


# Studying “Next Generation Malware” - NightHawk’s Attempt At Obfuscate and Sleep

---

 [web.archive.org/web/20220505170100/https://suspicious.actor/2022/05/05/mdsec-nighthawk-study.html](https://web.archive.org/web/20220505170100/https://suspicious.actor/2022/05/05/mdsec-nighthawk-study.html)

Austin Hudson

May 5, 2022

May 5, 2022 • Austin Hudson

Share on:

Over the last year and a half, I’ve often seen mentions of a self-proclaimed “next generation malware” of the name NightHawk. Ordinarily, I’d know most of those claims tend to be nothing more than hubris, and choose to ignore it, but, I get bored. As such, I’ve chosen to start analyzing and tearing about the malware based on samples I acquired via VirusTotal, a hub which contains a plethora of commercial, closed-source, and open source samples. This research is done on my own time, and is not associated with anyone other than myself. I’ve torn about other similiar malware such as Beacon from Cobalt Strike.

TLDR: A very simple, yet effective technique. Can this be replicated with ease? Yes! Was it something new? Fortunately, no. A little dissapointed? A bit.

## Understanding its PE-SIEVE / Moneta Evasion

---

A while back, a friend of mine notified me about a video which proclaimed that it was capable of circumventing [Hasherezade’s memory scanning tool PE-SIEVE](#), as well as [Forrest’s Moneta](#), something that peaked my interest, as I had developed a similiar capability a few years prior, to improve the original research named [Gargoyle](#) for x86/x64/WOW64.

At first, I was intrigued about the technique. Was there perhaps an easier method that I had missed? Fortunutely for me, not so much. Further study revealed that it supported numerous sleeping methods, such as leveraging `NtSignalAndWaitForSingleObject` by notifying an event, then awaiting on the current process while remaining non-alertable, or by leveraging `NtWaitForSingleObject` to await on the current process object as non-alertable.

```
Evt = CreateEventW( NULL, 0, 0, NULL );
```

```
if ( Evt != NULL ) {  
    Nst = NtSignalAndWaitForSingleObject( Evt, NtCurrentProcess(), FALSE, &Del );  
}
```

*Pseudo-C demonstrating the underlying concept of using NtSignalAndWaitForSingleObject*

Nothing new, fortunately. Its used as a means of circumventing the check on Hunting-Sleeping-Beacons to hide detections based on the `DelayExecution` wait status for threads. Its certainly sufficient, and something I was doing myself.

However, its evasion to hide traces of itself in memory are a little different. It first ( I believe ) leverages `RtlCaptureContext` as a callback to `kernel32!CreateTimerQueueTimer` with an argument to a context structure to capture the return address value to return to. A call below would replicate the similiar behavior:

```
if ( CreateTimerQueueTimer( &TimerObj, TimerQueue, RtlCaptureContext, ContextStruct,
0, 0, WT_EXECUTEINTIMERTHREAD ))
{
    WaitForSingleObject( TimerObj, 50 );
}
```

The callback will promptly be executed within a new thread, and on x64, RSP will be filled with the complete return address. After this has completed, NightHawk then fills in the function-call CONTEXT structures for `VirtualProtect`, `SuspendThread`, `GetThreadContext`, `SetThreadContext`, and `ResumeThread`. These context structures allow NightHawk to redirect execution to the specified functions with full control over `RSP`, `RCX`, `RDX`, `R8`, `R9`.

It does not have any further control over any functions that are over 4 arguments on x64, due to the usage of timers and heavy reliance on its termination callback from calling the specified callback. Furthermore, it adjusts `RSP` back to 8 bytes to accomodate the offset created by calling `RtlCaptureContext`.

```
__builtin_memcpy( & ContextVirtualProtect, & ContextStructure, sizeof(
ContextStructure ) );
ContextVirtualProtect.Rsp -= 8;
ContextVirtualProtect.Rip = VirtualProtect
ContextVirtualProtect.Rcx = NightHawkImageBase;
ContextVirtualProtect.Rdx = NightHawkImageLength;
ContextVirtualProtect.R8 = PAGE_READWRITE
ContextVirtualProtect.R9 = &OriginalProtect;

__builtin_memcpy( & ContextSuspendThread, & ContextStructure, sizeof(
ContextStructure ) );
ContextSuspendThread.Rsp -= 8;
ContextSuspendThread.Rip = SuspendThread;
ContextSuspendThread.Rcx = MyOriginalThreadHandle;
```

Once the call has been built, it then attempts to queue it using the same timer function with a callback set to either `NtContinue` or `RtlRestoreContext` which will leverage the context structure to execute the specified function with the set registers and return safely without causing a potential crash - While promptly adjusting the timing period between the calls to avoid any functions from being queued out of order.

```
LARGE_INTEGER Time;

RtlSecureZeroMemory( &Time, sizeof( Time ) );

Time.QuadPart += 100;
CreateTimerQueueTimer( TimerObj, TimerQueue, RtlRestoreContext,
&ContextVirtualProtect, &Time, 0, WT_EXECUTEINTIMERTHREAD );

Time.QuadPart += 100;
CreateTimerQueueTimer( TimerObj, TimerQueue, RtlRestoreContext,
&ContextSuspendThread, &Time, 0,
WT_EXECUTEINTIMERTHREAD );
```

As a result, the timer queue will first execute ContextVirtualProtect, before ContextSuspendThread to avoid issues of them conflicting or running before the other has completed safely. But first! To avoid issues of the calls running before it has reached a waitable state, it will then use WaitForSingleObject to block until the timer queue has completed. A very similiar style to how I accomplished my Foliage/Gargyoyle chain a few years ago, yet just as effective.

I will be sharing my PoC in the coming days replicating their varition to completion. For those of you who utilize Cobalt or other similiar toolsets, and dont want to waste \$30K, fortunately, this can be achieved with very little effort and tooling using something like a custom Reflective Loader, which I will be re-posting my Titan variant with their implementation contained.

---