USBiquitous: USB intrusion toolkit

Benoît Camredon
benoit.camredon@airbus.com

Airbus Group

Abstract. The USBiquitous project is a set of open source tools to interact with USB communications. It is composed of a hardware part embedding a Linux system with a bespoke kernel module, and a set of userland scripts and libraries, each designed to tackle a specific problem linked to USB communications. Emulating a USB host, device, or simply performing a man in the middle attack between a host and a device can then be done with a few lines of code in a userland script.

1 Introduction

1.1 Context

USB devices are everywhere: keyboards, mice, USB keys, webcams, WiFi adapters, phones... and USB interfaces are appearing on every equipment of our every day life [9]. The situation is not going to change with the Internet Of Things (IOT) development.

In industrial settings, USB interfaces are already widespread. Every new car already has one or more USB plugs, that can be used either to get electrical power to recharge our phones, iPods... or simply to play music. In can also be used to retrieve logs or to update firmware. USB is an all-purpose interface and even the most change-averse individuals are forced to adopt this standard.

Naturally, this interface is a security attack target and more and more tools exist to assess the robustness of systems using this interface [2, 5], that make it mandatory to improve our USB tools to protect systems against this not new, but nevertheless growing attack vector.

To conduct audits on systems having USB interfaces, we needed to have a good understanding of this protocol, and the low level layers of the Linux kernel that implement it. On this journey to understanding the USB protocol, we have developed the USBiquitous framework, as a means to experiment while learning.

This framework has accumulated enough useful features to become an effective tool during audits of systems that include a USB interface.

1.2 Framework design

More than a tool, USBiquitous (USBq) is a framework that can be easily adapted to suit several needs.

It is composed of four parts:

- A hardware part, named *USBq Board*, that has several USB interfaces, and runs a Linux operating system
- A Linux kernel module, named *USBq Core*, running on the *USBq Board*
- A userland library, which provides users with the framework of the *USBq API*, to be used to write applications
- Userland applications, named *USBq Apps*, which perform a specific task, using the *USBq API*

The *USBq Board* can be:

- connected to a host and act as a USB device
- connected to a device and act as a USB host
- connected between a host and a device and act as a USB proxy

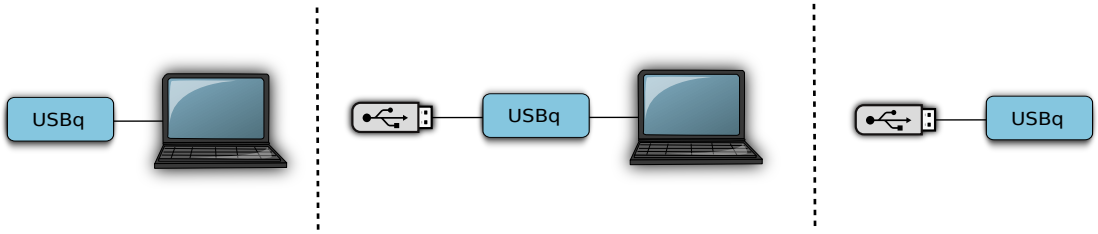

Fig. 1. USBq modes

Used together, the *USBq Core* and *USBq Board* forward USB communications in both directions between a USB entity (host or device), and an *USBq App*, through the userland *USBq API*, where that can conveniently be manipulated (see figure 2).

The task of the *USBq APP* depends on the problem to tackle. In device emulation it could be to assess the robustness of the targeted host USB stack, or simply to emulate a keyboard acting as a USB rubber ducky [12]. In proxy mode, it could be to perform USB communications recording, to allow offline investigations, or to apply mutations to fuzz a USB driver. In host mode, it could be to analyze responses from the USB device to fingerprint its USB stack.

The userland part is free to focus on the specific problem it is intended to resolve while abstracting away low level USB details.

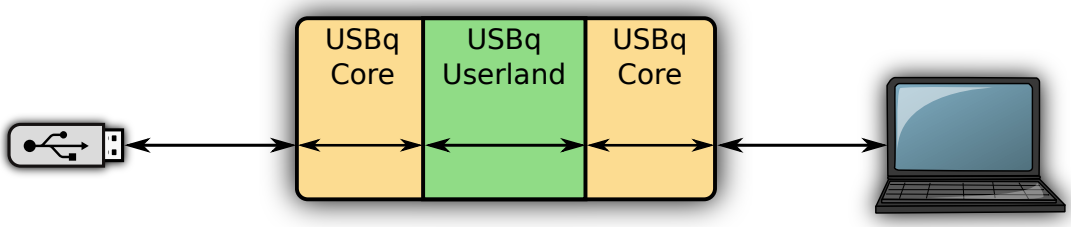

Fig. 2. USBq design

1.3 State of the art

Playing with USB communications is not something new and nowadays several opensource tools are already available.

Facedancer The facedancer [2–4] is the most famous one. It is a bespoke USB hardware device that can be configured to perform USB device emulation or USB host emulation, but not both at the same time. It is a very powerful tool as it allows implementing very easily a host or a device using provided python libraries.

The `umap` [16] tool, is based on the facedancer and can be used to fuzz USB stacks or detect which kind of USB devices can be handled by the host.

Because the facedancer cannot be used as a device and a host at the same time, it is not possible to use it as a *Man in the middle* tool. However, there is a project [14, 15] based on two facedancers to perform a MITM, but it has many performance issues.

USBSniffer `USBSniffer` [20] is the closest project to USBq in terms of design, but not in terms of objectives. It was developed in 2010 for the Google summer of code, but it is now inactive.

`USBSniffer` is a Linux kernel module running on a *BeagleBone black* [1] that implements a USB driver and emulates a USB device using the *GadgetAPI* [5]. Both components are used to forward USB communications from a device to a host. Then `usbmon` [19] and `tcpdump` are used to generate a PCAP file of the USB communication.

USBq tries to go further and isn't limited to sniffing only.

USBProxy `USBProxy` is a C++ userland program running on a *BeagleBone black* [1] and using `GadgetFS` [6] to emulate a device and the

`libusb` [8] to communicate with a USB device. It is possible to add plugins to interact with USB communications.

2 Background

This part outlines USB standard concepts and can be skipped, if they are already known. It is based on several sources [21, 22].

USB (Universal Serial Bus) is a protocol defining how a host and a device shall communicate. It was designed to standardize the connection between a computer and its peripherals, and to replace all older interfaces such as serial or parallel ports.

The first USB standard 0.8 was published in 1994. The latest is USB 3.1 issued in 2013. However, even if USB3 devices are prevalent, lots of USB2.0 devices are still in use. Each standard defines (among other things) the maximum theoretical speed of the communication. For now, USBq is limited to USB2.0 standard.

2.1 Tiered-star topology

The architecture of USB consists of a host and a multitude of downstream USB ports, and multiple peripheral devices connected in a tiered-star topology. The host controller contains the root hub, on which several devices can be connected, including other USB hubs (see figure 3).

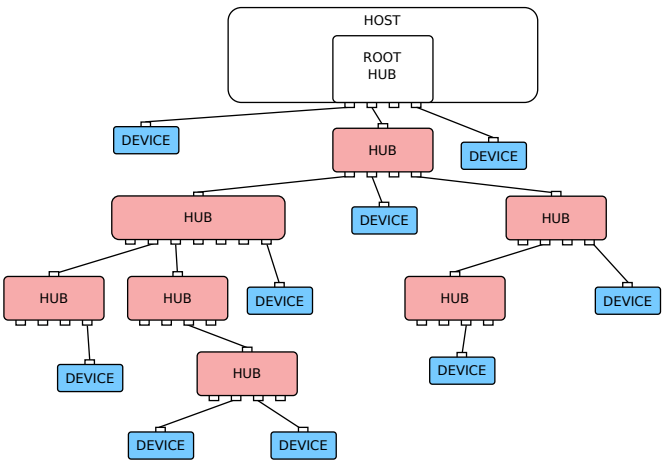

Fig. 3. USB Topology

Up to five USB hubs can be linked in series to handle a maximum of 127 devices.

2.2 USB communication

USB communications always happen between a host and a device[1]. The host manages traffic on the bus, and the device responds to requests from the host.

In order to communicate, hosts and devices use unidirectional pipes, called endpoints, to send or receive data. The host sends data to the device using an `OUT` endpoint, while it receives data using an `IN` endpoint (see figure 4).

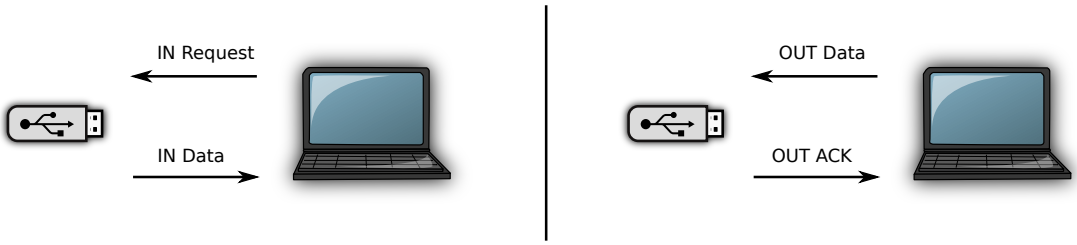

Fig. 4. IN/OUT communication

Each device has a list of endpoints that it is able to use. Each has a set characteristics that are communicated to the host using an endpoint descriptor structure.

The USB communication is done in two main steps: the enumeration process which is used to determine USB devices capabilities, and the data exchange.

2.3 Transfer Type

Because several devices with completely different purposes and requirements can be connected to a host, four types of data transfer have been defined:

– *Control Transfers* commonly used for command and status operations. For example, they are involved in the enumeration phase, allowing the host to learn about device capabilities. All devices must be able to handle this type of data transfer

[1] There is not device to device communication

– *Interrupt transfers* allow sending short messages, with a guaranteed latency and error detection. It is commonly used with HID USB devices, such as keyboards and mice
– *Bulk transfers* allow sending or receiving large messages, handling error detection but without any latency guarantees. They are commonly used in mass-storage USB devices
– *Isochronous transfers* used for messages requiring bounded latency, but no guarantee of delivery. They are commonly used for webcam and audio USB devices

The transfer type is part of attributes of an endpoint, and as such is included in the endpoint descriptor.

2.4 USB Descriptors

The USB standard allows several completely different devices to connect to a host through an identical protocol and wiring. Because several peripherals can be connected on a same port, sharing the same interface, protocol shall define a learning phase in order to discover its functionalities, called the enumeration process.

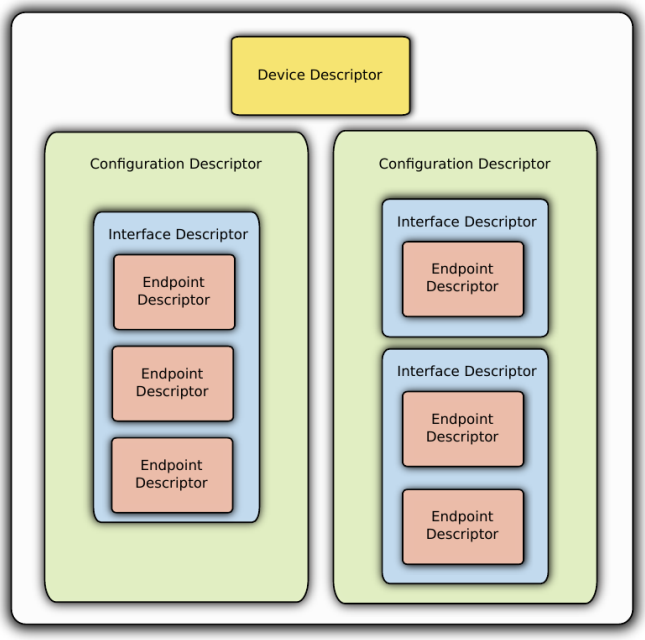

Fig. 5. Main USB Descriptors

Device descriptor Each USB device has one device descriptor, comprising information such as its product ID, vendor ID, class ID, the number of configurations... The operating system uses the information in this descriptor (among other things) to load to most appropriate driver.

```
struct usb_device_descriptor {
        __u8    bLength;              // Size of the Descriptor in Bytes (18 bytes)
        __u8    bDescriptorType;      // Device Descriptor (0x01)
        __le16  bcdUSB;               // USB Specification Number which device complies too.
        __u8    bDeviceClass;         // Class Code (Assigned by usb.org)
        __u8    bDeviceSubClass;      // Subclass Code (Assigned by usb.org)
        __u8    bDeviceProtocol;      // Protocol Code (Assigned by usb.org)
        __u8    bMaxPacketSize0;      // Maximum Packet Size for Zero Endpoint
        __le16  idVendor;             // Vendor ID
        __le16  idProduct;            // Product ID (Assigned by Manufacturer)
        __le16  bcdDevice;            // Device Release Number
        __u8    iManufacturer;        // Index of Manufacturer String Descriptor
        __u8    iProduct;             // Index of Product String Descriptor
        __u8    iSerialNumber;        // Index of Serial Number String Descriptor
        __u8    bNumConfigurations; // Number of Possible Configurations
} __attribute__ ((packed));
```

Listing 1. Device Descriptor

Configuration descriptor Each USB device can have one or more configurations depending on its functionnalities. However, only one can be active at the same time. For example, a USB device can have a mode where it can be upgraded, and a mode in which it provides its standard functionality. It is up to the operating system to choose the configuration it wants to enable. Most USB devices only have one configuration.

```
struct usb_config_descriptor {
        __u8    bLength;              // Size of Descriptor in Bytes
        __u8    bDescriptorType;      // Configuration Descriptor (0x02)
        __le16  wTotalLength;         // Total length in bytes of data returned
        __u8    bNumInterfaces;       // Number of Interfaces
        __u8    bConfigurationValue; // Value to use as an argument to select this
              configuration
        __u8    iConfiguration;       // Index of String Descriptor describing this
              configuration
        __u8    bmAttributes;         // self powered, remote wake up...
        __u8    bMaxPower;            // Maximum Power Consumption in 2mA units
} __attribute__ ((packed));
```

Listing 2. Configuration Descriptor

SetConfiguration The host requests the device to use a specific configuration using the `CTRL OUT` message `SetConfiguration`.

Interface descriptor For each configuration descriptor, one or more interface descriptors can exist. Each interface descriptor represents a device feature. For example in a webcam, an interface can represent the video feature, while another will represent the audio one.

```
struct usb_interface_descriptor {
        __u8  bLength;            // Size of Descriptor in Bytes (9 Bytes)
        __u8  bDescriptorType;    // Interface Descriptor (0x04)
        __u8  bInterfaceNumber;   // Number of Interface
        __u8  bAlternateSetting;  // Value used to select alternative setting
        __u8  bNumEndpoints;      // Number of Endpoints used for this interface
        __u8  bInterfaceClass;    // Class Code
        __u8  bInterfaceSubClass; // Subclass Code
        __u8  bInterfaceProtocol; // Protocol Code
        __u8  iInterface;         // Index of String Descriptor Describing this interface
} __attribute__ ((packed));
```

Listing 3. Interface Descriptor

Each interface can have several alternate settings, all describing the same function. These settings are mutually exclusive, only one is active at a time. Each setting has an interface descriptor and subordinate descriptors as needed. For instance, devices that use isochronous transfers can have alternate interface settings to be able to use more or less bandwidth.

SetInterface For devices that use several alternate interface settings, this `CTRL OUT` message allows the host to choose a specific one.

Endpoint descriptor For each interface descriptor, one or more endpoint descriptors can be used. Each endpoint will be used for the data transfer between the host and the device.

The following information can be found in this descriptor

- ID of the endpoint
- Direction of the endpoint (is it used to send data from the host, or to receive data from the host)
- Endpoint attributes including the kind of communication used (control, interrupt, bulk or isochronous)
- Maximum message size used by the endpoint
- Polling interval of transfers

```
struct usb_endpoint_descriptor {
        __u8    bLength;          // Size of Descriptor in Bytes (7 bytes)
        __u8    bDescriptorType;  // Endpoint Descriptor (0x05)
        __u8    bEndpointAddress; // Endpoint Address
        __u8    bmAttributes;     // Transfer, synchronization, usage type
        __le16  wMaxPacketSize;   // Maximum Packet Size
        __u8    bInterval;        // Interval for polling endpoint data transfers
} __attribute__ ((packed));
```

Listing 4. Endpoint Descriptor

String descriptor String descriptors are optional and are used to provide human readable information about the device, such as its name or information about its manufacturer.

```
struct usb_string_descriptor {
        __u8   bLength;         // Size of Descriptor in Bytes
        __u8   bDescriptorType; // String Descriptor (0x03)
        __le16 wData[1];        // Supported language
} __attribute__ ((packed));
```

Listing 5. String Descriptor

Other descriptors Other descriptors can exist that are class specific or vendor specific.

3 USBq Board

The *USBq Board* is the hardware component of the USBq Framework. It runs Linux, and includes the *USBq Core* linux module.

It must have several USB ports available, including one port that can be used in USB Host mode, in order to communicate with a USB device, and one port that can be used in USB device or OTG [18] (On-The-Go) mode, in order to communicate with a USB host.

Originally, the development of USBq started on a *IGEPv2* board [7] and was continued on a *BeagleBone Black* [1], because of its more active community. From a USBq point of view, there should be no difference between these boards, or others that provide features described earlier in this section.

4 USBq Core

4.1 Design

The *USBq core* is a Linux 4.1 kernel module running on top of GNU/Linux OS.

The general architecture described in the figure 6 is developed below.

Software The USBq core is composed of two elements:

- A driver part in charge of the communication with USB devices called *USBq driver*
- A gadget part in charge of the USB device emulation called *USBq gadget*

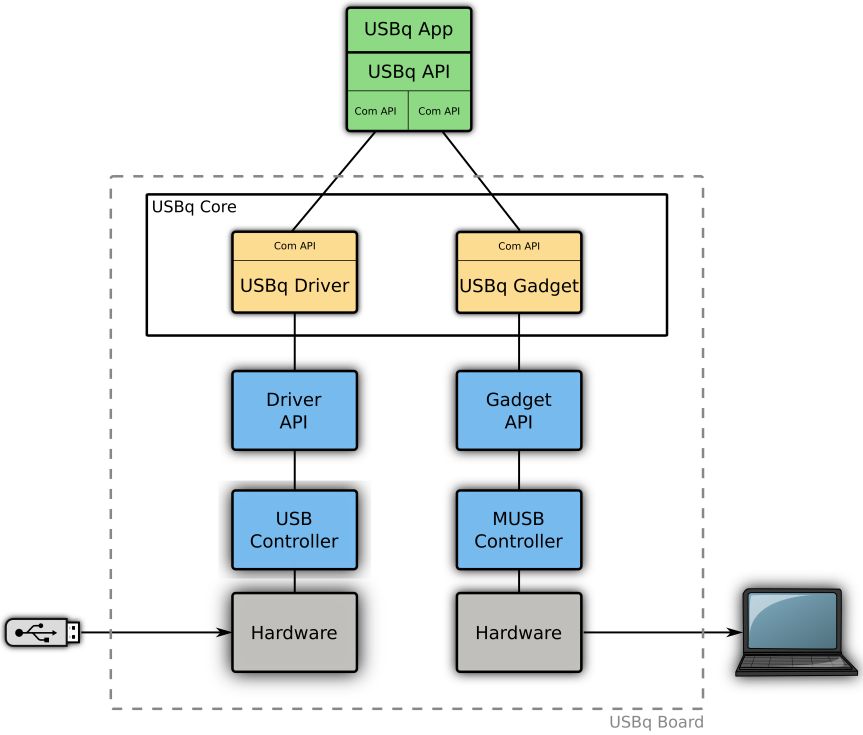

Fig. 6. USBq general design

While these parts are located in the same kernel module, they have no direct link and could be split. For the rest of the article, they will be considered as distinct modules.

This segmentation design objective allows using:

- only the driver part, and simulate a host in userland
- only the gadget part, and simulate a device in userland
- both parts, and act in MITM mode

4.2 Internals design

USBq core uses the USB driver API in order to handle communications with USB devices, acting as a USB Host, and uses the *GadgetAPI* in order to emulate a USB device. While these two API are different, our design of both elements shares the same base: an implementation driven by endpoints.

Endpoints API As described in the previous section, USB devices expose one or more endpoints through which communications are performed. The Endpoints API of the *USBq Core* kernel module exposes a consistent

interface to represent endpoints of a device, abstracting differences in the kernel's Gadget and Device APIs.

Within the *USBq Core* kernel module, every endpoint is represented as a structure composed of several functions pointers:

- `send_usb`: Used to send a message to the USB controller
- `recv_usb`: Called by the kernel (Gadget or Device API) when a USB message is received from the USB controller
- `send_userland`: Used to send a message to the userland program
- `recv_userland`: Called by the kernel when a USB message is received from the userland program

Whatever the kind of USB endpoints involved in the communication (control, interrupt, bulk or isochronous), their corresponding structures are composed of these main functions, in both driver and gadget parts. This endpoint structure is the USBq vision of the real endpoint device.

A very simplified example of an endpoint communication is described below:

```
// Called when a USB message comes from USB controller
int recv_usb(struct ep_t *ep, msg_t *msg) {
  return ep->send_userland(msg);
}

// Called when a USB message comes from userland
int recv_userland(struct ep_t *ep, msg_t *msg) {
  return ep->send_usb(msg);
}
```

Listing 6. Endpoint Algorithms

Initialization USB device endpoints characteristics are exchanged during the enumeration phase in the configuration descriptor of the device. This descriptor is parsed by the Linux kernel and forwarded by the *USBq driver* to to the *USBq gadget* with a `NEW_DEVICE` message (see 4.2). This descriptor is needed to create endpoint structures at USBq level.

During USB communications the host can choose to enable or disable a set of endpoints of a specific interface, in order for instance to increase or decrease bandwidth allocated to the device. USB messages linked to this functionality have to be intercepted to reflect the list of USBq enabled endpoints:

- `SetConfiguration`, allows enabling a set of interfaces
- `SetInterface`, allows enabling set of endpoints (and therefore disable others)

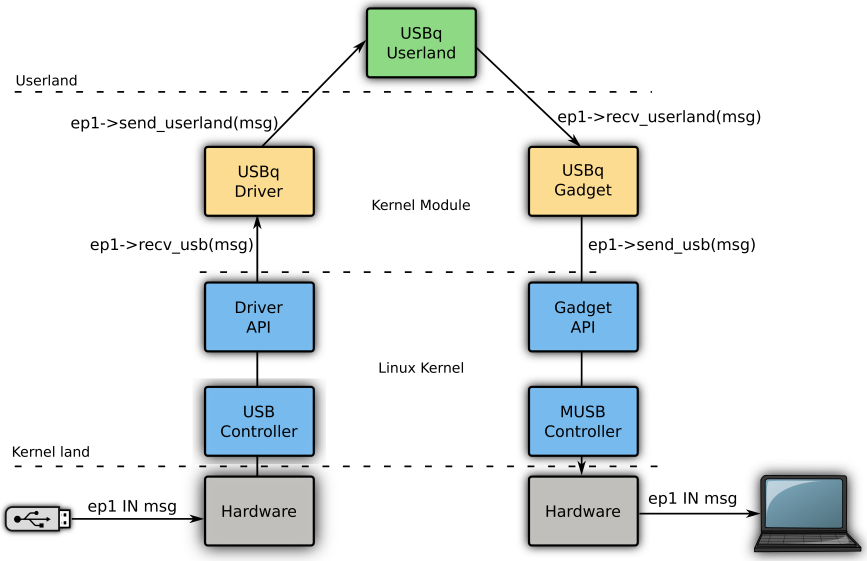

Fig. 7. Endpoint Communication

Depending on the module side (the gadget or the driver), endpoints `OUT` or `IN` respectively will be requested to send data during initialization, while others will wait for userland communication.

USB reception USB messages reception is in general done in a completion handler that has to be run in an uninterruptible context. Because userland communication is network based, it is not possible to use it in that kind of context. So, the completion handler uses a workqueue to dispatch processing of incoming data to a worker thread, which communicates with *USBq API*.

USBq protocol Communication between *USBq Core* kernel module and the *USBq API* (userland) are performed using a protocol that is specific to the USBq framework. This protocol is used to transfer USB communications but also to send meta-data.

Meta-data Three kinds of meta-data messages are exchanged between the core and the userland. The first one is a `NEW_DEVICE` message used to inform the gadget part that a new device has been connected. This message can be sent either by the device driver when it detects a device connection, or by a userland script to emulate a new device connection. It embeds information about the connected device, similar to the ones located in a full configuration descriptor message. It is necessary for the core to know which configurations, interfaces and endpoints the device is

composed of. Using meta-data to send features of a device allows fuzzing of the real USB message embedding this kind of information.

The second message is a `RESET` message used to inform the gadget part of the device disconnection.

Finally every USB data message is acknowledged using an `ACK` message.

USB data Every USB message exchanged between a host and a device, is encapsulated in a USBq message specifying the endpoint used by this message. The message content is then embedded without any modification.

4.3 Gadget module

The gadget module emulates a USB device using the Linux *GadgetAPI* [5].

A gadget module has to register with this API using the kernel `usb_gadget_probe_driver` function taking a `usb_gadget_driver` structure. This parameter is mainly composed of function pointers:

- `bind`: called to start the USB device emulation
- `setup`: called when a control message is received from the host

Thus those functions are implemented by *USBq gadget* to interact with USB communications for the device emulation.

Setup function Because the USB communication is driven by the host, the `setup` function is called as soon as a control request is received. Those requests can be:

- `GetDescriptor` to request a specific descriptor such as device, configuration, or string descriptor
- `SetConfiguration` to enable a specific device configuration
- `SetInterface` to activate a specific alternative interface
- ...

Some of those requests ask for specific device information, such as `GetDescriptor` and are `IN` requests, others are used to configure the USB device such as `SetConfiguration` or `SetInterface` and are `OUT` requests.

Sometimes, some USB control requests need to carry data that is not embedded directly during the `setup` called. An additional request is then necessary to retrieve the missing data.

The `setup` function is called in an uninterruptible kernel context, imposing the use of a workqueue in order to perform network exchanges.

USB communication At the gadget level, the USB communication is done with the kernel `usb_request` structure that has to be filled depending on the endpoint transfer type, either for sending data (`IN`) or receiving data (`OUT`). Sending is done with the kernel `usb_ep_queue` function, and reception is done by a callback specified in the `usb_request`. This function is wrapped inside the USBq `send_usb` function (see 4.2) for gadget relative USBq endpoints.

MUSB The kernel *GadgetAPI* is implemented on the top of MUSB which is a USB controller provided by the Linux kernel. It is is responsible for the interface with the USB OTG hardware.

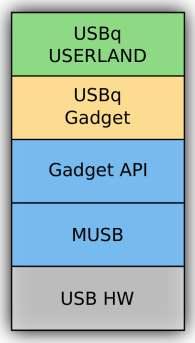

Fig. 8. Architecture

While the *GadgetAPI* allows to easily develop USB devices, it suffers from some limitations and constraints linked to the use of MUSB. These are described below.

Control request The `setup` (described in 4.3) is called when a control request has been sent by the host to the emulated device. Unfortunately, MUSB does not forward all those requests to the *GadgetAPI* (and thus to the gadget driver). For instance `SET_ADDRESS`, `CLEAR_FEATURE`, `SET_FEATURE` requests are not forwarded.

IN requests from a non control endpoint For non control endpoint, there is no function such as `setup`. So, from a USB device point of view, it is not possible to know exactly when a host is ready to receive information. The device has first to send its data, which will be consumed whenever the host is ready to do so. Unfortunately, these pending `IN` requests can have an impact on the way USBq works as will be described later (see 4.4).

Endpoint statically defined In a USB device, endpoint characteristics such as transfer type, address, and direction are fixed inside the hardware and cannot be changed. In the same way, MUSB has a statically defined set of endpoints structure with their address, direction and maximum packet size.

```
/* mode 2 - fits in 4KB */
static struct musb_fifo_cfg mode_2_cfg[] = {
{ .hw_ep_num = 1, .style = FIFO_TX,    .maxpacket = 512, },
{ .hw_ep_num = 1, .style = FIFO_RX,    .maxpacket = 512, },
{ .hw_ep_num = 2, .style = FIFO_TX,    .maxpacket = 512, },
{ .hw_ep_num = 2, .style = FIFO_RX,    .maxpacket = 512, },
{ .hw_ep_num = 3, .style = FIFO_RXTX,  .maxpacket = 256, },
{ .hw_ep_num = 4, .style = FIFO_RXTX,  .maxpacket = 256, },
};
```

Listing 7. MUSB Endpoints table

When a device is implemented using the *GadgetAPI*, endpoints that it can use must exist in this table.

For instance, if a gadget driver needs an `IN` endpoint 3 with packet size of 512 bytes and the MUSB endpoint 3 is a IN endpoint with a packet size of 256 bytes, MUSB will automatically adjust the endpoint address and assign the first available endpoint that meets the required characteristics.

In a classical device development, this address modification is not a problem. However, if the emulated USB device has to match a real device hardware (because it will forward the communication to this device for instance), it becomes a problem. It still possible to use a translation table only if the translated address is not referenced in the communication data of another endpoint...

Gadget specificity The gadget module will first initialize its communication module and wait for the initialization packet[2] describing the device that it has to emulate. As soon as this packet is received, it builds its endpoints list and starts the USB device emulation.

Descriptors requests will be received through its `setup` function, and forwarded to userland. Because the `setup` function is executed in a kernel uninterrupted context, a workqueue will handle the requests forwarding.

If `SET_CONFIGURATION` or `SET_INTERFACE` requests are received, the gadget module will enable the corresponding interfaces and will activate their endpoints. *Non control* `OUT` endpoints will then be requested, in order to be able to receive data from host. *Non control* `IN` endpoints are data coming from the device and then will be received from the userland, before being forwarded.

[2] `NEW_DEVICE` packet

As previously explained in 4.3, requested endpoints addresses can be modified by MUSB. To avoid this modification, the original kernel function[3] used to match endpoints has been rewritten. Nevertheless, there is a risk that no corresponding endpoint can be found. To reduce the risk the MUSB endpoints table (see listing 7) has been modified to be more generic. If no endpoint is found, it will not be possible to emulate the USB device.

4.4 Driver Module

The driver module is responsible for the communication with USB devices that will be plugged on the *USBq board*. Its task is to forward USB communications to userland, and userland communications to the USB peripheral.

USB host controller registration To handle USB devices, a Linux USB driver needs to register to the Linux USB host controller. This registration is done by calling the kernel `usb_register` function taking a `usb_driver` parameter, mainly composed of:

- The name of the driver
- A `probe` function: called when a new corresponding device is plugged on the board
- A `disconnect` function: called when the device is disconnected
- A `usb_device_id` table, used to determine what kind of devices can be handled by the driver

The `usb_device_id` table defined by *USBq driver*, allows matching USB devices by their characteristics such as:

- Their `ProductID`/`VendorID`
- Their USB class (HID, MassStorage...)
- ...

It is also possible to match all USB devices choosing a non-zero value in the `driver_info` attribute, which is what our driver does.

```
/* table of devices that work with this driver */
static struct usb_device_id driver_table [] = {
    { .driver_info = 64},
    {} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, driver_table);
```

Listing 8. Matching all USB devices

[3] `usb_ep_autoconfig`

Probe function The `probe` function of the driver is called for every USB device interface because an interface describes a device function such as audio or video. Some devices can have several interfaces and therefore be handled by several drivers. Our device module must handle all interfaces of all USB devices.

The `probe` function takes a parameter : the `usb_interface` structure that it may handle. This structure contains all the information relative to endpoints contained in this interface. When the function is called, the USB host controller has already exchanged information (especially configuration descriptor) with the USB device, to know which driver to load. It is up to the driver to perform again this exchange.

USB communication At driver level, the USB communication is done with the `URB` (USB Request Block) structure [17]. It has to be filled differently depending on the endpoint nature (control, bulk, interrupt or isochronous) and direction (`IN` or `OUT`). Sending is done with `usb_submit_urb`, and receiving is handled by a callback specified in the `URB` structure.

Driver specificity Some limitations of the gadget part (see 4.3) have consequences on the driver development. Indeed, because `IN` requests are not received by the gadget part, the *USBq driver* needs to send them to the USB peripheral, in order to allow it to send information[4]. The drawback of this approach is that USBq may query a device for data when the host has not yet done so.

When the driver detects that a new device is connected (its `probe` function is called), it sends device features to userland with a `NEW_DEVICE` packet. Whereas when it detects a device unplugged (its `disconnect` function is called) it sends a `RESET` packet.

4.5 Communication with userland

Every USB message coming from the device or from the host will first go through its corresponding driver, then will be forwarded to the userland, then will go through the other driver. Userland is then a mandatory crossing point between the two parts of the *USBq core*.

The communication mechanism between the kernel and userland programs has been abstracted to be easily upgradable in the future. During

[4] USB communication are driven by the host

the development of USBq, userland programs have been developed outside the *BeagleBone black* on a classical PC. A network communication channel between *USBq core* and userland was implemented to simplify testing.

The *USBq device* driver is bound on an UDP port and waits for data. The gadget driver component uses an UDP socket to send its messages coming from the host to the userland.

4.6 Limitations

The *USBq core* has several limitations. The first one is due to lack of time: isochronous packets are not yet handled, preventing the use of many USB devices such as webcams, or audio cards.

As was previously mentioned, there are many constraints linked to MUSB implementation choices.

- Some requests are not forwarded to the *GadgetAPI*, and thus cannot be handled. If a device uses them, it will probably not work.
- MUSB does not send `IN` requests for non control endpoints... so USBq has to generate them and can possibly do so with the wrong timing. Although no problems related to this issue were encountered yet, it is not entirely unlikely that this could impact certain USB devices.
- MUSB has a predefined set of statically defined endpoints. Even if with the modifications made to MUSB (see 4.3) to make endpoint allocation more flexible, there subsists a risk that the requirements of some USB devices will not be met and prevent them from working.

Finally, the design decision to handle USB communications in userland and the associated overhead could interfere with the proper functioning of devices that depend on tight timing constraints.

5 USBq userland

While the core kernel module is written in C, the userland part used to solve the problem can be written in any language. It was an objective of this project to easily develop, adapt or change the logic part of USBq, without any needs of module recompilation. The userland part is completely separate from the core part.

Usually the userland part will receive messages from one part of *USBq core*, will process and forward them to the other part. However, because there is no direct link between both parts, it is also possible to communicate with only one of either the driver or gadget sides. In this case, userland programs can completely emulate either a device or a host.

The userland part does not need to run on the same hardware as the core (although that remains a possibility). It is then possible to develop network based userland script, to implement network USB devices.

USBq API The USBq userland component referred to as the *USBq API* provides a unified framework to implement a proxy, device or host. It is a set of Python classes used to:

- Dissect, create and modify USB descriptors based on bespoke `scapy` classes [13]
- Handle communication between userland scripts and the kernel core
- Provide skeletons to implement hosts, devices and proxies

Several userland programs (*USBq APP*) have been developed to fit general needs, based on this API.

5.1 Proxy

Proxy programs are *USBq APP* that make the link between the driver and the gadget module, in other words between a USB device and a USB host. They receive messages from one side and forward them to the other side. They are used as MITM programs and can act on the communication to:

- Forward it
- Modify it
- Store it
- Block it
- ...

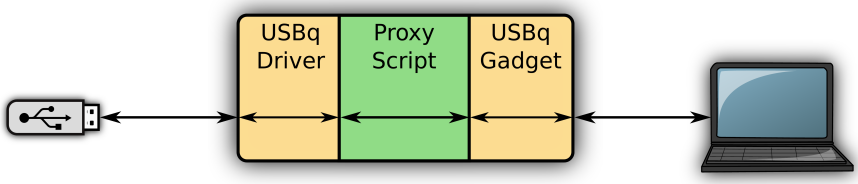

Fig. 9. Proxy Emulation

All proxy programs have a common base allowing them to receive and send data to and from the *USBq core*. Only a few hooks are necessary to interact with USB messages.

```
def hookDevice(self,data):
  """ Called each time a device message is received """
  return data

def hookHost(self,data):
  """ Called each time a host message is received """
  return data
```

Listing 9. Proxy API

Because all proxies use a common communication interface with the *USBq core*, they can be chained to provide several functionnalities.

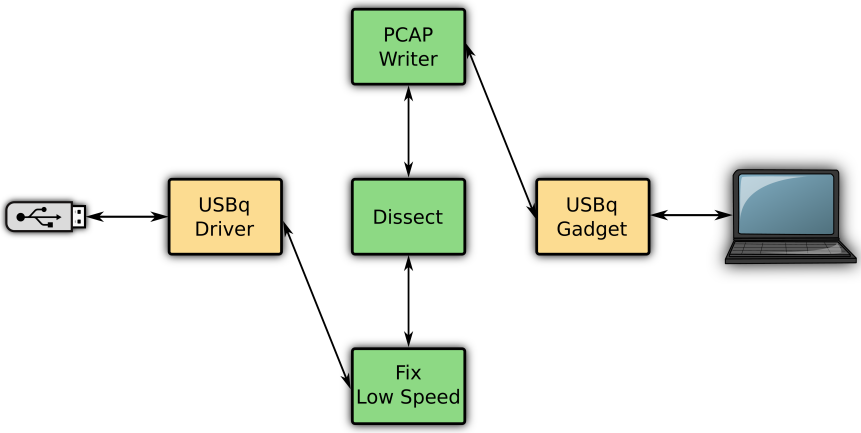

Fig. 10. Chained Proxy

Dissect This program is used to inspect packets' content and forward them without any modification. The output can be configured to provide more or less information:

- Display management packets or not
- Dissect USB descriptors
- Display hexadecimal payload

It is mainly used to understand or debug a USB communication, either between a real USB device and host, or between an emulated device and a host. The example 10 is the protocol dissection of a USB device using an unknown protocol.

FixLowSpeed The *BeagleBone black* used is a high speed board. There are some problems when a low speed device is forwarded through a

```
# ./dissect.py --server-ip beagle --dissect
< Ci0: GetDescriptor device [sz:18]
> Ci0: Device Descriptor vid:1130 pid:202 maxpkt:8 len:18
< Ci0: GetDescriptor configuration [sz:9]
> Ci0: Configuration Descriptor nintf:2
< Ci0: GetDescriptor configuration [sz:59]
> Ci0: Configuration Descriptor nintf:2
    Interface Descriptor ifnum:0 alt:0 class:hid nep:1
    HIDDescriptor
    Endpoint Descriptor EP1IN  Interrupt int:1 pkt:8 len:7
    Interface Descriptor ifnum:1 alt:0 class:hid nep:1
    HIDDescriptor
    Endpoint Descriptor EP2IN  Interrupt int:1 pkt:8 len:7
< Co0: SetConfiguration 1
< Ci0: GetDescriptor string [sz:255]
> Ci0: String Descriptor [Tenx Nonstandard Devic] len:46
< Co0: SetIDLE
< Ci0: GetDescriptor HID REPORT [sz:107]
> Ci0: HIDReportDescriptor
< Co0: SetIDLE
< Ci0: GetDescriptor HID REPORT [sz:87]
> Ci0: HIDReportDescriptor
< Co0: data: 'USBC\x00\x00\x04\x00'
< Co0: data: 'USBC\x00@\x02\x00'
< Co0: data: '\x00\x00\x01\x00\x00\x00\x08\x08\x00\x00\x00\x00'
< Co0: data: 'USBC\x00\x00\x04\x00'
< Co0: data: 'USBC\x00@\x02\x00'
```

Listing 10. Missile protocol dissection

high speed device, because from a hardware point of view, the device is presented to the host as a high speed device, but its descriptors describe a low speed one.

First the `bInterval` attribute of endpoint descriptors is used to specify the polling interval of certain transfers. The units are expressed in frames, thus this equates to either 1ms for low/full speed devices and 125μs for high speed devices. This bad interpretation results in a very high latency for a mouse for instance.

Then the `bMaxPacketSize` attribute of device descriptor has to be 64 for a high speed device, but in case of a low speed device forwarding this value is equal to 8. This difference does not have any impact on a Linux targeted host, but the device is simply not recognized on a Windows one.

The `FixLowSpeed` userland program fixes both values to be well interpreted by the targeted host.

The decision has been taken to not integrate this modification directly in the core but rather in userland to have the choice to apply it or not. It could be useful for example to detect if the targeted host is running Windows or Linux for instance.

Pcap writer In usual situations, USB communications can be investigated with a VMware virtual machine or the `USBMon` Linux kernel

module [19]. But it implies that we can modify the USB host, and this turns out to sometimes be impractical or impossible.

The `pcap_writer` program was developed to work around this problem. It stores all packets forwarded from one part to another in a PCAP file, in order to perform offline investigations.

Because the core is running at driver and gadget level, above USB controllers and hardware, USB acknowledgment messages do not reach the driver layer and are either intercepted by the controller or by the hardware. Even if those messages are not useful for the USB communication understanding, they are needed by many analyzing tools such as `wireshark`. Nevertheless, the content of those packets is predictable (acknowledgment without data), therefore the `pcap_writer` program adds them to the pcap file.

USB Firewall The purpose of this program is to restrict access to the USB host stack to only authorized USB devices. The filtering is done on the first packet between userland and core: the `NEW_DEVICE` packet. It contains the following descriptors:

- Device descriptor
- Configuration descriptor
- Interface descriptors
- Endpoint descriptors

This is sufficient for instance to only allow HID or MassStorage devices, or to filter on the `ProductID`/`VendorID`. Because filtering is done on this `NEW_DEVICE` packet and not on a forwarded real USB communication (because USB transfer has not yet started), if information does not match an authorized device, the protected host will not see any USB connection at all.

USBMutation The purpose of this program is to apply random modifications on a forwarded USB communication.

In order to assess the robustness of a USB driver, it is possible to implement a USB device emulation to attack it, which can be time consuming because it is necessary to understand all USB exchanges. It is also possible to forward the data between a real USB device and its driver and make modification on the fly. Even if the former should lead to better results, the latter has the advantage of being very easy to implement and being driver independent. This latter method is implemented by `USBMutation`.

Keylogger This program forwards keyboard communications to a host and logs all the keystroke. This capture can then be replayed with the `keyboard` device script described below. It interprets all the scancodes to determine the original input text.

This program is easy to understand for non technical people and is thus useful for demonstration purposes.

5.2 Device Emulator

The USBq design allows full userland device emulation, and its API allows users to focus on their specific objective.

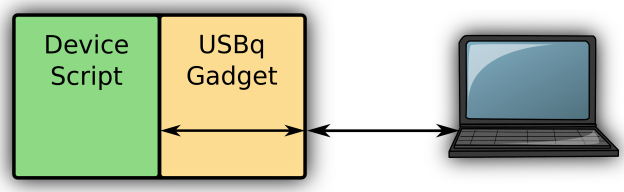

Fig. 11. Device Emulation

Device emulators can either directly communicate with the *USBq core* gadget, or through one or more proxy programs.

```
MassStorageInterface = Interface(descriptors=[Endpoint(1,BULK,IN,512),Endpoint(1,BULK,OUT
    ,512)],cls=8,subcls=6,proto=80)

class EvilMassStorage(USBDevice):
    """ Triggers vulnerability in Windows 8.1, found by QB """
    @classmethod
    def create_arg_subparser(cls,parser):
        parser.add_argument("--vid","-v",metavar="ID",default=0x64,type=int,help="VendorID
            to set")

    def __init__(self,args):
        ident = DeviceIdentity.from_interface(MassStorageInterface)
        ident.device.idVendor = args.vid
        ident.interface[0].bNumEndpoint = 0
        super(EvilMassStorage,self).__init__(args,ident)

if __name__ == "__main__":
    parser = EvilMassStorage.create_arg_parser()
    args = parser.parse_args()

    mass = EvilMassStorage(args)
    mass.init()
    mass.run()
```

Listing 11. Windows 8.1 Vulnerability

The code presented in 11 triggers a vulnerability discovered by QuarksLab [11] on Windows 8.1 with a few lines of code.

Keyboard This program acts as a keyboard and sends keystrokes to the host. It can act interactively, where keypress are received from the standard input, or read the keys from a file. This file can be manually created, or be the result of the `keylogger` proxy program. It can then become a low cost USB rubber ducky [12].

```
# cat key.txt
{SUPER_RIGHT}1rcmd.exe
net user /add toto toto12
net localgroup administrators toto /add
# keyboard.py --server-ip beagle -i key.txt
```

Listing 12. Add windows user

Fuzzer USBq can simulate connections or disconnections of USB devices using `NEW_DEVICE` and `RESET` management messages. The `fuzzer` program simulates several devices that are connected one after another and sends invalid USB descriptors in order to fuzz the host USB stack. It is similar to the fuzzer implemented by `umap` [16].

Fingerprint Like the `Fuzzer` program, `Fingerprint` program simulates USB devices connected one after another and analyses requests sent by the host to detect which kind of host USB stack it is communicating with. It could be improved to fully simulate devices instead of performing the enumeration process, and make it possible to obtain a deeper view into the target. It could be used not only to detect the target operating system but also target the driver version.

Pcap reader This program reads a pcap file from a previous communication and replays it. Replay is more complicated than only sending one packet after the other. The steps involved are to determine USB descriptors of the device that need to be replayed, then to connect to the host to respond to USB descriptor solicitations (that does not necessarily happened in the same order as the capture), and then to replay the data part of the communication.

It can be used for example to solve part of the 2015 SSTIC challenge (see figure 12).

USBScan The `USBScan` program is similar to the one developed in `umap` [16]. It emulates several classes of USB devices (HID, mass storage, Ethernet card...) connected one after another and tries to determine which

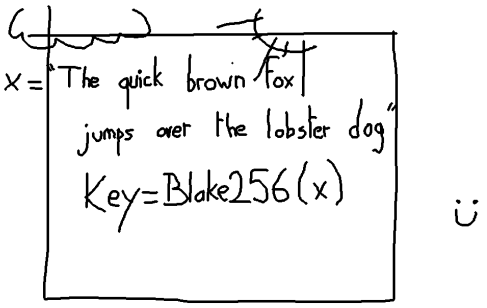

Fig. 12. SSTIC Challenge

ones can be handled by the host. It is useful to estimate the host attack surface, and to understand which USB drivers it embeds. For now, only mass storage and HID classes are handled.

5.3 Host emulator

Just like it is possible to emulate in device in userland, it is also possible to the same for a host. In this configuration, a valid device is plugged into an emulated host that will interact with it.

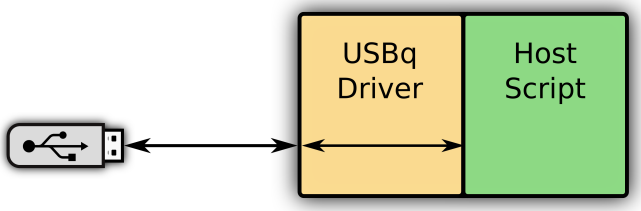

Fig. 13. Host Emulation

Several concepts such as fuzzing or fingerprinting used in device emulation can also be used in host emulation, but for now no USB host devices have been developed.

5.4 Way forward

USBq suffers from several limitations linked to devices that cannot be forwarded through the USBq core. Improvement are therefore needed to handle more USB devices, for example isochronous management.

The communication between the USBq core and the userland could be changed from UDP to netlink [10]. It would permit to have better performance if the userland is running on the same board as the core, while being flexible.

In parallel, many userland scripts are currently in a *proof of concept state*, and need to be improved to become robust enduser tools. For instance `USBScan` handle only two different classes... it needs to be completed to get an accurate overview of drivers managed by the host.

5.5 Conclusion

The USBq project provides a modular design allowing the userland implementation of devices, hosts or MITM USB programs. It is the result of a growing need of skills in the USB domain.

More than a toolbox it has to be seen as a flexible framework that can be very easily adapted for future needs and problems, either for finding vulnerabilities on USB host stacks and drivers, or for pentesting uncontrolled host.

References

1. Beagle Bone Black. https://www.isee.biz/products/igep-processor-boards/igepv2-dm3730.
2. Facedancer. http://goodfet.sourceforge.net/hardware/facedancer21/.
3. Facedancer Recon presentation. https://recon.cx/2012/schedule/attachments/57_recon2012-goodspeedbratus.pdf.
4. Fancedancer Hardware. http://goodfet.sourceforge.net/hardware/facedancer21/.
5. Gadget API. https://www.kernel.org/doc/htmldocs/gadget/.
6. GadgetFS. http://www.linux-usb.org/gadget/.
7. IGEPv2. https://www.isee.biz/products/igep-processor-boards/igepv2-dm3730.
8. libusb. http://www.libusb.org/.
9. Moulinex Cookéo USB. http://cookeo.moulinex.fr/cookeo/cookeo-usb.
10. Netlink Sockets. https://en.wikipedia.org/wiki/Netlink.
11. Quarkslab: From fuzzing to bug reporting. http://blog.quarkslab.com/usb-fuzzing-basics-from-fuzzing-to-bug-reporting.html.
12. Rubber Ducky. http://hakshop.myshopify.com/products/usb-rubber-ducky-deluxe?variant=353378649.
13. Scapy. http://www.secdev.org/projects/scapy/.
14. TTWE. https://www.usenix.org/system/files/conference/woot14/woot14-vantonder.pdf.
15. TTWE Github. https://github.com/rvantonder/ttwe-proto.
16. Umap. https://github.com/nccgroup/umap.
17. URB. https://www.kernel.org/doc/Documentation/usb/URB.txt.

18. USB On The Go. https://en.wikipedia.org/wiki/USB_On-The-Go.
19. USBMon. https://www.kernel.org/doc/Documentation/usb/usbmon.txt.
20. USBSniffer. http://beagleboard-usbsniffer.blogspot.fr/.
21. Wikipedia USB. https://en.wikipedia.org/wiki/USB.
22. Jan Axelson. *USB Complete fourth Edition: The developer's guide*. 2009.