Affordable USB Forensics

Philip A. Polstra, Sr.
Computer Information Systems
University of Dubuque
Dubuque, IA 52001
ppolstra@dbq.edu

Abstract

This talk/paper will provide a brief overview of USB basics. USB mass storage devices, the most common of which are USB flash (or thumb) drives will be covered in detail. Simple inexpensive devices for creating forensic duplicates of USB flash drives and blocking write access to flash drives will be discussed. Schematics and source code will be provided for all devices presented.

1 Introduction

Since its introduction USB has quickly taken over the PC peripheral market as the default interconnect standard. USB flash drives have replaced CD-ROM/DVD-ROMs and floppy drives as a means of exchanging data and providing booting alternatives.

While everyone uses USB devices, few understand how they work. A family of live Linux distributions has become popular among security practitioners. Forensic investigators are extremely likely to encounter evidentiary flash drives during the course of their work. In this paper, an inexpensive devices for creating forensic duplicates of USB flash drives and for blocking USB write operations are discussed.

1.1 USB History

Before the introduction of Universal Serial Bus (USB), devices were connected via non-universal serial, PS/2 ports, & LPT ports. In 1996 USB 1.0 was introduced. USB 1.0 supports transport speeds of 1.5 Mbps (low speed) and 12 Mbps (full speed). In 1998 USB 1.1 was introduced. USB 1.1 was essentially a bug fix for some problems in USB 1.0. Two years later in 2000 USB 2.0 was created. USB 2.0 added a third speed of 480 Mbps known as high speed. Following the flurry of activity and releases every two years, it was not until eight years later in 2008 that USB 3.0 came out. USB 3.0 introduced a new speed of up to 5 Gbps known as super speed. Unlike previous versions of USB, USB 3.0 requires a different connector with separate wires for super speed transmissions.[2]

1.2 USB Hardware

USB utilizes a simple 4-wire connection (power, ground, 2 data wires). Cabling prevents improper connections which were a problem in non-universal serial connections. USB devices are hot pluggable. Differential voltages are used to provide greater immunity to noise than what is achieved with a typical serial connection. Cable lengths up to 16 feet are possible. Cable length depends on speed, with high speeds dictating shorter cables.[1,2]

1.3 USB Software

Configuration of USB devices is automatic requiring no settable jumpers. The process by which a USB device is discovered by a host (PC) is known as enumeration. During the enumeration process the host queries the device for a set of descriptors. USB devices indicate their abilities by stating they support various standard device classes with corresponding drivers. Some of the more common USB device classes include human interface device (HID), printer, audio, and mass storage. This paper will primarily cover mass storage devices which are commonly referred to as USB flash or thumb drives.

1.4 Connecting a Device

Once a device is connected, a twelve-step process is begun.[2] The steps consist of:

1. Device is connected
2. Hub detects
3. Host (PC) is informed of new device
4. Hub determines device speed capability as indicated by location of pull-up resistors
5. Hub resets the device
6. Host determines if device is capable of high speed (using chirps)
7. Hub establishes a signal path
8. Host requests descriptor from device to determine max packet size
9. Host assigns an address
10. Host learns devices capabilities
11. Host assigns and loads an appropriate device driver (INF file)
12. Device driver selects a configuration

1.5 USB Endpoints

Endpoints are the virtual wires for USB communications. All endpoints are one way with in/out direction specified relative to host. In most cases, packet fragmentation, handshaking, etc. is done by hardware. The high bit of an endpoint address tells direction with 1 indicating in (from the device to the host) and 0 representing out (from the host to the device). There are four types of endpoints: control, bulk transport, interrupt, and isochronous.[2]

1.5.1 Control Endpoints

The primary mechanism for most devices to communicate with a host is via a control endpoint. Every device must have at least one in and out control endpoint which is often referred to as endpoint zero (EP0). Device must respond to standard requests on EP0. Standard requests include getting and setting address, descriptors, power, and status. Devices may also respond to class specific and vendor specific requests. Transfer have two or three transport stages: setup, data (optional), and status.[2]

In the setup stage the host sends a setup token, then data packet containing setup request. If the device receives a valid setup packet, an acknowledgment (ACK) is returned. The setup request is 8 bytes in length. The first byte is bitmap telling type of request & recipient (device, interface, endpoint). The remaining bytes are parameters for request and response.[1,2]

During the optional data stage requested information is transmitted to or from the host as appropriate.[1,2]

In the status stage a zero length data packet (ZLDP) is from the device to the host to acknowledge successful receipt of a valid command.[1,2]

1.5.2 Interrupt and Isochronous Endpoints

Interrupt endpoints are used to avoid polling and busy waits for devices with infrequent communications. Keyboards are a good example of a device that uses interrupt endpoints. Devices utilizing interrupt endpoints are typically low speed which allows for longer cables, and better error tolerances.[1]

Isochronous endpoints are used in situations where guaranteed bandwidth is required. Such endpoints are primarily used for time-critical apps such as streaming media. More information on these types of endpoints and USB in general can be found at [1].

1.5.3 Bulk Endpoints

Bulk endpoints are used when a large amount of data is to be transmitted. No latency guarantees are provided for bulk endpoints. Good performance is achieved with bulk endpoints on an otherwise idle bus. Bulk transports are superseded by all other transport types. Only full speed (8-64 byte packets) and high speed (512 byte packets) transmissions are supported when using bulk endpoints. Bulk endpoints are used extensively in USB flash drives (and external hard drives). Bulk transactions consist of a token packet, 0 or more data packets, and an ACK handshake packet (if successful).[1,2]

1.6 Descriptors

Descriptors are used to describe various USB objects. The have a standard format. The first byte is the length of the descriptor in bytes (so the host knows when to stop reading). The second byte determines type of descriptor. The remaining bytes are the descriptor itself. There are several common types of descriptors including device, configuration, interface, endpoint, and string descriptors.[1,2]

1.6.1 Device Descriptors

Devices descriptors are 18 bytes and length. They contain basic information about a device such as its class, packet size, vendor identifier, and number of configurations. Table 1 details the fields found in a device descriptor.[1,2]

Offset	Field	Size	Value	Description
0	bLength	1	Number	18 bytes
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	0x200
4	bDeviceClass	1	Class	Class Code
5	bDeviceSubClass	1	SubClass	Subclass Code
6	bDeviceProtocol	1	Protocol	Protocol Code
7	bMaxPacketSize	1	Number	Maxi Packet Size EP0
8	idVendor	2	ID	Vendor ID
10	idProduct	2	ID	Product ID
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manu Descriptor
15	iProduct	1	Index	Index of Prod Descriptor
16	iSerialNumber	1	Index	Index of SN Descriptor
17	bNumConfigurations	1	Integer	Num Configurations

Table 1: Device Descriptor Format

1.6.2 Configuration Descriptors

Every device has at least on configuration descriptor. The configuration descriptor consists of a 9 byte header and then one or more interface descriptors each of which contain one or more endpoint descriptors. Typically a host will ask for this descriptor which contains the total length including all sub-descriptors. A second request is then sent to the device for the full descriptor. Details for the configuration descriptor header are provided in Table 2.[1,2]

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size in Bytes
1	bDescriptorType	1	Constant	0x02
2	wTotalLength	2	Number	Total data returned
4	bNumInterfaces	1	Number	Num Interfaces
5	bConfigurationValue	1	Number	Configuration number
6	iConfiguration	1	Index	String Descriptor
7	bmAttributes	1	Bitmap	b7 Reserved, set to 1. b6 Self Powered b5 Remote Wakeup b4..0 Reserved 0.
8	bMaxPower	1	mA	Max Power in mA/2

Table 2: Configuration Descriptor Header

1.6.3 Interface Descriptors

Every device has at least one interface descriptor. The interface descriptor is described in Table 3.[1,2]

Offset	Field	Size	Value	Description
0	bLength	1	Number	9 Bytes
1	bDescriptorType	1	Constant	0x04
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints used
5	bInterfaceClass	1	Class	Class Code
6	bInterfaceSubClass	1	SubClass	Subclass Code
7	bInterfaceProtocol	1	Protocol	Protocol Code
8	iInterface	1	Index	Index of String Descriptor

Table 3: Interface Descriptor Format

1.6.4 Interface Descriptors

Each device has at least one endpoint. The endpoint descriptor format is presented in Table 4.[1,2]

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	B0..3 Endpoint Number. b4..6 Reserved. Set to Zero b7 Direction 0 = Out, 1 = In
3	bmAttributes	1	Bitmap	b0..1 Transfer Type 10 = Bulk b2..7 are reserved. I
4	wMaxPacketSize	2	Number	Maximum Packet Size
6	bInterval	1	Number	Interval for polling endpoint data

Table 4: Endpoint Descriptor Format

1.6.5 String Descriptors

Various items are described by Unicode strings stored in string descriptors. Mass storage devices are required to have a serial number string. At a minimum the the last 13 digits of the serial number must be unique for every device with a particular manufacturer and product identifier. The string descriptor format is provided in Table 5.[1,2]

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	String Descriptor (0x03)
2	bString	n	Unicode	Unicode Encoded String

Table 5: String Descriptor Format

2 USB Mass Storage Devices

USB mass storage devices have become a popular way to store, backup, and share data. The most common mass storage device uses NAND flash memory.[3] Such devices are commonly referred to as flash or thumb drives. Unlike many USB devices, mass storage devices utilize bulk endpoints for control commands (as opposed to using control endpoints). Because they use bulk endpoints, mass storage devices are also known as bulk only mass storage (BOMS) or bulk-bulk-bulk (BBB) devices.[3]

2.1 Communicating with Mass Storage devices

There are two or three phases in each mass storage transaction: command block wrapper transmission (CBW), data-transport (optional), and command status wrapper response (CSW). Commands are sent to drive using a CBW. Each CBW contains a command block (CB) with actual command. Nearly all drives use a (reduced) SCSI command set. Commands requiring data transport will send/receive on bulk endpoints. All transactions are terminated by a Command Status Wrapper (CSW). The CSW indicates success or failure for the transaction.[3]

2.1.1 Command Block Wrapper Phase

The 31-byte CBW is sent to the device on the bulk out endpoint. The last 16 bytes contain the command block (CB) itself. The length of each CB varies from 6-16 bytes depending on the command.[3] The following C structure describes the CBW:

```
typedef struct _USB_MSI_CBW {
        unsigned long dCBWSignature; //0x43425355 “USBC”
        unsigned long dCBWTag; // associates CBW with CSW response
        unsigned long dCBWDataTransferLength; // bytes to send or receive
        unsigned char bCBWFlags; // bit 7 0=OUT, 1=IN all others zero
        unsigned char bCBWLUN; // logical unit number (usually zero)
        unsigned char bCBWCBLength; // 3 hi bits zero, rest bytes in CB
        unsigned char bCBWCB[16]; // the actual command block (>= 6 bytes)
} USB_MSI_CBW;
```

The first byte of the CB is the command. The following C structures describe the CB for FORMAT UNIT and READ(10) commands.[6] Further details may be found in [3].

```
typedef struct _CB_FORMAT_UNIT {
        unsigned char OperationCode; //must be 0x04
        unsigned char LUN:3; // logical unit number (usually zero)
        unsigned char FmtData:1; // if 1, extra parameters follow command
        unsigned char CmpLst:1; // if 0, partial list of defects, 1, complete
        unsigned char DefectListFormat:3; //000 = 32-bit LBAs
        unsigned char VendorSpecific; //vendor specific code
        unsigned short Interleave; //0x0000 = use vendor default
        unsigned char Control;
} CB_FORMAT_UNIT;
typedef struct _CB_READ10 {
        unsigned char OperationCode; //must be 0x28
        unsigned char RelativeAddress:1; // normally 0
        unsigned char Resv:2;
        unsigned char FUA:1; // 1=force unit access, don't use cache
        unsigned char DPO:1; // 1=disable page out
        unsigned char LUN:3; //logical unit number
        unsigned long LBA; //logical block address (sector number)
        unsigned char Reserved;
        unsigned short TransferLength;
        unsigned char Control;
} CB_READ10;
```

2.1.2 The Data Transport Phase

Commands that involve the exchange of data will send or receive data on bulk endpoints as appropriate. Not all commands have a data phase.[3]

2.1.3 The CSW Phase

Each command returns a CSW. The CSW is used to report success or failure. There are two possible failure codes. One failure code indicates a general error and the other indicates that a data phase error has occurred. A host must immediately execute a REPORT SENSE command after an error has been reported in the CSW in order to determine the exact nature of the error that has occurred.[3] The CSW is described by the following C structure:

```
typedef struct _USB_MSI_CSW {
        unsigned long dCSWSignature; //0x53425355 "USBS"
        unsigned long dCSWTag; // associate CBW with CSW response
        unsigned long dCSWDataResidue; // difference between requested data
and actual
        unsigned char bCSWStatus; //00=pass, 01=fail, 02=phase error, reset
} USB_MSI_CSW;
```

3 Creating Forensic Duplicates

USB flash drives present themselves as SCSI hard drives with 512 byte blocks. Larger block sizes are possible, but uncommon. A 512 byte block requires 528 bytes of storage because the block requires 16 bytes of error correction code (ECC).[3]

Creating a forensic copy of a flash drive requires a sector by sector copy to be performed. Unlike hard drives , there is no place between sectors to store hidden information. Care should be taken when creating forensic duplicates of flash drives. Mounting a flash drive on most operating systems will result in alteration of the drive when access timestamps are updated. USB-friendly microcontrollers such as the Vinculum II by FTDI can be used to make such copies safely and without the need for a computer.[4] The author has fully described two portable forensic duplicator based on the FTDI chip at [7]. Source code and construction details for these duplicators are available from the author on request.

4 Blocking USB Mass Storage Write Operations

There are a couple of free ways to block USB write operations. Some, mostly older, flash drives have write protect switches. Additionally, for Windows users, all write operations to USB mass storage devices can be blocked by creating the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ StorageDevicePolicies\ WriteProtect key in the registry and setting its value to 1.[7]

Commercial write blockers are available. At several hundred dollars for device, they are not practical for everyday use. Utilizing techniques described in this paper, write blockers can be constructed for less than $30. Full source code and construction details are available upon request.

4.1 Chip Choice

The FTDI Vinculum II (VNC2) microcontroller was selected for this project. The VNC2 features include

- 2 full-speed USB 2.0 interfaces (host or slave capable)
- 256 KB E-flash memory
- 16 KB RAM
- 2 SPI slave and 1 SPI master interfaces
- Easy-to-use IDE
- Simultaneous multiple file access on BOMS devices
- Several development modules available[4,7]

A compact device can be constructed from a 32-pin V2DIP1-32 module. This requires 4 solder connections to attach a standard USB A cable. The Vinco Arduino-like development board can be used to create a slightly larger device which does not require any soldering.[4]

4.2 Write Blocker Operation

The device needs to block bad command blocks that could modify the drive. An easy approach would be to block the commands that could alter the flash drive. Best practice and future proofing would dictate white listing instead. All VNC2 chips have the same memory and flash storage. They only differ in number of GPIO lines available. Same software will almost run on both packages provided additional code required to power Vinco host port is included.

The write blocker presented here features a multi-threaded design. One thread is associated with the slave port to make it appear as a BOMS device to a PC. This thread watches the control endpoint and services requests from the PC. One thread is associated with the host port for talking to the flash drive. This thread enumerates the device and gets endpoints. This thread then periodically checks to see if the drive is still there. The main thread bridges the slave and host ports. Non-CBW packets (data packets) are passed through to the host port. Whitelisted CBWs are also passed on to the host port. USB host & slave drivers built in to the Vinculum operating system (VOS) create additional threads.

The main loop consists of an infinite loop which receives CBWs from the PC and then calls an appropriate handler. An excerpt appears here:

```
usbSlaveBoms_readCbw(cbw, slaveBomsCtx);
switch (cbw->cb.formated.command)
{
    case BOMS_INQUIRY:
    handle_inquiry(cbw);
    break;
    …
}
```

Handlers take various forms depending on expected data to be sent or received. Some commands that are blocked report success despite the fact that they fail. This is done for some commands because Windows handles failure of some commands such as write poorly. As an example, the BOMS INQUIRY handler appears here:

```
void handle_inquiry(boms_cbw_t *cbw)
{
    unsigned char buffer[64];
    unsigned short responseSize;
    boms_csw_t csw;
    if (forward_cbw_to_device(cbw))
    {
        if (responseSize = receive_data_from_device(&buffer[0], 36))
        {
            forward_data_to_slave(&buffer[0], responseSize);
            if (receive_csw_from_device(&csw))
```

```
                        {
                                forward_csw_to_slave(&csw);
                        }
                }
        }
}
```

4.3 Recommended Usage

There are two recommended uses for the device. The first is to block writes on Windows computers. This allow a security practitioner to avoid risking damage to flash drives containing security tools. Anti-virus software will often try to delete such tools as they can be interpreted as malware.
Forensic examination of mass storage devices is the other recommended use of the device. Linux is recommended for forensics work for a number of reasons. Windows might miss or mishandle upper locial units (LUNs) on subject flash drives. Additionally, Linux has all the non-FAT filesystems an investigator would likely encounter.

5 Conclusions

USB flash drives have become commonly used devices. Using the techniques described in this paper small, inexpensive, and effective devices can be constructed in order to preserve and replicate data stored on USB mass storage devices.

References

[1] The Universal Serial Bus Documentation http://www.usb.org/developers/docs.
[2] USB Complete: The Developers Guide (4th ed.) by Jan Axelson.
[3] USB Mass Storage: Designing and Programming Devices and Embedded Hosts by Jan Axelson.
[4] FTDI Application Notes http://www.ftdichip.com
[5] SCSI References http://seagate.com
[6] Embedded USB Design by Example by John Hyde
[7] Phil Polstra- 44Con USB Flash Drive Forensics Video http://www.youtube.com/watch?v=CIVGzG0W-DM