Revealing Embedded Fingerprints: Deriving intelligence from USB stack interactions

Andy Davis, Research Director NCC Group
UK Offices
Manchester - Head Office
Cheltenham
Edinburgh
Leatherhead
London
Thame

North American Offices
San Francisco
Atlanta
New York
Seattle

Australian Offices
Sydney

European Offices
Amsterdam - Netherlands
Munich – Germany
Zurich - Switzerland
Agenda

Part One:
• Overview of the USB enumeration phase
• Different USB stack implementations
• USB testing platform
• Installed drivers and supported devices
• Fingerprinting techniques
• Umap demo

Part Two:
• The Windows 8 RNDIS kernel pool overflow
• Challenges faced when exploiting USB bugs
• Conclusions
Part One: Information gathering

- Why do we care?
- If you connect to a device surely you already know the platform?
- Embedded devices are mostly based on Linux anyway aren't they?
- Allows you to focus your testing on only supported functionality
USB Background stuff
Overview of the USB enumeration phase

• What is enumeration for?
  • Assign an address
  • Speed of communication
  • Power requirements
  • Configuration options
  • Device descriptions
  • Identify class drivers

• Lots of information exchange – implemented in many different ways
The USB enumeration phase

< Get Device descriptor
> Set Address
< Get Device descriptor
< Get Configuration descriptor
< Get String descriptor 0
< Get String descriptor 2
< Get Configuration descriptor
< Get Configuration descriptor
> Set Configuration
Enumeration phase peculiarities

• Why is the device descriptor initially requested twice?

• Why are there multiple requests for other descriptors?

• Class-specific descriptors:
  < Get Hub descriptor
  < Get HID Report descriptor
Different USB stack implementations

- Typical components of a USB stack
- Windows USB driver stack
- Linux USB stack
- Embedded Access USB stack
Typical components of a USB stack

• Host Controller hardware

• USB System software:
  • Host Controller Driver – Hardware Abstraction Layer
  • USB Driver

• Class drivers

• Application software

Image from: www.wired.com
Windows USB driver stack

User Mode

Kernel Mode

USB Client Driver Layer

Data Flow

USB 3.0 Driver Stack

USB 2.0 Driver Stack

Hardware

Microsoft-Provided Inbox drivers

Microsoft-Provided Helper Libraries

Client or Class Driver

Legend

Image from: msdn.microsoft.com
Linux USB stack

(user-space) (Class drivers) (vendor-specific)

mp3 hub HID webcam

URBs or func parms

usbdts/ 
prodfs

usb_operations: URB or dev*

OHCI HCD

UHCI HCD

callbacks(urb)
Embedded Access USB stack
Interacting with USB
USB interaction requirements

• Need to capture and replay USB traffic

• Full control of generated traffic

• Class decoders extremely useful

• Support for Low/High/Full speed required

• USB 3.0 a bonus
USB testing – gold-plated solution

• Commercial test equipment
USB testing – the cheaper approach

• Facedancer (http://goodfet.sourceforge.net/hardware/facedancer21)
Best solution: A combination of both

- Device data can be carefully crafted
- Host response data can be captured
- Microsecond timing is also recorded
- All class-specific data is decoded
Information enumeration
Target list

• Windows 8
• Ubuntu Linux 12.04 LTS
• Apple OS X Lion
• FreeBSD 5.3
• Chrome OS
• Linux-based TV STB
Installed drivers and supported devices

- Enumerating supported class types – standard USB drivers
- Enumerating all installed drivers
- Other devices already connected
Enumerating supported class types

Where is USB class information stored?

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>18</td>
<td>Valid Length</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>1</td>
<td>DEVICE</td>
</tr>
<tr>
<td>bcdUSB</td>
<td>0x0200</td>
<td>Spec Version</td>
</tr>
<tr>
<td>bDeviceClass</td>
<td>0x09</td>
<td>Hub</td>
</tr>
<tr>
<td>bDeviceSubClass</td>
<td>0x00</td>
<td>Full Speed Hub</td>
</tr>
<tr>
<td>bDeviceProtocol</td>
<td>0x01</td>
<td></td>
</tr>
</tbody>
</table>

Device Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>9</td>
<td>Valid length</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>4</td>
<td>INTERFACE</td>
</tr>
<tr>
<td>bInterfaceNumber</td>
<td>0</td>
<td>Zero-based Number of this Interface.</td>
</tr>
<tr>
<td>bAlternateSetting</td>
<td>0</td>
<td>Value used to select this alternative setting for the interface identified in the prior field</td>
</tr>
<tr>
<td>bNumEndpoints</td>
<td>3</td>
<td>Number of endpoints used by this interface (excluding endpoint zero).</td>
</tr>
<tr>
<td>bInterfaceClass</td>
<td>0x06</td>
<td>Image</td>
</tr>
<tr>
<td>bInterfaceSubClass</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>bInterfaceProtocol</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>iInterface</td>
<td>0</td>
<td>Index of string descriptor describing this Interface</td>
</tr>
</tbody>
</table>

Interface Descriptor
Installed drivers and supported devices

- Drivers are referenced by class (Device and Interface descriptors)
- Also, by VID and PID:

| idVendor | 0x090C | Silicon Motion, Inc. - Taiwan |
| idProduct | 0x1000 | Memory Bar |

- For each device class VID and PID values can be brute-forced (can easily be scripted using Facedancer)
- Although there may be some shortcuts….  
- Valid PIDs and VIDs are available (http://www.linux-usb.org/usb.ids)
Enumerating installed drivers

Not installed:  

< Get Device descriptor  
> Set Address  
< Get Device descriptor  
< Get Configuration descriptor  
< Get String descriptor 0  
< Get String descriptor 2  
< Get Configuration descriptor  
> Set Configuration

All communication stops after “Set Configuration”

Installed:  

< Get Device descriptor  
> Set Address  
< Get Device descriptor  
< Get Configuration descriptor  
< Get String descriptor 0  
< Get String descriptor 2  
< Get Configuration descriptor  
< Get Configuration descriptor  
> Set Configuration  
> Set Idle (HID)  
< Get HID Report descriptor  
> Set Report (HID)
Sniffing the bus - Other connected devices

- Data from other devices will be displayed on other addresses

- Controlling other devices? (untested)
Fingerprinting techniques

- Descriptor request patterns
- Timing information
- Descriptor types requested
- Responses to invalid data
- Order of Descriptor requests
## OS Identification

### Linux-based TV STB

- < Get Max LUN (Mass Storage)
- > **CBW: INQUIRY**
- < MSC Data In
- < CSW - Status Passed
- > **CBW: TEST UNIT READY**
- < CSW - Status Passed
- > **CBW: READ CAPACITY**
- < MSC Data In
- < CSW - Status Passed
- > **CBW: MODE SENSE**

### Windows 8

- < Get Max LUN (Mass Storage)
- > **CBW: INQUIRY**
- < MSC Data In
- < CSW - Status Passed
- > **CBW: INQUIRY**
- < MSC Data In
- < CSW - Status Passed
- > **CBW: READ FORMAT CAPACITIES**
- < MSC Data In
- < CSW - Status Passed
Application identification

gphoto2 (Linux)
> Image: OpenSession
< Image: OK
> Image: GetDeviceInfo
< Image: DeviceInfo
< Image: OK
> Image: GetStorageIDs
< Image: StorageIDs
< Image: OK
> Image: GetStorageInfo
< Image: StorageInfo
< Image: OK
> Image: CloseSession
< Image: OK

“Photos” Metro app (Windows 8)
> Image: OpenSession
< Image: OK
> Image: GetDeviceInfo
< Image: DeviceInfo
< Image: OK
> Image: GetStorageIDs
< Image: StorageIDs
< Image: OK
> Image: SetDevicePropValue
> Image: DeviceProperty
< Image: OK
< Image: DeviceInfoChanged

DeviceProperty includes some text:
/Windows/6.2.9200
MTPClassDriver/6.2.9200.16384
Request patterns unique elements?

- Windows 8 (HID) – 3 x **Get Configuration** descriptor requests (others have two)
- Apple OS X Lion (HID) – **Set Feature** request right after **Set Configuration**
- FreeBSD 5.3 (HID) – **Get Status** request right before **Set Configuration**
- Linux-based TV STB (Mass Storage) – Order of class-specific requests
Timing information (work in progress...)

<table>
<thead>
<tr>
<th>Capture13</th>
<th>Capture14</th>
<th>Capture15</th>
<th>Capture16</th>
<th>Capture17*</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4..18</td>
<td>#4..18</td>
<td>#4..18</td>
<td>#4..18</td>
<td></td>
</tr>
<tr>
<td>3.035.975</td>
<td>3.150.387</td>
<td>2.846.062</td>
<td>3.385.656</td>
<td></td>
</tr>
<tr>
<td>#20..25</td>
<td>#20..25</td>
<td>#20..25</td>
<td>#20..25</td>
<td></td>
</tr>
<tr>
<td>3.085.990</td>
<td>3.211.403</td>
<td>2.893.073</td>
<td>3.303.360</td>
<td></td>
</tr>
<tr>
<td>#26..40</td>
<td>#26..40</td>
<td>#26..40</td>
<td>#26..40</td>
<td></td>
</tr>
<tr>
<td>3.104.993</td>
<td>3.232.408</td>
<td>2.924.081</td>
<td>3.301.365</td>
<td></td>
</tr>
<tr>
<td>#41..61</td>
<td>#41..61</td>
<td>#41..61</td>
<td>#41..61</td>
<td></td>
</tr>
<tr>
<td>3.110.995</td>
<td>3.238.409</td>
<td>2.930.084</td>
<td>3.307.366</td>
<td></td>
</tr>
<tr>
<td>#62..70</td>
<td>#62..70</td>
<td>#62..70</td>
<td>#62..70</td>
<td></td>
</tr>
<tr>
<td>3.122.997</td>
<td>3.249.410</td>
<td>2.941.083</td>
<td>3.319.369</td>
<td></td>
</tr>
<tr>
<td>#71..97</td>
<td>#71..97</td>
<td>#71..97</td>
<td>#71..97</td>
<td></td>
</tr>
<tr>
<td>3.127.999</td>
<td>3.254.413</td>
<td>2.946.088</td>
<td>3.324.370</td>
<td></td>
</tr>
<tr>
<td>#88..112</td>
<td>#88..112</td>
<td>#88..112</td>
<td>#88..112</td>
<td></td>
</tr>
<tr>
<td>3.637.094</td>
<td>3.333.430</td>
<td>4.000.085</td>
<td>3.765.370</td>
<td></td>
</tr>
<tr>
<td>#113..124</td>
<td>#113..124</td>
<td>#113..124</td>
<td>#113..124</td>
<td></td>
</tr>
<tr>
<td>3.543.095</td>
<td>3.338.430</td>
<td>4.000.085</td>
<td>3.765.370</td>
<td></td>
</tr>
<tr>
<td>#125..145</td>
<td>#125..145</td>
<td>#125..145</td>
<td>#125..145</td>
<td></td>
</tr>
<tr>
<td>3.548.095</td>
<td>3.344.433</td>
<td>3.032.104</td>
<td>3.760.473</td>
<td></td>
</tr>
<tr>
<td>#146..151</td>
<td>#146..151</td>
<td>#146..151</td>
<td>#146..151</td>
<td></td>
</tr>
<tr>
<td>3.556.098</td>
<td>3.352.435</td>
<td>3.045.107</td>
<td>3.774.475</td>
<td></td>
</tr>
<tr>
<td>#152..157</td>
<td>#152..157</td>
<td>#152..157</td>
<td>#152..157</td>
<td></td>
</tr>
<tr>
<td>#158..190</td>
<td>#158..190</td>
<td>#158..190</td>
<td>#158..190</td>
<td></td>
</tr>
<tr>
<td>3.552.101</td>
<td>3.358.438</td>
<td>3.051.109</td>
<td>3.780.475</td>
<td></td>
</tr>
<tr>
<td>#191..199</td>
<td>#191..199</td>
<td>#191..199</td>
<td>#191..199</td>
<td></td>
</tr>
<tr>
<td>3.612.112</td>
<td>3.403.447</td>
<td>3.089.118</td>
<td>3.834.489</td>
<td></td>
</tr>
</tbody>
</table>

End of Capture
Timing information (work in progress…)

<table>
<thead>
<tr>
<th>Capture13</th>
<th>Capture14</th>
<th>Capture15</th>
<th>Capture16</th>
<th>Capture17*</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4_18</td>
<td>#4_18</td>
<td>#4_18</td>
<td>#4_18</td>
<td>LS</td>
</tr>
<tr>
<td>3.035,975 s</td>
<td>3.150,387 s</td>
<td>2.846,062 s</td>
<td>3.858,656 s</td>
<td>Control Transfer</td>
</tr>
<tr>
<td>#20_25</td>
<td>#20_25</td>
<td>#20_25</td>
<td>#20_25</td>
<td>Addr Endp</td>
</tr>
<tr>
<td>3.085,993 s</td>
<td>3.211,403 s</td>
<td>2.933,073 s</td>
<td>3.983,665 s</td>
<td>Addr Endp</td>
</tr>
<tr>
<td>#26_40</td>
<td>#26_40</td>
<td>#26_40</td>
<td>#26_40</td>
<td>Data (18 bytes)</td>
</tr>
<tr>
<td>3.104,993 s</td>
<td>3.232,408 s</td>
<td>2.924,815 s</td>
<td>3.939,672 s</td>
<td>Data (18 bytes)</td>
</tr>
<tr>
<td>#41_61</td>
<td>#41_61</td>
<td>#41_61</td>
<td>#41_61</td>
<td>Status</td>
</tr>
<tr>
<td>3.119,993 s</td>
<td>3.238,409 s</td>
<td>2.930,084 s</td>
<td>3.945,678 s</td>
<td>Status</td>
</tr>
<tr>
<td>#62_70</td>
<td>#62_70</td>
<td>#62_70</td>
<td>#62_70</td>
<td>Status</td>
</tr>
<tr>
<td>3.122,993 s</td>
<td>3.249,410 s</td>
<td>2.957,035 s</td>
<td>3.957,678 s</td>
<td>Status</td>
</tr>
<tr>
<td>#71_97</td>
<td>#71_97</td>
<td>#71_97</td>
<td>#71_97</td>
<td>Status</td>
</tr>
<tr>
<td>3.127,993 s</td>
<td>3.254,413 s</td>
<td>2.946,654 s</td>
<td>3.982,677 s</td>
<td>Status</td>
</tr>
<tr>
<td>#98_112</td>
<td>#98_112</td>
<td>#98_112</td>
<td>#98_112</td>
<td>Status</td>
</tr>
<tr>
<td>3.637,094 s</td>
<td>3.333,430 s</td>
<td>4.006,085 s</td>
<td>3.755,678 s</td>
<td>Status</td>
</tr>
<tr>
<td>#113_124</td>
<td>#113_124</td>
<td>#113_124</td>
<td>#113_124</td>
<td>Status</td>
</tr>
<tr>
<td>3.543,094 s</td>
<td>3.333,430 s</td>
<td>4.006,085 s</td>
<td>3.755,678 s</td>
<td>Status</td>
</tr>
<tr>
<td>#125_145</td>
<td>#125_145</td>
<td>#125_145</td>
<td>#125_145</td>
<td>Status</td>
</tr>
<tr>
<td>3.548,096 s</td>
<td>3.344,430 s</td>
<td>4.011,677 s</td>
<td>3.786,473 s</td>
<td>Status</td>
</tr>
<tr>
<td>#146_151</td>
<td>#146_151</td>
<td>#146_151</td>
<td>#146_151</td>
<td>Status</td>
</tr>
<tr>
<td>3.556,096 s</td>
<td>3.344,430 s</td>
<td>4.011,677 s</td>
<td>3.786,473 s</td>
<td>Status</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#191_199</td>
<td>#191_199</td>
<td>#191_199</td>
<td>#191_199</td>
<td></td>
</tr>
<tr>
<td>3.612,112 s</td>
<td>3.403,447 s</td>
<td>3.089,118 s</td>
<td>4.055,698 s</td>
<td></td>
</tr>
</tbody>
</table>

== End of Capture ==
Using timing information? (work in progress…)

- Large amount of variance over entire enumeration phase:
  - 4.055s, 3.834s, 3.612s, 3.403s, 3.089s

- Much greater accuracy between specific requests:
  - Between String Descriptor #0 and #2 requests - 5002us, 5003us, 5003us, 4999us, 5001us

- If we know the OS can we potentially determine the processor speed?
Descriptor types requested

- Microsoft OS Descriptors (MOD)
- Used for “unusual” devices classes
- Devices that support Microsoft OS Descriptors must store a special USB string descriptor in firmware at the fixed string index of 0xEE. The request is:

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000B</td>
<td>GET_DESCRIPTOR</td>
<td>0x03EE</td>
<td>0x0000</td>
<td>0x12</td>
<td>Returned String</td>
</tr>
</tbody>
</table>
Responses to invalid data

- Different USB stacks respond to invalid data in different ways

- Maximum and minimum values

- Logically incorrect values

- Missing data

- In some cases: Crashes (potential vulnerabilities)

- Other cases: Unique behaviour
Invalid data unique elements?

Windows (all versions)

If you send a specific, logically incorrect HID Report descriptor this happens:
Invalid data unique elements?

Windows (all versions)

If you send a specific, logically incorrect HID Report descriptor this happens:
Order of Descriptor requests

- Some USB stacks request data from devices in a different order
- Different drivers may request different descriptors multiple times
- Sometimes descriptors are re-requested after enumeration is complete
Demo: umap
Umap overview

• Supported device classes can be enumerated
• Operating system information can be enumerated
• Devices with specific VID/PID/REV can be emulated
• The enumeration phase and class-specific data can be fuzzed
• Endpoint protection systems configuration can be assessed
• Endpoint protection systems USB protection can be circumvented
• USB host implementations can be comprehensively tested
Part Two: Potentially exploitable USB bugs
The Windows 8 RNDIS kernel pool overflow

- MS13-027

- `usb8023x.sys` - default (Microsoft-signed) Windows Remote NDIS driver that provides network connectivity over USB.

- When the following USB descriptor field is manipulated a Bug check occurs indicating a kernel pool overwrite:
  
  Configuration descriptor: `bNumInterfaces` field > actual number of USB interfaces
The Bug Check

BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.

<Truncated for brevity>

Arguments:
**Arg1: 00000020, a pool block header size is corrupt.**
Arg2: 83e38610, The pool entry we were looking for within the page.
Arg3: 83e38690, The next pool entry.
Arg4: 08100008, (reserved)

<Truncated for brevity>

WARNING: SystemResourcesList->Flink chain invalid. Resource may be corrupted, or already deleted.

WARNING: SystemResourcesList->Blink chain invalid. Resource may be corrupted, or already deleted.

**SYMBOL_NAME:** usb8023x!SelectConfiguration+1bd
The `SelectConfiguration()` function

```
SelectConfiguration(x)
SelectConfiguration(x)+2
SelectConfiguration(x)+3
SelectConfiguration(x)+5
SelectConfiguration(x)+8
SelectConfiguration(x)+9
SelectConfiguration(x)+A
SelectConfiguration(x)+D
SelectConfiguration(x)+E
SelectConfiguration(x)+11
SelectConfiguration(x)+14
SelectConfiguration(x)+16
SelectConfiguration(x)+1C
SelectConfiguration(x)+1F
SelectConfiguration(x)+26
SelectConfiguration(x)+27
SelectConfiguration(x)+2C
SelectConfiguration(x)+2F
SelectConfiguration(x)+31
SelectConfiguration(x)+37
SelectConfiguration(x)+39
SelectConfiguration(x)+3C
SelectConfiguration(x)+3E

mov    edi, edi
push   ebp
mov    ebp, esp
sub    esp, 10h
push   ebx
push   esi
mov    esi, [ebp+ptr_Pool_U802]
push   edi
mov    edi, [esi+1Ch] ; points to start of configuration descriptor
mov    al, [edi+4]   ; al = bNumInterfaces
cmp    al, 2         ; compares with 2 (what it should be)
jb     loc_11877     ; no jump
movzx  eax, al
lea    eax, ds:8[eax*8] ; multiply bNumInterfaces by 8 then add 8 = 24
push   eax
call   _AllocPool@4  ; AllocPool(x)
mov    [ebp+ptr_Pool_U802_24_bytes], eax
test   eax, eax
jz     loc_11877     ; no jump (AllocPool was successful)
xor    ebx, ebx
cmp    [edi+4], bl   ; compares bNumInterfaces with 0
jbe    short loc_1171F ; no jump
mov    esi, eax
```
The crash point

```
setConfiguration(x)+9B
setConfiguration(x)+9B
setConfiguration(x)+9D
setConfiguration(x)+9F
setConfiguration(x)+A1
setConfiguration(x)+A3
setConfiguration(x)+A4
setConfiguration(x)+A5
setConfiguration(x)+A6
setConfiguration(x)+AC
setConfiguration(x)+B0
setConfiguration(x)+B3
setConfiguration(x)+B6
setConfiguration(x)+B8
setConfiguration(x)+B8
setConfiguration(x)+B8
setConfiguration(x)+B8
setConfiguration(x)+BC
setConfiguration(x)+BC
setConfiguration(x)+BF
setConfiguration(x)+C2
setConfiguration(x)+C2
setConfiguration(x)+C8
setConfiguration(x)+C9
setConfiguration(x)+CC
setConfiguration(x)+CE

yet_more_interfaces_to_parse:
    push 0FFFFFFFFh
    push 0FFFFFFFFh
    push 0
    push ecx
    push edi
    call ds:imp_USB_ParserConfigurationDescriptorEx028
    test eax, eax
    jz short loc_11770
    mov al, [eax+5]
    mov [esi+4], al
    jmp short loc_11774

loc_11770:
    mov byte ptr [esi+4], 0 ; writes one null byte over the first byte of the next pool header
    ; this is where the corruption occurs

loc_11774:
    movzx eax, word ptr [esi]
    mov ecx, [ebp+ptr_Pool_U802_24_bytes]
    add esi, eax
    movzx eax, byte ptr [edi+4]
    inc ecx
    mov [ebp+ptr_Pool_U802_24_bytes], ecx
    cmp ecx, eax
    jb short yet_more_interfaces_to_parse
```
Analysis #1

When bNumInterfaces = 3 (one more than it should be) and bNumEndpoints = 2 (valid value)

Next kernel pool:

849c3b28 10 00 0a 04 56 61 64 6c-6b 8f 94 85 28 8c 90 85 ....Vadlk...(...

becomes:

849c3b28 00 00 0a 04 56 61 64 6c-6b 8f 94 85 28 8c 90 85 ....Vadlk...(...

So we’re overwriting "PreviousSize" in the next nt!_POOL_HEADER - this is what triggered the original Bug Check when ExFreePool() is called
Analysis #2

When `bNumInterfaces = 3` (one more than it should be) and `bNumEndpoints = 5` (three more than it should be)

Next kernel pool:

```
84064740  17 00 03 00 46 72 65 65-48 2d 09 84 30 a8 17 84 .....FreeH...0...
```

becomes:

```
84064740  17 00 03 00 00 72 65 65-48 2d 09 84 30 a8 17 84 .....reeH...0...
```

So we’re now overwriting "PoolTag" in the next `nt!POOL_HEADER`
What’s going on?

kd> dt nt!_POOL_HEADER
  - +0x000 PreviousSize : Pos 0, 8 Bits
  - +0x000 PoolIndex : Pos 8, 8 Bits
  - +0x000 BlockSize : Pos 16, 8 Bits
  - +0x000 PoolType : Pos 24, 8 Bits
  - +0x004 PoolTag : Uint4B
  - +0x008 ProcessBilled : Ptr64 _EPROCESS

By manipulating bNumInterfaces and bNumEndpoints in a USB Configuration descriptor we appear to have a degree of control over where in the next adjacent kernel memory pool we can overwrite a single byte with a null (the null write occurs four bytes after the end of the pool I control and I can also control its size and some elements of its contents so could also potentially overwrite the next pool header with something useful)
Some pseudo code

```c
for (i=0; i<something->count; i++)
{
    list[i].descriptor = USB_D_ParseConfigurationDescriptorEx (...);
    
    if(!list[i].descriptor)
    {
        break;
    }
}

list[i].descriptor = NULL;

newthing = USB_CreateConfigurationRequestEx(thing, list);

if(newthing)
{
    ptr = &newthing->somemember;
    for (i=0; i<something->count; i++)
    {
        descriptor = USB_D_ParseConfigurationDescriptorEx (...);
        
        if(descriptor)
        {
            ptr->someothermember = descriptor->whatever;
        }
        else
        {
            ptr->someothermember = 0;  // this is where I believe the corruption happens
        }
        ptr = ptr + ptr->Length;
    }
}
```
Challenges faced when exploiting USB bugs

- Lack of feedback channel
- The bug is often in kernel code
- Descriptors are generally very size-constrained
- Typical impact of USB exploitation typically restricted to privilege escalation
- Modern operating systems e.g. Windows 8 have comprehensive exploit mitigation
- What about USB over RDP?
Conclusions

• The USB enumeration phase reveals useful information for fingerprinting

• Class-specific communication is potentially even more revealing

• Even vendors with mature SDL processes have USB bugs

• USB bugs can potentially be exploited, to provide privilege escalation

• …but it is extremely difficult to achieve reliably
Questions?

Andy Davis, Research Director NCC Group
andy.davis ‘at’ nccgroup ‘dot’ com