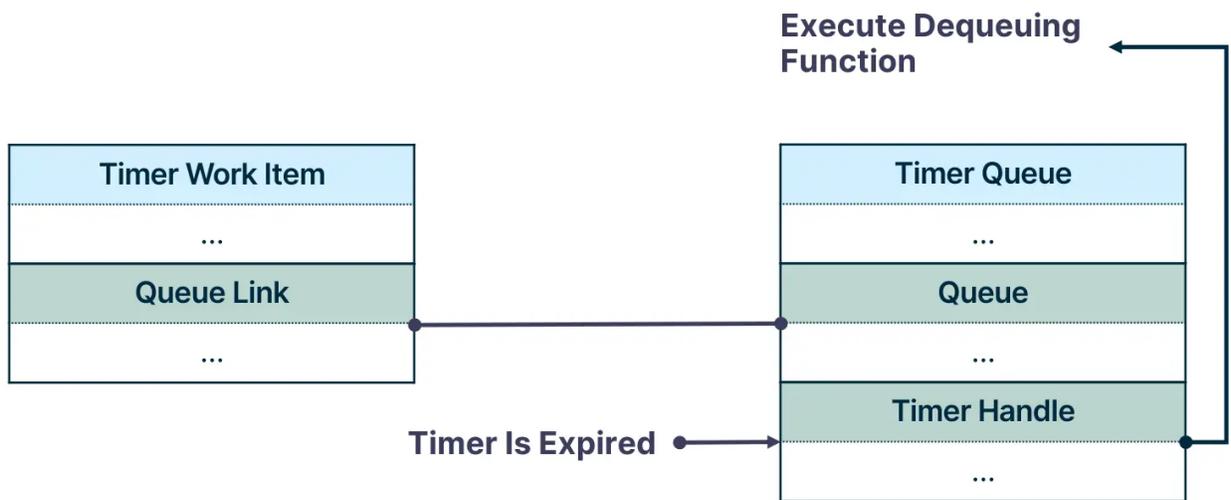
First, when looking at the creation and submission API of a timer work item, we noticed that no timer handle was supplied. The submission API, `SetThreadpoolTimer`, accepts some timer configuration such as `DueTime`, but it wasn't clear where the actual timer object resided.



It turns out that timer work items operate on an existing timer object, which resides in the timer queue. Once the `SubmitThreadpoolTimer` API is called, the work item is inserted into the queue, and the timer object residing in the queue is configured with the user-supplied configuration.



Once the timer is expired, a dequeuing function is called, which dequeues the work item from the queue and executes it.



Generally speaking, timer objects do not natively support callback execution at expiration. All you need to know is that the thread pool implements it using the `TP_WAIT` work item, which supports timers. So if we set the timer queue to expire, the dequeuing function is called. Now the question is, how do we correctly queue a timer to the queue?

The connectors between a timer and a timer queue are the `TP_TIMER`'s `WindowEndLinks` and `WindowStartLinks` fields.

For the sake of simplicity, we can think of these two fields as list entries of a doubly linked list.

```

typedef struct _TP_TIMER
{
    [snip]
    _TPP_PH_LINKS WindowEndLinks;
    _TPP_PH_LINKS WindowStartLinks;
    [snip]
} TP_TIMER, * PTP_TIMER;

```

Going down the call chain of SetThreadpoolTimer, we reached the queueing function named TppEnqueueTimer.

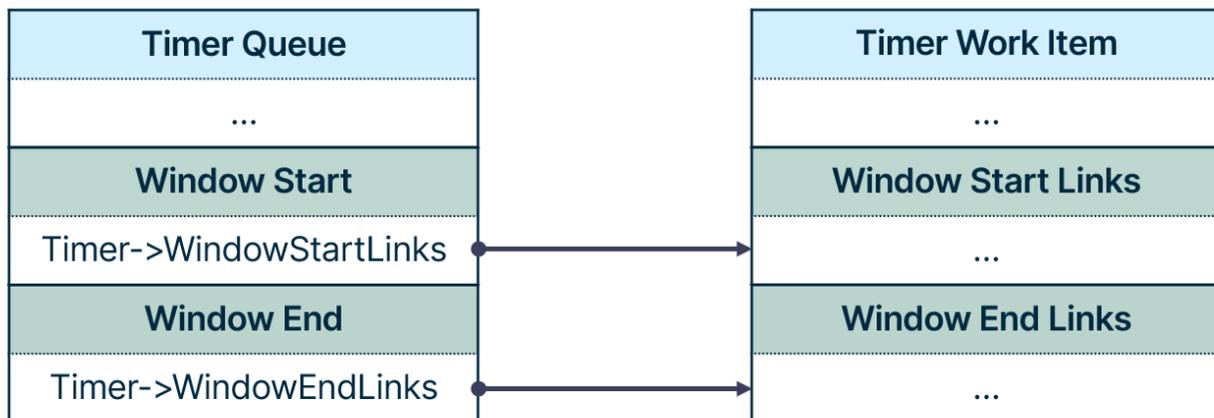
Ntdll:: TppEnqueueTimer

```

NTSTATUS NTAPI TppEnqueueTimer(TPP_TIMER_QUEUE* TimerQueue, TP_TIMER* TpTimer)
{
    [snip]
    TppPHInsert(&TimerQueue->WindowStart, &TpTimer->WindowStartLinks);
    TppPHInsert(&TimerQueue->WindowEnd, &TpTimer->WindowEndLinks);
    [snip]
}

```

TppEnqueueTimer inserts the TP_TIMER's WindowStartLinks to the queue WindowStart field, and the WindowEndLinks to the queue WindowEnd field.



The SetThreadPoolTimer API is responsible for two actions:

1. Queue the timer work item to the timer queue.
2. Configure the timer object residing in the queue.

As a result of these two actions, once the timer object expires, the dequeuing function executes, dequeuing and executing the queued timer work item. Given the thread pool structure of the target process, we can tamper with its timer queue to inject a malicious timer work item into it. Post queueing, we need to set the timer object that the queue uses to expire. Setting the timer requires a handle to it, and such handle could be duplicated using the DuplicateHandle API. And with that, we had our eighth Pool Party variant:

Pool Party Variant 8: Remote TP_TIMER Work Item Insertion

What’s even more astonishing about this variation is that after setting the timer, the attacker can exit the process and erase its identity from the system. As a result, the system appears clean, and the malicious code activates only when the timer runs out.

Tested EDR Solutions

As part of the research process, each Pool Party variant was tested against five leading EDR solutions, including:

- Palo Alto Cortex
- SentinelOne EDR
- CrowdStrike Falcon
- Microsoft Defender For Endpoint
- Cybereason EDR

We achieved a 100 percent success rate, as none of the EDRs were able to detect or prevent Pool Party attacks. We reported these findings to each vendor and believe they are making updates to better detect these types of techniques.

It is important to note that while we have done our best to test the EDR products we had access to, it is not feasible for us to test every product on the market. By making this information available to the security community, we hope to minimize the ability of malicious actors to exploit these techniques and provide EDR vendors and users with the knowledge they need to take immediate action on their own.

Pool Party Demo

Key Takeaways

We believe there are a few important takeaways based on the findings of this research:

1. Although EDRs have evolved, the current detection approach utilized by most solutions is unable to generically detect new process injection techniques like those we have developed here. While our research demonstrates how we were able to abuse thread pools specifically, malicious actors will undoubtedly find other features to leverage in a similar way. We believe it is critical for EDR vendors to develop and implement a generic detection approach to proactively defend against these possibilities.
2. We also believe it is important for individual organizations to enhance their focus on detecting anomalies, rather than placing complete trust in processes based solely on their identity. Our research demonstrates that executing code on the behalf of a trusted process can go undetected by an EDR. This underscores the importance of deeper inspection to ensure the legitimacy of operations performed by such processes.

Conclusion

Though modern EDRs have evolved to detect known process injection techniques, our research has proven that it is still possible to develop novel techniques that are undetectable and have the potential to make a devastating impact. Sophisticated threat actors will continue to explore new and innovative methods for process injection, and security tool vendors and practitioners must be proactive in their defense against them.

To help mitigate the potential impact of these techniques, we have:

- Responsibly disclosed our research findings to Microsoft, Palo Alto Networks, CrowdStrike, SentinelOne, and Cybereason.

- Shared our research openly with the broader security community here and at our recent Black Hat presentation to raise awareness about these issues.
- Added Original Attack content to the SafeBreach platform that enables our customers to validate their security controls against these flows and significantly mitigate their risk.

For more in-depth information about this research, please:

- Contact your customer success representative if you are a current SafeBreach customer
- [Schedule a one-on-one discussion](#) with a SafeBreach expert
- Contact Kesselring PR for media inquiries

About the Researcher

Alon Leviev is a self-taught security researcher with a diverse background. Alon started his professional career as a blue team operator, where he focused on the defensive side of cyber security. As his passion grew towards research, Alon joined SafeBreach as a security researcher. His main interests include operating system internals, reverse engineering, and vulnerability research. Before joining the cyber security field, Alon was a professional Brazilian jiu-jitsu athlete, where he won several world and European titles.

Get the latest research and news
