Revealing Embedded Fingerprints: Deriving intelligence from USB stack interactions

Andy Davis, Research Director NCC Group
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
< 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
Windows USB driver stack

Legend:
- Client or Class Driver
- Microsoft-Provided Helper Libraries
- Microsoft-Provided Inbox drivers
- Hardware

User Mode

Kernel Mode

USB Client Driver Layer

Winusb.dll
Client Driver

USB Driver Stack Layer

USB 3.0 Driver Stack
- Usbhbc.sys
- Ucm0100.sys
- Usbxhci.sys

USB 2.0 Driver Stack
- Usbhub.sys
- Usbport.sys
- Usbhci.sys
- Usbuhci.sys

Data Flow

URB, IOCTL

Image from: msdn.microsoft.com
Linux USB stack

Image from: www.linux-usb.org
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?

Device Descriptor

<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>

Interface 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>bInterfaceProtocol</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>bInterface</td>
<td>0</td>
<td>Index of string descriptor describing this Interface</td>
</tr>
</tbody>
</table>
Installed drivers and supported devices

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

<table>
<thead>
<tr>
<th>idVendor</th>
<th>0x090C</th>
<th>Silicon Motion, Inc. - Taiwan</th>
</tr>
</thead>
<tbody>
<tr>
<td>idProduct</td>
<td>0x1000</td>
<td>Memory Bar</td>
</tr>
</tbody>
</table>

• 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

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)

All communication stops after “Set Configuration”
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…)
Timing information (work in progress…)
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>1000 0000B</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:

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (81% complete)

If you'd like to know more, you can search online later for this error: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (HIDPARSE SYSI)
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

```assembly
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

```assembly
; CODE XREF: SelectConfiguration(x)*GB
push    0xFFFFFFFFh
push    0xFFFFFFFFh
push    0
push    ecx
push    edi
push    edi
call    ds:_imp_USB_ParseConfigurationDescriptorEx828
test    eax, eax
jz      short loc_11770
mov     al, [eax+5]
mov     [esi+4], al
jmp      short loc_11774

; CODE XREF: SelectConfiguration(x)*AE
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_11770:

; CODE XREF: SelectConfiguration(x)*B6
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 get_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 = USBD_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 = USBD_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