USB – Undermining Security Barriers

Andy Davis
Research Director
andy.davis@ngssecure.com

An NGS Secure Research Publication

4 August 2011

Contents

Introduction

USB ports are everywhere, from laptops and phones to TVs and even microwave ovens. They provide a convenient, reasonably fast interconnect that consumers are familiar with. However, many users of the technology are completely unaware of the amount of interaction that takes place between a USB device and its host at the point of insertion. Just by plugging in a USB device you trigger a flurry of communication in which the capabilities of the device are explained to the host, all before any real device-specific data is exchanged. It is this communications interchange that has been the focus of this research.

Although the concept of identifying and exploiting vulnerabilities in USB drivers is not new, the approach presented here is, as it provides the capability to test any USB platform or device (previous techniques have been either device or USB-host dependent). Although the approach is simple, its effectiveness has been clearly demonstrated by the identification of vulnerabilities in USB drivers of many of the well-known operating systems. This paper will cover USB fundamentals, typical USB vulnerability classes and also discuss the real-world impact of exploiting USB vulnerabilities along with their implications for endpoint security products.

Background

USB Primer

Before discussing USB vulnerabilities it is important that the reader has a basic understanding of the USB protocol and its usage. There are a number of protocol-specific terms that must be explained in order to understand USB and although there are many more technical details to USB, this section aims to provide a concise overview of the protocol focusing on information that will provide more clarity to the remainder of the paper. All the research performed to date has focused on USB v2.0 [1] – the most commonly used at the time of publication. Other versions will only be mentioned briefly for comparison purposes.

The primary aim of the USB interface was to develop a simple, flexible solution to the mixture of connection methods to the PC. Although USB v1.0 [2] was released in January 1996, the first widely used version of the protocol (v1.1) [3] was released in September 1998 and offered two different speeds:

- Low speed – 1.5Mbps (keyboard, mouse etc.)
- Full speed – 12Mbps (faster devices such as disk drives)

The currently most popular version of the standard (v2.0) was then released in April 2000, which offered an additional higher speed capability:

- High speed – 480Mbps (reputedly developed in response to FireWire)

The most recent version (v3.0) [4], which was released in November 2008, offers an even higher speed and backward compatibility with v2.0:

- SuperSpeed – 5Gbps (the version also offers lower power consumption)

The USB architecture is a tiered star topology (see Figure 1) with a single host controlling up to 127 slave devices (devices are officially called “functions” in USB terminology, however devices with more than one function are called composite devices). Devices can be plugged into hubs and hubs can be plugged into other hubs, however the maximum number of tiered hubs permitted by the protocol is six.

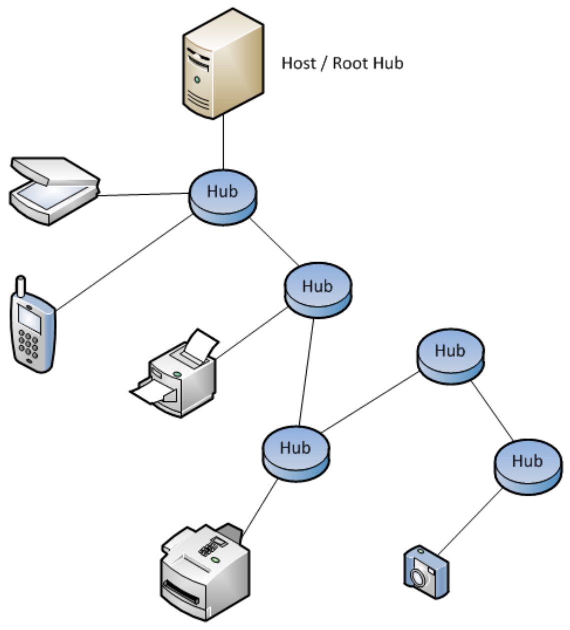


Figure 1 – The USB tiered star topology

The host is considered to be the Master and therefore, all communications on the bus are initiated by the host. Furthermore, as all USB devices are treated by the host as slaves, so devices cannot communicate directly with other devices (except when using the USB On-The-Go protocol [5]).

The USB bus

When the host is transmitting a packet of data, it is sent to every device connected to an active port on a hub. It travels downwards via each hub, which in turn relays the packet to the next in the chain until it reaches all connected devices. However, the packet is only accepted by the device for which the packet is addressed.

Endpoints

Each USB device has a number of endpoints, each of which is a source or sink of data. A device can have up to 16 OUT and 16 IN endpoints. OUT always means from host to device and IN always means from device to host. Endpoint 0 cannot be used for general communication, as it has been implemented purely for controlling the device.

Pipes

These are logical data connections between the host and endpoints. The device communicates using endpoints, but the client software communicates through pipes. There are a set of parameters associated with each pipe to describe capabilities such as allocated bandwidth, transfer type and data flow direction.

Device Configurations

The device contains a number of **Descriptors** (see Figure 2) which help to define the device's capabilities. A device can have more than one **Configuration**, though only one at a time and to change configuration the device must be reset. A device can have one or more **Interface** and each interface can have a number of **Endpoints** and represents a functional unit belonging to a particular class. Certain classes of device have class-specific descriptors, such as the HID (Human Interface Device) class.

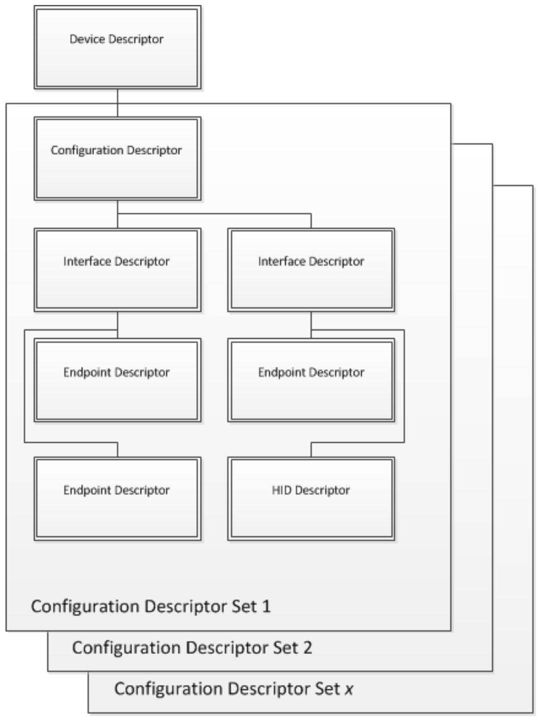
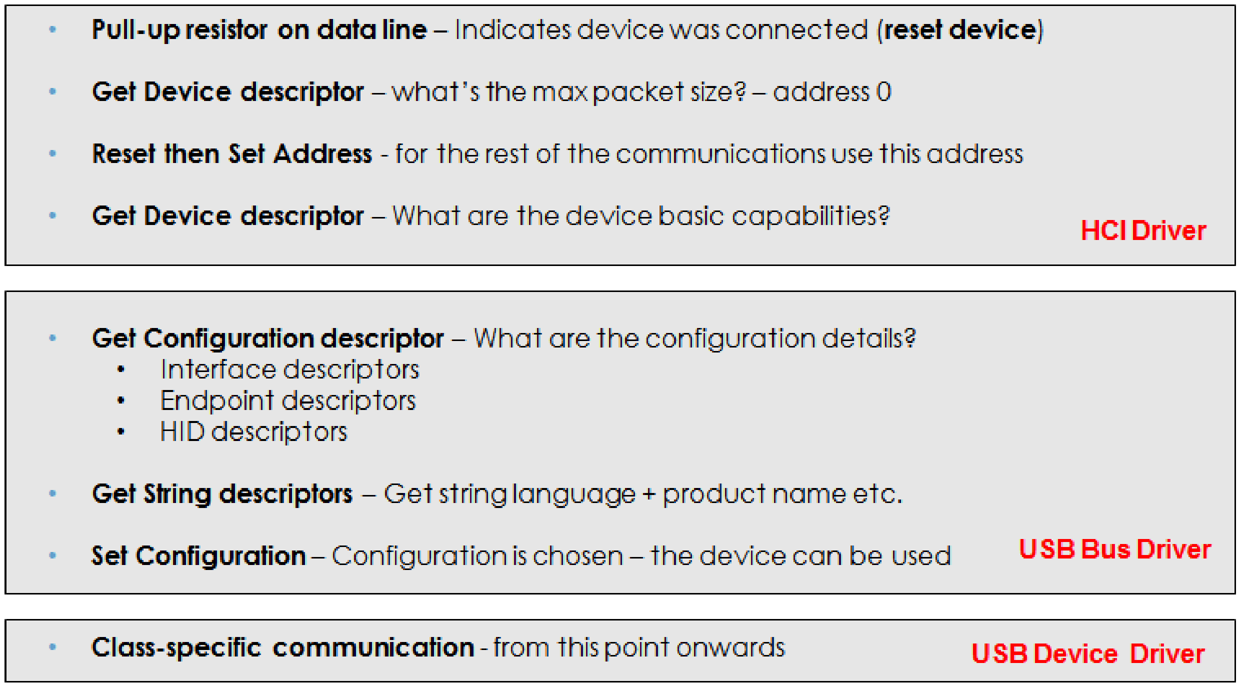


Figure 2 – USB Configuration Descriptor Set

Insertion of a USB device

When a USB device is inserted, a number of specific actions are performed which result in different software components processing data provided by either the host or device. Figure 3 summarises this process.



- **Pull-up resistor on data line** – Indicates device was connected (**reset device**)
- **Get Device descriptor** – what's the max packet size? – address 0
- **Reset then Set Address** - for the rest of the communications use this address
- **Get Device descriptor** – What are the device basic capabilities?

HCI Driver

- **Get Configuration descriptor** – What are the configuration details?
  - Interface descriptors
  - Endpoint descriptors
  - HID descriptors
- **Get String descriptors** – Get string language + product name etc.
- **Set Configuration** – Configuration is chosen – the device can be used

USB Bus Driver

- **Class-specific communication** - from this point onwards

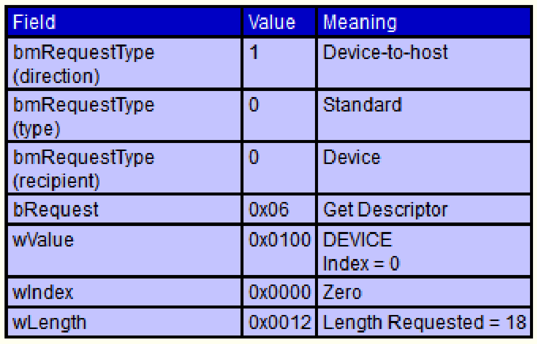
USB Device Driver

Figure 3 – Device insertion and enumeration process

1. The host first detects that a device has been inserted due to one of the data lines (if it's the D-data line then it is a "Low speed" device otherwise, the D+ line indicates the presence of a "Full speed" device) being pulled high by the 1.5k resistor at the device.
2. The host issues a RESET command to the device to ensure that it is in a known state.
3. The host issues a GET_DESCRIPTOR command for the Device descriptor to address 0.
4. The device sends the Device descriptor, which includes the maximum packet size defined for the control endpoint.
5. The host issues another RESET command and then a SET_ADDRESS command with an address >= 1 (all subsequent communication will use this address).
6. The host issues another GET_DESCRIPTOR command for the Device descriptor and the device replies with the Device Descriptor.
7. The host requests the Configuration descriptor, which includes the Interface and Endpoint descriptors (and in some instances other descriptors e.g. HID descriptor) to which the device replies.
8. The host requests String descriptors, which contain the list of supported string languages and various strings associated with the vendor name and product name of the device.
9. Sometimes the host may ask for some of the previously discussed information again, which is often the result of different drivers in the USB stack requesting the same information.

10. The host issues a SET_CONFIGURATION request and the device may now be used.
11. At this point class-specific descriptors may be requested from the device e.g. HID Report Descriptor
12. The host identifies which device driver to load based on the Vendor and Product ID in the Device descriptor. Class-specific communication is now performed between the host and the device.

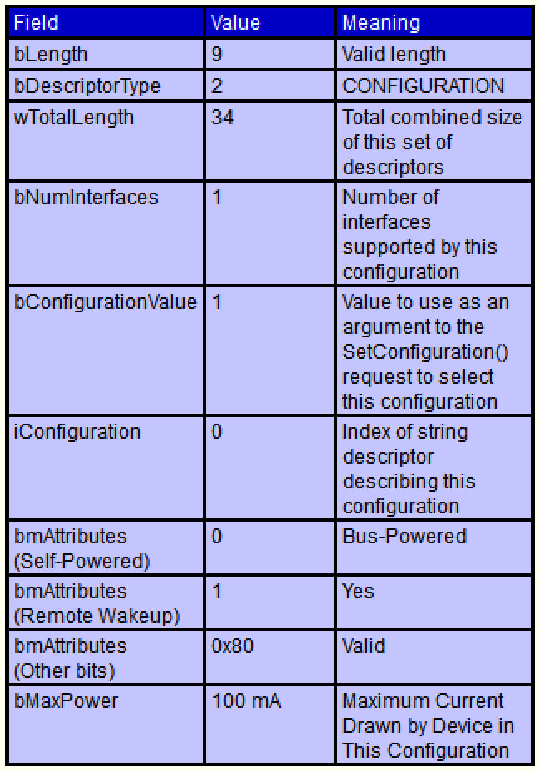
An example of a typical USB command packet sent from the host to a device is shown in Figure 4.



Field	Value	Meaning
bmRequestType (direction)	1	Device-to-host
bmRequestType (type)	0	Standard
bmRequestType (recipient)	0	Device
bRequest	0x06	Get Descriptor
wValue	0x0100	DEVICE Index = 0
wIndex	0x0000	Zero
wLength	0x0012	Length Requested = 18

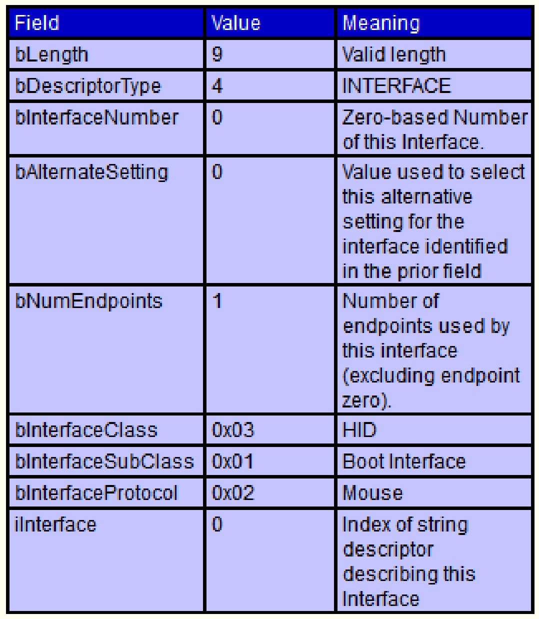
Figure 4 – A GET_DESCRIPTOR command to obtain the Device descriptor

Examples of the type of information found in common USB descriptors are shown in Figures 5 to 8.



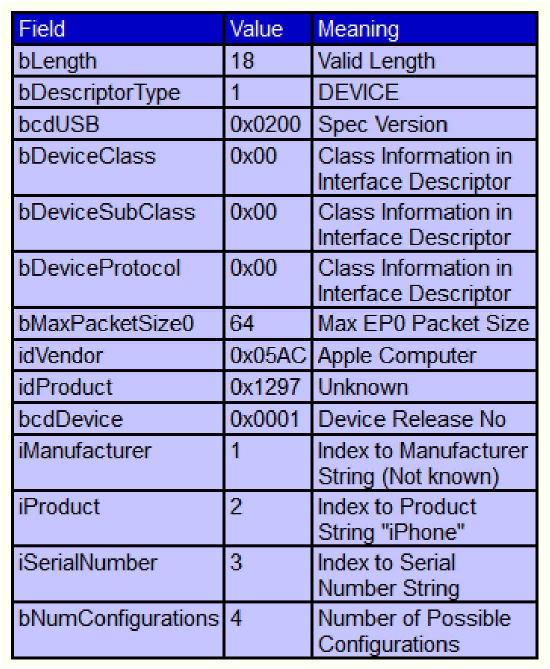
Field	Value	Meaning
bLength	9	Valid length
bDescriptorType	2	CONFIGURATION
wTotalLength	34	Total combined size of this set of descriptors
bNumInterfaces	1	Number of interfaces supported by this configuration
bConfigurationValue	1	Value to use as an argument to the SetConfiguration() request to select this configuration
iConfiguration	0	Index of string descriptor describing this configuration
bmAttributes (Self-Powered)	0	Bus-Powered
bmAttributes (Remote Wakeup)	1	Yes
bmAttributes (Other bits)	0x80	Valid
bMaxPower	100 mA	Maximum Current Drawn by Device in This Configuration

Figure 5 – Configuration descriptor



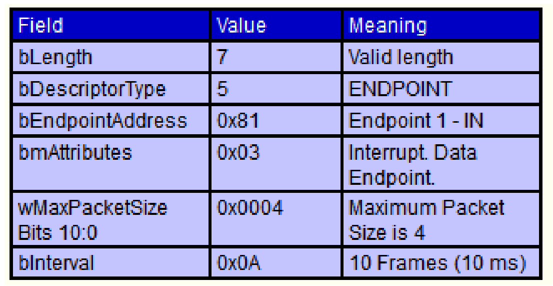
Field	Value	Meaning
bLength	9	Valid length
bDescriptorType	4	INTERFACE
bInterfaceNumber	0	Zero-based Number of this Interface.
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumEndpoints	1	Number of endpoints used by this interface (excluding endpoint zero).
bInterfaceClass	0x03	HID
bInterfaceSubClass	0x01	Boot Interface
bInterfaceProtocol	0x02	Mouse
iInterface	0	Index of string descriptor describing this Interface

Figure6 – Interface descriptor



Field	Value	Meaning
bLength	18	Valid Length
bDescriptorType	1	DEVICE
bcdUSB	0x0200	Spec Version
bDeviceClass	0x00	Class Information in Interface Descriptor
bDeviceSubClass	0x00	Class Information in Interface Descriptor
bDeviceProtocol	0x00	Class Information in Interface Descriptor
bMaxPacketSize0	64	Max EP0 Packet Size
idVendor	0x05AC	Apple Computer
idProduct	0x1297	Unknown
bcdDevice	0x0001	Device Release No
iManufacturer	1	Index to Manufacturer String (Not known)
iProduct	2	Index to Product String "iPhone"
iSerialNumber	3	Index to Serial Number String
bNumConfigurations	4	Number of Possible Configurations

Figure 7 – Device Descriptor



Field	Value	Meaning
bLength	7	Valid length
bDescriptorType	5	ENDPOINT
bEndpointAddress	0x81	Endpoint 1 - IN
bmAttributes	0x03	Interrupt. Data Endpoint.
wMaxPacketSize Bits 10:0	0x0004	Maximum Packet Size is 4
bInterval	0x0A	10 Frames (10 ms)

Figure 8 – Endpoint descriptor

Previous Work

Previous work in USB vulnerability discovery and exploitation has been done by a number of people, including:

- **Rafael Dominguez Vega** used a number of different approaches including USB over IP, QEMU and a microcontroller to discover Linux-based USB vulnerabilities in String descriptors [6] [7]
- **David Dewey and Darrin Barrall** used an SL811 USB controller to discover a heap overflow in Windows XP [8]
- **Moritz Jodeit** emulated USB devices in software and with a Netchip NET2280 peripheral controller [9]
- **Tobias Mueller** used virtualisation techniques with QEMU to identify USB vulnerabilities in a number of different operating systems [10]
- **Jon Larimer** investigated USB drive security and associated file system vulnerabilities in Windows and Linux [11]

Research

The primary goal of the research was to develop a USB host and device independent "black box" security testing capability, as all of the publicly discussed attempts to perform runtime security testing of USB driver software have been in some way tied to a specific operating system.

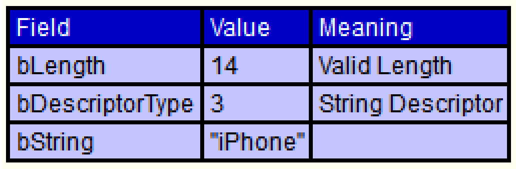
USB vulnerability classes

Before deciding how to approach the emulation of USB devices and hosts, it made sense to look at what could actually be tested and the types of vulnerability class that may be identified.

Stack overflows

This class of vulnerability is becoming rarer these days. However, they are still being discovered. As can be seen in [12] and [13], lazy programming associated with the processing of String Descriptors resulted in the presence of exploitable stack overflows.

As Figure 8 shows, each string descriptor has a *bLength* field associated with the length of *bString*, however, in both [12] and [13] a fixed length buffer, which was smaller than the maximum *bLength* value (255) had been allocated into which the strings were copied.



Field	Value	Meaning
bLength	14	Valid Length
bDescriptorType	3	String Descriptor
bString	"iPhone"	

Figure 8 – A typical String descriptor

Heap and Integer overflows

Scattered throughout the descriptors and class-specific data are length fields representing sizes of chunks of data that may result in drivers needing to dynamically allocate memory in which to store them; examples include:

- Each descriptor starts with a *bLength* field, which represents the length of the descriptor
- Configuration Descriptor – *wTotalLength*
- Endpoint Descriptor – *wMaxPacketSize*
- Image class: DeviceInfo - *Capture Formats Supported Array Size*
- Printer class: DeviceID - *Device ID Length*

Any of these, plus many others could potentially result in an integer overflow, which in turn could trigger another vulnerability such as a heap overflow.

Examples of publicly released USB-based heap overflows include:

- USB hub class configuration descriptor - Sony Playstation 3 [14]
- "0xA1,1"* USB control message – Apple iPod Touch [15]

* 0xA1 is the "RequestType", 1 is the "Request" (see Figure 4 for an example USB device request)

Null-pointer dereference

More often than not, null pointer dereference bugs cannot be exploited, however in some instances, such as the example below, data can be written to address zero and exploitation becomes possible.

- "0x21,2" USB control message – Apple iPhone and iPod Touch [16]

Logic errors

The HID report descriptor is a notoriously complicated structure and therefore, "the parser for the Report descriptor represents a significant amount of code" [17]. There is certainly scope for the presence of logic errors in this code.

Once it had been established that there were plenty of potential fuzz test cases, all that was needed was a fuzzer. The initial attempts followed a similar approach to a number of other researchers – using a microcontroller to emulate a USB device.

Phase One - Fuzzbox

The "Fuzzbox" approach was developed using an Arduino microcontroller.

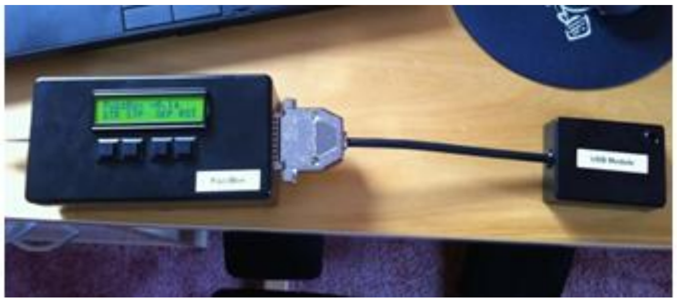


Figure 7 – Fuzzbox -the Arduino-based USB fuzzer

"Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments" [18]

In other words, it's great for prototyping hardware e.g. pretending you're a USB device. Also, much of the hard work had already been done by others; a circuit had been designed to interface the Arduino to a USB host, acting as a HID device [19]:

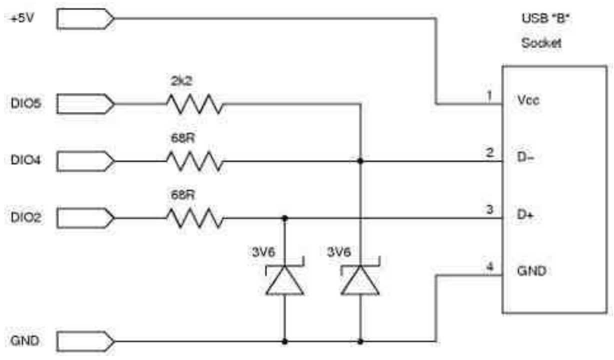


Figure 9 – A circuit to interface an Arduino microcontroller to a USB host

Driver code had also been written to emulate a HID device [20].

What was missing was the application of this technology to security testing. With an understanding of the likely vulnerability classes to test for and the ability to insert arbitrary data into the USB device enumeration phase, additional Arduino-based code was developed to automate the fuzzing of USB host drivers on any platform. Fuzzbox was created and quickly started to discover bugs:

- OS X - (at time of publication these are being investigated by Apple)
- Windows XP – a number of bugs were identified but nothing that appeared exploitable
- Windows 7 – (three bugs, which at time of publication are being investigated by Microsoft)

However, it soon became apparent that this approach had some pretty fundamental drawbacks. Firstly, new driver code would need to be discovered, adapted or written from scratch for each USB device class and secondly, the Arduino microcontroller would not be fast enough to emulate some of the higher speed devices. Therefore, a new approach was required – one that was more flexible.

Phase Two – Commercial test equipment

There are many commercially available (and some free, software-based) USB analysers, however only a very small number of these also allow USB traffic to also be generated. An analysis of the available devices at the time of the research revealed that the following USB analyser was the best fit for our requirements:

Packet Master USB500 AG

The Packet Master device [21] has the following capabilities:

- Dedicated USB test equipment hardware
- USB capture and playback
- Can emulates any USB host or device (USB v1.1 or v2.0)
- Understands and analyses the different USB device classes
- Uses a scripting language to generate USB traffic
- Costs approx. $1400 (plus specific class analysis options)
- Limitations – doesn't currently have a software API to control it

The three important capabilities are the ability to play back captured USB traffic, generate traffic using a scripting language and emulate either a USB host or device. These capabilities, combined with an understanding of where potential USB vulnerabilities may exist formed a solid base from which the USB fuzzer could be developed.

Phase Three - Frisbee

Although there is a comprehensive Win32 GUI application (GraphicUSB) to control the device, the main limitation of the Packet Master is the lack of a software control API. Therefore, we needed to develop a different way to control it. The initial plan was to write the fuzzer using the USB scripting language provided with the device, however, it soon became clear that the language was not fully-featured enough to do this e.g. there is no concept of variables or of conditional loops.

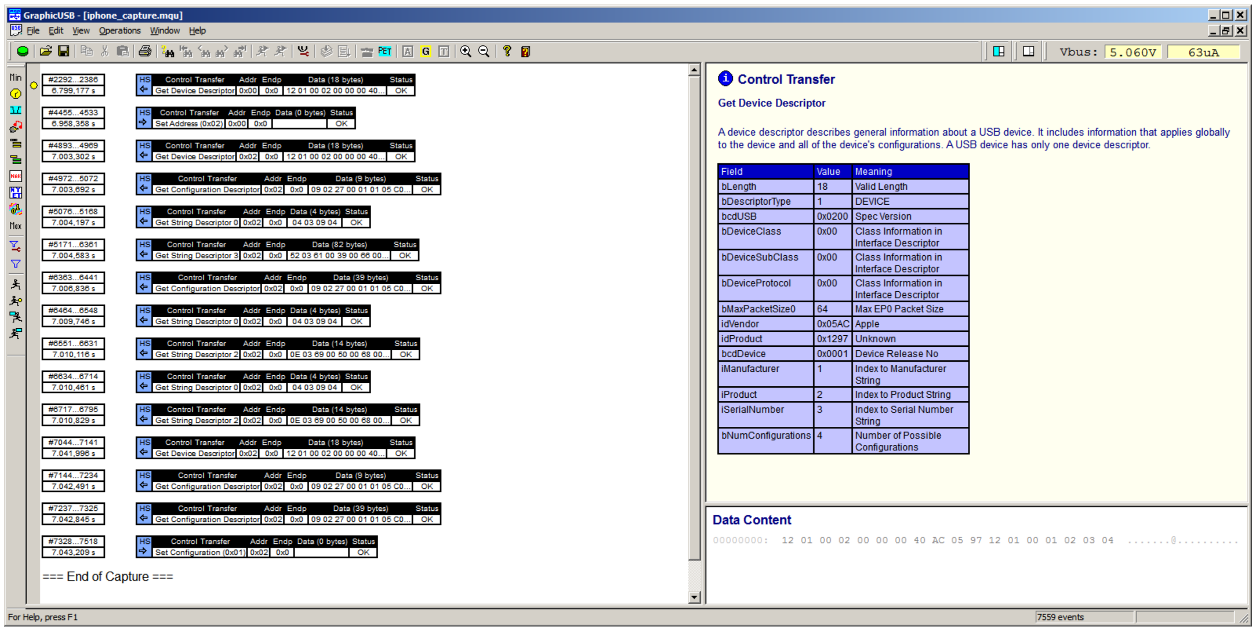
One issue with fuzzing USB that was discovered during Phase one was the speed limitations imposed by the protocol during the emulation of the device insertion and removal. In order to allow a host enough time to fully enumerate an inserted USB device, fuzz test cases could only be sent approximately every six seconds. Although frustrating, this limitation meant that the subsequent remote control technique developed for the Packet Master, which was itself quite slow, did not actually matter. It was decided that at least in the short-term, the easiest way to control the Packet Master was to inject Windows events into GraphicUSB, as there would only need to be limited interaction to perform the fuzzing.

The Packet Master has the ability to create a "Generator script" (See Figure 11) for either a host or device, based on a previous capture (See Figure 10). As can be seen in Figure 11, the bytes highlighted are the contents of the Device descriptor displayed in Figured 9. This script is saved as a text file, then compiled and uploaded to the Packet Master via USB. An option within GraphicUSB will then execute the script, replaying the traffic previously captured. Alternatively, scripts can be developed from scratch and executed.

The design of the fuzzer would therefore be as follows:

- The USB traffic associated with the insertion of a device (for either a host or device of interest) would be captured and a generator script created.
- The Python-based fuzzing engine (Frisbee) would then iteratively modify the generator script with fuzz test cases developed based on an understanding of the USB protocol and likely potential vulnerability classes.
- For each test case Frisbee would open GraphicUSB with the generator script, inject the appropriate Windows events into the application using the Python SendKeys library [22] in order to compile the generator script, execute it and then exit the application. Although this somewhat “clunky” approach was quite slow, it could still be performed for each test case faster than the enumeration phase could be performed by a USB host.
- Basic instrumentation would be performed via ICMP ping packets sent to and received from the target, based on the premise that most of the bugs, when triggered would cause a Windows Bugcheck, Unix kernel panic or reboot to either a USB device or host.

The fuzzer was successfully developed in Python and then used to identify bugs primarily in USB host implementations by capturing the traffic associated with a range of different classes of USB device and then fuzzing this traffic associated with USB enumeration. With respect to instrumentation, after using Frisbee for a while, it became apparent that in some circumstances the bug identified would crash part of the USB stack, but not the underlying operating system and hence the simple ICMP ping approach was considered inadequate. Therefore, it was updated to execute a “known good” generator script after each test case to establish if the USB stack on the target was still correctly functioning.


GraphicUSB - [iphone_capture.mqu]
File Edit View Operations Window Help
Vbus: 5.060V 63uA
For Help, press F1
7559 events

```
#2292...2386   HS  Control Transfer               Addr  Endp  Data (18 bytes)              Status
6.799,177 s    ←   Get Device Descriptor          0x00  0x0   12 01 00 02 00 00 00 40...   OK
#4455...4533   HS  Control Transfer               Addr  Endp  Data (0 bytes)               Status
6.958,358 s    →   Set Address (0x02)             0x00  0x0                                OK
#4893...4969   HS  Control Transfer               Addr  Endp  Data (18 bytes)              Status
7.003,302 s    ←   Get Device Descriptor          0x02  0x0   12 01 00 02 00 00 00 40...   OK
#4972...5072   HS  Control Transfer               Addr  Endp  Data (9 bytes)               Status
7.003,692 s    ←   Get Configuration Descriptor   0x02  0x0   09 02 27 00 01 01 05 C0...   OK
#5076...5168   HS  Control Transfer               Addr  Endp  Data (4 bytes)               Status
7.004,197 s    ←   Get String Descriptor 0        0x02  0x0   04 03 09 04                  OK
#5171...6361   HS  Control Transfer               Addr  Endp  Data (82 bytes)              Status
7.004,583 s    ←   Get String Descriptor 3        0x02  0x0   52 03 61 00 39 00 66 00...   OK
#6363...6441   HS  Control Transfer               Addr  Endp  Data (39 bytes)              Status
7.006,836 s    ←   Get Configuration Descriptor   0x02  0x0   09 02 27 00 01 01 05 C0...   OK
#6464...6548   HS  Control Transfer               Addr  Endp  Data (4 bytes)               Status
7.009,746 s    ←   Get String Descriptor 0        0x02  0x0   04 03 09 04                  OK
#6551...6631   HS  Control Transfer               Addr  Endp  Data (14 bytes)              Status
7.010,116 s    ←   Get String Descriptor 2        0x02  0x0   0E 03 69 00 50 00 68 00...   OK
#6634...6714   HS  Control Transfer               Addr  Endp  Data (4 bytes)               Status
7.010,461 s    ←   Get String Descriptor 0        0x02  0x0   04 03 09 04                  OK
#6717...6795   HS  Control Transfer               Addr  Endp  Data (14 bytes)              Status
7.010,829 s    ←   Get String Descriptor 2        0x02  0x0   0E 03 69 00 50 00 68 00...   OK
#7044...7141   HS  Control Transfer               Addr  Endp  Data (18 bytes)              Status
7.041,996 s    ←   Get Device Descriptor          0x02  0x0   12 01 00 02 00 00 00 40...   OK
#7144...7234   HS  Control Transfer               Addr  Endp  Data (9 bytes)               Status
7.042,491 s    ←   Get Configuration Descriptor   0x02  0x0   09 02 27 00 01 01 05 C0...   OK
#7237...7325   HS  Control Transfer               Addr  Endp  Data (39 bytes)              Status
7.042,845 s    ←   Get Configuration Descriptor   0x02  0x0   09 02 27 00 01 01 05 C0...   OK
#7328...7518   HS  Control Transfer               Addr  Endp  Data (0 bytes)               Status
7.043,209 s    →   Set Configuration (0x01)       0x02  0x0                                OK

=== End of Capture ===
```

Control Transfer

Get Device Descriptor

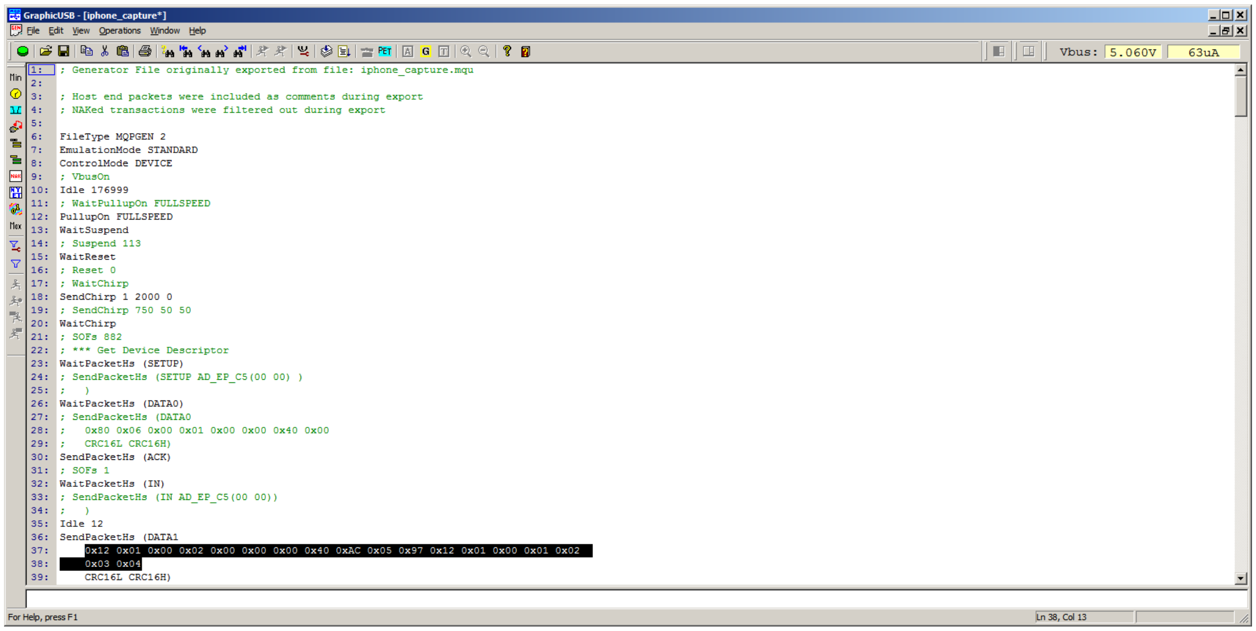
A device descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device has only one device descriptor.

Field	Value	Meaning
bLength	18	Valid Length
bDescriptorType	1	DEVICE
bcdUSB	0x0200	Spec Version
bDeviceClass	0x00	Class Information in Interface Descriptor
bDeviceSubClass	0x00	Class Information in Interface Descriptor
bDeviceProtocol	0x00	Class Information in Interface Descriptor
bMaxPacketSize0	64	Max EP0 Packet Size
idVendor	0x05AC	Apple
idProduct	0x1297	Unknown
bcdDevice	0x0001	Device Release No
iManufacturer	1	Index to Manufacturer String
iProduct	2	Index to Product String
iSerialNumber	3	Index to Serial Number String
bNumConfigurations	4	Number of Possible Configurations

Data Content

```
00000000:  12 01 00 02 00 00 00 40 AC 05 97 12 01 00 01 02 03 04  .......@..........
```

Figure 10 – A sample capture of USB traffic in GraphicUSB



```
; Generator File originally exported from file: iphone_capture.mqu

; Host end packets were included as comments during export
; NAKed transactions were filtered out during export

FileType MQPGEN 2
EmulationMode STANDARD
ControlMode DEVICE
; VbusOn
Idle 176999
; WaitPullupOn FULLSPEED
PullupOn FULLSPEED
WaitSuspend
; Suspend 113
WaitReset
; Reset 0
; WaitChirp
SendChirp 1 2000 0
; SendChirp 750 50 50
WaitChirp
; SOFs 882
; *** Get Device Descriptor
WaitPacketHs (SETUP)
; SendPacketHs (SETUP AD_EP_C5(00 00) )
;    )
WaitPacketHs (DATA0)
; SendPacketHs (DATA0
;    0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00
;    CRC16L CRC16H)
SendPacketHs (ACK)
; SOFs 1
WaitPacketHs (IN)
; SendPacketHs (IN AD_EP_C5(00 00))
;    )
Idle 12
SendPacketHs (DATA1
    0x12 0x01 0x00 0x02 0x00 0x00 0x00 0x40 0xAC 0x05 0x97 0x12 0x01 0x00 0x01 0x02
    0x03 0x04
    CRC16L CRC16H)
```

Figure 11 – A GraphicUSB generator script created from the capture in Figure 9

Why the new approach is different

Because there is no reliance on any underlying operating system (apart from for control, which needs to be Windows-based to run GraphicUSB), the approach is USB host and device independent, which allows for complete "black box" testing of USB targets. However, the hardest part of fuzzing driver code will always be effective instrumentation and subsequent debugging, which obviously would need to be target-specific.

What has been found so far

Although many of the technical details associated with these bugs cannot be revealed, as they are still being investigated by the vendors, below is a list of bugs discovered by Frisbee in the first half of 2011:

- Multiple HID class memory corruptions in Windows 7
- Hub class kernel stack overflow in Solaris 11 Express
- Printer class kernel stack overflow in Solaris 11 Express
- Multiple Image class integer overflows in Microsoft Xbox 360
- Interface Descriptor memory corruption in OS X

This therefore demonstrates that bugs are still present in the drivers of common operating systems, some of which are potentially exploitable, as can be seen below:

Windows 7 - HID class memory corruption #1

```
0: kd> !exploitable
Exploitability Classification: PROBABLY_EXPLOITABLE
Recommended Bug Title: Probably Exploitable - Data from Faulting Address
controls subsequent Write Address starting at Unknown Symbol @ 0x…
```

Windows 7 - HID class memory corruption #2

```
0: kd> !exploitable
Exploitability Classification: PROBABLY_EXPLOITABLE
Recommended Bug Title: Probably Exploitable - Memory Read Access Violation on
Block Data Move starting at nt!memcpy+0x…
This is a second chance read access violation in a kernel mode block data
move, and is therefore classified as probably exploitable.
```

Solaris 11 Express – kernel stack overflow #1

```
Jan 27 13:36:59 solaris ^Mpanic[cpu1]/thread=xxxxxxxx:
Jan 27 13:36:59 solaris genunix: [ID 549817 kern.notice] segkp_fault:
accessing redzone
Jan 27 13:36:59 solaris unix: [ID 100000 kern.notice]
Jan 27 13:36:59 solaris genunix: [ID 353471 kern.notice]
```

Solaris 11 Express – kernel stack overflow #2

```
Jan 26 10:47:08 solaris genunix: [ID 335743 kern.notice] BAD TRAP: type=e
(#pf Page fault) rp=xxxxxxxx addr=xxxxxxxx occurred in module "unix" due to
an illegal access to a user address
Jan 26 10:47:08 solaris unix: [ID 100000 kern.notice]
Jan 26 10:47:08 solaris unix: [ID 839527 kern.notice] sched:
Jan 26 10:47:08 solaris unix: [ID 753105 kern.notice] #pf Page fault
```

Security Impact

Many times in discussions about USB vulnerabilities with vendors they quote the standard line "…but if you have physical access to the machine then you already have full control" – yes, and no. Mobile devices have already been exploited using USB-based vulnerabilities to "jailbreak" them allowing for the unauthorised installation of software. Increasingly users are storing sensitive data in mobile-based apps, which could potentially be compromised on a "jailbroken" device. Agreed, with a PC, if there is no additional protection to the file system such as full disk encryption then yes, there are much easier ways to gain access to the data stored on the machine than by exploiting a memory corruption bug in a USB driver. However, if full disk encryption has been implemented and the operating system is running, a kernel-level attack could easily result in full access to all the data on the machine. Even if the vulnerability was in a userland application communicating with USB (this scenario has been observed by the author), access to data as the logged in user would still prove extremely damaging. Not only could an attacker exfiltrate data, they could also infect the machine with malware, just by inserting a rogue USB device. But surely that's what endpoint protection software is there to prevent, isn't it?

Endpoint protection software

Over the last few years a number of different vendors have produced what have been generically called "endpoint protection solutions" (in this context "endpoints" refer to network endpoints such as workstations or servers rather than the USB protocol definition of an endpoint provided earlier in this paper). Some examples of these products are:

- Lumension (formerly Sanctuary) - Device Control [23]
- CoSoSys – Endpoint Protector [24]
- DeviceLock - Endpoint DLP Suite [25]

The purpose of the products is to provide more granular control over what devices can be connected to an organisation's endpoints. A common requirement for organisations is the control of USB – to prevent users from inadvertently or maliciously adversely affecting the security of the IT estate by plugging in various USB devices. However, although they appear to effectively control conventional USB devices that are inserted into endpoints, our research has shown that kernel-level driver vulnerabilities are often triggered lower down in the USB stack, before the endpoint protection software has even registered that a device has been inserted – effectively undermining this security barrier.

Why USB vulnerabilities are still prevalent

Different vendors appear to view USB-based vulnerabilities in different ways. As explained earlier, some vendors just don't perceive USB vulnerabilities to be a security issue.

Quote from vendor x:
"Thank you for sending this to us. This is something that I will definitely pass on, however since this requires physical access it's not something that we will fix in a security update".

However, in other cases although they take it more seriously, they don't appear to have the capabilities to thoroughly test their own code.

Quote from vendor y:
"We think we've fixed this issue, but we'll need to get you to test it as we don't have the ability to replicate your attack".

Conclusion

Despite the best efforts of many vendors, there are still plenty of USB-based vulnerabilities out there, many of which will be exploitable. The reason they have not yet been discovered is that until now, runtime security analysis of USB has been challenging. The use of off-the-shelf test equipment combined with knowledge of the USB protocol and of different vulnerability classes, has resulted in NGS Secure developing a powerful USB security testing capability.

Hopefully, it has been shown that even though limited physical access is required, USB vulnerabilities still constitute a serious security risk to organisations and that endpoint protection software is in many cases unlikely to protect you from these attacks.

If you really don’t want people to exploit USB on your workstations, disable USB in the BIOS and prevent unauthorised access to the BIOS with a suitably strong password. However, for the ultimate solution, take the military approach and fill the USB sockets with epoxy resin ☺

References

1 - http://www.usb.org/developers/docs/usb_20_021411.zip

2 - http://fl.hw.cz/docs/usb/usb10doc.pdf

3 - http://esd.cs.ucr.edu/webres/usb11.pdf

4 - http://www.usb.org/developers/docs/usb_30_spec_020411d.zip

5 - http://www.usb.org/developers/onthego/USB_OTG_Intro.pdf

6 - http://labs.mwrinfosecurity.com/files/Advisories/mwri_linux-usb-buffer-overflow_2009-10-29.pdf

7 - http://labs.mwrinfosecurity.com/files/Advisories/mwri_caiaq-usb-drivers-buffer-overflow_2011-03-07.pdf

8 - http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Barrall-Dewey.pdf

9 - http://www.informatik.uni-hamburg.de/SVS/archiv/slides/09-01-13-OS-Jodeit-Evaluating_Security_Aspects_of_USB.pdf

10 - https://muelli.cryptobitch.de/paper/2010-usb-fuzzing.pdf

11 - https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf

12 - http://labs.mwrinfosecurity.com/files/Advisories/mwri_caiaq-usb-drivers-buffer-overflow_2011-03-7.pdf

13 - http://labs.mwrinfosecurity.com/files/Advisories/mwri_linux-usb-buffer-overflow_2009-10-29.pdf

14 - http://ps3wiki.lan.st/index.php?title=PSJailbreak_Exploit_Reverse_Engineering

15 - http://theiphonewiki.com/wiki/index.php?title=Usb_control_msg(0xA1%2C_1)_Exploit

16 - http://theiphonewiki.com/wiki/index.php?title=Usb_control_msg(0x21%2C_2)_Exploit

17 - http://www.usb.org/developers/devclass_docs/HID1_11.pdf

18 - http://www.arduino.cc

19 - http://www.practicalarduino.com/projects/virtual-usb-keyboard

20 - http://code.google.com/p/vusb-for-arduino/

21 - http://www.mqp.com/usb500.htm

22 - http://www.rutherfurd.net/python/sendkeys/

23 - http://www.lumension.com/device-control-software/usb-security-protection.aspx

24 - http://www.cososys.com/software/endpoint_protector.html

25 - http://www.devicelock.com/dl/index.htm