Intro to Windows Kernel Security Development (uCON-Conference 2009)
Who I am.

Stephen A. Ridley
Senior Security Researcher/Consultant
(Matiasano 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 me, I like my presentations to be conversational...
1. NT Kernel Introz
“[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)
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...
## Many Object Types in NT NS

<table>
<thead>
<tr>
<th>Adapter</th>
<th>File</th>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback</td>
<td>IoCompletion</td>
<td>SymbolicLink</td>
</tr>
<tr>
<td>Controller</td>
<td>Job</td>
<td>Thread</td>
</tr>
<tr>
<td>DebugObject</td>
<td>Key</td>
<td>Timer</td>
</tr>
<tr>
<td>Desktop</td>
<td>KeyedEvent</td>
<td>Token</td>
</tr>
<tr>
<td>Device</td>
<td>Mutant</td>
<td>Type</td>
</tr>
<tr>
<td>Directory</td>
<td>Port</td>
<td>Waitable Port</td>
</tr>
<tr>
<td>Driver</td>
<td>Process</td>
<td>WindowsStation</td>
</tr>
<tr>
<td>Event</td>
<td>Profile</td>
<td>WMIGuid</td>
</tr>
<tr>
<td>EventPair</td>
<td>Section</td>
<td></td>
</tr>
</tbody>
</table>
Peeking at the NT Object NS
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.)
The DRIVER_OBJECT structure is how your driver registers a “dispatch” function. This dispatch is just a callback that gets called...

Think of this an oldskool token ring network. Every driver gets all data and decides whether it wants it.
The DRIVER_OBJECT “registration” would look something like:

```c
DriverObject->MajorFunction[IRP_MJDEVICE_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

Result: File Object filled in by NTFS

I/O Manager

Object Manager

IopParseDevice

Fs filter drivers

NTFS

Volume Mgr

Disk Driver

HAL

I/O Manager

ObOpenObjectByName

NtCreateFile

IRP
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

- Flags
- Buffer Pointers
- MDL Chain
- Thread's IRPs
- Completion/Cancel Info
  - Completion
  - APC block
  - Driver Queuing & Comm.

IRP Stack Locations
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() / IoCreateSymoliclink(), 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() )
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
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 :-)

Getting Debuggers setup...
Debugging Over Serial

★ Edit boot.ini on debugee

★ 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.
Debugee (server) VMX file

```plaintext
49  uuid.action = "create"
50
51  virtualHW.productCompatibility = "hosted"
52
53  unity.wasCapable = "TRUE"
54  vmotion.checkpointFBSize = "134217728"
55
56  hqfs.mapRootShare = "TRUE"
57  hqfs.linkRootShare = "TRUE"
58  isolation.tools.hqfs.disable = "FALSE"
59
60  gui.fullScreenAtRequestBody = "TRUE"
61  gui.defaultModeAtPowerOn = "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

```
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
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.
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 :-( ...
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.
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 PDriverObject DriverObject, IN UNICODE_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

```c
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.InputBufferSizeLength;
    outBuf = Irp->AssociatedIrp.SystemBuffer;
    outLen = irpStack->Parameters.DeviceIoControl.OutputBufferSizeLength;
    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
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = information;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}```
#ifndef __KHD_H__
#define __KHD_H__

// driver IOCTLs
#define KHD 0x3adb33f
#define IOCTL_EXEC_SHELLCODE CTL_CODE(KHD, 0x07, METHOD_BUFFERED, FILE_READ_ACCESS)
#endif

#define CTL_CODE( DeviceType, Function, Method, Access ) ( //
  ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) //
)
Cleanup and “blank” dispatch

```c
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;
}
```
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
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.”


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 Reverseemode 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”
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”
Reversing out IOCTL codes...

We start reading....

```c
; 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:

```assembly
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
```
Reversing out IOCTL codes...

★ Following into _AfdDispatchDeviceControl we see:

```plaintext
; __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

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, let’s CTRL-O and fix these values:
Reversing out IOCTL codes...

Voila!
(our IOCTLs)
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](http://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...
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...

```assembly
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 intake this 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 got in here somehow
```
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...

```plaintext
1; Kernel loader with ResetDisplay VGA Text Mode PoC
2; www.dontstuffbeansupyourownose.com

CPU 686
BITS 32

; Not optimized for size, space, speed, or much of anything
pushad
mov ebx, [fs:0x34]; KdVersionBlock in NTOSKRNL --> NOT VERIFIED this always points at NTOSKRNL
mov edx, 0x1000
; page-align
dec edx
not edx
and ebx, edx
not edx
inc edx

; ok, i got lazy here with register allocation, i'm bored of this, just import my func will ya!
jmp functable
get_funcs:
pop ebp
; this - terrible
```
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 problem is located inside `KdGetDebuggerDataBlock` function, when the function try to read `KdVersionBlock` field an invalid pointer is returned because this field is only valid in the 1st processor KPCR.

```
lkd> dt nt!_KPCR ffdff000
 +0x000 NtTib : _NT_TIB
 +0x01c SelfPcr : 0xfffff000 _KPCR
 +0x020 Prcb : 0xfffff120 _KPRCB
 +0x024 Irql : 0 '
 +0x028 IRR : 0
 +0x02c IrrActive : 0
 +0x030 IDR : 0xffffff
 +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 : 0x141 0
```
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! ;-}
// Multi Processors (MP)
// To ensure that it's running on a specific processor.
//
_asm {
    mov eax, fs:[0x1C]  // SelfPCR
    mov eax, [eax + 0x34]  // KdVersionBlock
    mov KdVersionBlock, eax
}

//
// Go back to default affinity.
//
()
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)…
Get links to everything in this presentation at:

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!