Bypassing Endpoint Security for $20 or Less

Philip A. Polstra, Sr.
@ppolstra
ppolstra.blogspot.com
Roadmap

- Why this talk?
- Who is this dude talking at me?
- Brief history of USB
- How does USB work?
- It’s all descriptors and endpoints
- Bulk-only mass storage devices
- Bypassing endpoint security
- Microcontrollers are fun (and cheap)
- Food for thought
Why this talk?

- Many organizations have begun to use endpoint security programs to restrict use of portable media.
- Many software tools do the USB equivalent of MAC filtering – only allow authorized VID/PID.
- For $18-30 can easily construct device to allow any mass storage device to impersonate authorized device.
- Allows injection/extraction.
Who am I anyway?

• Teach computer security at a private university
• Like to hack hardware
• Have been known to fly and build airplanes
• Been known to play with USB devices
Brief History of USB

- Non-universal serial, PS/2 ports, & LPT
- 1996 USB 1.0 (1.5 or 12 Mbps)
- 1998 USB 1.1
- 2000 USB 2.0 (1.5, 12, or 480 Mbps)
- Long pause
- 2008 USB 3.0 (up to 5 Gbps)
HOW DOES USB WORK?
Hardware

- Simple 4-wire connection (power, ground, 2 data wires)
- Cabling prevents improper connections
- Hot pluggable
- Differential voltages provide greater immunity to noise
- Cable lengths up to 16 feet are possible

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Cable color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VBUS</td>
<td>Red</td>
<td>+5 V</td>
</tr>
<tr>
<td>2</td>
<td>D−</td>
<td>White</td>
<td>Data −</td>
</tr>
<tr>
<td>3</td>
<td>D+</td>
<td>Green</td>
<td>Data +</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
<td>Black</td>
<td>Ground</td>
</tr>
</tbody>
</table>
Software

- Automatic configuration
- No settable jumpers
- Enumeration
- Standard device classes with corresponding drivers
  - HID
  - Printer
  - Audio
  - Mass Storage
Connecting a Device

• Device is connected
• Hub detects
• Host (PC) is informed of new device
• Hub determines device speed capability as indicated by location of pull-up resistors
• Hub resets the device
• Host determines if device is capable of high speed (using chirps)
• Hub establishes a signal path
• Host requests descriptor from device to determine max packet size
• Host assigns an address
• Host learns devices capabilities
• Host assigns and loads an appropriate device driver (INF file)
• Device driver selects a configuration
IT’S ALL DESCRIPTORS AND ENDPOINTS
Endpoints

- The virtual wire for USB communications
- All endpoints are one way (direction relative to host)
- Packet fragmentation, handshaking, etc. done by hardware (usually)
- High bit of address tells direction 1=in 0=out
- Types of endpoints
  - Control
  - Bulk transport
  - Interrupt
  - Isochronous
Control Endpoints

• Primary mechanism for most devices to communicate with host

• Every device must have at least one in and out control endpoint EP0

• Device must respond to standard requests
  – Get/set address, descriptors, power, and status

• Device may respond to class specific requests

• Device may respond to vendor specific requests
Control Endpoints (continued)

- May have up to 3 transport stages: Setup, Data, Status
- Setup stage
  - Host sends Setup token then data packet containing setup request
  - If device receives a valid setup packet, an ACK is returned
  - Setup request is 8 bytes
    - 1st byte is bitmap telling type of request & recipient (device, interface, endpoint)
    - Remaining bytes are parameters for request and response
- Data stage (optional) – requested info transmitted
- Status stage – zero length data packet sent as ACK on success
Interrupt & Isochronous Endpoints

• Interrupt endpoints
  – Used to avoid polling and busy waits
  – Keyboards are a good example
  – Usually low speed (allows for longer cables, etc.)

• Isochronous endpoints
  – Guaranteed bandwidth
  – Used primarily for time-critical apps such as streaming media
Bulk Endpoints

• No latency guarantees
• Good performance on an idle bus
• Superseded by all other transport types
• Full (8-64 byte packets) & high speed (512 byte packets) only
• Used extensively in USB flash drives (and external hard drives)
• Transactions consist of a token packet, 0 or more data packets, and an ACK handshake packet (if successful)
Descriptors

- They describe things (duh!)
- Have a standard format
  - 1\textsuperscript{st} byte is the length in bytes (so you known when you’re done)
  - 2\textsuperscript{nd} byte determines type of descriptor
  - Remaining bytes are the descriptor itself
- Common types
  - Device: tells you basic info about the device
  - Configuration: how much power needed, number of interfaces, etc.
  - Interface: How do I talk to the device
  - Endpoint: Direction, type, number, etc.
  - String: Describe something in unicode text
## Device Descriptor

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

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>9 Bytes</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>0x04</td>
</tr>
<tr>
<td>2</td>
<td>bInterfaceNumber</td>
<td>1</td>
<td>Number</td>
<td>Number of Interface</td>
</tr>
<tr>
<td>3</td>
<td>bAlternateSetting</td>
<td>1</td>
<td>Number</td>
<td>Alternative setting</td>
</tr>
<tr>
<td>4</td>
<td>bNumEndpoints</td>
<td>1</td>
<td>Number</td>
<td>Number of Endpoints used</td>
</tr>
<tr>
<td>5</td>
<td>bInterfaceClass</td>
<td>1</td>
<td>Class</td>
<td>Class Code</td>
</tr>
<tr>
<td>6</td>
<td>bInterfaceSubClass</td>
<td>1</td>
<td>SubClass</td>
<td>Subclass Code</td>
</tr>
<tr>
<td>7</td>
<td>bInterfaceProtocol</td>
<td>1</td>
<td>Protocol</td>
<td>Protocol Code</td>
</tr>
<tr>
<td>8</td>
<td>iInterface</td>
<td>1</td>
<td>Index</td>
<td>Index of String Descriptor</td>
</tr>
</tbody>
</table>
## Endpoint Descriptor

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>Size of Descriptor in Bytes</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>String Descriptor (0x03)</td>
</tr>
<tr>
<td>2</td>
<td>bString</td>
<td>n</td>
<td>Unicode</td>
<td>Unicode Encoded String</td>
</tr>
</tbody>
</table>

Note: String 0 is a special case that lists available languages. Most common is 0x0409 – U.S. English
Now that we have learned a little about general devices, without further delay…

BULK-ONLY MASS STORAGE DEVICES
USB Flash Drives

- Hardware
- Software
- Filesystems
- Talk to a flash drive
Hardware
Hardware (continued)

- Typically utilize NAND flash memory
- Memory degrades after 10,000 write cycles
- Most chips not even close to high-speed USB speed (480 Mbps)
- Can only be written in blocks (usually 512, 2048, or 4096 bytes)
- Chips are somewhat easily removed from damaged drives for forensic recovery
- Some controllers have JTAG capability which can be used for memory access
- Some controller chips steal some flash memory for themselves
Hardware (continued)

- Nearly all flash drives present themselves as SCSI hard drives
- "Hard drive" sectors are typically 512, 2048, or 4096 bytes
- SCSI transparent command set is used
- Most drives are formatted as one partition or logical unit
  - Additional logical units can hide info from Windows machines
- Reported size may not match actual media size
  - Info can be hidden in higher sectors
  - Some cheap drives are out there that grossly over report size
  - A typical 512 byte sector needs 16 bytes for error correction
Software

- Usually implemented in firmware within specialized controller chips
- Must:
  - Detect communication directed at drive
  - Respond to standard requests
  - Check for errors
  - Manage power
  - Exchange data
Filesystems

- Most preformatted with FAT or FAT32
- NTFS
- TrueFFS
- ExtremeFFS
- JFFS
- YAFFS
- Various UNIX/Linux file systems
Talking to a Flash Drive

- Bulk-Only Mass Storage (aka BBB) protocol used
  - All communications use bulk endpoints
  - Three phases: CBW, data-transport (optional), CSW
  - Commands sent to drive using a Command Block Wrapper (CBW)
  - CBW contains 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)
Command Block Wrapper

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;
Command Block

- 6-16 bytes depending on command
- Command is first byte
- Format Unit Example:

```c
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;
```
Command Block (continued)

- Read (10) Example:

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;
Command Block (continued)

- Some Common SCSI Commands:
  FORMAT_UNIT=0x4, //required
  INQUIRY=0x12, //required
  MODE_SELECT6=0x15,
  MODE_SELECT10=0x55,
  MODE_SENSE6=0x1A,
  MODE_SENSE10=0x5A,
  READ6=0x08, //required
  READ10=0x28, //required
  READ12=0xA8,
  READ_CAPACITY10=0x25, //required
  READ_FORMAT_CAPACITY=0x23,
  REPORT_LUNS=0xA0, //required
  REQUESTSENSE=0x03, //required
  SEND_DIAGNOSTIC=0x1D, //required
  START_STOP_UNIT=0x1B,
  SYNCHRONIZE_CACHE10=0x35,
  TEST_UNIT_READ=0x00, //required
  VERIFY10=0x2F,
  WRITE6=0x0A, //required
  WRITE10=0x2A,
  WRITE12=0xAA
Command Status Wrapper

- Read Sense command can be used for details on failed operations

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;
Now that we know how bulk-only mass storage devices work…

HOW DO I BYPASS ENDPOINT SECURITY?
Impersonating another device

- Social engineering USB style
- Providing an authorized VID/PID allows device connection
  - Backdoors and other useful items can be injected
  - Information can be extracted to portable media
- Device design allows optional write blocking
Enough background. Let the fun begin…

MICROCONTROLLERS ARE FUN (AND CHEAP)
Fun with Microcontrollers

- Chip Choice
- A Microcontroller-Based Impersonator
Chip Choice Options

• AVR (as found in Arduino family)
  – Cheap
  – Well understood
  – Loads of code out there
  – Too underpowered to do USB without external components (<20MHz)

• PIC family
  – Relatively cheap
  – Programming somewhat more involved than AVR
  – Newer chips SMD only, not easy DIP package
  – Some USB device code, but not host code out there
Chip Choice Winner

- None of the above
- FTDI Vinculum II
  - Relatively new chip
  - A little faster than AVRs (48 MHz)
  - Real-time multi-threaded OS
  - Libraries for several standard USB classes
    - BOMS is one – but we can’t use it for this project, unfortunately
  - Unlike AVR, different pin packages differ only with GPIO lines available
    - Same flash memory
    - Same RAM
Chip Choice

• FTDI Vinculum II dual USB host/slave controller
  – 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
  – Convenient for prototyping (only SMD chips available)
  – Cheap enough to embed in final device
  – One format is Arduino clone (Vinco)
Chip Choice (continued)
Chip Choice (continued)
Chip Choice (continued)
Chip Choice (continued)
Package A - Small & only 4 Pins to Solder*

*If you aren’t fond of useful information displays, lights and buttons that is.
Package B – Slightly Larger-No Soldering*

*See disclaimer on previous slide.
Microcontroller-Based Impersonator

- Enumerate an attached mass storage drive
- When PC attempts to connect drive try to provide an authorized VID/PID
- If unsuccessful try another VID/PID till it works
Impersonator High-Level Design

- One thread associated with slave port to appear as a BOMS device
  - One thread watches control endpoint and services requests from host
- One thread associated with the host port for talking to flash drive
  - Thread enumerates the device and gets endpoints. Then periodically checks to see if the drive is still there
- Main thread bridges slave and host
  - Non-CBW packets (data packets) are passed through to host port
  - Whitelisted CBWs are also passed on to host port (if write blocking)
- Timer thread
  - When enumeration starts timer is set
  - If drive is not connected another VID/PID is tried
- Button thread
  - Reads buttons and adjusts status accordingly
The Main Thread

• Waits for CBW packets to arrive on Bulk Out endpoint
• Calls appropriate handler function based on command
  – Whitelisted commands:
    • Forward CBW to drive
    • Perform Data phase (if any) with drive and forward to PC
    • Received CSW from device and forward to PC
  – Non-whitelisted commands (when write blocking):
    • ACK CBW
    • Fake Data phase (if any)
    • Return CSW to PC
  – Some commands return success because Windows is unhappy with failures
Main Loop

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

    ...
}
```
Example Handler

```c
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);
            }
        }
    }
}
```
Timer Thread

- When device descriptor requested start 1 second timer
- When the enumeration complete reset timer
- If timer expires try the next VID/PID from list
- At end of list could resort to brute force
Complications

- Windows & Linux treat drives differently
  - Windows will try to look for and autoplay media
  - Windows doesn’t appear to see other than first LUN
  - Early prototype experience (with writeblocker this is based on)
    - Worked fine under Linux
    - Caused BSoD on Windows (exploit?)
  - Linux seems to pull in a lot of data up front
  - Windows misbehaves if you correctly fail some commands such as Write
Endpoint security on Linux

• Can use udev rules to emulate Windows endpoint security software on Linux

• Open source provides a great value
  – Better value
  – Equally ineffective, but at a better price
IT’S DEMO TIME!

And now what you really wanted to see...

All UR EndPoint
R belong 2 us
Food for thought

- Speed up process by searching registry for previously mounted devices
  - USBDevView or something similar might be helpful
- Use larger device to divine authorized device then use a collection of smaller devices preprogrammed to appropriate VID/PID
- Like all devices this may be thwarted
  - Device operates at full speed only
  - Endpoint software could use proprietary drivers
    - Security through obscurity?
References

- USB Complete: The Developers Guide (4th ed.) by Jan Axelson
- USB Mass Storage: Designing and Programming Devices and Embedded Hosts by Jan Axelson
- [http://www.usb.org](http://www.usb.org)
- [http://www.ftdichip.com](http://www.ftdichip.com) for more on VNC2
- [http://seagate.com](http://seagate.com) for SCSI references
- Embedded USB Design by Example by John Hyde
- My 44Con USB Flash Drive Forensics Video [http://www.youtube.com/watch?v=ClVGzG0W-DM](http://www.youtube.com/watch?v=ClVGzG0W-DM)
- Schematics and source code are available
  - Git hub [usb-impersonator](https://github.com/usb-impersonator)
  - Email [ppolstra@gmail.com](mailto:ppolstra@gmail.com)
  - Twitter [@ppolstra](https://twitter.com/PPolstra)
Questions?