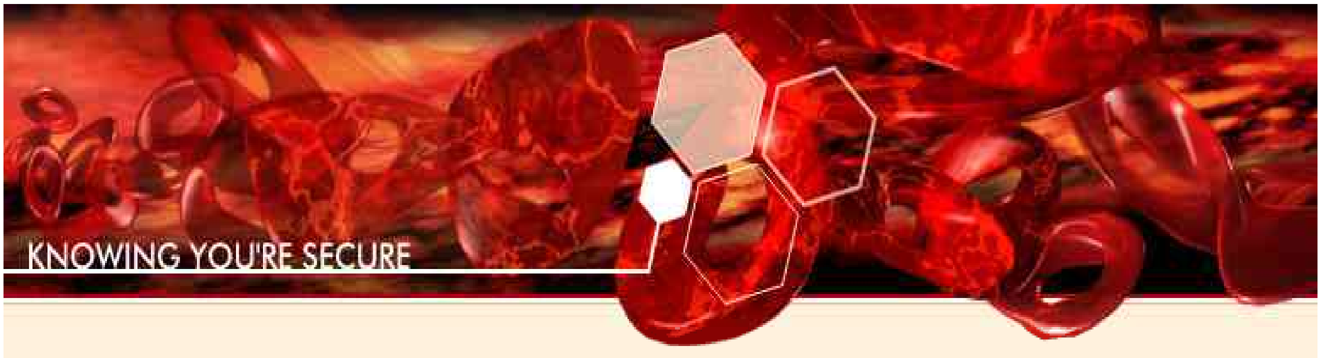


I2OMGMT Driver Impersonation Attack

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) 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 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:

```
00011785  $ 8BFF          MOV EDI,EDI
00011787  . 55            PUSH EBP
00011788  . 8BEC          MOV EBP,ESP
0001178A  . A1 18160100   MOV EAX,DWORD PTR DS:[11618]
0001178F  . 85C0          TEST EAX,EAX
00011791  . B9 40BB0000   MOV ECX,0BB40
00011796  . 74 04         JE SHORT i2omgmt.0001179C
00011798  . 3BC1          CMP EAX,ECX
0001179A  . 75 23         JNZ SHORT i2omgmt.000117BF
0001179C  > 8B15 DC120100 MOV EDX,DWORD PTR DS:[<&ntoskrnl.KeTickCount>]
000117A2  . B8 18160100   MOV EAX,i2omgmt.00011618
000117A7  . C1E8 08       SHR EAX,8
000117AA  . 3302          XOR EAX,DWORD PTR DS:[EDX]
000117AC  . 25 FFFF0000   AND EAX,0FFFF
000117B1  . A3 18160100   MOV DWORD PTR DS:[11618],EAX
000117B6  . 75 07         JNZ SHORT i2omgmt.000117BF
000117B8  . 8BC1          MOV EAX,ECX
000117BA  . A3 18160100   MOV DWORD PTR DS:[11618],EAX
000117BF  > F7D0          NOT EAX
000117C1  . A3 14160100   MOV DWORD PTR DS:[11614],EAX
000117C6  . 5D            POP EBP
000117C7  . E9 E0F7FFFF   JMP i2omgmt.00010FAC
```

Figure 1: i2omgmt.sys entry point.

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:

```
00010FAC   > 8BFF          MOV EDI,EDI
00010FAE  ┌. 55            PUSH EBP
00010FAF  │. 8BEC          MOV EBP,ESP
00010FB1  │. 83EC 70       SUB ESP,70
00010FB4  │. A1 18160100   MOV EAX,DWORD PTR DS:[11618]
00010FB9  │. 53            PUSH EBX
00010FBA  │. 8945 FC       MOV DWORD PTR SS:[EBP-4],EAX
00010FBD  │. 56            PUSH ESI
00010FBE  │. 8B75 08       MOV ESI,DWORD PTR SS:[EBP+8]
00010FC1  │. B8 06030100   MOV EAX,i2omgmt.00010306
00010FC6  │. 33DB          XOR EBX,EBX
00010FC8  │. 8946 38       MOV DWORD PTR DS:[ESI+38],EAX
00010FCB  │. 8946 40       MOV DWORD PTR DS:[ESI+40],EAX
00010FCE  │. C746 70 6C0D0 MOV DWORD PTR DS:[ESI+70],i2omgmt.00010D6C
00010FD5  │. C746 78 C40B0 MOV DWORD PTR DS:[ESI+78],i2omgmt.00010BC4
00010FDC  │. C746 34 CE050 MOV DWORD PTR DS:[ESI+34],i2omgmt.000105CE
00010FE3  │. 391D 20160100 CMP DWORD PTR DS:[11620],EBX
00010FE9  │. 895D B4       MOV DWORD PTR SS:[EBP-4C],EBX
00010FEC  │. C645 C7 01    MOV BYTE PTR SS:[EBP-39],1
00010FF0  │. 74 0A         JE SHORT i2omgmt.00010FFC
```

Figure 2: Function pointer setup at 0x00010FCE for IOCTL dispatch.

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 `0x00010D9A`:

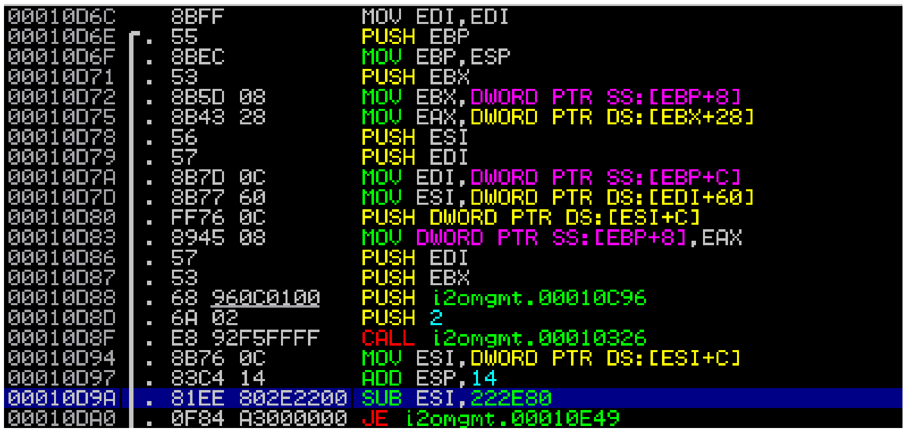


Figure 3: IOCTL dispatch first basic block.

So at `0x00010D9A` it is subtracting `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 `0x00010D9A` function, and then hitting CTRL+K in Immunity Debugger will show us a call tree as show in Figure 4 below:

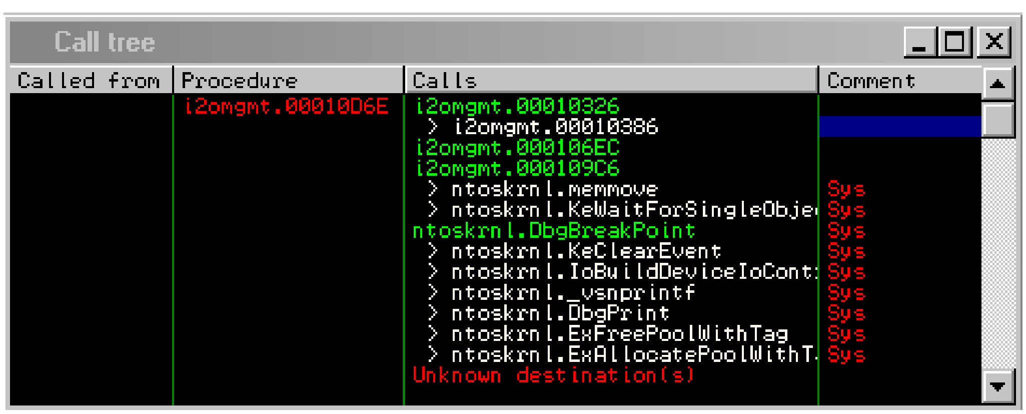


Figure 4: Call tree from IOCTL dispatch.

Investigating the first procedure (`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 `i2omgmt.000106EC` procedure we find something a little more interesting, let's look at the new call tree in Figure 5:

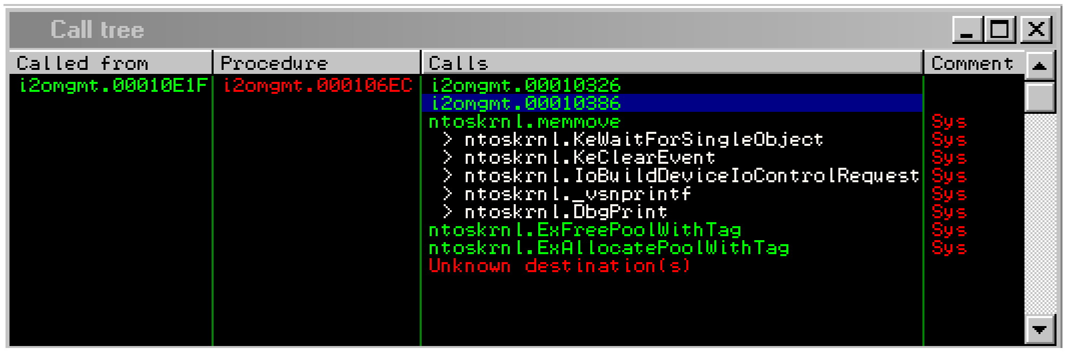


Figure 5: Call tree from procedure i2omgmt.000106Ec

We see a call to the aforementioned logging function, and a new call to `i2omgmt.00010386`. Let's take a look at this function:

```
00010386 ┌$ 8BFF           MOV EDI,EDI
00010388 │. 55             PUSH EBP
00010389 │. 8BEC           MOV EBP,ESP
0001038B │. 51             PUSH ECX
0001038C │. 51             PUSH ECX
0001038D │. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
00010390 │. 8B40 28        MOV EAX,DWORD PTR DS:[EAX+28]
00010393 │. 53             PUSH EBX
00010394 │. 33DB           XOR EBX,EBX
00010396 │. 395D 0C        CMP DWORD PTR SS:[EBP+C],EBX
00010399 │. 75 07          JNZ SHORT i2omgmt.000103A2
0001039B │. B8 C00000C0    MOV EAX,C00000C0
000103A0 │. EB 72          JMP SHORT i2omgmt.00010414
000103A2 │> 56             PUSH ESI
000103A3 │. 8D70 14        LEA ESI,DWORD PTR DS:[EAX+14]
000103A6 │. 56             PUSH ESI
000103A7 │. FF15 98120100  CALL DWORD PTR DS:[<&ntoskrnl.KeClearEvent>]
000103AD │. 8B45 10        MOV EAX,DWORD PTR SS:[EBP+10]
000103B0 │. 8B10           MOV EDX,DWORD PTR DS:[EAX]
000103B2 │. 8B48 18        MOV ECX,DWORD PTR DS:[EAX+18]
000103B5 │. 03CA           ADD ECX,EDX
000103B7 │. 8D55 F8        LEA EDX,DWORD PTR SS:[EBP-8]
000103BA │. 52             PUSH EDX
000103BB │. 56             PUSH ESI
000103BC │. 53             PUSH EBX
000103BD │. 51             PUSH ECX
000103BE │. 50             PUSH EAX                                1
000103BF │. 51             PUSH ECX
000103C0 │. 50             PUSH EAX
000103C1 │. FF75 0C        PUSH DWORD PTR SS:[EBP+C]
000103C4 │. 68 08D00400    PUSH 4D008
000103C9 │. FF15 94120100  CALL DWORD PTR DS:[<&ntoskrnl.IoBuildDeviceIoControlRequest>]
000103CF │. 3BC3           CMP EAX,EBX
000103D1 │. 75 07          JNZ SHORT i2omgmt.000103DA
000103D3 │. B8 9A0000C0    MOV EAX,C000009A
000103D8 │. EB 39          JMP SHORT i2omgmt.00010413
000103DA │> 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
000103DD │. 57             PUSH EDI
000103DE │. 8BD0           MOV EDX,EAX
000103E0 │. FF15 90120100  CALL DWORD PTR DS:[<&ntoskrnl.IofCallDriver>]   2
000103E6 │. 8BF8           MOV EDI,EAX
000103E8 │. 81FF 03010000  CMP EDI,103
000103EE │. 75 10          JNZ SHORT i2omgmt.00010400
000103F0 │. 53             PUSH EBX
000103F1 │. 53             PUSH EBX
000103F2 │. 53             PUSH EBX
000103F3 │. 53             PUSH EBX
000103F4 │. 56             PUSH ESI
000103F5 │. FF15 8C120100  CALL DWORD PTR DS:[<&ntoskrnl.KeWaitForSingleObject>]
000103FB │. 8B7D F8        MOV EDI,DWORD PTR SS:[EBP-8]
000103FE │. EB 10          JMP SHORT i2omgmt.00010410
00010400 │> 57             PUSH EDI
00010401 │. 68 5C030100    PUSH i2omgmt.0001035C
00010406 │. 6A 01          PUSH 1
00010408 │. E8 19FFFFFF    CALL i2omgmt.00010326
0001040D │. 83C4 0C        ADD ESP,0C
00010410 │> 8BC7           MOV EAX,EDI
00010412 │. 5F             POP EDI
00010413 │> 5E             POP ESI
00010414 │> 5B             POP EBX
00010415 │. C9             LEAVE
00010416 └. C2 0C00         RETN 0C
```

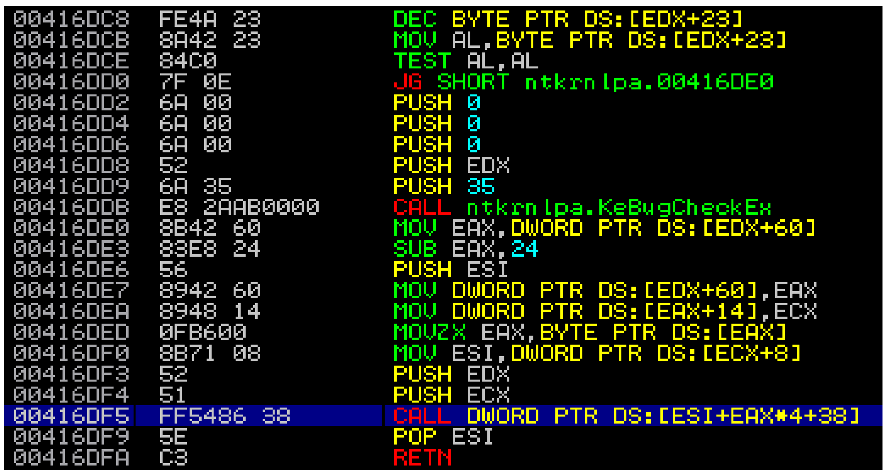
Figure 6: Disassembly of i2omgmt.00010386

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:

```
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.



```
00416DC8  FE4A 23      DEC BYTE PTR DS:[EDX+23]
00416DCB  8A42 23      MOV AL,BYTE PTR DS:[EDX+23]
00416DCE  84C0         TEST AL,AL
00416DD0  7F 0E        JG SHORT ntkrnlpa.00416DE0
00416DD2  6A 00        PUSH 0
00416DD4  6A 00        PUSH 0
00416DD6  6A 00        PUSH 0
00416DD8  52           PUSH EDX
00416DD9  6A 35        PUSH 35
00416DDB  E8 2AAB0000  CALL ntkrnlpa.KeBugCheckEx
00416DE0  8B42 60      MOV EAX,DWORD PTR DS:[EDX+60]
00416DE3  83E8 24      SUB EAX,24
00416DE6  56           PUSH ESI
00416DE7  8942 60      MOV DWORD PTR DS:[EDX+60],EAX
00416DEA  8948 14      MOV DWORD PTR DS:[EAX+14],ECX
00416DED  0FB600       MOVZX EAX,BYTE PTR DS:[EAX]
00416DF0  8B71 08      MOV ESI,DWORD PTR DS:[ECX+8]
00416DF3  52           PUSH EDX
00416DF4  51           PUSH ECX
00416DF5  FF5486 38    CALL DWORD PTR DS:[ESI+EAX*4+38]
00416DF9  5E           POP ESI
00416DFA  C3           RETN
```

Figure 7: Disassembly of nt!IopfCallDriver

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`:

```
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:

```
nt!_DEVICE_OBJECT
   +0x000 Type                                 : Int2B
   +0x002 Size                                 : Uint2B
   +0x004 ReferenceCount                       : Int4B
   +0x008 DriverObject                         : Ptr32 _DRIVER_OBJECT
   +0x00c NextDevice                           : Ptr32 _DEVICE_OBJECT
   +0x010 AttachedDevice                       : Ptr32 _DEVICE_OBJECT
   +0x014 CurrentIrp                           : Ptr32 _IRP
   +0x018 Timer                                : Ptr32 _IO_TIMER
   +0x01c Flags                                : Uint4B
   +0x020 Characteristics                      : 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
```

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 Flags            : Uint4B
   +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 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.

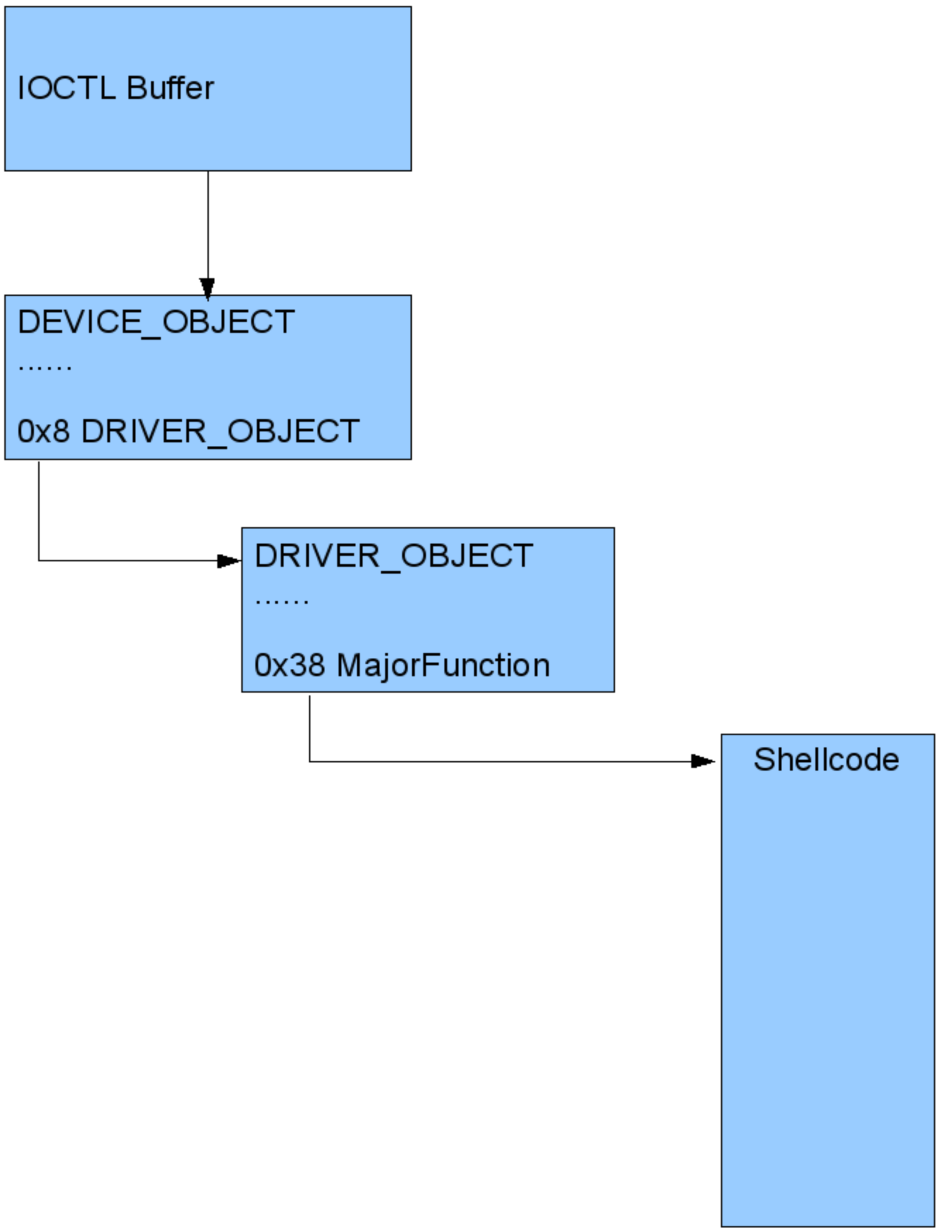


Figure 8: Layout of fake kernel objects.

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:

```
0x00010739:
 MOV ECX,DWORD PTR DS:[EBX+2C]
 LEA EAX,DWORD PTR DS:[EDI+EDI*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]
 PUSH 664F3249
 PUSH EDI
 PUSH 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