Driver opens a few values and queries them from the RegistryPath of the driver but I did not look at it yet. DriverIoControl is registered with quite a lot of IOCTL codes. Most IOCTL codes just triggers another unknown function that’s different for each IOCTL except one: 0xB200C

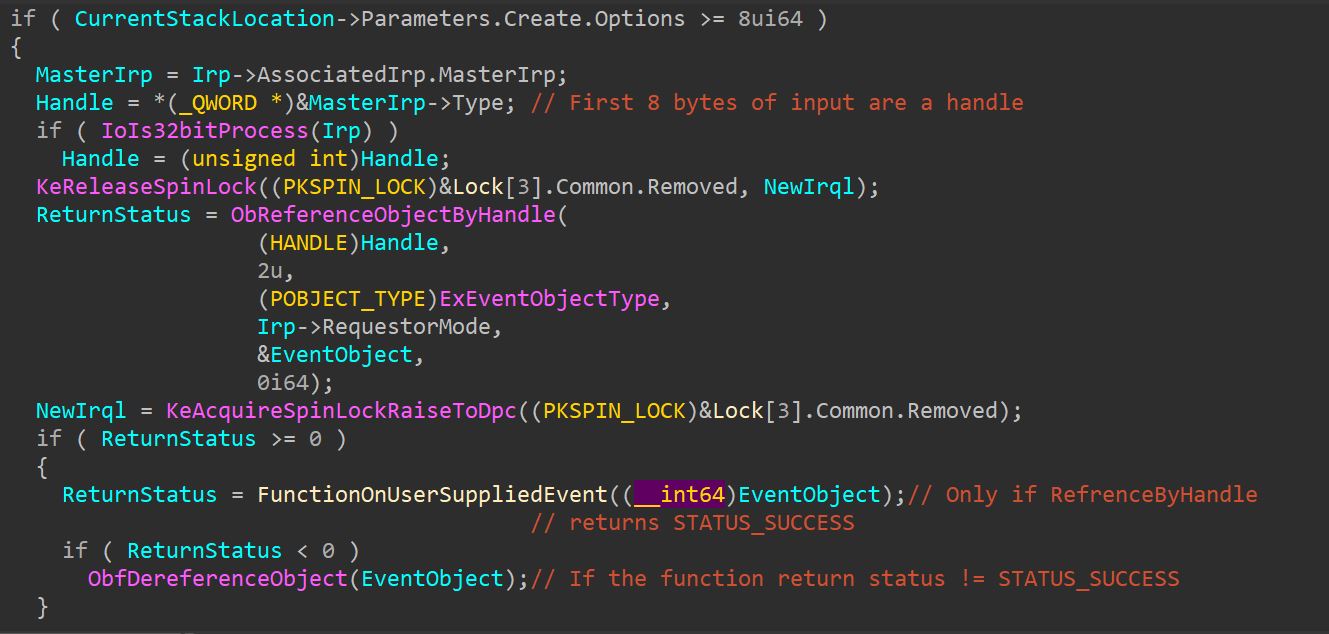
IOCTL 0xB200C:

Gets an 8 byte buffer as parameter and this value is interpreted as an HANDLE. This handle is passed to ObReferenceObjectByHandle as an event handle. Parameters:

* DesiredAccess = EVENT\_MODIFY\_STATE (2): valid
* HandleInformation = NULL: valid in this case
* All other parameters are pointers to defined structs / specifiers of the specific operation so they are all valid as well

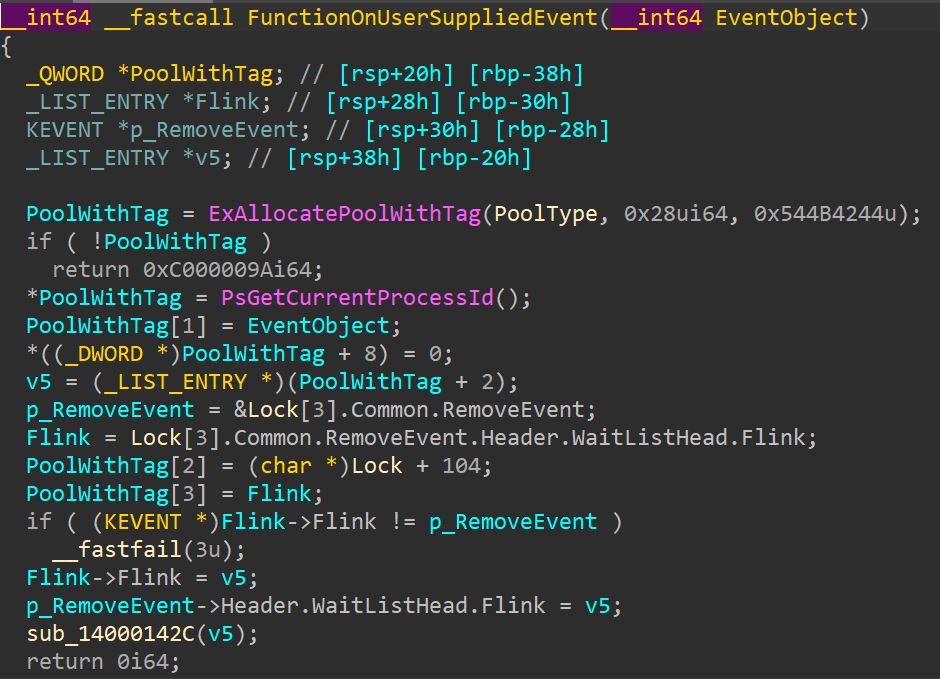
If this operation succeeds, some function is called and this function receives the event object as parameter. If that operation fails, ObfDereferenceObject is called on the event object (valid flow). If it succeeds – ObfDereferenceObject is not called on the event

**POSSIBLE PRIMITIVE** – if I provide the same event handle twice, 2 object references will be created and so forward. If there is a limitation to the references to the object I can maybe DOS the access to that specific event / even BSoD the whole machine. This might be a problem because the called function only fails if no NonPaged memory is available



FunctionOnUserSuppliedEvent called inside the IOCTL:

this function gets the event object as a parameter, performs a few operations on the event object and eventually returns 0 (STATUS\_SUCCESS). This only gets prevented if there isn’t enough nonpaged memory for the allocated buffer (each allocated instance is 0x28 bytes so it should not fail for the first several uses). So – this means I can use the primitive I mentioned earlier to reference the same object and not dereference the same object for a few times so eventually this might create a problem / somehow provide invalid parameters to reference-handle so it will create some attack vector



**summary of primitives until now:**

1. if I provide the same event handle twice, 2 object references will be created and so forward. If there is a limitation to the references to the object I can maybe DOS the access to that specific event / even BSoD the whole machine. This might be a problem because the called function only fails if no NonPaged memory is available so the function should constantly succeed
2. after reversing the ObRefrenceByHandle function, I can maybe find some weird execution flow with event handles in a way I can create an attack vector with my custom parameters

Now I will try to get into the other IOCTLs before examining ObReferenceObjectByHandle internally / observing the access to registry key values of the driver

IOCTL 0xB2044:

Gets a PVOID parameter (8 byte value) as a parameter, this is expected to be a pointer to an 8 byte buffer and this buffer is just filled with 1 for the first 4 bytes and 6 for the second 4 bytes.

**summary of primitives until now:**

1. …
2. …
3. If I provide an invalid address this might cause a BSoD, as this is written into a buffer that is not verified with ProbeForRead/Write
4. In a similar fashion, if I provide a kernel address I would be able to overwrite information over that kernel address with an 8 byte primitive. The value written cannot be controlled by me but the target address is (probably, because there aren’t any address verifications there might’ve been some done before this function as a simple UM address would crash this function) תמונה שמכילה טקסט, צילום מסך, גופן

   התיאור נוצר באופן אוטומטי

IOCTL 0xB204C:

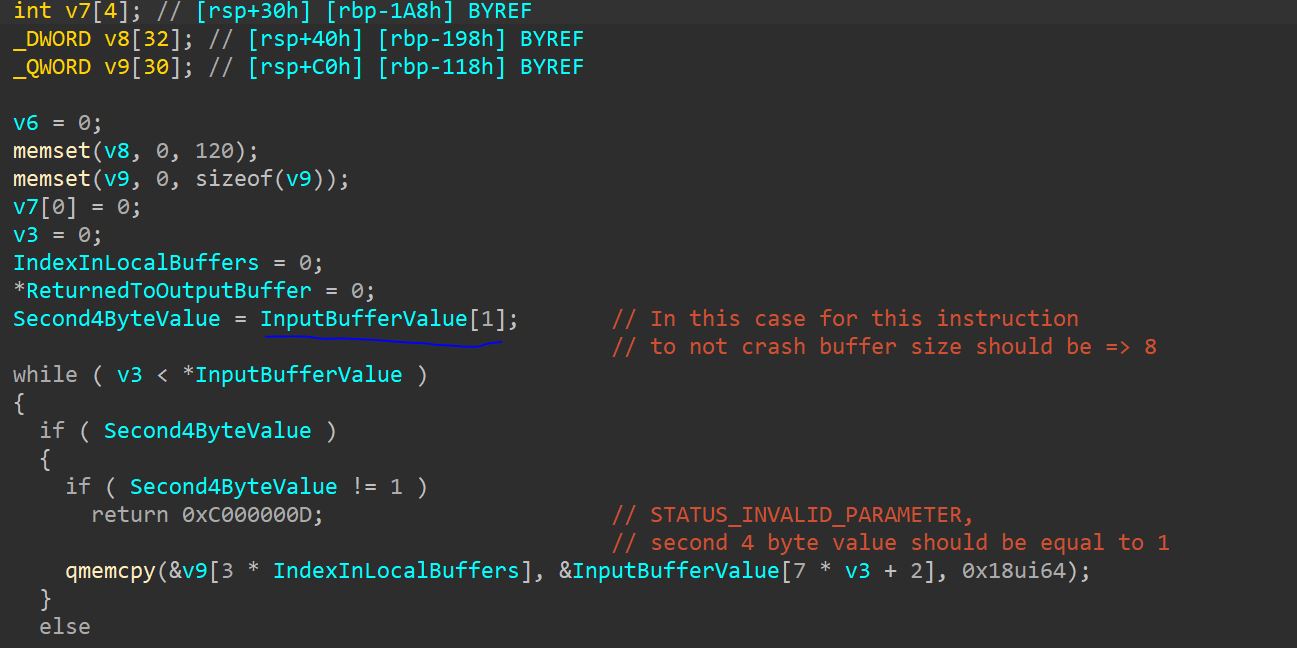
This function probably expects METHOD\_BUFFERED, which means:

1. Only one buffer is validated to be != NULL (which returns invalid parameter if not equal)
2. Two sizes from the IRP structure are validated to be atleast 4 bytes long, which means that one is the input size and one is the output size and like the passing expected in METHOD\_BUFFERED -> I/O buffers and sizes should be the same.

Then accordingly, the 4 byte value at the start of the supplied buffer is checked. From the validation done we can understand that the value should be <= 1024 and >= 0. This value is interpreted as a simple int type so we can freely control the value inside when calling the IOCTL. Then a calculation is done with the I/O buffer size (which has to be 4 or more, but 4 is the general answer):

* 28 \* (\*InputBuffer - 1) + 32 needs to be equal to the buffer size (which is 4 or more bytes). With the calculation when buffer size is exactly 4 bytes – value in input buffer should be 0

If the value is non-zero (which means buffer size is bigger than 4) another function is called and gets provided with the InputBuffer value and a pointer to a local int variable. If InputBuffer=0/buffersize=4 – the value of the second parameter stays zero. This value gets assigned into the buffer (METHOD\_BUFFERED, I/O buffers are the same) and so is returned in the output buffer. The value returned to the caller in the output buffer from the function is some kind of counter, a few calculations are done in the function but it might have a vulnerability because if the buffer size is only 4 then the instruction that assigns 0 to InputBuffer[1] will try to write into non-existent memory. If the address can be controlled for the input buffer I can cause a write into an invalid address



**summary of primitives until now:**

1. …
2. …
3. …
4. …
5. If the buffer size is only 4 then the instruction that assigns 0 to InputBuffer[1] will try to write into non-existent memory. If the address can be controlled for the input buffer I can cause a write into an invalid/valid kernel address but its probably not useful

IOCTL 0xB2040:

Some data from the lock information gets copied into the user supplied buffer that is 20 bytes long or more. There is not really any other interaction with memory in this dispatch function so I will skip it for now

Other IOCTLs cannot be manipulated or controlled in any way by my parameters, so this is the finally summary of attack vectors on this driver:

**summary of primitives until now:**

1. if I provide the same event handle twice, 2 object references will be created and so forward. If there is a limitation to the references to the object I can maybe DOS the access to that specific event / even BSoD the whole machine. This might be a problem because the called function only fails if no NonPaged memory is available so the function should constantly succeed (0xB200C)
2. after reversing the ObRefrenceByHandle function, I can maybe find some weird execution flow with event handles in a way I can create an attack vector with my custom parameters (0xB200C)
3. If I provide an invalid address this might cause a BSoD, as this is written into a buffer that is not verified with ProbeForRead/Write (0xB2044)
4. In a similar fashion, if I provide a kernel address I would be able to overwrite information over that kernel address with an 8 byte primitive. The value written cannot be controlled by me but the target address is (probably, because there aren’t any address verifications there might’ve been some done before this function as a simple UM address would crash this function) (0xB2044)
5. If the buffer size is only 4 then the instruction that assigns 0 to InputBuffer[1] will try to write into non-existent memory. If the address can be controlled for the input buffer I can cause a write into an invalid address (0xB204C)

**Exploitation of the attack vectors:**

“The reference count of an object is illegal for the current state of the object. Each time a driver uses a pointer to an object, the driver calls a kernel routine to increase the reference count of the object by one. When the driver is done with the pointer, the driver calls another kernel routine to decrease the reference count by one.

Drivers must match calls to the routines that increase (reference) and decrease (dereference) the reference count. This bug check is caused by an inconsistency in the object's reference count. Typically, the inconsistency is caused by a driver that decreases the reference count of an object too many times, making extra calls that dereference the object. This bug check can occur because an object's reference count goes to zero while there are still open handles to the object. It might also occur when the object's reference count drops below zero, whether or not there are open handles to the object.

“ – msdn <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0x18--reference-by-pointer>

The idea is to call RefrenceByHandle() with the same handle enough times so the counter of refrences will eventually get to zero, then this should BSoD the whole system and crash everything

**Primitive number 5:**

I actually exploited 5) and a BSoD was generated successfully, this vulnerability was actually discovered in 2010 and was not fixed since – “zero day” 😊

<https://www.exploit-db.com/exploits/15103>

it was actually “fixed”, meaning the exploit I mentioned is different than my exploit by the buffer size and the value stored inside the buffer, but it is still possibly exploitable. I will cover all possible values between 0-8 (8 not included as the BSoD will not be triggered):

buffer size = 4 -> value inside first 4 bytes of buffer should be zero, will not pass the last check to call the vulnurable function, not exploitable

buffer size = 5 -> value should be equal to 1, non-zero value, will trigger vulnurable function

buffer size = 6 -> the same (value = 1, exploitable)

buffer size = 7 -> the same (value = 1, exploitable)

buffer size = 8 and above -> not exploitable with this exploit

UPDATE: buffer sizes 5-7 will not work, when putting the value 1 or higher as the input value, v6 will equal to buffer sizes from 32 and above, and a value of 0 which will not go into the vulnurable function. Summary: not vulnurable

**primitive number 3 and 4:**

because the size of the buffer is validated (no outofbound write can be triggered) I still checked if I can control the input buffer address so the address will not be NULL, but it can still be an invalid non-null address. This should have probably worked (also the same with 5) as an invalid address and a buffer size provided as 4) but VMWare made it so I cannot get a handle to the driver as no symbolic link is created. Summary: vulnerable but is not exploitable because I cannot communicate with the driver

I did not continue to exploit 1 and 2 as I cannot exploit anything related to this driver