# Reverse Engineering USB Device Drivers

Jan Hauffa

(jhauffa@gmail.com)

# Outline

1. What's this all about?
2. USB fundamentals
3. Sniffing and analyzing USB traffic
4. Case study

# Why reverse engineering?

- hardware manufacturers that...
  - only provide Windows drivers
  - only provide Linux binary drivers for x86
  - don't provide any documentation

  → http://linuxdriverproject.org/foswiki/bin/view/Main/DriversNeeded
- „repurposing“ of hardware

> What exactly are you trying to protect?
> The communication protocol, some image processing
> functions, the firmware?

**Mainly, we're protecting the clever things we do in software to reduce hardware costs. It's not impossible to figure out our methods but we aren't willing to just hand it over to our competitors. There is some other stuff in there to protect but that's the main one.**

http://permalink.gmane.org/gmane.comp.graphics.scanning.sane.devel/19722

# Why USB devices?

- basic USB protocol is well documented and easy to understand
- USB devices...
  - can be programmed from user space
  - can be reliably „passed through“ to an OS running inside a VM
- sniffing USB traffic is easy:
  - can be done in software
  - FOSS tools available

# Why not disassemble the driver?

- legal „gray area“
  - http://en.wikipedia.org/wiki/Reverse_engineering#Legality
- some open source projects do not accept code based on disassembling proprietary software
  - http://kerneltrap.org/node/6692
- code obfuscation, anti-debugging techniques
  - http://www.ossir.org/windows/supports/2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf
- traffic analysis may actually be less work

# USB fundamentals, part 1

- device provides 1 - 32 *endpoints* for communication – also called *pipes*
- transfer modes:
    - *isochronous*: guaranteed bandwidth, bounded latency, possible data loss
    - *interrupt*: bounded latency
    - *bulk*: no guarantees on bandwidth or latency
    - *control*: specific request/response message format

# USB fundamentals, part 2

- endpoints for control transfers are bi-directional, all others uni-directional
- every device supports control transfers on endpoint 0
- host can request a configuration descriptor that contains information on the other endpoints

# USB fundamentals, part 3

- structure of a USB control transfer:

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bmRequestType | 1 | Bit-Map | **D7 Data Phase Transfer Direction**<br>0 = Host to Device<br>1 = Device to Host<br>**D6..5 Type**<br>0 = Standard<br>1 = Class<br>2 = Vendor<br>3 = Reserved<br>**D4..0 Recipient**<br>0 = Device<br>1 = Interface<br>2 = Endpoint<br>3 = Other<br>4..31 = Reserved |
| 1 | bRequest | 1 | Value | Request |
| 2 | wValue | 2 | Value | Value |
| 4 | wIndex | 2 | Index or Offset | Index |
| 6 | wLength | 2 | Count | Number of bytes to transfer if there is a data phase |

http://www.beyondlogic.org/usbnutshell/usb1.shtml

# Sending USB requests

**Windows:**

prepare URB

```
UsbBuildVendorRequest(urb,
            Function,  // bits 0..5 of bmRequestType
            sizeof(struct URB_CONTROL_TRANSFER),
            TransferFlags,  // bit 7 of bmRequestType
            0,
            bRequest,
            wValue,
            wIndex,
            buffer, NULL, wLength, NULL);
status = CallUSBDI(dev, urb);
```

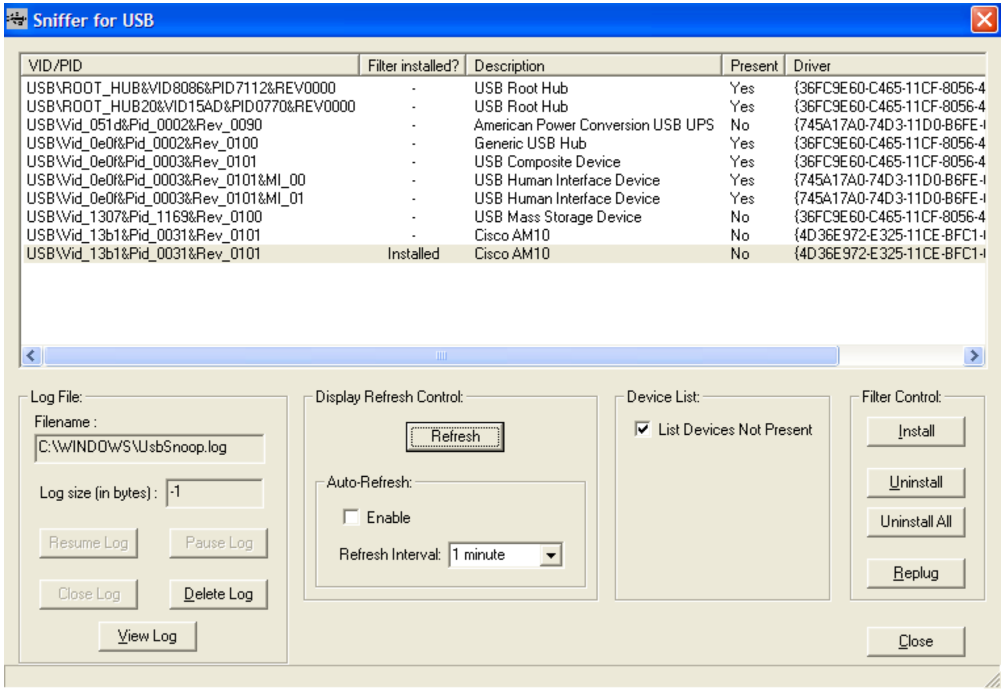
submit URB

**libusb:**

```
libusb_control_transfer (dev,
          bmRequestType,
          bRequest,
          wValue,
          wIndex,
          buffer, wLength,
          timeout);
```

# Sniffing USB traffic

- SniffUSB 2.0
  - http://www.pcausa.com/Utilities/UsbSnoop/
- installs a „filter“ between the device driver and the USB stack, writes all USB requests to disk



```
[1570 ms] UsbSnoop - FilterDispatchAny(8c37bfd2) : IRP_MJ_INTERNAL_DEVICE_CONTROL
[1570 ms] UsbSnoop - FdoHookDispatchInternalIoctl(8c37c1ea) :
          fdo=84432030, Irp=8458b4b8, IRQL=0
[1570 ms]  >>>  URB 36 going down  >>>
-- URB_FUNCTION_VENDOR_DEVICE:
  TransferFlags          = 00000000 (USBD_TRANSFER_DIRECTION_OUT, ~USBD_SHORT_TRANSFER_OK)
  TransferBufferLength = 00000001
  TransferBuffer       = 85c70df8
  TransferBufferMDL    = 00000000
    00000000: c0
  UrbLink                 = 00000000
  RequestTypeReservedBits = 00000000
  Request                 = 00000002
  Value                   = 00000000
  Index                   = 00000041
[1581 ms] UsbSnoop - MyInternalIOCTLCompletion(8c37c126) :
          fido=00000000, Irp=8458b4b8, Context=8462bd60, IRQL=2
[1581 ms]  <<<  URB 36 coming back  <<<
-- URB_FUNCTION_CONTROL_TRANSFER:
  PipeHandle           = 846451bc
  TransferFlags        = 0000000a (USBD_TRANSFER_DIRECTION_OUT, USBD_SHORT_TRANSFER_OK)
  TransferBufferLength = 00000001
  TransferBuffer       = 85c70df8
  TransferBufferMDL    = 84649ab8
  UrbLink              = 00000000
  SetupPacket          =
    00000000: 40 02 00 00 41 00 01 00
[1590 ms] UsbSnoop - FilterDispatchAny(8c37bfd2) : IRP_MJ_INTERNAL_DEVICE_CONTROL
[1590 ms] UsbSnoop - FdoHookDispatchInternalIoctl(8c37c1ea) :
          fdo=84432030, Irp=8458b4b8, IRQL=0
[1590 ms]  >>>  URB 37 going down  >>>
-- URB_FUNCTION_VENDOR_DEVICE:
  TransferFlags          = 00000000 (USBD_TRANSFER_DIRECTION_OUT, ~USBD_SHORT_TRANSFER_OK)
  TransferBufferLength = 00000001
  TransferBuffer       = 85c70df8
  TransferBufferMDL    = 00000000
    00000000: c1
```

Imagine 400 A4 pages of this stuff...

# Analyzing USB traffic

- start with simple operations over short periods of time
- write scripts to extract the interesting parts
- perform operations with different settings, compare traffic logs
- find patterns
- look at datasheets of similar chips / devices
- „replay“ traffic logs
    - http://lindi.iki.fi/lindi/darcs/usbsnoop2libusb/

# Case Study


## Aiptek Hyper VCam Mobile

Product ID: 0xa511
Vendor ID: 0x05a9 (OmniVision Technologies, Inc.)
Version: 1.00
Speed: Up to 12 Mb/sec
Location ID: 0x1a200000
Current Available (mA): 500
Current Required (mA): 500

## OmniVision OV511+

```
Event 6:
    level = 0x01
    req_type = URB_FUNCTION_VENDOR_DEVICE
    req_TransferFlags = 0
    req_TransferBufferLength = 0x01
    req_data =
3d
    req_Request = 0x02
    req_Value = 0
    req_Index = 0x50
    res_type = URB_FUNCTION_CONTROL_TRANSFER
    res_TransferFlags = 0x0a
    res_TransferBufferLength = 0x01
Event 7:
    level = 0x01
    req_type = URB_FUNCTIO
    req_TransferFlags = 0x
    req_TransferBufferLeng
    req_Request = 0x03
    req_Value = 0
    req_Index = 0x5f
    res_type = URB_FUNCTIO
    res_TransferFlags = 0x
    res_TransferBufferLeng
    res_data =
00
```

what you can't see:

- This is a log of plugging in the device and capturing a single frame.
- constant stream of control transfers as soon as the device is plugged in
- all of them single byte read/write requests as shown here

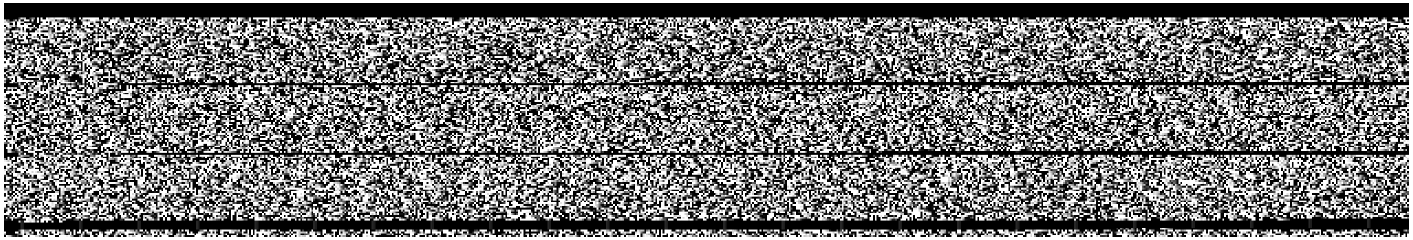
```
Event 6:
    level = 0x02
    OV511_command = write_register
    regIdx = 0x50
    value = 0x3d
Event 7:
    level = 0x02
    OV511_command = read_register
    regIdx = 0x5f
    value = 0
Event 8:
    level = 0x02
    OV511_command = write_register
    regIdx = 0x41
    value = 0xc0
Event 9:
    level = 0x02
    OV511_command = write_register
    regIdx = 0x44
    value = 0xc1
Event 10:
    level = 0x02
    OV511_command = write_register
    regIdx = 0x42
    value = 0x12
```

```
Event 348:
    level = 0x01
    req_type = URB_FUNCTION_ISOCH_TRANSFER
    req_endpoint = 0x51
    req_TransferFlags = 0x01
    req_TransferBufferLength = 0x00006020
    req_StartFrame = 0x00011b96
    req_NumberOfPackets = 0x20
    req_IsoPacket[0].Offset = 0
    req_IsoPacket[0].Length = 0
    req_IsoPacket[1].Offset = 769
    req_IsoPacket[1].Length = 0

     ...

    res_type = URB_FUNCTION_ISOCH_TRANSFER
    res_endpoint = 0x51
    res_TransferFlags = 0x01
    res_TransferBufferLength = 0x00006020
    res_StartFrame = 0x00011b76
    res_NumberOfPackets = 0x20
    res_ErrorCount = 0
    res_IsoPacket[0].Offset = 0
    res_IsoPacket[0].Length = 769
    res_IsoPacket[0].Status = 0
    res_IsoPacket[0].Status_data =
00 00 00 00 00 00 00 00 59 c8 f4 80 3e 80 01 02
01 01 02 42 1e c0 ff fe c7 f9 51 0d 01 fc 02 fe
```

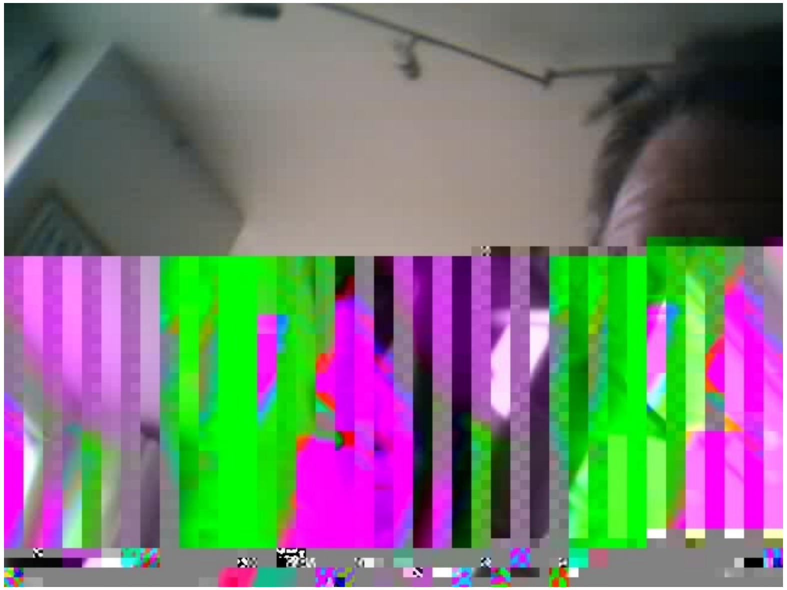
```
memcpy(buf, "\x3d", 0x0000001);
ret = libusb_control_transfer(devh,
   LIBUSB_REQUEST_TYPE_VENDOR | LIBUSB_RECIPIENT_DEVICE,
   0x0000002, 0x0000000, 0x0000050, buf, 0x0000001, 1000);
usleep(111*1000);

ret = libusb_control_transfer(devh,
   LIBUSB_REQUEST_TYPE_VENDOR | LIBUSB_RECIPIENT_DEVICE |
   LIBUSB_ENDPOINT_IN,
   0x0000003, 0x0000000, 0x000005f, buf, 0x0000001, 1000);
printf("control msg returned %d, bytes: ", ret);
print_bytes(buf, ret);
printf("\n");
usleep(39*1000);
```

```
stream = fopen ("frame.ppm", "wb");
snprintf (header, sizeof (header), "P5\n%d %d\n255\n", 768, 128);
fwrite (header, 1, strlen (header), stream);

for (i = 0; i < 4; i++)
{
  iso_trans = libusb_alloc_transfer(32);
  libusb_fill_iso_transfer(iso_trans, devh, 0x00000081, isobuf,
      32 * 769, 32, iso_callback, NULL, 1000);
  libusb_set_iso_packet_lengths(iso_trans, 769);
  ret = libusb_submit_transfer(iso_trans);
  while (!iso_transfer_finished)
    libusb_handle_events(ctx);
  iso_transfer_finished = 0;

  for (j = 0; j < iso_trans->num_iso_packets; j++)
  {
    offset = libusb_get_iso_packet_buffer_simple(iso_trans, j);
    fwrite(offset, 1, 768, stream);
    printf("0x%x\n", offset[768]);
  }

  libusb_free_transfer(iso_trans);
}

fclose (stream);
```

**result:**



**Ubuntu 10.10:**



In this case, we can „cheat“ by reading the manual (prepare for the worst):

http://mxhaard.free.fr/spca50x/Doc/Omnivision/ds_511P.pdf