nccgroup
freedom from doubt

An NCC Group Publication

Revealing Embedded Fingerprints: Deriving Intelligence from USB Stack Interactions

Prepared by:
Andy Davis
Research Director
andy.davis 'at' nccgroup.com

© Copyright 2013 NCC Group

Contents

1 List of Figures and Tables 3
2 Introduction 4
2.1 Previous Research 4
3 USB Background: The Enumeration Phase in Detail 4
4 USB Testing Platform 8
5 USB Stack Implementations 9
6 Identifying Supported Devices 10
6.1 USB Device Classes 10
6.2 Enumerating Installed Class Drivers 12
6.3 Other Devices Already Connected 13
7 Fingerprinting Techniques 14
7.1 Operating System Identification 14
7.2 Application Identification 14
7.3 Timing Information 15
7.4 Descriptor Types Requested 16
7.5 Responses to Invalid Data 17
8 Umap 17
9 Conclusions 23
10 References and Further Reading 24
11 Glossary 24

 © Copyright 2013 NCC Group

1 List of Figures and Tables

Figure 1: The use of a Facedancer board in conjunction with a Packet-master USB analyser
Figure 2: Enumeration stops at "Set Configuration" when a device class is not supported
Figure 3: Enumeration continues past the "Set Configuration" when a device class is supported
Figure 4: A Packet-master capture showing multiple USB devices connected to the same bus
Figure 5: Linux-based TV Set-Top-Box
Figure 6: Windows 8
Figure 7: gphoto2 (Linux)
Figure 8: "Photos" Metro app (Windows 8)
Figure 9: USB timing information during enumeration
Figure 10: Umap basic help
Figure 11: The USB device classes that umap currently understands
Figure 12: Umap identifying supported classes, sub-classes and protocols
Figure 13: The umap VID/PID lookup facility
Figure 14: The umap operating system identification capability
Figure 15: Umap emulating a USB camera
Figure 16: Generic USB fuzz test cases
Figure 17: Class-specific USB fuzz test cases
Figure 18: Umap fuzzing a USB host

Table 1: Get Device descriptor request
Table 2: Device descriptor
Table 3: Configuration descriptor
Table 4: Interface descriptor
Table 5: Endpoint descriptor
Table 6: HID descriptor
Table 7: String descriptor
Table 8: Set Configuration request
Table 9: USB Device classes
Table 10: Hub class information in a Device descriptor
Table 11: Image class information in an Interface descriptor
Table 12: VID and PID information in a Device descriptor
Table 13: Microsoft OS descriptor request

 © Copyright 2013 NCC Group

2 Introduction

Embedded systems are everywhere, from TVs to aircraft, printers to weapons control systems. As a security researcher when you are faced with one of these black boxes to test, sometimes *in situ*, it is difficult to know where to start. However, if there is a USB port on the device. there is useful information that can be gained. In this paper we will show how USB stack interaction analysis can be used to provide information such as the OS running on the embedded device, the USB drivers installed, and the devices supported. When testing the security of a USB host stack, knowledge of the installed drivers will increase the efficiency of the testing process dramatically.

2.1 Previous Research

There has been plenty of previous research into the security of USB in recent years, which has mainly focussed on different approaches to enable USB hosts to be tested for vulnerabilities [Davis][Dominguez Vega][Larimer]. However, the author is only aware of one reference to research involving the use of USB interactions to identify information about the host stack [Goodspeed].

3 USB Background: The Enumeration Phase in Detail

USB is a master-slave protocol, with the host as the master and devices as slaves. Only the master can make requests to slaves and not the other way round, which poses a problem as we are trying to identify information about the configuration of the host from the perspective of a slave (device). Therefore we need to observe the way the host requests information in great detail, and also to provide potentially unexpected answers to the host's requests, generating unique behaviour in the host, which can then also be observed.

The initial communication any USB device has with a host is during enumeration. Enumeration is the mechanism by which a USB host determines the status, configuration, and capabilities of an inserted USB device. The process begins when a device is mechanically inserted into the host and follows a number of steps:

There are four lines on a USB connector: Vcc (+5V), GND (0V), positive data (D+) and negative data (D-). Prior to a device being connected, D+ and D- are connected to GND via a 15K resistor. At the point of insertion, different resistors and differential signals are used to determine the speed of the connected device:

- A low speed device (1.5Mbps) connects D- to Vcc via a 1K5 pull-up resistor
- A full speed device (12Mbps) connects D+ to Vcc via a 1K5 pull-up resistor
- A high speed device (480Mbps) connects D+ to Vcc via a 1K5 pull-up resistor (and hence initially appears to be a full speed device). The host then attempts to communicate at 480Mbps with the device using J and K chirps (a J chirp is a differential signal on D+ and D- >= +300mV, whereas a K chirp is >= -300mV). If the communication fails the host assumes the device is a full speed device rather than a high speed device.

Now that the host knows what speed it can use to communicate with the device, it can start interrogating it for information. An 8-byte SETUP packet called the setup transaction (Table 1) is sent by the host in the first phase of a control transfer. It contains the request "GET_DESCRIPTOR" (for the device descriptor) and is sent using address 0.

The device then responds with an 18-byte device descriptor, also on address 0 (Table 2).

 © Copyright 2013 NCC Group

Field	Value	Meaning
bmRequestType (direction)	1	Device-to-host
bmRequestType (type)	0	Standard
bmRequestType (recipient)	0	Device
bRequest	0x06	Get Descriptor
wValue	0x0100	DEVICE Index = 0
wIndex	0x0000	Zero
wLength	0x0040	Length requested = 64

Table 1: Get Device descriptor request

Field	Value	Meaning
bLength	18	Descriptor length (including the bLength field)
bDescriptorType	1	Device descriptor
bcdUSB	0x0110	Spec version
bDeviceClass	0x00	Class information stored in Interface descriptor
bDeviceSubClass	0x00	Class information stored in Interface descriptor
bDeviceProtocol	0x00	Class information stored in Interface descriptor
bMaxPacketSize0	8	Max EP0 packet size
idVendor	0x413c	Dell Inc
idProduct	0x2107	Unknown
bcdDevice	0x0178	Device release number
iManufacturer	1	Index to Manufacturer string
iProduct	2	Index to Product string
iSerialNumber	0	Index to serial number
bNumConfigurations	1	Number of possible configurations

Table 2: Device descriptor

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	2	Configuration descriptor
wTotalLength	34	Total combined size of this set of descriptors
bNumInterfaces	1	Number of interfaces supported by this configuration
bConfigurationValue	1	Value to use as an argument to the SetConfiguration() request to select this configuration
iConfiguration	0	Index of String descriptor describing this configuration
bmAttributes (Self-powered)	0	Bus-powered
bmAttributes (Remote wakeup)	1	Yes
bmAttributes (Other bits)	0x80	Valid
bMaxPower	100mA	Maximum current drawn by device in this configuration

Table 3: Configuration descriptor

© Copyright 2013 NCC Group

The most important data in the device descriptor is:

- Device class information (if present)
- Maximum packet size in bytes of Endpoint 0
- Vendor and Product IDs (VID and PID)
- Number of configurations

The host resets the device, allocates an address to it (in the range of 1 to 127) and then re-requests the device descriptor using the new address.

For each possible configuration, the host will request a configuration descriptor, an example of which is shown in Table 3. The configuration descriptor includes a number of further descriptors (interface and endpoint, examples of which are shown in tables 4 and 5 respectively); however, the primary fields of interest are:

- Number of interfaces supported by this configuration
- The power attributes that indicate if the device is self- or bus-powered and the maximum current the device will draw.

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	4	Interface descriptor
bInterfaceNumber	0	Number of this interface
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumberEndpoints	1	Number of endpoints used by this interface
bDeviceClass	0x03	HID
bDeviceSubClass	0x01	Boot interface
bDeviceProtocol	0x01	Keyboard
iInterface	0	Index of string descriptor describing this interface

Table 4: Interface descriptor

Field	Value	Meaning
bLength	7	Descriptor length (including the bLength field)
bDescriptorType	5	Endpoint descriptor
bEndpointAddress	0x81	Endpoint 1 – OUT
bmAttributes	0x03	Interrupt data endpoint
wMaxPacketSize	0x0008	Maximum packet size is 8
bInterval	0x0a	10 frames (10ms)

Table 5: Endpoint descriptor

 © Copyright 2013 NCC Group

Within the interface descriptor, the important information is:

- Number of endpoints
- Class information (interface-specific information not provided in the device descriptor)

An endpoint descriptor contains:

- The endpoint address and type
- The maximum packet size in bytes of the endpoint

Sometimes class-specific descriptors are included within the configuration, for example the HID descriptor in Table 6:

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	0x21	HID
bcdHID	0x0110	HID Class Spec Version
bCountryCode	0	Not Supported
bNumDescriptors	1	Number of Descriptors
bDescriptorType	34	Report descriptor
wDescriptorLength	65	Descriptor length

Table 6: HID descriptor

If there are multiple configurations for a device then further configuration (as well as interface, endpoint, etc.) descriptors will be requested.

The next descriptors requested are string descriptors, which provide human-readable information about the device type and vendor. An example is shown in Table 7.

Field	Value	Meaning
bLength	48	Descriptor length (including the bLength field)
bDescriptorType	3	String descriptor
bString	"Dell USB Entry Keyboard"	

Table 7: String descriptor

The final step is for the host to select one of the device configurations and inform the device that it will be using that configuration. This is performed by issuing a "Set Configuration" request, as shown in Table 8.

© Copyright 2013 NCC Group

Field	Value	Meaning
bmRequestType (direction)	0	Host-to-device
bmRequestType (type)	0	Standard
bmRequestType (recipient)	0	Device
bRequest	0x09	Set Configuration
wValue	0x0001	Configuration No.
wIndex	0x0000	Zero
wLength	0x0000	Zero

Table 8: Set Configuration request

The enumeration phase is now complete, with the USB device configured and ready to use. From now until the device is removed, class-specific communication is used between the device and the host. However, as we will discuss later, there are variations to this enumeration phase which can be used to fingerprint different host implementations.

4 USB Testing Platform

Additional hardware is needed to interact with USB, so that different USB devices can be emulated. There are a number of requirements for this testing platform:

The ability to both capture and replay USB traffic: There are many USB analyser tools available, but only a few that allow captured traffic to be replayed; an ability that is crucial in this instance.

Full control of generated traffic: Many test-equipment-based solutions restrict the user to generating traffic that conforms to the USB specification. We need full control of all aspects of any generated traffic, as the host many behave in an unexpected way if it receives unconventional data, which is what we are hoping to observe.

Class decoders are extremely useful: For each USB device class (e.g. mass storage, printer), there are separate specification documents that detail the class-specific communications protocols. Having an application that understands and decodes these protocols makes understanding the class communication significantly easier.

Support for multiple speeds: USB devices, depending on their function, operate at a number of different speeds; therefore the ability to capture and generate data at these different speeds is crucial if the whole range of devices is to be emulated.

The solution chosen for this project comprised two primary components: A commercial USB analyser and generator – Packet-Master [MQP], and a bespoke device emulation board called Facedancer [GoodFET]. Figure 1 shows how they are used together.

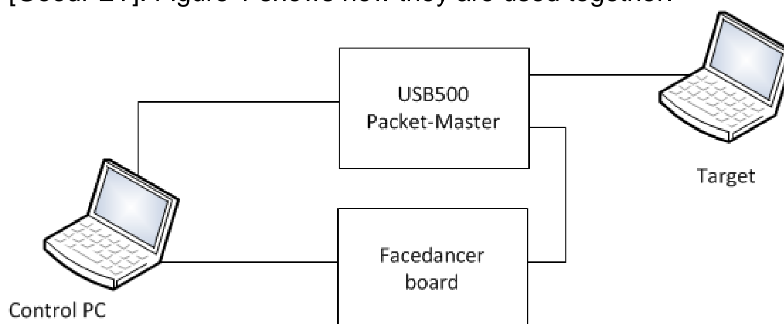


Figure 1: The use of a Facedancer board in conjunction with a Packet-master USB analyser

© Copyright 2013 NCC Group

The benefit of using both devices is that fully arbitrary USB traffic can be generated by Facedancer, acting as a USB device, and the responses from the host under test can be captured by the Packet-Master appliance. However, for the majority of the techniques described in this paper, just a Facedancer board would suffice.

5 USB Stack Implementations

USB is quite a complex protocol, especially as it provides backward compatibility to support older, slower devices. Therefore, implementations of the host stack on different operating systems can behave in different ways, as we hoped to observe during this research. Typical components within a USB host stack are as follows:

Host controller hardware: This performs the low-level timing and electrical aspects of the protocol and is communicated with via a host controller interface.

Host controller interface (HCI): There are a number of different HCIs that have been developed over the years, all of which have different capabilities, but the primary difference is their ability to support devices running at different speeds; they are:

- oHCI (Open Host Controller Interface)
- eHCI (Enhanced Host Controller Interface)
- uHCI (Universal Host Controller Interface)
- xHCI (Extensible Host Controller Interface)

Host controller driver: This provides a hardware abstraction layer so that the host can communicate via the controller interface to the hardware.

USB core: The component that performs core functionality such as device enumeration

Class drivers: Once enumeration is complete and control has been passed to a USB class driver, communication specific to the connected device is processed by the class driver

Application software: When a USB device is inserted a host may start an application specific to the class of that device (e.g. an application that displays photos when a camera device is connected).

 © Copyright 2013 NCC Group

6 Identifying Supported Devices

For USB host vulnerability assessment via fuzzing it is important to establish what device classes are supported. This is because USB fuzzing is a relatively slow process – each test case requires the virtual device to be "inserted" and "removed" via software, resulting in enumeration being performed each time. The USB protocol is designed to expect a human, rather than a computer, to insert a device, and so timing delays result in each test case taking several seconds to complete. If functionality that is not supported by the target host is fuzzed then this can waste large amounts of testing time.

6.1 USB Device Classes

There are a number of high level USB device classes; these are shown in Table 9.

Base Class	Descriptor Usage	Description
0x00	Device	Use class information in the Interface Descriptors
0x01	Interface	Audio
0x02	Both	CDC (Communication and Device Control)
0x03	Interface	HID (Human Interface Device)
0x05	Interface	Physical
0x06	Interface	Image
0x07	Interface	Printer
0x08	Interface	Mass Storage
0x09	Device	Hub
0x0a	Interface	CDC-Data
0x0b	Interface	Smart Card
0x0d	Interface	Content Security
0x0e	Interface	Video
0x0f	Interface	Personal Healthcare
0x10	Interface	Audio/Video Devices
0xdc	Both	Diagnostic Device
0xe0	Interface	Wireless Controller
0xef	Both	Miscellaneous
0xfe	Interface	Application Specific

Table 9: USB Device classes

USB device class information can be stored in a number of different places within the descriptors provided during enumeration. The information is provided in three-byte entries:

- `bDeviceClass` – the high level device class (e.g. mass storage)
- `bDeviceSubClass` – specific information about this device (e.g. SCSI command set)
- `bDeviceProtocol` – the protocol used (e.g. bulk transport (BBB))

Taking the mass storage class as an example, the following are the available sub-classes:

- De facto use
- RPC
- MMC-5 (ATAPI)

© Copyright 2013 NCC Group

- QIC-157
- UFI
- SFF-8070i
- SCSI
- LSD FS
- IEE 1667
- Vendor specific

For each of these mass storage sub-classes there are also a number of possible protocols:

- CBI with command completion interrupt
- CBI without command completion interrupt
- BBB
- UAS
- Vendor specific

So, as you can see, the potential attack surface of a USB host is enormous; but it is important to establish which functionality is supported prior to any active fuzz testing.

Some devices, such as the hub in Table 10, store their class information in the device descriptor.

Field	Value	Meaning
bLength	18	Descriptor length (including the bLength field)
bDescriptorType	1	Device descriptor
bcdUSB	0x0200	Spec version
bDeviceClass	**0x09**	**Hub**
bDeviceSubClass	**0x00**	**Full Speed Hub**
bDeviceProtocol	**0x01**	**Default**
...		

Table 10: Hub class information in a Device descriptor

However, more commonly, the class information is interface specific and is therefore stored in the interface descriptor (within a configuration descriptor), as with the image class device in Table 11.

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	4	Interface descriptor
bInterfaceNumber	0	Number of this interface
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumberEndpoints	3	Number of endpoints used by this interface
bDeviceClass	**0x06**	**Image**
bDeviceSubClass	**0x01**	**Default**
bDeviceProtocol	**0x01**	**Default**
…		

Table 11: Image class information in an Interface descriptor

 © Copyright 2013 NCC Group

When emulating specific device types, whether the class information is provided to the host in the device descriptor or in an interface descriptor depends on the device.

6.2 Enumerating Installed Class Drivers

To identify which device classes are supported by a USB host, emulated (class-specific) virtual devices need to be presented to the host iterating through each device class, sub-class, and protocol whilst monitoring the enumeration process. If a device is not supported then the enumeration phase will stop at the “Set Configuration” command, as shown in Figure 2.

```
< 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
```

Figure 2: Enumeration stops at “Set Configuration” when a device class is not supported

However, if the device is supported then class-specific communication starts after the “Set Configuration” command, as can be seen in the example of a HID device in Figure 3 (the class-specific communication is highlighted by the green box).

```
< 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)
```

Figure 3: Enumeration continues past “Set Configuration” when a device class is supported

Device class drivers are also referenced by their vendor ID (VID) and product ID (PID). If a specific device driver has been installed for a USB device then the host can reference this driver by using a combination of the class information, the VID and the PID, which are located in the device descriptor, as shown in Table 12.

 © Copyright 2013 NCC Group

Field	Value	Meaning
bLength	18	Descriptor length (including the bLength field)
bDescriptorType	1	Device descriptor
bcdUSB	0x0110	Spec version
bDeviceClass	0x00	Class information stored in Interface descriptor
bDeviceSubClass	0x00	Class information stored in Interface descriptor
bDeviceProtocol	0x00	Class information stored in Interface descriptor
bMaxPacketSize0	8	Max EP0 packet size
idVendor	**0x04DA**	**Panasonic Corporation**
idProduct	**0x2372**	**Lumix DMC-FZ10 Camera**

Table 12: VID and PID information in a Device descriptor

New VID and PID values must be registered with the USB Implementers Forum [USBIF] and are maintained in a number of public repositories. This information can be used to perform a brute-force attack against the host to identify any specific drivers that have been installed; however, this can be a very slow process.

6.3 Other Devices Already Connected

When testing a host that may have other devices, such as an HSPA modem, connected internally to the USB bus, these can be detected by sniffing the USB bus and looking for devices that are communicating using different addresses than that of the attached device, as shown in Figure 4.

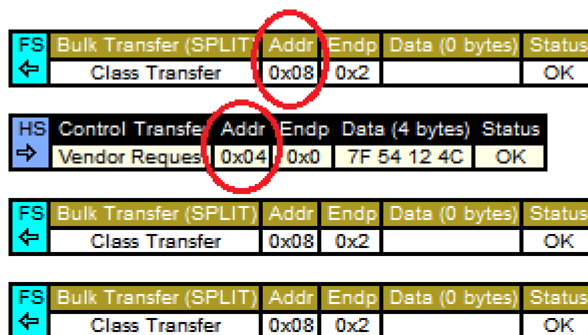


Figure 4: A Packet-master capture showing multiple USB devices connected to the same bus

One area of future research is to investigate if, using the Facedancer board to emulate the host to which it is connected, descriptor requests could be sent to these other devices to identify more information about them. Also, what happens if the Facedancer is configured to impersonate an already-connected device?

 © Copyright 2013 NCC Group

7 Fingerprinting Techniques

One of the targets of this research was to identify operating system and application information by observing USB stack interactions and sometimes using active techniques to prompt the host to perform different actions that may reveal useful information. This section will detail some of the techniques that were developed to do this.

7.1 Operating System Identification

Figures 5 and 6 show the start of class-specific communication once the enumeration phase has been completed for two different hosts. As you can clearly see, the class-specific commands used and the order in which the commands are issued are completely different for the two hosts and this technique can therefore be used to differentiate between different operating systems.

Note: The commands and the order of commands are the same each time a device is presented to the hosts

```
< 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
```

Figure 5: Linux-based TV Set-Top-Box

```
< 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
```

Figure 6: Windows 8

Other examples of unique behaviour of different operating systems:

- 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”

Further research in this area is expected to reveal techniques that will allow for more granular identification to be performed.

7.2 Application Identification

Applications that use USB devices to provide input (e.g. photo management applications) can also reveal useful information, as shown in Figures 7 and 8.

© Copyright 2013 NCC Group

```
> 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
```

Figure 7: gphoto2 (Linux)

```
> Image: OpenSession
< Image: OK
> Image: GetDeviceInfo
< Image: DeviceInfo
< Image: OK
> Image: SetDevicePropValue
> Image: DeviceProperty
< Image: OK
< Image: DeviceInfoChanged
```

Figure 8: “Photos” Metro app (Windows 8)

Figures 7 and 8 not only show that these two applications use different class-specific commands but the “Device Property” command sent by the host in Figure 8 contains the following data:

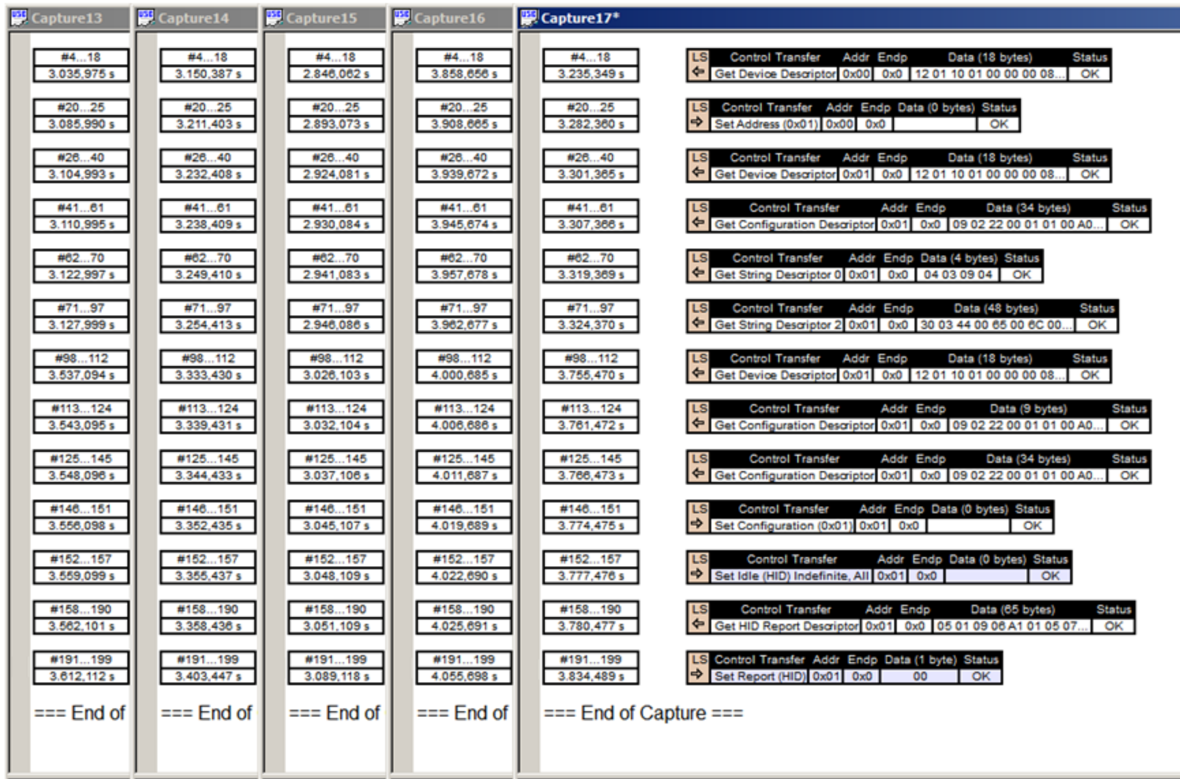
```
/Windows/6.2.9200 MTPClassDriver/6.2.9200.16384
```

This is specific information about the version of the operating system running on the host (Version 6.2 is the Microsoft internal representation for Windows 8 and 9200.16384 is the exact build revision).

7.3 Timing Information

The Packet-master analyser can differentiate between events occurring on the USB bus down to the microsecond. Figure 9 shows the capture information for five enumerations with the same device and same host.

 © Copyright 2013 NCC Group



#191...199	#191...199	#191...199	#191...199	#191...199
3.612,112 s	3.403,447 s	3.089,118 s	4.055,698 s	3.834,489 s

Figure 9: USB timing information during enumeration

Across the entire enumeration phase there is a large amount of variance between the times to enumerate the device. However, if the time is measured between specific requests e.g. between the requests for String descriptor 0 and String descriptor 2, something more interesting can be seen:

5002us, 5003us, 5003us, 4999us, 5001us

There is a maximum variance of 4 microseconds. Therefore, if the operating system is known can information be gleaned about the speed of the host? This hypothesis is still under investigation.

7.4 Descriptor Types Requested

Some operating systems have implemented their own USB descriptors —for example Microsoft has Microsoft OS descriptors (MODs). These were apparently developed for use with unusual device classes. Devices that support Microsoft OS descriptors must store a special string descriptor in firmware at the fixed string index of 0xee. The request is shown in Table 13.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	0x03ee	0x0000	0x12	Returned String

Table 13: Microsoft OS descriptor request

 © Copyright 2013 NCC Group

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 (this is for Windows XP only).

7.5 Responses to Invalid Data

Earlier in the paper we mentioned that the ability to send completely arbitrary USB packets to the host was required to determine how each host responds when a reply to one of its requests contains invalid data. Examples of invalid data include:

- Maximum and minimum values
- Logically incorrect values
- Missing data

During the research, various behaviours were observed as a result of sending this data. In some cases different "handled" error conditions occurred; however in many other situations unhandled errors were observed in the form of application errors, kernel panics and bug checks. The conclusions drawn from this area of the research were that invalid data was most useful in fuzzer test-cases for identifying bugs and potential security vulnerabilities.

8 Umap

A tool was developed to demonstrate many of the techniques described in this paper and forms the basis for a comprehensive USB security testing tool. Umap is written in Python and builds on the sample code provided with the Facedancer board. Figure 10 shows the basic help information.

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -h

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Usage: umap.py

Options:
  --version   show program's version number and exit
  -h, --help  show this help message and exit
  -L          list device classes
  -r REF      reference the VID/PID database (REF=VID:PID)
  -u          update the VID/PID database (Internet connectivity required)
  -i          identify all supported device classes on connected host
  -c CLS      identify if a specific class on the connected host is supported
  -m MASS     emulate a specific mass storage device (MASS=VID:PID:REV)
  -p PRINT    emulate a specific printer device (PRINT=VID:PID:REV)
  -k KEYB     emulate a specific keyboard device (KEYB=VID:PID:REV)
  -I IMAGE    emulate a specific image device (IMAGE=VID:PID:REV)
  -H HUB      emulate a specific hub device (IMAGE=VID:PID:REV)
  -f FUZZC    fuzz a specific class
  -O          Operating System identification (experimental)
  -d DLY      delay between enumeration attempts (seconds): Default=1
  -l LOG      log to a file
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 10: Umap basic help

Figure 11 shows the various USB device class types that umap currently understands.

 © Copyright 2013 NCC Group

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -L

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Name / ID
---------
Audio  -  0
Communications Device Class  -  1
Human Interface Device  -  2
Image  -  3
Printer  -  4
Mass Storage  -  5
Hub  -  6
Smart Card  -  7
Content Security  -  8
Video  -  9
Personal Healthcare  -  10
Application specific  -  11
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 11: The USB device classes that umap currently understands

Figure 12 shows umap identifying supported classes, sub-classes, and protocols

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -c 0

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Audio : Sub-class undefined : PR Protocol undefined
Audio : Audio control : PR Protocol undefined
Audio : Audio streaming : PR Protocol undefined
Audio : Midi streaming : PR Protocol undefined
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -c 3

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Image : Still image capture device : Bulk-only protocol... **SUPPORTED**

root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 12: Umap identifying supported classes, sub-classes and protocols

© Copyright 2013 NCC Group

Figure 13 shows the umap VID/PID lookup capability.

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -r 0079:0011

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Looking up VID= 0079 / PID= 0011
DragonRise Inc. Gamepad
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 13: The umap VID/PID lookup facility

Figure 14 shows umap performing operating system identification using some of the techniques described earlier in this paper.

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -O

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Attempting to identify the OS of the connected USB host.........
OS guess: Linux
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 14: The umap operating system identification capability

 © Copyright 2013 NCC Group

Figure 15 shows umap emulating an image class device (a digital stills camera).

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -I 1111:2222:3333

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Emulating image device: 1111 2222 3333
Facedancer reset
GoodFET monitor initialized
MAXUSB initialized
MAXUSB enabled
MAXUSB revision 19
MAXUSB connected device USB image device
USB image device received request dir=1, type=0, rec=0, r=6, v=256, i=0, l=64
USB image device received GET_DESCRIPTOR req 1, index 0, language 0x0000, length 64
MAXUSB wrote 12 01 00 02 00 00 00 40 da 04 74 23 10 00 01 02 03 01 to endpoint 0
USB image device received request dir=0, type=0, rec=0, r=5, v=7, i=0, l=0
USB image device received SET_ADDRESS request for address 7
USB image device received request dir=1, type=0, rec=0, r=6, v=256, i=0, l=18
USB image device received GET_DESCRIPTOR req 1, index 0, language 0x0000, length 18
MAXUSB wrote 12 01 00 02 00 00 00 40 da 04 74 23 10 00 01 02 03 01 to endpoint 0
USB image device received request dir=1, type=0, rec=0, r=6, v=1536, i=0, l=10
USB image device received GET_DESCRIPTOR req 6, index 0, language 0x0000, length 10
MAXUSB stalling endpoint 0
USB image device received request dir=1, type=0, rec=0, r=6, v=1536, i=0, l=10
USB image device received GET_DESCRIPTOR req 6, index 0, language 0x0000, length 10
```

Figure 15: Umap emulating a USB camera

Umap includes a large database of both generic and class-specific fuzzer test-cases, samples of which are shown in Figures 16 and 17.

© Copyright 2013 NCC Group

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap

testcases_class_independent = [
["Device_bLength_null","dev_bLength",0],
["Device_bLength_lower","dev_bLength",1],
["Device_bLength_higher","dev_bLength",20],
["Device_bLength_max","dev_bLength",0xff],
["Device_bDescriptorType_null","dev_bDescriptorType",0],
["Device_bDescriptorType_invalid","dev_bDescriptorType",0xff],
["Device_bMaxPacketSize0_null","dev_bMaxPacketSize0",0],
["Device_bMaxPacketSize0_null","dev_bMaxPacketSize0",0xff],

["String_Manufacturer_overflow","string_Manufacturer","A" * 126],
["String_Product_overflow","string_Product","A" * 126],
["String_Serial_overflow","string_Serial","A" * 126],
["String_Manufacturer_formatstring","string_Manufacturer","%x%n%n"],
["String_Product_formatstring","string_Product","%x%n%n"],
["String_Serial_formatstring","string_Serial","%x%n%n"],

["Configuration_bLength_null","conf_bLength",0],
["Configuration_bLength_lower","conf_bLength",1],
["Configuration_bLength_higher","conf_bLength",10],
["Configuration_bLength_max","conf_bLength",10],
["Configuration_bDescriptorType_null","conf_bDescriptorType",0],
["Configuration_bDescriptorType_invalid","conf_bDescriptorType",0xff],
["Configuration_wTotalLength_null","conf_wTotalLength",0],
["Configuration_wTotalLength_lower","conf_wTotalLength",1],
["Configuration_wTotalLength_higher","conf_wTotalLength",0xfff0],
["Configuration_wTotalLength_max","conf_wTotalLength",0xffff],
["Configuration_bNumInterfaces_null","conf_bNumInterfaces",0],
["Configuration_bNumInterfaces_higher","conf_bNumInterfaces",0xf0],
"testcases.py" 325L, 23863C                                   1,0-1         Top
```

Figure 16: Generic USB fuzz test cases

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap

# PIMA 15740
testcases_Image_class = [
["DeviceInfo_ContainerLength_null","DeviceInfo_ContainerLength",b'\x00\x00\x00\x00
'],
["DeviceInfo_ContainerLength_lower","DeviceInfo_ContainerLength",b'\x00\x00\x00\x0
1'],
["DeviceInfo_ContainerLength_higher","DeviceInfo_ContainerLength",b'\x00\x00\xff\x
ff'],
["DeviceInfo_ContainerLength_max","DeviceInfo_ContainerLength",b'\xff\xff\xff\xff'
],
["DeviceInfo_ContainerType_null","DeviceInfo_ContainerType",b'\x00\x00'],
["DeviceInfo_ContainerType_max","DeviceInfo_ContainerType",b'\xff\xff'],
["DeviceInfo_OperationCode_null","DeviceInfo_OperationCode",b'\x00\x00'],
["DeviceInfo_OperationCode_max","DeviceInfo_OperationCode",b'\xff\xff'],
["DeviceInfo_TransactionID_null","DeviceInfo_TransactionID",b'\x00\x00\x00\x00'],
["DeviceInfo_TransactionID_max","DeviceInfo_TransactionID",b'\xff\xff\xff\xff'],
["DeviceInfo_StandardVersion_null","DeviceInfo_StandardVersion",b'\x00\x00'],
["DeviceInfo_StandardVersion_max","DeviceInfo_StandardVersion",b'\xff\xff'],
["DeviceInfo_VendorExtensionID_null","DeviceInfo_VendorExtensionID",b'\x00\x00\x00
\x00'],
["DeviceInfo_VendorExtensionID_null","DeviceInfo_VendorExtensionID",b'\xff\xff\xff
\xff'],
["DeviceInfo_VendorExtensionVersion_null","DeviceInfo_VendorExtensionVersion",b'\x
00\x00'],
["DeviceInfo_VendorExtensionVersion_null","DeviceInfo_VendorExtensionVersion",b'\x
ff\xff'],
["DeviceInfo_VendorExtensionDesc_max","DeviceInfo_VendorExtensionDesc",b'\xff'],
["DeviceInfo_VendorExtensionDesc_overflow1","DeviceInfo_VendorExtensionDesc",b'\xf
e' + b'\x61\x00' * 254 + b'\x00\x00'],
@
                                                              58,0-1        18%
```

Figure 17: Class-specific USB fuzz test cases

© Copyright 2013 NCC Group

Figure 18 shows umap fuzzing a USB host.

```
root@ubuntu: ~/svn/goodfet/trunk/client/umap
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -f 3

---------------------------------------
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
---------------------------------------

Fuzzing Enumeration phase...
0000: Enumeration phase: Device_bLength_null
0001: Enumeration phase: Device_bLength_lower
0002: Enumeration phase: Device_bLength_higher
0003: Enumeration phase: Device_bLength_max
0004: Enumeration phase: Device_bDescriptorType_null
0005: Enumeration phase: Device_bDescriptorType_invalid
0006: Enumeration phase: Device_bMaxPacketSize0_null
0007: Enumeration phase: Device_bMaxPacketSize0_null
0008: Enumeration phase: String_Manufacturer_overflow
0009: Enumeration phase: String_Product_overflow
000a: Enumeration phase: String_Serial_overflow
000b: Enumeration phase: String_Manufacturer_formatstring
000c: Enumeration phase: String_Product_formatstring
000d: Enumeration phase: String_Serial_formatstring
000e: Enumeration phase: Configuration_bLength_null
000f: Enumeration phase: Configuration_bLength_lower
0010: Enumeration phase: Configuration_bLength_higher
0011: Enumeration phase: Configuration_bLength_max
0012: Enumeration phase: Configuration_bDescriptorType_null
```

Figure 18: Umap fuzzing a USB host

© Copyright 2013 NCC Group

9 Conclusions

The goal of this research was to identify ways of revealing configuration information about a connected USB host. This is useful because it allows us to streamline any subsequent fuzzing process by identifying supported USB functionality, and to enumerate operating system and application information that may be useful for other security testing. The major problem with trying to identify information about the configuration of the host is that USB is a master–slave relationship and the device is the slave, so a device cannot query a host.

By emulating specific USB device classes such as mass storage and printer, it was possible to identify which generic class drivers were supported by the connected host. This process was refined to also emulate (and therefore identify) supported sub-classes and protocols. In order to identify non-generic class drivers, which are referenced by their vendor and product IDs, a brute force approach was demonstrated which uses the public VID/PID database.

Due to the complexity of the USB protocol there are many different implementations of USB host functionality. A number of different techniques were developed to identify a host; these included analysing:

- The order of descriptor requests
- The number of times different descriptors were requested
- The use of specific USB commands
- Class-specific communication

These techniques demonstrated that the host operating system, and in some cases applications running on the host, could be identified.

A tool called umap was developed during the research to demonstrate these different techniques and also to perform targeted fuzzing once the information-gathering phase was complete. Possible uses for umap include Endpoint Protection System configuration assessment, USB host fuzzing and general host security audit (for USB).

 © Copyright 2013 NCC Group

10 References and Further Reading

Davis, Undermining Security Barriers, *media.blackhat.com*, <http://media.blackhat.com/bh-us-11/Davis/BH_US_11-Davis_USB_Slides.pdf>, accessed 6 August 2013

Dominguez Vega, USB Attacks: Fun with Plug and 0wn, *labs.mwrinfosecurity.com*, <http://labs.mwrinfosecurity.com/assets/135/mwri_t2-usb-fun-with-plug-and-0wn_2009-10-29.pdf>, accessed 6 August 2013

GoodFET, GoodFET – Facedancer21, *goodfet.sourceforge.net*, <http://goodfet.sourceforge.net/hardware/facedancer21/>, accessed 6 August 2013

Goodspeed, Writing a thumbdrive from scratch: Prototyping active disk antiforensics, *www.youtube.com*, <http://www.youtube.com/watch?v=D8Im0_KUEf8>, accessed 6 August 2013

Larimer, Beyond Autorun: Exploiting vulnerabilities with removable storage, *media.blackhat.com*, <https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf >, accessed 6 August 2013

MOD, Microsoft OS Descriptors, *msdn.microsoft.com*, <http://msdn.microsoft.com/en-us/library/windows/hardware/gg463179.aspx>, accessed 6 August 2013

MQP, Packet-Master USB500 AG, *www.mqp.com*, <http://www.mqp.com/usb500.htm>, accessed 6 August 2013

USBIF, USB Implementers Forum, *www.usb.org*, < http://www.usb.org/about>, accessed 6 August 2013

11 Glossary

ATAPI - AT Attachment Packet Interface
BBB - Bulk-only transport (also called BOT)
CBI - Control/Bulk/Interrupt
CDC - Communication and Device Control
eHCI - Enhanced Host Controller Interface
HID - Human Interface Device
HSPA - High Speed Packet Access
IEE 1667 .Protocol for Authentication in Host Attachments of Transient Storage Devices
LSD FS - Lockable Storage Devices Feature Specification
MOD - Microsoft OS descriptor
oHCI - Open Host Controller Interface
PID - Product ID
QIC-157 Quarter Inch Cartridge (standard for streaming tape)
RPC - Remote Procedure Call
SCSI - Small Computer System Interface
SFF-8070i - ATAPI specification for floppy disks
UAS - USB Attached SCSI
UFI - USB Floppy Interface
uHCI - Universal Host Controller Interface
USBIF - Universal Serial Bus Implementers Forum
USB . Universal Serial Bus
VID . Vendor ID
xHCI - Extensible Host Controller Interface

 © Copyright 2013 NCC Group