Further adventures with USB
13/01/2012

Andy Davis
Research Director
Telephone: +44 (0) 208 401 0070
e-mail: andy.davis@ngssecure.com
Agenda

• **USB Primer**

• Previous work

• USB vulnerability classes

• Phase one – Fuzzbox

• Phase two – Commercial test equipment

• Phase three – Frisbee

• Phase four – Frisbee Lite

• Security impact

• Conclusion
USB – it’s everywhere!

- PCs
- Tablets
- Cameras
- TVs
- Cable / Satellite set-top-boxes
- Telephones
- Microwave ovens!
The aim of USB was to find a solution to the mixture of connection methods to the PC.

Currently three speeds of data transfer (USB 2.0):
- Low – 1.5Mbps (keyboard, mouse etc.)
- Full – 12Mbps (originally for all other devices)
- High – 480Mbps (developed in response to FireWire)

Architecture is a tiered star topology:
- Single host controller and up to 127 slave devices
- A device can be plugged into a hub, and that hub can be plugged into another hub and so on. The maximum number of tiers permitted is six

- Host is Master - all communications on this bus are initiated by the host
- Devices cannot communicate directly with other devices (except for USB On-The-Go protocol)
The USB bus

- When the host is transmitting a packet of data, it is sent to every device connected to an enabled port. It travels downwards via each hub in the chain which resynchronises the data transitions as it relays it. Only one device, the addressed one, actually accepts the data.

Endpoints

- Each USB device has a number of endpoints. Each endpoint 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.
- IN always means from device to host.
- Endpoint 0 is a special case which is a combination of endpoint 0 OUT and endpoint 0 IN, and is used for controlling the device.

Pipe

- A logical data connection between the host and a particular endpoint, in which we ignore the lower level mechanisms for actually achieving the data transfers.
• Configurations, Interfaces, and Endpoints

  • The device contains a number of **descriptors** (as shown to the right) 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 whole device would have to stop functioning

  • A device can have one or more interfaces. Each **interface** can have a number of endpoints and represents a functional unit belonging to a particular class

  • Each **endpoint** is a source or sink of data
• **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?

• **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

• **Class-specific communication** - from this point onwards
A typical Device Descriptor

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

- USB Primer
- **Previous work**
  - USB vulnerability classes
  - Phase one – Fuzzbox
  - Phase two – Commercial test equipment
  - Phase three – Frisbee
  - Phase four – Frisbee Lite
- Security impact
- Conclusion
Various approaches to interface fuzzing

- **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.

- **David Dewey and Darrin Barrall** used an SL811 USB controller to discover a heap overflow in Windows XP.

- **Moritz Jodeit** emulated USB devices in software and with a Netchip NET2280 peripheral controller.

- **Tobias Mueller** used virtualisation techniques with QEMU to identify USB vulnerabilities in a number of different operating systems.

- **Jon Larimer** investigated USB drive security and associated file system vulnerabilities in Windows and Linux.
Difficulties in testing USB

• Black box Testing of drivers for interfaces on embedded systems can be tough:
  • No low-level view of the firmware
  • Hard to debug
  • Hard to hook into firmware

• Testing some interfaces is tough using conventional approaches:
  • Getting a PC to pretend to be a USB device requires some serious driver development – need to turn the root hub into a USB device.
Agenda

• USB Primer
• Previous work
• **USB vulnerability classes**
  • Phase one – Fuzzbox
  • Phase two – Commercial test equipment
  • Phase three – Frisbee
  • Phase four – Frisbee Lite
• Security impact
• Conclusion
USB vulnerability classes

- Stack overflows
  - String descriptors in Linux and Windows XP
  - **Solaris 11 Express hub-class descriptor**

- Integer and Heap overflows
  - Sony PS3 hub-class descriptors
  - Apple iPod Touch jailbreak “0xA1,1” USB control message
  - **Xbox 360 image-class descriptors**

- Null-pointer dereference (exploitable)
  - Apple iPhone jailbreak “0x21,2” USB control message

- Logic errors
  - **Windows 7 HID report descriptors**
Agenda

- USB Primer
- Previous work
- USB vulnerability classes
- **Phase one – Fuzzbox**
- Phase two – Commercial test equipment
- Phase three – Frisbee
- Phase four – Frisbee Lite
- Security impact
- Conclusion
Phase one - FuzzBox

- Arduino microcontroller
- Fuzzer written in C++
- Only emulates USB HID devices
- Only allows semi-automated fuzzing
- Has found bugs in:
  - Windows 7
  - Windows XP
  - OS X
- Limitations – not really fast enough to emulate most USB devices
Simple circuit to interface with an Arduino:

![Circuit Diagram]

Driver code had already been written to emulate a HID device: [http://code.google.com/p/vusb-for-arduino/](http://code.google.com/p/vusb-for-arduino/)
Agenda

• USB Primer
• Previous work
• USB vulnerability classes
• Phase one – Fuzzbox
• **Phase two – Commercial test equipment**
• Phase three – Frisbee
• Phase four – Frisbee Lite
• Security impact
• Conclusion
Packet Master USB500 AG

- Dedicated USB test equipment hardware
- **USB capture and playback**
- **Emulates any USB host or device**
- Understands and analyses the different USB device classes
- **Uses a scripting language to generate USB traffic**
- Costs approx. US$1200 (plus specific class analysis options)
- **Limitations – doesn’t currently have a software API to control it**
GraphicUSB – Capture and Analysis

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>18</td>
<td>Valid Length</td>
</tr>
<tr>
<td>Descriptor Type</td>
<td>1</td>
<td>TLP-OUT</td>
</tr>
<tr>
<td>DeviceClass</td>
<td>0x00</td>
<td>Class Information in</td>
</tr>
<tr>
<td>SubClass</td>
<td>0x00</td>
<td>Interface Descriptor</td>
</tr>
<tr>
<td>Protocol</td>
<td>0x00</td>
<td>Class Information in</td>
</tr>
<tr>
<td>Interface</td>
<td>0x00</td>
<td>Interface Descriptor</td>
</tr>
<tr>
<td>MaxPacketSize</td>
<td>0x04</td>
<td>Max USB Packet Size</td>
</tr>
<tr>
<td>Vendor</td>
<td>0x05AC</td>
<td>Apple Computer</td>
</tr>
<tr>
<td>Product</td>
<td>0x1297</td>
<td>Unknown</td>
</tr>
<tr>
<td>Device</td>
<td>0x0001</td>
<td>Device Release Number</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>1</td>
<td>Index to Manufacturer</td>
</tr>
<tr>
<td>Product</td>
<td>2</td>
<td>String (Not known)</td>
</tr>
<tr>
<td>SerialNumber</td>
<td>3</td>
<td>Index to Serial Number</td>
</tr>
<tr>
<td>NumConfigurations</td>
<td>4</td>
<td>Number of Possible</td>
</tr>
</tbody>
</table>

Data Content

00000000: 12 01 00 02 00 00 00 00 AC 05 97 12 01 00 01 02 03 04    

For Help, press F1
GraphicUSB – Generator Scripts

1: ; Generator File originally exported from file: webcam.mqu
2: ; Host end packets were included as comments during export
3: ; NARmed transactions were filtered out during export
4: 
5: FileType HQPGEN 2
6: EmulationMode STANDARD
7: ControlMode DEVICE
8: ; VbusOn
9: Idle 181799
10: ; WaitPullupOn FULLSPEED
11: PullupOn FULLSPEED
12: WaitSuspend
13: ; Suspend 0
14: WaitSuspend
15: ; Suspend 215
16: WaitReset
17: ; Reset 0
18: ; WaitChirp
19: SendChirp 1 2000 0
20: ; SendChirp 750 50 50
21: WaitChirp
22: ; *** Get Device Descriptor
23: WaitPacketHs (SETUP)
24: ; SendPacketHs (SETUP AD_EF_C5(00 00))
25: ;
26: ; SendPacketHs (DATA0)
27: ; SendPacketHs (DATA0)
28: ; 0x80 0x00 0x00 0x01 0x00 0x00 0x10 0x00
29: ; CRC16L CRC16H
30: ; SendPacketHs (ACK)
31: ; WaitPacketHs (IN)
32: ; SendPacketHs (IN AD_EF_C5(00 00))
33: ;
34: ;
35: ; Idle 5
36: ; SendPacketHs (DATA1)
37: ; 0x12 0x01 0x00 0x02 0xEF 0x02 0x01 0x04 0x6D 0x04 0x21 0x08 0x10 0x00 0x00 0x00
38: ; 0x01 0x01
39: ; CRC16L CRC16H
40: ; WaitPacketHs (ACK)
So what can we fuzz?

USB hosts:

- The Device Descriptor (HCI driver)
- String Descriptors (HCI driver / USB device drivers)
- The Configuration Descriptor: Interface Descriptor (USB device driver)
- The Configuration Descriptor: Endpoint Descriptor (USB device driver)
- Class-specific descriptors e.g. HID Report Descriptor (USB device driver)
- Class-specific communication (USB device driver and client software)

USB devices:

- Host -> device USB control messages
Device and Platform Independent:

- Previous approaches have mainly focussed on either testing or emulating specific devices or hosts
- We don't care (for the most part) what the host or device is
- If we understand USB and where vulnerabilities may be present then we can use a black-box fuzzing approach
- It certainly produces results…
Agenda

- USB Primer
- Previous work
- USB vulnerability classes
- Phase one – Fuzzbox
- Phase two – Commercial test equipment
- **Phase three – Frisbee**
- Phase four – Frisbee Lite
- Security impact
- Conclusion
Frisbee – the USB fuzzer

• GraphicUSB currently has no API for controlling it

• Frisbee modifies the GraphicUSB generator script created from a capture

• The generator script must be loaded, compiled then executed to generate USB traffic

• Frisbee uses the SendKeys Python library to inject Windows events into GraphicUSB

• Although this is slow, the HCI driver needs time to identify a device, which has ostensibly just been plugged in

• The host under test is connected to the same Ethernet network as the host running Frisbee and is periodically pinged to identify if the host has crashed
Demos
What have we found so far?

- HID class descriptor memory corruption in **Windows 7**
- Hub class descriptor stack overflow in **Solaris 11Express**
- Printer class communication memory corruption in **Solaris 11Express**
- Image class communication memory corruption in **Solaris 11Express**
- Image class communication integer overflows in **Microsoft Xbox 360**
- Image class descriptor memory corruption in **Apple OS X**
Reusing the Arduino approach

Small, portable, programmable USB device to trigger USB vulnerabilities:
Why are USB vulnerabilities still prevalent?

• Some vendors don’t seem to view USB vulnerabilities as 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 its not something that we will fix in a security update”

• The knowledge of how to emulate USB devices is not widespread

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”
Agenda

- USB Primer
- Previous work
- USB vulnerability classes
- Phase one – Fuzzbox
- Phase two – Commercial test equipment
- Phase three – Frisbee
- **Phase four – Frisbee Lite**
- Security impact
- Conclusion
A bit more USB primer:

- **bmRequestType**
  - This is a bitmapped field that describes the characteristics of the request.

- **bRequest**
  - This is the actual request that is being sent.

- **wValue**
  - The contents of this field are request-specific.

- **wIndex**
  - The contents of this field are request-specific. However, it is often used to specify an endpoint or interface.

- **wLength**
  - This specifies the length of the data transferred during the second phase of the control transfer. If this field is zero, there is no second (data transfer) phase.
## USB standard requests

<table>
<thead>
<tr>
<th><code>bmRequestType</code></th>
<th><code>bRequest</code></th>
<th><code>wValue</code></th>
<th><code>wIndex</code></th>
<th><code>wLength</code></th>
<th><code>Data</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000B</td>
<td>CLEAR_FEATURE</td>
<td>Feature Selector</td>
<td>Zero</td>
<td>One</td>
<td>Configuration Value</td>
</tr>
<tr>
<td>00000000B</td>
<td>GET_CONFIGURATION</td>
<td>Zero</td>
<td>Zero</td>
<td>One</td>
<td>Configuration Value</td>
</tr>
<tr>
<td>10000000B</td>
<td>GET_DESCRIPTOR</td>
<td>Descriptor Type and Descriptor Index</td>
<td>Zero or ID Language</td>
<td>One</td>
<td>Alternate Interface</td>
</tr>
<tr>
<td>10000001B</td>
<td>GET_INTERFACE</td>
<td>Zero</td>
<td>Interface</td>
<td>Two</td>
<td>Device, Interface, or Endpoint Status</td>
</tr>
<tr>
<td>10000000B</td>
<td>GET_STATUS</td>
<td>Zero</td>
<td>Zero</td>
<td>Two</td>
<td>Device, Interface, or Endpoint Status</td>
</tr>
<tr>
<td>00000000B</td>
<td>SET_ADDRESS</td>
<td>Device Address</td>
<td>Zero</td>
<td>Zero</td>
<td>None</td>
</tr>
<tr>
<td>00000000B</td>
<td>SET_CONFIGURATION</td>
<td>Configuration Value</td>
<td>Zero</td>
<td>Zero</td>
<td>None</td>
</tr>
<tr>
<td>00000000B</td>
<td>SET_DESCRIPTOR</td>
<td>Descriptor Type and Descriptor Index</td>
<td>Zero or ID Language</td>
<td>One</td>
<td>Alternate Interface</td>
</tr>
<tr>
<td>00000000B</td>
<td>SET_FEATURE</td>
<td>Feature Selector</td>
<td>Zero</td>
<td>One</td>
<td>None</td>
</tr>
<tr>
<td>00000001B</td>
<td>SET_INTERFACE</td>
<td>Alternate Setting</td>
<td>Interface</td>
<td>Zero</td>
<td>None</td>
</tr>
<tr>
<td>10000010B</td>
<td>SYNCH_FRAME</td>
<td>Zero</td>
<td>Endpoint</td>
<td>Two</td>
<td>Frame Number</td>
</tr>
</tbody>
</table>
Frisbee Lite – The USB device fuzzer

The user interface is simple and intuitive:
Frisbee Lite – The USB device fuzzer

• Written in wxPython
• Uses LibUSB to provide low level access to USB on Windows
• Uses pyUSB to send raw USB setup request packets
• Download from: http://www.ngssecur.com/research/research-overview/Public-Tools.aspx
• Send all bugs / feedback / feature requests to me
Bugs discovered by Frisbee Lite

- A memory corruption vulnerability in iOS 5 (iPhone 4) causes a kernel panic – potentially exploitable
- A memory corruption vulnerability in iOS 5 (iPhone 4) causes the USB stack to stop responding – exploitability currently being investigated
Agenda

• USB Primer
• Previous work
• USB vulnerability classes
• Phase one – Fuzzbox
• Phase two – Commercial test equipment
• Phase three – Frisbee
• **Security impact**
• Conclusion
The impact of exploiting USB

Typical scenarios of exploiting USB vulnerabilities:

- “Jailbreaking” embedded devices
- Unlocking a locked workstation
- Installing malware in seconds
- Exfiltrating sensitive data in seconds
- What about Endpoint Protection Systems?...
Endpoint protection software:

- Lumension (formerly Sanctuary) - Device Control
- CoSoSys – Endpoint Protector
- DeviceLock - Endpoint DLP Suite

They work higher up the USB stack than many USB driver vulnerabilities exist.

Exploitable USB driver vulnerabilities will often circumvent them.

Some of the vulnerabilities we have discovered, when exploited will circumvent Endpoint Protection Software.
Agenda

• USB Primer
• Previous work
• USB vulnerability classes
• Phase one – Fuzzbox
• Phase two – Commercial test equipment
• Phase three – Frisbee
• Security impact
• Conclusion
Conclusion

• There are still plenty of USB vulnerabilities out there – many of which will be exploitable

• Even though limited physical access is required, USB vulnerabilities still constitute a serious security risk

• Endpoint Protection software is unlikely to protect you from many of these attacks

• If you really don’t want people to exploit USB on your PCs, disable USB in the BIOS

• For ultimate protection, fill the USB sockets with epoxy resin!
Questions?

Andy Davis
Research Director
Telephone: +44 (0) 208 401 0070
e-mail: andy.davis@ngssecure.com

NCC Group Plc, Manchester Technology Centre, Oxford Road, Manchester M1 7EF www.nccgroup.com