An NCC Group Publication

Lessons learned from 50 bugs: Common USB driver vulnerabilities

Prepared by:
Andy Davis
Research Director
andy.davis ‘at’ nccgroup.com

Contents

List of Figures and Tables

1 Introduction

Over the past few years NCC Group has identified over fifty USB driver bugs in all the major operating systems and many of these have affected more than one OS. Based on these discoveries, this paper presents common USB vulnerabilities and how to identify them from a black box testing[1] perspective. The first paper[2] the author wrote on the subject of USB vulnerabilities was presented at Black Hat USA 2011 and this paper updates and extends the information initially presented. Although exploiting USB-based vulnerabilities often requires physical access to a host, the ability to execute arbitrary code (sometimes with kernel privileges) purely by inserting a device into a computer still represents a significant risk to businesses and to governments around the world.

The paper will first discuss the various different approaches to testing USB host-based drivers, some of which are OS-specific and others which require specialist hardware and are as a result OS-agnostic. However, all the techniques outlined in this paper essentially enable malformed data to be inserted into USB descriptors and other class-specific fields which is then processed during the enumeration phase that occurs when a USB device is inserted into a host. This capability facilitates fuzz testing of USB-related host software. The remaining sections will then detail the different vulnerability classes that have been observed in real-world drivers and more importantly, in which USB descriptors these vulnerabilities have been identified. Conclusions are then presented about the current state of USB driver security and the various security best-practice approaches that can be adopted in order to mitigate the issues identified during NCC Group’s research in this domain.

1.1 Previous Research

Security researchers have been investigating USB for at least ten years and during that time some great research has been performed[3][4][5]. This paper will present a range of bugs commonly observed in USB host-based drivers and a USB testing checklist is included in an appendix to assist security researchers and testers to quickly and easily identify vulnerabilities in future.

2 USB terminology overview

Rather than repeat information that is available elsewhere, this section only covers the main concepts relevant to the remainder of the paper. There are a number of USB primers[6] available and the full protocol specifications[7] can be downloaded free of charge from the USB Implementers Forum.

2.1 Enumeration

When a USB device is inserted into a host nothing is known about the device and therefore, a question-answer session begins with the host interrogating the device to identify its properties e.g. what type of device it is, how much power it requires and which drivers need to be loaded. This process is known as enumeration.

2.2 Descriptors

Descriptors are data structures containing information that represents various properties and capabilities of a USB device. The majority of the vulnerabilities encountered have been in driver code that parses the data stored in these structures.

2.3 Class-specific communication

USB devices are categorised based on their class e.g. Image class (cameras), mass storage class (flash memory devices) and HID class (Human Interface Device – mouse / keyboard). Once the enumeration process has completed, the device may then communicate actual data to the host and this is known as class-specific communication. Some vulnerabilities have been observed in code that parses this communication data.

3 Testing methodology

Testing USB drivers on host machines is not a straightforward process, because you either need to emulate a USB device or proxy the traffic between a device and the host. As a result of how the protocol works it would be extremely difficult to convert a USB host e.g. a PC into a USB device and therefore, if you are not modifying the traffic en-route via some kind of hooking or proxy solution, you need to use a hardware-based approach. This section details the various different approaches to testing USB hosts and compares the relative merits of each.

3.1 VM-based

If the target can be run in a VM (Virtual Machine) then USB communication between the host computer and the guest OS will pass through a virtualisation layer and is therefore controllable by the tester. The use of QEMU[8] for this purpose has been previously documented[9] however, the main drawback with this approach is the target must be capable of being run in a VM, so it couldn't be used to test the host USB stack of devices such as tablets, games consoles or Smart TVs.

3.2 Function hooking

If a specific driver is being tested then another approach is to hook functions within the driver and modify the data presented by a real USB device before it is processed by the driver. This can be accomplished using a tool such as Uhooker[10] however, the obvious disadvantage with this approach is it is driver-specific.

3.3 Bespoke hardware

Many researchers who have investigated USB security have concluded that the best approach is to develop their own bespoke hardware device that enables USB devices to be emulated and their characteristics easily manipulated. Some of these devices have been limited to only being able to emulate specific device classes e.g. HID, however, other more recent hardware solutions, such as Facedancer[11] provide a much greater degree of flexibility.

3.4 Native test support

In some circumstances the stack natively supports test frameworks, such as the Microsoft 3.0 USB stack[12]. However, as with the function hooking approach this is OS-specific.

3.5 Test equipment

The final testing option utilises commercial test equipment[13], which records all the traffic between a USB device and a host and can then replay that traffic to emulate the insertion of the device into the host. Furthermore, before the traffic is replayed it can be modified in order to identify vulnerabilities. The author has previously demonstrated how this testing could be automated using Frisbee[14].

Approach	OS agnostic?	Platform agnostic?	Software-only solution	Easy to modify data?
VM-based	✓	✗	✓	✓
Function hooking	✗	✗	✓	✓
Bespoke hardware	✓	✓	✗	✗
Native test support	✗	✗	✓	✓
Test equipment	✓	✓	✗	✓

Table 1: Comparison of different USB host testing approaches

As can be seen in Table 1, there are pros and cons to each testing approach, however, although more expensive (approximately $1400), the most flexible approach (and the approach favoured by the author) is to use USB test equipment controlled using Python scripts.

4 The USB driver stack

Different operating systems implement USB in a number of subtly different ways, however one of the simpler implementations is FreeBSD[15]. As explained in its documentation, drivers can be split into three layers:

"*The lowest layer contains the host controller driver, providing a generic interface to the hardware and its scheduling facilities. It supports initialisation of the hardware, scheduling of transfers and handling of completed and/or failed transfers. Each host controller driver implements a virtual hub providing hardware independent access to the registers controlling the root ports on the back of the machine.*

The middle layer handles the device connection and disconnection, basic initialisation of the device, driver selection, the communication channels (pipes) and does resource management. This services layer also controls the default pipes and the device requests transferred over them.

The top layer contains the individual drivers supporting specific (classes of) devices. These drivers implement the protocol that is used over the pipes other than the default pipe. They also implement additional functionality to make the device available to other parts of the kernel or userland."

A more modern, larger operating system such as Microsoft Windows 8 has a suitably complex USB driver stack[16]. In versions of Windows earlier than Windows XP with Service Pack 2 (SP2) all USB device drivers were required to operate in kernel mode[17], however more recent Windows versions support both kernel mode and user mode[18] drivers and therefore, if a vulnerability exists in a user mode driver then the impact is significantly lower than in a kernel mode driver as the code is not running at ring 0.[19]

Depending on which descriptors are being modified and at what stage during enumeration they are being parsed will result in different parts of the USB driver stack being tested. Sometimes the same descriptor is requested and parsed multiple times during enumeration, each time by a different driver and therefore, in order to fully test the driver stack each request for a descriptor should be tested.

In the next section we will discuss some real-world examples of USB vulnerabilities, the classes of vulnerability that commonly affect USB drivers and which descriptors needed to be modified in order to trigger them.

5 Common USB vulnerabilities

This section details the classes of vulnerability that have been observed in USB drivers and the specific USB communications data that was manipulated in order to identify them.

5.1 Unspecified DoS

Not all USB driver bugs are exploitable in a way which is useful to an attacker. Some, such as null-pointer dereferences[20] or out-of-bound reads[21] in most cases simply result in the driver crashing or a kernel panic occurring. These are still bugs, but not considered to be security-related bugs in the context of USB drivers.

5.2 Buffer overflows

Buffer overflows are the result of inadequate bounds checking when data is written to memory. When excessive data is supplied, program control information is overwritten, which if carefully crafted by an attacker can result in the modification of the control flow of a program. The most common types of buffer overflow are stack-based overflows[23] and heap-based overflows[22], both of which if successfully exploited can result in arbitrary code execution by the attacker. However, most mainstream OS vendors have over the last few years started to implement exploit mitigation strategies, such as DEP (Data Execution Prevention) and ASLR (Address Space Layout Randomisation) to raise the bar of capability required to successfully exploit software vulnerable to memory corruption bugs. Information leakage bugs, which used to be considered much lower impact, are now required in order to defeat ASLR.

String descriptors

Buffer overflows are often associated with string data and therefore, String descriptors are an obvious candidate. These descriptors provide human-readable information about a USB device, such as the manufacturer name or model number. An example String descriptor is shown in Table 2.

Field	Value	Meaning
bLength	52	Descriptor length (including the bLength field)
bDescriptorType	3	String descriptor
bString	“HP Color LaserJet CP1515n”	The string to be stored (in UNICODE UTF-16LE format i.e. two bytes per character)

Table 2: Example String descriptor

The field `bLength` represents the length in bytes of the whole descriptor (the length in bytes of `bString` + one byte for `bDescriptorType` + one byte for the `bLength` byte). The `bString` field is in UTF16-LE format so each character requires two bytes and therefore, the maximum length of string that can be stored is 252 bytes. So, if a driver has allocated a fixed length buffer of less than 252 bytes to store the UTF-16LE representation of the string (or 126 bytes for an ASCII representation of the string) then a buffer overflow could occur.

In one publicly reported Linux vulnerability[24] the driver code looked like this:

```
struct snd_pcm {
      struct snd_card *card;
      struct list_head list;
      int device; /* device number */
      unsigned int info_flags;
      unsigned short dev_class;
      unsigned short dev_subclass;
      char id[64];
      char name[80];
      <cut>
```

Figure 1: Source code from /linux-2.6.38/include/sound/pcm.h

```
dev->pcm->private_data = dev;
strcpy(dev->pcm->name, dev->product_name);
```

Figure 2: Source code from /linux-2.6.38/sound/usb/caiaq/audio.c

In Figure 1 it can clearly be seen that a fixed length buffer of 80 bytes has been allocated and then in Figure 2 the USB “Product Name”, which is stored in a String descriptor is copied into that buffer resulting in a classic stack-based buffer overflow.

5.3 Integer overflows and other length-related bugs

"*An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is too large to be represented within the available storage space*"[25].

If a length field representing some data is one byte long then the maximum (unsigned) length it can represent is 255 bytes. Often memory is dynamically allocated based on a length field that represents the length of some data that will then be copied into the newly allocated memory buffer. If the code adds anything to the data (and accordingly to the length field) value prior to allocating the memory and the length field is already 255 then adding more will cause it to "roll over" past zero. This will result in a small value in the length field representing a much larger set of data. Therefore, a small buffer is allocated and a larger amount of data is subsequently copied into it, resulting in a heap overflow (or potentially a stack overflow where a function such as `strncpy()` is used). In other scenarios that have been observed the code parsing data structures just trusts that the length field value is correct and if it has been set to a value smaller than the actual length of the data then, again, this can result in buffer overflow conditions.

Hub descriptors

An example that highlights this well involves the `bNbrPorts` field in a Hub descriptor (an example Hub descriptor is shown in Table 3). This field (which can hold values 0-255) represents the number of physical downstream ports on a USB hub. Now, bearing in mind that the USB specifications[7] state that the maximum number of physical USB devices that can be connected to a root hub is 127, a device driver programmer might make a dangerous assumption here. This is exactly what happened in the driver code for Apple Mac OS X Lion – by setting the `bNbrPorts` field to the value `0xFF` (255) an attacker could trigger a buffer overflow. This was documented and released in a public advisory[26] by the author.

Configuration descriptors

Many of the USB descriptors contain length fields associated with data contained within them, in fact all descriptors start with a `bLength` field that represents the total length of the descriptor. However, another length example within a Configuration descriptor is the `wTotalLength` field. Configuration descriptors contain other descriptors such as Interface and Endpoint descriptors and the `wTotalLength` field represents the combined length of all these descriptors. Table 4 shows an example Configuration descriptor.

The author has observed vulnerabilities on a number of occasions in this particular descriptor field, where a value of `0xFFFF` (65535) has resulted in a memory corruption vulnerability such as a heap overflow.

Endpoint descriptors

As explained above, Endpoint descriptors are embedded within Configuration descriptors and also contain a field that represents the size of some data – `wMaxPacketSize`. An example Endpoint descriptor is shown in Table 5. In a publicly disclosed advisory[27] the author revealed that setting this field to be greater than a specific value, in this case `0x1125` (4389) would result in a kernel stack overflow on Solaris 11.

A number of other vulnerabilities in different operating systems have been triggered by setting the `wMaxPacketSize` field to the value `0x0000`.

Field	Value	Meaning
bDescLength	9	Descriptor length (including the bLength field)
bDescriptorType	0x29	Hub descriptor
bNbrPorts	4	Number of downstream ports
wHubCharacteristics Logical power switching mode	0	Ganged power switching
wHubCharacteristics Compound device	0	Not Compound device
wHubCharacteristics Over-current protection mode	0	Global over-current protection
wHubCharacteristics TT Think time	3	32 FS bit times
wHubCharacteristics Port Indicators support	1	Port Indicators supported
bPwrOn2PwrGood	100ms	Time from power on till power good
bHubContrCurrent	100mA	Max current required by hub controller
DeviceRemovable[0]	0	Reserved
DeviceRemovable[0]	0	Removable
DeviceRemovable[0]	0	Removable
DeviceRemovable[0]	0	Removable
DeviceRemovable[0]	0	Removable
PortPwrCtrlMask[1]	1	Valid
PortPwrCtrlMask[1]	1	Valid
PortPwrCtrlMask[1]	1	Valid
PortPwrCtrlMask[1]	1	Valid

Table 3: Example Hub descriptor

HID Descriptors

Another vulnerability that was only recently discovered and reported to the vendor relates to a different length field, `wDescriptorLength`, which is present in a HID descriptor (an example is shown in Table 6) and represents the length of another descriptor – a HID report descriptor, which is discussed later in this paper. If the value was set to `0x0000` it resulted in a kernel panic due to a buffer overflow.

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	2	Configuration descriptor
wTotalLength	55	Total combined size of this set of descriptors
bNumInterfaces	2	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)	1	Self-powered
bmAttributes (Remote wakeup)	0	No
bmAttributes (Other bits)	0x80	Valid
bMaxPower	2mA	Maximum current drawn by device in this configuration

Table 4: Example Configuration descriptor

Field	Value	Meaning
bLength	7	Descriptor length (including the bLength field)
bDescriptorType	5	Endpoint descriptor
bEndpointAddress	0x01	Endpoint 1 - OUT
bmAttributes	0x02	Bulk data endpoint
wMaxPacketSize	0x0200	Maximum packet size is 512
bInterval	0xFF	At most one NAK per 255 micro frames

Table 5: Example Endpoint descriptor

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	0x21	HID
bcdHID	0x0110	HID class spec version
bCountryCode	0	Not supported
bNumDescriptors	1	Number of descriptors
bDescriptorType	34	Report
wDescriptorLength	65	Descriptor length

Table 6: Example HID descriptor

Image class data transfers

The next vulnerability relates not to a field within a descriptor, but instead to USB class-specific communication – an Image class response to a `GetDeviceInfo` operation (example data is shown in Table 7). The Image class is used for USB-based cameras to transfer images to a host computer and as can be seen in the example data there are a number of different size fields:

- `Container Length`
- `Operations Supported Array Size`
- `Events Supported Array Size`
- `Device Properties Supported Array Size`

- `Capture Formats Supported Array Size`
- `Image Formats Supported Array Size`

A number of vulnerabilities have been identified in USB drivers where if any of the …`Supported Array Size` fields are set to a value larger than the legitimate value or in some cases to the value `0xFFFF` (65535), the bug is triggered. In some cases the bugs were buffer overflows, in others non-exploitable out-of-bound reads. These bugs have been observed by the author in a popular tablet device and also a well-known games console.

Printer class data transfers

Finally in this section, another example of class-specific communication, this time the printer class and the response to a `GetDeviceId` operation (example data is shown in Table 8). The data is formatted as an IEEE 1284 Device ID string[28] and one of the fields is the `Device ID Length`. A vulnerability discovered by the author in a popular Unix operating system resulted in a buffer overflow if this two-byte field was set to a value greater than a specific number.

Field	Value	Meaning
Container Length	0x000000D3	211 bytes
Container Type	0x0002	Data Block
Operation Code	0x1001	"GetDeviceInfo"
Transaction ID	0x00000001	1
StandardVersion	0x0064	Version 1.00
VendorExtensionID	0x00000006	Microsoft Corporation
VendorExtensionVersion	0x0064	Version 1.00
VendorExtensionDesc	0 chars	
FunctionalMode	0x00	Standard mode
Operations Supported Array Size	0x00000010	16 Operations supported
Operation Supported	0x1001	GetDeviceInfo
Operation Supported	0x1002	OpenSession
Operation Supported	0x1003	CloseSession
Operation Supported	0x1004	GetStorageIDs
Operation Supported	0x1005	GetStorageInfo
Operation Supported	0x1006	GetNumObjects
Operation Supported	0x1007	GetObjectHandles
Operation Supported	0x1008	GetObjectInfo
Operation Supported	0x1009	GetObject
Operation Supported	0x100A	GetThumb
Operation Supported	0x100C	SendObjectInfo
Operation Supported	0x100D	SendObject
Operation Supported	0x1014	GetDevicePropDesc
Operation Supported	0x1015	GetDevicePropValue
Operation Supported	0x1016	SetDevicePropValue
Operation Supported	0x101B	GetPartialObject
Events Supported Array	0x00000004	4 events supported

Size		
Event Supported	0x4004	StoreAdded
Event Supported	0x4005	StoreRemoved
Event Supported	0x4008	DeviceInfoChanged
Event Supported	0x4009	RequestObjectTransfer
Device Properties Supported Array Size	0x00000002	2 properties supported
Device Property supported	0xD406	Unknown property
Device Property supported	0xD407	Unknown property
Capture Formats Supported Array Size	0x00000000	0 formats supported
Image Formats Supported Array Size	0x00000006	6 formats supported
Image Format Supported	0x3001	Association (folder)
Image Format Supported	0x3002	Script
Image Format Supported	0x3006	DPOF
Image Format Supported	0x300D	Unknown image format
Image Format Supported	0x3801	EXIF/JPEG
Image Format Supported	0x380D	TIFF
Manufacturer	9 chars	“Panasonic”
Model	7 chars	“DMC-FS7”
Device version	3 chars	“1.0”
Serial number	31 chars	“0000000000000000000000000000001”

Table 7: Example Image class data

Field	Value	Meaning
wIndex	0x0000	Interface number 0 Alternative setting 0
wValue	0x0000	Configuration Index
Device ID Length	171	IEEE 1284 device ID string length
Device ID element	Key: Value:	MFG: Hewlett-Packard;
Device ID element	Key: Value:	CMD: PJL,PML,PCLXL,POSTSCRIPT,PCL;
Device ID element	Key: Value:	MDL: HP Color LaserJet CP1515n;
Device ID element	Key: Value:	CLS: PRINTER;
Device ID element	Key: Value:	DES: Hewlett-Packard Color LaserJet CP1515n;
Device ID element	Key: Value:	MEM: MEM=55MB;
Device ID element	Key: Value:	COMMENT: RES=600X8;

Table 8: Example Printer class data

5.4 Format string bugs

Format string bugs[29] arise due to the combination of two factors; insecure programming and the use of “dangerous” functions in the “C” programming language. Combine them both and the result is the ability for an attacker to read data from arbitrary memory locations and write (almost) arbitrary data to arbitrary memory locations, therefore potentially executing arbitrary code or otherwise gaining unauthorised control of a program. The attacker just needs to supply a carefully crafted string of format specifiers[30]. It must be noted that if the driver has been compiled with a modern compiler, such as a recent version of Microsoft’s Visual Studio[31] then the “%n” format specifier has been deprecated (and hence format string bugs which write to memory cannot be exploited).

String Descriptors

As format string bugs relate (as their name suggests) to the formatting of string data, the obvious place to test for their presence is String descriptors. A Chrome OS developer[32] recently discovered a USB-based format string vulnerability in X11 by setting the “Device” or “Manufacturer” String descriptors in a HID (Human Interface Device) device to “%n%n%n%n”.

Other text fields

It is not just String descriptors that are potentially vulnerable to format string bugs, as can be seen in Table 8, the Printer class data is full of strings. There are also a number of strings in the Image class data in Table 7. All strings processed by USB drivers are potentially vulnerable to format string bugs if “dangerous” functions have been implemented in an insecure way.

5.5 Logic errors

Logic errors produce unintended or undesired output or other behavior as a result of specific actions or data input. They are very implementation-specific, as the logic of driver code is often very different in different operating systems.

Field	Value
Usage Page (Generic Desktop Controls)	05 01
Usage (Keyboard)	09 06
Collection (Application)	A1 01
Usage Page (Keyboard / Keypad)	05 07
Usage Minimum (224)	19 E0
Usage Maximum (231)	29 E7
Logical Minimum (0)	15 00
Logical Maximum (1)	25 01
Report Count (8)	95 08
Report Size (1)	75 01
Input (Data, Variable, Absolute, Bit Field)	81 02
Report Count (8)	95 08
Report Size (1)	75 01
Input (Constant, Array, Absolute, Bit Field)	81 01
Usage Page (LEDs)	05 08
Usage Minimum (1)	19 01
Usage Maximum (3)	29 03
Report Count (3)	95 03
Report Size (1)	75 01
Output (Data, Variable, Absolute, Bit Field)	91 02
Report Count (1)	95 01
Report Size (5)	75 05
Output (Constant, Array, Absolute, Bit Field)	91 01
Usage Page (Keyboard / Keypad)	05 07
Usage Minimum (0)	19 00
Usage Maximum (255)	2A FF 00
Logical Minimum (0)	15 00
Logical Maximum (255)	26 FF 00
Report Count (6)	95 06
Report Size (8)	75 08
Input (Data, Array, Absolute, Bit Field)	81 00
End Collection	C0

Table 9: Example HID Report descriptor

HID Report descriptors

The HID Report descriptor is a hard-coded array of bytes that describe the device's data packets. This includes: how many packets the device supports, the size of the packets and the purpose of each byte and bit in the packet. An example HID Report descriptor is shown in Table 9.

The HID report descriptor is a notoriously complicated structure and therefore, *"the parser for the Report descriptor represents a significant amount of code"* [33]. This was certainly the case in a HID driver for one major operating system tested by the author where a number of bugs were identified in the parsing of this data structure. However, in that particular example none of the bugs were exploitable. To trigger the bugs the `Usage Page` entries (`0x05`) were replaced with the values `0x81` – `Input()` and `0xB1` – `Feature()`, which should not have been present at that point in the data structure and hence should have generated a handled error. Full details of the HID Report descriptor structure are available in the Device Class Definition for Human Interface Devices (HID) Firmware Specification[25].

All descriptors

The second field in all descriptors is `bDescriptorType`, which represents (as one would guess) the type of the descriptor – there are different values for each e.g. the value `0x02` represents a Configuration descriptor. The author has observed logic errors that have resulted in memory corruption in a number of different USB implementations where this value has been set to `0xff` (255).

6 Conclusions

This paper has described a range of different vulnerabilities that are commonly identified in USB host driver stacks using fifty bugs identified by the author as the primary reference. What this research demonstrates is that bugs and potential security vulnerabilities in USB driver stacks are still relatively common and with the right testing approach and requisite knowledge of where to look the barrier of entry required to identify, if not exploit, is relatively low.

Unfortunately for a number of reasons NCC Group is not able to publish full details of all fifty bugs; however Figure 3 shows the range of different USB fields in which malformed data triggered the bugs identified during this research and also indicates how many instances of each were discovered.

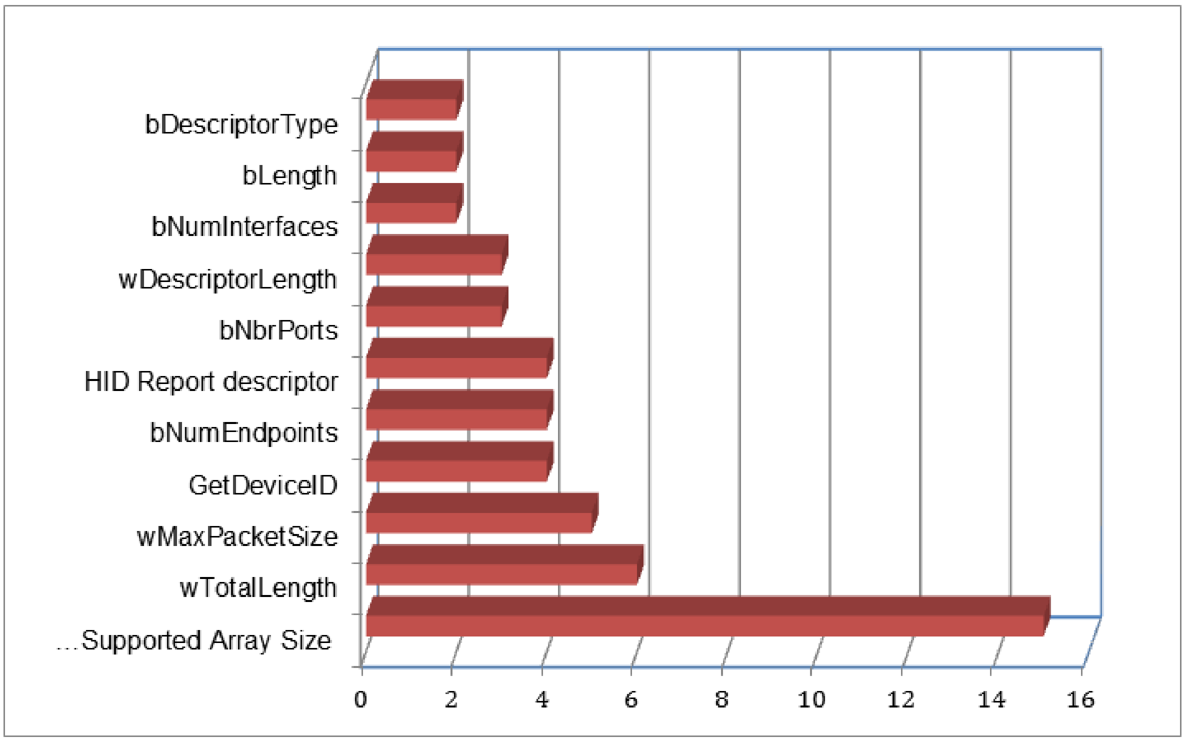

Figure 3: Summary of the 50 USB bugs referenced

Some interesting statistics from the research are as follows:

- The most common class of bug identified was the buffer overflow, which included both stack and heap-based overflows.
- The most common bug was in the processing of Image class data, more specifically length values associated with …`Supported Array Size` fields.
- The most bugs discovered on one operating system were in a well-known games console, closely followed by a well-known Unix-based operating system.

The list of operating systems / products in which USB bugs were identified is as follows:
(Note that not all bugs are vulnerabilities)

- Oracle Solaris
- Google Chromium OS
- Sony PlayStation 3
- Apple Mac OS X
- Apple iOS
- RIM QNX
- Microsoft Windows
- Microsoft Xbox
- Linux

Some of the vendors certainly took USB vulnerabilities more seriously than others. I have mentioned these quotes previously, but they are worth repeating:

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".

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".

With regard to mitigation strategies for USB driver writers, there are a number of best-practice recommendations, which if followed would improve the security of drivers developed in future:

- Where possible use a modern compiler and if security features exist within compiler options then enable them.
- If the driver can be implemented in user mode rather than kernel mode then this option should be taken as part of a defence-in-depth approach. If vulnerabilities are subsequently discovered in the driver then their exploitation is unlikely to result in privilege escalation.
- Never make any assumptions about length fields and other numeric values based on the protocol specifications. Just because you can't physically connect more than 127 devices to a hub doesn't mean that the eight bits that represent the number of downstream ports can't be set to a value greater than 127.
- Never use "dangerous" functions such as `sprintf()`, `strcpy()`,`strcat()` etc. and when using functions that require format specifiers, ensure that those specifiers are hard-coded within the program and cannot be externally influenced in any way.
- Never assume that driver code is less likely to be attacked just because emulating a USB device to launch an attack is more difficult than attacking a network service.
- Ensure that all driver code is tested by security-aware testers using a combination of static source code analysis and black-box fuzz testing approaches.

There are still many vulnerable USB drivers in use, as often they have not been adequately tested, but the information provided in this paper will help USB driver developers and also security researchers and testers to assess them more thoroughly in future.

7 USB Host Testing Checklist

The following tables represent many common fields parsed by USB drivers in which malformed data may result in vulnerabilities being identified:

All classes of USB device		
Descriptor	**Field**	**Bug class**
Device	`bLength`	Buffer overflow
Device	`bDescriptorType`	Logic error
Device	`bMaxPacketSize0`	Buffer overflow
Configuration	`bLength`	Buffer overflow
Configuration	`bDescriptorType`	Logic error
Configuration	`wTotalLength`	Buffer overflow
Configuration	`bNumInterfaces`	Buffer overflow
Configuration -> Interface	`bLength`	Buffer overflow
Configuration -> Interface	`bNumEndpoints`	Buffer overflow
Configuration -> Endpoint	`bLength`	Buffer overflow
Configuration -> Endpoint	`bEndpointAddress`	Logic error
Configuration -> Endpoint	`wMaxPacketSize`	Buffer overflow
String	`bLength`	Buffer overflow
String	`bString`	Buffer overflow Format String

HID class devices		
Descriptor	**Field**	**Bug class**
Configuration -> HID	`bLength`	Buffer overflow
Configuration -> HID	`wDescriptorLength`	Buffer overflow
HID report	Every field	Logic error

Image class devices e.g. Cameras		
Bulk-In data transfer type	**Field**	**Bug class**
DeviceInfo	`Container Length`	Buffer overflow
DeviceInfo	`Container Type`	Logic error
DeviceInfo	`Operation Code`	Logic error
DeviceInfo	`Operations Supported Array Size`	Buffer overflow
DeviceInfo	`Events Supported Array Size`	Buffer overflow
DeviceInfo	`Device Properties Supported Array Size`	Buffer overflow
DeviceInfo	`Capture Formats Supported Array Size`	Buffer overflow

DeviceInfo	`Image Formats Supported Array Size`	Buffer overflow
DeviceInfo	`Manufacturer`	Buffer overflow Format String
DeviceInfo	`Model`	Buffer overflow Format String
DeviceInfo	`Device version`	Buffer overflow Format String
DeviceInfo	`Serial number`	Buffer overflow Format String
StorageIDArray	`Container Length`	Buffer overflow
StorageIDArray	`Container Type`	Logic error
StorageIDArray	`Operation Code`	Logic error
StorageInfo	`Container Length`	Buffer overflow
StorageInfo	`Container Type`	Logic error
StorageInfo	`Operation Code`	Logic error
StorageInfo	`StorageDescription`	Buffer overflow Format String
StorageInfo	`VolumeLabel`	Buffer overflow Format String
ObjectHandleArray	`Container Length`	Buffer overflow
ObjectHandleArray	`Container Type`	Logic error
ObjectHandleArray	`Operation Code`	Logic error
ObjectHandleArray	`Object Handle Array Size`	Buffer overflow
ObjectInfo	`Container Length`	Buffer overflow
ObjectInfo	`Container Type`	Logic error
ObjectInfo	`Operation Code`	Logic error
ObjectInfo	`Filename`	Buffer overflow Format String
ObjectInfo	`CaptureDate`	Buffer overflow Format String
ObjectInfo	`ModificationDate`	Buffer overflow Format String
ObjectInfo	`Keywords`	Buffer overflow Format String

Hub class devices e.g. USB hubs or mass storage devices with embedded hubs		
Descriptor	**Field**	**Bug class**
Hub	`bDescLength`	Buffer overflow
Hub	`bNbrPorts`	Buffer overflow

Printer class devices		
Class request	**Field**	**Bug class**
Device ID string (IEEE 1284)	`Device ID Length`	Buffer overflow

Mass storage class devices		
Class request	**Field**	**Bug class**
CBW Inquiry Response	`Peripheral Device Type`	Logic error
CBW Inquiry Response	`Additional length`	Buffer overflow
CBW Inquiry Response	`Vendor ID`	Buffer overflow Format String
CBW Inquiry Response	`Product ID`	Buffer overflow Format String
CBW Inquiry Response	`Product Revision Level`	Buffer overflow Format String

8 References & further reading

1 - http://en.wikipedia.org/wiki/Black-box_testing
2 - http://media.blackhat.com/bh-us-11/Davis/BH_US_11-Davis_USB_WP.pdf
3 - http://www.ps3news.com/ps3-hacks-jailbreak/ps-jailbreak-ps3-exploit-reverse-engineering-is-detailed/
4 - http://recon.cx/2012/schedule/attachments/57_recon2012-goodspeedbratus.pdf
5 - http://theiphonewiki.com/wiki/index.php?title=Usb_control_msg(0x21,_2)_Exploit
6 - http://www.beyondlogic.org/usbnutshell/usb1.shtml
7 - http://www.usb.org/developers/docs/usb_20.zip
8 - http://www.qemu.org/
9 - https://muelli.cryptobitch.de/paper/2010-usb-fuzzing.pdf
10 - http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Uhooker
11 - http://hackaday.com/2012/07/05/facedancer-board-lets-your-python-programs-pretend-to-be-usb-hardware/
12 - http://msdn.microsoft.com/en-us/library/windows/hardware/jj672841(v=vs.85).aspx
13 - http://www.mqp.com/usb500.htm
14 - http://media.blackhat.com/bh-us-11/Davis/BH_US_11-Davis_USB_Slides.pdf
15 - http://www.freebsd.org/doc/en/books/arch-handbook/usb.html
16 - http://msdn.microsoft.com/en-gb/library/windows/hardware/hh406256(v=vs.85).aspx
17 - http://msdn.microsoft.com/en-gb/library/windows/hardware/hh706187(v=vs.85).aspx
18 - http://msdn.microsoft.com/en-gb/library/windows/hardware/hh706184(v=vs.85).aspx
19 - http://en.wikipedia.org/wiki/Ring_(computer_security)
20 - https://www.owasp.org/index.php/Null-pointer_dereference
21 - http://cwe.mitre.org/data/definitions/125.html
22 - http://en.wikipedia.org/wiki/Stack_buffer_overflow
23 - http://en.wikipedia.org/wiki/Heap_overflow
24 - http://labs.mwrinfosecurity.com/assets/153/mwri_caiaq-usb-drivers-buffer-overflow_2011-03-07.pdf
25 - http://en.wikipedia.org/wiki/Integer_overflow
26 - http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3723
27 - http://www.securityfocus.com/bid/48790/
28 - ftp://ftp.pwg.org/pub/pwg/candidates/cs-pmp1284cmdset10-20100531-5107.2.pdf
29 - http://www.thenewsh.com/~newsham/format-string-attacks.pdf
30 - http://en.wikipedia.org/wiki/Printf_format_string
31 - http://www.microsoft.com/visualstudio/
32 - http://www.outflux.net/blog/archives/2012/05/16/usb-avr-fun/
33 - http://www.usb.org/developers/devclass_docs/HID1_11.pdf