



Intro to Windows Kernel Security Development (uCON-Conference 2009)

Who I am.

Stephen A. Ridley

**Senior Security Researcher/Consultant
(Matasano Security)**

- Previously Senior Security Architect at McAfee Inc.
- Intrusion Engineer at ManTech Security and Mission Assurance (supporting U.S. Defense and Intelligence)
- columnist for/interviewed by IT magazines (Wired, Ping!, HostingTech, etc.)
- Kenshoto DefCon CTF organizers
- focus: Software Reverse Engineering, tool development, software security

Matasano: What We Do.

- Independent Security R&D firm (New York, and Chicago)
- Work with vendors and enterprises at all phases of the software development life-cycle to pinpoint and eradicate security flaws:
 - Penetration Testing
 - Reverse Engineering
 - Source Code Review
 - Custom tool development
- Our customers span the Fortune 500

Matasano: What We've Done.

- Former @stake co-founders
- First published X86 Stack Overflow
- Invented IDS/IPS evasion attacks
- First published iSCSI protocol vulnerability
- First VT-x (hypervisor) Rootkit proof-of-concept and detection

Check out our blog...



<http://www.matasano.com/log>

What am I talkin' about today?

★ Intro to the Kernel

- Layout
- I/O, drivers, Object namespace, etc.

★ Developing for the NT Kernel

- Writing drivers
- Analysis/Reversing
- A little shellcoding

★ Kernel Debugging (it's “quiet” up here.)

★ Reversing NT Kernel stuff (drivers)

- for bug-hunting (fuzzing, etc)

Please feel free to interrupt.

Please feel free to
interrupt me, I like my
presentations to be
conversational...

1. NT Kernel Introz

The “why” is obvious!

**“[The Agents] are the gatekeepers Neo,
they are guarding all the doors, they are
holding all the keys...”**

-Morpheus “The Matrix”

The Layout of the Kernel

- ★ **There are a few presentations on this, most notably:**
 - “Windows Kernel Internals Overview” (9 Oct 2008)
Dave Probert: Windows Kernel Group

- ★ **Several great books:**
 - “Undocumented Windows 2000 Secrets”
 - Gary Nebbett’s “The Windows 2000 Native API Reference”
 - “Windows Internals” Russinovich (several editions)

Organized in 3 major groups

★ NTOS (Kernel Mode Services)

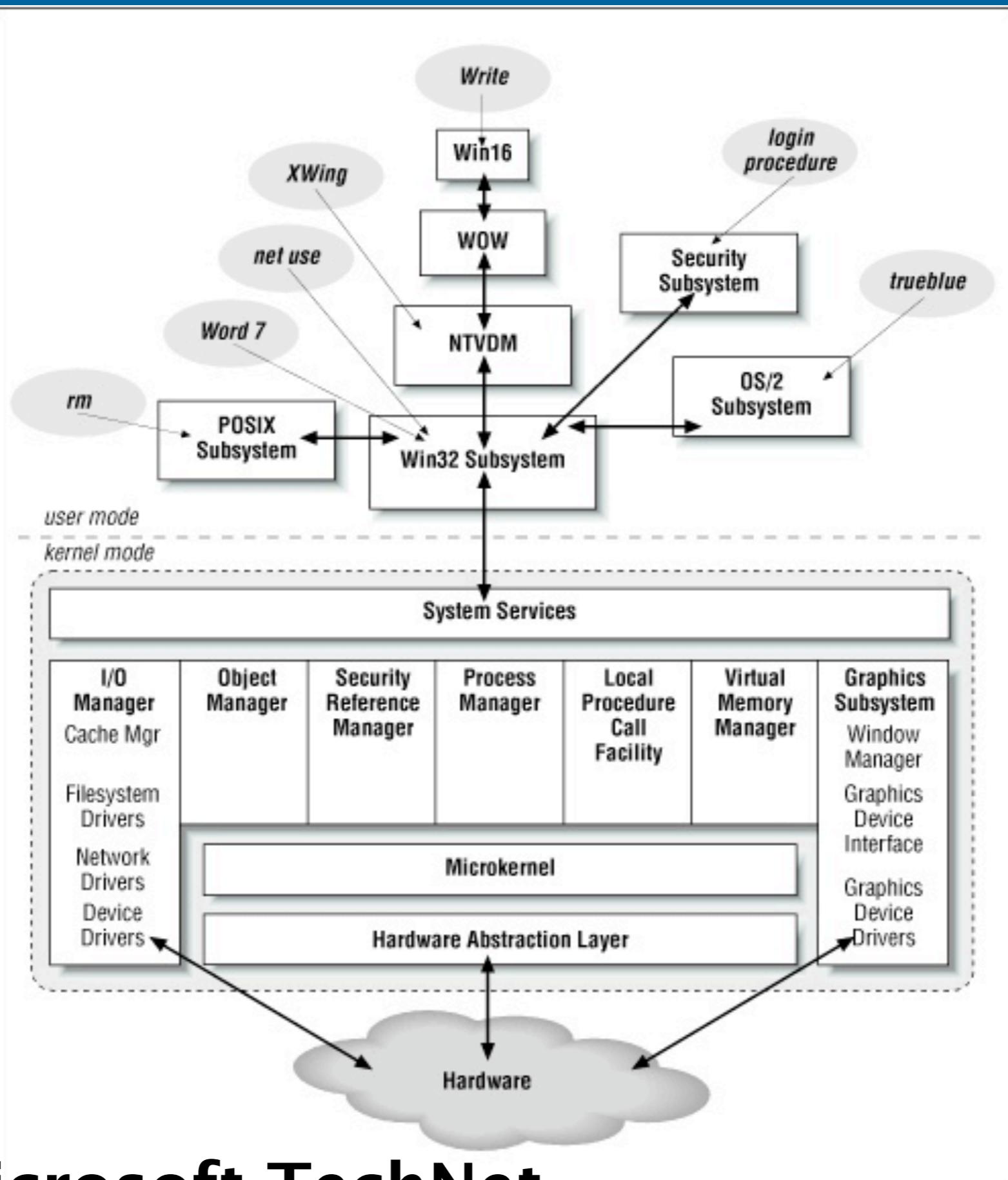
- RTL stuff, executive services, object management, I/O stuff, memory stuff, process loading, scheduling/priority queuing, etc.

★ HAL (Hardware Abstraction Layer)

- Abstraction layer so that NTOS and drivers don't need to know about the nitty-gritty hardware details.
- Has all the API stuff you'd expect for dealing with hardware (timers, mutexes, locks, spinlocks, etc.)

★ Drivers

- Kernel extensions



(credit) Microsoft TechNet

Kernel's Major Components

- ★ Object Manager (OB)
- ★ Security Reference Monitor (SE)
- ★ Process/Thread Management (PS)
- ★ Memory Manager (MM)
- ★ Cache Manager (CACHE)
- ★ Scheduler (KE)
- ★ I/O Manager, PnP, power, GUI (IO)
- ★ Devices, FS Volumes, Net (DRIVERS)
- ★ Lightweight Procedure Calls (LPC)
- ★ Hardware Abstraction Layer (HAL)
- ★ Executive Functions (EX)
- ★ Run-Time Library (RTL)
- ★ Registry/Consistent Configuration (CONFIG)

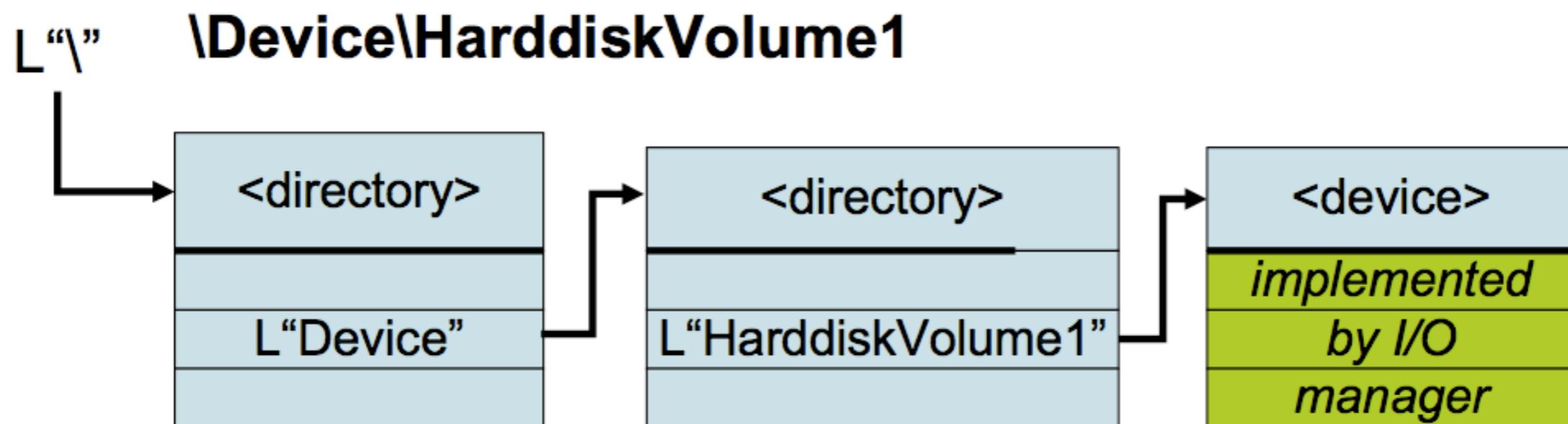
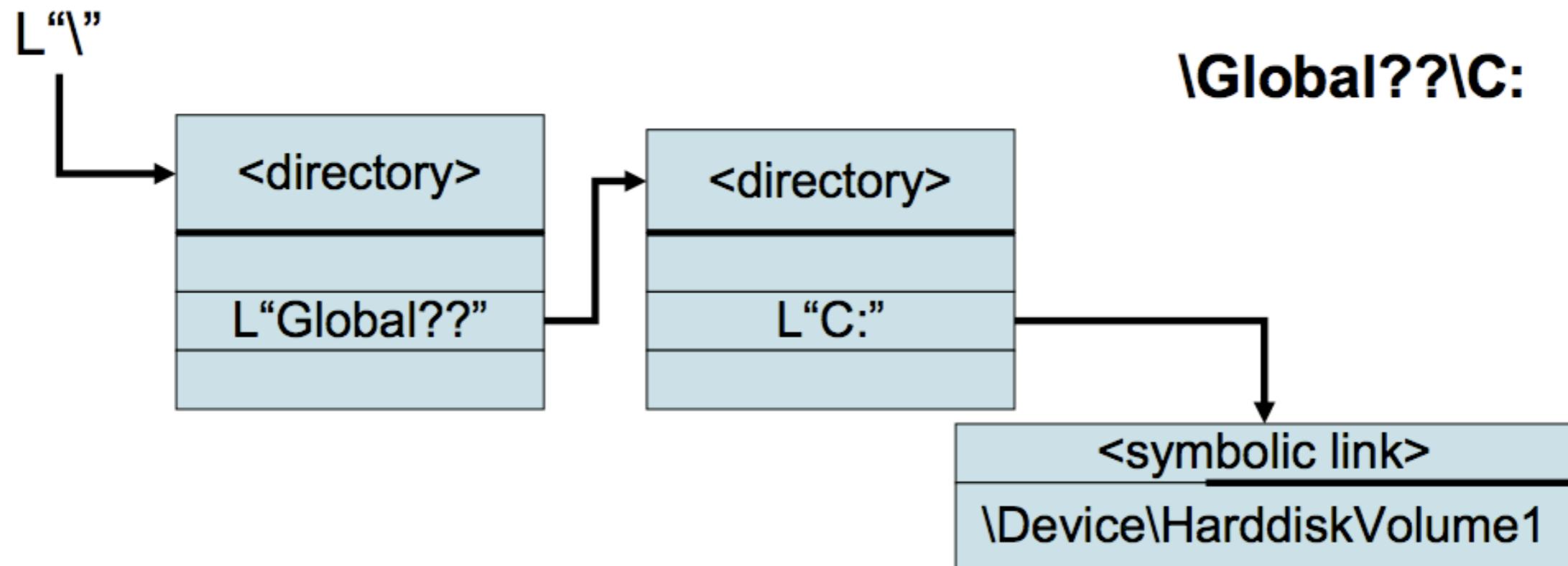
The stuff we care about...

- ★ **Object Manager (OB)**
- ★ **Security Reference Monitor (SE)**
- ★ **Process/Thread Management (PS)**
- ★ **Memory Manager (MM)**
- ★ **Cache Manager (CACHE)**
- ★ **Scheduler (KE)**
- ★ **I/O Manager, PnP, power, GUI (IO)**
- ★ **Devices, FS Volumes, Net (DRIVERS)**
- ★ **Lightweight Procedure Calls (LPC)**
- ★ **Hardware Abstraction Layer (HAL)**
- ★ **Executive Functions (EX)**
- ★ **Run-Time Library (RTL)**
- ★ **Registry/Consistent Configuration (CONFIG)**

Object Manager

- ★ An “abstraction layer”: the same thing maybe be known by many names
- ★ Handles/Descriptors are a perfect example of this. You do OpenFile() and get back a number...
- ★ It provides operations (read, write, delete, etc.)
- ★ Since the Object Manager does this “name conversion” this is the perfect place to also do security checks!
 - Security Reference Monitor sits “behind” the Object Manager to check ACLs and stuff...

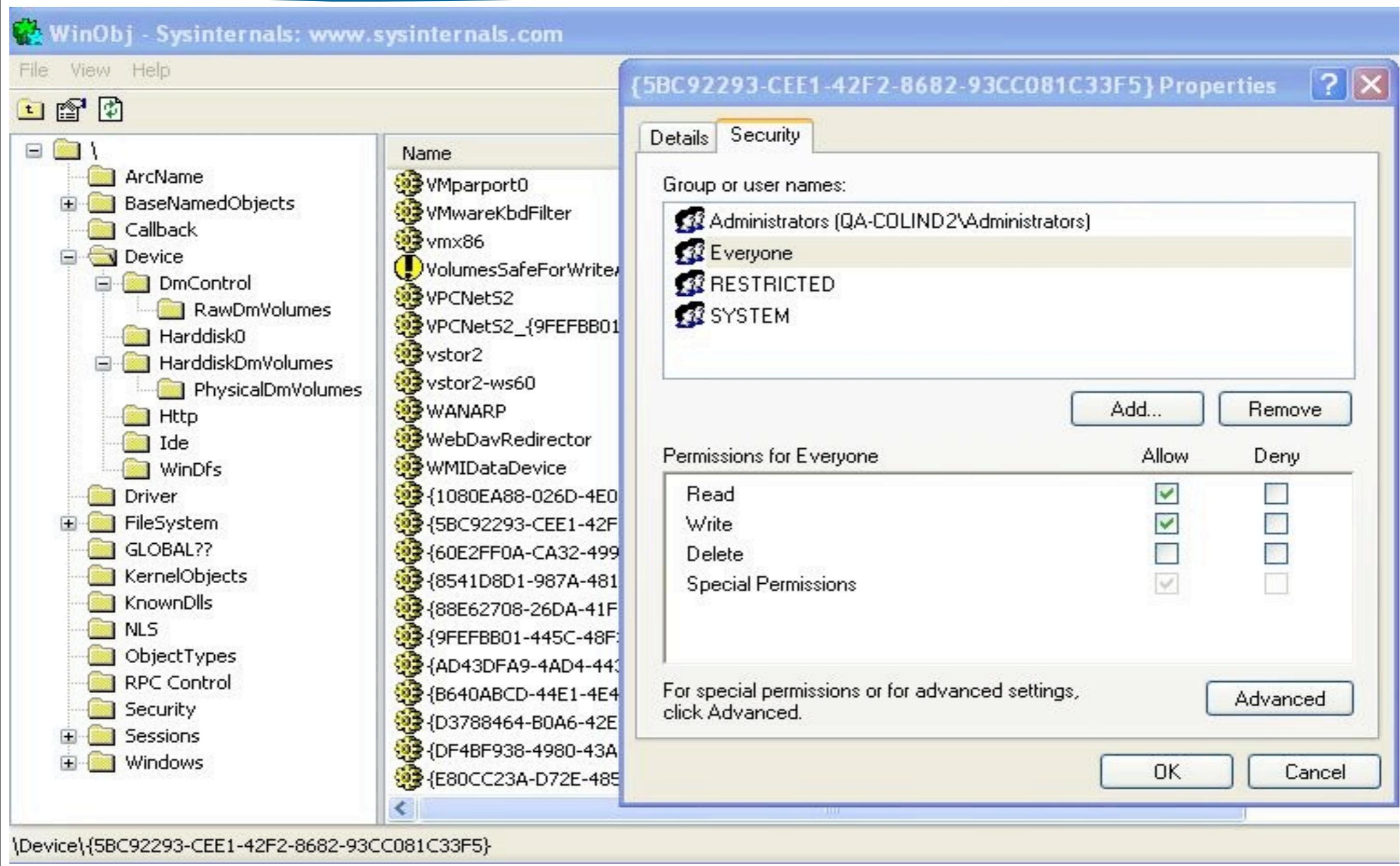
NT Object Conversion



Many Object Types in NT NS

Adapter	File	Semaphore
Callback	IoCompletion	SymbolicLink
Controller	Job	Thread
DebugObject	Key	Timer
Desktop	KeyedEvent	Token
Device	Mutant	Type
Directory	Port	Waitable Port
Driver	Process	WindowsStation
Event	Profile	WMIGuid
EventPair	Section	

Peeking at the NT Object NS



Kernel I/O

- ★ The Kernel has to communicate with stuff somehow!
- ★ Drivers communicate with userland components in a number of ways most commonly via IOCTLs

IOCTLS

- ★ IOCTLS are like “special functions” called from userland processes that kernel drivers “listen” for.
- ★ Each driver “listens” by registering a unique identifier (called an `IOControlCode`) to listen for
- ★ I like think of this mechanism much like User32. How everything evolves around a few “extensible” functions (like `SendMessage()`, `PeekMessage()`, etc.)

Kernel I/O

- ★ The `DRIVER_OBJECT` structure is how your driver registers a “dispatch” function. This dispatch is just a callback that gets called...
- ★ Think of this as an oldskool token ring network. Every driver gets all data and decides whether it wants it.

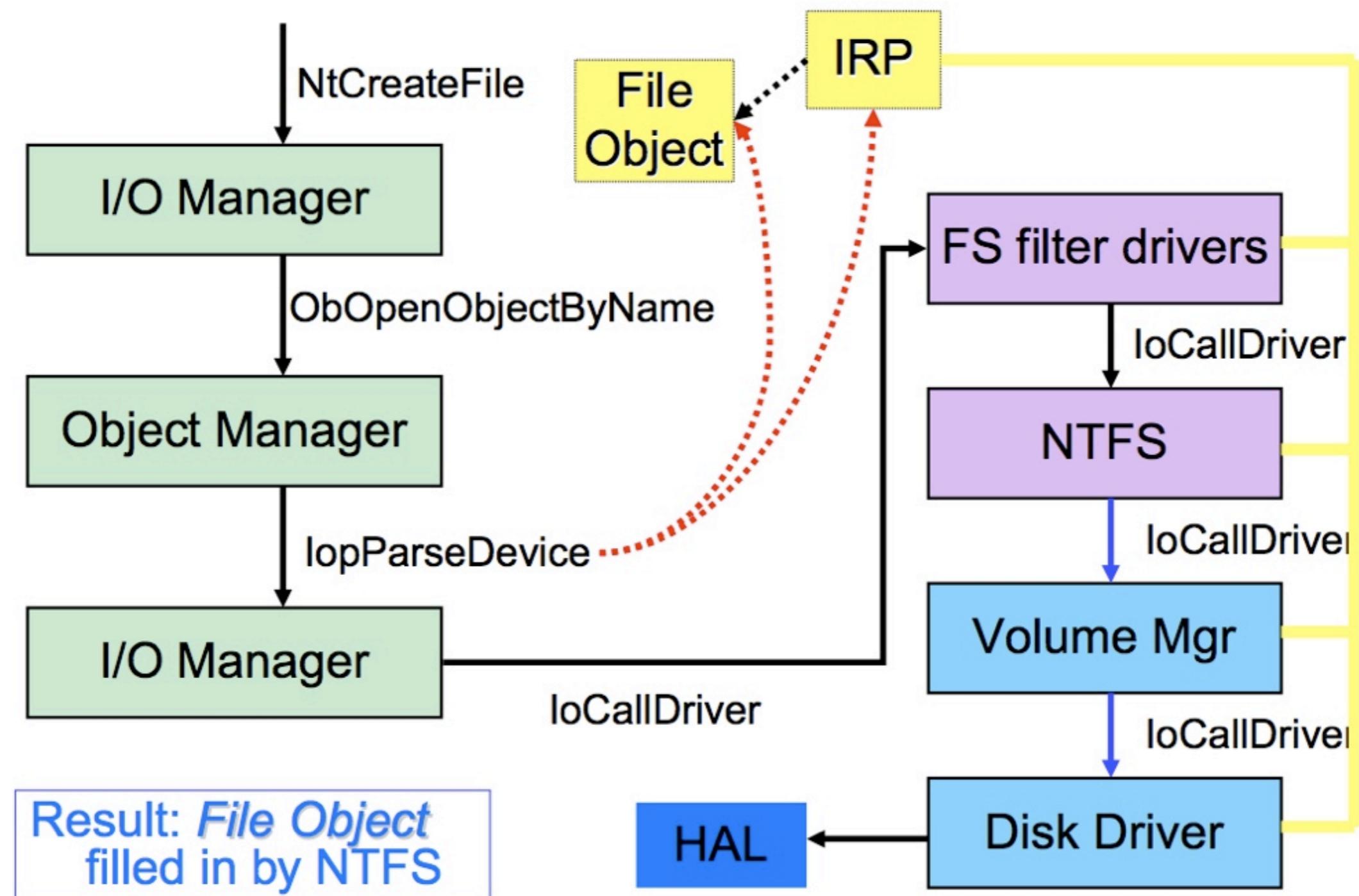
DRIVER_OBJECT (Kernel I/O)

- ★ The DRIVER_OBJECT “registration” would look something like:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =  
mydispatchfunc;
```

- ★ mydispatchfunc then gets called when anyone sends an IOCTL to the driver stack
- ★ IOCTL data comes in as a special structure called Interrupt Request Packet (_IRP)
- ★ Keep in mind the actual IOCTL “opcode” can be reversed out of a binary (.sys, .dll, etc) More on that later.

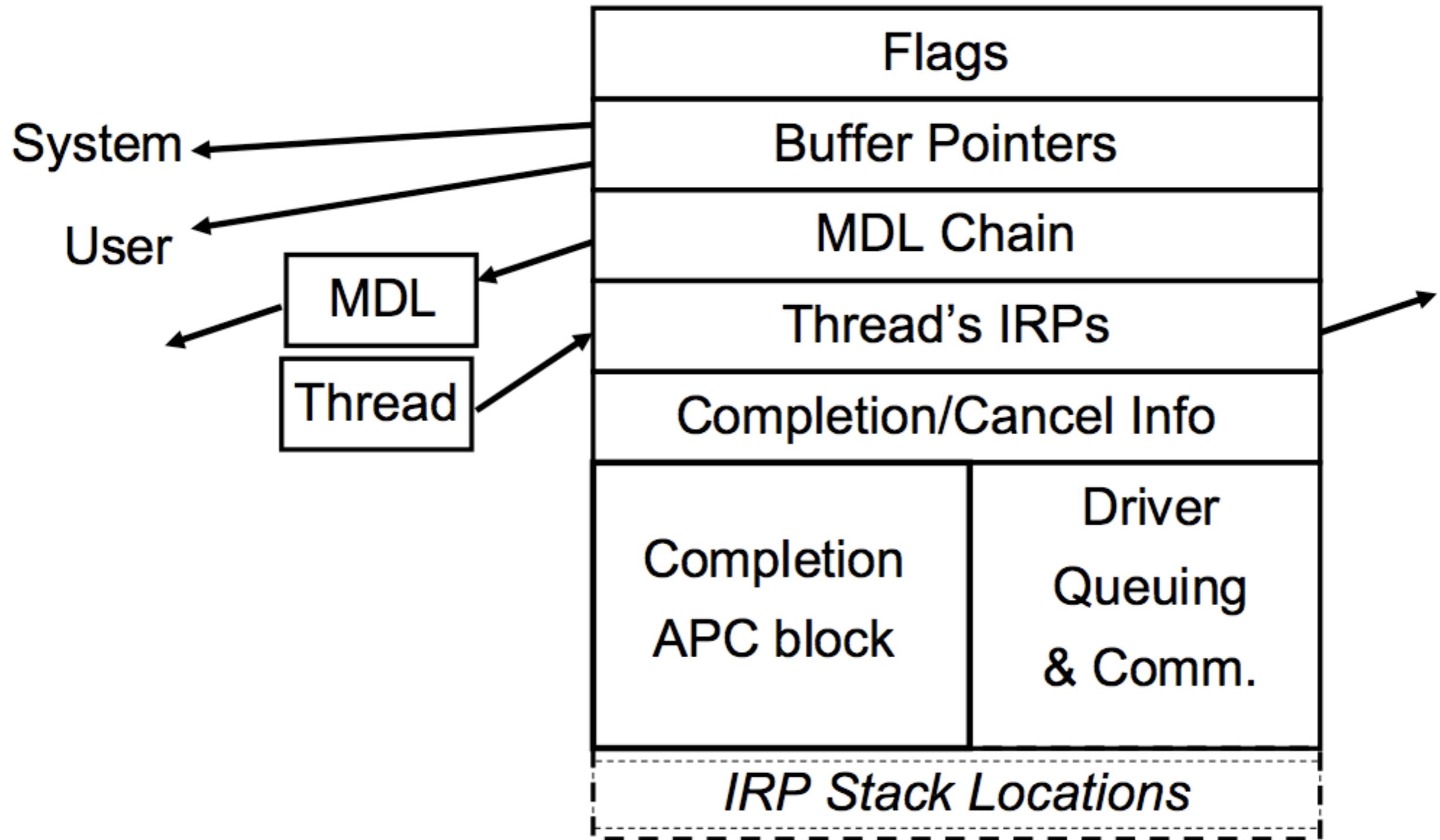
Kernel I/O ... IRP



IRP Structure

- ★ In windows *all* I/O events boil down to some IRP structure being passed to some dispatch somewhere.
- ★ Again it helps to think of this as User32 where every action (even movement of the mouse) is a SendMessage() to some window *somewhere*.
- ★ The associated IOControlCode (“opcode”) is inside the _IRP structure and is how drivers decide they care about the interrupt.

IRP Structure



Device “Layering”/the Stack

- ★ Drivers are “layered” one on top of the other when they “register” using the `IOAttachDevice()` API
- ★ (Actually I’ve never used that function, I’ve used `IOCreateDevice()`/
`IoCreateSymboliclink()`, same thing but creates instead of attaching to existing)

Device “Layering”/the Stack

- ★ The I/O manager sends all IRPs to the top of the stack
- ★ Drivers are linked together as a linked list, so each driver has pointer to next device driver down.
- ★ Driver “unregistering” and deconstruction happens with `IODetachDevice()` (`I've only ever used IODeleteDevice()`)

Synchronous vs Asynchronous

- ★ The way that the driver handles the Interrupt request when it comes in is more or less what determines what I/O mode the driver uses.
- ★ If the DriverEntry() (the “main” of a driver returns “**STATUS_PENDING**”) then its asynchronous and can continue processing and notify the manager using IOCompleteRequest()

2. Getting started Dev'ing

Getting Debuggers setup...

- ★ WinDBG users are vindicated! You endured ridicule before, but now that SoftIce is gone *now* everyone is using your debugger like it was always cool.
- ★ Extremely well documented
- ★ Powerful scripting engine (you get to keep your old WinDBG scripts :-)

Debugging Over Serial

- ★ Edit boot.ini on debuggee
- ★ Serial Debugging and VMWare makes it all possible without a “hardware box”.
- ★ Works by creating “virtual serial port” that is a named pipe on host OS.
- ★ On VMWare Fusion some virtual serial port configuration “gotchas”
 - Found solutions in VMWare developer forums.

Debuggee (server) VMX file

```
Terminal — vim — 81x30

49 uuid.action = "create"
50
51 virtualHW.productCompatibility = "hosted"
52
53 unity.wasCapable = "TRUE"
54 vmotion.checkpointFBSIZE = "134217728"
55
56 hgfs.mapRootShare = "TRUE"
57 hgfs.linkRootShare = "TRUE"
58 isolation.tools.hgfs.disable = "FALSE"
59
60 gui.fullScreenAtBoot = "FALSE"
61 gui.windowModeAtPowerOn = "windowed"
62
63 serial0.present = "TRUE"
64 serial0.fileType = "pipe"
65 serial0.yieldOnMsrRead = "TRUE"
66 serial0.startConnected = "TRUE"
67 serial0.fileName = "/data/kernel_debug_serial_port"
68
69 pciBridge0.present = "TRUE"
70 ehci.present = "TRUE"
71 pciBridge4.present = "TRUE"
72 pciBridge4.virtualDev = "pcieRootPort"
73 pciBridge4.pciSlotNumber = "21"
74 pciBridge4.functions = "8"
75 pciBridge5.present = "TRUE"
76 pciBridge5.virtualDev = "pcieRootPort"
77 pciBridge5.pciSlotNumber = "22"
```

Debugger (client) VMX file

```
Terminal — vim — 86x30

50 vmotion.checkpointFBSIZE = "134217728"
51 checkpointFBSIZE = "16777216"
52 sharedFolder0.present = "TRUE"
53 sharedFolder0.enabled = "TRUE"
54 sharedFolder0.readAccess = "TRUE"
55 sharedFolder0.writeAccess = "TRUE"
56 sharedFolder0.hostPath = "/data"
57 sharedFolder0.guestName = "data"
58 sharedFolder0.expiration = "never"
59
60 ethernet0.connectionType = "nat"
61
62 ethernet0.startConnected = "TRUE"
63
64 serial0.present = "TRUE"
65 serial0.fileType = "pipe"
66 serial0.pipe.endPoint = "client"
67 serial0.yieldOnMsrRead = "TRUE"
68 serial0.startConnected = "TRUE"
69 serial0.fileName = "/data/kernel_debug_serial_port"
70
71
72 gui.fullScreenAtPowerOn = "FALSE"
73 gui.viewModeAtPowerOn = "windowed"
74
75 pciBridge0.present = "TRUE"
76 ehci.present = "TRUE"
77 pciBridge4.present = "TRUE"
78 pciBridge4.virtualDev = "pcieRootPort"
```

Finally connected.

```
Visual Studio .NET 2003 Command Prompt - kd.exe
C:\Program Files\Debugging Tools for Windows>kd.exe

Microsoft (R) Windows Debugger Version 6.5.0003.7
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\com1
Waiting to reconnect...
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is: *** Invalid ***
*****
* Symbol loading may be unreliable without a symbol search path. *
* Use .sympath to have the debugger choose a symbol path. *
* After setting your symbol path, use .reload to refresh symbol locations. *
*****
Executable search path is:
*****
* Symbols can not be loaded because symbol path is not initialized. *
*
* The Symbol Path can be set by:
*   using the _NT_SYMBOL_PATH environment variable. *
*   using the -y <symbol_path> argument when starting the debugger. *
*   using .sympath and .sympath+
*****
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ntkr
nlp.exe -
Windows XP Kernel Version 2600 (Service Pack 2) UP Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 2600.xpsp_sp2_gdr.080814-1233
Kernel base = 0x804d7000 PsLoadedModuleList = 0x80553420
Debug session time: Fri Feb 27 22:21:29.969 2009 (GMT-8)
System Uptime: 0 days 0:37:52.968
Break instruction exception - code 80000003 (first chance)
*****
*
* You are seeing this message because you pressed either
*   CTRL+C (if you run kd.exe) or,
*   CTRL+BREAK (if you run WinDBG),
* on your debugger machine's keyboard.
*
THIS IS NOT A BUG OR A SYSTEM CRASH
*
*
* If you did not intend to break into the debugger, press the "g" key, then
* press the "Enter" key now. This message might immediately reappear. If it
* does, press "g" and "Enter" again.
*
*****
nt!DbgBreakPointWithStatus+0x4:
80526fe8 cc          int     3
kd>
```

Bite the bullet.

- ★ If you are like me you prefer to dev with ViM or something and use a CLI compiler.
- ★ You still can!
 - VMWare Shared Folders and batch files that use cl.exe
- ★ You can, but Visual Studio really will make your life easier if you let it.
- ★ Visual Studio can seem overwhelming at first, if you aren't used to IDEs. Don't let it intimidate you :-) ...

Getting everything...

- ★ For driver development (beginners like us) most of what I have been talking about implies NT5.
- ★ Grab the Windows Driver Development Kit (DDK) and the Platform SDK from Microsoft.
- ★ MSDN is your friend! We all may dislike Microsoft products but you must agree how well documented many are. You'll find this even more so in the DDK.

Taking a look at my driver...

★ Starting out you will probably develop two things:

- a kernel mode component to do your first ‘thing’.
- a “controller” to speak to the driver from userspace

★ **DriverEntry()**

★ **CreateDevice()**

★ **MajorFunction registration**

★ **The driver guts...**

★ **DeleteDevice()**

★ **return to IO Manager**

KHD: Kernel Humpty Dumpty

- ★ My old shellcode test harness “Humpty Dumpty” (HD) was for regular userland shellcoding
 - Loaded compiled assembly from disk and executed
 - It had features to load libraries (for you to practice algorithms on), do user32 injection, dll injection, etc.
- ★ KHD is the “kernel version” that simply loads compiled assembly from IOCTL and jumps into it.
- ★ We can use this to see basic structure of a driver

The start (driver entrypoint)

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath) {
    NTSTATUS status;
    UNICODE_STRING devName, devLink;
    int i;

    RtlInitUnicodeString(&devName, L"\Device\sa7");
    RtlInitUnicodeString(&devLink, L"\DosDevices\sa7");

    status = IoCreateDevice(DriverObject,
                           0,
                           &devName,
                           KTRACER_DRV,
                           0,
                           TRUE,
                           &g_devObj);

    if(!NT_SUCCESS(status)){
        IoDeleteDevice(DriverObject->DeviceObject);
        DbgPrint("Failed to create device\n");
        return status;
    }

    status = IoCreateSymbolicLink(&devLink, &devName);
    if(!NT_SUCCESS(status)) {
        IoDeleteDevice(DriverObject->DeviceObject);
        DbgPrint("Failed to create symbolic link\n");
        return status;
    }

    for(i=0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++) {
        DriverObject->MajorFunction[i] = KHDDispatch;
    }
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = KHDIoControl;
    DriverObject->DriverUnload = KHDUnload;
```

Device Control dispatch

```
NTSTATUS KHDIoControl(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
    PIO_STACK_LOCATION irpStack;
    ULONG ioControl;
    NTSTATUS status = STATUS_SUCCESS;
    ULONG information = 0;
    PVOID inBuf, outBuf;
    ULONG inLen, outLen;
    irpStack = IoGetCurrentIrpStackLocation(Irp);
    inBuf = Irp->AssociatedIrp.SystemBuffer;
    inLen = irpStack->Parameters.DeviceIoControl.InputBufferLength;
    outBuf = Irp->AssociatedIrp.SystemBuffer;
    outLen = irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    ioControl = irpStack->Parameters.DeviceIoControl.IoControlCode;
    switch(ioControl) {
        case IOCTL_EXEC_SHELLCODE:
            // Do a buncha stuff omitted for screenshot
        default:
            DbgPrint("Unknown IOCTL\n");
            status = STATUS_INVALID_DEVICE_REQUEST;
    }
    // complete IRP
    // http://msdn.microsoft.com/en-us/library/ms796109.aspx
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = information;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

The IOCTL Code

```
#ifndef __KHD_H__  
#define __KHD_H__  
  
// driver IOCTLs  
#define KHD      0xd3adb33f  
#define IOCTL_EXEC_SHELLCODE    CTL_CODE(KHD, 0x07, METHOD_BUFFERED, FILE_READ_ACCESS)  
  
#endif|
```

Extracted from winioctl.h

```
#define CTL_CODE( DeviceType, Function, Method, Access ) (           \  
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \  
)
```

Cleanup and “blank” dispatch

```
L  
└─ void KHDUnload(IN PDRIVER_OBJECT DriverObject) {  
    // Do nothing. free memory or something if we cared.  
}  
  
└─ NTSTATUS KHDDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp){  
    Irp->IoStatus.Status = STATUS_SUCCESS;  
    IoCompleteRequest(Irp, IO_NO_INCREMENT);  
    return STATUS_SUCCESS;  
}
```

Ok...

Now that we have taken a look at a skeletal driver, let's take a step back and remember why we even started.

- 1. Writing drivers ourselves to do....fun tasks for us >:-)**
- 2. Vulnerability research of existing drivers.**

Poking at Drivers

Poking at stuff...

- ★ Often times as security people we miss the “big picture”...
- ★ As a security person, sometime it’s best to initially approach a project (or technology) as just a “curious developer” (as we did earlier in this presentation)
- ★ Now that we know what “regular” kernel developers start with, lets look take a look with the purpose of vuln research...

Approaching a target...

★ Take a look at the driver list with Kartoffel:

“a extensible command-line tool developed with the aim of helping developers to test the security and the reliability of a driver.”

<http://kartoffel.reversemode.com/>

1. **kartoffel.exe -r > drivers-clean.txt**
2. **Install the software to be tested**
3. **kartoffel.exe -r > drivers-installed.txt**
4. **diff the two text files**

Approaching a target...

★ Check NTObj ACLs with WinObj:

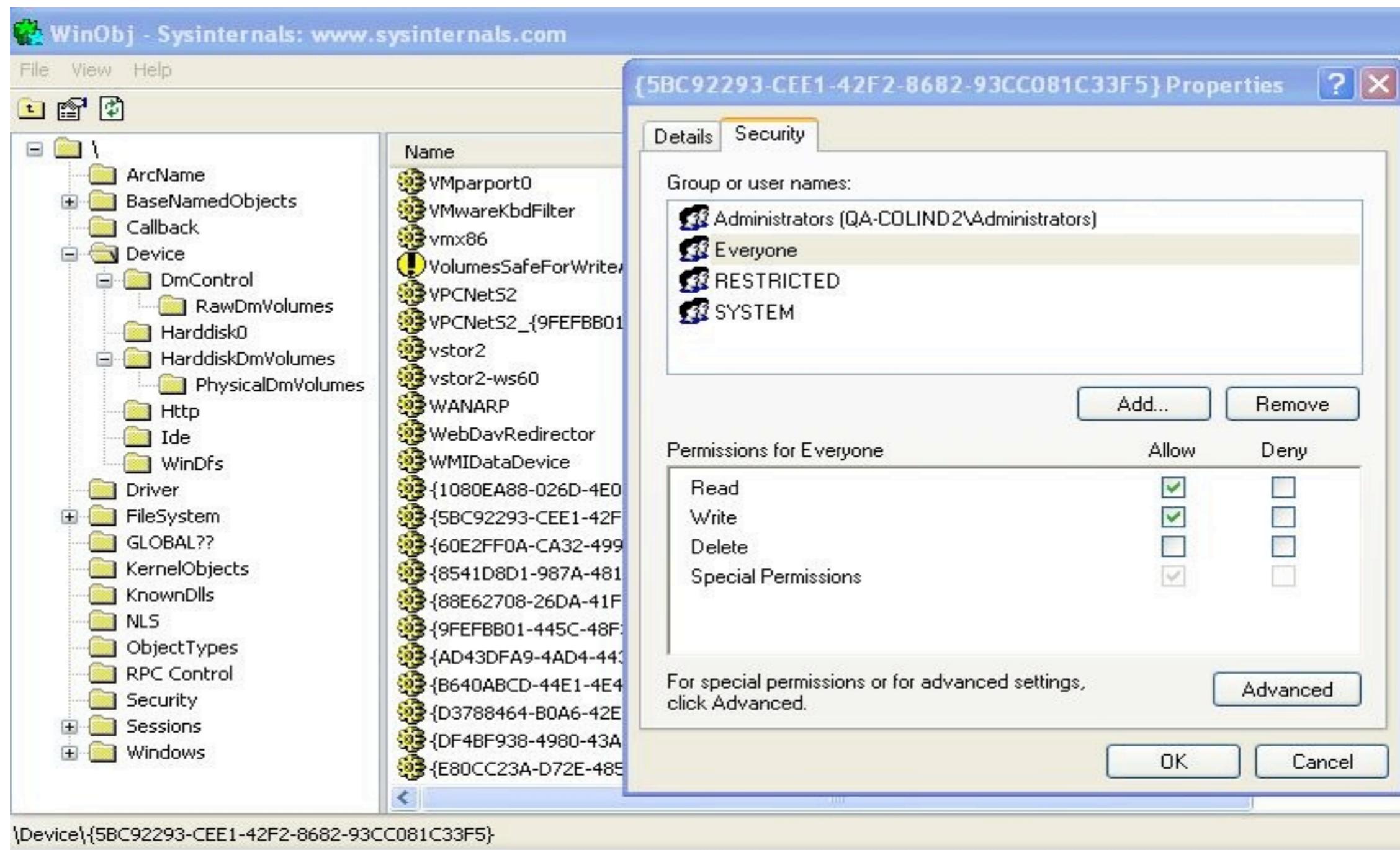
“a 32-bit Windows NT program that uses the native Windows NT API to access and display information on the NT Object Manager's name space.”

[http://technet.microsoft.com/en-us/
sysinternals/bb896657.aspx](http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx)

1. Launch WinObj
2. Open the \Device node
3. For each driver, right-click / Properties
4. Navigate to the Security tab
5. Select the Everyone group
6. Audit the allowed permissions

Approaching a target...

- ★ Driver endpoint permissions are commonly overlooked... "Read/Write Everyone" is generally not good...



Approaching a target...

- ★ Next you want to identify the IOCTLs used by the driver
- ★ If source is available you are looking for the main switch/if statements in the IoControlCode dispatch
- ★ If source is not available then we have to reverse the control codes out

Reversing out IOCTL codes...

**There are a number of great papers and presentations on this already:
(all of these links provided later)**

- (SK of Scan Associates) XCon 2004 presentation
- Ruben Santamarta's Reversemode MRXDMB.SYS paper
- Justin Seitz's (of Immunity Inc.) "Driver Impersonation Attack paper".
- Barnaby Jack's seminal "Step Into The Ring" papers
- NGS Security's "Attacking the Windows Kernel"

Not going to echo-chamber...

★ But let's take a quick look at how to reverse out IOCTLs from a driver:
AFD.SYS

★ Why AFD?

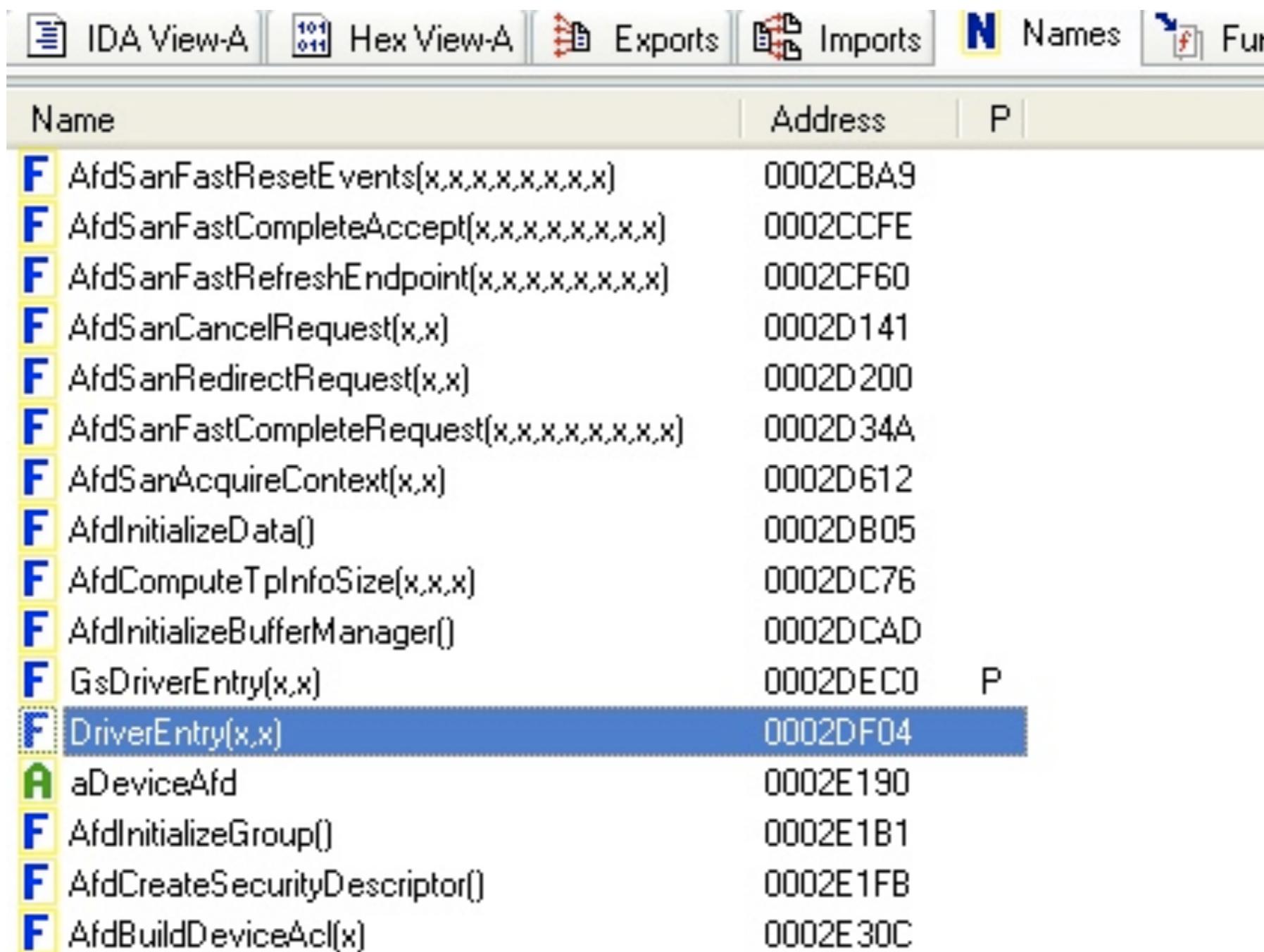
- Because there have been bugs there in the past >:-)
- AFD happens to handle many IOCTLs...

Reversing out IOCTL codes...

- ★ Fire up IDA!
- ★ Everyone has a different technique but I am new so I just start at DriverEntry() since the IOManager has to ;-)
- ★ There are apparently Driver Development Frameworks within the DDK (RDBSS) that can sometimes obscure my simple technique of starting at DriverEntry (but I have yet to see those for myself)

Reversing out IOCTL codes...

★ Locate “DriverEntry”



Name	Address	P
F AfdSanFastResetEvents(x,x,x,x,x,x,x,x)	0002CBA9	
F AfdSanFastCompleteAccept(x,x,x,x,x,x,x)	0002CCFE	
F AfdSanFastRefreshEndpoint(x,x,x,x,x,x,x)	0002CF60	
F AfdSanCancelRequest(x,x)	0002D141	
F AfdSanRedirectRequest(x,x)	0002D200	
F AfdSanFastCompleteRequest(x,x,x,x,x,x,x)	0002D34A	
F AfdSanAcquireContext(x,x)	0002D612	
F AfdiInitializeData()	0002DB05	
F AfdComputeTpInfoSize(x,x,x)	0002DC76	
F AfdiInitializeBufferManager()	0002DCAD	
F GsDriverEntry(x,x)	0002DEC0	P
F DriverEntry(x,x)	0002DF04	
A aDeviceAfd	0002E190	
F AfdiInitializeGroup()	0002E1B1	
F AfdCreateSecurityDescriptor()	0002E1FB	
F AfdBuildDeviceAcl(x)	0002E30C	

Reversing out IOCTL codes...

★ We start reading....

The screenshot shows the assembly view in IDA Pro. The top menu bar includes options like IDA View-A, Hex View-A, Exports, Imports, Names, Functions, Strings, Structures, and Enums. The assembly code is as follows:

```
; Attributes: bp-based frame
; void DriverEntry
_DriverEntry@8 proc near

DeviceName= LSA_UNICODE_STRING ptr -0Ch
var_4= dword ptr -4
DriverObject= dword ptr  8

; FUNCTION CHUNK AT 0002EB0C SIZE 000000EF BYTES

mov    edi, edi
push   ebp           ; Tag
mov    ebp, esp
sub    esp, 0Ch       ; Free
push   ebx           ; Allocate
push   esi           ; Lookaside
push   edi
push   offset word_2E18E ; SourceString
lea    eax, [ebp+DeviceName]
push   eax           ; DestinationString
call   ds:_imp__RtlInitUnicodeString@8 ; RtlInitUnicodeString(x,x)
push   offset _AfDDeviceObject ; DeviceObject
xor    ebx, ebx
.
```

Reversing out IOCTL codes...

★ Reading through DriverEntry you stumble upon:

Hex View-A Exports Imports Names Functions Strings Structures Enums

jz

mov edx, [ebp+DriverObject]
push 1Ch
lea edi, [edx+38h]
pop ecx
mov eax, offset _AfdDispatch@8 ; AfdDispatch(x,x)
rep stosd
mov dword ptr [edx+70h], offset _AfdDispatchDeviceControl@8 ; AfdDispatchDeviceControl(x,x)
mov dword ptr [edx+28h], offset _AfdFastIoDispatch
mov dword ptr [edx+34h], offset _AfdUnload@4 ; AfdUnload(x)
mov eax, _AfdDeviceObject
or dword ptr [eax+1Ch], 10h
mov eax, _AfdDeviceObject
mov cl, _AfdIrpStackSize
mov [eax+30h], cl
call ds:_imp_IoGetCurrentProcess@0 ; IoGetCurrentProcess()
cmp _AfdParametersNotifyHandle, ebx
mov _AfdSystemProcess, eax
jnz loc_2EBF0

push
call
mov

Reversing out IOCTL codes...

★ Following into
`_AfdDispatchDeviceControl` we see:

The screenshot shows the assembly view in IDA Pro. The top window displays the entry point of `_AfdDispatchDeviceControl@8`. The assembly code is as follows:

```
; __stdcall AfdDispatchDeviceControl(x, x)
_AfdDispatchDeviceControl@8 proc near

arg_4= dword ptr 0Ch

    mov     edi, edi
    push    ebp
    mov     ebp, esp
    mov     ecx, [ebp+arg_4]
    mov     edx, [ecx+60h]
    push    esi
    push    edi
    mov     edi, [edx+0Ch]
    mov     eax, edi
    shr     eax, 2
    and    eax, 3FFh
    cmp     eax, 46h
    jnb    loc_21B73
```

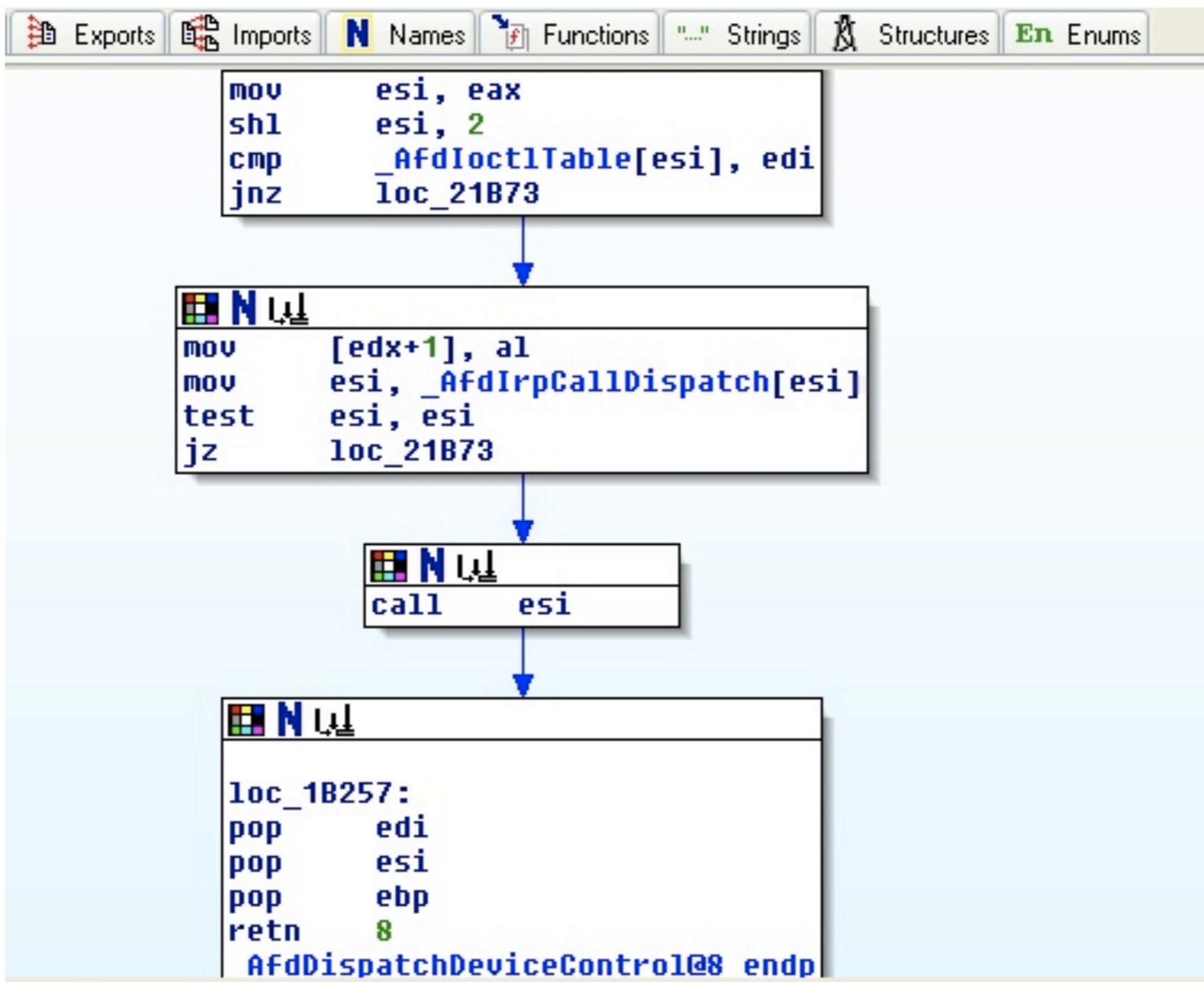
An arrow points from the end of this code to the bottom window, which contains the following assembly code:

```
N 41
    mov     esi, eax
    shl     esi, 2
    cmp     _AfdIoctlTable[esi], edi
    jnz    loc_21B73
```

★ +60h IoGetCurrentIrpStack thnx Lawler!

Reversing out IOCTL codes...

- ★ We can see that this is really our dispatch, let's Investigate _AfdIoctlTable



Reversing out IOCTL codes...

★ IDA once again “helped” us too much,
lets CTRL-O and fix these values:

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code is as follows:

```
View-A Hex View-A Exports Imports Names Functions Strings Structures Enums
• .data:0001209B db 0
• .data:0001209C db 0
• .data:0001209D db 0
• .data:0001209E db 0
• .data:0001209F unk_1209F db 0 ; DATA XREF: .data:0001213C↓o
• .data:000120A0 _AfdIoctlTable dd offset _AfdMediumBufferSize+3 ; DATA XREF: .data:00012140↓o
• .data:000120A0 ; AfdFastIoDeviceControl(x,x,x,x,x,x,x,x,x)
• .data:000120A4 off_120A4 dd offset _AfdSmallBufferSize+3 ; DATA XREF: .data:00012144↓o
• .data:000120A8 off_120A8 dd offset _AfdDefaultTransmitWorker+3
• .data:000120A8 ; DATA XREF: .data:00012148↓o
• .data:000120AC off_120AC dd offset _AfdStandardAddressLength
• .data:000120AC ; DATA XREF: .data:0001214C↓o
• .data:000120B0 off_120B0 dd offset _AfdIrpStackSize ; DATA XREF: .data:00012150↓o
• .data:000120B4 off_120B4 dd offset _AfdFastSendDatagramThreshold+3
• .data:000120B4 ; DATA XREF: .data:00012154↓o
• .data:000120B8 off_120B8 dd offset _AfdTPacketsCopyThreshold+3
• .data:000120B8 ; DATA XREF: .data:00012158↓o
• .data:000120BC dd offset _AfdMaxFastTransmit+3
• .data:000120C0 off_120C0 dd offset _AfdMaxFastCopyTransmit+3
• .data:000120C0 ; DATA XREF: .data:00012160↓o
• .data:000120C4 off_120C4 dd offset _AfdUseTdiSendAndDisconnect
• .data:000120C4 ; DATA XREF: .data:00012164↓o
• .data:000120C8 off_120C8 dd offset _AfdDefaultTpInfoElementCount+3
• .data:000120C8 ; DATA XREF: .data:00012168↓o
• .data:000120CC off_120CC dd offset unk_1202F ; DATA XREF: .data:0001216C↓o
```

Reversing out IOCTL codes...

Voila!
(our IOCTLs)

•	.data:0001209D	db	0	
•	.data:0001209E	db	0	
•	.data:0001209F unk_1209F	db	0	; DATA XREF: .data:0001213C↓o
•	.data:000120A0 _AfdIoctlTable	dd	12003h	; DATA XREF: .data:00012140↓o ; AfdFastIoDeviceControl(x,x,x,
.	.data:000120A0			
•	.data:000120A4 dword_120A4	dd	12007h	; DATA XREF: .data:00012144↓o
•	.data:000120A8 dword_120A8	dd	1200Bh	; DATA XREF: .data:00012148↓o
•	.data:000120AC dword_120AC	dd	1200Ch	; DATA XREF: .data:0001214C↓o
•	.data:000120B0 dword_120B0	dd	12010h	; DATA XREF: .data:00012150↓o
•	.data:000120B4 dword_120B4	dd	12017h	; DATA XREF: .data:00012154↓o
•	.data:000120B8 dword_120B8	dd	1201Bh	; DATA XREF: .data:00012158↓o
•	.data:000120BC	dd	1201Fh	
•	.data:000120C0 dword_120C0	dd	12023h	; DATA XREF: .data:00012160↓o
•	.data:000120C4 dword_120C4	dd	12024h	; DATA XREF: .data:00012164↓o
•	.data:000120C8 dword_120C8	dd	1202Bh	; DATA XREF: .data:00012168↓o
•	.data:000120CC dword_120CC	dd	1202Fh	; DATA XREF: .data:0001216C↓o ; AfdFastIoDeviceControl(x,x,x,
.	.data:000120CC			
•	.data:000120D0 dword_120D0	dd	12033h	; DATA XREF: .data:00012170↓o
•	.data:000120D4 dword_120D4	dd	12037h	; DATA XREF: .data:00012174↓o
•	.data:000120D8 dword_120D8	dd	1203Bh	; DATA XREF: .data:00012178↓o
•	.data:000120DC dword_120DC	dd	1203Fh	; DATA XREF: .data:0001217C↓o
•	.data:000120E0 dword_120E0	dd	12043h	; DATA XREF: .data:00012180↓o
•	.data:000120E4 dword_120E4	dd	12047h	; DATA XREF: .data:00012184↓o
•	.data:000120E8 dword_120E8	dd	1204Bh	; DATA XREF: .data:00012188↓o
•	.data:000120EC dword_120EC	dd	1204Fh	; DATA XREF: .data:0001218C↓o

Fuzzing Drivers

- ★ Now with all the information gathered you can begin fuzzing
 - IOCTLs, DRIVER_OBJECT, endpoints, etc.
- ★ Kartoffel seems to be the most popular fuzzer for kernel things
- ★ I am more partial to doing this with custom tools, I personally use my fuzzer called Ruxxer (www.ruxxer.org) as the “engine” for test case generation.
- ★ Python and CTypes is excellent for the “glue code” that gets test-cases into the driver.
 - Opening devices, making IOCTLS, etc.

Kernel Shellcoding...

Kernel Shellcoding...

- ★ Shellcode “loaders” make it so that you don’t have to statically code in function addresses
- ★ Everyone basically ripped off the same userspace loader:
 - The fs:30 hashing “ror 0xd” GetProcAddress loader (probably originally by Dino Dai Zovi)
 - I am guilty of ripping this off as well ;-)
- ★ This loader found PEB Base via FS:30 then from there basically found GetProcAddress, and resolved functions

Kernel Shellcoding...

Terminal — vim — 111x44

```
start:    ; tell linker entry point, oh and also tell nasm the grow the fuck up
          ; and learn how to calculate relative offsets like an adult.

        mov ebp, esp
        sub esp, byte 0xc      ;sub esp, SIZEOF_BSS_IMPORTER    I need to find a way for nasm to calc and i
s value
        jmp GetHashDataAddr0    ;jmp GetHashDataAddr0

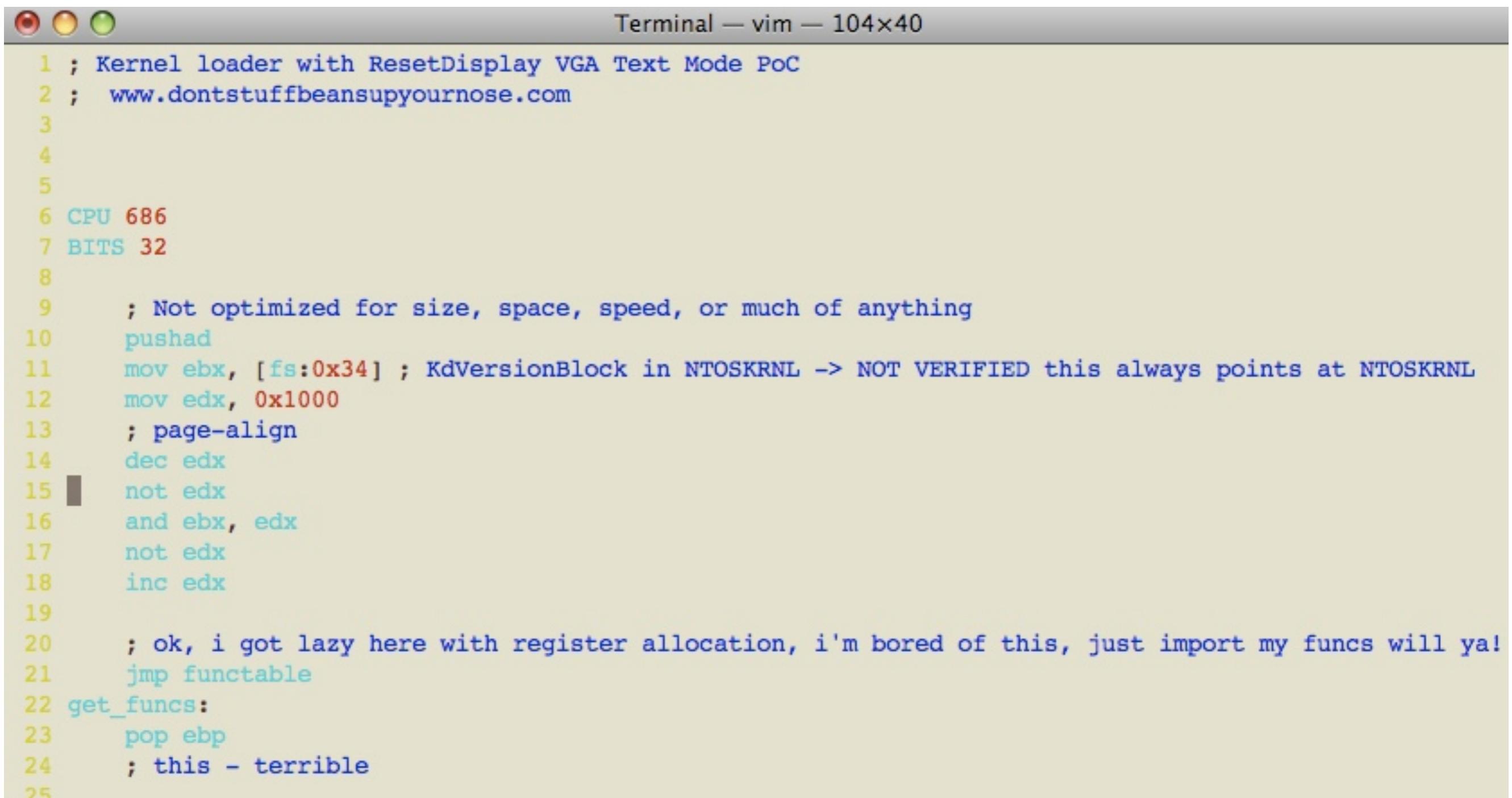
GetHashDataAddr1:
        pop esi
        mov [ebp-0xc], esi    ;mov bss.pHashStart, esi...why not mov [esp], esi?
        jmp short GetDoImportsAddr0 ;jmp GetDoImportsAddr0

GetDoImportsAddr1:
        pop edi
        ;Find kernel32 handle, walk through PEB module list to second entry
        mov eax, [fs:0x30]    ;PEB
        mov eax, [eax+0xc]    ;PEB_LDR_DATA
        mov eax, [eax+0x1c]    ;initorder link_entry in ldr_module for ntdll
        push byte 0x2        ;number of ntdll imports !!!CHANGE THIS BASED ON YOUR HASH TABLE SIZE
        push dword [eax+0x8]  ;ntdll handle
        mov eax, [eax]        ;initorder, link_entry in ldr_module for kernel32.dll
        push byte 0xd        ;number of kernel32 imports 13
        push dword [eax+0x8]  ;push Kernel32 base address
        call edi             ;call doImports
        ;call edi            ;call doImports this second one qot in here somehow
```

Kernel Shellcoding...

- ★ A “new” Kernel loader at:
www.dontstuffbeansupyournose.com
- ★ Uses FS:34 to find base of ntoskrnl.exe and from there uses similar hash technique to locate function exports.
- ★ Proof of Concept shellcode resets VGA driver and displays a neat message...

Kernel Shellcoding...



The image shows a terminal window titled "Terminal — vim — 104x40". The window contains assembly code with line numbers and comments. The code is written in AT&M assembly syntax.

```
1 ; Kernel loader with ResetDisplay VGA Text Mode PoC
2 ; www.dontstuffbeansupyournose.com
3
4
5
6 CPU 686
7 BITS 32
8
9 ; Not optimized for size, space, speed, or much of anything
10 pushad
11 mov ebx, [fs:0x34] ; KdVersionBlock in NTOSKRNL -> NOT VERIFIED this always points at NTOSKRNL
12 mov edx, 0x1000
13 ; page-align
14 dec edx
15 not edx
16 and ebx, edx
17 not edx
18 inc edx
19
20 ; ok, i got lazy here with register allocation, i'm bored of this, just import my funcs will ya!
21 jmp functable
22 get_funcs:
23 pop ebp
24 ; this - terrible
25
```

Kernel Shellcoding...

- ★ Interestingly, the structure we reference at FS:0x34 (KPCR!KdVersionBlock) is not guaranteed to exist in multiprocessor systems if you are not executing on the first processor.

Kernel Shellcoding...

The screenshot shows a web browser window with the following details:

- Title Bar:** Multi-Processors and KdVersionBlock - Matthieu S
- Address Bar:** http://www.msuiche.net/2009/01/05/multi-processors-and-kdversionblock/
- Toolbar:** Back, Forward, Stop, Refresh, New Tab.
- Menu Bar:** Bookmarks, Beans, Matasano, NinjaChat, computer stuff, RuXXer, Help.
- Content Area:** A blog post by Matthieu S. The visible text discusses a problem in the `KdGetDebuggerDataBlock` function related to reading the `KdVersionBlock` field from the 1st processor's KPCR. It includes a screenshot of a debugger session showing the memory dump of a `_KPCR` structure.
- Debugger Session:**

```
1kd> dt nt!_KPCR fffdff000
+0x000 NtTib          : _NT_TIB
+0x01c SelfPcr       : 0xffffdff000 _KPCR
+0x020 Prcb           : 0xffffdff120 _KPRCB
+0x024 Irql           : 0 ''
+0x028 IRR            : 0
+0x02c IrrActive      : 0
+0x030 IDR             : 0xffffffff
+0x034 KdVersionBlock : 0x805562b8
+0x038 IDT             : 0x8003f400 _KIDTENTRY
+0x03c GDT             : 0x8003f000 _KGDTENTRY
+0x040 TSS             : 0x80042000 _KTSS
+0x044 MajorVersion    : 1
+0x046 MinorVersion    : 1
+0x048 SetMember        : 1
+0x04c StallScaleFactor: 0x6bb
+0x050 DebugActive      : 0 ''
+0x051 Number           : 0 ''
+0x052 Spare0           : 0 ''
+0x053 SecondLevelCacheAssociativity: 0x10 ''
+0x054 VdmAlert         : 0
+0x058 KernelReserved   : 0141 0
```

Kernel Shellcoding...

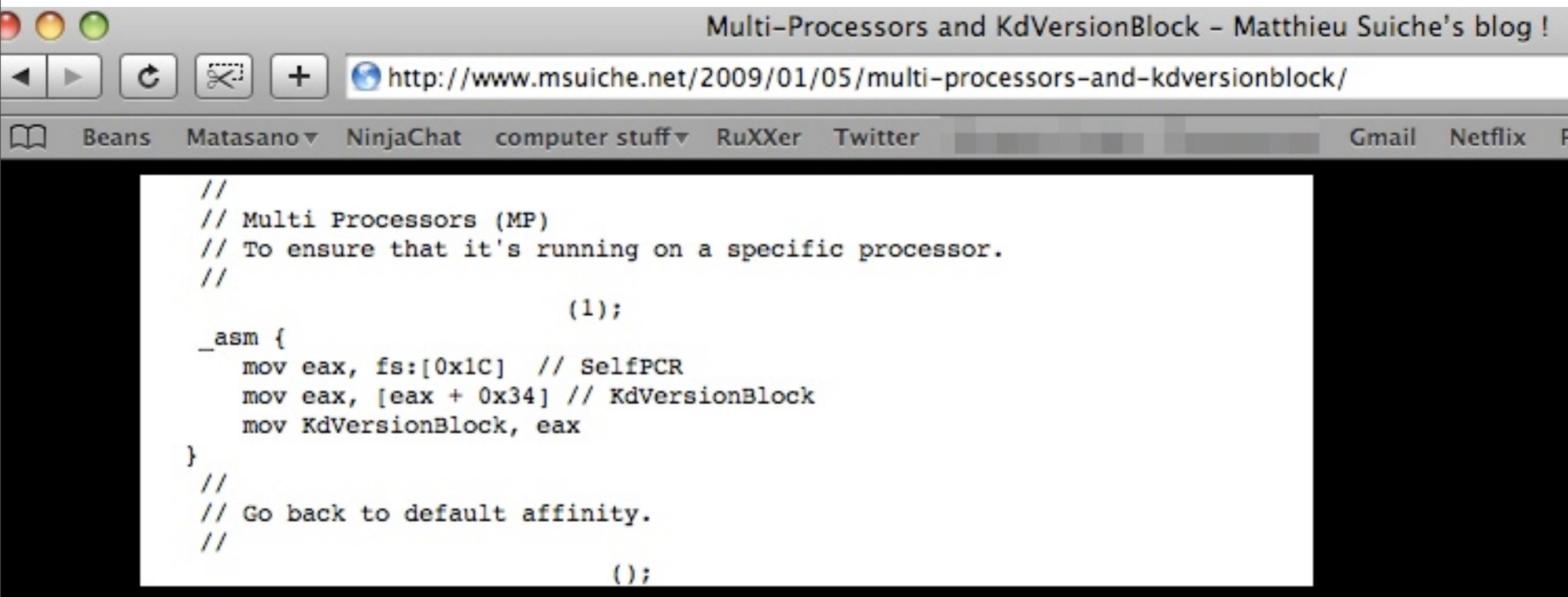
The screenshot shows the Immunity Debugger interface with a memory dump window. The title bar reads "Multi-Processors and KdVersionBlock - Matthieu S". The address bar contains the URL "http://www.msuiche.net/2009/01/05/multi-processors-and-kdversionblock/". The memory dump window displays the structure of the KPCR (Kernel Processor Control Block) at address f9c2c000. The dump shows various fields including NtTib, SelfPcr, Prcb, Irql, IRR, IrrActive, IDR, KdVersionBlock (which is highlighted in red), IDT, GDT, TSS, MajorVersion, MinorVersion, SetMember, StallScaleFactor, DebugActive, Number, Spare0, SecondLevelCacheAssociativity, VdmAlert, KernelReserved, SecondLevelCacheSize, HalReserved, InterruptMode, Spare1, KernelReserved2, and PrcbData.

```
+0x120 PrCbData      : _KPRCB
1kd> dt nt!_KPCR f9c2c000
+0x000 NtTib          : _NT_TIB
+0x01c SelfPcr        : 0xf9c2c000 _KPCR
+0x020 Prcb           : 0xf9c2c120 _KPRCB
+0x024 Irql            : 0 ''
+0x028 IRR             : 0
+0x02c IrrActive       : 0
+0x030 IDR              : 0xffffffff
+0x034 KdVersionBlock  : (null)
+0x038 IDT              : 0xf9c30590 _KIDTENTRY
+0x03c GDT              : 0xf9c30190 _KGDTENTRY
+0x040 TSS              : 0xf9c2cd70 _KTSS
+0x044 MajorVersion     : 1
+0x046 MinorVersion      : 1
+0x048 SetMember         : 2
+0x04c StallScaleFactor  : 0x650
+0x050 DebugActive       : 0 ''
+0x051 Number            : 0x1 ''
+0x052 Spare0           : 0 ''
+0x053 SecondLevelCacheAssociativity : 0x10 ''
+0x054 VdmAlert          : 0
+0x058 KernelReserved    : [14] 0
+0x090 SecondLevelCacheSize : 0x80000
+0x094 HalReserved       : [16] 1
+0x0d4 InterruptMode     : 0
+0x0d8 Spare1            : 0 ''
+0x0dc KernelReserved2   : [17] 0
+0x120 PrCbData          : _KPRCB
```

Kernel Shellcoding...

- ★ Mathieu Suiche (www.msuiche.net) has a note on this (instead of directly referencing KdVersionBlock) you first reference “selfPCR” at fs:0x1C
- ★ This is an example of interesting stuff you learn while developing/coding for the kernel! ;-)

Kernel Shellcoding...



Conclusions

- ★ Don't be intimidated by the kernel it's just another executable ;-)
- ★ Contrary to popular belief a lot of kernel stuff is surprisingly well documented
- ★ It's fun new territory (for me at least)...

Links, Notes, References...

Get links to everything in this presentation
at:

[www.dontstuffbeansupyournose.com/
ucon09](http://www.dontstuffbeansupyournose.com/ucon09)

stephen@matasano.com

Special Thanks

Julio Cesar Fort (of course)

Matasano

Stephen C. Lawler

Nia

**THANK YOU
FOR LISTENING!
Good Luck!**

