I2OMGMT Driver Impersonation Attack

Justin Seitz
justin@immunityinc.com
Immunity Inc. 2008
**Introduction**

This paper stems from the notes, and general pain I had to go through in order to write the exploit for the i2omgmt.sys local privilege escalation attack. This vulnerability was reported by iDefense and discovered by Reuben Santamarta, and it is not your typical kernel-mode overflow which makes it a unique bug (hence the paper).

Let's take a look at the advisory and see what we can find, and then we will walk through the steps that I took in order to get a proper local privilege escalation attack working. The advisory ([http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=699](http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=699)) mentions a couple of key pieces of information:

- a device name `\\I2OExec`
- we are going to use an IOCTL to send information to the device
- we can forge a `DEVICE_OBJECT` at some point

Now that we have some basic information on the bug we are hunting down, let's start figuring out where the vulnerability lies.

NOTE: I am leaving out significant portions of the dynamic analysis that was done for brevity's sake. As well, I already assume that you are aware of how to send IOCTLS to device drivers, and all of that jazz.

**Finding the Bug**

The first step is to determine what IOCTL call looks like it is going to be useful to us, warm up your favourite disassembler and crack open i2omgmt.sys

NOTE: If you don't have this in C:\WINDOWS\system32\drivers then unpack it from C:\WINDOWS\Driver Cache\i386\sp2.cab then reboot.

I use Immunity Debugger for doing the driver analysis throughout, so installing it from [http://debugger.immunityinc.com](http://debugger.immunityinc.com) may be a good idea for following along. As well, in the figures displayed I didn't have room to include any decoding information, so using the debugger to follow along will show you quite a bit more detail than what my word processor permits.
The first thing we need to find is the primary dispatch function for handling IOCTLs to this device. The entry point of the driver is shown in Figure 1:

![Figure 1: i2omgmt.sys entry point.](image)

Nothing amazing here, does some setup then unconditionally jumps to 0x00010FAC. Let’s head there and in the first basic block we will see a key piece of information, Figure 2 shows the code:

![Figure 2: Function pointer setup at 0x00010FCE for IOCTL dispatch.](image)

The instruction at 0x00010FCE:

```
MOV DWORD PTR DS:[ESI+70], i2omgmt.00010D6C
```

is setting the function pointer for handling IOCTLs for the I2OExec device and it is here that we have
to begin our investigation. When we disassemble the first basic block of the dispatch function (Figure 3) we see the first IOCTL code calculation at \textcolor{purple}{0x00010D9A}:

![Figure 3: IOCTL dispatch first basic block.](image)

So at \textcolor{purple}{0x00010D9A} it is subtracting \textcolor{purple}{0x222E80} from our IOCTL code and if we look further into the function code we see that from there it continues subtracting our IOCTL code until it determines what action to take (of course once the IOCTL code calculation is equal to zero). This is a classic switch statement, and is the most common way for IOCTL dispatch routines to handle separate IOCTL calls. If you are viewing this under the debugger, full switch decoding will be available. We will briefly revisit this later in order to send the proper IOCTL code to trigger the bug.

Before we start tearing through all of the possible IOCTL codes in the dispatch, let's first have a quick glance at all CALLs that can occur that are a result of a specific IOCTL code. We can then determine the various exit points for the IOCTL codes and begin trying to track down the vulnerability.

Highlighting the \textcolor{purple}{0x00010D9A} function, and then hitting CTRL+K in Immunity Debugger will show us a call tree as show in Figure 4 below:

![Figure 4: Call tree from IOCTL dispatch.](image)

Investigating the first procedure (\texttt{i20mgmt.0010326}) in the disassembler quickly shows us that it is
a logging function (complete with a DbgPrint() call for us who are debugging). We aren't interested in
this particular function call for our purposes so we move on. Investigating the \texttt{i2omgmt.000106EC}
procedure we find something a little more interesting, let's look at the new call tree in Figure 5:

![Call tree](image)

\textbf{Figure 5: Call tree from procedure i2omgmt.000106Ec}

We see a call to the aforementioned logging function, and a new call to \texttt{i2omgmt.00010386}. Let's
take a look at this function:
Now we are seeing something interesting. At (1) we see that we are building a device IO control request to be sent to another driver. Shortly after that at (2) we see the call to the ntoskrnl.IofCallDriver function. We don't see any static constants, or any reference to a driver name that gets called, so we assume that whatever IofCallDriver() is calling must be passed to it somehow. Backtracking through the functions leading up to this, we also know that we control significant pieces of data on the way up to this call, which all originates from the IOCTL we send to the driver.
Let's take a look at the prototype for IofCallDriver; from MSDN:

```c
NTSTATUS IofCallDriver(
    IN PDEVICE_OBJECT DeviceObject,
    IN OUT PIRP Irp
);
```

We can see at this point that IofCallDriver is going to take in a pointer to a `DEVICE_OBJECT` struct, and a pointer to a valid IRP record. When looking back, we remember that the advisory is mentioning that we can forge a `DEVICE_OBJECT` struct to achieve code execution. Let's just make sure we understand what this call is actually looking for, Figure 7 shows the code listing for IofCallDriver.

![Figure 7: Disassembly of nt!IofCallDriver](image)

The first thing we notice is it decrements the number stored at `[EDX+23]` and then tests to make sure it doesn't equal zero. If it does equal zero it throws the bugcheck code 0x35 and the world comes crashing down. From MSDN the bug check code 0x00000035 has a value of: NO_MORE_STACK_IRP_LOCATIONS and is exclusively used to indicate that a IofCallDriver call has failed because there are no more stack locations available for the request packet. If the stack size check passes, we move on to the final call at `0x00416DF5`:

```assembly
CALL DWORD PTR DS:[ESI+EAX*4+38]
```

So this is making a call to a pointer stored at offset 0x38 in some struct. Let's take a look at the `DEVICE_OBJECT` struct and see what we have:
Well we don't see anything at an offset of 0x38 but we do see a DRIVER_OBJECT pointer at 0x8 and at 0x30 we see a StackSize member that we will make sure has a high enough value to bypass the bugcheck. Both are valuable pieces of information. Let's take a look at what the DRIVER_OBJECT holds:

```
nt! DRIVER_OBJECT
+0x000 Type             : Int2B
+0x002 Size             : Int2B
+0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
+0x008 DriverObject     : Ptr32 DRIVER_OBJECT
+0x010 NextDevice       : Ptr32 _DEVICE_OBJECT
+0x014 AttachedDevice   : Ptr32 _DEVICE_OBJECT
+0x018 CurrentIrp       :Ptr32 _IRP
+0x01c Timer            : Ptr32 _IO_TIMER
+0x020 Flags            : Uint4B
+0x024 Vpb              : Ptr32 _VPB
+0x028 DeviceExtension  : Ptr32 Void
+0x02c DeviceType       : Uint4B
+0x030 StackSize        : Char
+0x034 Queue            : __unnamed
+0x05c AlignmentRequirement : Uint4B
+0x060 DeviceQueue      : __KDEVICE_QUEUE
+0x074 Dpc              : __KDPC
+0x094 ActiveThreadCount : Uint4B
+0x098 SecurityDescriptor : Ptr32 Void
+0x09c DeviceLock       : __KEVENT
+0x0ac SectorSize       : Uint2B
+0x0ae Spare1           : Uint2B
+0x0b0 DeviceObjectExtension : Ptr32 _DEVOBJ_EXTENSION
+0x0b4 Reserved         : Ptr32 Void
+0x0b8 MajorFunction    : [28] Ptr32 long
```
Well look at that! At 0x38 we see a pointer to MajorFunction, which we can naturally assume is a function pointer and one that we control. So the advisory is both right and wrong, we can forge a **DEVICE_OBJECT** but it's really the **DRIVER_OBJECT** that gets the member function called and ultimately runs our shellcode. So we now have a fairly accurate picture of what kind of input we need to craft to get actual shellcode execution, Figure 8 below depicts a general layout of the kernel objects.

**Figure 8: Layout of fake kernel objects.**
The interesting thing is that we get to create all of these objects in userland buffers, and point the IofCallDriver call at them, which runs our kernel mode shellcode. Cool, let's move on.

**Crafting the Input**

Now we have to determine how to get full code execution out of being able to forge a DEVICE_OBJECT struct. Looking back at our IOCTL calculation if you follow the subtractions leading up to the call being made to `i2omgmt.00010386` (which ultimately calls the IofCallDriver routine), we see that the final IOCTL code we need is 0x222F80. From there we now have to look at what we have to pass in for input so that ultimately we get shellcode execution.

In order for us to properly get this working we have to work in reverse. The pseudo-steps that we need to take are:

1. Allocate memory and store shellcode.
2. Forge the fake DRIVER_OBJECT and point its MajorFunction member at the shellcode.
3. Forge the fake DEVICE_OBJECT, set its stack size to something high and point its DriverObject member at the fake DRIVER_OBJECT we created in step 2.
4. Create two pointers to the DEVICE_OBJECT which get validated as real pointers and nothing more.

The first step is straightforward, as we have specific ring0 shellcode generators directly in CANVAS. Step 2 is also straightforward, just create a string buffer, and set the 0x38 offset as a pointer to the address where we stored the shellcode.

Step 3 appears to be innocuous, it requires a bit more work than first anticipated. The reason we have extra work is because of the code snippet shown in Figure 9 which is a basic block taken from our i2omgmt.000106EC function:

```
0:00010739:
  MOV ECX,DWORD PTR DS:[EBP+2C]
  LEA EAX,DWORD PTR DS:[EBP+EDX*2]
  MOV EAX,DWORD PTR DS:[ECX+EAX*4+8]
  MOV DWORD PTR SS:[EBP-10],EAX
  MOV EAX,DWORD PTR SS:[EBP-4]
  ADD EAX,ESI
  LEA EDI,DWORD PTR DS:[EAX+50]
  PLUSH 064F3249
  PUSH EDI
  PLUSH 0
  MOV DWORD PTR SS:[EBP-20],EDI
  CALL DWORD PTR DS:[<&ntoskrnl.ExAllocatePoolWithTag> ntoskrnl.ExAllocatePoolWithTag]
  MOV EDX,EAX
  TEST EDX,EDX
  MOV DWORD PTR SS:[EBP-C],EDX
  JNZ SHORT i2omgmt.00010781
```

*Figure 9: Basic block containing troublesome pointer math.*
So at the head of this basic block, the following two instructions are of importance.

```
LEA EAX, DWORD PTR DS:[EDI+EDI*2]
MOV EAX, DWORD PTR DS:[ECX+EAX*4+8]
```

In this case we control the value of EDI and when these two instructions finish their calculations, they load the final value into a local variable using:

```
MOV DWORD PTR SS:[EBP-10], EAX
```

This local variable is our DEVICE_OBJECT, so we need to first figure out a sane value for EDI so that it will result in a valid pointer to the DEVICE_OBJECT that we control. Kostya contributed the math to help me out:

```
device_object_ptr = (brute_device_address - 8) / 12
```

The brute_device_address variable is the final address we want the calculations to equal. Why the brute in the variable name? In order for this address to be valid we have to make sure it is divisible by 12. So we allocate a large amount of memory, and then iterate through it until we find an address that is divisible by 12. Once we find the appropriate address we use it as the place to store our fake DEVICE_OBJECT and using the formula above, when we pass in device_object_ptr (EDI in the assembly code above) which will be calculated out to point to our DEVICE_OBJECT. If this pretty pointer dance isn't done correctly, we blue screen and it's game over!

The final step is simple enough, we create two pointers that get validated against each other, and both point to the DEVICE_OBJECT that we created in memory. I can't be certain why it expects the pointers in this fashion but it does, it's either an undocumented MS struct or a proprietary way for this driver to receive IOCTLs.

**Conclusion**

This was an interesting bug to work with, as the forging of these kernel objects in userland requires very careful setup, calculation, and there were no overflows involved. I definitely suspect that everyone will be taking a closer eye to this type of attack, as many drivers will accept unfettered IOCTL requests from userland without first checking the validity (not just the length or size) of the objects being passed in. The source for this exploit is part of the CANVAS tree, and the exploit is well documented throughout the code if any further clarification is needed.

There were a lot of lessons learned throughout coding this exploit, and in the near future you should see a fresh release of the IOCTL fuzzer we are continually improving on at Immunity. The fuzzer also comes with an automated method for calculating IOCTL codes using Immunity Debugger which is extremely useful for reveresing these types of bugs out.

Thanks again to Kostya, Nico, and the rest of the team at Immunity for their help. Congratulations to Reuben for discovering such a neat bug! For any questions, comments, etc. please contact me, and the rest of the team can be reached at support@immunityinc.com