Waiting-Thread Hijacking

Despite updates to the EDR/AV world, injection and other primitive techniques not only live on, but remain integral to offensive security in their evolved implementations. Much of EDR detection logic still revolves around a combination of observable behavior and historic data to derive values to compare benign applications to their potentially malicious counterparts. The biggest paradigm shift within the offensive security community has been the realization that less is more – and more importantly, “recycling” is king. Living off the land through essential binaries and services has significantly increased reliability and integrity of engagements (optimized applications with well-developed cleanup/error handling routines) as well as drastically improving rates of success when undetected at the point of initial execution. This change caused a paradigm shift of its own in the EDR world, as a shift from basic patterning such as expected/unexpected process trees was abandoned in favor of dynamically calculated behavioral analysis. Circumventing complex behavioral scoring is difficult, and those on the cutting edge are moving away from thread creation. The rest of this document is a breakdown of this newer school of thought, with an example of the newest injection technique: Waiting-Thread Hijacking.

Key Concepts

Part 1 of EDR Underpinnings briefly discusses some of the most important APIs used for almost every injection technique that has been documented. There are a few more that are necessary to know, as well as additional concepts for code execution and Windows process internals.

Winternals: Processes

The easiest way to visualize a process is to imagine a series of labeled boxes. Box A is large to represent the total memory existent (both allocated and unallocated). Within box A sits box B (newly created process), completely empty and takes up a large chunk of box A. One portion of Box B is stuffed with documents (image data). In order to read these documents, scanner C is placed in the box – powered, but awaiting instructions. With all of this assembled, the scanner is now acting as the thread running inside of allocated process space. Until given the instruction to execute user code, the thread will remain queued. This is not the same as being created in a suspended state, which requires a call to resume the thread.

* API Calls to Note
  + NtQuerySystemInformation
    - Parameter: SystemProcessInformation
  + GetThreadContext
    - Returns current state and data for thread
      * This is used for efficiency to allow a thread to execute newly prioritized code and return to the original execution flow
  + GetThreadDescription
    - Gets name of thread (handle passed parameter) within a given process
* Thread Pools
  + Collections of threads controlled by worker processes in order to delegate and efficiently manage multi-threaded processes
    - Waiting-Thread
      * Some pools will be constantly busy, and others will be kept readily available for any incoming instructions
      * Inspecting a thread’s callstack will show Tp\* prepended functions when Thread Pool related
  + Interesting item to note is that the efficiency of thread-pools comes from the user-land created IO Completion ports
    - Researching this topic currently to determine potential injection without elevation
* Dynamic Code Prohibited (DCP)
  + Prevents process from loading non-Microsoft signed DLLs
    - Can significantly reduce EDR visibility and analysis

Injection via Waiting-Thread

Evasion is not only about stealthy execution of malware, but also avoiding alarms being raised when a process’s integrity is compromised and the application crashes or becomes unstable. This technique avoids creating a new thread or using an existing thread to explicitly execute code via API. Instead, indirect execution is done by overwriting the stack pointer within an identified waiting thread. The easiest approach for stability is to use a thread with the wait reason of WrQUEUE, which is awaiting the KQUEUE kernel object that manages queues of IRPs. The reason this is so effective, is that regardless of where the ‘wait’ is occurring, the wrappers for the function being called do not create stack frames. The first instruction within these wrapper functions (NtRemoveIOCompletion and NtWaitForWorkViaWorkerFactory) is the return address for the stack pointer upon completion of the syscall. This return address and current register values can be stored and overwritten with the address of the code to be executed. When one of these two wrapper syscalls gets called from main after modification, the true return address and register values are stored, execution of the syscall continues until normal, and upon completion E|RSP jmps to the malicious code and performs execution. Upon exit, the original return address is jumped to and normal execution flow continues.

Why This is a Big Deal

One issue covered in EDR Underpinnings part 1 is the complexity required to achieve process injection in 2025 with hooking of essential injection related APIs being integral to any solid EDR currently being shipped. The previous explanation can better be understood with an analogy:

Railroad tracks are required to move a train along to its destination. The train itself has no concept of space or the ability to ascertain if trajectory corrections are expected or indicative of a problem. Processes and threads can be thought of as trains and their engines. Without an engine, the train sits idle. Without tracks, the train derails and crashes, terminating its movement. The conductor (like an EDR hook or monitoring solution) uses what they know about previous trips taken to make decisions on whether to allow the train to continue moving, or to halt its course. From previous journeys, the conductor knows to halt the train if a switch is seen in the track as bandits have used them (legitimate usage if given previous notice by the railroad company) to stealthily steal cargo while the train is moving. Bandits, unbeknownst to the conductor, have decided to nix switching in lieu of simply moving the tracks. They carefully mark the location of removed tracks to replace them and continue on the trains original course once theft has concluded. The conductor – expecting the train to continue moving without a deviation – does not notice a bend in the horizon. The train continues onward and follows the deviation created by the bandits, who successfully rob the train and allow for track replacement to return the train to its original course. The conductor is none the wiser.