Revealing Embedded Fingerprints: Deriving intelligence from USB stack interactions

Andy Davis, Research Director NCC Group

Image from: p1fran.com

nccgroup
freedom from doubt

UK Offices

Manchester - Head Office

Cheltenham

Edinburgh

Leatherhead

London

Thame

European Offices

Amsterdam - Netherlands

Munich – Germany

Zurich - Switzerland

North American Offices

San Francisco

Atlanta

New York

Seattle

Australian Offices

Sydney

Agenda

Part One:

- Overview of the USB enumeration phase
- Different USB stack implementations
- USB testing platform
- Installed drivers and supported devices
- Fingerprinting USB stacks and OS versions

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?
- May provide information useful for other attacks

USB Background stuff

Image from: blog.brickhousesecurity.com

Overview of the USB enumeration phase

- What is enumeration for?
  - Assign an address
  - Speed of communication
  - Power requirements
  - Configuration options
  - Device descriptions
  - Class drivers
- Lots of information exchange – implemented in many different ways

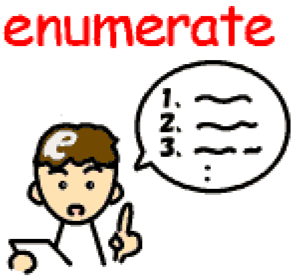


Image from :http://ewalk2.blog117.fc2.com

The USB enumeration phase

LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
⇦	Get Device Descriptor	0x00	0x0	12 01 10 01 00 00 00 08...	OK

LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
⇨	Set Address (0x01)	0x00	0x0		OK

LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
⇦	Get Device Descriptor	0x01	0x0	12 01 10 01 00 00 00 08...	OK

LS	Control Transfer	Addr	Endp	Data (34 bytes)	Status
⇦	Get Configuration Descriptor	0x01	0x0	09 02 22 00 01 01 00 A0...	OK

LS	Control Transfer	Addr	Endp	Data (4 bytes)	Status
⇦	Get String Descriptor 0	0x01	0x0	04 03 09 04	OK

LS	Control Transfer	Addr	Endp	Data (48 bytes)	Status
⇦	Get String Descriptor 2	0x01	0x0	30 03 44 00 65 00 6C 00...	OK

LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
⇦	Get Device Descriptor	0x01	0x0	12 01 10 01 00 00 00 08...	OK

LS	Control Transfer	Addr	Endp	Data (9 bytes)	Status
⇦	Get Configuration Descriptor	0x01	0x0	09 02 22 00 01 01 00 A0...	OK

LS	Control Transfer	Addr	Endp	Data (34 bytes)	Status
⇦	Get Configuration Descriptor	0x01	0x0	09 02 22 00 01 01 00 A0...	OK

LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
⇨	Set Configuration (0x01)	0x01	0x0		OK

Enumeration phase peculiarities

- Why is the device descriptor initially requested twice?
- Why are there multiple requests for other descriptors?
- Class-specific descriptors:

LS	Control Transfer	Addr	Endp	Data (64 bytes)	Status
←	Get HID Report Descriptor	0x01	0x0	05 01 09 06 A1 01 05 07...	OK

LS	Control Transfer	Addr	Endp	Data (1 byte)	Status
→	Set Report (HID)	0x01	0x0	00	OK

Different USB stack implementations

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

Image from: blogs.msdn.com

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

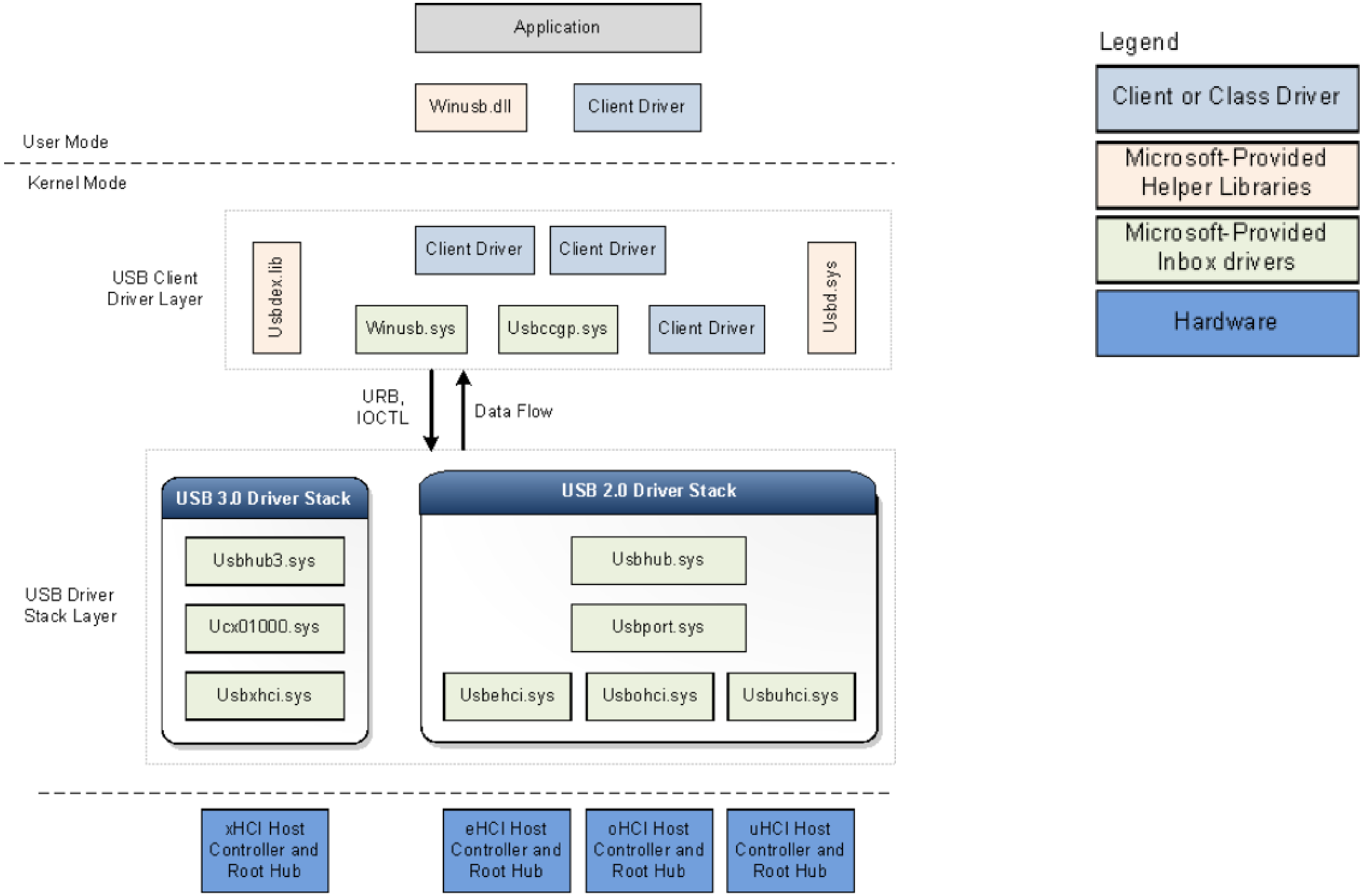


Image from: msdn.microsoft.com

Linux USB stack

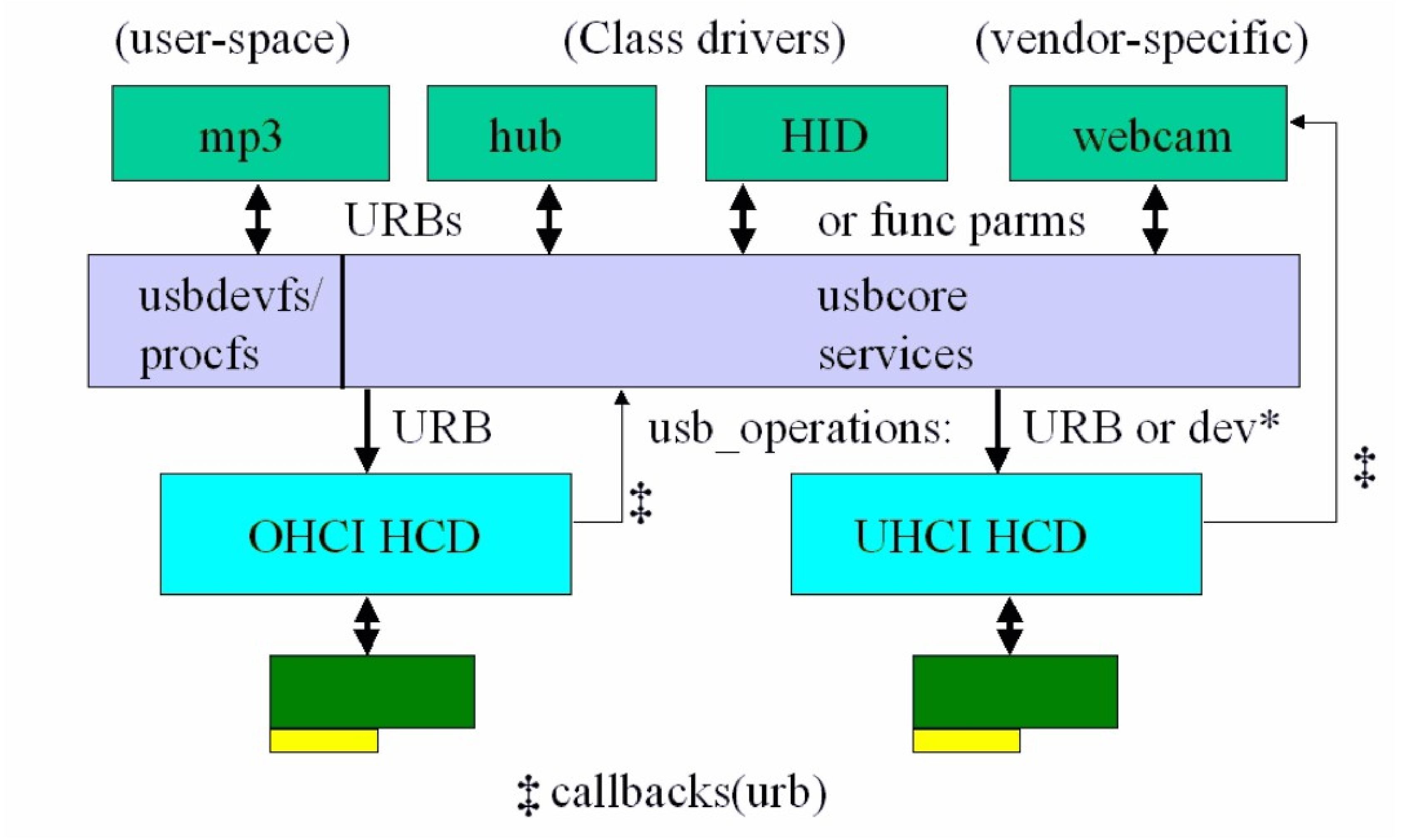


Image from: www.linux-usb.org

Embedded Access USB stack

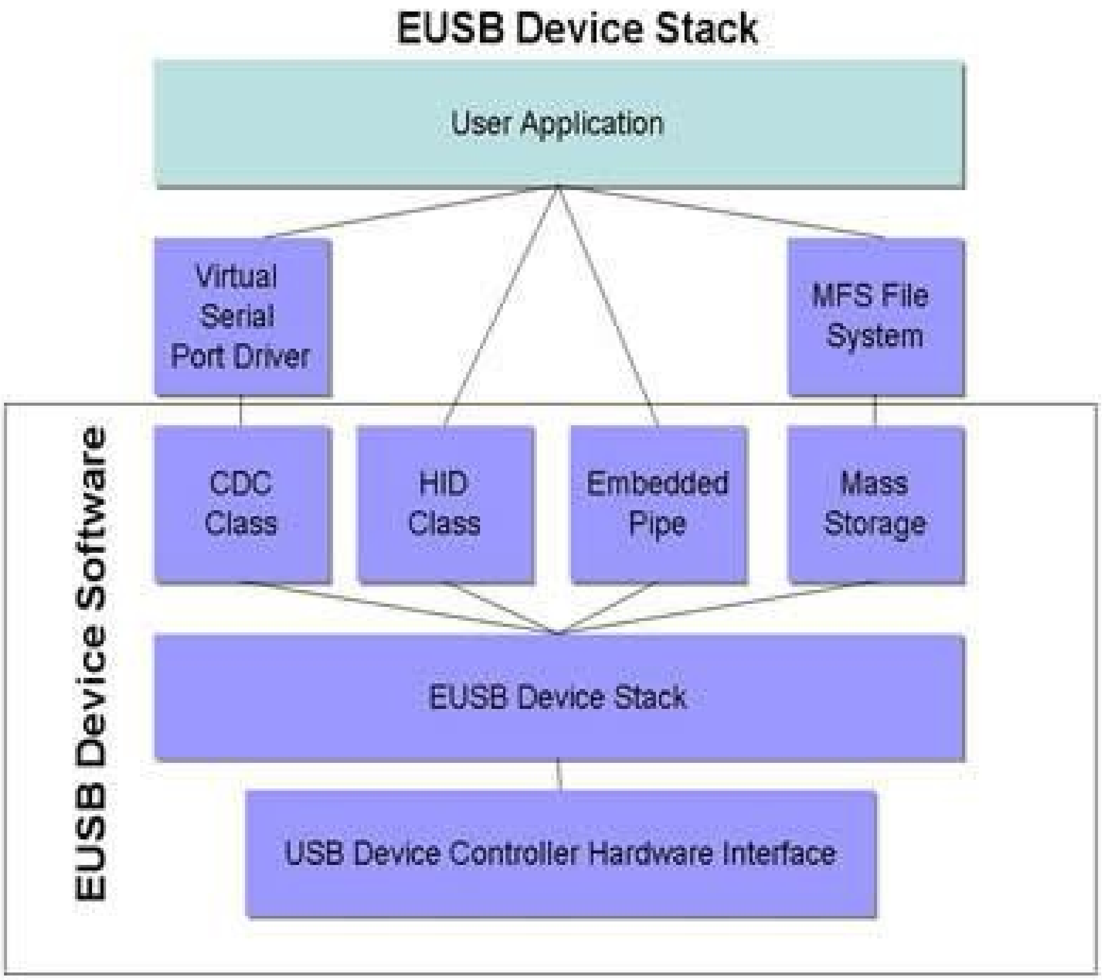


Image from: www.embedded-access.com

Interacting with USB

Image from: www.nvish.com

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

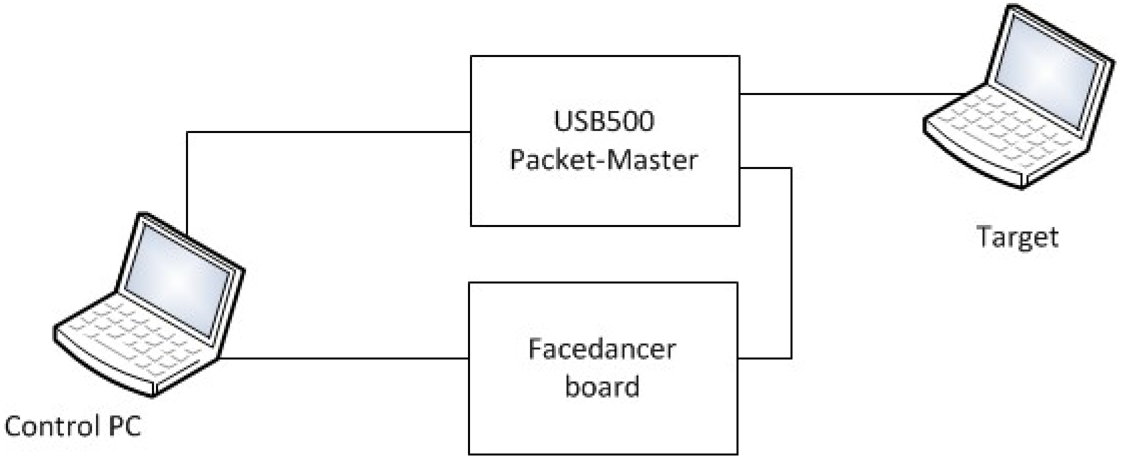
USB testing – gold-plated solution

- 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

Image from: network.nature.com

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?

Field	Value	Meaning
bLength	18	Valid Length
bDescriptorType	1	DEVICE
bcdUSB	0x0110	Spec Version
bDeviceClass	0x00	Class Information in Interface Descriptor
bDeviceSubClass	0x00	Class Information in Interface Descriptor

Device Descriptor

Field	Value	Meaning
bLength	9	Valid length
bDescriptorType	4	INTERFACE
bInterfaceNumber	0	Zero-based Number of this Interface.
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumEndpoints	1	Number of endpoints used by this interface (excluding endpoint zero).
bInterfaceClass	0x03	HID
bInterfaceSubClass	0x01	Boot Interface

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)
- Valid PIDs and VIDs are available (http://www.linux-usb.org/usb.ids)

Enumerating installed drivers

nccgroup
freedom from doubt

Not installed:

HS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
←	Get Device Descriptor	0x00	0x0	12 01 00 02 00 00 00 40...	OK

HS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
→	Set Address (0x1f)	0x00	0x0		OK

HS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
←	Get Device Descriptor	0x1F	0x0	12 01 00 02 00 00 00 40...	OK

HS	Control Transfer	Addr	Endp	Data (9 bytes)	Status
←	Get Configuration Descriptor	0x1F	0x0	09 02 2E 00 01 01 00 80...	OK

HS	Control Transfer	Addr	Endp	Data (46 bytes)	Status
←	Get Configuration Descriptor	0x1F	0x0	09 02 2E 00 01 01 00 80...	OK

HS	Control Transfer	Addr	Endp	Data (4 bytes)	Status
←	Get String Descriptor 0	0x1F	0x0	04 03 09 04	OK

HS	Control Transfer	Addr	Endp	Data (42 bytes)	Status
←	Get String Descriptor 2	0x1F	0x0	2A 03 38 00 30 00 32 00...	OK

HS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
←	Get String Descriptor 1	0x1F	0x0	12 03 52 00 65 00 61 00...	OK

HS	Control Transfer	Addr	Endp	Data (26 bytes)	Status
←	Get String Descriptor 3	0x1F	0x0	1A 03 30 00 30 00 65 00...	OK

HS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
→	Set Configuration (0x01)	0x1F	0x0		OK

All communication stops after “Set Configuration”

Installed:

LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
←	Get Device Descriptor	0x00	0x0	12 01 10 01 00 00 00 08...	OK

LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
→	Set Address (0x02)	0x00	0x0		OK

LS	Control Transfer	Addr	Endp	Data (18 bytes)	Status
←	Get Device Descriptor	0x02	0x0	12 01 10 01 00 00 00 08...	OK

LS	Control Transfer	Addr	Endp	Data (9 bytes)	Status
←	Get Configuration Descriptor	0x02	0x0	09 02 22 00 01 01 00 A0...	OK

LS	Control Transfer	Addr	Endp	Data (34 bytes)	Status
←	Get Configuration Descriptor	0x02	0x0	09 02 22 00 01 01 00 A0...	OK

LS	Control Transfer	Addr	Endp	Data (4 bytes)	Status
←	Get String Descriptor 0	0x02	0x0	04 03 09 04	OK

LS	Control Transfer	Addr	Endp	Data (48 bytes)	Status
←	Get String Descriptor 2	0x02	0x0	30 03 44 00 65 00 6C 00...	OK

LS	Control Transfer	Addr	Endp	Data (10 bytes)	Status
←	Get String Descriptor 1	0x02	0x0	0A 03 44 00 65 00 6C 00...	OK

LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
→	Set Configuration (0x01)	0x02	0x0		OK

LS	Control Transfer	Addr	Endp	Data (0 bytes)	Status
→	Set Idle (HID) Indefinite, All	0x02	0x0		OK

LS	Control Transfer	Addr	Endp	Data (65 bytes)	Status
←	Get HID Report Descriptor	0x02	0x0	05 01 09 06 A1 01 05 07...	OK

LS	Control Transfer	Addr	Endp	Data (1 byte)	Status
→	Set Report (HID)	0x02	0x0	00	OK

Sniffing the bus - Other connected devices

- Data from other devices will be displayed on other addresses

FS	Bulk Transfer (SPLIT)	Addr	Endp	Data (0 bytes)	Status
←	Class Transfer	0x08	0x2		OK

HS	Control Transfer	Addr	Endp	Data (4 bytes)	Status
→	Vendor Request	0x04	0x0	7F 54 12 4C	OK

FS	Bulk Transfer (SPLIT)	Addr	Endp	Data (0 bytes)	Status
←	Class Transfer	0x08	0x2		OK

FS	Bulk Transfer (SPLIT)	Addr	Endp	Data (0 bytes)	Status
←	Class Transfer	0x08	0x2		OK

Fingerprinting USB stacks and OS versions

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

Matching req. patterns to known stacks

nccgroup
freedom from doubt

Linux-based TV STB

HS	Transfer	Description	Addr	Endp	Data	Status
←	Control Transfer	Get Max LUN (Mass Storage)	0x21	0x0	(1 byte) 00	OK
→	Bulk Transfer	CBW: INQUIRY	0x21	0x2	(31 bytes) 55 53 42 43 01 00 00 00...	OK
←	Bulk Transfer	MSC Data In	0x21	0x1	(36 bytes) 00 80 04 02 1F 73 6D 69...	OK
←	Bulk Transfer	CSW - Status:Passed	0x21	0x1	(13 bytes) 55 53 42 53 01 00 00 00...	OK
→	Bulk Transfer	CBW: TEST UNIT READY	0x21	0x2	(31 bytes) 55 53 42 43 02 00 00 00...	OK
←	Bulk Transfer	CSW - Status:Passed	0x21	0x1	(13 bytes) 55 53 42 53 02 00 00 00...	OK
→	Bulk Transfer	CBW: READ CAPACITY(10)	0x21	0x2	(31 bytes) 55 53 42 43 03 00 00 00...	OK
←	Bulk Transfer	MSC Data In	0x21	0x1	(8 bytes) 00 77 7F FF 00 00 02 00	OK
←	Bulk Transfer	CSW - Status:Passed	0x21	0x1	(13 bytes) 55 53 42 53 03 00 00 00...	OK
→	Bulk Transfer	CBW: MODE SENSE(6)	0x21	0x2	(31 bytes) 55 53 42 43 04 00 00 00...	OK

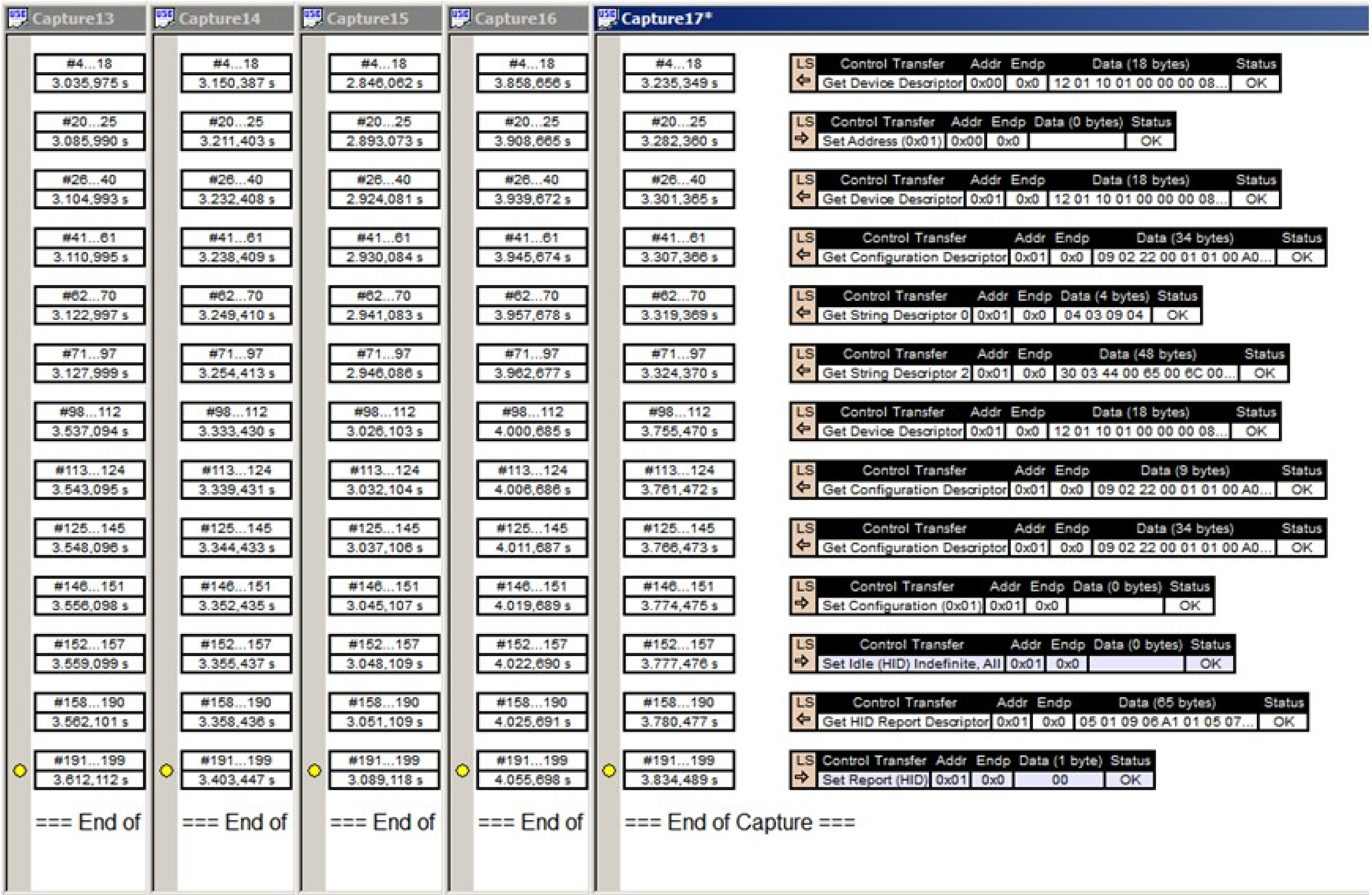
Windows 8

HS	Transfer	Description	Addr	Endp	Data	Status
←	Control Transfer	Get Max LUN (Mass Storage)	0x02	0x0	(1 byte) 00	OK
→	Bulk Transfer	CBW: INQUIRY	0x02	0x2	(31 bytes) 55 53 42 43 28 1E A6 83...	OK
←	Bulk Transfer	MSC Data In	0x02	0x1	(36 bytes) 00 80 04 02 1F 73 6D 69...	OK
←	Bulk Transfer	CSW - Status:Passed	0x02	0x1	(13 bytes) 55 53 42 53 28 1E A6 83...	OK
→	Bulk Transfer	CBW: INQUIRY	0x02	0x2	(31 bytes) 55 53 42 43 80 3C AB 83...	OK
←	Bulk Transfer	MSC Data In	0x02	0x1	(36 bytes) 00 80 04 02 1F 73 6D 69...	OK
←	Bulk Transfer	CSW - Status:Passed	0x02	0x1	(13 bytes) 55 53 42 53 80 3C AB 83...	OK
→	Bulk Transfer	CBW: READ FORMAT CAPACITIES (or VS)	0x02	0x2	(31 bytes) 55 53 42 43 28 8E B2 83...	OK
←	Bulk Transfer	MSC Data In	0x02	0x1	(12 bytes) 00 00 00 08 00 77 80 00...	OK
←	Bulk Transfer	CSW - Status:Passed	0x02	0x1	(13 bytes) 55 53 42 53 28 8E B2 83...	OK

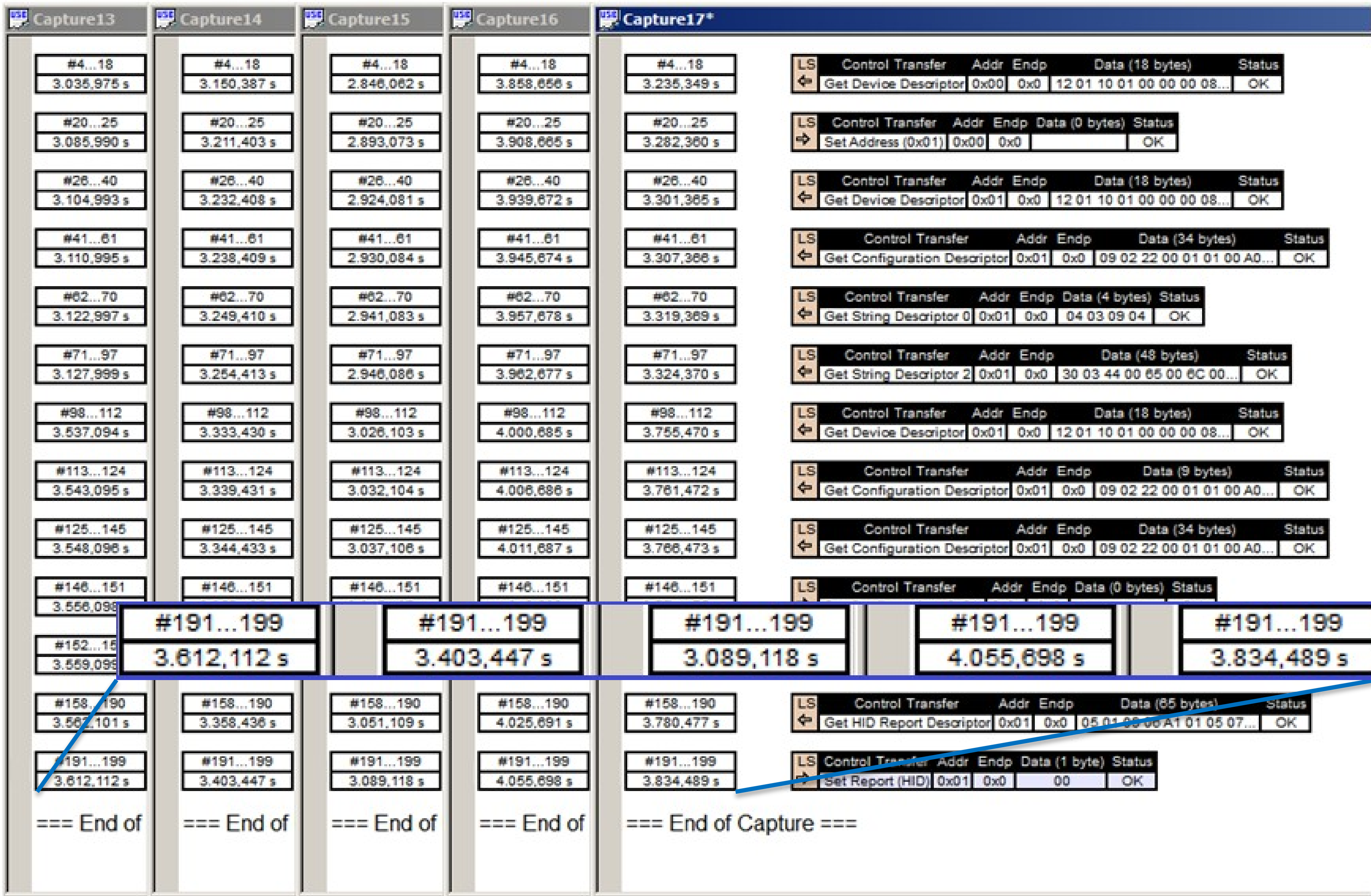
Request patterns unique elements?

- Windows 8 (HID) – Three 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



Timing information



Using timing for fingerprinting?

- 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 we can 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:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0000B	GET_DESCRIPTOR	0x03EE	0x0000	0x12	Returned String

- If a device does not contain a valid string descriptor at index 0xEE, it must respond with a stall packet. If the device does not respond with a stall packet, the system will issue a single-ended zero reset packet to the device, to help it recover from its stalled state (Windows XP only).

Responses to invalid data

- Different USB stacks respond to invalid data in different ways
- Maximum and minimum values
- Logically incorrect values
- Missing data

Image from: windows7.iyogi.com

Invalid data unique elements?

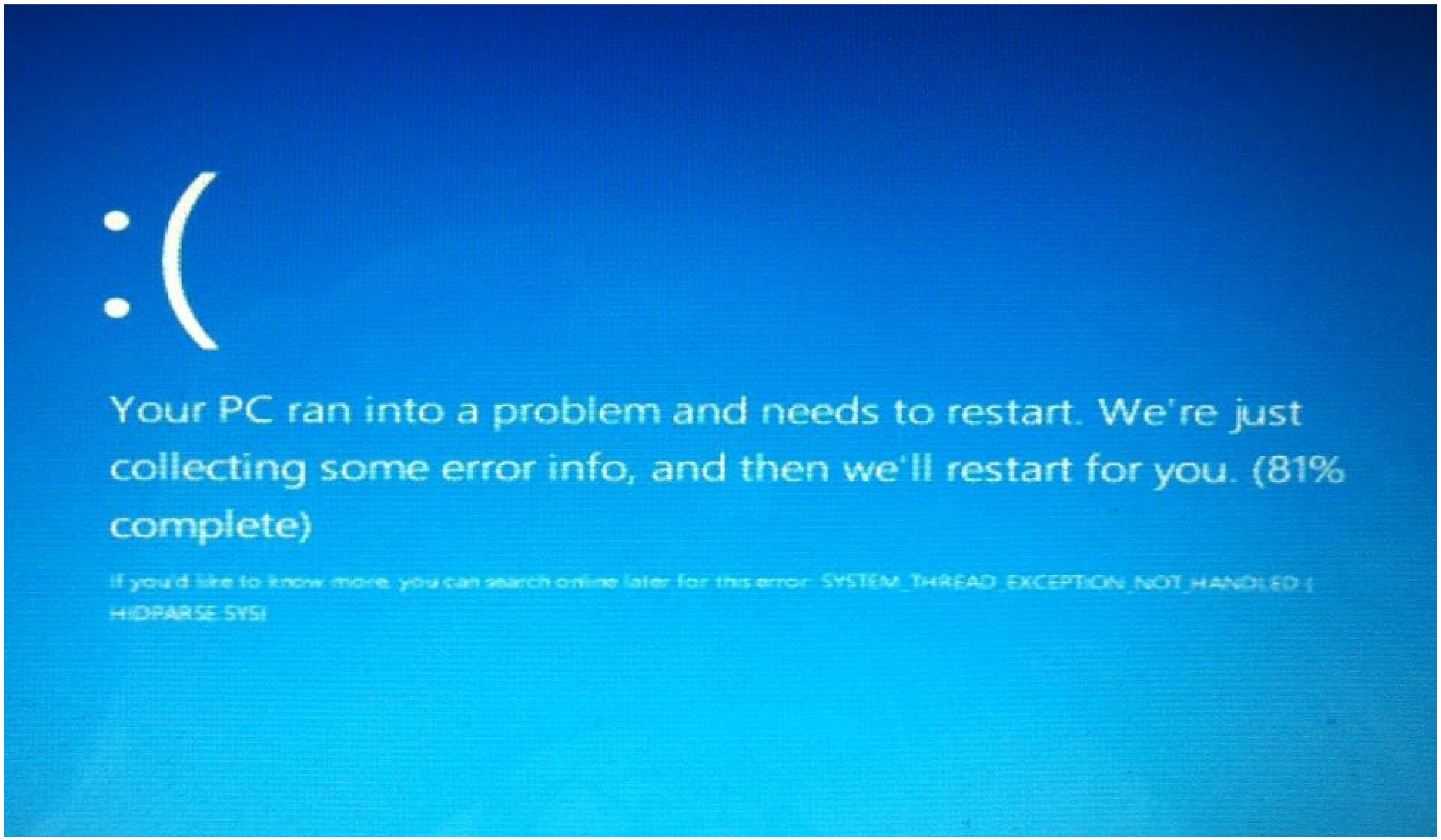
Windows 8 (all versions)

If you send a specific, logically incorrect HID Report descriptor this happens:

Invalid data unique elements?

Windows 8 (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 Device descriptors are re-requested after enumeration is complete

Part Two: Potentially exploitable USB bugs

Image from: www.biro-media.hr

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 a USB device that uses this driver is inserted into a Windows host, during the enumeration phase the USB Configuration descriptor is requested and parsed
- 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 field is “bNumInterfaces” in Table A2: USB Configuration Descriptor (http://msdn.microsoft.com/en-us/windows/hardware/gg463298)

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)               mov     edi, edi
SelectConfiguration(x)+2             push    ebp
SelectConfiguration(x)+3             mov     ebp, esp
SelectConfiguration(x)+5             sub     esp, 10h
SelectConfiguration(x)+8             push    ebx
SelectConfiguration(x)+9             push    esi
SelectConfiguration(x)+A             mov     esi, [ebp+ptr_Pool_U802]
SelectConfiguration(x)+D             push    edi
SelectConfiguration(x)+E             mov     edi, [esi+1Ch]  ; points to start of configuration descriptor
SelectConfiguration(x)+11            mov     al, [edi+4]     ; al = bNumInterfaces
SelectConfiguration(x)+14            cmp     al, 2           ; compares with 2 (what it should be)
SelectConfiguration(x)+16            jb      loc_11877       ; no jump
SelectConfiguration(x)+1C            movzx   eax, al
SelectConfiguration(x)+1F            lea     eax, ds:8[eax*8] ; multiply bNuminterfaces by 8 then add 8 = 24
SelectConfiguration(x)+26            push    eax
SelectConfiguration(x)+27            call    _AllocPool@4    ; AllocPool(x)
SelectConfiguration(x)+2C            mov     [ebp+ptr_Pool_U802_24_bytes], eax
SelectConfiguration(x)+2F            test    eax, eax
SelectConfiguration(x)+31            jz      loc_11877       ; no jump (AllocPool was successful)
SelectConfiguration(x)+37            xor     ebx, ebx
SelectConfiguration(x)+39            cmp     [edi+4], bl     ; compares bNumInterfaces with 0
SelectConfiguration(x)+3C            jbe     short loc_1171F ; no jump
SelectConfiguration(x)+3E            mov     esi, eax
```

The crash point

```
SelectConfiguration(x)+9B   yet_more_interfaces_to_parse:           ; CODE XREF: SelectConfiguration(x)+CE↓j
SelectConfiguration(x)+9B                   push    0FFFFFFFFh
SelectConfiguration(x)+9D                   push    0FFFFFFFFh
SelectConfiguration(x)+9F                   push    0FFFFFFFFh
SelectConfiguration(x)+A1                   push    0
SelectConfiguration(x)+A3                   push    ecx
SelectConfiguration(x)+A4                   push    edi
SelectConfiguration(x)+A5                   push    edi
SelectConfiguration(x)+A6                   call    ds:__imp__USBD_ParseConfigurationDescriptorEx@28
SelectConfiguration(x)+AC                   test    eax, eax
SelectConfiguration(x)+AE                   jz      short loc_11770
SelectConfiguration(x)+B0                   mov     al, [eax+5]
SelectConfiguration(x)+B3                   mov     [esi+4], al
SelectConfiguration(x)+B6                   jmp     short loc_11774
SelectConfiguration(x)+B8   ; ---------------------------------------------------------------------------
SelectConfiguration(x)+B8
SelectConfiguration(x)+B8   loc_11770:                              ; CODE XREF: SelectConfiguration(x)+AE↑j
SelectConfiguration(x)+B8                   mov     byte ptr [esi+4], 0 ; writes one null byte over the first byte of the next pool header
SelectConfiguration(x)+B8                                           ; this is where the corruption occurs
SelectConfiguration(x)+BC
SelectConfiguration(x)+BC   loc_11774:                              ; CODE XREF: SelectConfiguration(x)+B6↑j
SelectConfiguration(x)+BC                   movzx   eax, word ptr [esi]
SelectConfiguration(x)+BF                   mov     ecx, [ebp+ptr_Pool_U802_24_bytes]
SelectConfiguration(x)+C2                   add     esi, eax
SelectConfiguration(x)+C4                   movzx   eax, byte ptr [edi+4]
SelectConfiguration(x)+C8                   inc     ecx
SelectConfiguration(x)+C9                   mov     [ebp+ptr_Pool_U802_24_bytes], ecx
SelectConfiguration(x)+CC                   cmp     ecx, eax
SelectConfiguration(x)+CE                   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 I 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

```
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;
      }
  }
```

nccgroup
freedom from doubt

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
- What about USB over RDP?

Image from: leadershipfreak.wordpress.com

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