GET IN THE RING0

Graham Sutherland – Penetration Tester

CORPORATE INFORMATION

UK Headquarters

Portcullis Computer Security Ltd

Portcullis House
Unit 2, Century Court
Tolpits Lane
Watford
WD18 9RS
United Kingdom

Telephone: +44 (0) 20 8868 0098
Fax: +44 (0) 20 8868 0017
Email: enquiries@portcullis-security.com
Website: www.portcullis-security.com
Website: http://labs.portcullis.co.uk

US Headquarters

Portcullis Inc.

505 Montgomery Street
10th and 11th Floor
San Francisco
California
94111
United States

Telephone: +1 415 874 3101
Email: enquiries@portcullis-security.us
Website: www.portcullis-security.us

GREETINGS

- Hello
- Bonjour
- Guten abend
- Ciao
- Goedeavond
- Alatúlië
- Qaleghqa'neS
- uwotm8

WHOIS/THIS_ASSHAT

- Graham “gsuberland” Sutherland
- Penetration Tester at Portcullis
- Specialities: Binary Applications, Cryptography, Reverse Engineering, Hardware
- “Polynomial” on StackExchange
- HARNESS THE POWER OF THE PARTYHAT

DRIVER? I HARDLY KNOW 'ER!

- Weird stigma against ring0 / driver code
- Not that hard
- You're gonna learn! (but not in 7 days)

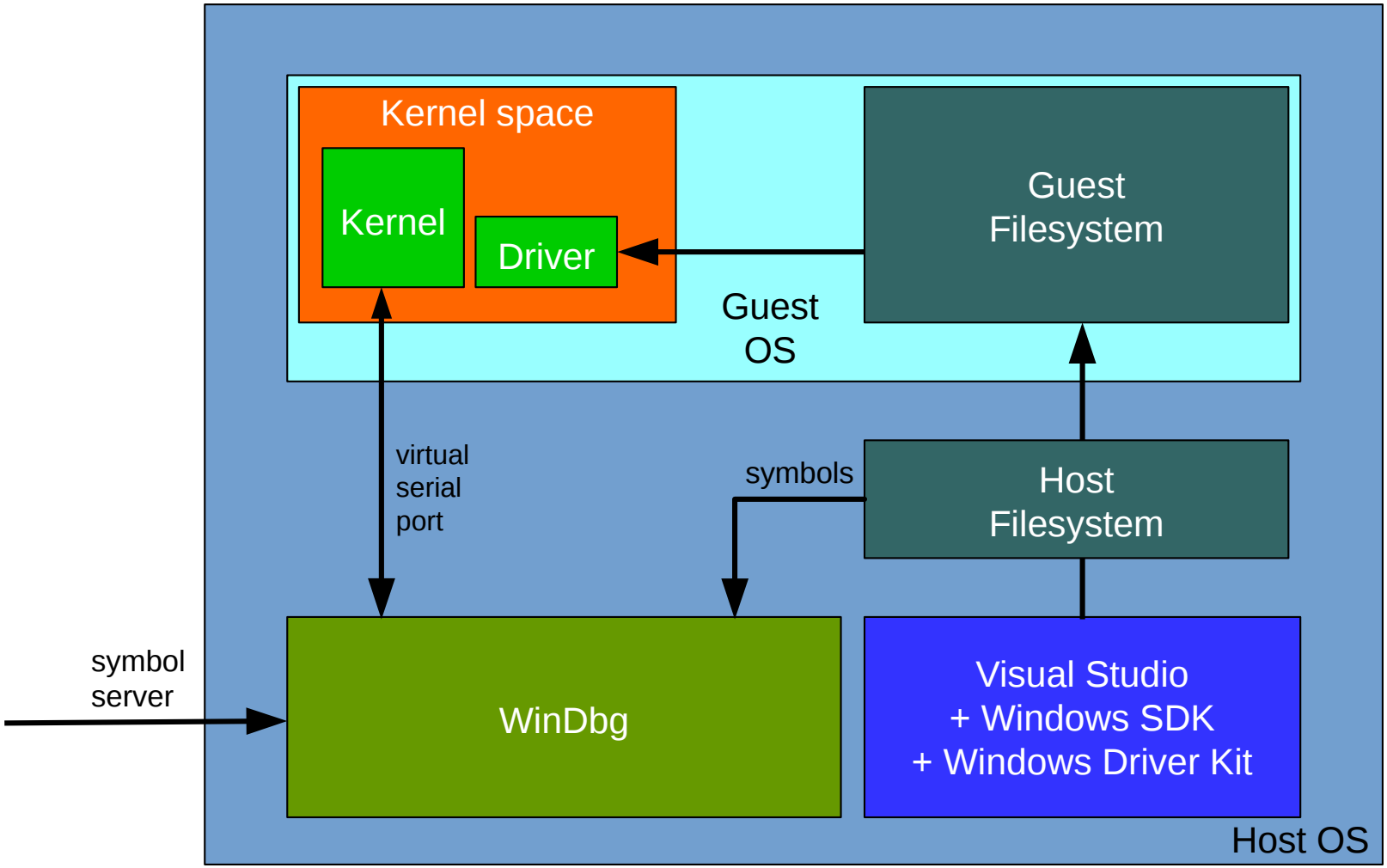
LEARN TO DRIVE(R)

- Same basic concepts as writing usermode apps
- Some additional bits
  - Talking between usermode / kernelmode
  - Major functions, IRPs, IOCTLs
  - Special concepts like IRQLs
- (mostly) officially documented on MSDN!
- (most of) the rest is reverse engineered.

YOU'RE DRIV(ER)ING ME CRAZY

- Soooooooooooooooo many abbreviations
- Setting up the initial environment can be a PITA
- Test signing is annoying (you can turn it off!)
- WinDbg has a learning curve
- Debugging can sometimes be clunky
  - Y U NO ATTACH!?!?!
- Some of this got better with recent WDK (e.g. 8.1)

GENERAL SETUP



REQUIRED TOOLS

- Virtual Machine
  - Virtual serial ports
  - Clipboard and directory sharing useful
  - VirtualBox / VMWare are good options
- Windows Driver Kit (WDK)
- Debugging Tools for Windows (WinDbg)
- Visual Studio
- SysInternals suite
- Notepad++ or similar is useful

ENVIRONMENT [1/3]

- Set up VM
  - Install OS
  - Set up VM tools package
  - Install SysInternals suite
  - Configure full (or at least full kernel) crash dumps
  - Set up shared directory to drop new driver builds and test harnesses into
  - Set up virtual serial port at max baud rate, tied to a pipe on the host system
  - Turn off driver signature enforcement (?)
  - Use bcdedit to enable kernel debugging

ENVIRONMENT [2/3]

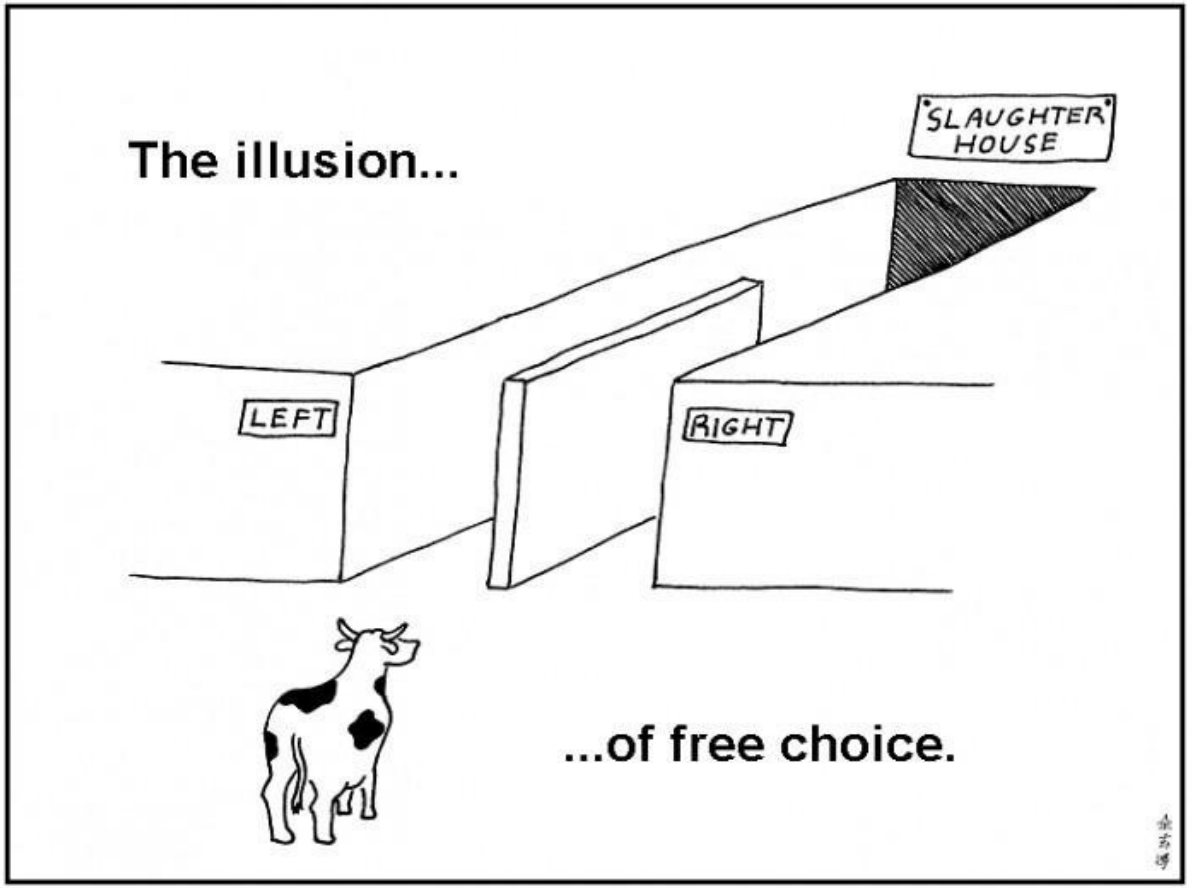
- Configure host machine
  - Install Windows SDK, WDK, Visual Studio, etc.
  - Set up WinDbg to use Microsoft symbol servers and local symbol cache directory
  - Configure WinDbg to attach to pipe for kernel debugging
  - Set up Visual Studio for building drivers
  - If needed, set up the tools for test signing

ENVIRONMENT [3/3]

- Check everything works!
  - Boot the VM with kernel debugging enabled
  - Make sure that WinDbg attaches
  - Use break and the 'g' command to continue
  - Use 'dbgview' to view debug output messages on the guest machine
    - Capture → Capture Global Win32
    - Capture → Capture Kernel
    - Capture → Enable Verbose Kernel Output

PICK A DRIVER

- What kind of driver should you create?
  - KMDF?
  - UMDF?
  - WDM?
  - Filter?
  - Protocol?
  - Hardware?
  - Software?
  - Filesystem?
  - Miniport?
  - Miniclass?
  - Minidriver?
  - Minibus?
  - Taxi?
  - Limo?




SCREW IT

SHUT UP AND DRIVE(R)

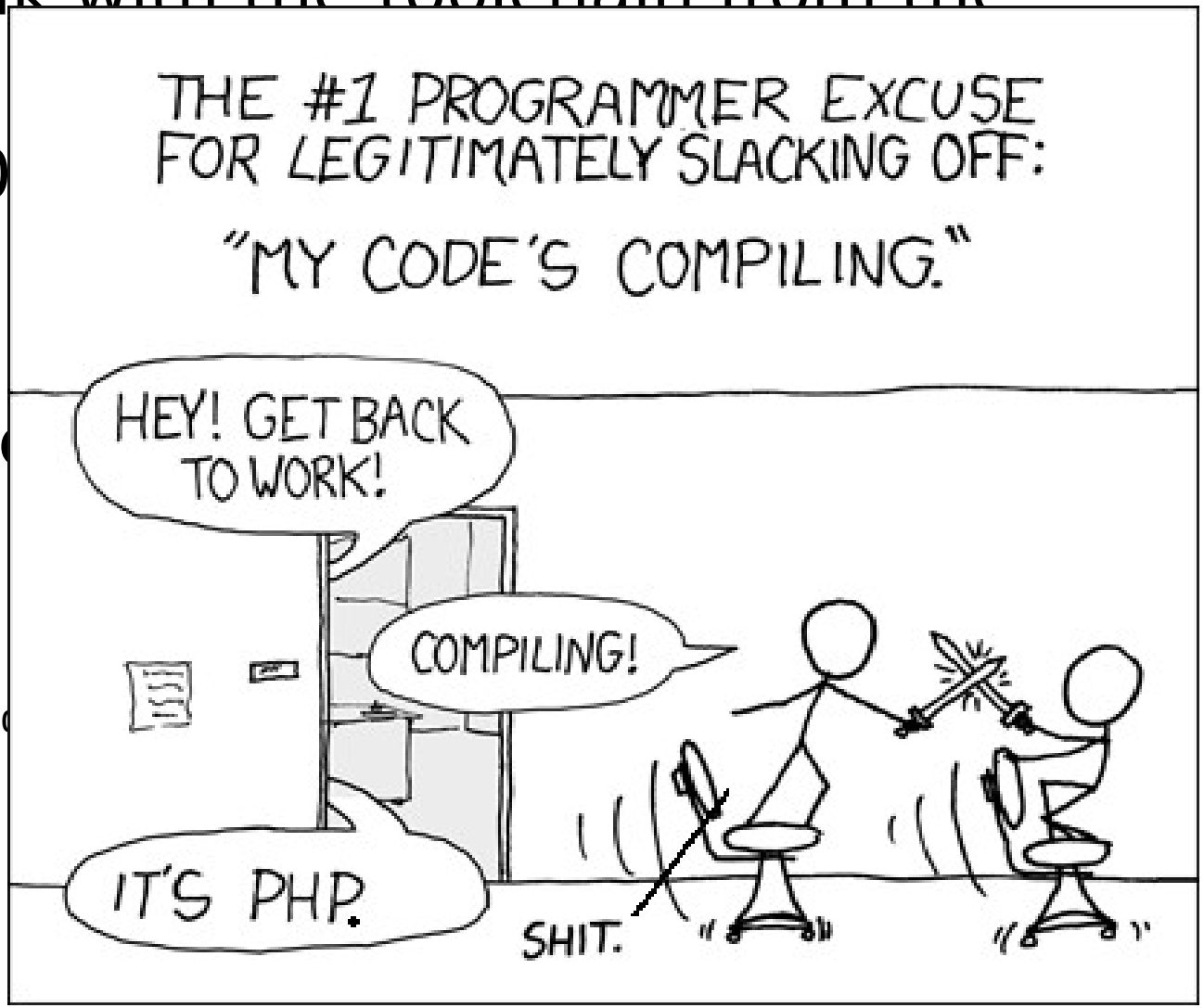
```
00 #include <ntddk.h>
01
02 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
03 {
04     DbgPrint("I <3 alpacas!\n");
05     return STATUS_SUCCESS;
06 }
```

COMPILE ALL THE THINGS

- Compile & link with the toolchain from the WDK
- Since WDK10 it's fully integrated with VS2015
- Or… you can use 'msbuild' from the command line!
- Produces the .sys (and .inf, maybe?)
- Install using 'sc':
  - `sc create mydriver binPath= c:\driver.sys type= kernel`

COMPILE ALL THE THINGS

- Compile & link with the toolchain from the WDK
- Since WDK10
  VS2015
- Or… you can
  command lin
- Produces the
- Install using '
  - `sc create myc`



OK, SO NOW WHAT?

- You just wrote a driver, but it has no female sheep.
  - (it's *yews-less*)

DO SOMETHING USEFUL

- Creating a driver handle
  - IoCreateDevice
- Communicating with the driver
  - I/O Request Packets (IRPs)
  - Major functions
  - Custom IOCTLs
  - Shared memory
- Synchronisation
  - Mutexes, semaphores, events, etc.

DEVICE OBJECT CREATION

```
00 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
01 {
02     PDEVICE_EXTENSION pExt;
03     PDEVICE_OBJECT pDevice;
04     NTSTATUS status;
05
06     status = IoCreateDevice(DriverObject, sizeof(DEVICE_EXTENSION),
07                             &szDriverName, FILE_DEVICE_UNKNOWN,
08                             FILE_DEVICE_SECURE_OPEN, FALSE, &pDevice);
09
10     if (NT_SUCCESS(status))
11     {
12         pDevice->Flags &= ~DO_DEVICE_INITIALIZING;
13         pDevice->Flags |= DO_BUFFERED_IO;
14         pExt = pDevice->DeviceExtension;
15         pExt->DeviceObject = pDevice;
16     }
17
18     return STATUS_SUCCESS;
19 }
```

MAJOR FUNCTION HANDLERS

```
00 #include <ntddk.h>
01
02 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
03 {
04     /* . . . */
05
06     DriverObject->DriverUnload = DriverUnload;
07     DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
08     DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
09     DriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
10     DriverObject->MajorFunction[IRP_MJ_WRITE] = DispatchWrite;
11     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchDeviceControl;
12     DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
13     DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnP;
14
15     return STATUS_SUCCESS;
16 }
```

DISPATCH HANDLERS

- Used to perform certain actions when I/O calls are performed on the device object, or when certain events occur.
- Standard I/O
  - `CreateFile` – e.g. initialise context on driver-side
  - `CloseFile` – e.g. clear context on driver-side
  - `ReadFile` – buffered I/O
  - `WriteFile` – buffered I/O
- Custom I/O
  - `DeviceIoControl` – custom I/O control codes
- Events
  - Power events
  - Plug-n-Play (PnP) events

I/O REQUEST PACKETS

- IRPs
- Each call to a usermode I/O API sends an IRP to the driver.
- The IRP contains the major function (MJ) number and some optional data, plus information about I/O buffers.
- Driver dispatches the IRP to an appropriate handler using the dispatch table (MajorFunctions)

MAJOR FUNCTION HANDLERS

```
00 #include <ntddk.h>
01
02 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
03 {
04     /* . . . */
05
06     DriverObject->DriverUnload = DriverUnload;
07     DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
08     DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
09     DriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
10     DriverObject->MajorFunction[IRP_MJ_WRITE] = DispatchWrite;
11     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchDeviceControl;
12     DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
13     DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnP;
14
15     return STATUS_SUCCESS;
16 }
```

HANDLING IRPs

- Get stack location with `IoGetCurrentIrpStackLocation`
- Access IRP parameters via `PIO_STACK_LOCATION`
  - `stack→Parameters` contains a union struct
    - Create
    - Read
    - Write
    - Close
    - DeviceIoControl
    - Etc…
- Use `CompleteRequest` to complete the IRP

BUFFERED I/O

- Buffers are not “shared” directly between userspace and kernelspace; they are exchanged.
- Obvious use-case: data transfer on hardware device drivers, e.g. NIC or disk.
- Relatively trivial, but takes up a fair bit of code to show a working example, so won't show it in this presentation.

CUSTOM CONTROL CODES

```
00 #include <ntddk.h>
01
02 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
03 {
04     /* . . . */
05
06     DriverObject->DriverUnload = DriverUnload;
07     DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
08     DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
09     DriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
10     DriverObject->MajorFunction[IRP_MJ_WRITE] = DispatchWrite;
11     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchDeviceControl;
12     DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
13     DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnP;
14
15     return STATUS_SUCCESS;
16 }
```

I/O CONTROL CODES

- IOCTLs
- “Custom” functionality in drivers
- Just like any other IRP, but has an integer code that identifies which IOCTL is being called.
- Triggered via `DeviceIoControl` calls.
- Usually dispatched in a switch statement in the `IRP_MJ_DEVICE_CONTROL` handler.
- Always fun to reverse engineer drivers by looking for these; lots of fun findings!

EXAMPLE IOCTL DISPATCH

```
00 NTSTATUS DispatchControl(PDEVICE_OBJECT Device, PIRP Irp)
01 {
02     NTSTATUS status = STATUS_SUCCESS;
03     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
04     ULONG cbInput = stack->Parameters.DeviceIoControl.InputBufferLength;
05     ULONG cbOutput = stack->Parameters.DeviceIoControl.OutputBufferLength;
06     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
07
08     switch(code)
09     {
10         case 0x13371234:
11             status = DispatchSomething(Device, Irp);
12             break;
13         default:
14             status = STATUS_INVALID_DEVICE_REQUEST;
15             break;
16     }
17     return status;
18 }
```

EXAMPLE USERMODE CALL

```
00 HANDLE hDevice;
01 hDevice = CreateFile("\\\\.\\device\\MyDevice", GENERIC_WRITE, 0, NULL,
02                      OPEN_EXISTING, 0, NULL);
03
04 if (hDevice == INVALID_HANDLE_VALUE)
05 {
06     printf("Couldn't open device.\n");
07 }
08 else
09 {
10     BOOL status;
11     status = DeviceIoControl(hDevice, 0x13371234, NULL, 0, NULL, 0, NULL, NULL);
12     if (status == TRUE)
13     {
14        // . . .
15     }
16 }
17
18
```

MEMORY ACCESS

- Paged vs unpaged
  - Paged may not be in system memory!
- Be careful of IRQLs (interrupt masking)

INTERRUPT REQUEST LEVELS

- IRQL is an integer from 0 to 31
- DISPATCH is “normal” level.
- Other levels used to mask interrupts.
- Don't want memory fetches interrupted by mouse movement interrupts or similarly less-important interrupts.
- Accessing paged memory at IRQL > DISPATCH gives you instant BSoD:
  - IRQL_NOT_LESS_OR_EQUAL

END OF PRESENTATION

Presentation by: Graham Sutherland
Job Title: Penetration Tester
Twitter: @gsuberland