eForensics Magazine

Computer

VOL.2NO.13

USB Forensics

90+ PAGES

PHIL POLSTRA TALKS ABOUT HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

Issue 13/2013 (17) September ISSN 2300-6986

eForensics Magazine
TEAM

Editors:
Artur Inderike
artur.inderike@eforensicsmag.com

Betatesters/Proofreaders:
JohanScholtz, Brett Shavers, Kishore P.V., JAMES FLEIT, Andrew J. Levandoski, Richard C. Leitz Jr., Dr. D.B. Karron, Larry Smith, Olivier Caleff, Tracy Foremantracy, M1ndl3ss, Alex Rams, Robert Vanaman.

Senior Consultant/Publisher:
Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@software.com.pl

Production Director: Andrzej Kuca
andrzej.kuca@software.com.pl

Marketing Director: Joanna Kretowicz
jaonna.kretowicz@eforensicsmag.com

Art Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@software.com.pl

DTP: Ireneusz Pogroszewski

Publisher: Hakin9 Media Sp. z o.o. SK
02-676 Warszawa, ul. Postępu 17D
Phone: 1 917 338 3631
www.eforensicsmag.com

Dear eForensics readers!

I'm very excited sharing our team's progress and direction with you. We are adding more topics of the day to the discovery of digital forensic science and test new technics, strengthened by continuous investigations and suggestions from our authors. We have already discussed computer and network forensics, databases and e-discovery, mobile forensics and malware analysis.
This time we are covering something brand new – USB drive forensics and security.
Our new expert – Philip Polstra – wrote a whole series on this subject which consists of 6 excellent articles. This interesting series also cater for those who have no USB forensics experience. He starts from a very basic level, extends to usable techniques and finishes with an open source tool comparison. I hope everyone will learn something new in this series.
By the way, if you have any suggestion or ideas for the issue – forward your requests to artur.inderike@eforensicsmag.com.
Also, if you would like to discuss an already published article, please feel free to contact us, our experts are always open to your commends, so please, take a minute and send us your opinion about every issue.

Peace, love, unity!
Artur Inderike
eForensics Team

HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

PART 1: USB BASICS

by Dr. Phil Polstra

USB mass storage devices have become the standard for backup and transfer of files. The popularization of this media has led to challenges for forensic specialists used to traditional magnetic media. This first article in a multi-part series will provide a necessary overview of how USB devices work at a low level.

What you will learn:

- Different versions of USB
- How USB devices are identified
- Standard USB device classes
- USB device communication protocols

What you should know:

- A basic understanding of programming in C

In recent years USB mass storage devices using NAND flash storage (also known as thumb drives or flash drives) have replaced magnetic media, such as floppy discs, and optical media, such as CD/DVD, as the standard means for backup and file exchange. The ultimate aim is to understand how to perform forensics on USB mass storage devices. In order to achieve this one must first understand the basics of USB devices. Given the vastness of this topic, it is not feasible to cover everything in one article. This first installment from this series with cover the basics of USB. From there we will move on to learn more about USB mass storage devices. Once the foundations have been set we will cover some advanced topics such as USB write blocking and device impersonation.

BRIEF HISTORY

Up until the early 1990s peripherals were connected to computers via serial connections (RS-232), parallel connections (LPT), or some proprietary method. While RS-232 is a standard, there are several variations in cabling which leads to complication. Furthermore, serials devices have several choices of protocols leading to a potentially non-user-friendly configuration.

In 1996 the first Universal Serial Bus (USB) standard was released. This initial version allowed for plug and play operations with low-speed devices operating at 1.5 Mbps and full-speed devices operating at 12 Mbps. In 1998 some minor revisions and corrections were made and the USB 1.1 standard was released. An improved version, USB 2.0, was released in 2000. The most notable feature of USB 2.0 was the introduction of a new high-speed rate of 480 Mbps. Through USB 2.0 no changes in cabling or other hardware were required.

In 2008 USB 3.0 was introduced. One of the most touted features of USB 3.0 was the introduction of 5.0 Gbps super-speed. The new super-speed came at the cost of adding additional wires, however. USB 3.0 connectors are backwards compatible with USB 2.0. This is accomplished by adding connections which are recessed inside standard USB 2.0 connectors. The difference in connectors is shown in Figure 1 which is from *wikimedia.org*.

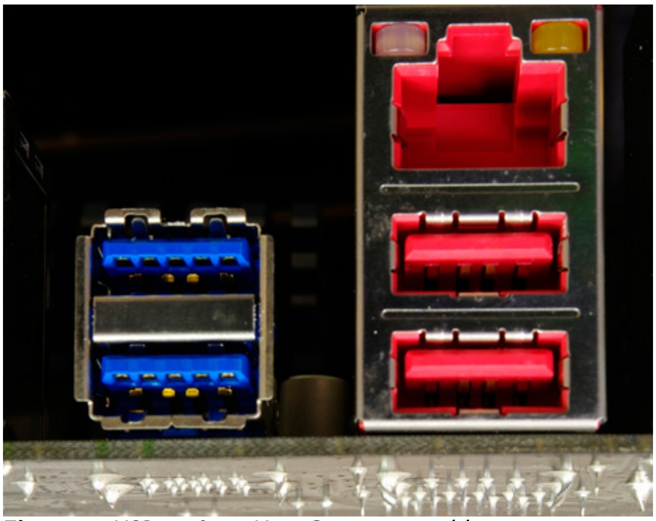


Figure 1. *USB 3.0 & 2.0 Host Connectors – blue connectors are USB 3.0*

HARDWARE

USB uses power wires (5 Volts and ground), and differential signal wires for each communication channel. The use of differential voltage makes USB less susceptible to noise than older standards which measure signals relative to ground. Through USB 2.0 only one signal channel was used. As a result, USB 2.0 connections require only four wires (while some connector types have extra shield or ground wires). USB 3.0 adds two additional super-speed channels which require their own ground bringing the minimum number of wires for a USB 3.0 connection to nine.

Unlike some of the older standards, USB devices are hot-pluggable. As a consequence of this devices must tolerate the application and removal of power without damage. Having learned a lesson from non-universal serial connections, the designers of USB ensured that improperly connecting devices and hosts would be impossible using standard cables. In some cases these standard cables can be up to 16 feet long.

SOFTWARE

From the end user perspective USB is easy. Just plug in a device, wait for the chimes to sound and start using the device. As one might expect, things are a bit more complicated under the covers. There are no settable jumpers or other complications from a user perspective. Through a process known as enumeration a host will discover a newly connected device, determine the speeds it is capable of communicating at, learn what capabilities the device possesses, and what protocols should be used to communicate with the device.

The USB standards define several device classes including Human Interface Device (HID), printer, audio, and mass storage. In many cases developers and users need not worry about special drivers for a particular device that falls into one of the standard classes.

CONNECTING A DEVICE

Connecting a device is a 12 step process. Perhaps that is why working with USB is so addictive. Some of the details will be covered in this article. For the full description I highly recommend *USB Complete: The Developer's Guide (4th ed.)* by Jan Axelson. Here are the 12 steps:

- The device is connected and in most cases receives power from the host.
- The hub detects that a new device has been connected.
- The host (PC) is informed of the new device.
- The hub determines device speed capability as indicated by location of pull-up resistors. This is only a choice of low or full speed as higher speeds are only available after the device is fully enumerated.
- The hub resets the device so it can begin communicating with it in a less generic manner.
- The host determines if device is capable of high speed by sending a series of pulses known as chirps.
- The hub establishes a signal path.
- The host requests a descriptor (more about descriptors later) from the device to determine max packet size to be used.
- The host assigns an address to the device so that communication may commence. Addresses are required because more than one device may operate on a single USB bus.
- The host learns the devices capabilities by asking for a set of structures that describe the device known as descriptors.
- The host assigns and loads an appropriate device driver (INF file under Windows). In many cases a driver included with the operating system is loaded if the device is from a standard USB device class.
- The device driver selects a configuration. Devices are not required to support more than one configuration, but multiple configuration devices are not uncommon.

ENDPOINTS

All communication between USB devices and hosts is via endpoints. Endpoints are an abstracted unidirectional communications pipe. All packet fragmentation, handshaking, etc. is handled by the hardware in most cases. The direction of each endpoint is specified relative to the host. For example, an in endpoint receives data from the device in to the host. The high bit of the endpoint address is used to indicate direction where 1 and 0 indicate in and out, respectively. There are four types of endpoints: control, bulk transport, interrupt, and isochronous.

All devices must have at least one control endpoint. Devices with more than one control endpoint are extremely rare. Often the control endpoint is referred to as endpoint 0 or EP0. This is the endpoint used to determine a devices capabilities. For many devices this is the primary mechanism for communicating with a host. As will be seen in later articles, mass storage devices are an exception.

The control endpoint is used to handle standard requests from the host. The requests include things such as setting or getting and address, returning descriptors, setting power levels and modes, and providing status. The device may also respond to standard USB class (HID, mass storage, etc.) requests. Additionally the USB standard allows vendors to add additional requests to be handled by their devices, but this is rarely done as it would require the vendor to provide a proprietary driver for every supported operating system.

Transfers on control endpoints can involve up to three stages: setup, data, and status. During the required setup stage a setup token and then an eight byte request is sent to the device. The first byte of the request is a bitmap telling the type of request and recipient (device, interface, endpoint). The remaining bytes in the request are parameters for the request and response. If a valid setup request is received, the device responds with an acknowledgement (ACK) packet. During the optional data stage the requested information is sent to the host. In the status stage a zero length data packet is sent as an acknowledgement of success.

Interrupt endpoints are used for communicating with low-speed devices such as keyboards. Interrupt endpoints are useful for avoiding polling and busy waits. The lower speeds typically used in such devices also allow for longer cables. These are not used in mass storage devices.

Isochronous endpoints provide a guaranteed amount of band width. This is useful for time-critical applications such as streaming media. While on the face of things it might seem like isochronous endpoints would be ideal for transferring large amounts of data, this is not the case. As it turns out, the overhead of using isochronous endpoints decreases the total throughput.

Bulk transport or more simply bulk endpoints are used to transfer large quantities of information efficiently. Bulk endpoints have no latency guarantees (unlike isochronous endpoints), but they have the highest throughput on an idle bus. Bulk transfers are superseded by all other transfer types which means they are not the best choice on a busy bus. Low-speed (1.5 Mbps) endpoints may not be used for bulk transfers. These are used extensively in mass storage devices.

DESCRIPTORS

As previously mentioned, descriptors are structures used to describe devices and their capabilities. Descriptors have a standard format. The first byte gives the length of the descriptor (so a host

Table 1. *Device descriptors*

Offset	Field	Size	Value	Description
0	bLength	1	Number	18 bytes
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	0x200
4	bDeviceClass	1	Class	Class Code
5	bDeviceSubClass	1	SubClass	Subclass Code
6	bDeviceProtocol	1	Protocol	Protocol Code
7	bMaxPacketSize	1	Number	Maxi Packet Size EP0
8	idVendor	2	ID	Vendor ID
10	idProduct	2	ID	Product ID
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manu Descriptor
15	iProduct	1	Index	Index of Prod Descriptor
16	iSerialNumber	1	Index	Index of SN Descriptor
17	bNumConfigurations	1	Integer	Num Configurations

will know when it has received the entire descriptor). The second byte tells the type of descriptor and the remaining bytes are the descriptor itself. The standard descriptor types are device, configuration, interface, endpoint, and string.

A device descriptor provides basic information about a device. Some of the more interesting information it provides includes the vendor ID, product ID, and the device class. Manufacturers of USB devices must purchase a vendor ID. Product IDs are set by the manufacturer. A class code of zero indicates that the device class is specified in another (interface) descriptor. A zero class code is quite common and is the norm for mass storage devices. Index values in any descriptors refer to the string descriptor number for that value. The format for a device descriptor is provided in Table 1.

Configuration descriptors describe the power needs, number of interfaces, etc. for each supported configuration of a USB device. Devices are only required to support one configuration. Because the configuration descriptor contains the total bytes in all subordinate descriptors a request is normally made for nine bytes of the configuration descriptor followed by a request for the configuration descriptor and all subordinate interface and endpoint descriptors. Devices should not request more power than needed as requests for more power than a host can provide result in a failed device enumeration. The format for configuration descriptors is provided in Table 2. An interface descriptor describes how to communicate to a device. For devices with a class code of zero in the configuration descriptor the device class is provided in the interface descriptor(s). Many devices present themselves as composite devices (devices with more than one device class). A camera is a good example of a composite device as it is both a camera and a mass storage device in most cases. The subclass code and protocol code are defined for each device class and are optional. The format for interface descriptors is presented in Table 3.

Each non-control endpoint is described by and endpoint descriptor. The descriptor provides the direction, type, and number of each endpoint. The format for endpoint descriptors is provided in Table 4.

String descriptors are used to allow devices to return Unicode text strings. String descriptor numbers (indexes) can be found in several other descriptors. String descriptor 0 is a special case that returns the languages the device supports. The most commonly supported language is 0x0409 US English regardless of country of origin. The format for string descriptors appears in Table 5.

Table 2. *Configuration descriptors*

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size in Bytes
1	bDescriptorType	1	Constant	0x02
2	wTotalLength	2	Number	Total data returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Configuration number
6	iConfiguration	1	Index	String Descriptor
7	bmAttributes	1	Bitmap	b7 Reserved, set to 1. b6 Self Powered b5 Remote Wakeup b4..0 Reserved 0.
8	bMaxPower	1	mA	Max Power in mA/2

Table 3. *Interface descriptors*

Offset	Field	Size	Value	Description
0	bLength	1	Number	9 Bytes
1	bDescriptorType	1	Constant	0x04
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints used
5	bInterfaceClass	1	Class	Class Code
6	bInterfaceSubClass	1	SubClass	Subclass Code
7	bInterfaceProtocol	1	Protocol	Protocol Code
8	iInterface	1	Index	Index of String Descriptor

ON THE WEB

- *http://www.usb.org/developers/docs/* – The official source for all things USB
- *http://lvr.com/usbc.htm* – USB expert Jan Axelson's main USB page
- *http://www.youtube.com/watch?v=3D9uGCvtoFo* – Phil's DEFCON XX Talk on USB impersonation
- *http://www.youtube.com/watch?v=CIVGzG0W-DM* – Phil's 44CON 2011 Talk on USB flash drive forensics
- *http://www.instructables.com/id/Cheap-and-Effective-USB-Write-Blocker/* – Phil's instructable on how to build a USB write blocker based on his BlackHat EU 2012 presentation
- *http://www.concise-courses.com/infosec/20130404/* – Phil's Hacker Hotshots presentation on building a USB impersonator
- *http://ppolstra.blogspot.com* – Phil's blog
- *http://twitter.com/ppolstra* – Phil's Twitter page (@ppolstra)

SUMMARY

At this point it might be helpful to reiterate all that happens when we connect our USB device. First some basic information is exchanged to determine what capabilities a device possesses. The device is then reset and it is further probed to determine whether or not it is capable of of supporting high-speed communications. Using a control endpoint the host requests a series of descriptors in the following order: device, configuration(s), interface(s), endpoint(s), and optionally string(s). Finally, now that the host knows what type of device it is dealing with a device driver can be loaded and the device may be used. While much has been covered in this article, we have merely scratched the surface.

USB is a huge topic, but hopefully this introduction and pointers to more information contained within are sufficient to provide a basic understanding of the topic. In the next installment of this series we will take an in depth look at the workings of USB mass storage. From there we will discuss various forensic techniques and devices that can be developed when working with USB mass storage devices. Feel free to contact Phil on twitter, on his blog, or via e-mail with any questions related to this series.

ABOUT THE AUTHOR

Dr. Phil Polstra is currently an Associate Professor and Hacker in Residence at a private university in the Midwestern United States. He also teaches online computer security courses for a private university in Tempe, Arizona. His primary research focus over the last few years has been on the use of small, low-powered devices for forensics and penetration testing. As part of this work, he has developed his own custom pentesting Linux distribution that runs on the BeagleBoard family of ARM-based computers. Phil has presented his work on USB forensics and pentesting with small computers at several conferences around the world including BlackHat, DEFCON, 44CON, GrrCON, ForenSecure, and B-sides, to name a few. When not working he can be found spending time with family, hacking electronics, picking locks, flying, or building airplanes.

Table 4. *Endpoint descriptors*

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	b0..3 Endpoint Number. b4..6 Reserved. Set to Zero b7 Direction 0 = Out, 1 = In
3	bmAttributes	1	Bitmap	b0..1 Transfer Type 10 = Bulk b2..7 are reserved. I
4	wMaxPacketSize	2	Number	Maximum Packet Size
6	bInterval	1	Number	Interval for polling endpoint data

Table 5. *String descriptors*

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	b0..3 Endpoint Number. b4..6 Reserved. Set to Zero b7 Direction 0 = Out, 1 = In
3	bmAttributes	1	Bitmap	b0..1 Transfer Type 10 = Bulk b2..7 are reserved. I
4	wMaxPacketSize	2	Number	Maximum Packet Size
6	bInterval	1	Number	Interval for polling endpoint data

HOW TO FORENSIC USB DEVICES

CHARACTERISTICS OF USB DEVICES ON WINDOWS ENVIRONMENT AND CARES TO IMAGE ACQUISITION

by Carlos Castro

In this article there is a description of difficulties added to computer forensic by the diversity of devices that were included at investigation scope after the creation and popularization of USB interface. The principal focus will be the investigation at Windows environment, describing some characteristics of this operational system, how it deals with USB devices and the attention points for the forensic image acquisition.

What you will learn:

- How to identify traces of USB device usage in a Windows computer
- How Windows deals with USB devices
- Structure of disk storage units
- What is image acquisition and why forensic needs it

What you should know:

- What is the Windows Registry?
- How to read registry contents using the RegEdit utility
- Basic knowledge of Linux operation

Currently the diversity of devices that could be connected to a computer is huge and has a constant growth, challenging the most creative minds and the creation of USB (Universal Serial Bus) interface, has a great responsibility in this tendency. USB was created by a consortium of big IT companies that were searching for a cheaper and more flexible serial interface for their devices. The consortium launched the version 0.7 in 1984, but began the commercialization in 1998 with version 1.1.

When the subject is computer forensics, it is necessary for a wide vision that takes into account the various possibilities of an investigation scope understanding; that many others devices could be connected to the computer that are at the beginning, could be considered the principal and unique focus of an investigation. So, it is not possible to ignore pen-drives, memory cards connected by USB adapters, removable hard disks, pens, TVs, video games and anything else that is plugged in. The article will show how to identify that a USB device was plugged into Windows, the cares when connecting devices to Windows and how to acquire an image using Linux. The examples of this article were collected from a notebook with a 1 TB removable Samsung hard disk and a 16 GB pen-drive.

USB CHARACTERISTICS

As specified by the USB Org, every USB device must have a unique code identifier based in three fields: *VendorId, ProductID* and *SerialString*. Nevertheless, Windows uses *VendorId, ProductID* and *BcdDevice* (revision number) to compose registry keys that should be searched in the Registry database to ensure the usage of the USB device. The USB field's *bInterfaceClass*, *bInterfaceSubClass* and *bInterfaceProtocol* also influence how Windows deals with the device and its driver. At USB Org site (usb.org) there are complete descriptions for all of these fields.

The best way to verify if a USB device was connected to a Windows computer is searching the Registry by the USB unique identifier in the trees described below. In Figure1 there is the content of these fields for a pen-drive and in Figure 2 the contents for the removable HD, both retrieved by the USBVIEW utility.

REGISTRY ENTRY FOR THE USB DEVICE

These are some Registry keys that identify the usage of the USB device (Samsung 1TB HDD) at the computer:

- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\usbflags\064EF2030211
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\DeviceClasses\{6994AD05-93EF-11D0-A3CC-00A0C9223196}\##?#USB#VID_064E&PID_F203&MI_00#6&1ADBB508&0&0000#{6994ad05-93ef-11d0-a3cc-00a0c9223196}

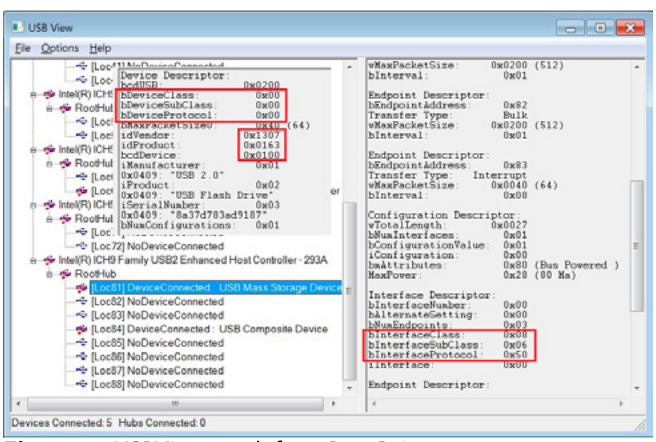


Figure 1. *USBView result for a Pen-Drive*

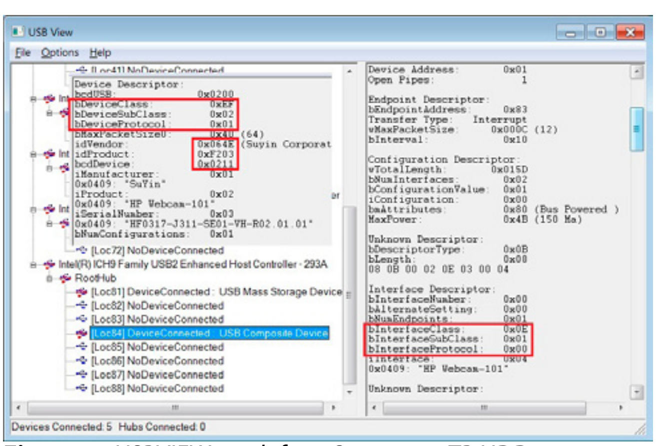


Figure 2. *USBVIEW result for a Samsung 1TB HDD*

- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Enum\USB\VID_064E&PID_F203&MI_00\6&1adbb508&0&0000

FIRST CARE: WHAT HAPPENS WHEN WE PLUG A USB DEVICE INTO WINDOWS

When we connect a USB device into Windows, beyond inserting entries into the Registry database, Windows writes two hidden files into the device. The first file refers to the recycle bin and receives a name according to the Windows version, could be called: RECYCLED, RECYCLER or `$Recycle.`Bin. The second one refers to the point of restoring used by the system function "System Restore" and is called "System Volume Information" however, the USB devices are not treated the same way by Windows. When pen-drives are connected, these files are not written automatically, but they are written to the removable hard disk (Figure 5 and Figure 6). When the subject is forensics, the evidence preservation is fundamental and any change on it

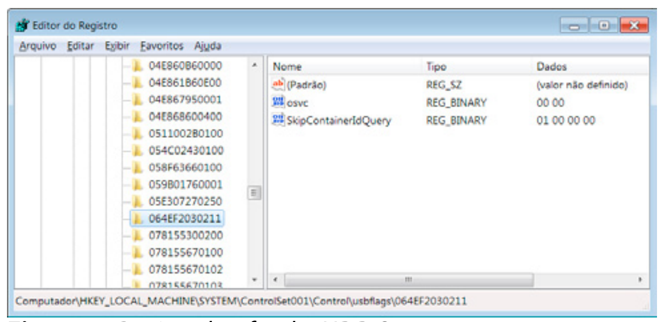


Figure 3. *Registry key for the HDD Samsung*

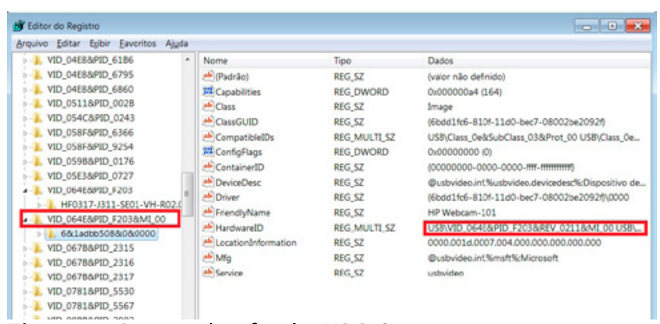


Figure 4. *Registry key for the HDD Samsung*

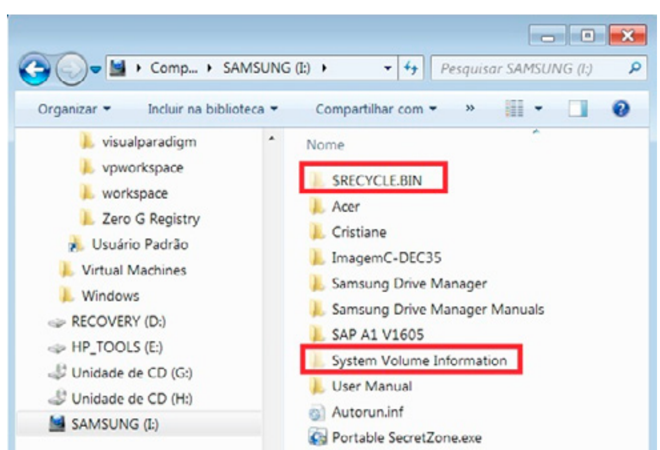


Figure 5. *Samsung HDD File System*

must be avoided to not invalidate the proof; therefore, any writes to the evidence are prohibited, even with Windows files. As this is a Windows exclusive behavior the recommendation is to connect the device to a Linux computer to make an Image of the device, so the investigation will be done using only the Image (always). In the Forensics work model, this phase of the investigation is where the image is created, it is called Acquire and could be done using some free tools like Helix and FTK Imager. There are also some software and hardware solutions to perform image acquisition.

SECOND CARE: WHY A FORENSIC IMAGE AND NOT JUST A COPY ON ACQUISITION?

Image and Copy produce results extremely different and for forensic purposes only image must be used. To know why the image usage is an obligation, lets understand how data is stored on disks by the operating system (remember that even USB Flash Drives are managed like normal disks by a specific software emulation).

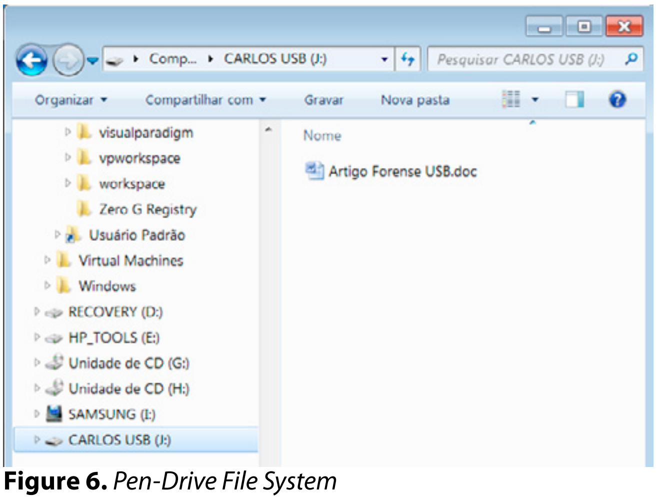


Figure 6. *Pen-Drive File System*

The storage of data on devices is done by the operating system, which uses file structures that occupies storage units called Clusters. Every file access, reads or writes data of a cluster or a set of them. The Cluster is a logical unit that organizes the disk by the operating system and its size differs depending on operating system version, file system type and the size of disk in use (Table1). In this example, for both devices (1 TB HDD and a 16 GB PenDrive), formatted as Windows 7 NTFS, the cluster size will be 4Kb.

Sector is the other storage unit that must be known. Unlike Clusters, sectors are physical storage units, the smallest physical storage unit on a disk almost always has 512 bytes (0.5 KB) in size. So, in a 1 TB device formatted with NTFS, where the cluster size is 4KB, there are 8 sectors per cluster (4096 / 512 = 8).

When the operating system tries to save a file, it will use the necessary number of clusters to save the file. To save a file with 453KBs size (463.872bytes) it will need to use 114 clusters of 4KB (463.872 / 4096 = 113.25) but, the last one will not be completely used. As this last cluster is already partially used by a file, no other file could be saved at this remained space (Figure 7).

Imagine what happens after some months of usage with many file creations, deletions, and expansions, the disk will have much unused space. Its continuous disk activity probably generates unused spaces that could have been used by other files in the past, having parts of those deleted files. Now, with a forensic thinking, imagine how much evidence could be retrieved from this space. Many forensic tools try to recover information directly from this unused or unallocated space.

Coming back to acquire, lets conclude why an image and not a copy. When a copy runs it considers only the file entries addressed in the file system

Table 1. *Cluster Size According to OS, File System and Device Size*

		NTFS		FAT32		exFAT
Volume Size		Windows NT 4.0	Windows 7, Server 2008, Vista, Server 2003, XP and 2000	Windows NT 4.0	Windows 7, Server 2008, Vista, Server 2003, XP and 2000	Windows 7, Server 2008, Vista, Server 2003, XP and 2000
16 GB	Cluster Size	4 KB	4 KB	8 KB	8 KB	32 KB
1 TB		4 KB	4 KB	Not Supported	Not Supported	128 KB

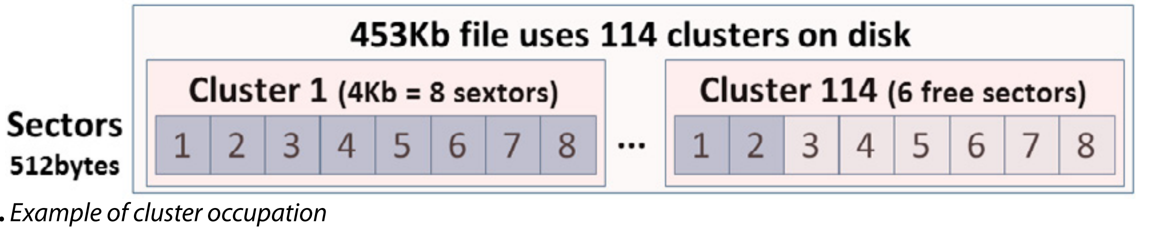


Figure 7. *Example of cluster occupation*

directory, copying only cluster used by existing files however images, copies all the sectors contents giving the possibility to retrieve data from the unused space. So the result of a forensic analysis from evidence reproduced by a copy or by an image process, will be completely different. In the following example a copy only retrieves 456 KB (unique file size) while the image retrieves all 16 GB of the device.

BEGINNING THE ACQUISITION USING LINUX

So, let´s go a little bit further, let´s create an Image of the Pen Drive. As has been said, the acquisition should not be made from Windows, because it will compromise the proof, so the acquisition will be made from Linux.

Even using Linux, there are many ways to create an image (exact sector by sector copy) using free tools like FTK Imager, Helix, etc., but the article will show an example using only Linux commands. The Linux utility to be used must ignore the occupation of the disk, it has to copy every sector regardless of their occupation, so the choice will be the DD utility (something like Data Description). Data dump.

Be careful with DD usage to not lose data. The basic syntax is `dd if=input of=output` and you can specify devices or files as Input and Output.

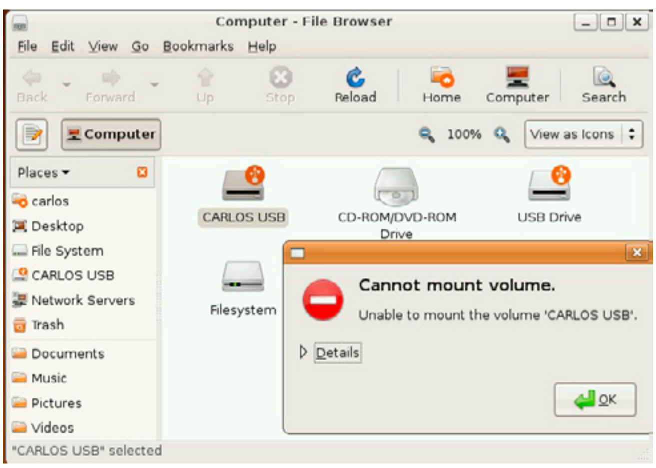


Figure 8. *Menu Places / Computer. List disk devices*

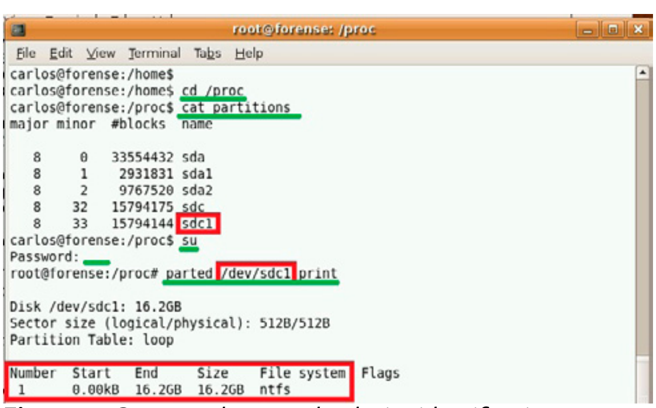


Figure 9. *Commands to get the device identification*

To make an image of the pen drive saving it as a file on the computer hard disk, follow the steps:

- Start Linux
- Plug the USB Flash Memory (pen drive) in the computer
- Open a Terminal (Menu System / Terminal)
- Discover how Linux is identifying the pen drive. If there is some problem to mount the device (Figure 7), proceed like showed in Figure 8.
- Assuming that Linux named your device (pen drive) as `/dev/sdc1`, run DD utility like this: `dd if=/dev/sdc1 of=image _ 001.dd`. The image will be created at default home directory. To save to another USB device, like the HDD Samsung, assuming that Linux recognizes it like `/dev/sdc2`, run DD like this: `dd if=/dev/sdc1 of=/dev/sdc2/image _ 002.dd`.

Obs: Is not necessary to have Linux installed in your computer, a bootable Linux CDROM could be used, but in this case, the Image could not be saved onto the Linux hard disk partition. In this example Linux Ubuntu is running and installed in a local virtual machine where the Linux hard disk partition is writable.

As shown in the following figure, the Pen Drive only has a unique 454 KB file but, the image that will be generated from it, will have 16 GB because all of the Pen Drive sectors must be copied.

To make the image the native Linux DD utility was used. There is a possibility that there is not enough privileges to access devices at Linux. If it happens, login as ROOT or use the SU command. Observe that the DD is executing in the terminal

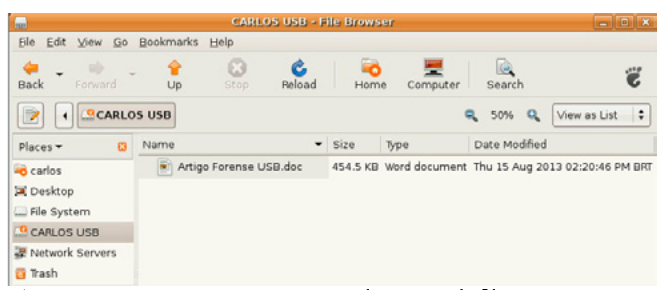


Figure 10. *Pen-Drive Content (only a 454Kb file)*

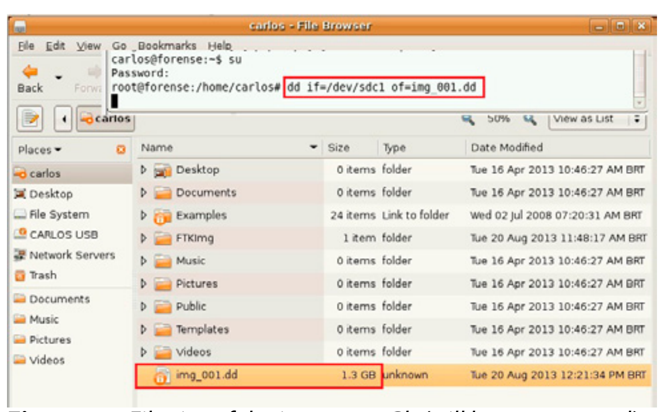


Figure 11. *File size of the image, 1.3Gb (still been processed)*

and the image file (img_001.dd) that is still processing, already has 1.3GB. Why? Because DD is copying all sectors of the Pen Drive not just its occupied sectors (Figure 11).

THIRD CARE: GUARANTEE IMAGE INTEGRITY

DD finished its job and the work now could continue using the image, so the proof will be documented and stored in the drawer. There is still a question to answer: Does this image correct? It has exactly the same data (and metadata) of the original device? To answer this question there is a technique based on HASH algorithms that give us the answer. This algorithm evaluates all the data in a string, file, disk, etc and always produces the same sequence code (HASH) if the data gave to him is the same. If just one byte is changed in a string, file or disk the sequence code returned will be another one. So if the Pen Drive and the Image are identical, the HASH algorithm will always return the same sequence code for then.

Many HASH algorithms were produced, but for more security purposes, use at least the SHA-1. Executing the HASH algorithm over the Pen Drive and over its image using the SHA1SUM Linux utility, the same sequence code above was returned. This behavior ensures that both have exactly the same content (Figure 12).

FOURTH CARE: CUSTODY – REGISTRY THE EVIDENCE ACQUISITION

The acquire phase must not be finalized without generating the "Chain of Custody" where all evidences will be documented. If some questioning

```
488afdf6384ebcda377d6771879ad52e2570b7d8

root@forense:/home/carlos# sha1sum img_001.dd
488afdf6384ebcda377d6771879ad52e2570b7d8  img_001.dd
root@forense:/home/carlos# sha1sum /dev/sdc1
488afdf6384ebcda377d6771879ad52e2570b7d8  /dev/sdc1
root@forense:/home/carlos#
```

Figure 12. *HASH results executed over pendrive (/dev/sdc1) and its image (img_001.dd)*

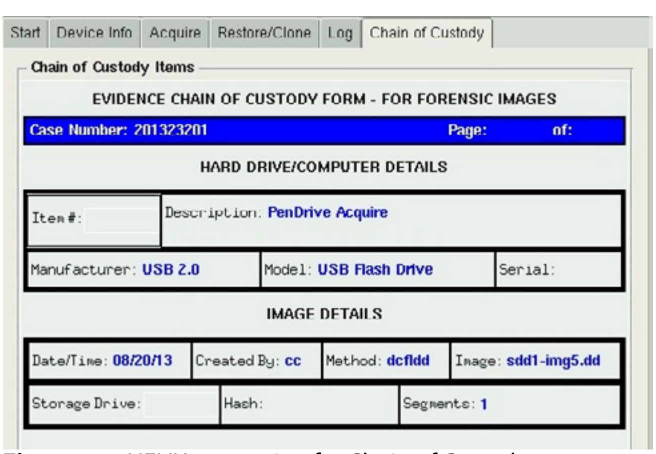


Figure 13. *HELIX suggestion for Chain of Custody*

ON THE WEB

- *http://digital.ni.com/public.nsf/ad0f282819902a1986256f79005462b1/335a90747734097886257070006415b9/$FILE/usbview.zip*
- *http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx*
- *http://msdn.microsoft.com/en-us/library/windows/hardware/jj649944(v=vs.85).aspx*
- *http://support.microsoft.com/kb/140365/pt-br*
- *http://www.ntfs.com/hard-disk-basics.htm#Sectors%20and%20Clusters*

appear about the investigation this document will be very important to show that all the correct procedures were followed and the evidence and its image are exactly the same. It is also important to know who and when the procedure has been executed (Figure 13).

CONCLUSION

The article presents how to identify that a USB device was connect to a Windows computer, the care that must be taken to not invalidate the proof. As a final tip, if just the image was given for the investigation and there is no way to get the device identifier (*VendorId* + *ProductId* + *SerialString*) to search the registry for evidences of device usage, try to find files in the device (pen-drive) with the same SHA-1 HASH (identical files) of those on Windows computer file system. This is also a type of proof that the pen-drive was used in the computer.

The forensic work must be done with patience and attention to each detail... Always preserve the evidence and the "crime" scene.

ABOUT THE AUTOR

I have been working with computers since 1985, acting as support technician, developer, data administrator, project manager, and forensic specialist. IT and security are completely integrated and stimulant marriage has many interesting developments. My complete profile can be found at http://br.linkedin.com/pub/carlos-eduardo-motta-de-castro/1a/68/759/.

HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

PART 2: UNDERSTANDING USB MASS STORAGE DEVICES

by Dr. Phil Polstra

USB mass storage devices have become the standard for backup and transfer of files. The popularization of this media has led to challenges for forensic specialists trying to find data on fixed memory storage media instead of traditional magnetic media. This article in a multi-part series will provide readers with a basic understanding of how USB mass storage devices work at a low level.

What you will learn:

- USB mass storage communications protocols
- Common USB mass storage SCSI commands
- Filesystems used in USB mass storage devices
- Limitations of NAND flash memory used in USB flash drives

What you should know:

- A basic understanding of programming in C
- A basics understanding of USB (possibly from the first part in this series)

In recent years USB mass storage devices using NAND flash storage (also known as thumb drives or flash drives) have replaced magnetic media, such as floppy discs, and optical media, such as CD/DVD, as the standard means for backup and file exchange. This article completes our coverage of USB fundamentals by presenting the basics of USB mass storage devices.

HARDWARE

While hard drives and floppy discs store information magnetically on media which is rotated under a magnetic read/write head, flash drives use NAND flash memory chips. NAND flash memory is reasonably compact and straightforward to access, however it does have some limitations, which we will be covered later.

A typical USB flash drive consists of a NAND flash chip, specialized microcontroller, and supporting electronics. The most common supporting electronics include power regulators (to drop the 5 Volts supplied by USB to 3.3 or 1.8 Volts used by the memory and microcontroller), oscillator crystals (12 MHz is the most common frequency), and, in some cases, status LEDs. Figure 1 shows a typical flash drive. The larger rectangular chip is NAND flash memory. The smaller square chip is the microcontroller. Some flash drives may be more integrated than the one shown in Figure 1, but they all work the same. Highly integrated drives in which everything is on a single chip (including the 4 contacts) that fits inside the USB connector are not uncommon.

Figure 1. *Typical USB Flash Drive*

NAND FLASH LIMITATIONS

USB NAND flash memory has revolutionized data storage, but it is not without its limitations. A well-known limitation of NAND flash storage is that it has limited write cycles. After as memory block has been overwritten too many times it becomes unreliable. A typical mean writes before failure is 10,000 cycles. High quality chips may have mean failures after 100,000 writes. While this may sound like a large number of writes, the way that writes are performed tends to result in more writes than are absolutely necessary.

Writes to NAND flash are performed one block at a time. Typical blocks sizes are 512, 2048, 4096, and 16,384 bytes. What this means is that in order to change a single bit the entire block must be read into a buffer, the bit changed, the block erased, and then the entire block re-written. Most controllers will implement some form of wear leveling, in which the memory blocks are dynamically mapped, in order to improve the life of a flash drive. This wear leveling may be performed on the flash chip itself in some cases. Even with wear leveling in place, if writes are requested one byte at a time by the operating system will result in premature failures.

Write speeds of flash drives vary widely. Generally speaking, larger and more expensive flash drives tend to have higher write speeds than their more affordable counterparts. Some of the cheaper devices have write speeds in the 1 MB/s range while 15 MB/s might be possible with high-end drives. Toshiba has recently developed a NAND flash chip they claim can sustain a write speed of 25 MB/s. Even the Toshiba chip is not capable of writes at even half of the USB 2.0 high-speed transfer rate of 480 Mbps (60 MB/s). Read speeds are generally higher, but still well under the maximum USB 2.0 rate.

The mechanism for recovering information from a damaged flash drive depends on the device's construction. NAND flash chips may be unsoldered from most devices with the exception of the completely integrated design described earlier. After chips have been removed they are most easily read by inserting them into a chip test socket, which has been wired to the appropriate electronics (which may be another flash drive with the flash chip removed). Alternatively, chips may be soldered onto a functional flash drive, which utilizes the same style chip. Some devices feature JTAG interfaces, which permit data recovery without messy unsoldering and/or soldering. Data recovery from completely integrated flash drives may be impossible without specialized equipment.

As with hard drives, flash drives rarely store as much information as their size would indicate. Generally speaking, flash drives use a minimum of 1/32 of their capacity for error correction codes (ECC). For example, a 512 byte block typically consumes 528 bytes of memory (512 bytes data + 16 bytes ECC). Additionally, some controllers use a portion of the flash memory whereas others have internal flash storage. As a result, care should be taken when performing forensic copies from a flash drive to another flash drive of a different brand. Even in the case of identical flash drives; the target drive may not have sufficient capacity to complete the copy if some of the blocks have been marked as bad. Use of an oversized and/or brand new target drive might alleviate this difficulty in many situations.

FLASH DRIVE PRESENTATION

Nearly all flash drives present themselves as SCSI hard drives when connected to a computer. The sector on these pseudo hard drives are typically 512, 2048, or 4096 bytes, with 512 bytes being the most common size in all but the largest drives. Often the devices support a reduced SCSI instruction set, as the full set of SCSI commands doesn't make sense for a memory-based device.

Just as hard drives can be organized into partitions, flash drives may be partitioned using standard operating system tools. Hard drive partitions are referred to as logical units (LU). Most drives come preformatted as one LU. The partitions are assigned logical unit numbers (LUN) starting at zero. Some operating systems (in particular older versions of Windows) do not recognize LUNs above zero. As a result, data can be hidden in upper LUNs from users of outdated operating systems.

Options abound for filesystems to be used on flash drives. In addition to the normal filesystems used on hard drives, numerous filesystems that are optimized for flash memory are available. At least, they are available to Linux users, as in typical fashion, Windows users are left out. These specialized filesystems include the TrueFFS, ExtremeFFS, Journaling Flash File System (JFFS), and Yet Another FFS (YAFFS), among others.

Most flash drives are preformatted as a single FAT or FAT32 partition. Drives formatted with NTFS are not uncommon with larger drives.

TALKING TO MASS STORAGE DEVICES

Most USB devices use control endpoints for all but data transfer. This is not the case with mass storage devices. Mass storage devices use control endpoints primarily during the initial enumeration process and then use bulk endpoints for all of the real work.

Communication over the bulk endpoints consists of three phases: command block wrapper (CBW), data transport (optional), and command status wrapper (CSW). Commands are sent in a in a command block (CB) which is wrapped inside the aptly named CBW. If the command requires the exchange of data, bulk endpoints are used to transfer data in the data transport phase. All commands are terminated by the device sending a CSW.

COMMAND BLOCK WRAPPERS

The format for a command block wrapper is presented in Listing 1. The CBW begins with the signature "USBC" or 0x43425355 in hexadecimal. The tag value is used to associate the CBW with the CSW. The data transfer length varies by command and is zero in some cases. The flags byte is all zeros with the exception of the high bit which is used to indicate transfer direction with 1 indicating in (out of the device and in to the host) and 0 indicating out (in to the device). The command block length will vary from 6-16 bytes (3 high bits will

Listing 1. *Command block wrapper structure*

```
typedef struct _USB_MSI_CBW {
   unsigned long dCBWSignature; //0x43425355 "USBC"
   unsigned long dCBWTag; // associates CBW with CSW response
   unsigned long dCBWDataTransferLength; // bytes to send or receive
   unsigned char bCBWFlags; // bit 7 0=OUT, 1=IN all others zero
   unsigned char bCBWLUN; // logical unit number (usually zero)
   unsigned char bCBWCBLength; // 3 hi bits zero, rest bytes in CB
   unsigned char bCBWCB[16]; // the actual command block (>= 6 bytes)
} USB_MSI_CBW;
```

Listing 2. *Format unit command block*

```
typedef struct _CB_FORMAT_UNIT {
   unsigned char OperationCode; //must be 0x04
   unsigned char LUN:3; // logical unit number (usually zero)
   unsigned char FmtData:1; // if 1, extra parameters follow command
   unsigned char CmpLst:1; // if 0, partial list of defects, 1, complete
   unsigned char DefectListFormat:3; //000 = 32-bit LBAs
   unsigned char VendorSpecific; //vendor specific code
   unsigned short Interleave; //0x0000 = use vendor default
   unsigned char Control;
} CB_FORMAT_UNIT;
```

Listing 3. *Read(10) command block*

```
typedef struct _CB_READ10 {
   unsigned char OperationCode; //must be 0x28
   unsigned char RelativeAddress:1; // normally 0
   unsigned char Resv:2;
   unsigned char FUA:1; // 1=force unit access, don't use cache
   unsigned char DPO:1; // 1=disable page out
   unsigned char LUN:3; //logical unit number
   unsigned long LBA; //logical block address (sector number)
   unsigned char Reserved;
   unsigned short TransferLength;
   unsigned char Control;
} CB_READ10;
```

always be zero). Unused bytes in the command block are padded with zeros. While the format for the command block varies with the command, the first byte is always the command. Some sample command blocks are presented in Listing 2 and Listing 3. More examples of command block wrappers will be found embedded in code in future installments in this series.

A set of common SCSI commands is presented in the code fragment in Listing 4. This code fragment is from code for a USB write blocker and a USB impersonator, which will be, covered in future articles in this series.

DATA TRANSPORT PHASE

Some commands involve a data transport phase. In the case of read and write commands that transfer a considerable amount of data, data is automatically broken up into packets. The maximum packet size is determined by the connection speed. Bulk endpoints are not permitted to operate at low speed. The maximum permissible packet size for full-speed and high-speed connections is 64 and 512 bytes, respectively. One thing to keep in mind when working with USB mass storage devices is that while they are required to operate at full-speed rates, they may suffer from poor performance when connected at this speed. This is in part due to the extra overhead of sending 512 byte or larger blocks in 64 byte chunks. Additionally, the controller may be optimized for high-speed operations.

COMMAND STATUS WRAPPER

Every command terminates with a command status wrapper. The command status wrapper structure is presented in Listing 5. The CSW starts with the signature "USBS" or 0x53425355 in hexadecimal. Next comes the tag, which is used to link the CSW to the correct CBW. The data residue indicates any data that remains to be transferred.

The status byte will be one of three values: 00, 01, or 02 for pass, fail, and phase error, respectively. A status of fail (01) does not indicate the nature of the failure aside from indicating it is not the result of a phase error. Upon receiving a failure

Listing 4. *Common SCSI commands*

```
#define BOMS_FORMAT_UNIT 0x04 //definitely block!
#define BOMS_INQUIRY 0x12
#define BOMS_MODE_SELECT_6 0x15
#define BOMS_MODE_SELECT_10 0x55
#define BOMS_MODE_SENSE_6 0x1a
#define BOMS_MODE_SENSE_10 0x5a
#define BOMS_PREVENT_ALLOW_REMOVAL 0x1e
#define BOMS_READ_6 0x08
#define BOMS_READ_10 0x28
#define BOMS_READ_12 0xa8
#define BOMS_READ_CAPACITY 0x25
#define BOMS_READ_FORMAT_CAPACITIES 0x23
#define BOMS_READ_TOC_PMA_ATIP 0x43
#define BOMS_REPORT_LUNS 0xa0
#define BOMS_REQUEST_SENSE 0x03
#define BOMS_SEND_DIAGNOSTIC 0x1d
#define BOMS_START_STOP_UNIT 0x1b
#define BOMS_SYCHRONIZE_CACHE 0x35
#define BOMS_TEST_UNIT_READY 0x00
#define BOMS_VERIFY 0x2f
#define BOMS_WRITE_6 0x0a //block
#define BOMS_WRITE_10 0x2a //block
#define BOMS_WRITE_12 0xaa //block
```

Listing 5. *Command status wrapper format*

```
typedef struct _USB_MSI_CSW {
   unsigned long dCSWSignature; //0x53425355 or "USBS"
   unsigned long dCSWTag; // associate CBW with CSW response
   unsigned long dCSWDataResidue; // difference between requested data and actual
   unsigned char bCSWStatus; //00=pass, 01=fail, 02=phase error, reset
} USB_MSI_CSW;
```

ON THE WEB

- *http://www.usb.org/developers/docs/* – The official source for all things USB
- *http://lvr.com/usbc.htm* – USB expert Jan Axelson's main USB page
- *http://lvr.com/mass_storage.htm* – Jan Axelson's USB mass storage page
- *http://www.youtube.com/watch?v=3D9uGCvtoFo* – Phil's DEFCON XX Talk on USB impersonation
- *http://www.youtube.com/watch?v=CIVGzG0W-DM* – Phil's 44CON 2011 Talk on USB flash drive forensics
- *http://www.instructables.com/id/Cheap-and-Effective-USB-Write-Blocker/* – Phil's instructable on how to build a USB write blocker based on his BlackHat EU 2012 presentation
- *http://www.concise-courses.com/infosec/20130404/* – Phil's Hacker Hotshots presentation on building a USB impersonator
- *http://ppolstra.blogspot.com* – Phil's blog
- *http://twitter.com/ppolstra* – Phil's Twitter page (@ppolstra)

status a read sense command should immediately be issued to discover the exact error that has occurred.

The host and drive alternate data phases (between DATA0 and DATA1) according to a set of rules. Sometimes the host and drive get out of sync and a reset status is sent to indicate they should both return to the starting phase (DATA0). This process is normally automatic and need not concern the forensic specialist.

SUMMARY

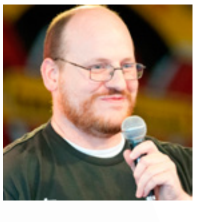
At this point it might be helpful to reiterate the processes when we connect our USB flash drive.

- First the standard enumeration process is performed, as it normally is with all USB devices.
- Once the device has been identified as a USB mass storage device an in bulk endpoint and out bulk endpoint are created.
- The host sends commands to the drive by embedding them in CBWs. If data needs to be exchanged, packets are sent in a data transport phase.
- Once the command has been processed, a CSW is sent from the drive to the host to terminate the transaction.
- The drive then waits for further commands from the host.

While much has been covered in this article, we have merely scratched the surface. USB mass storage devices are a huge topic, but hopefully this introduction and pointers to more information contained within are sufficient to provide a basic understanding of the topic. For a more detailed coverage of this topic I would highly recommend *USB Mass Storage: Designing and programming devices and embedded hosts* by Jan Axelson.

In the next installment of this series we will begin our journey into building devices to be used when performing USB mass storage device forensics. Feel free to contact Phil on twitter, on his blog, or via e-mail with any questions related to this series.

ABOUT THE AUTHOR



Dr. Phil Polstra is currently an Associate Professor and Hacker in Residence at a private university in the Midwestern United States. He also teaches online computer security courses for a private university in Tempe, Arizona. His primary research focus over the last few years has been on the use of small, low-powered devices for forensics and penetration testing. As part of this work, he has developed his own custom pentesting Linux distribution that runs on the BeagleBoard family of ARM-based computers. Phil has presented his work on USB forensics and pentesting with small computers at several conferences around the world including BlackHat, DEFCON, 44CON, GrrCON, ForenSecure, and B-sides, to name a few. When not working he can be found spending time with family, hacking electronics, picking locks, flying, or building airplanes.

HOW TO PREVENT

YOUR CORPORATE ENVIRONMENT FROM BEING INTRUDED BY INFECTED USB DEVICES

by Wimpie Britz

In today's ever evolving computer landscape; employees are constantly bombarded by new technologies aimed at speeding up and improving the way that they conduct business. USB Devices are no exception to the rule, but can the corporate environment afford the risks associated with USB Devices.

What you will learn:

- How to identify infected usb and mobile devices.
- How to protect your organisation/company from Usb and mobile device intrusions.
- How to monitor your organisation/company IT Network and computers for usb and mobile device infections/intrusions.
- Actions taken once your IT environment have been infected by Usb and mobile devices
- To prevent is better than to rectify.

What you should know:

- Usb and mobile devices have got weaknesses.
- Usb and mobile devices is used by cyber criminals.
- Usb and mobile devices does get infected by malware applications.
- Usb and mobile devices are used to penetrate IT Networks and Computer.
- Usb and mobile devices monitoring is important to prevent intrusions in your IT environment.

We have come a long way from the limited and sometimes unreliable Floppy Disks of yester-year to the high speed USB Devices, "thumb drives", on the market today, with storage capacities of +/- 4000 times greater than that of Floppy Disks. "USB" stands for *Universal Serial Bus,* and was first introduced in 1994. It was designed to provide a standardized way of connecting peripherals to a computer. USB Devices are small, robust, easily transported and excellent for quickly copying a few files or presentations to take with you to a meeting. Due to its low cost and custom branding options, more and more Companies use USB Devices for mass marketing at conventions and other exhibition.

Not only are these devices a great way to share information, they can also prove to be eco-friendly. To create a scenario, imagine a teacher doing different lectures at a University and needs to print teaching manuals for different courses, each manual consisting of a few hundred pages. Now, instead of printing all these manuals, they could rather be copied to USB Devices and handed to students. After the information on the USB Devices has been saved to a computer these USB Devices could be re-used again and again. With a 1 Gigabyte USB Device, you can provide students with more information at a lower cost than you would have by printing all the manuals. Content like video's, graphics and audio recording could easily be added, creating a richer learning environment.

A new feature of USB Devices is to run portable applications directly from the USB Device. Like antivirus programs, games, web browsers and many other useful programs. This means users can take all of their favourite programs with them and have access to them on any computer that the USB Device is plugged into. Usually USB Devices with such features have a toolbar that automatically pops up, providing easy access to the installed applications. These USB Devices are also susceptible to malicious attacks, where the attacker would replace the popup toolbar with a program designed to steal information from the compromised computer. As soon as the altered USB Device is plugged into a computer, the computer could automatically launch the attacker's software, allowing the collection of sensitive information.

USB Devices come in many shapes and sizes and has morphed into powerful tools used by the IT Professional and Malicious Users alike. Due to the increase in size of USB Devices it has become possible to load and run an entire OS *Operating System* like Windows or Linux on a USB Device. Proving to be a great platform for IT Administrators to load additional software tools on the OS and create a rescue platform for systems that has crashed, or to perform data recovery. Once such a USB Device has been configured, it can be plugged into any computer's USB port and the OS can be booted directly from the USB Device, making available all the tools that an IT Administrator would normally have available on his own system.

Secure Data Store on USB Devices. Many USB Device manufacturers have a range of USB Devices that carry the "Secure" label. These devices usually have some means of encrypting or protecting the information copied to them. Certain USB Devices have an additional chip installed into the devices tasked to handle the transparent encryption of data as it is being copied onto the USB Device. Some types of USB Devices have either keypads or combination locks physically built into them to further secure the data stored on it, requiring a pin code before allowing access to the drive. Other secure drives feature self-destruct sequences, tamper-event designs and centralized remote management. These types of USB Devices come with a rating as specified by the U.S National Institute of Standards and Technology (NIST), who published a document outlining cryptographic security levels called FIPS 140-2. The highest of this rating is FIPS 140-2 level 3. When purchasing secure drives, be sure to check what FIPS 140-2 rating is given to the drive. Also ensure that the drive has at least 128-bit AES encryption. Because of all these features, even if the USB Device is lost or stolen, the information contained on these encrypted drives remains inaccessible without the necessary decryption keys or passwords. Some manufacturer names to look out for are SanDisk, Kingston, Verbatim and Transcend.

DISADVANTAGES OF USB DEVICE USAGE

THE USB DEVICE INTRUDER

Just as this new feature of USB Devices can be a great advantage to the IT Administrators, it can also prove to be their worst nightmare; Malicious Users can create custom hacking platforms on these Bootable USB Devices, giving them access to all the tools needed to perform less than credible tasks. From stealing Company Information, user login credential or personal files, these new features of USB Devices poses an enormous security risk to any corporate environment.

Some of the first computer viruses used Floppy Disks to spread themselves and infect other computers. USB Devices has now become a carrier, spreading malware like wild fire when a USB Device is infected with a malware, malicious code etc. Depending on the code of the malware, it will spread the malware to every computer that it is connected to. Computers running Windows have an AutoRun feature, allowing users to select certain programs and task to auto execute whenever a device has been plugged in. The autorun.inf file contained on most USB Devices, lists the programs to be auto executed when USB and other devices are plugged into a computer. This feature has also been exploited by malware applications and used USB Devices as a means to easily spread them. A recent study showed that USB Devices are responsible for 26% of all malware infections found on Windows systems due to malware exploiting the AutoRun feature, simply because users do not scan USB Devices for malicious applications before usage or copying files from infected devices. Examples of such infections are the Stuxnet worm and Flame modular computer malware.

Another way Malicious Users have exploited USB Devices is a bit more subtle and many won't even know that they have been compromised until it is too late. As part of a Security study, USB Devices were covertly placed in the parking area of a target Company. Excited employees started to collect these "misplaced" USB Devices when they arrived for work and plugged them into the first computer they came across, their work computer. Unbeknown to them, these USB Devices had Trojan horse viruses loaded onto them before they were placed in the parking lot. The Trojan virus became active and started infecting the Companies systems the moment they were plugged into the employees' computers. Companies, who allow their employees to use unsecured USB Devices to conduct their day-to-day tasks, stand the chance to be negatively affected in an unforeseen way. Because of their small size and portability, these devices are easily misplaced, left behind at clients or stolen, either way the end result is a loss of company data. Depending on the nature of the data, this misplacement could potentially mean expensive lawsuits or the loss of business.

Because of all these negative side-effects of USB Devices, many companies have placed an outright ban on these devices in the work place. Some companies have taken this ban to an extreme level, using silicone or glue to physically close up the USB ports on computers and laptops, making it impossible to use any other USB peripherals, not only USB Devices. Ultimately, these extreme measures are not the answer to the problem. Instead companies need to focus their efforts to find other means of managing USB Devices usage in the work place.

THE SOLUTION TO THE PROBLEM – PREVENTING USB DEVICE INTRUSIONS (BETTER KNOWN AS USB FORENSICS)

INTRODUCTION

The CSFS real time IT Business Intelligence Solution is developed based on combining IT Security preventative measures and procedures and IT Forensics measures and procedures. Combining these two methodologies, gives CSFS the ability to provide the client with a solution that provide Business Intelligence Information.

Business Intelligence information can be defined as; to be in a position at any given time to have insight information on communication inside your network, against your network and leaving your network.

The Solution is focused on two main aspects inside the network environment, namely; to identify malicious / irregular or suspicious network communication and to assist in preventing such communication. This Solution is NOT an Anti-Virus or Intrusion Detection solution and does not take the responsibility of such solutions. The functionality of the Solution is imbedded into identifying known malicious / irregular activities and the ability to identify and trend unknown or suspicious network communication and activity.

PRIMARY SOLUTION FUNCTIONALITIES

- Identification of Malicious / Suspicious / Irregular Communication on external devices such as USB Devices, hard drives;
- Identification of Malicious applications inside a network environment;
- Identification of Malicious / Suspicious / Irregular activities inside a network environment;

PROTECTION

The Solution has the ability to drop any communication identified through its signature files by means of a custom CSFS Firewall module implemented inside the Solution. This Firewall module only has the ability to custom add Firewall drop rules not to permit unauthorised communication entering or leaving the network.

- The Solution makes provision to segment business critical workstations from a normal LAN environment into demilitarized zones (DMZ), by segmenting business critical workstations into such environments makes it possible to protect and manage communication to and from these secure environments and protect highly sensitive information from the rest of the network users.
- The Solution makes use of several alerting and logging methods to analyze data and alert upon them. All information captured on the solution is stored in two ways. Firstly; A Forensic Data Base to secure all alerts and communication information in a tamperproof environment that can be produced in any legal procedure. Secondly the data is stored in a file for easy accessibility, analysis and keeping of proper audit trails. The solution also provides the use of SMS or E-mail to alert on critical signatures.

There exist other ways in disabling usb computer port usage by changing the computers registry file. My personal opinion this involve altering the state of the computer registry which I will not promote unless you have extensive experience in managing computer registry files on your network. This is definitely a not to do for inexperienced computer users. Unfortunately on this tip if you can disable a parameter on a computer then you can also enable it again so this tip is not a permanent solution to the problem but merely a temporary change to implementing the real solution, which will be responsible for managing mobile devices and usb storage devices.

USB DEVICE RISKS CASE STUDIES

CASE STUDY 1

Computer Security and Forensic Solution have had many instances regarding incidents involving infected memory sticks. In one case, while monitoring the clients network it was *noticed by our security team in the Secure Operation Centre* (a 24/7 activity monitoring centre) that employee computers in a Clients' network reported *irregular communication (large amounts of network communication detected from the employee's computers).* Further analysis pointed to *malware infections through usb Memory devices* as being the culprit for these continues irregular communication detections. It was later revealed that employees used their own *personal usb Memory devices to copy data to and from their home computers*, onto their work computers. Most of the employees' home computers did not have the proper active antivirus program installed or updated on their home computers. *What employee's don't realise is that when the laptop is connected inside the company network it is being protected by million dollars of security applications and when this laptops is connected at home its protection is limited to maybe a few dollars of security applications to protect them from the thieves of the internet*. This incident resulted in a companywide ban of company and personal usb Memory devices and also other related Storage devices. By using the correct management software, CSFS was able to create a Whitelist of authorized usb Memory devices and blocked any other unknown USB storage devices, but still allowing other USB peripherals to be used on employee work computers. The IT department managed which employees had the authority to use usb Memory devices and kept track of the whereabouts of these authorized usb Memory devices with asset registers. A process was also implemented that any usb memory devices had to be scanned for infections and cleared by IT before plugged into any computer or company IT devices inside the corporate work environment.

CASE STUDY 2

Another incident occurred when an IT Administrator *covertly downloaded malicious applications and installed these malicious applications* for user activity recording on key employees', like the CFO and CEO's, work computers. The users reported suspicious computer behaviour and slow performance. After a *full scale Forensic investigation conducted by the CSFS Forensic Department*, it was revealed that the recording software originated from an *usb Memory device* and that the malicious application was *surreptitiously installed* while the IT Administrator "worked" on these individuals' work computers under the *alias of fixing network problems* on the computers. The end result was that the IT administrator obtained the *company bank account details like the usernames and, passwords;* he then did fraudulent transactions using the CFO and the Ceo's credentials to log into the company bank accounts and created new beneficiaries and transferred funds to them. This incident in the company resulted in new policies being introduced regarding the usage of usb Memory devices in the workplace environment and implementing proper management solutions to control and prevent the use of unauthorised usb Memory devices usage.

CASE STUDY 3

A trend has developed where people share music and movies with one another and due to this fact, copyright infringement is also another issue CSFS had to deal with in the past on more than one occasion. The most notable was a Client who asked us to *conduct a vulnerability assessment on their network*. During the assessment of certain computers it was identified that the computers purported *suspicious behaviour*. After analysing these computers it was found that the suspicious behaviour was due to the fact that the computers were *heavily infected with malware*. While analysing recently accessed documents on the computers it directed the assessment to *tons of music and movie files,* both on the employees work computers and on the company's network Storage Servers. Users used torrent downloading software to download movies onto the Storage Server and later used external usb memory devices to copy movies and music files to one another and also to take them home. This infected the users work as well as their home computers. CSFS then immediately implemented Network Management and Monitoring solutions and started a virus/malware removal process to clean the virus/malware of the IT Network. Once again the incorrect usage of usb Memory devices and other Storage devices proved to be a real life issue to the security and integrity of the corporate network environment. *What computer users don't realise is that most free music and video files on the internet is infected by malware situated on the provider's network and in this way the computer used to download these files get infected by malware or malicious applications*.

CASE STUDY 4

Through our *Business Intelligence solution implemented* at a client CSFS *identified irregular communication activities* on a client's network. Further investigation pointed to a computer that was *overseen* by the IT department during an *antivirus/malware rollout process* a few months before the incident. This lack of the update of the antivirus/malware program left the computer exposed and vulnerable to malicious attacks. Upon further investigation of the computer it was determined that the user plugged in his own home usb memory device which led to the infection of his work computer and the companies IT network. The infection spread to other workstations on the clients' network and send out communications to a Command and Control Server which is used to perform large scale *Denial of Service* (DoS) attacks against websites. CSFS was able to assist the client by providing a management solution to control the use of usb Memory devices in the workplace and reduce the risk of further attacks against the network environment.

CSFS ACCEPTABLE MOBILE DEVICE AND USB STORAGE DEVICE USAGE POLICY

"POLICY EXAMPLE"

This Usage Policy mandate is to manage the organisational/company information and use of CSFS information on IT equipment. It mainly focuses on the use of mobile device equipment and Usb storage devices. This policy applies to all CSFS employees, contractors and agents hereafter referred to as 'employees'.

This policy applies to all information, in whatever form, relating to CSFS business activities worldwide, and to all information handled by CSFS relating to other organisations with whom it deals. It also covers IT information communications facilities operated by CSFS or on its behalf in relation to mobile devices and Usb storage devices.

MOBILE DEVICE AND USB STORAGE DEVICE ACCESS CONTROL – INDIVIDUAL'S RESPONSIBILITY

Access to the CSFS mobile device and usb storage devices is controlled by the use of User IDs, passwords and/or tokens. All User IDs and passwords are to be uniquely assigned to named employees and consequently, employees are accountable for all actions on the CSFS mobile device and usb storage devices.

EMPLOYEES MUST NOT ENGAGE IN THE FOLLOWING

- Allow anyone else to use their user ID/token and password on any CSFS devices or IT system.
- Leave their user accounts logged in at an unattended and unlocked computer.
- Use someone else's user ID and password to access CSFS devices or IT systems.
- Leave their password unprotected for example writing it down.
- Perform any unauthorised changes to CSFS devices or IT systems or information.
- Attempt to access data that they are not authorised to use or access.
- Exceed the limits of their authorisation or specific business need to interrogate the system data.
- Connect any non-CSFS authorised device to the CSFS network or IT systems.
- Store CSFS data on any non-authorised CSFS equipment.
- Give or transfer CSFS data or software to any person or organisation outside CSFS without the written permission from CSFS.

Managers must ensure that employees are given clear direction on the extent and limits of their authority with regard to IT systems, mobile devices and data.

MOBILE DEVICE AND USB STORAGE DEVICES EXPLICIT CONDITIONS OF USE

Use of mobile devices or usb storage devices must be intended for business use only. Personal use is not permitted. All employees are accountable for their actions on the mobile devices or usb storage devices.

EMPLOYEES MUST NOT ENGAGE IN THE FOLLOWING

- Use the mobile devices or usb storage devices for the purposes of harassment or abuse.
- Use profanity, obscenities, or derogatory remarks in mobile devices or usb storage devices communications.
- Access, download, send or receive any data including images, which CSFS considers offensive in any way, including sexually explicit, discriminatory, defamatory or libellous material.
- Use the mobile devices or usb storage devices to make personal gains or conduct a personal business.
- Use the mobile devices or usb storage devices in a way that could affect its reliability or effectiveness.
- Place any mobile devices or usb storage devices information on the Internet that relates to CSFS, alter any information about it, or express any opinion about CSFS, unless they are specifically authorised to do this.
- Send unprotected sensitive or confidential information externally.
- Send CSFS information on mobile devices or usb storage devices to personal non-CSFS email accounts.
- Make official commitments through the use of mobile devices or usb storage devices on behalf of CSFS unless authorised to do so.
- Download copyrighted material on mobile devices or usb storage devices such as music media MP3 files, film and video files without appropriate approval from CSFS.
- In any way infringe any copyright, database rights, trademarks or other intellectual
- Property by making use of mobile devices or usb storage devices.
- Download any software on mobile devices or usb storage devices from the internet without prior approval of the IT Department.
- Connect CSFS mobile devices or usb storage devices to the internet using non-standard connections.

WORKING AT OTHER PREMISES APART FROM THE ORGANISATIONAL/COMPANY OFFICE

It is understandable that organisational/company laptops, mobile devices and usb storage devices might be taken off-site if authorised. The following controls must be applied in such instants:

- Working away from the office must be in line with CSFS remote working policy.
- Equipment and media taken off-site must not be left unattended in public places and not left in sight in a car.
- Laptops must be carried as hand luggage when travelling.
- Information should be protected against loss or compromise when working remotely for example at home or in public places. Laptop, mobile devices or usb storage devices encryption must be used.
- Particular care should be taken with the use of mobile devices such as laptops, mobile phones, Smartphone's, tablets and usb storage devices. They must be protected at least by a password or a PIN and where available encryption.

MOBILE STORAGE DEVICES

Mobile devices such as memory sticks, CDs, DVDs and removable hard drives must be used only in situations when network connectivity is unavailable or there is no other secure method of transferring data. Only CSFS authorised mobile devices or mass storage devices with encryption enabled must be used when transferring sensitive or confidential data.

EMPLOYEES MUST NOT ENGAGE IN THE FOLLOWING

Store personal files such as music, video, photographs or games on CSFS IT equipment, mobile devices or Usb storage devices.

VIRUSES

The IT department has implemented automated virus detection and virus Software updates within the CSFS IT network infrastructure. All PCs, mobile devices or storage devices have antivirus software installed to detect and remove any virus automatically from pc's, mobile devices or usb storage devices.

EMPLOYEES MUST NOT ENGAGE IN THE FOLLOWING

- Remove or disable anti-virus software.
- Attempt to remove virus-infected files or clean up an infection on mobile devices or usb storage devices, the IT department will by the use of approved CSFS anti-virus software and procedures remove such.

ACTIONS UPON TERMINATION OF CONTRACT

All CSFS equipment and data, for example laptops and mobile devices including telephones, Smartphone's, USB storage devices and CDs/DVDs must be returned to CSFS at termination of the employment contract. All CSFS data or intellectual property developed or gained during the period of employment remains the property of CSFS and must not be retained beyond termination or reused for any other purpose.

MONITORING OF ORGANISATIONAL/ COMPANY DATA AND DEVICES

All data created and stored on CSFS computers, mobile devices or usb storage devices is the property of CSFS and there is no official provision for individual data privacy, however when possible CSFS will avoid opening personal data.

IT system logging will take place where appropriate, and investigations will commence where reasonable suspicion exists of a breach of this or any other policy. CSFS has the right to monitor activity on its IT systems, including mobile devices or usb storage devices used, in order to ensure systems security and effective operations and to protect against any misuse. Any monitoring will be carried out in accordance with audited controlled internal processes and all related country laws will be adhered to in this regard. It is the employee's responsibility to report suspected breaches of organisational/company policies to management, IT department and the information security department.

All breaches of information security policies will be investigated. Where investigations reveal misconduct, disciplinary action may follow in line with CSFS disciplinary procedures.

DATA LEAK PREVENTION / PROTECTION – DLP SERVICES

OVERVIEW

As we are familiar by now with the fact that mobile devices and usb storage devices can be used for good purposes unfortunately it can be used for bad ones also. One of these bad ways is to steal important company and confidential information. With the use of usb devices in the corporate environment employees now have the ability to easily copy company data to the usb device and go home and mail it to for instants competitors or crime syndicates. Fortunately for the employer there is a tool like Data Leak Prevention (DLP) which is an added advantage in managing the company critical data.

MOBILE DEVICES

Data Leak Prevention (DLP) is a set of technologies used to identify monitor and protect critical company information like client's lists, bank account, employee information etc. It doesn't matter if the information is in either a rest motion, or in a movement motion, or in a use motion. Data Leak Prevention is based on policies and depth analysis of organisation information content.

Data Leak Prevention is a massive concern for IT Security specialists. Companies have to take responsibility and address Data Leak Prevention due to the fact that information systems are more and more open to mobility, collaborative tools, legislations, private data protection and to insistent cut-throat environments. The following aspects of Data Leak Prevention needs to be addressed in organisations doesn't matter the size.

- Identify data
- Data monitoring
- Data protection
- Competitive environment analysis

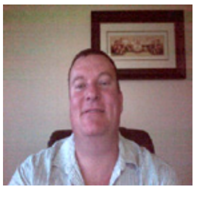
There are a couple companies that specialise in Data Leak Prevention (DLP) tools. I would suggest that proper research be done before purchasing any tools so that the organisational needs of the organisation/company are met at the end of the day.

CONCLUSION

The *real question though is* who is responsible to manage the risks regarding usb memory devices in the work place, and secondly who will be held accountable and liable for the financial loss the organisation/company suffer regarding the risks involving usb memory/storage devices. Virus and malware infections cost companies millions per year to fix and prevent, so rather be preventative and pro-active than becoming one of the statistics.

To summarize; Yes, there are some big risk involved when using mobile devices and USB Devices in the work place environment, but If these devices are properly managed by a solution as described above they could prove to be a great asset to any company's IT arsenal, allowing them to conduct business faster, more conveniently and affectively. *One shouldn't be afraid of the changing environment regarding technologies but rather take hold of the opportunity that it presents in a controlled and manageable way.*

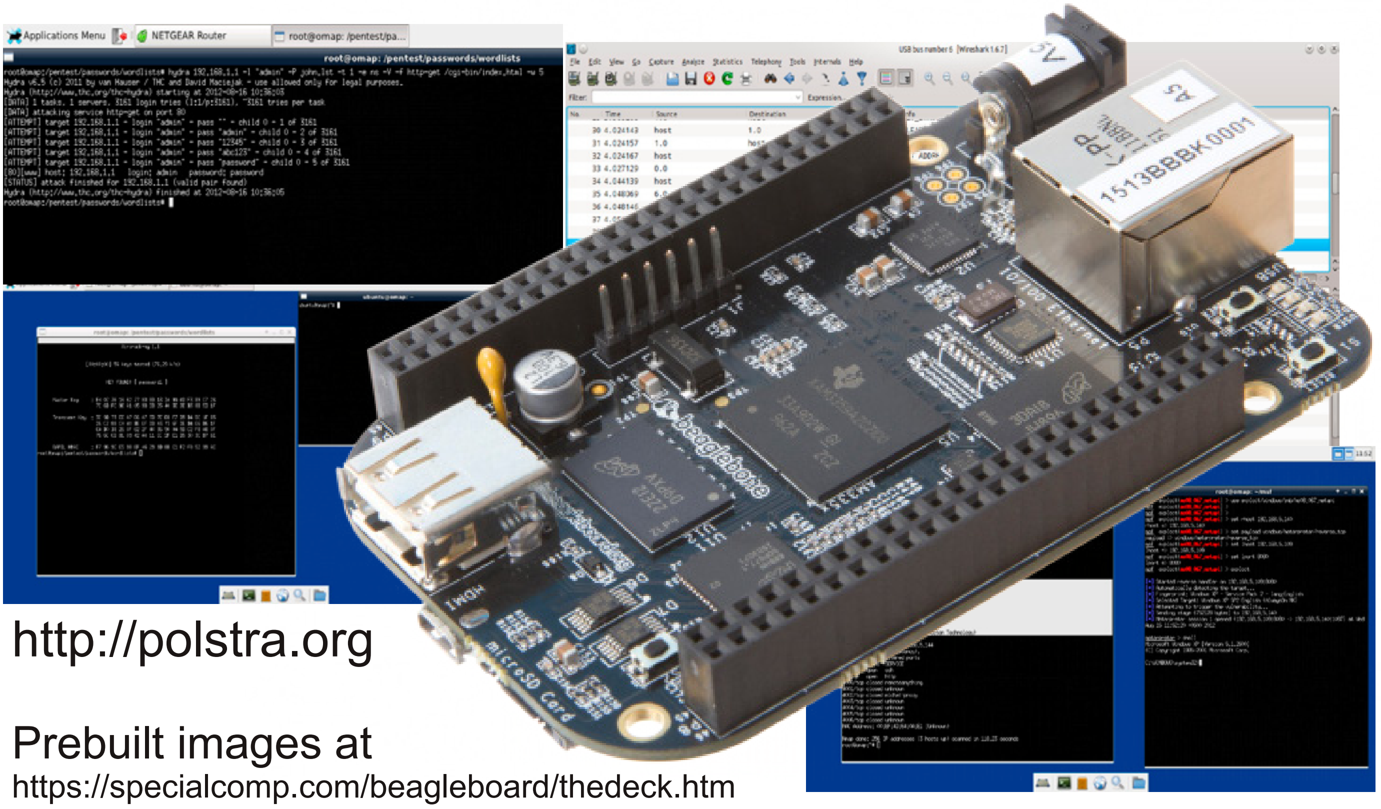
ABOUT THE AUTHOR



Wimpie Britz the Managing Director of the company Computer Security and Forensic Solutions (Pty) Ltd was employed by the South African Police Services for eight years of which the last five years his primary function was to investigate on a National basis in South Africa "Cyber and Computer related Crimes, investigation of IT Security breaches, how they occur and identifying preventative measure". Computer Security & Forensic Solutions was formed in 2000 by the founding member Wimpie Britz as a specialist IT Forensic Investigation and IT Security Solutions Company to help counter the worlds growing computer and cyber related crimes. The e-commerce and electronic communication age showed that systems are penetrable, that hackers are not a myth, employees do defraud companies and industrial/corporate espionage is a reality.

Dr. Phil Polstra is proud to introduce

The Deck



http://polstra.org

Prebuilt images at
https://specialcomp.com/beagleboard/thedeck.htm

The pentesting and forensics Linux distribution for low-power ARM-based devices. Featuring:

- Compact, battery-powered devices from $45
- Hundreds of powerful pentesting and forensics tools
- Ability to pentest from up to 1 mile away using multiple devices
- The ultimate drop box
- Aerial pentesting drone under development
- Full X-window system
- 100% open source software **and** hardware

As seen at these top conferences:

HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

PART 3: DUPLICATING USB MASS STORAGE DEVICES

by Dr. Phil Polstra

USB mass storage devices have become the standard for backup and transfer of files. The popularization of this media has led to challenges for forensic specialists trying to find data on fixed memory storage media instead of traditional magnetic media. This article in a multi-part series will demonstrate how to construct cheap and compact USB mass storage device forensic duplicators.

What you will learn:

- USB mass storage forensic duplication techniques
- How to use FTDI microcontrollers
- Methods for maximizing performance with microcontrollers

What you should know:

- A basic understanding of programming in C
- A basics understanding of USB (possibly from the first two parts in this series)

In recent years USB mass storage devices using NAND flash storage (also known as thumb drives or flash drives) have replaced magnetic media, such as floppy discs, and optical media, such as CD/DVD, as the standard means for backup and file exchange. This article will cover the use of FTDI microcontrollers for creating forensic images and duplicates of USB mass storage devices.

MAKING FORENSICS IMAGES AND DUPLICATES

When making images or forensic duplicates there are a number of options. One straightforward method is to connect a source device to a personal computer (hopefully using through some sort of write blocker). While this is certainly possible, an appropriate workstation and write blocker might not always be on hand. Additionally, commercial USB write blockers tend to be expensive. The next article in this series will describe an inexpensive write blocker based on the FTDI microcontroller.

In this article we will describe another option also based on the FTDI microcontroller. FTDI produces USB-related chips such as those, found in older versions of the Arduino and many USB to serial cables. Despite being introduced several years ago, the FTDI Vinculum 2 (VNC2) remains one of the few microcontrollers on the market, which is capable of operating as either a slave (device), or host. After a brief introduction to the VNC2, details for a pocket-sized USB mass storage device duplicator will be provided.

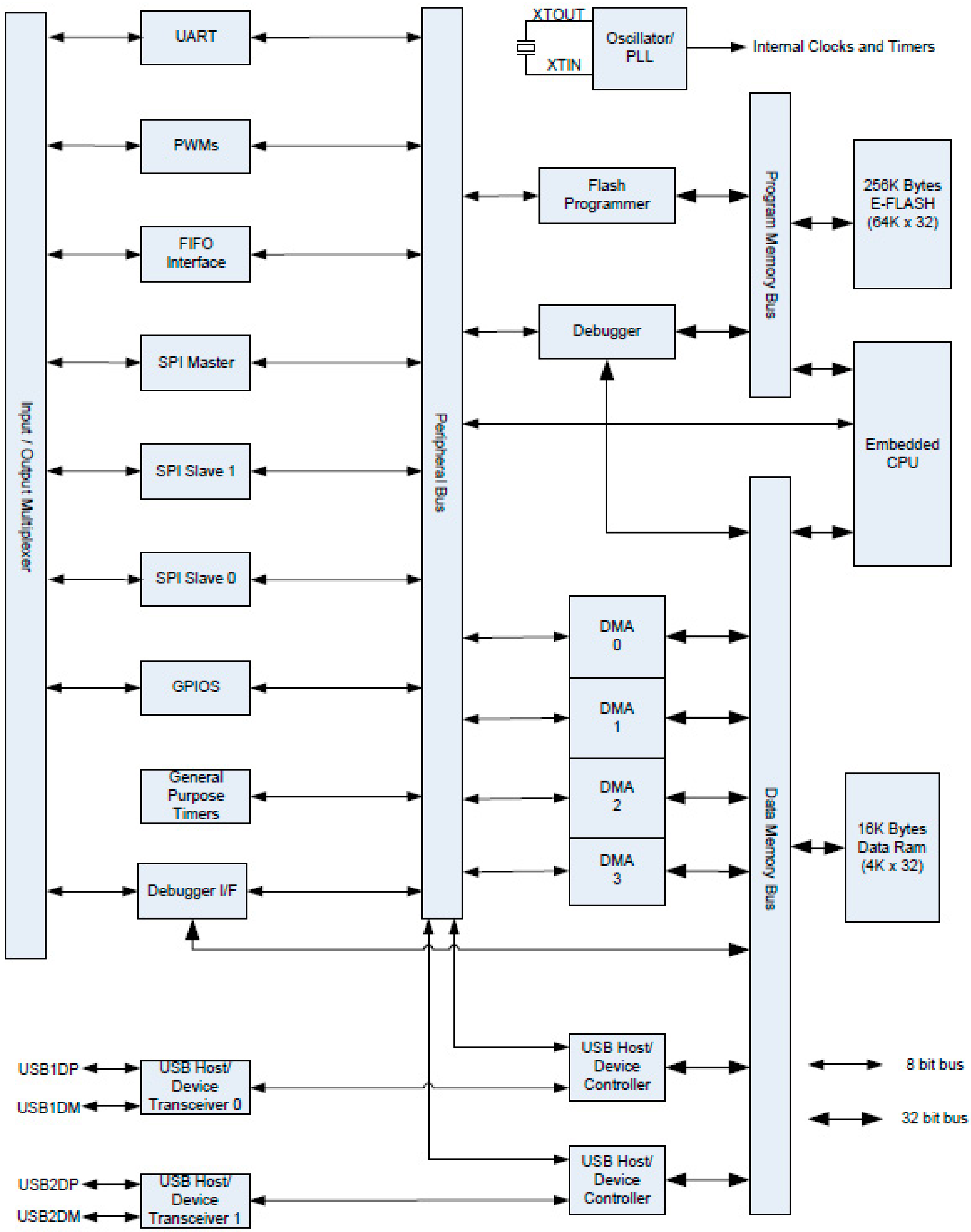


Figure 1. *Vinculum II block diagram*

MEET THE FTDI VINCULUM 2

The FTDI Vinculum II dual USB host controller has many nice features including:

- Embedded 16-bit Harvard architecture MCU core, with 256KBytes of Flash memory and 16Kbytes RAM.
- 2 x Full-Speed / Low-speed USB 2.0 ports supporting Host or Slave operation.
- Programmable UART interface, supports up to 6MBaud transfers.
- 8-bit wide FIFO interface.
- 2 x SPI slave interfaces, 1 x SPI master interface.
- PWM (Pulse Width Modulation) interface.
- 4 channel DMA controller, and general-purpose timers.
- Support for reduced power modes.
- Multiple packages size options (32-/48-/64-pin QFN and LQFP packages).
- Backwards compatible with VNC1L with 48-pin LQFP package.
- RoHS compliant, and extended temperature support (-40°C to +85°C).
- Enhanced features with Vinculum Software Tool Suite
- Based on royalty-free flexible 'C' based Integrated Development Environment.
- Includes compiler, drivers and RTOS kernel to support user firmware development.

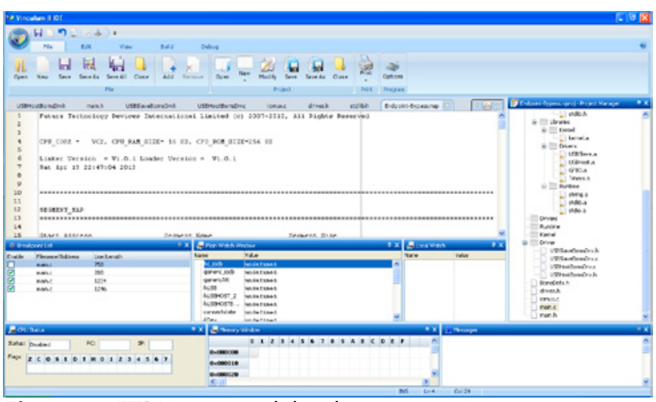


Figure 2. *FTDI Integrated development environment*

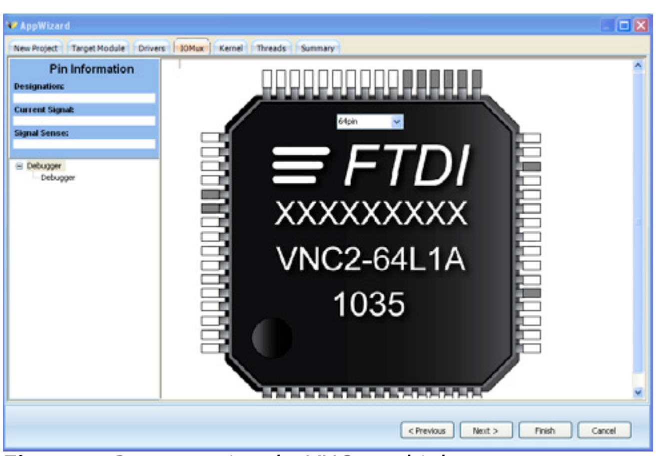


Figure 3. *Programming the VNC2 multiplexer*

- Debug interface for real-time hardware based code debug.
- Pre-compiled libraries for several USB classes including FAT file system support.

The VNC2 is available in 32, 48, and 64 pin versions. Unlike the AVR line of microcontrollers as found in the Arduinos, the only difference between chips in the VNC2 family is the number of general-purpose input/output (GPIO) lines available. This makes it possible to easily scale solutions up and down without any code changes. Unfortunately, the VNC2 is only available in surface mount (SMD) versions, which can make prototyping difficult. However, several relatively inexpensive development modules, including the Arduino-style Vinco, are available.

The VNC2 is a full-featured microcontroller as can be seen from the block diagram in Figure 1. In addition to providing unrivaled USB functionality, the inclusion of pulse width modulation (PWM), inter-integrated circuit (I2C), serial, GPIO, and serial peripheral interface (SPI) support ensures that the VNC2 can be interfaced with virtually any device. The VNC2 may be operated at clock speeds of up to 48 MHz (over twice the typical Arduino speeds). Additionally, the VNC2 normally runs a full-featured real-time operating system known as VOS. Some of the nicer features of VOS include: threads, semaphores, mutexes, and timers. FTDI provides an integrated development environment (IDE) for developing VNC2 applications in C and Assembly. The IDE is shown in Figure 2. The IDE includes a GUI facility for programming the GPIO multiplexer as shown in Figure 3. While the IDE is a Windows-only application, it runs perfectly well in a virtual box under Linux.

A SIMPLE AND COMPACT DUPLICATOR

FTDI provides a number of useful drivers for the VNC2 platform, including drivers for reading mass storage devices. Creating a sector-by-sector copy of a mass storage device is a simple matter of reading in each sector using the FTDI framework and outputting to another compatible mass storage device or storing an image to appropriate media. Full code can be found at *http://polstra.org*. Due to the total code length, I will only provide snippets of code in this article.

The most straightforward way to read sectors (which are typically 512 bytes) is shown in Listing 1. Writing to a target mass storage device is very similar, as recalled from earlier articles discussing that data is read and written in blocks. Maximum performance is achieved when the amount of data read and more importantly written is a multiple of the devices read and write block sizes. An improved method for reading multiple sectors is presented later in this article.

Listing 1. *Reading sectors*

```
unsigned char FatReadSector(unsigned long sec,
unsigned char *buffer, unsigned short sl)
{
   // transfer buffer
   static msi_xfer_cb_t xfer;
   unsigned char stat;
   xfer.sector = sec;
   xfer.buf = buffer;
   xfer.total_len = sl;
   xfer.buf_len = sl;
   xfer.status = MSI_NOT_ACCESSED;
   xfer.s = &semRead;
   xfer.do_phases = MSI_PHASE_ALL;
   stat = vos_dev_read(hBOMS_2, (unsigned char
*)&xfer, sizeof(msi_xfer_cb_t ), NULL);
   if (stat == MSI_OK)
   {
      stat = FAT_OK;
   }
   else
   {
      stat |= FAT_MSI_ERROR;
   }
   return stat;
}
```

Listing 2. *Main processing loop for simple duplicator*

```
void firmware(void)
{
   unsigned short sectorSize2, sectorSize1;
   unsigned char led[5]={0x08, 0x10, 0x20, 0x40,
0x80};
   unsigned char connectstate;
   unsigned char status;
   unsigned short ledStep;
   usbhost_device_handle *ifDev;
   usbhost_ioctl_cb_t hc_iocb, be_iocb;
   usbhost_ioctl_cb_class_t hc_iocb_class;
   // BOMS device variables
   msi_ioctl_cb_t boms_iocb;
   boms_ioctl_cb_attach_t boms_att;
   // FAT file system variables
   fat_ioctl_cb_t fat_ioctl;
   fatdrv_ioctl_cb_attach_t fat_att;
   FILE *file;
   msi_xfer_cb_t xfer;
   usbhost_xfer_t uxfer;
   // completion semaphore
   vos_semaphore_t semRead, semWrite;
   unsigned char *pBuffer;
   unsigned long sector=0;
   unsigned short clusterSize;
   short allDone=0;
   int i; //counting variable

   // open host controller
   hUSBHOST_2 = vos_dev_open(VOS_DEV_USBHOST_2);

   // buffer for reading and writing sectors
   pBuffer = malloc(BUFFER_SIZE);
   do
   {
      // use ioctl to see if bus available
      hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_
GET_CONNECT_STATE;
      hc_iocb.get = &connectstate;
      vos_dev_ioctl(hUSBHOST_2, &hc_iocb);
         if (connectstate == PORT_STATE_ENUMERATED)
      {
         // find and connect a BOMS device
         // USBHost ioctl to find first BOMS
device on host
         hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_
DEVICE_FIND_HANDLE_BY_CLASS;
         hc_iocb.handle.dif = NULL;
         hc_iocb.set = &hc_iocb_class;
         hc_iocb.get = &ifDev;
         hc_iocb_class.dev_class = USB_CLASS_MASS_STORAGE;
         hc_iocb_class.dev_subclass = USB_
SUBCLASS_MASS_STORAGE_SCSI;
         hc_iocb_class.dev_protocol = USB_
PROTOCOL_MASS_STORAGE_BOMS;

         if (vos_dev_ioctl(hUSBHOST_2, &hc_iocb)
!= USBHOST_OK)
         {
            break;
         }

         // now we have a device, intialise a
BOMS driver for it
         hBOMS_2 = vos_dev_open(VOS_DEV_BOMS_2);

         // BOMS ioctl to attach BOMS driver to
device on host
         boms_iocb.ioctl_code = MSI_IOCTL_BOMS_ATTACH;
         boms_iocb.set = &boms_att;
         boms_iocb.get = NULL;
         boms_att.hc_handle = hUSBHOST_2;
         boms_att.ifDev = ifDev;

         status = vos_dev_ioctl(hBOMS_2, &boms_
iocb);
         if (status != MSI_OK)
         {
            break;
         }

         boms_iocb.ioctl_code = MSI_IOCTL_GET_
SECTOR_SIZE;
         boms_iocb.get = &sectorSize2;
```

```
        vos_dev_ioctl(hBOMS_2,&boms_iocb);

        // now connect to the drive to be
written to in port 1
        hUSBHOST_1 = vos_dev_open(VOS_DEV_USBHOST_1);
        // use ioctl to see if bus available
        hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_
GET_CONNECT_STATE;
        hc_iocb.get = &connectstate;

        do
        {
           vos_dev_ioctl(hUSBHOST_1, &hc_iocb);
           vos_delay_msecs(250);
        } while (connectstate!=PORT_STATE_
ENUMERATED);

        // find and connect a BOMS device
        // USBHost ioctl to find first BOMS
device on host
        hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_
DEVICE_FIND_HANDLE_BY_CLASS;
        hc_iocb.handle.dif = NULL;
        hc_iocb.set = &hc_iocb_class;
        hc_iocb.get = &ifDev;
        hc_iocb_class.dev_class = USB_CLASS_
MASS_STORAGE;
        hc_iocb_class.dev_subclass = USB_
SUBCLASS_MASS_STORAGE_SCSI;
        hc_iocb_class.dev_protocol = USB_
PROTOCOL_MASS_STORAGE_BOMS;

        if (vos_dev_ioctl(hUSBHOST_1, &hc_iocb)
!= USBHOST_OK)
        {
           break;
        }

        // now we have a device, intialise a
BOMS driver for it
        hBOMS_1 = vos_dev_open(VOS_DEV_BOMS_1);

        // BOMS ioctl to attach BOMS driver to
device on host
        boms_iocb.ioctl_code = MSI_IOCTL_BOMS_ATTACH;
        boms_iocb.set = &boms_att;
        boms_iocb.get = NULL;
        boms_att.hc_handle = hUSBHOST_1;
        boms_att.ifDev = ifDev;

        status = vos_dev_ioctl(hBOMS_1, &boms_iocb);
        if (status != MSI_OK)
        {
           break;
        }

        boms_iocb.ioctl_code = MSI_IOCTL_GET_
SECTOR_SIZE;
        boms_iocb.get = &sectorSize1;
        vos_dev_ioctl(hBOMS_1,&boms_iocb);
        clusterSize = BUFFER_SIZE/sectorSize1;

        // time to copy
        vos_init_semaphore(&semRead, 0);
        vos_init_semaphore(&semWrite, 0);
        xfer.sector = sector;
        xfer.buf = pBuffer;
        xfer.total_len = BUFFER_SIZE;
        xfer.buf_len = BUFFER_SIZE;
        xfer.status = MSI_NOT_ACCESSED;
        xfer.s = &semRead;
        xfer.do_phases = MSI_PHASE_ALL;
        do
        {
              status = FatReadSector(sector,
pBuffer, BUFFER_SIZE);
              if (status == FAT_OK)
              {
                 status = FatWriteSector(sector,
pBuffer, BUFFER_SIZE);
              } else
              {
                 allDone = 1;
                 break;
              }
              sector+=clusterSize;
           vos_dev_write(hGPIO_PORT_A,
&led[ledStep%5],1,NULL);
        } while((status == FAT_OK)&& !allDone);

        // TO DO: use SCSI command 0x25 to find
drive size instead of going till error
        allDone=1;

        boms_iocb.ioctl_code = MSI_IOCTL_BOMS_DETACH;
        if (vos_dev_ioctl(hBOMS_2, &boms_iocb)
!= MSI_OK)
        {
           break;
        }
        vos_dev_close(hBOMS_2);

        boms_iocb.ioctl_code = MSI_IOCTL_BOMS_DETACH;
        if (vos_dev_ioctl(hBOMS_1, &boms_iocb)
!= MSI_OK)
        {
           break;
        }
        vos_dev_close(hBOMS_1);
        vos_power_down(VOS_WAKE_ON_USB_1); //
go to sleep till next time
     }
  } while (!allDone);
}
```

Figure 4. *An extremely compact USB duplicator*

Figure 5. *A more user-friendly duplicator*

Listing 3. *Main threads for a double-buffered duplicator*

```
void thread_1()
{
   unsigned short i;
   unsigned char status;
   sector1 = 0;
   sector2 = clusterSize;

   while (!enumed2)
   {
      vos_delay_msecs(1000);
   }

   do
   {
      // this funny for loop is to speed up
processing
      // by eliminating as much as possible from
a tight loop
      // while still providing status through
LEDs
      for(i=0;i<500;i++)
      {
         vos_wait_semaphore(&semBuf1);
         vos_lock_mutex(&mBuf1);
         status = FatReadSector(sector1, pBuf1,
BUFFER_SIZE);
         vos_unlock_mutex(&mBuf1);
         if (status == FAT_OK)
         {
            vos_wait_semaphore(&semBuf2);
            vos_lock_mutex(&mBuf2);
            status = FatReadSector(sector2,
pBuf2, BUFFER_SIZE);
            vos_unlock_mutex(&mBuf2);
         } else
         {
            allDone = 1;
            break;
            }
            sector1 += 2*clusterSize;
            sector2 += 2*clusterSize;
         }
         ledStep++;
         vos_dev_write(hGPIO_PORT_A,
&led[ledStep%5],1,NULL);
   } while(!allDone);
}

void thread_2()
{
   unsigned char status;

   while(!enumed1)
      vos_delay_msecs(1000);

   do
   {
      vos_lock_mutex(&mBuf1);
      status = FatWriteSector(sector1, pBuf1,
BUFFER_SIZE);
      vos_unlock_mutex(&mBuf1);
      vos_signal_semaphore(&semBuf1);
      if (status == FAT_OK)
      {
         vos_lock_mutex(&mBuf2);
         status = FatWriteSector(sector2, pBuf2,
BUFFER_SIZE);
         vos_unlock_mutex(&mBuf2);
         vos_signal_semaphore(&semBuf2);
      } else
      {
         allDone = 1;
         break;
      }
   } while(!allDone);
}
```

The main portion of the main processing loop for a duplicator that reads one mass storage device sector by sector and creates a duplicate on a target drive that is at least as large as the source is presented in Listing 2.

So far I have presented the majority of the code required for a simple duplicator.

There is an issue, however; the user has no clue how to operate the device and is not provided with any status information. The simplest situation

Listing 4. *LCD helper functions*

```
void write_lcd_cmd(VOS_HANDLE hLCD, unsigned
char byte)
{
    unsigned char cmd;
    // Write High nibble data to LCD
    cmd = (((byte >> 4) &0x0F) | lcd_e);
    cmd = (cmd &(~lcd_rs)); // Select Registers
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Write Low nibble data to LCD
    cmd = ((byte &0x0F) | lcd_e);
    cmd = (cmd &(~lcd_rs)); // Select Registers
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    vos_delay_msecs(1);

}

void write_lcd_data(VOS_HANDLE hLCD, unsigned
char byte)
{
    unsigned char cmd;

    // Write High nibble data to LCD
    cmd = (((byte >> 4)&0x0F) | lcd_rs);
    cmd = (cmd | lcd_e); // Select DDRAM
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Write Low nibble data to LCD
    cmd = ((byte & 0x0F) | lcd_rs);
    cmd = (cmd | lcd_e); // Select DDRAM
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    vos_delay_msecs(1);

}
void write_lcd_str(VOS_HANDLE hLCD, unsigned
char *str)
{
    while(*str != '\0')
    {
        write_lcd_data(hLCD, *str);
        ++str;
    }
}

void lcd_ini(VOS_HANDLE hLCD)
{
    vos_delay_msecs(100);
    // Send Reset command
    write_lcd_cmd(hLCD, 0x03);
    vos_delay_msecs(2);
    // Send Function Set
    write_lcd_cmd(hLCD, 0x28);
    vos_delay_msecs(2);
    write_lcd_cmd(hLCD, 0x28);
    vos_delay_msecs(2);
    // Send Display control command
    write_lcd_cmd(hLCD, 0x0C);
    vos_delay_msecs(2);
    // Send Display Clear command
    write_lcd_cmd(hLCD, 0x01);
    vos_delay_msecs(2);
    // Send Entry Mode Set command
    write_lcd_cmd(hLCD, 0x06);
    vos_delay_msecs(2);
}

void lcd_clear(VOS_HANDLE hLcd)
{
   // Send Display Clear command
    write_lcd_cmd(hLcd, 0x01);
    vos_delay_msecs(2);
}

void write_lcd_line1(VOS_HANDLE hLcd, unsigned
char* str)
{       // Set 1-st line address
    write_lcd_cmd(hLcd, (0x01 | 0x80));
    // Send string to LCD
    write_lcd_str(hLcd, str);
}

void write_lcd_line2(VOS_HANDLE hLcd, unsigned
char* str)
{
    // Set 2-nd line address
    write_lcd_cmd(hLcd, (0x40 | 0x80));
    // Send string to LCD
    write_lcd_str(hLcd, str);
}
```

would be to use a set of LEDs to indicate status. A more user-friendly and slight less compact device would use an LCD screen in addition to the LEDs.

A very simple duplicator based on the 32-pin development board with 2 USB host ports is shown in Figure 4 alongside its Minty Boost USB power supply. I would challenge readers to find a smaller forensic duplicator. This device blinks to indicate waiting for a target drive to be inserted. While it is copying the lights strobe. A single LED is illuminated to indicate completion. Full source code and schematics for this device are available at *http://polstra.org*.

Users wanting a little user-friendlier device could add an LCD screen. This requires the use of a 48 or 64-pin VNC2 chip assuming that the status LEDs will still be utilized. Thanks to the design of the VNC2 the exact same code can be run on both devices. Reads and writes to nonexistent GPIO lines are ignored on the smaller device. This duplicator is shown alongside its carrying case in Figure 5. The helper functions for printing to the LCD screen are shown in Listing 4. The LCD code is also included in the full code available online.

At this stage we now have some devices that will fit in our pocket and provide use the ability to duplicate flash drives if we find ourselves needing to do so without our complete forensics toolkits. Recall from previous articles in this series that NAND flash write speeds are slow and that changing a single byte requires changing an entire block. To greatly speed up this process we can make use of threads and buffering. The two main methods for a multithreaded and double-buffered duplicator are presented in Listing 3. Note that one thread is used for reading and the second for writing.

FORENSIC INVESTIGATION

In this article we have seen how to create a simple and compact forensic duplicator for USB flash drives. One might ask how to perform an investigation on the duplicated device or image, as appropriate. Because the flash drives emulate conventional hard disk drives and most are formatted with FAT or NTFS file system, forensics of typical drives encountered is similar to that of hard drives.

There are a few notable differences, however. On a magnetic hard disk it is possible to hide information between sectors and tracks. This is not possible with flash memory media. As with other media, when files are deleted they are not normally removed. Rather, the space is marked as available. There is a facility to blank flash media, although this seems rarely used. Some Microsoft operating systems don't properly detect anything beyond the first partition (more accurately called a logical unit) on flash drives. For this reason, PCs running Linux are ideal for performing investigations on flash drives.

ON THE WEB

- *http://ftdichip.com* – The FTDI webpage where IDE and other software may be downloaded
- *http://www.usb.org/developers/docs/* – The official source for all things USB
- *http://lvr.com/usbc.htm* – USB expert Jan Axelson's main USB page
- *http://lvr.com/mass_storage.htm* – Jan Axelson's USB mass storage page
- *http://www.youtube.com/watch?v=3D9uGCvtoFo* – Phil's DEFCON XX Talk on USB impersonation
- *http://www.youtube.com/watch?v=CIVGzG0W-DM* – Phil's 44CON 2011 Talk on USB flash drive forensics
- *http://www.instructables.com/id/Cheap-and-Effective-USB-Write-Blocker/* – Phil's instructable on how to build a USB write blocker based on his BlackHat EU 2012 presentation
- *http://www.concise-courses.com/infosec/20130404/* – Phil's Hacker Hotshots presentation on building a USB impersonator
- *http://ppolstra.blogspot.com* – Phil's blog
- *http://twitter.com/ppolstra* – Phil's Twitter page (@ppolstra)

SUMMARY

There are several options when it comes to making forensic duplicates of USB mass storage devices. In this article we presented a couple of compact duplicators, which utilize the FTDI VNC2 microcontroller. We also discussed simple ways to improve performance of these duplicators. Code and schematics for the devices presented are available online at *http://polstra.org*. While not explicitly presented, modifying these duplicators to output image files is straightforward.

In the next article we will again make use of the FTDI VNC2. In that case we will discuss building an inexpensive and compact USB write blocker. This will require the use of both a USB host and USB slave. Should you have any question on this article, feel free to contact on Twitter @ppolstra or at *http://polstra.org*.

ABOUT THE AUTHOR

Dr. Phil Polstra is currently an Associate Professor and Hacker in Residence at a private university in the Midwestern United States. He also teaches online computer security courses for a private university in Tempe, Arizona. His primary research focus over the last few years has been on the use of small, low-powered devices for forensics and penetration testing. As part of this work, he has developed his own custom pentesting Linux distribution that runs on the BeagleBoard family of ARM-based computers. Phil has presented his work on USB forensics and pentesting with small computers at several conferences around the world including BlackHat, DEFCON, 44CON, GrrCON, ForenSecure, and B-sides, to name a few. When not working he can be found spending time with family, hacking electronics, picking locks, flying, or building airplanes.

HOW TO DETECT A FILE WRITTEN TO

AN USB EXTERNAL DEVICE IN WINDOWS FROM THE MRU LISTS

by Carlos Dias da Silva

Today one of the principal company asset is the digital information. The digital information can be used of a lot of methods and also can be copied using different modes. To know and to control what files were sent to out of the company is a problem nowadays and never is a little the investment to guarantee the data secure.

What you will learn:

- How to use MRU Lists for detect files written to an USB drive;
- How to use the Regripper for mount the Windows registry keys;
- How to use the Encase Imager and FTK Imager for to navigate, export and analyze structure file systems;
- How to detect a file written to an USB external device

What you should know:

- Familiarity with the Encase Imager and FTK Imager;
- Familiarity with the Windows' Registry.

A lot of files are copied and accessed in external storage devices in a company, further when the BYOD concept became a normal practice. In a forensic work, to know what files was accessed from USB devices can help very much. This information can be discovering with the exam of MRU list and some keys of Windows Registry. This article will teach you how to do this.

WHAT IS MRU LISTS

MRU are Most Recently Used lists that most applications and even the operating system uses indiscriminately and in different ways. The original purpose of the MRU was to enable users to keep track of where files landed and to re-visit them for editing. It is less and less useful when the Windows Indexer enables you to find recently `modified/created/read` (CMR) directory lists.

HOW THIS CAN HELP US

Windows is a nosey operating system. But it is not Microsoft's fault (entirely); it tries to support users by not letting them loose files they show an interested in. Loosing files is a big problem; there are so many places users can put them. Windows tries to restrict the locations to removable devices, and the portion of the local file system tree user the `\user\<USERNAME>\documents` tree (In Windows 7+8). An extension to this is that users generally can write to external visiting file systems such as a USB thumb drive, USB writable CD drive, or a USB portable giant 3TB hard drive.

To determine what files have been copied to external devices is not an easy task. The Windows operating system has no consistent policy to record these activities. Many of these records are made by applications in their own registry space in the Windows Registry. Window Explorer tries to keep a MRU (Most Recently Used) documents list in Explorer's application scratch directories in various places, how for example at directories `\users\<USERNAME>\AppData\Roaming\Microsoft\Windows\Recent\` – `\users\<USERNAME>\AppData\Roaming\Microsoft\Office\Recent`. All of this information is persistent and discoverable in the windows registry and in files Explorer keeps.

In Microsoft Windows systems we demonstrate technique that relies on the exam of the MRU lists along with the exam of the Windows Registry detect which files were written to in removable devices. Similarly we can discover if files from the network or server computer were copied to USB devices.

This tutorial will require the download of tree free tools, Encase Imager, FTK Imager and Regripper, which can be found in the following websites:

- *www.guidancesoftware.com/Order-Forensic-Imager.aspx*
- *www.marketing.accessdata.com/acton/form/4390/0119:d-0002/0/index.htm*
- *www.regripper.wordpress.com*

This tutorial starts from the examining stage, and previous knowledge is required on evidence collection and keeping. The following use case was developed to improve the understanding of the tutorial's final purpose.

USE CASE

The company's monitoring software showed that 3 confidential documents were downloaded from the company's server to a computer of the internal network, as follows: MyDocument 01.docx, MyDocument 02.docx and MyDocument 03.docx.

The company's forensics team assigned to check the situation made a bit stream copy of the suspected computer hard disk and conducted technical examinations to identify what was done with the confidential documents downloaded from the company's server.

PART 01 OF EXAMINATIONS

Open the forensic image of the disk using the tool Encase Imager to have access to the files structure (Figure 1).

After opening the forensic image in Encase, the first step is to find the moved files. In order to do that, click on the listbox to show all the files of the disk, as seen in Figure 2 and Figure 3.

All files of the directory structure will be listed in the right-side table of Encase. Double click the column "Name" to arrange the files list by name, as seen in Figure 4.

After the classification, click on the panel and type the name of one of the files that need to be inspected, in this case, we will start with "MyDocument 01". Scroll down the files until you find

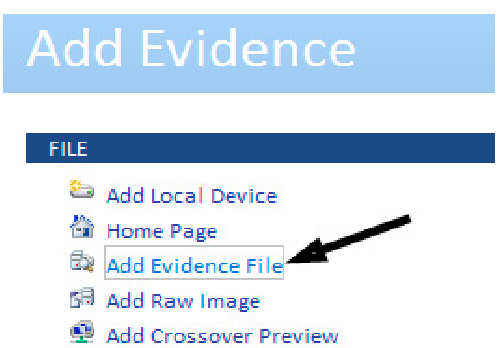


Figure 1. *Opening an evidence with Encase Imager*

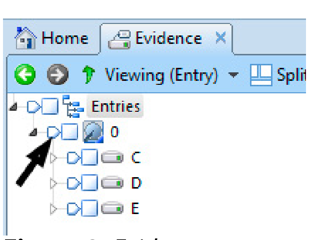


Figure 2. *Evidence on Encase Imager*

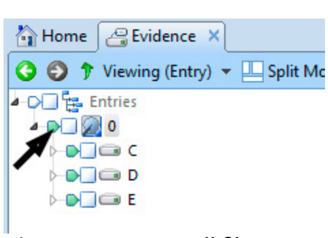


Figure 3. *Listing all files on Encase Imager*

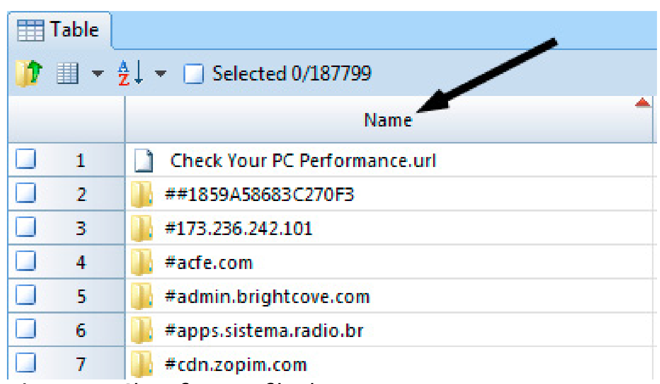


Figure 4. *Classification files by name on Encase Imager*

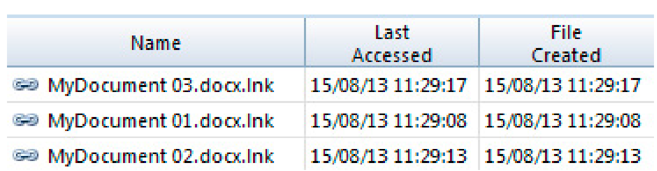


Figure 5. *Searching files on Encase Imager*

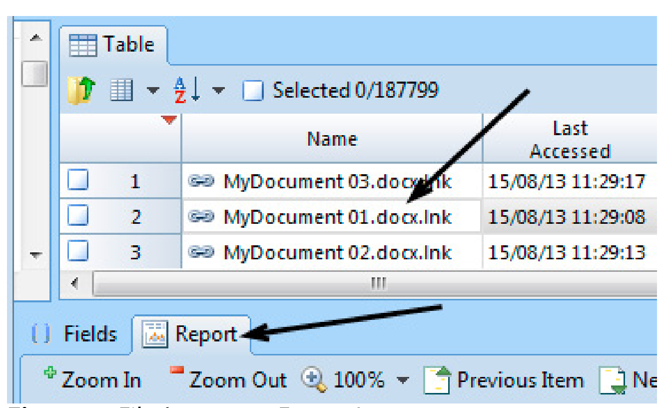


Figure 6. *Files' report on Encase Imager*

them, as seen in Figure 5. It can be seen that files with the extension "Ink" were found. These "Ink" files are automatically created by the operating system when a certain file is opened. They serve as a shortcut to the original file, and are stored in the directory "Systemroot\user\AppData\Roaming\Microsoft\Windows\Recent\" in Microsoft Windows 7 systems.

An "Ink" file has the path to the original file, to which it provides the shortcut, i.e., with the file path it is possible to identify in which device it was stored. To obtain a report on the "Ink" file, click on any of them to select it and then on the tab "Report" of Encase Imager, as seen in Figure 6.

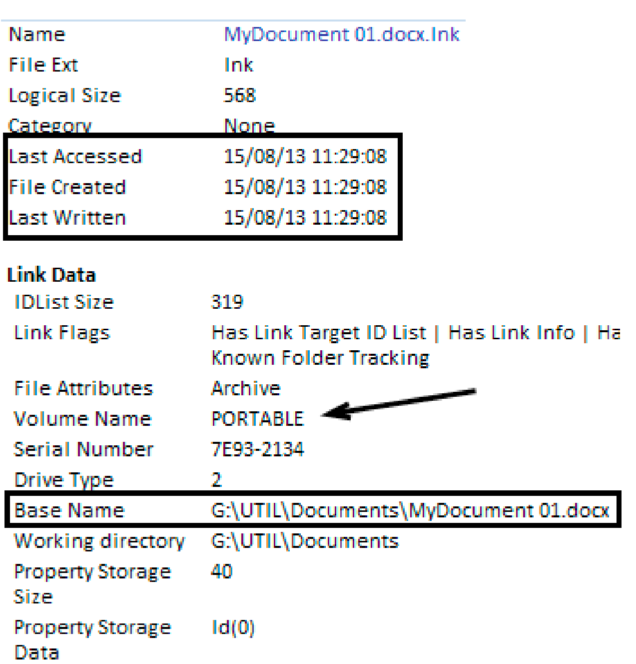
A report on the "Ink" file will be shown, with the following metadata and relevant information:

Table 1. *Details on metadata*

Metadata	Description
Last Accessed	Date of the last access to "Ink"
File Created	Date on which the "Ink" file was created, this date indicates the date on which the original file was first open
Last Written	Last change in the "Ink" file
Volume Name	This name is given by the operating system or manually to the storage unit established in the system.
Base Name	Directory path of the original file

The relevant information and metadata on the file "MyDocument 01.docx.Ink" are the following: Figure 7.

Accordingly, the shortcut (Ink) to the file "MyDocument 01.docx" allowed us to obtain the following information:



Name	MyDocument 01.docx.lnk
File Ext	lnk
Logical Size	568
Category	None
Last Accessed	15/08/13 11:29:08
File Created	15/08/13 11:29:08
Last Written	15/08/13 11:29:08

Link Data

IDList Size	319
Link Flags	Has Link Target ID List | Has Link Info | Ha Known Folder Tracking
File Attributes	Archive
Volume Name	PORTABLE
Serial Number	7E93-2134
Drive Type	2
Base Name	G:\UTIL\Documents\MyDocument 01.docx
Working directory	G:\UTIL\Documents
Property Storage Size	40
Property Storage Data	Id(0)

Figure 7. *File's report on Encase Imager*

- The file "MyDocument 01.docx" was last accessed on 08/15/2013 at 11:29:08;
- The volume that stores this file is named "PORTABLE";
- The file "MyDocument 01.docx" is stored on the directory structure "G:\UTIL\Documents\".

This first part of examinations has indicated us that it is necessary to be certain about what the "G:" unit named "PORTABLE" refers to, given that it is in this unit that one of the confidential files is stored. In order to do that, we will sort the records of the operating system by exporting them through the tool FTK Imager. We will use the FTK Imager in case the Encase Imager does not allow exporting the files.

PART 02 OF EXAMINATIONS

Open the forensic image in the FTK Imager: Figure 8. After assembling the image in the FTK, access "C:\Windows\System32\Config" and export the files "System" and "Software", as follows: Figure 9.

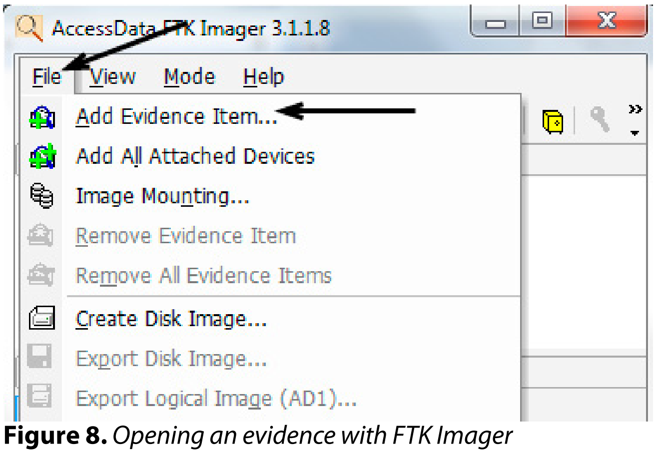


Figure 8. *Opening an evidence with FTK Imager*

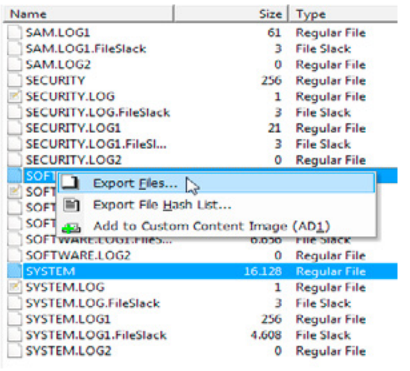


Figure 9. *Export files on FTK Imager*

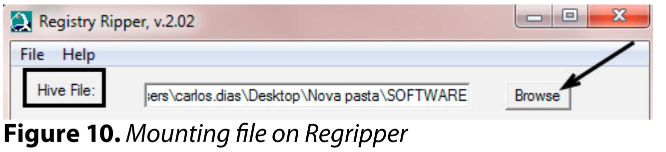


Figure 10. *Mounting file on Regripper*

Figure 11. *Mounting file on Regripper*

Open the tool RegRipper to assemble the exported files, according to the following steps.

Click in the "Browser" button of the field "Hive File" and point the exported file "Software" (Figure 10). Then click on the "Browser" button of the field "Report File" and point where you want the record report to be created (Figure 11). Select the file "Software" in the field "Plugin File" (Figure 12).

Click in the "Rip it" button to create the report on the Software record (Figure 13).

Follow the same procedure for the file System (Figure 14). Now that the record files are assembled, we will examine them in order to find the required answers.

Figure 12. *Mounting file on Regripper*

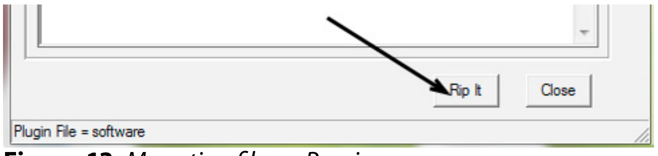


Figure 13. *Mounting file on Regripper*

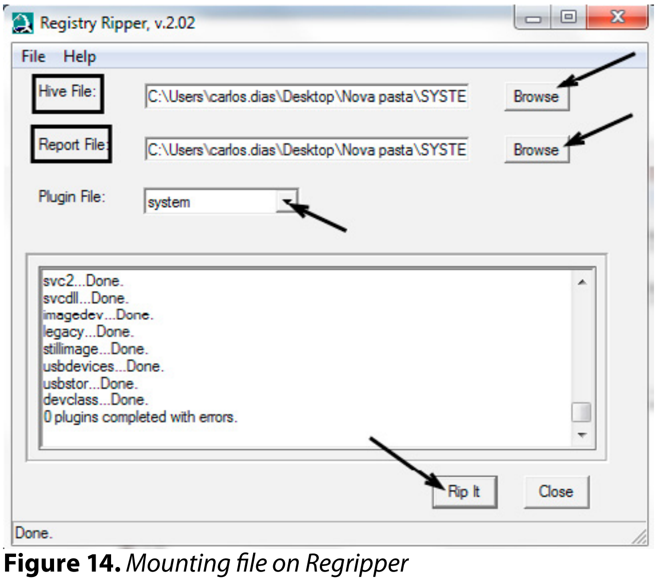


Figure 14. *Mounting file on Regripper*

```
Device    : DISK&VEN_SANDISK&PROD_CRUZER_BLADE&REV_1.26
LastWrite : Thu Aug 15 11:17:56 2013 (UTC)
SN        : 4C532006840524104580&0
Drive     : PORTABLE
```

Figure 15. *Registry key*

```
Device: _??_USBSTOR#Disk&Ven_SanDisk&Prod_Cruzer_Blade&Rev_1.26#4C532006840524104580&0#
    \??\Volume{bdbc55b0-ac80-11e2-80a0-20689d5ecfab}
    \DosDevices\G:
```

Figure 16. *Registry key*

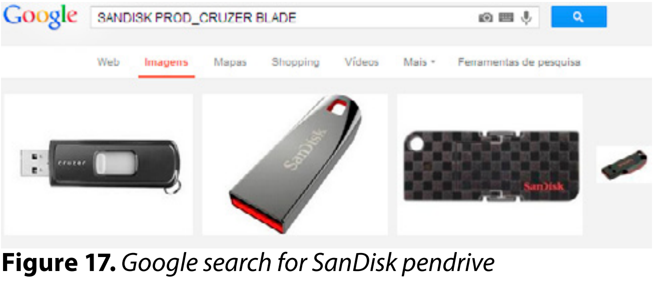


Figure 17. *Google search for SanDisk pendrive*

ON THE WEB

- Details about lnk files *http://www.forensicswiki.org/wiki/LNK*
- *www.guidancesoftware.com/Order-Forensic-Imager.aspx*
- *www.marketing.accessdata.com/acton/form/4390/0119:d-0002/0/index.htm*
- *www.regripper.wordpress.com*

In the file "SOFTWARE" there is the registry key "Microsoft\Windows Portable Devices\Devices". Locate this key and examine the portable devices that were connected to the computer.

It can be seen that the device "SANDISK&PROD_CRUZER_BLADE" named "PORTABLE" was connected to the computer in the same day and time of the accesses of the confidential files "lnk" (that were accessed at 11:29) (Figure 15).

This device has a serial number and it is possible to identify from that number which unit letter the operating system has attributed to this device. In order to do that, open the assembled file "System" and search for the serial number of the device. The result will be similar to the following: Figure 16.

When searching for the device SanDisk Cruzer Blade, we found that it is a flash drive, as follows: Figure 17.

Examining the other lnk files referring to the remaining confidential files, we found that both point to the same "G:" unit, which proves that they are stored in a flash drive.

CONCLUSION

The conclusion of this analysis is that files with the same name of the files downloaded from the company's server were found inside a flash drive of SanDisk.

It was not possible to compare the files because there are only indications to the external device. The analysis of the flash drive is required to prove the deviation of information.

ABOUT THE AUTHOR

Carlos Dias is a systems analyst specialized in Digital Forensics. He has 10 years of experience in technology plus 5 years of experience in Digital Forensics. Nowadays he is coordinator at BDO Brazil conducting Digital Forensics projects in litigation, intellectual property, frauds and cyber crimes. Certifications: ACE, ISFS

HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

PART 4: BLOCK WRITES TO USB MASS STORAGE DEVICES

by Dr. Phil Polstra

USB mass storage devices have become the standard for backup and transfer of files. The popularization of this media has led to challenges for forensic specialists trying to find data on fixed memory storage media instead of traditional magnetic media. This article in a multi-part series will demonstrate how to construct a cheap and compact write blocker for USB mass storage devices.

What you will learn:

- How to block writes to USB mass storage devices
- How to use FTDI microcontrollers
- Differences in how various operating systems handle USB mass storage devices

What you should know:

- A basic understanding of programming in C
- A basics understanding of USB mass storage devices (possibly from the first three articles in this series)

In recent years USB mass storage devices using NAND flash storage (also known as thumb drives or flash drives) have replaced magnetic media, such as floppy discs, and optical media, such as CD/DVD, as the standard means for backup and file exchange. This article will cover a simple and inexpensive device based on the FTDI Vinculum II microcontroller which can block write operations for USB mass storage devices.

MOTIVATION

Has this ever happened to you? A friend asks you for help with a troublesome PC. You insert a flash drive with your security and diagnostics tools into their computer. Their antivirus instantly detects your security tools as malware and begins deleting them. You scramble to yank out the drive, but it is too late. The drive will need to be reloaded. Wouldn't it be great if all this trouble could be easily and cheaply avoided?

Perhaps you need to examine a possibly interesting USB mass storage device, but you don't have an expensive commercial USB write blocker handy. I have spoken to cybercrime units that could only afford one shared USB write blocker for the entire unit. Additionally, if a flash drive has been identified as interesting, you will need to make additional working copies and having multiple cheap write blockers could avoid the bottleneck inherent in having to share a single blocker.

BLOCKING USB WRITES

There are a number of ways to block write operations to USB mass storage devices. Some older flash drives featured at write-protect switch.

This feature is quite rare in modern devices. On certain versions of Windows USB write operations can be blocked by creating a registry key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ StorageDevicePolicies\ WriteProtect. Creating this registry key will block write operations to *all* USB mass storage devices, however. There are several commercial solutions that can be used to block writes. Software can be used to restrict which devices (vendor ID, product ID, and possibly serial number) can be opened for writing. A number of hardware write blockers costing hundreds of dollars are also available. Alternatively, the device described here can be constructed for less than US$20.

IMPLEMENTING A WRITE BLOCKER

The FTDI Vinculum II (VNC2) is a convenient and inexpensive microcontroller for building a write

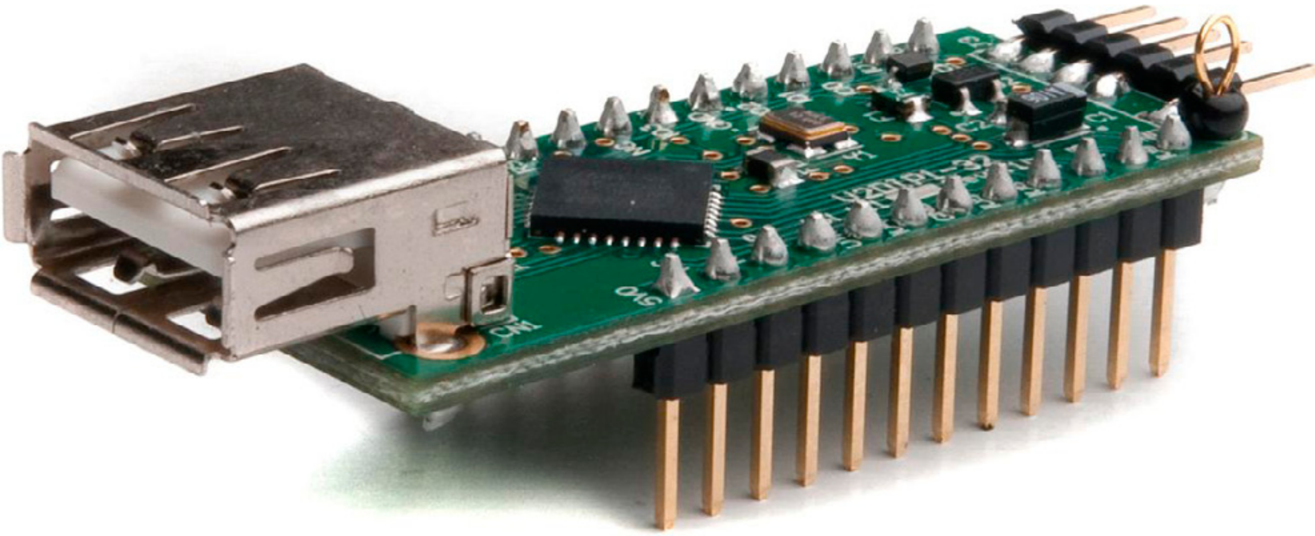


Figure 1. *V2DIP1-32*

Figure 2. *FTDI Vinco*

blocker. While a custom device could be created, the fact that the VNC2 chips are only available in surface-mount formats makes this somewhat inconvenient. Fortunately, FTDI provides development boards which can be used for this device. The best choice for this project is a V2DIP1-32 which has one USB host port and is based on the 32-pin VNC2 chip. The V2DIP1-32 is shown in Figure 1. This user will need to solder on a USB cable for the USB slave port. This is easily done by cutting the device end off of an unwanted USB cable. The wires should be color coded as red, white, green, and black for +5V, data-, data+, and ground, respectively. The corresponding pins on the V2DIP1-32 are labeled 5V0, GND, U1P, and U1M for +5V, ground, data+, and data-, respectively. The user may optionally wish to either trim or desolder the header pins in order to make the device smaller and not as painful to carry in a pocket.

Alternatively, a device could be created using FTDI's Arduino-like board known as the Vinco. This has the advantage of not requiring the user to do any soldering. It should be noted that there have been some reported issues when using the Vinco. In particular, the way that power is applied to the USB host port in software can lead to timing issues which may prevent a flash drive from enumerating properly in some cases. The Vinco is shown in Figure 2. Because all the VNC2 chips have the same memory and flash storage, the same code is easily run on both devices.

HIGH LEVEL DESIGN

The device needs to intercept and block any commands that could potentially alter the USB drive. A lazy way to implement this would be to block blacklisted commands. This is not the proper way to implement this, however. Rather, the proper way to implement and future-proof the device is to whitelist benign commands. The original design would return "unsupported command" for commands to be blocked. The code was redesigned to fake completing a command

Listing 1. *Host enumeration thread methods*

```
unsigned char usbhost_connect_state(VOS_HANDLE
hUSB)
{
   unsigned char connectstate = PORT_STATE_
DISCONNECTED;
   usbhost_ioctl_cb_t hc_iocb;

   if (hUSB)
   {
      hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_
GET_CONNECT_STATE;
      hc_iocb.get = &connectstate;
      vos_dev_ioctl(hUSB, &hc_iocb);
   }

   return connectstate;
}

void open_drivers(void)
{
   gpio_ioctl_cb_t gpio_iocb;
   unsigned char leds;

   /* Code for opening and closing drivers -
move to required places in Application Threads
*/
   /* FTDI:SDA Driver Open */
   hGPIO_PORT_E = vos_dev_open(VOS_DEV_GPIO_
PORT_E);

   // power up Vinco USB Host
   // this must happen before we want to
enumerate the flash drive
   gpio_iocb.ioctl_code = VOS_IOCTL_GPIO_SET_
MASK;
   gpio_iocb.value = 0x60;            // set
power and LED as output
   vos_dev_ioctl(hGPIO_PORT_E, &gpio_iocb);
   leds = 0x00;
   vos_dev_write(hGPIO_PORT_E, &leds, 1, NULL);

   hUSBHOST_2 = vos_dev_open(VOS_DEV_USBHOST_2);

   /* FTDI:EDA */

   hUSBHOSTBOMS = vos_dev_open(VOS_DEV_
USBHOSTBOMS);

   hUSBSLAVE_1 = vos_dev_open(VOS_DEV_
USBSLAVE_1);

   hUSBSLAVEBOMS = vos_dev_open(VOS_DEV_
USBSLAVEBOMS);

}

void attach_drivers(void)
{
   common_ioctl_cb_t bomsAttach;

   /* FTDI:SUA Layered Driver Attach Function
Calls */
   /* FTDI:EUA */

   // attach BOMS to USB Host port B
   bomsAttach.ioctl_code = VOS_IOCTL_
USBHOSTBOMS_ATTACH;
   bomsAttach.set.data = (void *) hUSBHOST_2;
   vos_dev_ioctl(hUSBHOSTBOMS, &bomsAttach);
```

```
   // attach BOMS to USB Slave port A
   bomsAttach.ioctl_code = VOS_IOCTL_
USBSLAVEBOMS_ATTACH;
   bomsAttach.set.data = (void *) hUSBSLAVE_1;
   vos_dev_ioctl(hUSBSLAVEBOMS, &bomsAttach);

}

void close_drivers(void)
{

   vos_dev_close(hUSBHOST_2);
   vos_dev_close(hUSBHOSTBOMS);
   vos_dev_close(hUSBSLAVE_1);
   vos_dev_close(hUSBSLAVEBOMS);
   vos_dev_close(hGPIO_PORT_E);
}

void hostEnum()
{

   unsigned char i;
   unsigned char status;
   unsigned char buf[64];
   unsigned short num_read;
   unsigned int handle;

   usbhostBoms_ioctl_t generic_iocb;
   usbhost_device_handle_ex ifDev;
   usbhost_ioctl_cb_t hc_iocb;
   usbhost_ioctl_cb_class_t hc_iocb_class;
   usbhostBoms_ioctl_cb_attach_t genericAtt;

   open_drivers(); // open all drivers including
USB host and slave
   attach_drivers(); // enumerate flash drive
then connect slave

   do
   {
      // see if bus available
      if (usbhost_connect_state(hUSBHOST_2) ==
PORT_STATE_ENUMERATED)
      {
            // ultimately want to find a mass
storage device using SCSI protocol
            hc_iocb_class.dev_class = USB_CLASS_
MASS_STORAGE;
            hc_iocb_class.dev_subclass = USB_
SUBCLASS_MASS_STORAGE_SCSI;
            hc_iocb_class.dev_protocol = USB_
PROTOCOL_MASS_STORAGE_BOMS;

            // user ioctl to find first hub
device
            hc_iocb.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_FIND_HANDLE_BY_CLASS;
            hc_iocb.handle.dif = NULL;
            hc_iocb.set = &hc_iocb_class;
            hc_iocb.get = &ifDev;

            if (vos_dev_ioctl(hUSBHOST_2, &hc_
iocb) == USBHOST_OK)
            {
               // optionally notify user of
status via LEDs, etc.
            }

            genericAtt.hc_handle = hUSBHOST_2;
            genericAtt.ifDev = ifDev;
            generic_iocb.ioctl_code = VOS_IOCTL_
USBHOSTBOMS_ATTACH;
            generic_iocb.set.att = &genericAtt;

            // we use a simple variable to
indicate if the flash drive
            // is attached
            // this is not  as elegant as using
a semaphore, but
            // this is the only thread that
updates this variable and
            // if the device is disconnected and
reconnected that is hard
            // to handle with a semaphore
            if (vos_dev_ioctl(hUSBHOSTBOMS,
&generic_iocb) == USBHOSTBOMS_OK)
            {
               slaveBomsCtx->flashConnected = 1;
               vos_signal_
semaphore(&slaveBomsCtx->enumed);
            } else
            {
               slaveBomsCtx->flashConnected = 0;
            }// if attach
            // this code is in here so that if
the drive gets disconnected
            // we can try to restart it
            // also, hopefully the the traffic
every few seconds will keep
            // the drive from going to sleep
            vos_delay_msecs(2000);
      } // if enumerated
         vos_delay_msecs(10); // recheck every
.01 seconds for new connect
   } // outer do
   while (1);

}
```

Listing 2. *Command block wrapper handling thread*

```
void handleCbw()
{
   unsigned short num_read, num_written;
   boms_cbw_t *cbw = vos_malloc(sizeof(boms_
cbw_t));

   vos_wait_semaphore(&slaveBomsCtx->enumed);
   vos_signal_semaphore(&slaveBomsCtx->enumed);

   while(1)
   {
      if(slaveBomsCtx)
      {
         while(slaveBomsCtx && slaveBomsCtx-
>flashConnected)
            {
            // get the CBW
            memset(cbw, 0, sizeof(boms_cbw_t));
            usbSlaveBoms_readCbw(cbw,
slaveBomsCtx);
            // TO DO: Check for valid CBW

            switch (cbw->cb.formated.command)
            {
               case BOMS_INQUIRY:
                  handle_inquiry(cbw);
                  break;
               case BOMS_MODE_SELECT_6:
               case BOMS_MODE_SELECT_10:
                  handle_mode_select(cbw);
                  break;
               case BOMS_MODE_SENSE_6:
               case BOMS_MODE_SENSE_10:
                  handle_mode_sense(cbw);
                  break;
               case BOMS_READ_6:
               case BOMS_READ_10:
               case BOMS_READ_12:
                  handle_read(cbw);
                  break;
               case BOMS_READ_CAPACITY:
                  handle_read_capacity(cbw);
                  break;
               case BOMS_REPORT_LUNS:
                  handle_report_luns(cbw);
                  break;
               case BOMS_REQUEST_SENSE:
                  handle_request_sense(cbw);
                  break;
               case BOMS_TEST_UNIT_READY:
                  handle_test_unit_ready(cbw);
                  break;
               case BOMS_SEND_DIAGNOSTIC:
                  handle_send_diagnostic(cbw);
                  break;
               case BOMS_START_STOP_UNIT:
                  handle_start_stop_unit(cbw);
                  break;
               case BOMS_SYCHRONIZE_CACHE:
                  handle_synchronize_cache(cbw);
                  break;
               case BOMS_READ_FORMAT_CAPACITIES:
                  handle_read_format_capacities(cbw);
                  break;
               case BOMS_PREVENT_ALLOW_REMOVAL:
                  handle_prevent_allow_removal(cbw);
                  break;
               case BOMS_READ_TOC_PMA_ATIP:
                  handle_read_toc_pma_atip(cbw);
                  break;
               case BOMS_VERIFY:
               case BOMS_FORMAT_UNIT:
                  // tell them NO! by failing command
                  handle_illegal_request(cbw);
                  break;
               case BOMS_WRITE_6:
               case BOMS_WRITE_10:
               case BOMS_WRITE_12:
                  handle_illegal_write_request(cbw);
                  break;
               default:
                  handle_illegal_request(cbw);
                  break;
            } // switch
         } // inner while
      } else
      {
         vos_delay_msecs(1000);
      }
   } // outer while

   vos_free(cbw);
}
```

Listing 3. *Handler methods*

```
unsigned short forward_cbw_to_device(boms_cbw_t
*cbw)
{
   unsigned short num_written;
   usbhostBoms_write((void*)cbw, sizeof(boms_
cbw_t), &num_written, hostBomsCtx);

   return num_written;
}

unsigned short receive_data_from_device(void*
buffer, unsigned short expected)
{
   unsigned short num_read;
   unsigned char status;

   status = usbhostBoms_read(buffer, expected,
```

```
&num_read, hostBomsCtx);
   if (status == USBHOST_EP_HALTED)
   {
      // the endpoint is halted so let's halt
the slave endpoint
      usbslaveboms_stall_bulk_in(slaveBomsCtx);
   }

   return num_read;
}

unsigned short forward_data_to_slave(void*
buffer, unsigned short bytes)
{
   unsigned short num_written;
   usbSlaveBoms_write(buffer, bytes, &num_
written, slaveBomsCtx);

   return num_written;
}

unsigned short forward_data_to_slave_then_
stall(void* buffer, unsigned short bytes)
{
   unsigned short num_written;
   usbSlaveBoms_short_write(buffer, bytes, &num_
written, slaveBomsCtx);

   return num_written;
}

unsigned short receive_csw_from_device(boms_
csw_t *csw)
{
   unsigned short num_read;
   usbhostBoms_read((void*)csw, 13, &num_read,
hostBomsCtx);

   return num_read;
}

unsigned short forward_csw_to_slave(boms_csw_t
*csw)
{
   unsigned short num_written;
   usbSlaveBoms_write((void*)csw, 13, &num_
written, slaveBomsCtx);

   return num_written;
}

void handle_inquiry(boms_cbw_t *cbw)
{
   unsigned char buffer[64];
   unsigned short responseSize;
   boms_csw_t csw;

   // forward the CBW to device
   if (forward_cbw_to_device(cbw))
   {
      // receive response from device
      // note we will assume that only the
standard 36 bytes will be requested
      if (responseSize = receive_data_from_
device(&buffer[0], 36))
      {
         // forward response to slave
         forward_data_to_slave(&buffer[0],
responseSize);

         // receive CSW from device
         if (receive_csw_from_device(&csw))
         {
            // forward CSW to slave
            forward_csw_to_slave(&csw);
         }
      }
   }
}

void handle_test_unit_ready(boms_cbw_t *cbw)
{
   boms_csw_t csw;

   // forward the CBW to device
   if (forward_cbw_to_device(cbw))
   {
      // receive response from device
      if (receive_csw_from_device(&csw))
      {
         // forward CSW to slave
         forward_csw_to_slave(&csw);
      }
   }
}

void handle_read(boms_cbw_t *cbw)
{
   // this same routine handles all 3 possible
read commands
   // most likely read command is read(10)

   unsigned long lba; // logical block address
for start block
   unsigned short blocks; // number of blocks to
read
   unsigned short i;
   boms_csw_t csw;
   unsigned char *buffer;
   unsigned short num_read;
   unsigned short num_written;

   switch (cbw->cb.formated.command)
   {
```

```
      case BOMS_READ_6:
         lba = cbw->cb.raw[1]*65536 + cbw->cb.
raw[2]*256 + cbw->cb.raw[3];
         blocks = cbw->cb.raw[4];
         break;
      case BOMS_READ_10:
         lba = cbw->cb.raw[2]*16777216 + cbw-
>cb.raw[3]*65536 + cbw->cb.raw[4]*256 +cbw->cb.
raw[5];
         blocks = cbw->cb.raw[7] * 256 + cbw-
>cb.raw[8];
         break;
      case BOMS_READ_12:
         lba = cbw->cb.raw[2]*16777216 + cbw-
>cb.raw[3]*65536 + cbw->cb.raw[4]*256 +cbw->cb.
raw[5];
         // we are being a little bad here the
number of blocks is actually a long
         // it is extremely unlikely that anyone
would request this much at once, however
         blocks = cbw->cb.raw[8] * 256 + cbw-
>cb.raw[9];
         break;
   }

   // now forward the cbw to the device
   forward_cbw_to_device(cbw);

   // receive the appropriate number of blocks
from the device
   // forward the blocks to the slave
   // most requests are probably 1 block of 512 bytes
   // read in 512 byte chunks (packet size is 64
bytes, but VOS should handle this)
   // If devices with larger blocks are
encountered, 512 should still work
   buffer = vos_malloc(blockSize);
   while(blocks>0)
   {
      usbhostBoms_read((void*)buffer, blockSize,
&num_read, hostBomsCtx);
      usbSlaveBoms_write((void*)buffer, num_
read, &num_written, slaveBomsCtx);
      blocks--;
   }
   vos_free(buffer);

   // receive the csw from the device
   receive_csw_from_device(&csw);

   // forward the csw to the slave
   forward_csw_to_slave(&csw);
}

void handle_read_capacity(boms_cbw_t *cbw)
{
   boms_csw_t csw;
   unsigned char buffer[8];
   unsigned short received;

   // forward cbw to device
   forward_cbw_to_device(cbw);

   // receive response from device
   if (received = receive_data_from_
device(&buffer[0], 8))
   {
      deviceCapacity = buffer[0]*16777216 +
buffer[1]*65536 + buffer[2]*256 +buffer[3];
      blockSize = buffer[4]*16777216 +
buffer[5]*65536 + buffer[6]*256 +buffer[7];
      // forward response to slave
      forward_data_to_slave(&buffer[0], received);
   }

   // receive csw from device
   receive_csw_from_device(&csw);

   // forward csw to slave
   forward_csw_to_slave(&csw);
}

void handle_report_luns(boms_cbw_t *cbw)
{
   boms_csw_t csw;
   unsigned char buffer[64];
   unsigned short received;

   // forward cbw
   forward_cbw_to_device(cbw);

   // receive response from device
   // response is 8 bytes + maxLuns * 8
   if (received = receive_data_from_
device(&buffer[0], 8 + 8 * maxLuns))
   {
      // forward response to slave
      forward_data_to_slave(&buffer[0],
received);
   }

   // receive csw from device
   receive_csw_from_device(&csw);

   // forward csw to slave
   forward_csw_to_slave(&csw);
}

void handle_request_sense(boms_cbw_t *cbw)
{
   boms_csw_t csw;
   unsigned char bytesRequested;
   unsigned short bytesRead, bytesWritten;
   unsigned char *buffer;
```

```
   request_sense_response_t rsr;

   if (illegalRequest)
   {
      // if we are here this a request sense
that came right after an illegal
      // command request - perhaps somebody
tried to modify our drive!
      // we return the appropriate error
directly and a CSW
      // the device is never touched

      illegalRequest = 0; // reset so next
request goes to actual device

      rsr.formated.responseCode =0x70; //0x70
current error
      rsr.formated.valid = 0; // 1=INFORMATION
field valid
      rsr.formated.obsolete = 0;
      rsr.formated.senseKey = 0x05; // 0x05 for
illegal request
      rsr.formated.resvered = 0;
      rsr.formated.ili = 0; // incorrect length
indicator
      rsr.formated.eom = 0; // end of media for
streaming devices
      rsr.formated.filemark = 0; // for
streaming devices
      rsr.formated.information = 0; // device
specific info
      rsr.formated.addSenseLen = 0x0a; //
additional bytes that follow 244 max
      rsr.formated.cmdSpecInfo = 0; // command
specific info
      rsr.formated.asc = 0x20; // additional
sense code 0x20 for illegal command
      rsr.formated.ascq = 0; // additional sense
code qualifier 0-unused
      rsr.formated.fruc = 0; // field
replaceable unit code set to 0
      rsr.formated.senseKeySpecific[0] = 0; //
senses key spec info if b7=1
      rsr.formated.senseKeySpecific[1] = 0;
      rsr.formated.senseKeySpecific[2] = 0;
      bytesWritten = forward_data_to_slave(&rsr, 18);

      // now send an appropriate CSW to indicate
success of this command
      csw.sig[0] = 'U'; //"USBS"
      csw.sig[1] = 'S';
      csw.sig[2] = 'B';
      csw.sig[3] = 'S';
      csw.tag = cbw->tag;
      csw.residue = 0;
      csw.status = 0; // 0x00=success
0x01=failure 0x02=phase error
      forward_csw_to_slave(&csw);
   } else
   {
      // forward cbw
      bytesRequested = cbw->cb.raw[4];
      forward_cbw_to_device(cbw);
      buffer = vos_malloc((unsigned short)
bytesRequested);

      // receive data from device
      if (bytesRead = receive_data_from_
device(buffer, (unsigned short)bytesRequested))
      {
         bytesWritten = forward_data_to_
slave(buffer, bytesRead);
      }

      vos_free(buffer);
      // receive csw from device
      receive_csw_from_device(&csw);

      // forward csw to slave
      forward_csw_to_slave(&csw);
   }
}

void handle_mode_sense(boms_cbw_t *cbw)
{
   boms_csw_t csw;
   unsigned short allocLength=0;
   unsigned char *buffer=NULL;
   unsigned short bytesReceived=0;

   // forward the cbw to the device
   switch (cbw->cb.formated.command)
   {
      case BOMS_MODE_SENSE_6:
         allocLength = cbw->cb.raw[4];
         break;
      case BOMS_MODE_SENSE_10:
         allocLength = cbw->cb.raw[7]*256 + cbw-
>cb.raw[8];
         break;
   }
   forward_cbw_to_device(cbw);

   // receive data from device
   if (allocLength)
   {
      buffer = vos_malloc(allocLength);
      bytesReceived = receive_data_from_
device(buffer, allocLength);
      // forward data to slave
      forward_data_to_slave(buffer,
bytesReceived);
      vos_free(buffer);
   }
```

```
   // receive csw from device
   receive_csw_from_device(&csw);

   // forward csw to slave
   forward_csw_to_slave(&csw);
}

void handle_mode_select(boms_cbw_t *cbw)
{
   boms_csw_t csw;
   unsigned short allocLength=0;
   unsigned char *buffer=NULL;
   unsigned short bytesReceived=0;

   // forward the cbw to the device
   switch (cbw->cb.formated.command)
   {
      case BOMS_MODE_SELECT_6:
         allocLength = cbw->cb.raw[4];
         break;
      case BOMS_MODE_SELECT_10:
         allocLength = cbw->cb.raw[7]*256 + cbw-
>cb.raw[8];
         break;
   }
   forward_cbw_to_device(cbw);

   // receive data from device
   if (allocLength)
   {
      buffer = vos_malloc(allocLength);
      bytesReceived = receive_data_from_
device(buffer, allocLength);
      // forward data to slave
      forward_data_to_slave(buffer,
bytesReceived);
      vos_free(buffer);
   }

   // receive csw from device
   receive_csw_from_device(&csw);

   // forward csw to slave
   forward_csw_to_slave(&csw);
}

void handle_illegal_request(boms_cbw_t *cbw)
{
   usbslave_ioctl_cb_t iocb;
   boms_csw_t csw;

   // now send the CSW
   csw.sig[0]='U';
   csw.sig[1]='S';
   csw.sig[2]='B';
   csw.sig[3]='S';//"USBS"
   csw.tag=cbw->tag;
   csw.residue=0;
   csw.status=0x01; // 0x00=success 0x01=failure
0x02=phase error
   forward_csw_to_slave(&csw);

   // flag the error for the anticipated call to
REQUEST SENSE
   illegalRequest=1;
}

void handle_illegal_write_request(boms_cbw_t
*cbw)
{
   usbslave_ioctl_cb_t iocb;
   boms_csw_t csw;
   unsigned short blocks;
   unsigned char *buffer;
   unsigned short num_read;
   unsigned short i;


   // as strange as it may seem, there is no way
to tell the host to quit
   // instead we need to receive all this data
and throw it away!
   switch (cbw->cb.formated.command)
   {
      case BOMS_WRITE_6:
         blocks = cbw->cb.raw[4];
         break;
      case BOMS_WRITE_10:
         blocks = cbw->cb.raw[7] * 256 + cbw-
>cb.raw[8];
         break;
      case BOMS_WRITE_12:
         // we are being a little bad here the
number of blocks is actually a long
         // it is extremely unlikely that anyone
would request this much at once, however
         blocks = cbw->cb.raw[8] * 256 + cbw-
>cb.raw[9];
         break;
   }

   buffer = vos_malloc(512);
   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
   iocb.handle = slaveBomsCtx->out_ep;
   iocb.request.setup_or_bulk_transfer.buffer =
buffer;
   iocb.request.setup_or_bulk_transfer.size =
512;
   iocb.request.setup_or_bulk_transfer.bytes_
transferred = 0;
   for (i = 0; i < (blocks * (512/blockSize));
i++)
```

```
   {
      // process bytes received from host
      vos_dev_ioctl(slaveBomsCtx->handle,&iocb);
   }
   vos_free(buffer);

   // now send the CSW
   csw.sig[0]='U';
   csw.sig[1]='S';
   csw.sig[2]='B';
   csw.sig[3]='S';//"USBS"
   csw.tag=cbw->tag;
   csw.residue=0;
   csw.status=0x00; // 0x00=success 0x01=failure
0x02=phase error
   //forward_csw_to_slave(&csw);
   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
   iocb.handle = slaveBomsCtx->in_ep;
   iocb.request.setup_or_bulk_transfer.buffer =
&csw;
   iocb.request.setup_or_bulk_transfer.size =
sizeof(boms_csw_t);
   vos_dev_ioctl(slaveBomsCtx->handle, &iocb);

}

void handle_send_diagnostic(boms_cbw_t *cbw)
{
   usbslave_ioctl_cb_t iocb;
   boms_csw_t csw;

   // first send ZLDP to ACK the command
   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
   iocb.handle = slaveBomsCtx->in_ep;
   iocb.request.setup_or_bulk_transfer.buffer =
NULL;
   iocb.request.setup_or_bulk_transfer.size = 0;
   vos_dev_ioctl(slaveBomsCtx->handle, &iocb);

   // now send the CSW
   csw.sig[0]='U';
   csw.sig[1]='S';
   csw.sig[2]='B';
   csw.sig[3]='S';//"USBS"
   csw.tag=cbw->tag;
   csw.residue=0;
   csw.status=0x00; // 0x00=success 0x01=failure
0x02=phase error
   forward_csw_to_slave(&csw);

}

void handle_start_stop_unit(boms_cbw_t *cbw)
{
   boms_csw_t csw;
   // forward the CBW to device
   if (forward_cbw_to_device(cbw))
   {
      // receive response from device
      if (receive_csw_from_device(&csw))
      {
         // forward CSW to slave
         forward_csw_to_slave(&csw);
      }
   }

}

void handle_synchronize_cache(boms_cbw_t *cbw)
{
   usbslave_ioctl_cb_t iocb;
   boms_csw_t csw;

   // first send ZLDP to ACK the command
   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
   iocb.handle = slaveBomsCtx->in_ep;
   iocb.request.setup_or_bulk_transfer.buffer =
NULL;
   iocb.request.setup_or_bulk_transfer.size = 0;
   vos_dev_ioctl(slaveBomsCtx->handle, &iocb);

   // now send the CSW
   csw.sig[0]='U';
   csw.sig[1]='S';
   csw.sig[2]='B';
   csw.sig[3]='S';//"USBS"
   csw.tag=cbw->tag;
   csw.residue=0;
   csw.status=0x00; // 0x00=success 0x01=failure
0x02=phase error
   forward_csw_to_slave(&csw);

}

void handle_read_format_capacities(boms_cbw_t
*cbw)
{
   unsigned char *buffer;
   unsigned short responseSize;
   unsigned short allocLength;
   boms_csw_t csw;

   allocLength = cbw->cb.raw[7] * 256 + cbw->cb.
raw[8];
   buffer = vos_malloc(allocLength);

   // forward the CBW to device
   if (forward_cbw_to_device(cbw))
   {
      // receive response from device
```

```
      if (responseSize = receive_data_from_
device(buffer, allocLength))
      {
         // forward response to slave
         forward_data_to_slave(&buffer[0],
responseSize);

         // receive CSW from device
         if (receive_csw_from_device(&csw))
         {
            // forward CSW to slave
            forward_csw_to_slave(&csw);
         }
      }
   }

   vos_free(buffer);
}

void handle_read_toc_pma_atip(boms_cbw_t *cbw)
{
   unsigned char *buffer;
   unsigned short responseSize;
   unsigned short allocLength;
   boms_csw_t csw;

   allocLength = cbw->cb.raw[7] * 256 + cbw->cb.
raw[8];
   buffer = vos_malloc(allocLength);

   // forward the CBW to device
   if (forward_cbw_to_device(cbw))
   {
      // receive response from device
      if (responseSize = receive_data_from_
device(buffer, allocLength))
      {
         // forward response to slave
         forward_data_to_slave(&buffer[0],
responseSize);

         // receive CSW from device
         if (receive_csw_from_device(&csw))
         {
            // forward CSW to slave
            forward_csw_to_slave(&csw);
         }
      }
   }

   vos_free(buffer);
}

// This function handles the call to prevent/
allow removal
// If we fail this command when prevent=1 (true)
then
// Windows will not cache writes.  This actually
leads
// to better performance.
void handle_prevent_allow_removal(boms_cbw_t
*cbw)
{
   usbslave_ioctl_cb_t iocb;
   boms_csw_t csw;

   // now send the CSW
   csw.sig[0]='U';
   csw.sig[1]='S';
   csw.sig[2]='B';
   csw.sig[3]='S';//"USBS"
   csw.tag=cbw->tag;
   csw.residue=0;
   csw.status=0x01; // 0x00=success 0x01=failure
0x02=phase error
   forward_csw_to_slave(&csw);

   // flag the error for the anticipated call to
REQUEST SENSE
   illegalRequest=1;

}
```

Listing 4. *USB mass storage slave driver*

```
#include "vos.h"
#include "devman.h"
#include "memmgmt.h"

#include "ioctl.h"

#include "USB.h"
#include "USBHID.h"
#include "USBSlave.h"
#include "USBHost.h"

#include "USBSlaveBomsDrv.h"
#include "USBHostBomsDrv.h"

unsigned char standard_request(usbSlaveBoms_
context *ctx);
unsigned char class_request(usbSlaveBoms_context
*ctx);
void set_control_ep_halt(usbSlaveBoms_context
*ctx);


unsigned char usbSlaveBoms_ioctl(common_ioctl_
cb_t *cb, usbSlaveBoms_context *ctx);

// thread states
#define UNATTACHED 0
#define ATTACHED   1
```

```
// Every USB device has an 18 byte descriptor
// This descriptor is immediately retrieved by
the host/hub in order
// to determine how to talk to the device
// Note: All USB values are little-endian (LSB
first)
unsigned char device_descriptor[18] =
{
    18, //length in bytes
   1, // descriptor type 1=device
   0x0, 2, // USB version BCD USB version
   0, // device class 0=actual device class in
interface descriptor
   0, // device subclass 0=actual device class
in interface descriptor
   0, // device protocol 0=actual device class
in interface descriptor
   64, // max packet size is 64 bytes for full
speed endpoints
   0x4b, 0x15, // vendor id  // currently spoof
PNY drive
   0x40, 0, // product id
   0, 0x01, // device release number in BCD
   1, // string descriptor index for
manufacturer
   2, // string descriptor index for product
   3, // string descriptor index for serial
number
   1 // number of possible configurations
};

// The configuration header is actually a
composite of
// configuration, interface, and endpoint
descriptors
// The host will typically ask for the first
part of the
// descriptor (first 9 bytes) which contains the
total descriptor length.
// The host will then make a second request for
the entire descriptor
unsigned char config_descriptor[32] =
{
// Configuration Header
    9, // length
   2, // descriptor type 2 = configuration
   32, 0, // bytes in this and all subordinate
descriptors
   1, // number of interfaces
   1, // configuration value
   0, // index of string descriptor for
configurations
   0x80, // self/bus power and remote wakeup
   50, // max power in milliamps /2 asking for
too much can cause enum to fail
// Interface
    9, // length
   4, // descriptor type 4 = interface
   0, // interface number
   0, // alternate setting - 00 is default
   2, // number of endpoints other than control
(1 bulk in, 1 bulk out)
   8, // class type 8 for mass storage
   6, // subclass 6 means SCSI transport
   0x50, // interface protocol
   0, // index of string descriptor for the
interface
// Endpoint, EP1_In
    7, // length
   5, // endpoint descriptor
   0x81, // endpoint address bit7-1 = in
   2, // attributes 2=bulk
   0x40, 0x00, // max packet size 64 bytes
   1, // polling interval, ignored for bulk
endpoints
// Endpoint, EP2_Out
    7, // length
   5, // endpoint descriptor
   0x02, // endpoint address bit7-1 = in
   2, // attributes 2=bulk
   0x40, 0x00, // max packet size
   1 // polling interval, ignored for bulk
endpoints
};


// language descriptor
// Requests for string descriptor 0 return a
code for the default
// language.  Devices could support multiple
languages.  In practice,
// everyone seems to support US English only,
even if they are from
// Glasgouw, UK.
unsigned char str0_descriptor[4] =
{  0x04, // length
   0x03, // type 3=string
   0x09,
   0x04 // US English
};

// According to our configuration descriptor,
string descriptor 2
// is a product ID.  This Product descriptor was
borrowed from an
// actual flash drive.  Alternatively, we could
query the actual
// information from the flash drive, but we fake
it to make things
// a bit simplier.  These descriptors all use
UNICODE
unsigned char str2_descriptor[22] =
{  22, // length
   3, // string descriptor
```

```
   0x55, 0, //USB 2.0 FD
   0x53, 0,
   0x42, 0,
   0x20, 0,
   0x32, 0,
   0x2e, 0,
   0x30, 0,
   0x20, 0,
   0x46, 0,
   0x44, 0
};

// According to our configuration descriptor,
string descriptor 1
// is the manufacturer.  In this case we have
borrowed PNY.
unsigned char str1_descriptor[8] =
{  8, // length
   3, // string descriptor
   0x50, 0, // PNY
   0x4e, 0,
   0x59, 0
};

// Every mass storage device is required to have
a serial number
// the last 12 digits must be unique.  There is
no other specification
// on how these are assigned.  Note that Windows
will store this and
// lots of other information in the registry.
USBDevView is a nice
// free utility for viewing this information.
// More information on USB forensics can be
found in my 44Con video
// which is available on SecurityTube or
youtube.
unsigned char str3_descriptor[30] =
{  30, // length
   3, // string descriptor
   0x55, 0, // UTYM0832030481
   0x54, 0,
   0x59, 0,
   0x4d, 0,
   0x30, 0,
   0x38, 0,
   0x33, 0,
   0x32, 0,
   0x30, 0,
   0x33, 0,
   0x30, 0,
   0x34, 0,
   0x38, 0,
   0x31, 0
};

// global variables are evil, but sometimes when
// dealing with memory-constrained
microcontrollers
// they are somewhat unavoidable
usbSlaveBoms_context *slaveBomsCtx=NULL;
extern usbhostBoms_context_t *hostBomsCtx;

// These are the only 2 Class requests sent on
the control endpoint
// BOMS devices send requests in the CBW on the
bulk out endpoint
#define GET_MAX_LUN 0xfe
#define BOMS_RESET 0xff

// The following functions are for stalling and
clearing bulk endpoints
// There are some conditions which require us to
stall these endpoints.
// For example, if we return less data than
expected to the PC we
// must stall the endpoint so it doesn't hang
forever waiting for the rest
// of the data.
void usbslaveboms_stall_bulk_in(usbSlaveBoms_
context *ctx)
{
   usbslave_ioctl_cb_t iocb; // this is a
structure used by underlying VOS driver

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_STALL;
   iocb.ep = 0x81; // bulk in endpoint
   vos_dev_ioctl(ctx->handle, &iocb);
}

void usbslaveboms_stall_bulk_out(usbSlaveBoms_
context *ctx)
{
   usbslave_ioctl_cb_t iocb; // this is a
structure used by underlying VOS driver

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_STALL;
   iocb.ep = 0x02; // bulk out endpoint
   vos_dev_ioctl(ctx->handle, &iocb);
}

void usbslaveboms_clear_bulk_in(usbSlaveBoms_
context *ctx)
{
   usbslave_ioctl_cb_t iocb; // this is a
structure used by underlying VOS driver

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_CLEAR;
   iocb.ep = 0x81; // bulk in endpoint
   vos_dev_ioctl(ctx->handle, &iocb);
}
```

```
void usbslaveboms_clear_bulk_out(usbSlaveBoms_
context *ctx)
{
   usbslave_ioctl_cb_t iocb; // this is a
structure used by underlying VOS driver

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_CLEAR;
   iocb.ep = 0x02; // bulk out endpoint
   vos_dev_ioctl(ctx->handle, &iocb);
}


// This function MUST BE CALLED BEFORE THE
SCHEDULER IS STARTED
// It initializes (initialises for you Brits
reading this) variables
// in the context structure, registers our
driver with the
// VOS device manager, and creates the thread
for handling
// requests on the control endpoint.
unsigned char usbslaveboms_init(unsigned char
vos_dev_num)
{
   vos_driver_t *usbSlaveBoms_cb;

   slaveBomsCtx = vos_
malloc(sizeof(usbSlaveBoms_context));

   if (slaveBomsCtx == NULL)
      return USBSLAVEBOMS_ERROR;  //somehow ran
out of RAM

   usbSlaveBoms_cb = vos_malloc(sizeof(vos_
driver_t));

   if (usbSlaveBoms_cb == NULL)
   {
      vos_free(slaveBomsCtx);
      return USBSLAVEBOMS_ERROR;
   }

   // Set up function pointers for our driver
   usbSlaveBoms_cb->flags = 0;
   usbSlaveBoms_cb->read = usbSlaveBoms_read;
   usbSlaveBoms_cb->write = usbSlaveBoms_write;
   usbSlaveBoms_cb->ioctl = usbSlaveBoms_ioctl;
   usbSlaveBoms_cb->interrupt = (PF_INT) NULL;
   usbSlaveBoms_cb->open = (PF_OPEN) NULL;
   usbSlaveBoms_cb->close = (PF_CLOSE) NULL;

   // OK - register with device manager
   vos_dev_init(vos_dev_num, usbSlaveBoms_cb,
slaveBomsCtx);
   // defaults to not connected and no flash
drive yet
   slaveBomsCtx->attached = 0;
   slaveBomsCtx->flashConnected = 0;

   // create the thread that handles standard
control requests
   slaveBomsCtx->tcbSetup = vos_create_thread_
ex(31, SIZEOF_BOMS_SETUP_MEMORY, usbslaveboms_
setup, "BOMSSetup", 2, slaveBomsCtx);

   // initialize the sempahore to 0 so that
anyone waiting for
   // the device to enum will block
   vos_init_semaphore(&slaveBomsCtx->enumed, 0);

   if (slaveBomsCtx->tcbSetup)
      return USBSLAVEBOMS_OK;

   return USBSLAVEBOMS_ERROR;
}

// This function is run inside the thread
created in init
// It will respond to standard USB requests
// some of the requests are forwarded to the
flash drive when appropriate
void usbslaveboms_setup(usbSlaveBoms_context
*ctx)
{
   usbslave_ioctl_cb_t iocb; // this is a
structure used by underlying VOS driver
   usb_deviceRequest_t *devReq; // this struct
is defined in usb.h and is used to store the 9
byte setup request
   unsigned char bmRequestType; // The request
type is defined by USB standard
   unsigned char state = UNATTACHED;  // assume
unattached at the start

   // This will wait till the flash drive is
enumed so we don't
   // start responding to commands right away
   vos_wait_semaphore(&ctx->enumed);
   vos_signal_semaphore(&ctx->enumed);

   while (1)
   {
      switch (state)
      {
      case UNATTACHED:

         if (!ctx->attached)
            vos_delay_msecs(100); // this delay
is to avoid a tight loop
         else
         {
```

```
            state = ATTACHED;
        }

        break;

    case ATTACHED:

        if (!ctx->attached) // check to see if
we somehow became unattached
        {
            state = UNATTACHED;
            break;
        }

        // we now make a blocking call
requesting the 9 byte setup
        // packet on the control endpoint
        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
WAIT_SETUP_RCVD;
        iocb.request.setup_or_bulk_transfer.
buffer = ctx->setup_buffer;
        iocb.request.setup_or_bulk_transfer.
size = 9;
        vos_dev_ioctl(ctx->handle, &iocb);

        // decode the raw data by pointing to
our structure
        devReq = (usb_deviceRequest_t *) ctx-
>setup_buffer;
        // valid types here are standard,
class, and vendor
        // BOMS devices do not have vendor
specific calls
        // even if they did, we wouldn't
support them.
        bmRequestType = devReq->bmRequestType &
(USB_BMREQUESTTYPE_STANDARD | USB_BMREQUESTTYPE_
CLASS);

        // we only need to handle standard and
class requests for BOMS
        if (bmRequestType == USB_BMREQUESTTYPE_
STANDARD)
        {
            standard_request(ctx); // standard
request that all USB devices support
        }
        else if (bmRequestType == USB_
BMREQUESTTYPE_CLASS)
        {
            class_request(ctx); // the request
is specific to this device class (only 2 in our
case)
        }

        break;

        default:
            asm {HALT}; // if we somehow got here
the fecal matter has hit the turbine
            break;
        }
    }

    return;
}

// this function just marks our device as
detached
void usbSlaveBoms_detach(usbSlaveBoms_context
*ctx)
{
    ctx->attached = 0;

    return;
}

// this function will present our device to the
PC and cause it
// to be enumed.
unsigned char usbSlaveBoms_attach(VOS_HANDLE
handle, usbSlaveBoms_context *ctx)
{
    usbslave_ioctl_cb_t iocb;
    unsigned char status = USBSLAVEBOMS_OK;

    // save usb slave handle
    ctx->handle = handle;

    if (!ctx->attached)
    {
        // issue connect IOCTL call here to
present ourselves to the host
        // MUST be called before configuring
endpoints
        // Note: If you used older versions of the
VNC2 toolchain this function
        // was added and the need to make this
call is buried in the release
        // notes.  Upgraders of old code be
warned!
        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
CONNECT;
        iocb.set = (void *) 0;
        vos_dev_ioctl(ctx->handle,&iocb);

        // get endpoint handles and set max packet
sizes
        // these should match the descriptor
values!
        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_
CONTROL_ENDPOINT_HANDLE;
        iocb.ep = USBSLAVE_CONTROL_IN;
        iocb.get = &ctx->in_ep0;
```

```
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_
ENDPOINT_MAX_PACKET_SIZE;
      iocb.handle = ctx->in_ep0;
      iocb.request.ep_max_packet_size =
USBSLAVE_MAX_PACKET_SIZE_64;
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_
CONTROL_ENDPOINT_HANDLE;
      iocb.ep = USBSLAVE_CONTROL_OUT;
      iocb.get = &ctx->out_ep0;
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_
ENDPOINT_MAX_PACKET_SIZE;
      iocb.handle = ctx->out_ep0;
      iocb.request.ep_max_packet_size =
USBSLAVE_MAX_PACKET_SIZE_64;
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_
BULK_IN_ENDPOINT_HANDLE;
      iocb.ep = USBSLAVEBOMS_IN;
      iocb.get = &ctx->in_ep;
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_
ENDPOINT_MAX_PACKET_SIZE;
      iocb.handle = ctx->in_ep;
      iocb.request.ep_max_packet_size =
USBSLAVE_MAX_PACKET_SIZE_64;
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_
BULK_OUT_ENDPOINT_HANDLE;
      iocb.ep = USBSLAVEBOMS_OUT;
      iocb.get = &ctx->out_ep;
      vos_dev_ioctl(ctx->handle, &iocb);

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_
ENDPOINT_MAX_PACKET_SIZE;
      iocb.handle = ctx->out_ep;
      iocb.request.ep_max_packet_size =
USBSLAVE_MAX_PACKET_SIZE_64;
      vos_dev_ioctl(ctx->handle, &iocb);

      ctx->attached = 1; // mark our device as
attached and configured

   }

   return status;
}

// write function
// This function will send up to the specified
number of bytes to
// the bulk in endpoint
unsigned char usbSlaveBoms_write(
   char *xfer,
   unsigned short num_to_write,
   unsigned short *num_written,
   usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   if(ctx->attached)
   {
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
      iocb.handle = ctx->in_ep;
      iocb.request.setup_or_bulk_transfer.buffer
= xfer;
      iocb.request.setup_or_bulk_transfer.size =
num_to_write;
      vos_dev_ioctl(ctx->handle, &iocb);
      if (num_written) // callers who don't care
might pass NULL here let's not crash!
         *num_written = iocb.request.setup_or_
bulk_transfer.bytes_transferred;
   } else {
      return USBSLAVEBOMS_ERROR;
   }

   return USBSLAVEBOMS_OK;
}

// read function
// This function reads up to num_to_read bytes
from the bulk out endpoint.
unsigned char usbSlaveBoms_read(
   char *xfer,
   unsigned short num_to_read,
   unsigned short *num_read,
   usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   if (ctx->attached)
   {
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
      iocb.handle = ctx->out_ep;
      iocb.request.setup_or_bulk_transfer.buffer
= xfer;
      iocb.request.setup_or_bulk_transfer.size =
num_to_read;
      vos_dev_ioctl(ctx->handle, &iocb);
      if (num_read) // allow caller to pass NULL
if they don't care how many bytes
         *num_read = iocb.request.setup_or_bulk_
transfer.bytes_transferred;
```

```
   } else {
      return USBSLAVEBOMS_ERROR;
   }

   return USBSLAVEBOMS_OK;
}

// write function
// This function will send up to the specified
number of bytes to
// the bulk in endpoint then it will stall
(short write)
unsigned char usbSlaveBoms_short_write(
   char *xfer,
   unsigned short num_to_write,
   unsigned short *num_written,
   usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   if(ctx->attached)
   {
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
      iocb.handle = ctx->in_ep;
      iocb.request.setup_or_bulk_transfer.buffer
= xfer;
      iocb.request.setup_or_bulk_transfer.size =
num_to_write;
      vos_dev_ioctl(ctx->handle, &iocb);
      if (num_written) // callers who don't care
might pass NULL here let's not crash!
         *num_written = iocb.request.setup_or_
bulk_transfer.bytes_transferred;
      usbslaveboms_stall_bulk_in(ctx);
   } else {
      return USBSLAVEBOMS_ERROR;
   }

   return USBSLAVEBOMS_OK;
}

// read function
// This function reads up to num_to_read bytes
from the bulk out endpoint.
// This function will stall the endpoint when
done
unsigned char usbSlaveBoms_short_read(
   char *xfer,
   unsigned short num_to_read,
   unsigned short *num_read,
   usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   if (ctx->attached)
   {
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_TRANSFER;
      iocb.handle = ctx->out_ep;
      iocb.request.setup_or_bulk_transfer.buffer
= xfer;
      iocb.request.setup_or_bulk_transfer.size =
num_to_read;
      vos_dev_ioctl(ctx->handle, &iocb);
      if (num_read) // allow caller to pass NULL
if they don't care how many bytes
         *num_read = iocb.request.setup_or_bulk_
transfer.bytes_transferred;
      usbslaveboms_stall_bulk_out(ctx);
   } else {
      return USBSLAVEBOMS_ERROR;
   }

   return USBSLAVEBOMS_OK;
}
// This function reads the 31 byte CBW from the
bulk out endpoint
// The value is returned directly into the boms_
cbw_t structure
unsigned char usbSlaveBoms_readCbw(
   boms_cbw_t *cbw,
   usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   if (ctx->attached)
   {
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
TRANSFER;
      iocb.handle = ctx->out_ep;
      iocb.request.setup_or_bulk_transfer.buffer
= (unsigned char*)cbw;
      iocb.request.setup_or_bulk_transfer.size =
31; //CBW is 31 bytes
      vos_dev_ioctl(ctx->handle, &iocb);

   } else {
      return USBSLAVEBOMS_ERROR;
   }

   return USBSLAVEBOMS_OK;

}

// USB Slave IOCTL function
unsigned char usbSlaveBoms_ioctl(common_ioctl_
cb_t *cb, usbSlaveBoms_context *ctx)
{
   unsigned char status = USBSLAVEBOMS_INVALID_
PARAMETER;

   switch (cb->ioctl_code)
   {
   case VOS_IOCTL_USBSLAVEBOMS_ATTACH:
```

```
      status = usbSlaveBoms_attach((VOS_HANDLE)
cb->set.data, ctx);
      break;

   case VOS_IOCTL_USBSLAVEBOMS_DETACH:
      usbSlaveBoms_detach(ctx);
      status = (unsigned char)USBSLAVEBOMS_OK;
      break;

   default:
      break;
   }

   return status;
}

// All commands on the control endpoint must be
acknowledged.
// This is done by sending a Zero Length Data
Packet ZLDP
// on the control in endpoint.
void ack_request(usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->in_ep0;
   iocb.request.setup_or_bulk_transfer.buffer =
(void *) 0;
   iocb.request.setup_or_bulk_transfer.size = 0;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// When initially attached to a host devices
have no address and are
// assigned an adddress after enumeration.  Then
the device is reset
// and fully enumerated.
void set_address_request(usbSlaveBoms_context
*ctx, unsigned char addr)
{
   usbslave_ioctl_cb_t iocb;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_ADDRESS;
   iocb.set = (void *) addr;
   vos_dev_ioctl(ctx->handle, &iocb);

   ack_request(ctx);
}

// Most devices support only one configuration.
However, every
// device must respond to this call asking them
to select a
// configuration even if it is the default.
void set_configuration_request(usbSlaveBoms_
context *ctx, unsigned char config)
{
   usbslave_ioctl_cb_t iocb;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_
CONFIGURATION;
   iocb.set = (void *) config;
   vos_dev_ioctl(ctx->handle, &iocb);

   ack_request(ctx);
}

// handle requests for a descriptor
void get_descriptor_request(usbSlaveBoms_context
*ctx)
{
   unsigned char *buffer;  // buffer for pass
thru to drive
   usbhost_ioctl_cb_t hc_ioctl;
   usbslave_ioctl_cb_t iocb;
   usb_deviceRequest_t *devReq;
   unsigned char hValue; // high byte of the
descriptor requested
   unsigned char lValue; // low byte of the
descriptor requested
   unsigned short wLength;
   unsigned short siz;
   uint32 ul_siz;
   unsigned char *src;
   unsigned char cond;

   devReq = (usb_deviceRequest_t *) ctx->setup_
buffer;

   hValue = devReq->wValue >> 8; // shift away
the low byte
   lValue = devReq->wValue & 0xff; // and away
the high byte

   wLength = devReq->wLength;

   switch (hValue) // the high byte determines
type of descriptor requested
   {
   case USB_DESCRIPTOR_TYPE_DEVICE:
      ul_siz = (uint32) wLength;
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
      iocb.handle = ctx->in_ep0;
      iocb.request.setup_or_bulk_transfer.buffer
= device_descriptor;
      iocb.request.setup_or_bulk_transfer.size =
(int16) ul_siz;
      vos_dev_ioctl(ctx->handle, &iocb);
      return;
      break;
```

```
   case USB_DESCRIPTOR_TYPE_CONFIGURATION:
      // host will initially ask for first 9
bytes of configuration descriptor
      // this descriptor header has the size of
the full descriptor which
      // is actually a composite of the
configuration/interface/endpoints.
      // Once host knows the complete descriptor
size it makes a second
      // request for the whole thing
      siz = wLength == 9?9:sizeof(config_
descriptor);
      ul_siz = (uint32) siz;

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
      iocb.handle = ctx->in_ep0;
      iocb.request.setup_or_bulk_transfer.buffer
= config_descriptor;
      iocb.request.setup_or_bulk_transfer.size =
(int16) ul_siz;
      vos_dev_ioctl(ctx->handle, &iocb);
      return;

   case USB_DESCRIPTOR_TYPE_STRING:

      if (lValue == 0) // language type
      {
         src = str0_descriptor;
         siz = sizeof(str0_descriptor);
      }
      else if (lValue == 1) // manufacturer
      {
         src = str1_descriptor;
         siz = sizeof(str1_descriptor);
      }
      else if (lValue == 2) // product
      {
         src = str2_descriptor;
         siz = sizeof(str2_descriptor);
      }
      else if (lValue == 3) // serial number
      {
         src = str3_descriptor;
         siz = sizeof(str3_descriptor);
      }

      cond = (unsigned char) (wLength != siz);

      if (siz > wLength) // don't return more
than was asked for
         siz = wLength;

      ul_siz = (uint32) siz;

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
      iocb.handle = ctx->in_ep0;
      iocb.request.setup_or_bulk_transfer.buffer
= src;
      iocb.request.setup_or_bulk_transfer.size =
(int16) ul_siz;
      vos_dev_ioctl(ctx->handle, &iocb);
      return;

   default:
      // if drive is connected get descriptor
from it
      if (ctx->flashConnected)
      {
         buffer = vos_malloc(wLength);
         hc_ioctl.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_SETUP_TRANSFER;
         hc_ioctl.handle.ep = hostBomsCtx-
>epCtrl;
         hc_ioctl.set = &(ctx->setup_buffer[0]);
         hc_ioctl.get = buffer; // descriptor
from drive
         vos_dev_ioctl(hostBomsCtx->hc, &hc_ioctl);

         iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
         iocb.handle = ctx->in_ep0;
         iocb.request.setup_or_bulk_transfer.
buffer = buffer;
         iocb.request.setup_or_bulk_transfer.
size = wLength;
         vos_dev_ioctl(ctx->handle, &iocb);

         vos_free(buffer);
      } else {
         // respond with Request Error
         set_control_ep_halt(ctx);
      }
   }
}

// This function will set a feature.  If it is
directed at an
// endpoint the endpoint will stall.
// Note: the endpoint passed in is the USB
endpoint from the setup request
// packet, not the VOS endpoint handle.
Unfortunately, most of the
// FTDI defined types are are just typedefs so
no real type checking
// is going on here.
void set_feature_request(usbSlaveBoms_context
*ctx, unsigned char ep)
{
   usbslave_ioctl_cb_t iocb;
   usbhost_ep_handle_ex hep; // host endpoint to
pass stall to
   usbhost_ioctl_cb_t host_ioctl_cb;
```

```
   ack_request(ctx); // first ack the request
then decide what to do

   // is this directed at an endpoint or the
device?
   if (ctx->setup_buffer[0] & USB_BMREQUESTTYPE_
ENDPOINT)
   {
      // directed to an endpoint
      //ep 1 is IN 2 is OUT on my fake device
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_STALL;
      iocb.ep = (ep & 0x02)?ctx->out_ep:ctx->in_
ep;
      vos_dev_ioctl(ctx->handle, &iocb);

      // if flash drive is attached pass along
request
      if (ctx->flashConnected)
      {
         //need to figure out which endpoint to
stall
         // if b7=1 then IN else OUT
         if (ep & 0x80)
         {
            hep = hostBomsCtx->epBulkIn;
         } else {
            hep = hostBomsCtx->epBulkOut;
         }
         host_ioctl_cb.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_SET_HOST_HALT;
         host_ioctl_cb.handle.ep = hep;
         // clear halt state on endpoint
         vos_dev_ioctl(hostBomsCtx->hc, &host_
ioctl_cb);
      }
   } else {
      // this is a device request
      host_ioctl_cb.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_SETUP_TRANSFER;
      host_ioctl_cb.handle.ep = hostBomsCtx-
>epCtrl;
      host_ioctl_cb.set = &(ctx->setup_
buffer[0]);
      host_ioctl_cb.get = NULL;
      vos_dev_ioctl(hostBomsCtx->hc, &host_
ioctl_cb);

   }
}

// This function is the complement to the set_
feature_request function.
void clear_feature_request(usbSlaveBoms_context
*ctx, unsigned char ep)
{
   usbslave_ioctl_cb_t iocb;
   usbhost_ep_handle_ex hep; // host endpoint to
pass stall to
   usbhost_ioctl_cb_t host_ioctl_cb;

   ack_request(ctx);

   // is this directed at an endpoint or the
device?
   if (ctx->setup_buffer[0] & USB_BMREQUESTTYPE_
ENDPOINT)
   {
      // directed to an endpoint
      //ep 1 is IN 2 is OUT on my fake device
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_CLEAR;
      iocb.ep = (ep & 0x02)?ctx->out_ep:ctx->in_
ep;
      vos_dev_ioctl(ctx->handle, &iocb);

      // if flash drive is attached pass along
request
      if (ctx->flashConnected)
      {
         //need to figure out which endpoint to
clear
         // if b7=1 then IN else OUT
         if (ep & 0x80)
         {
            hep = hostBomsCtx->epBulkIn;
         } else {
            hep = hostBomsCtx->epBulkOut;
         }
         host_ioctl_cb.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_CLEAR_HOST_HALT;
         host_ioctl_cb.handle.ep = hep;
         // clear halt state on endpoint
         vos_dev_ioctl(hostBomsCtx->hc, &host_
ioctl_cb);
      }
   } else {
      // this is a device request
      host_ioctl_cb.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_SETUP_TRANSFER;
      host_ioctl_cb.handle.ep = hostBomsCtx-
>epCtrl;
      host_ioctl_cb.set = &(ctx->setup_
buffer[0]);
      host_ioctl_cb.get = NULL;
      vos_dev_ioctl(hostBomsCtx->hc, &host_
ioctl_cb);

   }
}

// This function returns a 1 byte status code
for our endpoint
```

```
void get_ep_status(usbSlaveBoms_context *ctx, 
usbslave_ep_handle_t ep)
{
   usbslave_ioctl_cb_t iocb;
   char state;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_STATE;
   iocb.ep = ep;
   iocb.get = &state;
   vos_dev_ioctl(ctx->handle, &iocb);

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->out_ep0;
   iocb.request.setup_or_bulk_transfer.buffer = 
(void *) &state;
   iocb.request.setup_or_bulk_transfer.size = 1;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// This function will halt the control endpoint.
This function will only
// be called when an illegal request has been 
passed to the device.
void set_control_ep_halt(usbSlaveBoms_context *ctx)
{
   usbslave_ioctl_cb_t iocb;

   ack_request(ctx);

   // Performs a protocol stall on endpoint 0
   // Indicates that a request is unsupported
   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
ENDPOINT_STALL;
   iocb.ep = 0;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// This returns the device status.  Only the two 
lowest bits
// have any meaning here.  B0=1 means device is 
self-powered.
// B0 = 0 means the device is bus-powered.
// B1 = 1 means that remote wakeup is enabled
// This request can also be used for interface 
status which
// should just return all zeroes or endpoint 
status.  Endpoint
// status halted/cleared is indicated by the 
LSB.
void get_status_request(usbSlaveBoms_context 
*ctx)
{
   unsigned short status = 0;
   usbslave_ioctl_cb_t iocb;
   usbhost_ioctl_cb_t hc_ioctl;

   // if drive is connected get status from it
   if (ctx->flashConnected)
   {
      hc_ioctl.ioctl_code = VOS_IOCTL_USBHOST_
DEVICE_SETUP_TRANSFER;
      hc_ioctl.handle.ep = hostBomsCtx->epCtrl;
      hc_ioctl.set = &(ctx->setup_buffer[0]);
      hc_ioctl.get = &status; // status returned 
from the drive
      vos_dev_ioctl(hostBomsCtx->hc, &hc_ioctl);
   }

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->in_ep0;
   iocb.request.setup_or_bulk_transfer.buffer = 
&status;
   iocb.request.setup_or_bulk_transfer.size = 2;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// This function returns one byte.  If the byte 
is zero this indicates
// that the device is not ready.  Non-zero means 
everything is OK.
void get_configuration_request(usbSlaveBoms_
context *ctx)
{
   unsigned char status = 0;
   usbslave_ioctl_cb_t iocb;
   usbhost_ioctl_cb_t hc_ioctl;

   // if drive is connected get status from it
   if (ctx->flashConnected)
   {
      hc_ioctl.ioctl_code = VOS_IOCTL_USBHOST_
DEVICE_SETUP_TRANSFER;
      hc_ioctl.handle.ep = hostBomsCtx->epCtrl;
      hc_ioctl.set = &(ctx->setup_buffer[0]);
      hc_ioctl.get = &status; // status returned 
from the drive
      vos_dev_ioctl(hostBomsCtx->hc, &hc_ioctl);
   }

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->in_ep0;
   iocb.request.setup_or_bulk_transfer.buffer = 
&status;
   iocb.request.setup_or_bulk_transfer.size = 1;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// This request should never be sent to our 
device, but since this
// is a standard request we will accept it and
```

```
just do nothing.
void set_descriptor_request(usbSlaveBoms_context
*ctx, unsigned short wLength)
{
   unsigned char *buffer;
   usbslave_ioctl_cb_t iocb;
   usbhost_ioctl_cb_t hc_ioctl;

   ack_request(ctx);

   // We read in the info here but throw it away
   buffer = vos_malloc(wLength);
   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->out_ep0;
   iocb.request.setup_or_bulk_transfer.buffer =
buffer;
   iocb.request.setup_or_bulk_transfer.size = wLength;
   vos_dev_ioctl(ctx->handle, &iocb);
   vos_free(buffer);
}

// This function returns a single byte.  We will
just fake
// it and return zero.
void get_interface_request(usbSlaveBoms_context
*ctx)
{
   unsigned char status = 0;
   usbslave_ioctl_cb_t iocb;
   usbhost_ioctl_cb_t hc_ioctl;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->in_ep0;
   iocb.request.setup_or_bulk_transfer.buffer =
&status;
   iocb.request.setup_or_bulk_transfer.size = 1;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// We just ack this request.
void set_interface_request(usbSlaveBoms_context
*ctx)
{
   ack_request(ctx);
}

// This is the main handler function for
requests coming in on
// the control endpoint.  These are all requests
every USB device
// must respond to.
unsigned char standard_request(usbSlaveBoms_
context *ctx)
{
   usb_deviceRequest_t *devReq;
   unsigned char status = USBSLAVE_OK;
   unsigned char bReq;

   devReq = (usb_deviceRequest_t *) ctx->setup_buffer;
   bReq = devReq->bRequest;

   switch (bReq) // request is 1 byte value
   {
      case USB_REQUEST_CODE_GET_STATUS:
         get_status_request(ctx);
         break;

      case USB_REQUEST_CODE_SET_ADDRESS:
         set_address_request(ctx, devReq->wValue
& 0xff);
         break;

      case USB_REQUEST_CODE_GET_DESCRIPTOR:
         get_descriptor_request(ctx);
         break;

      case USB_REQUEST_CODE_SET_DESCRIPTOR:
         set_descriptor_request(ctx, devReq-
>wLength);
         break;

      case USB_REQUEST_CODE_SET_CONFIGURATION:
         set_configuration_request(ctx, devReq-
>wValue & 0xff);
         break;

      case USB_REQUEST_CODE_GET_CONFIGURATION:
         get_configuration_request(ctx);
         break;

      case USB_REQUEST_CODE_CLEAR_FEATURE:
         clear_feature_request(ctx, devReq->wIndex
>> 8);
         break;

      case USB_REQUEST_CODE_SET_FEATURE:
         set_feature_request(ctx, devReq->wIndex
>> 8);
         break;

      case USB_REQUEST_CODE_GET_INTERFACE:
         get_interface_request(ctx);
         break;

      case USB_REQUEST_CODE_SET_INTERFACE:
         set_interface_request(ctx);
         break;
      default:
         // force a protocol stall
         set_control_ep_halt(ctx);
         break;
   }
```

```
   return status;
}

void class_ack(usbSlaveBoms_context *ctx)
{
   ack_request(ctx);
}

void class_control_out(usbSlaveBoms_context
*ctx, char *buffer, unsigned short len)
{
   usbslave_ioctl_cb_t iocb;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->out_ep0;
   iocb.request.setup_or_bulk_transfer.buffer =
(void *) buffer;
   iocb.request.setup_or_bulk_transfer.size = len;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// This is one of two class requests for BOMS
devices.
// If we are not connected we return 0 which is
the most
// common situation.  Windows may not recognize
higher LUNs.
void get_max_lun_request(usbSlaveBoms_context
*ctx)
{
   unsigned char maxLun = 0;
   usbslave_ioctl_cb_t iocb;
   usbhost_ioctl_cb_t hc_ioctl;

   // if drive is connected get max lun from it
   if (ctx->flashConnected)
   {
      hc_ioctl.ioctl_code = VOS_IOCTL_USBHOST_
DEVICE_SETUP_TRANSFER;
      hc_ioctl.handle.ep = hostBomsCtx->epCtrl;
      hc_ioctl.set = &(ctx->setup_buffer[0]);
      hc_ioctl.get = &maxLun; // maximum LUN
from drive
      vos_dev_ioctl(hostBomsCtx->hc, &hc_ioctl);
   }

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_
TRANSFER;
   iocb.handle = ctx->in_ep0;
   iocb.request.setup_or_bulk_transfer.buffer =
&maxLun;
   iocb.request.setup_or_bulk_transfer.size = 1;
   vos_dev_ioctl(ctx->handle, &iocb);
}

// The USB standards define this function.
After it is called
// the drive should be ready to respond to CBWs.
The silly
// thing about this is that the drive should
always be ready
// for this and if it isn't then it probable
crashed long ago.
void boms_reset_request(usbSlaveBoms_context *ctx)
{
   usbhost_ioctl_cb_t hc_ioctl;

   ack_request(ctx);

   // forward this command to a drive if connected
   if (ctx->flashConnected)
   {
      hc_ioctl.ioctl_code = VOS_IOCTL_USBHOST_
DEVICE_SETUP_TRANSFER;
      hc_ioctl.handle.ep = hostBomsCtx->epCtrl;
      hc_ioctl.set = &(ctx->setup_buffer[0]);
      hc_ioctl.get = NULL; //no return value
from this call
      vos_dev_ioctl(hostBomsCtx->hc, &hc_ioctl);
   }
}

unsigned char class_request(usbSlaveBoms_context
*ctx)
{
   usb_deviceRequest_t *devReq;
   usbslave_ioctl_cb_t iocb;
   unsigned char status = USBSLAVE_OK;
   unsigned char bReq;

   devReq = (usb_deviceRequest_t *) ctx->setup_
buffer;
   bReq = devReq->bRequest;

   // force a protocol stall since there are no
class requests in BOMS
   switch (bReq)
   {
      case GET_MAX_LUN :
         get_max_lun_request(ctx);
         break;

      case BOMS_RESET :
         boms_reset_request(ctx);
         break;

      default:
         set_control_ep_halt(ctx);
   }

   return status;
}
```

in order to prevent infinite loops on Windows systems which stupidly would retry commands that were reported as unsupported.

The application code consists of two main threads. One thread is used to enumerate and communicate with an attached USB mass storage device. The other thread presents the device as a USB drive to an attached host and decides which commands to forward to the actual drive and which to emulate. It should be noted that the underlying FTDI library code may result in the creation of additional threads.

The code for the host enumeration thread is shown in Listing 1. The thread does not appear to do much based on the this short thread function. The majority of the work is performed by the FTDI libraries from the calls contained within the open_drivers and attach_drivers methods. Global handles to various devices are initialized in these function calls. These handles are used in order to forward commands to the USB drive from the other thread.

The thread method for the second thread is presented in Listing 2. This thread is little more than a large switch statement which calls various handler functions based on the received command. In the initial version of the write blocker only a couple of generalized handlers were used. This proved problematic, however, so these were replaced with individual handler methods for each command.

Each whitelisted command handler follows the general format of forwarding the command block wrapper (CBW) to the drive, receiving data from the drive, sending the data to the USB slave, receiving a command status wrapper (CSW) from the drive, and forwarding the CSW to the USB slave. Illegal command handlers are similar, but without the communication with the drive faked. The handler methods are provided in Listing 3.

The VNC2 microcontroller is somewhat unique in that it can be a host for USB devices. It is not unique in its ability to be embedded in a USB device, however. The FTDI libraries support easy implementation of a USB mass storage device which FTDI refers to as a USB Bulk-Only Mass Storage (BOMS) slave. To make our device appear as a flash drive when plugged into a computer we must simply create a series of handler functions and register our driver. One of the handlers will need to respond to standard requests and return device, configuration, and a collection of string descriptors. The required source code is provided in Listing 4.

The descriptors are defined at the top of the listing. The vendor and product IDs provided have been borrowed from an actual flash drive. Feel free to pick your favorite IDs. These IDs will be manipulated in the USB impersonator presented in the next article in this series. The primary handler functions call specialized handlers in order to keep the code somewhat neat.

ON THE WEB

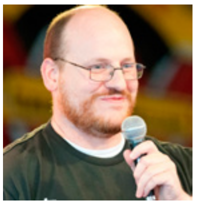
- *http://ftdichip.com* – The FTDI webpage where IDE and other software may be downloaded
- *http://www.usb.org/developers/docs/* – The official source for all things USB
- *http://lvr.com/usbc.htm* – USB expert Jan Axelson's main USB page
- *http://lvr.com/mass_storage.htm* – Jan Axelson's USB mass storage page
- *http://www.youtube.com/watch?v=3D9uGCvtoFo* – Phil's DEFCON XX Talk on USB impersonation
- *http://www.youtube.com/watch?v=CIVGzG0W-DM* – Phil's 44CON 2011 Talk on USB flash drive forensics
- *http://www.instructables.com/id/Cheap-and-Effective-USB-Write-Blocker/* – Phil's instructable on how to build a USB write blocker based on his BlackHat EU 2012 presentation
- *https://www.blackhat.com/html/bh-eu-12/bh-eu-12-archives.html#polstra* – Phil's BlackHat EU 2012 presentation on USB write blocking
- *http://www.concise-courses.com/infosec/20130404/* – Phil's Hacker Hotshots presentation on building a USB impersonator
- *http://ppolstra.blogspot.com* – Phil's blog
- *http://twitter.com/ppolstra* – Phil's Twitter page (@ppolstra)

SUMMARY

There are a number of options for blocking write operations to USB mass storage devices. In this article we presented a couple of compact write blockers which utilize the FTDI VNC2 microcontroller. We also briefly covered alternative methods of write blocking using software or somewhat pricey commercial hardware. Code and schematics for the devices presented are available online at *http://polstra.org*.

In the next article we will again make use of the FTDI VNC2. In that article we will discuss building an inexpensive and compact device which can impersonate other authorized USB drives. This is done in order to bypass endpoint security software which only permits whitelisted devices to be mounted. Should you have any question on this article, feel free to contact me on Twitter @ppolstra or at *http://polstra.org*.

ABOUT THE AUTHOR



Dr. Phil Polstra is currently an Associate Professor and Hacker in Residence at a private university in the Midwestern United States. He also teaches online computer security courses for a private university in Tempe, Arizona. His primary research focus over the last few years has been on the use of small, low-powered devices for forensics and penetration testing. As part of this work, he has developed his own custom pentesting Linux distribution that runs on the BeagleBoard family of ARM-based computers. Phil has presented his work on USB forensics and pentesting with small computers at several conferences around the world including BlackHat, DEFCON, 44CON, GrrCON, ForenSecure, and B-sides, to name a few. When not working he can be found spending time with family, hacking electronics, picking locks, flying, or building airplanes.

USING SYNCBEE TO SYNCHRONIZE YOUR

COMPUTER WITH A PORTABLE HARD DRIVE

by CHEN, JUN-CHENG (Jerry)

To avoid computer crashes and data loss, people jump on the "online backup" bandwagon to store their data to the Cloud in this data-booming era. Online backup is a good method for saving data. However, we need to be aware of problems when our data is stored in a risky remote space environment. Also note that Internet bandwidth can drastically slow down our backup time and work efficiency.

What you will learn:

- How to automatically back up your computers to a USB drive device.
- How to work your data from anywhere with a portable USB drive device.
- Why a USB device is useful in storing your data safely.

What you should know:

- Familiarity with basic use of Windows operation system

In this article, we will introduce a USB drive device, SyncBee, with a *FarStone* program installed. We will demonstrate how you can easily back up, recover, or synchronize between your computer and SyncBee with a few simple clicks. You can work from office to home and bring it everywhere. SyncBee can automatically search file types or you choose your files to back up or synchronize from the computer to SyncBee. Your data-loss-prevention plan can be executed without any time-consuming procedures.

FEATURES

Everyone has probably faced the catastrophe when data loss occurs. The reasons can be many; accidental deletions, viruses, file corruption and etc. Backing up can make extra copies of your data for safe keeping and restoring can recover your lost and corrupted file from the backed up data. With SyncBee, you can copy your computer files and folders to the hard drive and retrieve files back with peace-of-mind. SyncBee can also facilitate your work efficiency for its synchronization. Besides, you can synchronize your office and home computers to work without interruptions. This device has intuitive features with useful advantages.

- Back up computers to a safe personal hard drive
- Back up and synchronize multiple computers
- Plug-in and play, no installation needed, no learning curve
- Back up without interrupting your work
- Automatically search and categorize all of your personal files

- Back up open files
- No change of any registries on your system
- One-way or two-way synchronization
- Back up only changes to your files – incremental backup
- Automatically merge recovery points to save space
- Password protection
- High-speed USB 3.0 drive (USB 2.0 compatible)

SYNCBEE SPECIFICATIONS

It provides multi-lingual interfaces of English, Simplified and Traditional Chinese, Japanese, German, French, Italian, Spanish, Portuguese, Russian, Polish, and Korean.

Table 1. *SyncBee Specifications*

Interface	• USB3.0 Super Speed / USB2.0 High Speed / 2.5" SATA HDD
Drive Interface Support	• USB3.0 5Gbps(≥70MB/s) USB 2.0 480Mb/s(≥30MB/s)
OS supported	• Windows XP / Windows Vista / Windows 7 / Windows 8
Power	• Bus power
LED indication	• USB 3.0 Orange USB2.0 Green; Access – Blinking
Accessories	• USB3.0 Micro B cable
Dimension	• 130.5 (W) X 78.5 (L) X 13.0 (H) mm
Certification	• CE, FCC
RoHS Compliance	• Yes

HOW TO BACK UP

- Plug SyncBee in your computer, the easy-to-use interface will show up. Or you can enter "My Computer" to find the icon of SyncBee. Double click the icon to enter the main menu.

Figure 1. *Main Menu*

- Click "Options" to set up your preference. Set up for the first time and just click "Backup Now" in the future. Choose files types to back up. You can also add types or files/folders in the right-down buttons. Click "OK" and you will go back to the main menu.

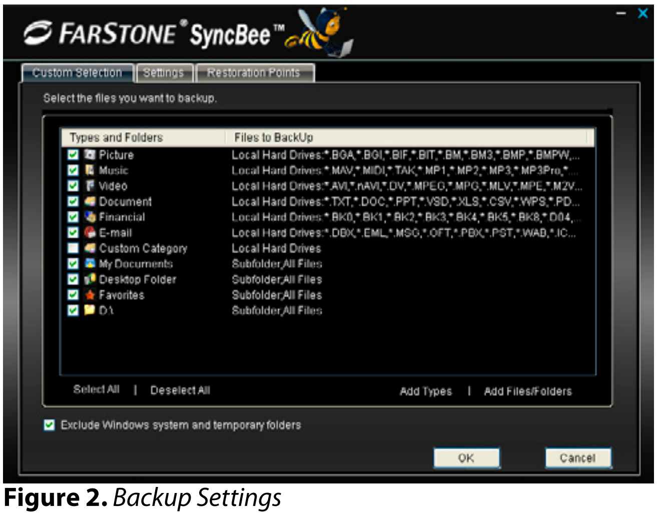


Figure 2. *Backup Settings*

- Click "Backup Now" and your files will be backed up. Note that SyncBee does not back up Windows files, application, and other executable files to save your time and space.

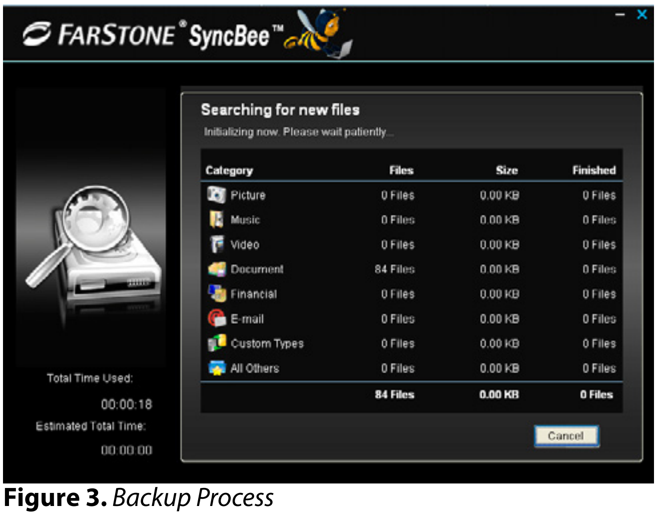


Figure 3. *Backup Process*

During the backup process, you can see the details of files and time. You can finish your backup with these simple clicks.

HOW TO RESTORE

- Click "Restore" in the main menu.
- Because you can use SyncBee on different computers, choose one target computer to restore.
- Select a restore point.
- Browse files you want to restore.
- You can restore files to the desktop, to the original location, or a new location.

- Click “OK” to restore files.

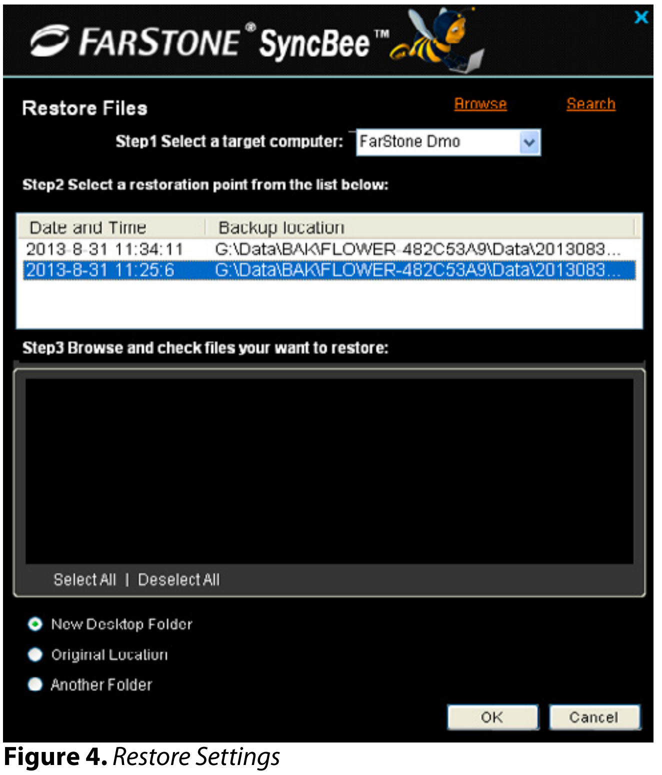


Figure 4. *Restore Settings*

HOW TO SYNCHRONIZE

Click “Sync Options” to set up your preference. Set up for the first time and just click “Sync Now” in the future.

- The default directories for synchronization are Documents, Desktop, Favorites and Outlook files. You can unselect items or add your preferred folders.

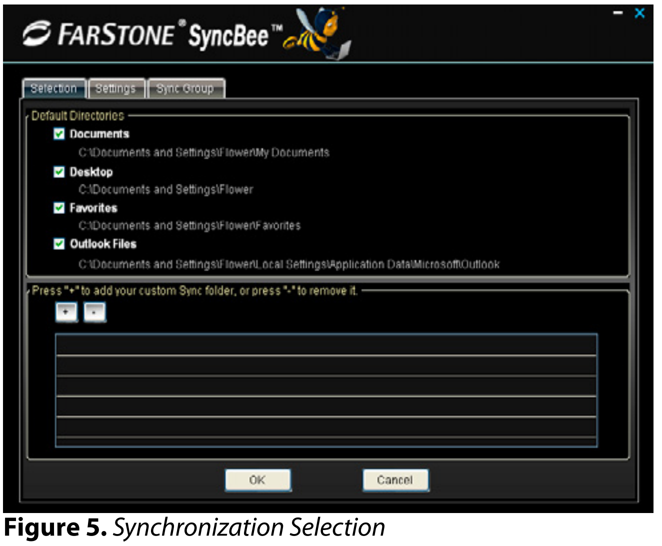


Figure 5. *Synchronization Selection*

- In the up navigation bar, select “Settings.” You can decide what to do in the scenarios below.
  - When there are duplicate files names found during synchronization.
  - When user names of source and target are different USBs
  - When a disk drive can not be found in target USB

Figure 6. *Synchronization Settings*

- In the up navigation bar, select “Sync Group.” You can create a sync group to synchronize all your computers and you can keep all the files in the latest status.

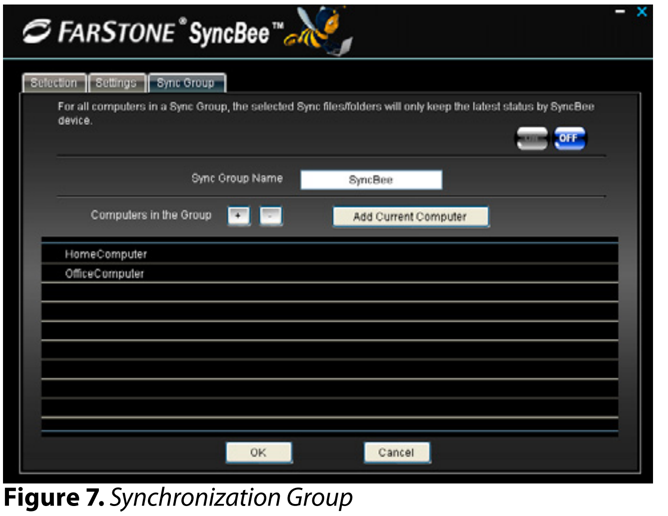


Figure 7. *Synchronization Group*

After setting up, you go back to the main menu and click “Sync Now.” Then you can see “Two way

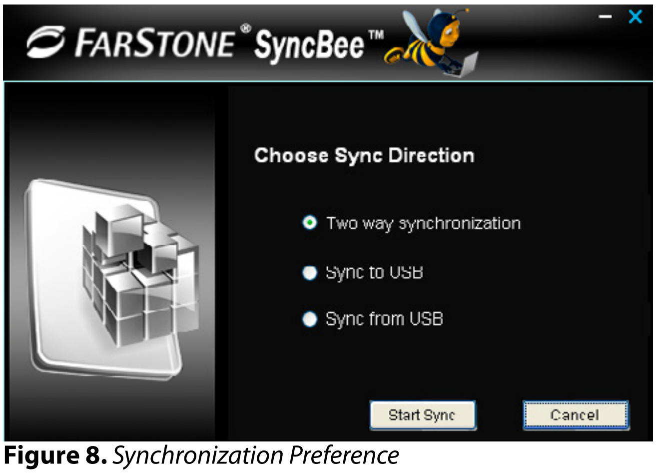


Figure 8. *Synchronization Preference*

synchronization", "Sync to USB", "Sync from USB." Select one and you will finish your synchronization (Figure 8).

HOW TO ENCRYPT YOUR DATA

You can click the icon in the main menu and enter your password (at least 6 characters).

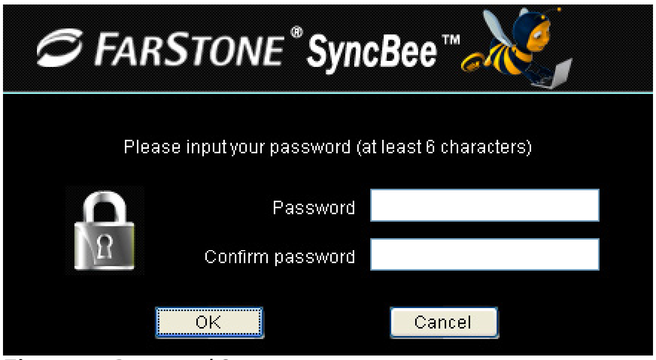


Figure 9. *Password Setup*

When you use SyncBee next time, it will request you to enter password.

CONCLUSION

There are a variety of methods to back up and synchronize your data and to keep worries of data loss at bay. You can consider backing up and synchronizing your computers to a hard drive device. However, manually store data to a hard drive could be easy but

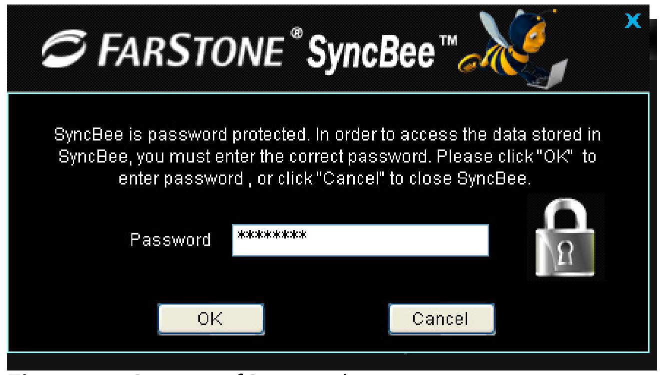
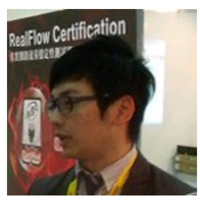


Figure 10. *Request of Password*

not wise enough. A good hard drive device can well organize your data and wisely bring them back when you need them. For more information, you can log on FarStone's website and read more about the best-selling portable backup device, SyncBee. *http://www.farstone.com/hardware/syncbee.htm*.

ABOUT THE AUTHOR



The author has an MS degree in Industrial Engineering from National Taiwan University and has been working in high-tech and information industry for several years. Currently he is working as a Business Development Manager at FarStone Technology, Inc. and in charge of global business planning and development. You can contact him at jerry@farstone.com.

HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

PART 5: IMPERSONATING USB DEVICES

by Dr. Phil Polstra

USB mass storage devices have become the standard for backup and transfer of files. The popularization of this media has led to challenges for forensic specialists trying to find data on fixed memory storage media instead of traditional magnetic media. In this firth part of a multi-part series a simple and inexpensive device for bypassing some endpoint security software by allowing any USB mass storage device to present itself as an authorized (whitelisted) device is presented.

What you will learn:

- How to block writes to USB mass storage devices
- How to use FTDI microcontrollers
- Differences in how various operating systems handle USB mass storage devices
- How to use microcontroller timers
- How to impersonate USB devices

What you should know:

- A basic understanding of programming in C
- A basic understanding of USB mass storage devices (possibly from the first three articles in this series)
- A basic understanding of the USB write blocker presented in the fourth article in this series

Recently some organizations have begun to take steps to help deter the outward flow of information. Because USB mass storage devices are now ubiquitous they are a commonly used in the extraction of proprietary or classified data. Several companies now offer endpoint security software which allows only authorized USB mass storage devices to be mounted. Additionally, some administrators use operating system policies to accomplish the same effect. Those familiar with network security will recognize this as the USB equivalent of MAC filtering. Just as MAC filtering is easily bypassed, so are attempts at blocking all but authorized USB mass storage devices. This article will present a simple device which could be used to bypass such efforts.

SHORT STORY

The USB write blocker presented in the fourth article in this series was originally presented at BlackHat Europe 2012. Just before BlackHat that year, I was approached by someone I had met at 44Con in London the previous September. He asked for my advice as to the best way to bypass endpoint security software by impersonating authorized devices. Serendipitously 90% of the code for such a device already existed in my USB write blocker. He was too busy to pursue the last 10% himself so I implemented the solution presented here which first appeared publicly at DEFCON 2012.

The completed device (with a FTDI programmer board attached) is shown in Figure 1.

FORENSIC SERVICES

FRAUD, INVESTIGATION AND DISPUTE

The FID department from BDO Brazil provides public, private-sector and not-for-profit clients with a wide range of investigations and compliance services. We are experienced in assisting major clients and their counsel with significant investigations across a variety of industries. We also field teams of seasoned pro-fessionals domestically and globally in rapid response to allegations of suspected fraud, wrongdoing, corpo-rate misconduct and in the protection and recovery of assets.

OUR SERVICES

- Financial institution consulting
- Fraud prevention
- Investigative Due Diligence
- IPSIGs & Monitorships
- Mergers & acquisitions
- Mortgage lending investigations
- Regulatory compliance
- Anti-corruption compliance
- Investigations e Background check
- Anti-money laundering
- Computer forensics & e-discovery
- Corporate investigations
- Employee misconduct investigations
- Emergency Economic Stabilization Act & TARP
- Intellectual Property

Our professionals provide clients with high-quality and individualized service, drawing upon experience gai-ned from a variety of former disciplines and professional backgrounds, including positions with federal law enforcement, corporate investigation departments, accounting regulatory bodies and financial reporting departments in publicly traded companies. Our well-credentialed professionals include certified fraud exa-miners, forensic accountants, former federal and local prosecutors and law enforcement officers and a-gents, licensed private investigators and computer forensic specialists.

Brazil
www.bdobrazil.com.br

Latin America
www.bdolatam.con

World
www.bdointernational.com

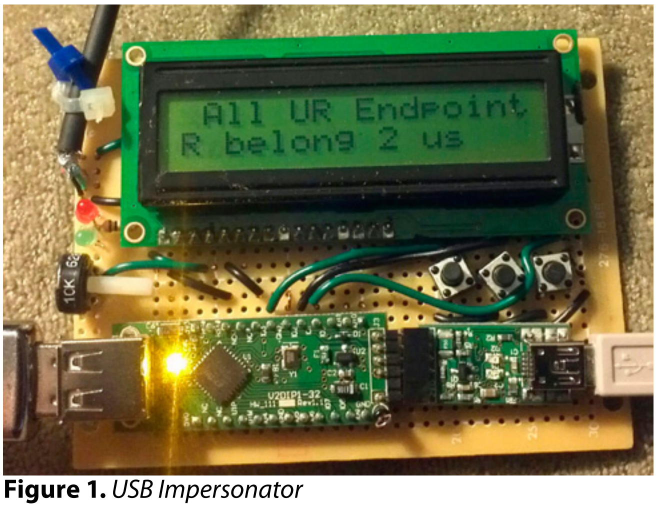


Figure 1. *USB Impersonator*

HIGH LEVEL DESIGN

As previously stated, the USB impersonator is based on the USB write blocker which was presented in the previous article in this series. The V2DIP1-32 development board from FTDI (in the lower left corner of Figure 1) provides sufficient GPIO lines to drive an LCD display, light 2 LEDs, and receive input from three buttons. The three buttons are used to toggle write blocking and to scroll through the list of 500 most common vendor/product ID (VID/PID) combinations. The 10K potentiometer in Figure 1 is used to adjust the contrast on the LCD screen. The two LEDs are used to indicate write-blocking status.

The impersonator has two modes of operation. In manual mode two of the buttons are used to select a known good VID/PID combination. In automatic mode the impersonator starts a timer as soon as the PC starts communicating with it. If the PC stops talking to the impersonator the timer expires which is interpreted as blocking so the device is disconnected via software and reconnected using the next VID/PID combination in the list.

The impersonator adds two new threads to the write blocker. One thread is used to detect button

Listing 1. *Timer thread method*

```
// This timer is used to determine if our device
has successfully connected
// When the device is first enumerated this
timer is set.  This thread then
// blocks till the timer expires.  When the
timer expires we increment the
// index into our VID/PID list.  If the device
is connected the timer is
// cancelled.
void timer()
{
   tmr_ioctl_cb_t tmr_iocb;
   usbslave_ioctl_cb_t siocb;

   // here we set up the timer, but don't start
it
   // timer is started by the USB slave driver
when someone
   // starts talking to it
   hTimer = vos_dev_open(TIMER0);
   tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_SET_
TICK_SIZE;
   tmr_iocb.param = TIMER_TICK_MS;
   vos_dev_ioctl(hTimer, &tmr_iocb);
   tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_SET_
COUNT;
   tmr_iocb.param = 1000; // 1s
   vos_dev_ioctl(hTimer, &tmr_iocb);
   tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_SET_
DIRECTION;
   tmr_iocb.param = TIMER_COUNT_DOWN;
   vos_dev_ioctl(hTimer, &tmr_iocb);
   tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_SET_
MODE;
   tmr_iocb.param = TIMER_MODE_SINGLE_SHOT;
   vos_dev_ioctl(hTimer, &tmr_iocb);

   // if our device is connected this never gets
past blocking call
   while (1)
   {
      tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_
WAIT_ON_COMPLETE;
      vos_dev_ioctl(hTimer, &tmr_iocb);  //only
returns if enumeration doesn't complete
      if (autoMode)
      {
         vos_lock_mutex(&vidPidMutex);
         currentVidPidIndex += 2;
         if (currentVidPidIndex >
sizeof(vidPid))
            currentVidPidIndex = 0;
         vos_unlock_mutex(&vidPidMutex);
         // Disconnect the slave device
         siocb.ioctl_code = VOS_IOCTL_USBSLAVE_
DISCONNECT;
         siocb.set = (void *) 0;
         vos_dev_ioctl(hUSBSLAVE_1,&siocb);
         // Now reconnect with new VID/PID
         siocb.ioctl_code = VOS_IOCTL_USBSLAVE_
CONNECT;
         siocb.set = (void *) 0;
         vos_dev_ioctl(hUSBSLAVE_1,&siocb);
      }
   }
}
```

Listing 2. *Descriptor request handler method*

```
// handle requests for a descriptor
void get_descriptor_request(usbSlaveBoms_context
*ctx)
{
   unsigned char *buffer;  // buffer for pass
thru to drive
   usbhost_ioctl_cb_t hc_ioctl;
   usbslave_ioctl_cb_t iocb;
   usb_deviceRequest_t *devReq;
   unsigned char hValue; // high byte of the
descriptor requested
   unsigned char lValue; // low byte of the
descriptor requested
   unsigned short wLength;
   unsigned short siz;
   uint32 ul_siz;
   unsigned char *src;
   unsigned char cond;
   tmr_ioctl_cb_t tmr_iocb;

   devReq = (usb_deviceRequest_t *) ctx->setup_
buffer;

   hValue = devReq->wValue >> 8; // shift away
the low byte
   lValue = devReq->wValue & 0xff; // and away
the high byte

   wLength = devReq->wLength;

   switch (hValue) // the high byte determines
type of descriptor requested
   {
   case USB_DESCRIPTOR_TYPE_DEVICE:
      ul_siz = (uint32) wLength;
      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
      iocb.handle = ctx->in_ep0;
      // update the device descriptor VID/PID
from our list
      vos_lock_mutex(&vidPidMutex);
      device_descriptor[8] =
vidPid[currentVidPidIndex] & 0xff;
      device_descriptor[9] =
vidPid[currentVidPidIndex] >> 8;
      device_descriptor[10] =
vidPid[currentVidPidIndex+1] & 0xff;
      device_descriptor[11] =
vidPid[currentVidPidIndex+1] >> 8;
      vos_unlock_mutex(&vidPidMutex);
      //update LCD
      update_lcd_vidpid();

      iocb.request.setup_or_bulk_transfer.buffer
= device_descriptor;
      iocb.request.setup_or_bulk_transfer.size =
(int16) ul_siz;
      vos_dev_ioctl(ctx->handle, &iocb);

      // start this timer and if it is not
killed then the next VID/PID will be selected
      // this is done to detect an unsuccessful
enumeration which is assumed
      // to result from endpoint security
blocking
      if (autoMode)
      {
         tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_
START;
         vos_dev_ioctl(hTimer, &tmr_iocb);
      }
      return;
      break;

   case USB_DESCRIPTOR_TYPE_CONFIGURATION:
      // host will initially ask for first 9
bytes of configuration descriptor
      // this descriptor header has the size of
the full descriptor which
      // is actually a composite of the
configuration/interface/endpoints.
      // Once host knows the complete descriptor
size it makes a second
      // request for the whole thing
      siz = wLength == 9?9:sizeof(config_
descriptor);
      ul_siz = (uint32) siz;

      iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
      iocb.handle = ctx->in_ep0;
      iocb.request.setup_or_bulk_transfer.buffer
= config_descriptor;
      iocb.request.setup_or_bulk_transfer.size =
(int16) ul_siz;
      vos_dev_ioctl(ctx->handle, &iocb);

      // stop the timer because we are being
asked to enumerate
      if(autoMode)
      {
         tmr_iocb.ioctl_code = VOS_IOCTL_TIMER_
STOP;
         vos_dev_ioctl(hTimer, &tmr_iocb);
      }
      return;

   case USB_DESCRIPTOR_TYPE_STRING:

      if (lValue == 0) // language type
      {
         src = str0_descriptor;
```

presses. This is a low priority thread. The second additional thread is used to handle the timer for function for automatic mode. The timer is set to one second when the PC begins communicating with the impersonator. If the PC doesn't complete the enumeration process before the timer expires a handler function is called. This handler function disconnects the impersonator, increments the current VID/PID pointer, and then reconnects the impersonator.

TIMERS

Like many microcontrollers, the FTDI Vinculum II (VNC2) supports multiple hardware timers. The timer thread function is presented in Listing 1. Note that a blocking call is made and if the timer is reset the code to change the VID/PID is never reached. The code which returns descriptors to the PC and also sets and resets the timers is presented in Listing 2. Complete code may be found at *https://github.com/ppolstra/usb-impersonator*.

GENERAL PURPOSE INPUT AND OUTPUT

The changes presented so far are really all that is required to transform a write blocker into an impersonator. While such a device could be somewhat useful, having displays to indicate write-blocking status, current VID/PID, and allowing input to change mode of operation makes the device much more useful. Enhancing the basic impersonator requires an understanding of how to do general purpose input and output (GPIO) with microcontrollers.

The first thing to understand about doing GPIO with microcontrollers is that one must deal with ports not pins. Those familiar with the AVR microcontrollers as found in the Arduino line or products might dispute this statement, but they would be

```
          siz = sizeof(str0_descriptor);
       }
       else if (lValue == 1) // manufacturer
       {
          src = str1_descriptor;
          siz = sizeof(str1_descriptor);
       }
       else if (lValue == 2) // product
       {
          src = str2_descriptor;
          siz = sizeof(str2_descriptor);
       }
       else if (lValue == 3) // serial number
       {
          src = str3_descriptor;
          siz = sizeof(str3_descriptor);
       }

       cond = (unsigned char) (wLength != siz);

       if (siz > wLength) // don't return more
than was asked for
          siz = wLength;

       ul_siz = (uint32) siz;

       iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
       iocb.handle = ctx->in_ep0;
       iocb.request.setup_or_bulk_transfer.buffer
= src;
       iocb.request.setup_or_bulk_transfer.size =
(int16) ul_siz;
       vos_dev_ioctl(ctx->handle, &iocb);
       return;

    default:
      // if drive is connected get descriptor
from it
      if (ctx->flashConnected)
      {
         buffer = vos_malloc(wLength);
         hc_ioctl.ioctl_code = VOS_IOCTL_
USBHOST_DEVICE_SETUP_TRANSFER;
         hc_ioctl.handle.ep = hostBomsCtx-
>epCtrl;
         hc_ioctl.set = &(ctx->setup_buffer[0]);
         hc_ioctl.get = buffer; // descriptor
from drive
         vos_dev_ioctl(hostBomsCtx->hc, &hc_
ioctl);

         iocb.ioctl_code = VOS_IOCTL_USBSLAVE_
SETUP_TRANSFER;
         iocb.handle = ctx->in_ep0;
         iocb.request.setup_or_bulk_transfer.
buffer = buffer;
         iocb.request.setup_or_bulk_transfer.
size = wLength;
         vos_dev_ioctl(ctx->handle, &iocb);

         vos_free(buffer);
      } else {
         // respond with Request Error
         set_control_ep_halt(ctx);
      }
   }
}
```

Listing 3. *GPIO multiplexer setup*

```
void iomux_setup(void)
{
   unsigned char packageType;
   tmr_context_t tmrCtx; // timer context for timer to sequence VID/PID

   packageType = vos_get_package_type();

   // This is for the smaller package.  This is probably what you want
   // for a couple of reasons.  First off the LCD doesn't fit on a
   // Vinco shield very well.  Also, you can leave off the buttons
   // and LCD if you don't want all the functionality, those items
   // will simply be ignored.  If you leave out the LCD then you could
   // still build a pretty compact device with just a button and LEDs
   // for the write protect functionality.
   if (packageType == VINCULUM_II_32_PIN)
   {
      // Debugger to pin 11 as Bi-Directional.
      vos_iomux_define_bidi(199, IOMUX_IN_DEBUGGER, IOMUX_OUT_DEBUGGER);
      // GPIO_Port_E_1 to pin 12 as Input.
      vos_iomux_define_input(12, IOMUX_IN_GPIO_PORT_E_1); //VID+ button
      // GPIO_Port_E_2 to pin 14 as Input.
      vos_iomux_define_input(14, IOMUX_IN_GPIO_PORT_E_2); //VID- button
      // GPIO_Port_E_3 to pin 15 as Input.
      vos_iomux_define_input(15, IOMUX_IN_GPIO_PORT_E_3); //write protect button
      // GPIO_Port_B_0 to pin 23 as Output.
      vos_iomux_define_output(23, IOMUX_OUT_GPIO_PORT_B_0); //LCD DB4
      // GPIO_Port_B_1 to pin 24 as Output.
      vos_iomux_define_output(24, IOMUX_OUT_GPIO_PORT_B_1); //LCD DB5
      // GPIO_Port_B_2 to pin 25 as Output.
      vos_iomux_define_output(25, IOMUX_OUT_GPIO_PORT_B_2); //LCD DB6
      // GPIO_Port_B_3 to pin 26 as Output.
      vos_iomux_define_output(26, IOMUX_OUT_GPIO_PORT_B_3); //LCD DB7
      // GPIO_Port_B_4 to pin 29 as Output.
      vos_iomux_define_output(29, IOMUX_OUT_GPIO_PORT_B_4); //LCD RS
      // GPIO_Port_B_5 to pin 30 as Output.
      vos_iomux_define_output(30, IOMUX_OUT_GPIO_PORT_B_5); //LCD E
      // GPIO_Port_B_6 to pin 31 as Output.
      vos_iomux_define_output(31, IOMUX_OUT_GPIO_PORT_B_6); //Green LED
      // GPIO_Port_B_7 to pin 32 as Output.
      vos_iomux_define_output(32, IOMUX_OUT_GPIO_PORT_B_7); //Red LED
   }

   // This is for people who hate to solder and want to base their device
   // on the Vinco.  Personally, I think that basing your device off the
   // 32-pin V2DIP1-32 is a better idea.  Note that setup for pins 40
   // and 41 is only needed for the Vinco, so if you are running
   // a dev board with a 64-pin chip it is not really needed.
   if (packageType == VINCULUM_II_64_PIN)
   {
      // Debugger to pin 11 as Bi-Directional.
      vos_iomux_define_bidi(199, IOMUX_IN_DEBUGGER, IOMUX_OUT_DEBUGGER);

      // GPIO_Port_E_1 to pin 12 as Input.
      vos_iomux_define_input(12, IOMUX_IN_GPIO_PORT_E_1); //VID+ button
      // GPIO_Port_E_2 to pin 13 as Input.
```

wrong to do so. The Processing language used by the Arduino allows the programmer to set modes on individual pins. The underlying libraries deal with individual pins in predefined ports, however. Because there are more ports with associated pins (up to 8 per port) than there are pins and there is some flexibility in associating pins with ports, a multiplexer (MUX) is used.

At program startup appropriate function calls are made to associate pins with ports which are labeled with letters and pins within the ports (labeled 0 through 7). Additionally, the pin mode (input, output, bidirectional) can be reset from the default which is typically input. Listing 3 shows these functions calls for the impersonator. In this application port E is used for the buttons, and the LCD and LEDs are connected to port B.

BUTTONS

Pushbuttons are obviously input devices. Normally open pushbuttons (the most common type) are traditional connected to the positive power rail (+5V in our case) through a pull up resistor (5-10k Ohms) on one side and ground on the other. The microcontroller input pin is connected after the resistor on the high voltage side. This is known as an active low switch. When the switch is not pressed the voltage on the pin is approximately 5V. When the switch is pushed the voltage drops to approximately 0V. The pull up resistor limits the current flowing through the switch (shorting directly to ground would be a very bad idea).

As previously mentioned, microcontrollers deal with ports, not pins. The value of the port is read and an appropriate mask is applied to determine if a particular pin was pressed. When reading switches mechanical bouncing in which the internal parts of the switch vibrate causing a series of openings and closings leads to complications. Dealing with this situation is known as debouncing a switch. In the impersonator we use a very simple debouncing method of just waiting for a set period of time and rereading the switch before taking any action. The code for handling the buttons is shown in Listing 4.

USING LEDS

Connecting LEDs to a microcontroller is fairly straightforward. Connect the positive LED pin (the one with the longer lead) to the microcontroller pin. Connect the other side to ground through a current limiting resistor. You may need to do some

```
        vos_iomux_define_input(13, IOMUX_IN_GPIO_PORT_E_2); //VID- button
        // GPIO_Port_E_3 to pin 14 as Input.
        vos_iomux_define_input(14, IOMUX_IN_GPIO_PORT_E_3); //write protect button
        // for Vinco need to set pins to output for LED and power on host
        // both are active low
        vos_iomux_define_output(40, IOMUX_OUT_GPIO_PORT_E_5); // USB host LED
        vos_iomux_define_output(41, IOMUX_OUT_GPIO_PORT_E_6); // USB host power
        // GPIO_Port_B_0 to pin 24 as Output.
        vos_iomux_define_output(24, IOMUX_OUT_GPIO_PORT_B_0); //LCD DB4
        // GPIO_Port_B_1 to pin 25 as Output.
        vos_iomux_define_output(25, IOMUX_OUT_GPIO_PORT_B_1); //LCD DB5
        // GPIO_Port_B_2 to pin 26 as Output.
        vos_iomux_define_output(26, IOMUX_OUT_GPIO_PORT_B_2); //LCD DB6
        // GPIO_Port_B_3 to pin 27 as Output.
        vos_iomux_define_output(27, IOMUX_OUT_GPIO_PORT_B_3); //LCD DB7
        // GPIO_Port_B_4 to pin 28 as Output.
        vos_iomux_define_output(28, IOMUX_OUT_GPIO_PORT_B_4); //LCD RS
        // GPIO_Port_B_5 to pin 29 as Output.
        vos_iomux_define_output(29, IOMUX_OUT_GPIO_PORT_B_5); //LCD E
        // GPIO_Port_B_6 to pin 31 as Output.
        vos_iomux_define_output(31, IOMUX_OUT_GPIO_PORT_B_6); //Green LED
        // GPIO_Port_B_7 to pin 32 as Output.
        vos_iomux_define_output(32, IOMUX_OUT_GPIO_PORT_B_7); //Red LED
    }
    // setup the timer used to cycle through VID/PID
    tmrCtx.timer_identifier = TIMER_0;
    tmr_init(TIMER0, &tmrCtx);

}
```

Listing 4. *Button handling thread method*

```
// This function runs in a low priority thread.
Its only
// function is to read the buttons to see if
someone has pressed
// them to toggle write protect or change the
VID/PID.
// This could have been implemented using GPIO
lines with
// interupt capabilities, but interupting the
USB threads might
// not be a great idea.  Additionally, enabling
interupts introduces
// additional overhead.
void handleButtons()
{
   unsigned char buttonBits, leds, counter=0,
firstTime=1;
   char str1[17], str2[17];

   while (1)
   {
      vos_dev_read(hGPIO_PORT_E, &buttonBits, 1,
NULL);

      // if this is non-zero, somebody is
pushing buttons
      // we prioritize the write protect button
then +, then -
      while (~buttonBits)
      {
         if ((buttonBits ^ 0xF7)==0) // write
protect button pressed
         {
            // since this is pretty serious
business we require
            // a long keypress to toggle
            vos_delay_msecs(1000);
            if((buttonBits ^ 0xF7)==0)
            {
               writeProtect = (~writeProtect & 0x01);
               leds = writeProtect?led_green:led_red;
               vos_dev_write(hGPIO_PORT_B, &leds, 1,
NULL);
            }
         } else if((buttonBits ^ 0xFB)==0) //
VID/PID +
         {
            autoMode=0; // disable VID/PID scan
            counter = 0; // reset the counter
            vos_lock_mutex(&vidPidMutex);
            currentVidPidIndex += 2;
            if (currentVidPidIndex > vidPidSize)
               currentVidPidIndex = 0;
            vos_unlock_mutex(&vidPidMutex);
            update_lcd_vidpid();
         } else if((buttonBits ^ 0xFD)==0) //
VID/PID -
         {
            autoMode=0; // disable VID/PID scan
            counter = 0; // reset the counter
            vos_lock_mutex(&vidPidMutex);
            currentVidPidIndex -= 2;
            if (currentVidPidIndex < 0)
               currentVidPidIndex = 0;
            vos_unlock_mutex(&vidPidMutex);
            update_lcd_vidpid();
         }
         vos_dev_read(hGPIO_PORT_E, &buttonBits,
1, NULL);
         vos_delay_msecs(50);
      }

      vos_delay_msecs(100);
      // if we haven't hit a button for 5
seconds then it is time to
      // move on and start using the device
      if (firstTime)
         counter++; // increment the counter
      if (counter > 50)
      {
         firstTime=0;
         vos_signal_
semaphore(&setupDoneSemaphore);
      }
   }
}
```

Listing 5. *LED methods*

```
// LED port masks
#define led_green 0x80
#define led_red 0x40

void light_red_led(VOS_HANDLE hLED)
{
   unsigned char toggle;
   // first we read port to not mess up the LCD
display
   vos_dev_read(hLED, &toggle, 1, NULL);
   toggle &= (~led_green);
   toggle |= led_red;
   vos_dev_write(hLED, &toggle, 1, NULL);
}

void light_green_led(VOS_HANDLE hLED)
{
   unsigned char toggle;
   vos_dev_read(hLED, &toggle, 1, NULL);
   toggle &= (~led_red);
   toggle |= led_green;
   vos_dev_write(hLED, &toggle, 1, NULL);
}
```

Listing 6. *LED methods*

```
/// LCD control signals
#define lcd_rs    0x10
#define lcd_e     0x20

// Send a command to our LCD display
void write_lcd_cmd(VOS_HANDLE hLCD, unsigned char byte)
{
    unsigned char cmd, leds;
   // first read state to not mess up LEDs on same port
   leds = writeProtect?led_green:led_red;

    // Write High nibble data to LCD
    cmd = (((byte >> 4) &0x0F) | lcd_e);
   cmd = (cmd &(~lcd_rs)) | leds; // Select Registers
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Write Low nibble data to LCD
    cmd = ((byte &0x0F) | lcd_e);
   cmd = (cmd &(~lcd_rs)) | leds; // Select Registers
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    vos_delay_msecs(1);

}

// Send data to LCD display
void write_lcd_data(VOS_HANDLE hLCD, unsigned char byte)
{
    unsigned char cmd, leds;

   // first read state to not mess up LEDs on same port
   leds = writeProtect?led_green:led_red;

   // Write High nibble data to LCD
    cmd = (((byte >> 4)&0x0F) | lcd_rs);
    cmd = (cmd | lcd_e) | leds; // Select DDRAM
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Write Low nibble data to LCD
    cmd = ((byte & 0x0F) | lcd_rs);
    cmd = (cmd | lcd_e) | leds; // Select DDRAM
    vos_dev_write(hLCD,&cmd,1,NULL);
    // Toggle 'E' pin
    cmd &= (~lcd_e);
    vos_dev_write(hLCD,&cmd,1,NULL);
    vos_delay_msecs(1);

}

// Write a string at the current cursor position
void write_lcd_str(VOS_HANDLE hLCD, char *str)
{
    while(*str != '\0')
    {
        write_lcd_data(hLCD, (unsigned char*)*str);
         ++str;
    }
}

// Attempt to init the LCD display
void lcd_ini(VOS_HANDLE hLCD)
{
    vos_delay_msecs(100);
    // Send Reset command
    write_lcd_cmd(hLCD, 0x03);
    vos_delay_msecs(2);
    // Send Function Set
    write_lcd_cmd(hLCD, 0x28);
    vos_delay_msecs(2);
    write_lcd_cmd(hLCD, 0x28);
    vos_delay_msecs(2);
    // Send Display control command
    write_lcd_cmd(hLCD, 0x0C);
    vos_delay_msecs(2);
    // Send Display Clear command
    write_lcd_cmd(hLCD, 0x01);
    vos_delay_msecs(2);
    // Send Entry Mode Set command
    write_lcd_cmd(hLCD, 0x06);
    vos_delay_msecs(2);
}

// Clear LCD and reset cursor
void lcd_clear(VOS_HANDLE hLcd)
{
   // Send Display Clear command
    write_lcd_cmd(hLcd, 0x01);
    vos_delay_msecs(2);
}

// Write to the top line of our display
void write_lcd_line1(VOS_HANDLE hLcd, char* str)
{       // Set 1-st line address
    write_lcd_cmd(hLcd, (0x01 | 0x80));
    // Send string to LCD
    write_lcd_str(hLcd, str);
}

// Write to the bottom line of our display
void write_lcd_line2(VOS_HANDLE hLcd, char* str)
{
    // Set 2-nd line address
    write_lcd_cmd(hLcd, (0x40 | 0x80));
    // Send string to LCD
    write_lcd_str(hLcd, str);
}
```

experimentation with the current limiting resistor to obtain the desired brightness. Also, if you go too large with this resistor you might fail to turn on the LED. Because I only have two LEDs and only one is lit at a time, I used a single current limiting resistor connected to the negative side of both LEDs. The code for turning the LEDs on and off is presented in Listing 5.

USING LCD SCREENS

Small LCD screens with 16 characters by 2 lines are a standard output device for microcontrollers. As a result, there are many libraries available for these devices. I shamelessly modified Arduino LCD library code to work with the Vincullum II. These displays have 16 pins, not all of which are typically used. The standard pinout for these displays and usage in the impersonator are shown in Table 1. Note that the 10K contrast potentiometer is connected to +5V and ground with the center pin connected to Vo on the LCD.

Table 1. *LCD screen pinouts*

Pin	Signal	Usage	Comments
1	Ground	Ground	
2	Vcc	+5V	
3	Vo	Contrast	Connected to 10K pot center tap
4	Rs	B4	Read select
5	Rw	Ground	Active low write enable
6	E	B5	Enable display
7	DB0	Not used	DB0-3 only used 4 sending 8-bit values
8	DB1	Not used	
9	DB2	Not used	
10	DB3	Not used	
11	DB4	B0	DB4-7 are used for sending 4-bit values
12	DB5	B1	
13	DB6	B2	
14	DB7	B3	
15	BL+	Not used	Positive for backlight if present
16	BL-	Not used	Negative for backlight if present

Because GPIO pins are often at a premium only 4 data pins are used to send messages to the screen. The screen responds to a number of standard commands that include things such as clearing the screen and moving the cursor. The methods used to interface with the LCD are shown in Listing 6. Note that the LCD and LEDs are on the same port. As a result, the port value is first read to avoid messing up the LEDs when printing to the LCD.

ON THE WEB

- *http://ftdichip.com* – The FTDI webpage where IDE and other software may be downloaded
- *http://www.usb.org/developers/docs/* – The official source for all things USB
- *http://lvr.com/usbc.htm* – USB expert Jan Axelson's main USB page
- *http://lvr.com/mass_storage.htm* – Jan Axelson's USB mass storage page
- *http://www.youtube.com/watch?v=3D9uGCvtoFo* – Phil's DEFCON XX Talk on USB impersonation
- *http://www.youtube.com/watch?v=CIVGzG0W-DM* – Phil's 44CON 2011 Talk on USB flash drive forensics
- *http://www.instructables.com/id/Cheap-and-Effective-USB-Write-Blocker/* – Phil's instructable on how to build a USB write blocker based on his BlackHat EU 2012 presentation
- *https://www.blackhat.com/html/bh-eu-12/bh-eu-12-archives.html#polstra* – Phil's BlackHat EU 2012 presentation on USB write blocking
- *http://www.concise-courses.com/infosec/20130404/* – Phil's Hacker Hotshots presentation on building a USB impersonator
- *http://ppolstra.blogspot.com* – Phil's blog
- *http://twitter.com/ppolstra* – Phil's Twitter page (@ppolstra)
- *https://github.com/ppolstra/usb-impersonator* – Github for impersonator code

SUMMARY

In this article we have seen how a powerful USB impersonator can be created by extending the USB write blocker from a previous article. Along the way we also received a brief introduction to performing GPIO with microcontrollers. Code and schematics for the devices presented are available online at *http://polstra.org*.

In the next article we exam some open source tools which can prove useful when performing USB forensics and/or debugging USB devices. Should you have any question on this article, feel free to contact me on Twitter @ppolstra or at *http://polstra.org*.

ABOUT THE AUTHOR

Dr. Phil Polstra is currently an Associate Professor and Hacker in Residence at a private university in the Midwestern United States. His primary research focus over the last few years has been on the use of small, low-powered devices for forensics and penetration testing. As part of this work, he has developed his own custom pentesting Linux distribution that runs on the BeagleBoard family of ARM-based computers. Phil has presented his work on USB forensics and pentesting with small computers at several conferences around the world including BlackHat, DEFCON, 44CON, GrrCON, ForenSecure, and B-sides, to name a few. When not working he can be found spending time with family, hacking electronics, picking locks, flying, or building airplanes.

HOW TO PERFORM FORENSICS ON USB MASS STORAGE DEVICES

PART 6: LEVERAGING OPEN SOURCE

by Dr. Phil Polstra

USB mass storage devices have become the standard for backup and transfer of files. The popularization of this media has led to challenges for forensic specialists trying to find data on fixed memory storage media instead of traditional magnetic media. In this sixth article of a multi-part series we will examine how to leverage open source software in order to perform forensics on USB devices.

What you will learn:

- How to identify USB devices
- How to intercept USB traffic
- How to use udev rules to mount USB devices as read only

What you should know:

- A basic understanding of Linux
- A basic understanding of USB mass storage devices (possibly from earlier articles in this series)

Previous articles in this series have focused on how to construct devices and perform forensics investigation of USB mass storage devices. This final installment is intended to show investigators how to leverage open source software in order to investigate USB devices. Everything described here could also be done with commercial hardware and software, but budgets do not always allow for commercial solutions. All forensic investigators will find these tools to be powerful and extremely useful, regardless of their organization's level of financial resources.

IDENTIFYING DEVICES

When encountering an unknown USB device, the first step is to identify the device. We will Linux and standard tools to identify the device. Linux is used because it is operating systems by programmers and for programmers. As such, the selection of open source tools available on the Linux platform is unmatched anywhere else.

For this tutorial I will use a small flash drive and "unknown" USB device. The first step in identification is to use the lsusb utility (which should be installed by default in most Linux distributions) to list all properly enumerated devices. The results of running this command on my system are shown in Figure 1.

Notice that a series of USB hub devices are listed in the lsusb results. There are a couple of important things to note about this. There are two USB 2.0 root hubs. One of these, on Bus 001, is used for built in USB devices such as the multi-card reader. The other high-speed hub, on Bus 002, is used to communicate with devices capable of high-speed communication that have been plugged into USB ports (possibly downstream from high-speed hubs). From the results we see the flash drive is a SanDisk Cruzer Blade, which is capable of high-speed. The mystery device reports itself as a Prolific PL2303 serial port.

The PL2303 is attached to Bus 006. This tells us that it is not capable of high-speed. When a slow device is attached it is given its own bus so that it doesn't bog down the high-speed buses. It can be helpful to connect devices through a USB 1.1 hub when debugging and/or investigating them, as this will facilitate sniffing by creating a bus with a single device. Care should be taken, however, as the behavior of some devices changes with different speeds.

Our two devices have enumerated successfully. When this isn't the case or when more information is desired, the device message logs may prove helpful. Linux systems all ship with the dmesg utility for listing these messages. The results from the dmesg command are shown in Figure 2.

The dmesg results provide more information about the SanDisk drive. We see that it is 4 gigabytes, use 512 byte blocks, has no write protection, and uses a SCSI command set. Additionally, we see that it has been labeled as `/dev/sdc`. The PL2303 has been attached to Bus 006 and also attached to ttyUSB0.

SNIFFING USB TRAFFIC

Using two standard tools we have made progress in the investigation of our two devices. In order to go any further we must start intercepting USB traffic. Fortunately, this is quite easy to do on Linux systems. Loading the usbmon module by executing the command "sudo modprobe usbmon" easily creates a USB monitoring device. Once this command has been executed new capture devices will appear in Wireshark and similar packet capture tools. These new interfaces are shown in Figure 3.

Here is a breakdown of the command above for those unfamiliar with Linux modules. The modprobe command is used to load a new module into the Linux kernel. For obvious reasons, this command must be run as root. A list of currently loaded modules can be obtained by executing the "lsmod" command. The usbmon module creates a monitor interface for each USB bus. Those familiar with the aircrack-ng suite will recognize this behavior as being very similar to that of the airmon-ng utility used to create a monitor interface for a wireless adapter.

Let's have a look at the PL2303 traffic, which we know will appear on Bus 006. As shown in Figure 4,

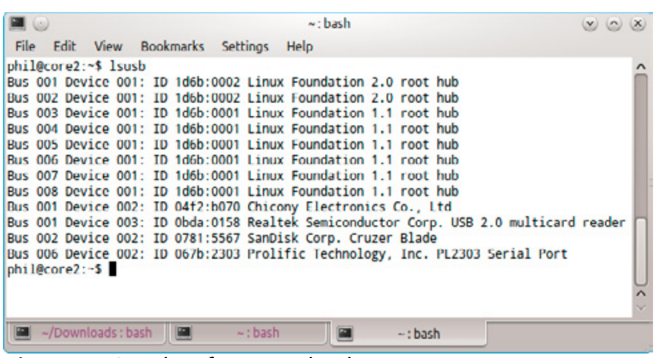


Figure 1. *Results of running lsusb*

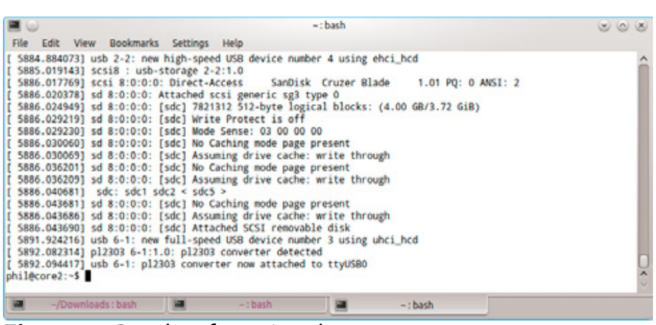


Figure 2. *Results of running dmesg*

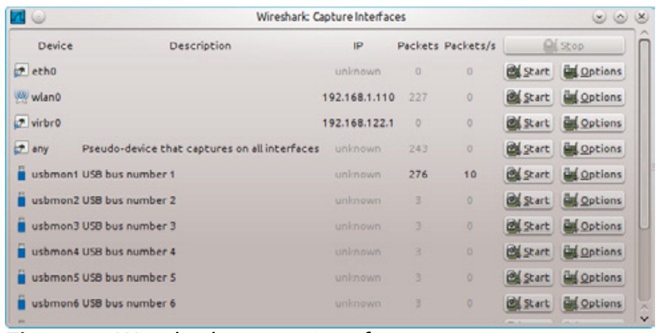


Figure 3. *Wireshark capture interfaces*

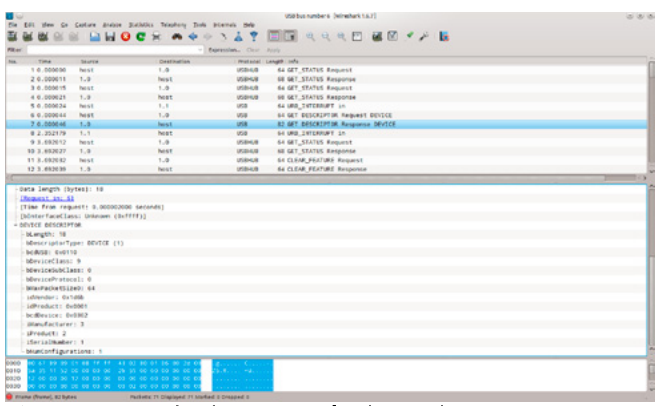


Figure 4. *Wireshark capture of a device descriptor*

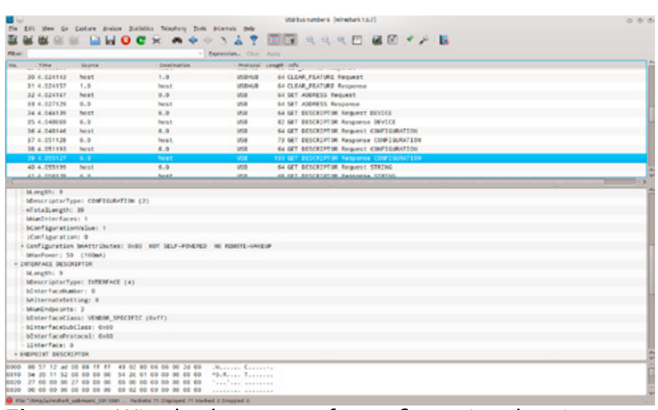


Figure 5. *Wireshark capture of a configuration descriptor*

Wireshark understands and interprets USB traffic as it does with network traffic.

From the configuration descriptor, which is partially shown in Figure 5, we can see the USB device type and other useful information. The device is not self-powered, does not respond to remote wakeup requests, requires no more than 100mA of current, and has one interface, one out endpoint, and two in endpoints. While not shown here, the PL2303 is also queried for description and manufacturer strings.

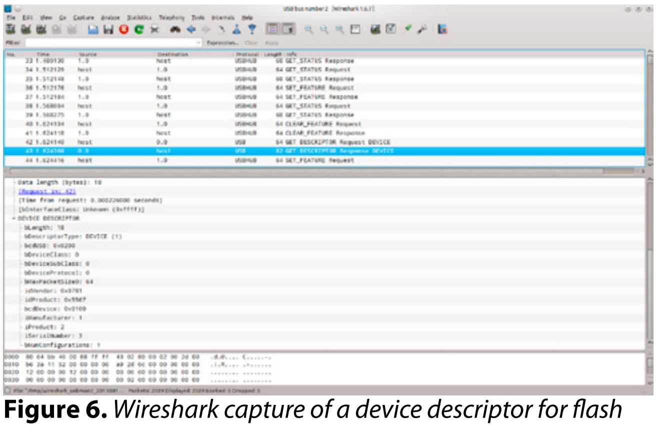


Figure 6. *Wireshark capture of a device descriptor for flash drive*

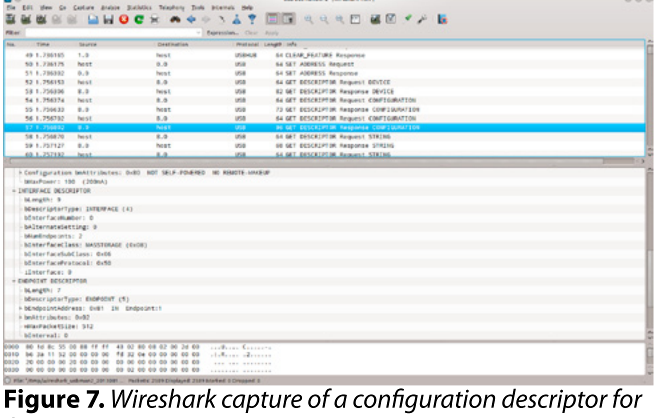


Figure 7. *Wireshark capture of a configuration descriptor for flash drive*

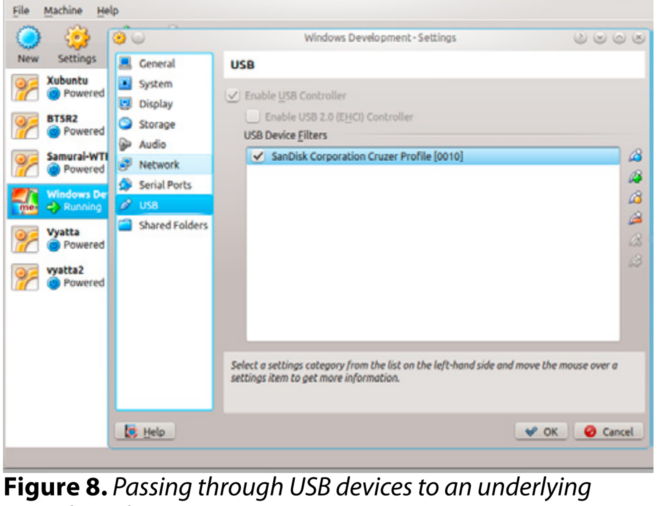


Figure 8. *Passing through USB devices to an underlying virtual machine*

With very little effort, we have captured the enumeration traffic for the PL2303. If desired we could also capture traffic to this device. Let us now turn our attention to the SanDisk drive instead. As previously mentioned, because the device is high-speed capable, it is connected to high-speed Bus 002. There is considerable chatter on this bus. Similar to floods of beacon frames in wireless sniffing, USB sniffing is done in a sea of status requests and test unit ready requests. Many operating systems will continuously ping mass storage devices with test unit ready requests to keep them from sleeping. The device descriptor is shown in Figure 6. Notice the device descriptor indicates type 0, which means the type, will be revealed later in the configuration descriptor. A partial capture of the configuration descriptor is shown in Figure 7. We see from the descriptor that the SanDisk can require up to 200 mA of current. This much current could power an entire small computer board, such as the BeagleBone Black. The descriptor also tells us that it is a mass storage device that uses the SCSI protocol and that each of the bulk endpoints has a maximum packet size of 512 bytes (one block). As with the PL2303, we can use Wireshark to sniff the traffic to and from the device if desired.

DEALING WITH WINDOWS-ONLY DEVICES

So far we have learned how to handle devices, which operate on all the standard operating systems. One might ask how to deal with devices that only work with Windows. The solution is actually quite simple. Setup your Linux system to capture USB traffic as previously described and then run Windows virtually using VirtualBox or similar virtualization software.

For example, the SanDisk Cruzer Profile drive is a Windows-only mass storage device that only permits access to the stored data after a successful fingerprint scan is performed with the integrated fingerprint scanner. To access such a device in our virtual machine we must tell VirtualBox to pass all the traffic through and to have the host operating system ignore it. We can either do this each time we use the device from the Devices menu in the window, which contains the Windows virtual machine, or permanently make this the case using the settings for our Windows Virtual box.

A screenshot for this more permanent method is shown for the Profile drive in Figure 8. The first button on the right with the blue circle allows a new empty filter to be added which can then be edited by clicking the third button with the orange circle. Clicking on the second button with the green plus sign allows a filter for a currently connected device to be added. I will leave it as an exercise to the reader to reverse engineer the Profile drive and/or develop a Linux driver.

FUN WITH UDEV RULES

In an earlier article, we discussed using microcontrollers to block writes to USB mass storage devices. When running Linux we can accomplish the same thing using udev rules. Udev rules affect the way devices are labeled, what drivers are loaded, etc. We can use udev rules to force any mass storage devices connected downstream from a hub with a particular VID/PID combination to be mounted read-only. One of the big advantages this has over our microcontroller solution is that we can operate at high speed. While full speed is fine for flash drives, it would take much too long to create an image of a large hard drive with our microcontroller-based write blocker. What is presented here is not intended as a complete tutorial on udev rules, but rather as a concrete example of their power. This information is taken from a forensics module for The Deck. The Deck is a custom penetration testing and forensics Linux distribution that runs on the BeagleBoard-xM, BeagleBone, and BeagleBone Black ARM-based devices. Because this module was implemented entirely using udev rules it can be used on any Linux distribution. More information on The Deck can be found at my website *http://polstra.org*. This module,

Listing 1. *Sample udev rules*

```
ACTION==”add”, SUBSYSTEM==”block”, KERNEL==”sd?[1-9]”, ATTRS{idVendor}==”1a40”,
ATTRS{idProduct}==”0101”, ENV{PHIL_MOUNT}=”1”, ENV{PHIL_DEV}=”%k”, RUN+=”/etc/udev/scripts/
protmount.sh %k”
ACTION==”remove”, SUBSYSTEM==”block”, KERNEL==”sd?[1-9]”, ATTRS{idVendor}==”1a40”,
ATTRS{idProduct}==”0101”, ENV{PHIL_UNMOUNT}=”1”, RUN+=”/etc/udev/scripts/protmount3.sh %k”

ENV{PHIL_MOUNT}==”1”, ENV{UDISKS_PRESENTATION_HIDE}=”1”, ENV{UDISKS_AUTOMOUNT_HINT}=”never”,
RUN+=”/etc/udev/scripts/protmount2.sh”
ENV{PHIL_MOUNT}!=”1”, ENV{UDISKS_PRESENTATION_HIDE}=”0”, ENV{UDISKS_AUTOMOUNT_HINT}=”always”

ENV{PHIL_UNMOUNT}==”1”, RUN+=”/etc/udev/scripts/protmount4.sh”
```

known as the 4Deck, can also be downloaded there. All Linux systems ship with a standard set of udev rules. While these could be modified directly, doing so is not recommended. The suggested practice is to create new rules in the `/etc/udev/rules.d` directory. Similar to the workings of start-up scripts, filenames are used to determine the order in which udev rules are processed. As a result, rules normally start with a priority number followed by a descriptive name and must end with a ".rules" extension. The appropriate rules for a hub with a VID/PID of 0x1a40/0x0101 are shown in Listing 1.

The first line is matched whenever a block device is added downstream from a device with the appropriate VID/PID (our special hub). When this happens two environment variables PHIL_MOUNT and PHIL_DEV are set. The kernel name for the new device (i.e. "sdb1") is substituted for %k when the rule is run. In addition to setting the environment variables we add protmount.sh with a parameter of our device's kernel name to a list of scripts to be ran. The second line is similar and is matched upon device removal.

The next two lines are used to prevent the message box from popping up when a read-only device is connected through our hub as indicated by PHIL_MOUNT having a value of 1 and to cause the message box to pop up otherwise (PHIL_MOUNT not equal to 1). The last line adds a script to the run list when the read-only device is mounted.

The protmount.sh and protmount3.sh scripts simply create protmount2.sh and protmount4.sh, respectively. These files are stored in the `/etc/udev/scripts` directory. The reason that these scripts create other scripts is that the device has not been fully loaded at the point when our rules are matched. The protmount.sh and protmount3.sh scripts are presented in Listing 2 and Listing 3, respectively. The protmount.sh mounts the mass storage device in the standard place, but read-only. The protmount3.sh script performs cleanup tasks such as unmounting the drive and removing the mount point directory.

Listing 2. *protmount.sh script*

```
#!/bin/bash
echo "#!/bin/bash" > /etc/udev/scripts/
protmount2.sh
echo "mkdir /media/$1" >> /etc/udev/scripts/
protmount2.sh
echo "chmod 777 /media/$1" >> /etc/udev/
scripts/protmount2.sh
echo "/bin/mount /dev/$1 -o ro,noatime /
media/$1" >> /etc/udev/scripts/protmount2.sh
chmod +x /etc/udev/scripts/protmount2.sh
```

Listing 3. *protmount3.sh script*

```
#!/bin/bash
echo "#!/bin/bash" > /etc/udev/scripts/
protmount4.sh
echo "/bin/umount /dev/$1" >> /etc/udev/
scripts/protmount4.sh
echo "rmdir /media/$1" >> /etc/udev/scripts/
protmount4.sh
chmod +x /etc/udev/scripts/protmount4.sh
```

SUMMARY

We have covered quite a bit of ground in this series. First, we took an in depth look at the inner workings of USB. Next, we described the operation of USB mass storage in detail. Once the groundwork had been laid, we dove into how to leverage microcontrollers to create forensic duplicates of USB mass storage devices sans computer. Next, we saw how to use these same microcontrollers to block write requests to mass storage devices plugged into a computer. By extending and slightly modifying our write blocker, we were able to create a device capable of USB device impersonation which allowed us to bypass endpoint security software. Finally, we saw how to leverage open source tools to make USB forensics easier and considerably cheaper.

I hope that the reader has learned something useful in our journey. While this article series has now drawn to a close, the fun with USB forensics continues. Should you have any question on this article, feel free to contact me on Twitter @ppolstra or at *http://polstra.org*.

ON THE WEB

- *http://www.reactivated.net/writing_udev_rules.html* – udev tutorial
- *http://ppolstra.blogspot.com* – Phil's blog
- *http://twitter.com/ppolstra* – Phil's Twitter page (@ppolstra)

ABOUT THE AUTHOR

Dr. Phil Polstra is currently an Associate Professor and Hacker in Residence at a private university in the Midwestern United States. His primary research focus over the last few years has been on the use of small, low-powered devices for forensics and penetration testing. As part of this work, he has developed his own custom pentesting Linux distribution that runs on the BeagleBoard family of ARM-based computers. Phil has presented his work on USB forensics and pentesting with small computers at several conferences around the world including BlackHat, DEFCON, 44CON, GrrCON, ForenSecure, and B-sides, to name a few. When not working he can be found spending time with family, hacking electronics, picking locks, flying, or building airplanes.