Beyond Autorun: Exploiting vulnerabilities with removable storage

Jon Larimer jlarimer@us.ibm.com, jlarimer@gmail.com

IBM X-Force Advanced Research

BlackHat – Washington, DC - 2011

January 18, 2011

Contents

1. Abstract 5
2. Introduction 6
  2.1. A brief history of removable storage malware 6
  2.2. AutoRun and AutoPlay 6
  2.3. Stuxnet and the LNK vulnerability 7
  2.4. Attacks on physical systems 7
3. USB Architecture 9
  3.1. About USB 9
  3.2. Host controllers 10
  3.3. Devices 10
    3.3.1. Hubs 10
    3.3.2. Functions 10
    3.3.3. Interfaces 10
    3.3.4. Endpoints 11
    3.3.5. Device classes 11
    3.3.6. USB descriptors 12
  3.4. Mass storage class devices 13
  3.5. Attacks using the USB protocols 14
  3.6. Fuzzing USB drivers 14
    3.6.1. Windows Device Simulation Framework 15
    3.6.2. QEMU/BOCHS 15
4. USB operation on Windows 7 16
  4.1. USB driver stack 16
    4.1.1. Core stack 16
    4.1.2. Class drivers 17
    4.1.3. USB device recognition 18
    4.1.4. The danger of drivers from Windows Update 20
  4.2. Mass storage devices 21
    4.2.1. USB storage port driver and Windows disk class driver 21

4.2.2. Partition and volume management 22
4.2.3. File system drivers 22
4.2.4. Fuzzing filesystem drivers on Windows 23
4.3. Exploiting USB and file system drivers 24
4.4. PnP Manager 24
4.4.1. Kernel mode PnP manager 24
4.4.2. User mode PnP manager 25
4.5. AutoPlay 25
4.5.1. Shell Hardware Detection Service 25
4.5.2. ReadyBoost 27
5. Windows Explorer 28
5.1. Shell Extension Handlers 28
5.1.1. Registered file types and perceived types 29
5.1.2. Icon handlers 30
5.1.3. Thumbnail handlers 32
5.1.4. Image handlers 34
5.1.5. Preview handlers 35
5.1.6. Infotip handlers 36
5.1.7. COM object persistence and type confusion 36
5.1.8. Fuzzing shell extensions 36
5.1.9. Exploiting shell extensions 36
5.2. Property system 37
5.3. Folder customization 38
5.3.1. Shell namespace extensions 39
6. USB operation on GNU/Linux 40
6.1. Core 40
6.2. USB interface drivers 40
6.3. USB mass storage class driver 40
6.4. udev, udisks, D-Bus 41
6.5. File systems in Linux 41
7. GNOME and Nautilus 43
7.1. Automatic mounting of storage devices 43

7.2. Autorun capabilities 44
7.3. Thumbnailers 45
7.3.1. Exploiting thumbnailers 45
8. Conclusion 48
8.1. Acknowledgements 48
9. Appendix 49
9.1. USB descriptors for a mass storage class device 49
9.2. Default Shell Extension Handlers in Windows 7 Professional (32 bit) 50
9.2.1. Icon handlers 50
9.2.2. Image handlers 50
9.2.3. Thumbnail handlers 51
9.2.4. Property handlers 52
9.2.5. Preview handlers 54
9.3. Default GNOME Desktop thumbnailers in Ubuntu Desktop Linux 10.10 (32 bit) 57
10. Works cited 60
11. Legal notices 66

1. Abstract

Malware has been using the **AutoRun** functionality in Microsoft Windows for years to spread through removable storage devices. Although the feature is easy to disable, the Stuxnet worm was able to spread through USB drives by exploiting a vulnerability in Windows. This paper examines different ways that attackers could potentially abuse operating system functionality to execute malicious payloads from USB devices without relying on **AutoRun**. There's a lot of code that runs between the USB drivers and the desktop software that renders icons and thumbnails for files, providing security researchers and hackers with a rich landscape of potentially vulnerable software to exploit. Understanding what this code does is crucial for discovering and fixing vulnerabilities that could be exploited from removable storage devices.

2. Introduction

2.1. A brief history of removable storage malware

The very first computer virus that spread in the wild through removable storage was **Elk Cloner**, which spread on floppy disks used by Apple II computers (1). **Elk Cloner** was a boot sector virus, meaning it would infect the computer's memory when it was booted from an infected disk. The first virus for MS-DOS that spread through removable storage was **Brain**, which was developed in January of 1986 (2). It was a boot sector virus that infected 5 ¼" 360KB floppy disks. Whenever an MS-DOS PC was booted with an infected floppy disk, any other floppy disks accessed would also be infected with **Brain**. Later in 1986, Ralf Burger developed the **Videm** virus which was the first file infecting virus for DOS (3). The first virus to infect PE files – the executable format used by Windows95, was called **Bizach**, and was developed in 1996 (4). In 2002, the **Roron** worm made use of the `autorun.inf` file to spread to remote network drives, although it would not infect USB devices (5). The **Bacros** worm, discovered in 2004, would spread to CD-Rs by creating an `autorun.inf` file on CD-ROM drives, but it also specifically avoided copying itself to USB devices (6). 2004 was also the year that Microsoft released service pack 2 for Windows XP. While this service pack was designed to enhance security by including a firewall and enabling automatic updates by default, it also enabled **AutoRun** for floppy disks and some USB mass storage devices (7), eventually leading to a flood of USB malware. In 2005, Darrin Barrall and David Dewey presented a talk at BlackHat USA 2005 about using USB flash drives to install malware (8). In early 2007 the first worm to spread through USB sticks using `autorun.inf`, **SillyFD-AA**, was reported by Sophos (9). It didn't take long for other malware authors to include this functionality, and in 2008 the U.S. Strategic Command banned all removable storage devices, including floppy disks and USB drives (10). Finally, in 2010, the **Stuxnet** worm was discovered using a vulnerability in the Windows LNK file shell icon handler to infect PCs from USB devices, even with **AutoRun** disabled.

2.2. AutoRun and AutoPlay

The **AutoRun** feature was introduced in the Windows95 operating system. It was originally designed to allow software developers to have an application or installer to execute whenever a user inserted a CD-ROM into their PC (11). To use this feature, a file named `autorun.inf` is placed in the root directory of the CD-ROM disc. This file is a text file which contains instructions on which program to launch when a disc is inserted. Here's an example of a simple `autorun.inf` file that specifies `program.exe` should be executed when the disc is inserted:

```
[autorun]
open=program.exe
icon=helper.dll,1
label=Awesome Program
```

AutoPlay was introduced in Windows XP to launch applications automatically when a supported device or file system is connected to the computer (12). This lets Windows launch a movie player to play VideoCD and DVD disks and an audio application for CDs, for example. The **AutoRun** feature is now considered a subset of **AutoPlay**.

In Windows 7, Microsoft changed the behavior of **AutoPlay** so that **AutoRun** only works on removable optical media, CDs and DVDs, but not USB drives (13).

2.3. Stuxnet and the LNK vulnerability

The LNK vulnerability used by the **Stuxnet** worm is interesting for a number of reasons. Until it was fixed, the vulnerability was present and exploitable on all supported versions of Windows. The vulnerability allowed the execution of an arbitrary DLL file on a removable storage device without relying on the **AutoRun** feature. The Stuxnet worm was able to spread for months using this vulnerability without being detected.

The details of the flaw exploited by **Stuxnet** aren't important for the topic of this paper, and have been documented in Symantec's whitepaper *W32.Stuxnet Dossier* (14) and by Peter Ferrie in *The Missing LNK* (15). What is important to understand is the reason this vulnerability exists – because Windows (and other operating systems) will render custom icons for certain files when displaying them in a folder on the desktop. The custom icon code will sometimes parse file content in order to determine what icon to display and a malicious file can exploit a vulnerability in that icon handling code. This class of vulnerability provided **Stuxnet** with a way to spread through USB drives without relying on **AutoRun**.

The LNK vulnerability won't be the last vulnerability found in custom icon code, and Stuxnet won't be the last malware to take advantage of such a vulnerability. This type of vulnerability is especially suited to worm-like malware that can spread with very little user interaction. The purpose of this paper is to bring more attention to this and other related avenues of attack with USB mass storage devices and removable storage in general.

2.4. Attacks on physical systems

The attacks on software described in this paper depend on physical access to the target machine. Many people consider physical access "game over" – that if an attacker has physical access, they can do anything they want, and this is true. If someone has physical access, they could boot the machine from external media or even just steal the machine itself. However, if the machine is protected with BIOS passwords, a hard disk password, or full disk encryption, then the best way to attack it is under the context of a logged-on user or at the OS kernel level – after the OS has been booted and the file system mounted. One possible method for attacking a system in this state is by using so-called cold boot attacks on the RAM (16), but that requires being able to boot from a device under your control, which could be disabled in the BIOS. Another very effective attack on physical machines is to use DMA (direct memory access) through an IEEE 1349 (FireWire) port to read or write directly to or from physical memory on a target machine. A practical DMA attack using FireWire was first presented at PacSec04 in Tokyo by Maximillian Dornseif in his talk "0wned by an iPod" (17). At Ruxcon in 2006, Adam Boileau demonstrated a tool called WinLockPwn that was able to bypass the Windows XP screensaver lock over a FireWire connection (18). Of course, these attacks are only applicable if the target PC has a 1394 port.

USB is still an excellent choice for vector of physical attack, and the entire attack surface available through USB (and other avenues for removable storage, such as eSATA) has not been fully explored. The beauty of attacks with USB devices, and other forms of removable storage, is that they can also be used

to attack a machine where you don't have direct physical presence if you can convince someone to connect the device to their PC. People may suspect they're safe from malware because they have **AutoRun** disabled and keep their AntiVirus software updated, or because they run Linux or MacOS, but the point of this paper is to show that that isn't necessarily true – there can be software vulnerabilities lurking anywhere from the low level kernel mode USB drivers up to the high-level graphical interface of the OS. If a hacker can execute arbitrary code by inserting a malicious USB device, the exploit payload could do any number of evil things – dump physical memory back to the USB device, copy the victim's home directory to the USB device, install malware on the PC, or even unlock the screensaver to get full access to the PC as the logged-on user. The damage that can be done is limited only by the privilege level of the executing code or the availability of privilege escalation exploits.

3. USB Architecture

To understand how USB devices can be used to attack a PC, it's important to know what USB is and how it works. This section is intended to be a brief introduction to USB and how it's used by removable storage devices.

3.1. About USB

USB, or Universal Serial Bus, is a standard that allows peripheral devices to talk to computers. It's used by keyboards, mice, digital cameras, printers, removable storage devices, and many other peripherals. USB is an asymmetric, speak-when-spoken-to, tiered-star network topology system. Peripherals, referred to as functions in the USB specifications, are connected to hubs which are in turn connected to a host controller.

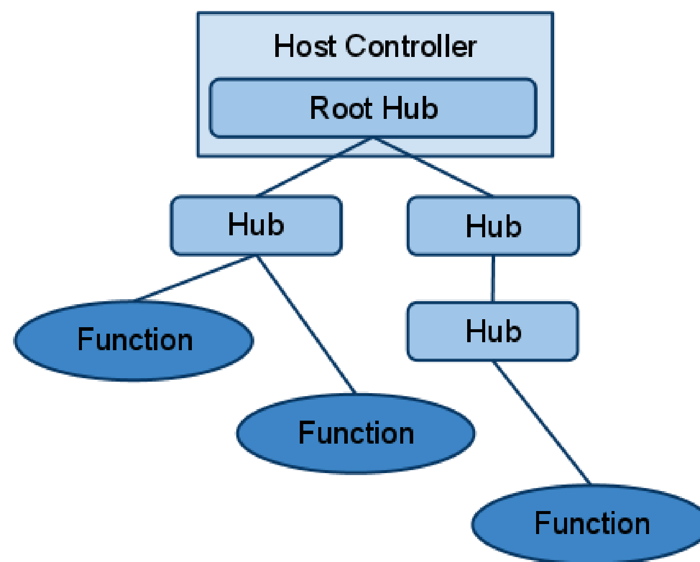

Figure 1 - USB device relationships

The operating system typically provides a device driver for the host controller and generic USB device classes, such as the Human Interface Device (HID) and the Mass Storage Device (MSD) classes, and the manufacturer of a USB device will provide more specific device drivers.

USB is a polled bus, and all transactions are initiated by the host. The host controller sets the schedule for transactions to and from devices. This means that a USB device doesn't send commands to the host controller – it only responds to commands sent to it. This also means that USB devices cannot communicate with each other. This is different than the IEEE1394 bus, which uses a peer-to-peer protocol and devices are able to send commands to the host controller as well as communicate with other devices on the bus.

The authoritative source of information on the USB architecture and protocols are the USB specifications available from the USB Implementers Forum (19). The information on the Wikipedia page on USB (20) is also helpful for a broader understanding of the history, purpose, and design of USB. The

USB In a Nutshell website (21) is another great guide to the standard. This section is an overview of USB, but the published technical specifications should be consulted for more detail.

3.2. Host controllers

There are three types of host controllers currently used by USB: **UHCI** (Universal Host Controller Interface), **OHCI** (Open Host Controller Interface), and **EHCI** (Enhanced Host Controller Interface). **UHCI** is used for low-speed USB 1.1 devices such as keyboards and mice. **OHCI** is an alternative to **UHCI** for USB 1.1 devices, and can be found on PCs that are not based on Intel's chipsets. **EHCI** is used for high-speed USB 2.0 devices, like printers, scanners, and flash drives. Of these three types of host controllers, only **EHCI** and **OHCI** are "open" specifications (22) (23). Although the specifications for **UHCI** are proprietary, a design guide is available from Intel (24).

3.3. Devices

A USB device is either a **hub** or a **function**. In USB terms, "device" refers to a physical device, and a physical device can have more than one function.

3.3.1. Hubs

A USB hub is basically a "wiring concentrator" - it allows one to plug multiple devices into a single port. A hub's upstream connection is either another hub or the host controller itself. The USB specifications permit 6 tiers of hubs, including the root hub. This means that 5 non-root hubs can be present between the host controller and a function. The root hub is inside the PC and part of the host controller, and the USB ports on the outside of a PC are generally connected to the root hub. Consumers can buy additional USB hubs that contain ports for connecting multiple devices. Some consumer USB devices also include built-in hubs. For example, keyboards and monitors can contain built-in USB ports for connecting additional devices.

3.3.2. Functions

A function plugs into a hub. These are the peripheral devices themselves that communicate over the USB bus. Many USB devices serve a single purpose, but there are devices that provide multiple functions over a single USB connection, such as mouse/keyboard combinations. A device providing multiple functions is either a composite or a compound device. A **composite device** has multiple functions at the same bus address. An example of this would be an external USB keyboard with a built-in pointing stick. This keyboard contains the functionality of a keyboard and a mouse, but connects to a single physical USB port. This port will contain two **interfaces** – one for the keyboard and one for the mouse, which use different protocols for communicating with the system. A **compound device** has multiple functions at different bus addresses that are connected to a hub inside of the device. That means that each functional aspect of the device has its own port. Because the device has a hub built-in, it only needs to connect to a single port on the PC.

3.3.3. Interfaces

A single physical USB connection supports one or more interfaces, and an interface can be thought of as a logical grouping of **endpoints**. Each interface can support a different device class, which means it exposes a different set of functionality. The example above was a combination mouse and keyboard

device. Each of those device functions has a separate set of endpoints on the bus so that the host controller can send requests to each separate function.

3.3.4. Endpoints

Each logical USB device has one endpoint labeled 'endpoint zero' that is used for control transactions, for example to retrieve the device, interface, endpoint, and configuration descriptors. The interface and endpoint descriptors describe how many endpoints each interface has and how each endpoint works – which direction the data flows (in/out), what protocol is used, etc. A connection between a single endpoint and the host controller is called a **pipe**. Each endpoint on a device gets a separate address. For an example of how endpoints are used, a USB mass storage device will generally have three endpoints and a single interface – the control endpoint (endpoint zero), and separate endpoints for input and output of data. The following diagram shows the relationship between interfaces and endpoints in a composite device – one with multiple interfaces:

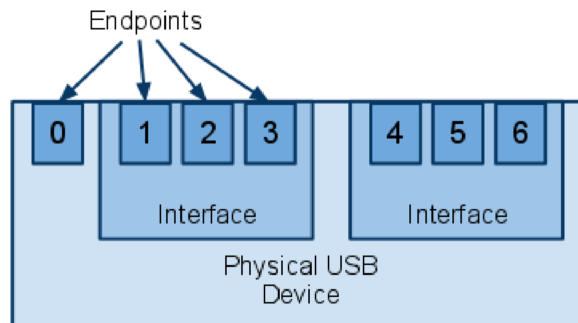

Figure 2 - Interfaces and Endpoints in a composite USB device

3.3.5. Device classes

Each device is identified by a class code (sometimes called a class ID) that the host can use to determine which set of drivers to load for it. Examples of class IDs are *Mass Storage Device Class* and *Human Interface Device Class (HID)*. Mass Storage Devices, the focus of this research, are designed to provide access to storage devices such as floppy disks, CD-ROMs, and flash storage devices. HIDs provide an interface for keyboards, mice, joysticks, and other input devices. A list of all device class codes can be found on the USB Implementers Forum website (25).

3.3.6. USB descriptors

Each USB device includes a number of descriptors that the host queries to determine a device's capabilities. These descriptors tell the host what protocol to use, what class the device is, which interfaces are supported, etc. The host operating system's USB driver stack uses these descriptors to determine which driver to load for each device. For a real-world example of what these descriptor values look like, see *USB descriptors for a mass storage class device* in the Appendix. There are five commonly used types of descriptors.

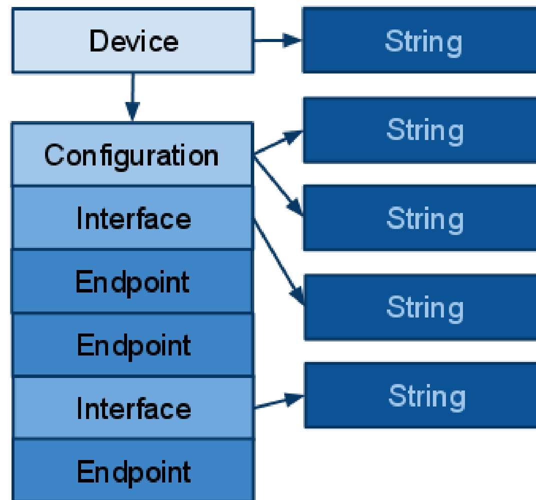

Figure 3 - USB descriptor relationships

3.3.6.1. Device descriptor

Each USB device has a single device descriptor that contains critical information about the device – the class, subclass, protocol, vendor, and product identification information required by the OS to load the required drivers. In many cases, the class, subclass, and protocol are set to **00** to indicate that the class information should come from the interface descriptor instead. It also contains a serial number and specifies the number of configuration descriptors on the device.

3.3.6.2. Configuration descriptor

There can be multiple configuration descriptors for each device. This descriptor contains information on the power requirements of the device (self-powered or bus powered), as well as the number of interfaces on the device. The interface and endpoint descriptors, described below, are actually part of the configuration descriptor – the configuration descriptor has a 'total length' field that specifies how much data should be read to obtain all of the data for interfaces and endpoints.

3.3.6.3. Interface descriptor

There is an interface descriptor for each function of the device. A composite device will include more than one interface. Each interface specifies a single class, subclass, and protocol. Each interface can support a number of endpoints.

3.3.6.4. Endpoint descriptor

There is a separate endpoint descriptor for each endpoint other than "endpoint 0", which is the control endpoint. The endpoint descriptor specifies communication protocol information, such as the transfer and synchronization type.

3.3.6.1. String descriptor

String descriptors are special descriptors that hold string data. Strings are referenced by language and are stored in the Unicode format. Examples of information stored in string descriptors can be found in the **iManufacturer**, **iProduct**, and **iSerialNumber** fields of the device descriptor. On a test device used for this research, those values were respectively "SanDisk", "Cruzer", and "20060266120EDEE311F8". These values can be used by the operating system to provide a description of the device to the end user.

3.4. Mass storage class devices

The mass storage class defines the interface used by mass storage devices, such as flash drives, external hard disk drives, floppy drives, and CD-ROM drives. Most USB mass storage devices use the bulk-only transport (**BOT**) protocol to send and receive responses to the host using the bulk transfer capability of USB (26), as defined by the USB Implementers Forum (27). This standard specifies which descriptors USB mass storage devices need to support, and which values certain descriptor fields must contain. For example, the standard specifies that the **bDeviceClass** in the Device Descriptor should be set to zero, and the mass storage class and subclass ID should be identified in the Interface Descriptor instead. The Interface Descriptor's **bInterfaceClass** should be set to **0x08**, the **bInterfaceSubclass** value is generally set to **0x06** indicating that the device uses the Small Computer Systems Interface (**SCSI**) command set, and the **bInterfaceProtocol** value is set to **0x50**, indicating support for **BOT**. Mass storage devices are also required to contain a string descriptor for a 12 digit serial number that's unique for each **idVendor** and **idProduct** pair.

Mass storage devices using **BOT** operate with a Command/Data/Status protocol. The USB host initiates communication with a Command Block Wrapper (**CBW**). The device will acknowledge this, and the host will either send data or receive data from the device. When the data transfer is finished, the device responds with a status message in a Command Status Wrapper (**CSW**). For a much more detailed description of this process, see the bulk-only transport documentation (27).

Most USB mass storage devices use the **SCSI** command set. To work with **SCSI**, the last 16 bytes of the **CBW** are a **SCSI** Command Descriptor Block (**CDB**). **CDBs** represent **SCSI** commands, such as INQUIRY, READ, or WRITE. The INQUIRY command is used to request information about the device, such as the level of compliance with the **SCSI** specifications indicating commands are supported. Unless there's an error processing the command, the response to the command occurs in the data transfer phase, and then the status is returned in the **CSW**. The full **SCSI** standards are published by INCITS Technical Committee T10 (28), but most USB mass storage devices only support a small subset of the **SCSI** command set.

A USB mass storage device isn't aware of file systems – data is written to and read from storage blocks by using Logical Block Addressing (LBA). It's up to the operating system to provide a file system driver to support high-level concepts such as files and directories on the device.

USB MASS STORAGE (29), by Jan Axelson, is an excellent source of information on USB mass storage devices and how they work.

3.5. Attacks using the USB protocols

Data coming from any USB device must not be trusted to comply with the standards. Just like a network protocol stack, the USB protocol stack should be hardened against exploitation.

Exploits against USB driver stacks have been described before. At BlackHat Las Vegas in 2005, David Dewey and Darrin Barrall of SPI Dynamics demonstrated an attack against the Windows XP USB drivers by using a malicious device (8). In 2009, Rafael Dominguez Vega of MWR InfoSecurity found a vulnerability in a USB device driver in Linux (30).

One recent real-world example of how a flaw in a USB driver was able to compromise a system is the Sony PlayStation3 (PS3) jailbreak USB 'modchip', **PSJailbreak** (31). This is a USB key that someone can insert into their PS3 that will allow them to load unauthorized software, such as pirated games. One analysis (32) revealed that this USB key emulates a 6 port USB hub, attaching and removing fake devices in a certain order to manipulate the heap, eventually resulting in a buffer overflow that allows code execution.

A blog post from Fizalkhan Peermohamed on the Microsoft Windows USB Core Team Blog (33) talks about how to properly read and parse USB descriptors. Peermohamed alludes to a case where a USB device driver could allocate a certain amount of memory for a configuration descriptor based on a length field from an initial read, and then the device reports a different length for the second read. This could cause an API call, `USBD_ParseConfigurationDescriptorEx()`, to reference memory outside of the allocated length, resulting in a crash. While this specific scenario has yet to be proven exploitable for code execution, it does provide an interesting example of how a misbehaving device could cause a poorly written device driver to access memory in an unintended way – which is the basis for many security vulnerabilities.

Purposely creating a misbehaving USB device is cheaper and easier than many people are aware of. There are inexpensive development boards that can be programmed to act as any type of USB device. They can send the host anything the programmer wants to send in the USB descriptors, making them useful for carrying USB driver exploit payloads. One of these development boards was used as an attack vector in the Social Engineering Toolkit – it's programmed to act as a keyboard and send keystrokes to a PC, triggering it to download malware (34). The development board currently sells for around US $18.

3.6. Fuzzing USB drivers

A number of techniques can be used to locate security vulnerabilities in device drivers. Reverse engineering and static analysis techniques will work, but fuzz testing can sometimes lead to quicker results. Some research into USB device fuzzing has already been done – Moritz Jodeit at the University

of Hamburg implemented a fuzzer with a combination of hardware and software (35), and Tobias Mueller at Dublin City University created a fuzzer based on QEMU (36).

3.6.1. Windows Device Simulation Framework

The Windows Device Simulation Framework (DSF) is included in the Windows Driver Kit and allows developers to test device drivers by implementing a simulated device in the programming language of their choice (37). It's implemented as a set of drivers that run in kernel mode and a COM API to build applications on. These applications can be rapidly developed a high-level scripting language such as JScript or VBScript, or any language that can interface with COM. The DSF framework for the USB bus includes a simulated EHCI controller that talks to the host OS, which treats it like any other USB controller, making its use transparent to applications. This means that the DSF can be useful for testing USB class and device drivers, but it's probably not useful for stressing host controller drivers themselves.

3.6.2. QEMU/BOCHS

QEMU and BOCHS are open source virtual machine implementations that allow implementing virtual USB devices. This makes it possible to install an OS on a virtual machine and attach simulated devices to it in order to exercise the OS's USB driver stack.

Since these tools also simulate CPUs, it could be possible to implement a USB device fuzz testing framework that traces each instruction to ensure adequate code coverage, similar to what others have done with dynamic instrumentation in user mode (38).

4. USB operation on Windows 7

4.1. USB driver stack

USB 2.0 support has been included in all Windows versions since XP. Support for peripheral devices is implemented as a driver stack (39). The information presented here applies to the 32 bit version of Microsoft Windows 7 Professional, and the function and symbol names were taken from the publicly available debug symbols.

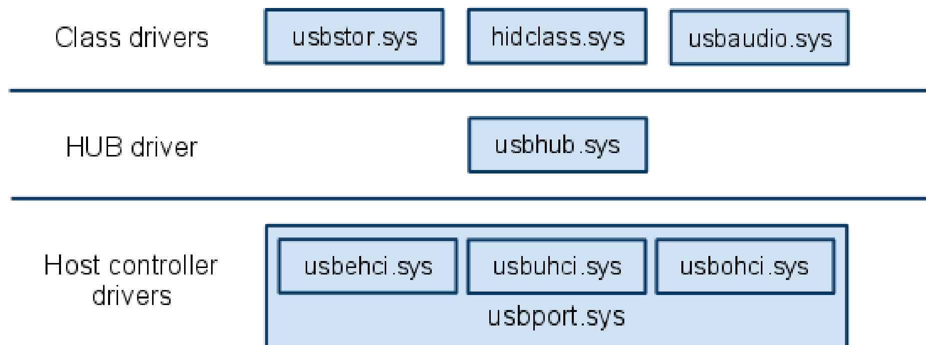

Figure 4 - USB driver stack

4.1.1. Core stack

The USB "core stack" is the set of drivers for the host controller and hubs. These are device independent and handle communicating with the hardware that is already in the PC itself – the USB host controllers and their root hubs.

At the very bottom of the stack are the host controller miniport drivers: `usbuhci.sys`, `usbehci.sys`, and `usbohci.sys`. The purpose of a miniport driver is to communicate directly with the hardware – handling interrupts and I/O.

The miniport drivers are linked against the generic USB port driver (`usbport.sys`) and make a call to the exported function `USBPORT_RegisterUSBPortDriver()` to register themselves as port drivers that can be called from other drivers in the USB stack. The 3rd argument to `USBPORT_RegisterUSBPortDriver()` is a **RegistrationPacket** structure that includes a long list of function pointers that the port driver uses to call into the miniport driver.

To handle hardware interrupts from the USB host controller, the `USBPORT_StartDevice()` function in `usbport.sys` makes a call to `IoConnectInterrupt()` in `ntoskrnl.exe` to register the interrupt service routine (**ISR**) - `USBPORT_InterruptService()`. That ISR will in turn call the ISR that the miniport driver has registered with the port driver, such as `EHCI_InterruptService()` in `usbehci.sys` and `UhciInterruptService()` in `usbuhci.sys`. The ISR for the miniport driver was specified in the **RegistrationPacket** structure. The miniport driver immediately services the interrupt, and then the port driver queues the miniport's

deferred procedure call (**DPC**) to handle the request from the hardware. See MSDN (40) for a description of how ISRs are generally written for Windows.

The miniport driver's ISR and DPC routines communicate directly with the hardware. For EHCI, `EHCI_InterruptService()` and `EHCI_InterruptDpc()` use the `hal.dll` READ_REGISTER_ULONG() function to read status information from the EHCI controller's memory mapped I/O registers. The UHCI miniport driver's `UhciInterruptService()` and `UhciInterruptDpc()` functions use the `READ_PORT_USHORT()` function to communicate with I/O ports.

Above the port driver is the bus driver, or hub driver: `usbhub.sys`. Each hub connected to the host controller is controlled by the bus driver. The bus driver is as close as most USB client drivers will get to the USB device – all interactions with USB devices go through the bus driver. USB client drivers communicate with the bus driver using I/O control calls (**IOCTLs**). The main **IOCTL** used – the one that does the bulk of the work for most USB devices once the device is set up – is `IOCTL_INTERNAL_USB_SUBMIT_URB` (`0x220003`). This **IOCTL** submits an **URB**, or USB request block, to the bus driver. This IOCTL is passed to the bus driver by creating an I/O request packet (**IRP**) with the `IoBuildDeviceIoControlRequest()` function, then the IRP is submitted with `IofCallDriver()`. The bus driver will then pass the URB on to the port driver by doing another **IOCTL** call, where the `USBPORT_ProcessUrb()` function processes it, communicating with the miniport driver if necessary. Because of this architecture, it's not necessary for USB client drivers to know anything about the USB hub and device topology or host controller interface. This makes writing client drivers for USB devices relatively easy.

Composite devices are handled by the USB Generic Parent Driver, `usbccgp.sys`. A generic parent driver means that separate client drivers can handle different functions in the USB device. The purpose of this is to allow selective use of Microsoft-supplied driver support for some interfaces (41). For example, a really fancy gaming mouse could have the mouse pointer aspects handled by the built-in Windows mouse driver, and custom buttons and other functionality handled by a vendor-supplied driver.

4.1.2. Class drivers

The USB Device Working Group (DWG) specifies a list of generic classes of USB devices (25). Each class of device shares a common set of interfaces that allows a single driver to work for devices from any vendor that support that class interface. Windows provides drivers for many of these classes (42). Examples are the HID class (`hidclass.sys`, `hidusb.sys`) that encompasses human interface devices (mice, keyboards, and joysticks), and the mass storage class (`usbstor.sys`) that includes external USB hard drives and flash drives. These class drivers communicate with the bus driver lower in the stack, and more generic feature drivers higher in the stack. For example, the USB Mass Storage Class Driver (`usbstor.sys`) will talk to the native disk class driver (`disk.sys`) that controls all disks in the system, regardless of how they're connected.

4.1.3. USB device recognition

4.1.3.1. Querying the new device

An excellent description of the process Windows 7 goes through when a new USB device is plugged in can be found in Martin Borve's blog post at (43). What follows is a simplified version of the process that focuses on the parts that are of interest to people reverse engineering USB drivers.

When a new device is inserted into a USB port, the host controller will enumerate the devices and functions on that port. The USB bus driver will then request the device descriptor from the device. The device descriptor contains information about the hardware device that the OS needs to load a driver for it – the USB specification version, the USB device class and sub-class, the vendor and product IDs, and other important data. The device descriptor is defined in the USB specifications, but there's also a handy reference at the *USB in a NutShell* page (44) and an example in the Appendix. To read a descriptor, a driver uses the `UsbBuildGetDescriptorRequest()` macro to create an URB with the **GetDescriptor** request, and the URB is passed to the USB bus driver with an `IOCTL_INTERNAL_USB_SUBMIT_URB` request.

The bus driver also requests the configuration descriptor for the device and validates it before it continues to obtain more information about the device. Windows queries a number of other descriptors from the device, depending on fields in the device descriptor: the MS OS Descriptor, Serial Number String Descriptor, MS OS Extended Configuration Descriptor, MS OS Container ID Descriptor, Language ID, Product ID String, and Device Qualifier (43).

Once all of this information is obtained, the kernel mode PnP (Plug and Play) manager is notified of the new device, and the PnP manager will decide which driver to load.

4.1.3.2. Locating the correct driver

A good overview of how the PnP manager handles new devices can be found in the MSDN Library (45).

The kernel mode PnP manager is a core part of the kernel (`ntoskrnl.exe` – the functions named `PnpXXX` and `PipXXX`). A bus driver notifies the PnP manager that a new device has been added by calling `IoInvalidateDeviceRelations()` in `ntoskrnl.exe` (45). A chain of other function calls is made, and eventually `PnpQueryDeviceRelations()` will issue an `IRP_MN_QUERY_DEVICE_RELATIONS` request to the bus driver. In the case of USB, this will be handled by `UsbhFdoPnp_QueryDeviceRelations()` in `usbhub.sys`, which calls `UsbhQueryBusRelations()` to enumerate the devices on the USB bus. When a new device is found, `PiProcessNewDeviceNode()` will add information on it to the system registry in `HKLM\System\CurrentControlSet\Enum\USB` and then call `PnpSetPlugPlayEvent()` with the flag `GUID_DEVICE_ENUMERATED` to let the system know the new device has been enumerated.

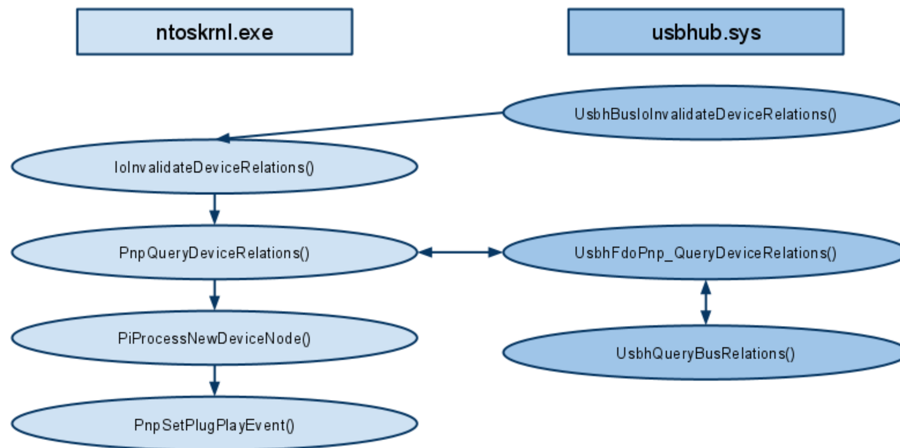

Figure 5 - Processing a new USB device

When the PnP manager learns that a new device is present, it will attempt to find a driver for the device. There are three important identifiers that the USB stack will build for each USB device: the device ID, the hardware ID, and the compatible ID.

For single-interface USB devices, the device ID is a string that has one of these forms:

- `USB\VID_`***v(4)***`&PID_`***d(4)***
- `USB\VID_`***v(4)***`&PID_`***d(4)***`&REV_`***r(4)***

The hardware ID is the same as the device ID in most cases, and the compatible ID has the form:

- `USB\CLASS_`***c(2)***`&SUBCLASS_`***s(2)***`&PROT_`***p(2)***

The values ***v(4)***, ***d(4)***, ***r(4)***, ***c(2)***, ***s(2)***, and ***p(2)*** all come from the USB device descriptor (or the interface descriptor if certain fields in the device descriptor are set to **00**) that was obtained when the device was connected. This table shows which field each value comes from:

Item	Value	Device Descriptor Value
v(4)	vendor ID	idVendor
d(4)	product ID	idProduct
r(4)	revision ID	bcdDevice
c(2)	class code	bDeviceClass
s(2)	subclass	bDeviceSubClass
p(2)	protocol	bDeviceProtocol

There are a slightly different set of device, hardware, and compatible IDs generated by composite devices that use values from the USB interface descriptor; these IDs are described in the MSDN Library (46).

The first place the PnP manager looks for a driver is in the registry in HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum. USB devices that already have drivers installed will have entries for their device IDs in the USB subkey of Enum. There are also special identifiers for mass storage devices and printer devices, under the keys USBSTOR and USBPRINT.

If a match is found in the registry, the PnP manager loads the driver, calls the DriverEntry() function, and then calls the driver's AddDevice() routine which creates the function device object (FDO) and attaches it to the bus's stack.

If a driver is not found, Windows will search Windows Update for the correct driver, and then local **DriverStore**. This process happens in the Plug and Play service, umpnpmgr.dll. To check Windows Update for drivers, the user mode plug and play manager makes calls to an external DLL file – chkwudrv.dll, which is configured as the driver finding 'plugin' in the registry value HKLM\Software\Microsoft\Windows\CurrentVersion\DriverSearching\Plugin\WUSearchLibrary. chkwudrv.dll makes calls to the Windows Update API COM objects (in wuapi.dll) to download an appropriate driver from Windows Update, if one is found.

4.1.4. The danger of drivers from Windows Update

It's possible for any 3rd party who can obtain a VeriSign Class 3 Organizational Certificate and write a device driver that passes the WHQL Windows Logo requirements to submit a driver to **WinQual** (the Microsoft site that gives developers access to application error reporting logs and allows uploading of drivers for Windows Update) that can be automatically installed on a Windows Vista or Windows 7 system when a matching device is plugged in.

While the Level 3 certificate requirement and WHQL testing are designed to ensure only high quality, professionally written drivers are uploaded, it could be possible for a malicious entity to create a corporation, obtain a certificate, develop a backdoored driver for a non-existent device, fulfill the testing requirements, submit the signed driver, then wait for their submission to be approved. Since Microsoft accepts signed driver binaries and not source code, it could be possible for a driver author to slip in some obfuscated malicious code. Unless Microsoft has a team of super-star analysts manually disassembling and analyzing each submitted driver, it's possible for something malicious to slip through the cracks.

Even without going through the effort of obtaining the Class 3 certificate, it could be possible for a malicious person or group to simply steal the WinQual and certificate credentials. The Stuxnet authors used stolen certificates and credentials to sign rootkit drivers used by the worm, and it's not a stretch to think that someone could use a stolen key to upload malicious drivers to WinQual.

For this attack to work, a malicious driver would be registered with WinQual with an INF file for a specific device ID. In the case of USB, it needs to be a unique Vendor and Product ID within the USB device descriptor. When a device matching that ID is inserted into a Windows Vista or Windows 7 PC with automatic driver installation enabled (as it is by default), the PC will connect to Windows Update and request that driver. Once installed, the driver can send a message to the device indicating that it's running, and the device itself can send a "secret message" back to the driver. This could be a response

indicating that the driver should initiate a backdoor, perhaps to unlock the screensaver. It could even exploit a purposely placed vulnerability to exploit malicious code. Since the WHQL Windows Logo tests are self-administered and driver vendors submit the testing logs for approval, it's possible to tweak the backdoored driver if it fails any tests before submitting it for approval.

For this reason, it's not wise to have automatic driver installation enabled on systems in potentially hostile environments. Windows 7 will install drivers for new devices even when nobody is logged into the system, since the Plug and Play service doesn't require user interaction to install new drivers as it did with Windows XP.

4.2. Mass storage devices

The Storage Management chapter of Mark Russinovich's **WINDOWS INTERNALS** (5th Edition) contains a lot of information about how disks and the storage subsystem work in Windows. The MSDN library also has a couple of sections dedicated to storage (47) (48). The information presented here is just an overview of the storage subsystem as it applies to USB mass storage devices.

4.2.1. USB storage port driver and Windows disk class driver

USB devices with the Mass Storage Class code are handled by the generic USB mass storage class driver, `usbstor.sys`, which is a storage port driver. It registers with the Windows generic PnP disk class driver (`disk.sys`) to provide an interface between the Windows storage subsystem and the physical USB hardware. The disk class driver creates disk device objects that are accessed by other parts of the storage subsystem. These device objects are named `\Device\HardDiskX\DRX`, where X starts with 0 for the boot disk and increases as disks are added.

Layer	Driver
Filesystem Driver	ntfs.sys
Volume Management	volmgr.sys
	fvevol.sys
	volsnap.sys
Partition Mgmt	partmgr.sys
Storage Class	disk.sys
	usbstor.sys
Bus Drivers	usbhub.sys
	usbehci.sys
	usbport.sys

Figure 6 - Windows USB mass storage stack

4.2.2. Partition and volume management

The partition manager, partmgr.sys, manages partitions. This driver reads partition tables from disks as they're connected using the IoReadPartitionTableEx() function in ntoskrnl.exe. This function builds an IRP_MJ_READ request for the first four sectors of the disk (2048 bytes) using the IoBuildSynchronousFsdRequest() function, then sends the IRP to the driver specified by the **DeviceObject** argument. IoReadPartitionTableEx() makes sure the boot sector is valid by ensuring the last 2 bytes are **0xAA55** (**55AA** on disk) and then fills out the _DRIVE_LAYOUT_INFORMATION_EX structure based on the partition table to return to the caller. This function works with both MBR (Master Boot Record) partitions and GPT (GUID Partition Table) partitions. The partition manager notifies volume managers when new disks are online or new partitions are created or removed.

Volumes are the basic unit of disks that are visible to the rest of the Windows OS, and the volume manager is responsible for keeping track of them and allowing other components to interact with them. The basic disk volume manager is volmgr.sys. Besides basic disks, Windows supports dynamic disks which use a different volume manager driver, but removable storage devices are always considered basic disks.

When a new volume is added, a drive letter is assigned. It's not actually mounted until the first attempt to access a file or directory on the volume. During the mounting process, which happens as part of the I/O manager in IopMountVolume() in ntoskrnl.exe, each file system driver registered for that type of storage device (Network, CD-ROM, Disk, Tape) is called until one recognizes and claims the volume.

4.2.3. File system drivers

File system drivers exist above the volume manager and provide an interface for interacting with file system concepts such as files and directories. File system drivers register with the system by making a call to IoRegisterFileSystem() in ntoskrnl.exe, which inserts the driver's device object into one of four global queues, depending on the storage medium type as set by the **DeviceType** member of **DeviceObject**.

The drivers for file systems natively supported by Windows 7 don't actually register themselves – there is a driver called fs_rec.sys, the File System Recognizer Driver, which registers on behalf of the drivers. The reason for this is so Windows doesn't need to unnecessarily keep drivers loaded in memory for file systems that aren't being utilized – fs_rec.sys will recognize and load the file system drivers supported by the OS. The file systems and device types recognized by fs_rec.sys are in the table below.

Filesystem	Device Type	Driver
CDFS	CD-ROM	cdfs.sys
UDF	CD-ROM	udfs.sys
UDF	DISK	udfs.sys
FAT	DISK	fastfat.sys
FAT	CD-ROM	fastfat.sys
NTFS	DISK	ntfs.sys
ExFAT	DISK	exfat.sys

CDFS is the file system used by CD-ROM discs and is also known as ISO 9660. **UDF**, or Universal Disk Format, is used by DVD ±R/W devices, but the driver also handles CD ±R/W disks since it was designed with the ability to write to optical disks as well as read from them. The UDF specifications are available from OSTA, the Optical Storage Technology Association (49). The **FAT**, or File Allocation Table, file system was used by Microsoft Standalone Disk BASIC and later used in DOS (50). It's still widely used today and many USB flash drives are pre-formatted with FAT32. The specifications for FAT32 are available from Microsoft (51). **NTFS** is the preferred file system for Windows systems because of the security and reliability features it offers. It's also the most complex – `ntfs.sys` is around 1.2MB compared to 150KB for `fastfat.sys`. The NTFS specifications are not public. **ExFAT**, or Extended FAT, is a new version of FAT developed especially for USB devices and included with Vista SP1 and later versions of Windows (52). The ExFAT specifications are licensed by Microsoft but not publicly available.

The source code for versions of `cdfs.sys` and `fastfat.sys` are included in the Windows Driver Kit (WDK) as sample file system drivers and can be very useful for understanding the structure of file system drivers when attempting to reverse engineer one of the drivers that don't have source available.

4.2.4. Fuzzing filesystem drivers on Windows

It's possible to implement a very simple file system driver fuzzer by using **FileDisk** (53), by Bo Brantén. This tool allows mounting a file image on an existing volume as a file system. I developed two proof-of-concept fuzzers – one that randomly perturbed bytes in a file image and then mounting it, and one that was part of the driver itself that modified bytes as they were read from a disk image. Both techniques were able to crash (blue screen) systems, but more research should be done in this area for smarter fuzzing and crash analysis. Since Windows will search for certain files and directories and read certain files when a file system is mounted, even with **AutoPlay** disabled, it could be possible to construct a file system image that stresses different parts of the driver. An example of this is putting a large `autorun.inf` file on an image and compressing it at the NTFS level or even simply spreading the file out across many sectors.

4.3. Exploiting USB and file system drivers

There have been a number of successful exploits for USB drivers that were mentioned above in the section Attacks using the USB protocols. I don't know of any successful exploits for the Windows 7 USB driver stack that have been made public.

I'm also not aware of any publicly disclosed vulnerabilities that could be exploited with a maliciously crafted file system image written to a USB disk. There was a vulnerability in the MacroVision SafeDisc filter driver (MS07-067), but that was a local privilege escalation vulnerability exploited by communicating directly with the driver from a user-mode application. In 2008 Daniel Roethlisberger of Compass Security AG presented a demonstration of an attack on the VMWare Tools `HGFS.sys` file system driver, but again that was a local privilege escalation exploit making use of IOCTL calls.

The Windows 7 kernel includes several mitigations designed to prevent successful exploitation of bugs in kernel drivers. There's a form of Address Space Layout Randomization (ASLR) for drivers that will randomly load each driver module into one of 64 different addresses. The kernel also implements safe unlinking for kernel pools, making exploitation of kernel pool overwrites somewhat more difficult (54).

4.4. PnP Manager

Once the volume manager gets control of a newly-connected volume, it notifies the Plug and Play manager – the same subsystem that's in charge of recognizing that a new USB device was connected.

4.4.1. Kernel mode PnP manager

The kernel mode PnP manager code resides in the NT kernel executive (`ntoskrnl.exe`). It works in concert with the I/O manager to handle devices being added and removed from the system. It also ensures that PnP devices receive proper notifications for power management events, like entering sleep mode or shutting down. The PnP manager tracks the device tree – the hierarchical structure of devices that are connected to the system. As devices are added or removed, device drivers make calls to the PnP manager to have it refresh the device tree and take action on any changes that are found. The device tree can be viewed on a Windows PC through the Device Manager application – there is a "View by Connection" option in the View menu that will display devices in a tree format similar to how the PnP manager tracks them. DeviceTree (55), an application developed by Mark Cariddi, provides a more detailed view of the device tree and the relationships between devices and drivers.

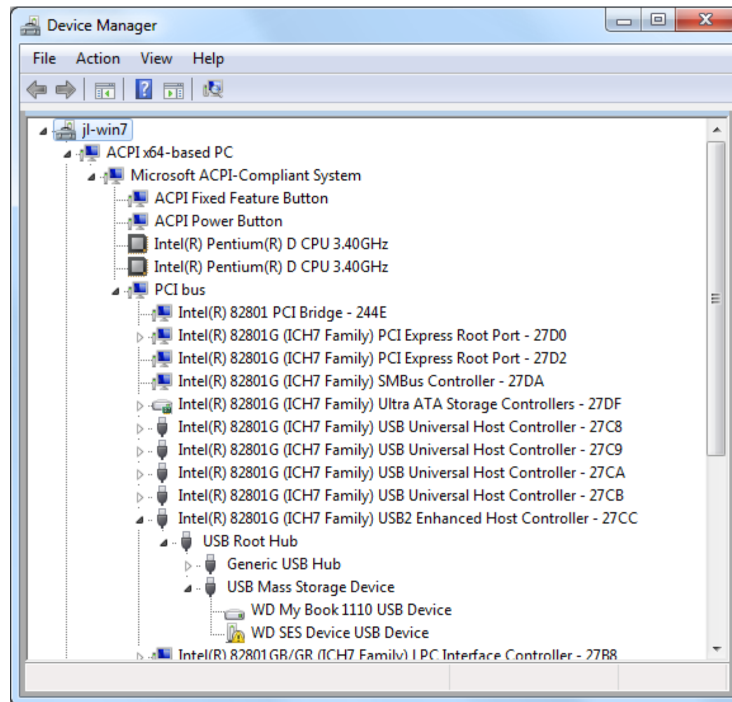

Figure 7 - Device Manager

The USB Device Recognition section provides an overview of how the PnP manager works with USB devices and drivers.

4.4.2. User mode PnP manager

The user mode plug and play manager is the "Plug and Play" service, `umpnpmgr.dll`. This service has a thread running that waits for plug and play events from the kernel by using the undocumented system call `NtGetPlugPlayEvent()`.

An application that wants to receive notifications on plug and play events can use the `RegisterDeviceNotifications()` API call in `user32.dll`, which uses `cfgmgr.dll` to make a RPC call to the PnP service for event registration. This RPC call is handled in the function `PNP_RegisterNotification()` in `umpnpmgr.dll`, which adds the recipient handle to a list of handles to notify for events.

4.5. AutoPlay

4.5.1. Shell Hardware Detection Service

4.5.1.1. Detecting new devices

Most of the functionality of AutoPlay is in the Shell Hardware Detection Service, `shsvcs.dll`. This service registers for callbacks when a new device is connected. The first registration happens during the `PnPServices::Initialize()` function, when `CRegisterNotificationOnAllInterfaces::Register()` calls `RegisterDeviceNotification()` in `user32.dll`. There are two flags set in the Flags

argument. DEVICE_NOTIFY_SERVICE_HANDLE (0x1) indicates that the notifications are sent to the service control handler with the SERVICE_CONTROL_DEVICEEVENT (0x0B) control code, and DEVICE_NOTIFY_ALL_INTERFACE_CLASSES (0x4) specifies that the recipient should receive notifications for all devices classes.

The service control codes are handled by the GSM::HandleServiceControls() function in shsvcs.dll. When the device event code is sent, PnPServices::HandleDeviceEvent() takes control. PnPServices::HandleInterfaceEvent() gets called for interface events. At this point, the code will also register for events specific to this device by calling RegisterDeviceNotification() again, but specifying the device handle for the newly connected device.

There are two events that are sent and handled when a new storage device is connected – because the service registers for both DBT_DEVTYP_DEVICEINTERFACE (0x5) and DBT_DEVTP_HANDLE (0x6) events. The interface event is handled by PnPServices::HandleInterfaceEvent(), the handle event by PnpServices::HandleBroadcastHandleEvent(), but eventually they'll both call Storage::CVolumeInfo::UpdateMediaInfo() to determine what kind of media are on the storage device. This function checks for the presence of certain files and directories in the root directory of the device and setting certain flags if the files exist that are later checked by other user mode components. The list of files and directories, and what their existence signifies, is:

File	**Purpose**
autorun.inf	Autorun file
desktop.ini	Desktop.ini file
video_ts\\video_ts.ifo	DVD Video
dvd_rtav\\vr_mangr.ifo	DVD Video
audio_ts\\audio_ts.ifo	DVD Audio
VCD\entries.vcd	Video CD
SVCD\entries.svd	Super Video CD
SVCD\entries.vcd	Super Video CD
DCIM	Photos
BDMV	Blu-ray disc
BDAV	Blu-ray disc

The shell hardware detection service doesn't actually read and parse any of these files or directories (other than autorun.inf) – it just uses their existence to determine what kind of media are on the storage device so that the AutoPlay dialog knows which options to display.

The Storage::CVolumeInfo::UpdateMediaInfo() function also reads the label and icon path from the autorun.inf file (in Storage::CVolume::_ExtractAutorrunIconAndLabel()). This function also checks to see if the **UseAutoPlay** setting is present and if it's equal to 1, signifying that Windows should ignore the

autorun.inf file and use **AutoPlay** to handle the device. The final check on the autorun.inf file is to see if it contains an **Open** or **UseShellExecute** line, meaning that the autorun.inf file is specifying a file to execute.

UpdateMediaInfo() also checks to see if the volume is encrypted with **BitLocker** by dynamically loading fveapi.dll and calling the FveOpenVolumeW() and FveGetStatus() functions.

These checks are done even if AutoPlay is disabled and when no user is logged into the machine. This code is also running at a very high privilege level as **LocalSystem**. While the amount of parsing done on files is extremely limited, it could be possible to use the fact that the OS is reading files from the file system to attack the file system driver.

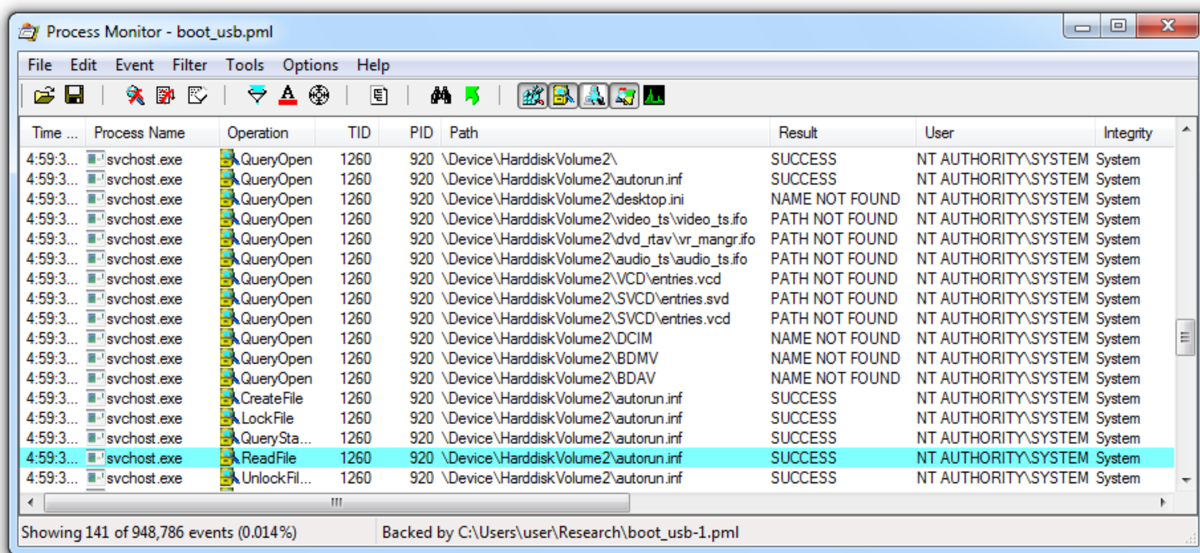

Figure 8 - Process Monitor showing USB access before user logon

4.5.2. ReadyBoost

ReadyBoost is a feature introduced in Windows Vista that allows a user to use a flash storage device as a disk cache. The idea behind this is that random data accesses are much faster on flash media than on a hard disk, so performance of the system can be increased by caching access to files. The **SuperFetch** service (sysmain.dll hosted by svchost.exe) registers for device events using RegisterDeviceNotificationW(). When a new flash storage device is inserted into the PC, the function RdbDeviceProcessExisting() in sysmain.dll will delete the file ReadyBoostPerfTest.tmp if it exists. If the file ReadyBoost.sfcache exists, the first 65536 bytes are read into memory and the first 4 bytes are checked to see if they match the ReadyBoost file header ('EcMg'). If the header is there, the file is removed. These checks are done even if AutoPlay is disabled.

5. Windows Explorer

Windows Explorer is the shell of the Windows operating system – it provides the graphical interface that users interact with every time they use the PC. To make the shell more pleasant to use, Windows allows the development of extensions that allow customization of the user experience. For example, a program can install a Shell Icon Handler that can render custom icons for files of a certain type. This can be used to show icons that are thumbnail images of pictures or previews of documents.

Much of the information in this section comes from documentation in the MSDN Library, supplemented by reverse engineering of relevant code. DLL file names and symbols are given to aid other reverse engineers in understanding potential vulnerabilities in the code. Symbol names and reverse engineering information comes from the 32 bit version of Windows 7 Professional. The information should also be valid for Windows Vista, but definitely not for Windows XP or Windows Server 2003.

5.1. Shell Extension Handlers

Shell Extension Handlers are used to provide custom capabilities for certain files and folders in the Windows shell. The shell will query these extensions before file operations are performed to determine if an extension should handle that operation. Shell extensions can provide custom icons for files, custom file drop handlers for folders, custom property sheets displayed in a file's Properties view, and custom Infotips that are displayed when users hover the mouse over a file or folder. The MSDN Library has a lot of information on how these extension handlers are implemented (56).

A vulnerability in a shell extension handler could allow code to be executed without a user's knowledge. The LNK exploit used by Stuxnet is an example of how a vulnerability in the icon handler for .LNK files lead to code execution, causing a PC to become infected with malware from a USB drive without relying on `autorun.inf`. In early 2009, Didier Stevens wrote a blog post about how to trigger a certain PDF exploit without actually opening the PDF file by abusing the preview, thumbnail, and property (metadata) handlers (57). In December of 2010, Moti and Xu Hao presented "A Vulnerability in my Heart" at the POC2010 conference in South Korea, detailing a vulnerability in a thumbnail handler in Windows XP (58), and in January 2011 Joshua Drake released a Metasploit module for the vulnerability.

Shell extension handlers are implemented as COM objects. The screenshot below shows an example of which COM interface is responsible for displaying each part of the Explorer window. In this example, from Windows 7, the view was 'Large Icons' and the preview pane was enabled.

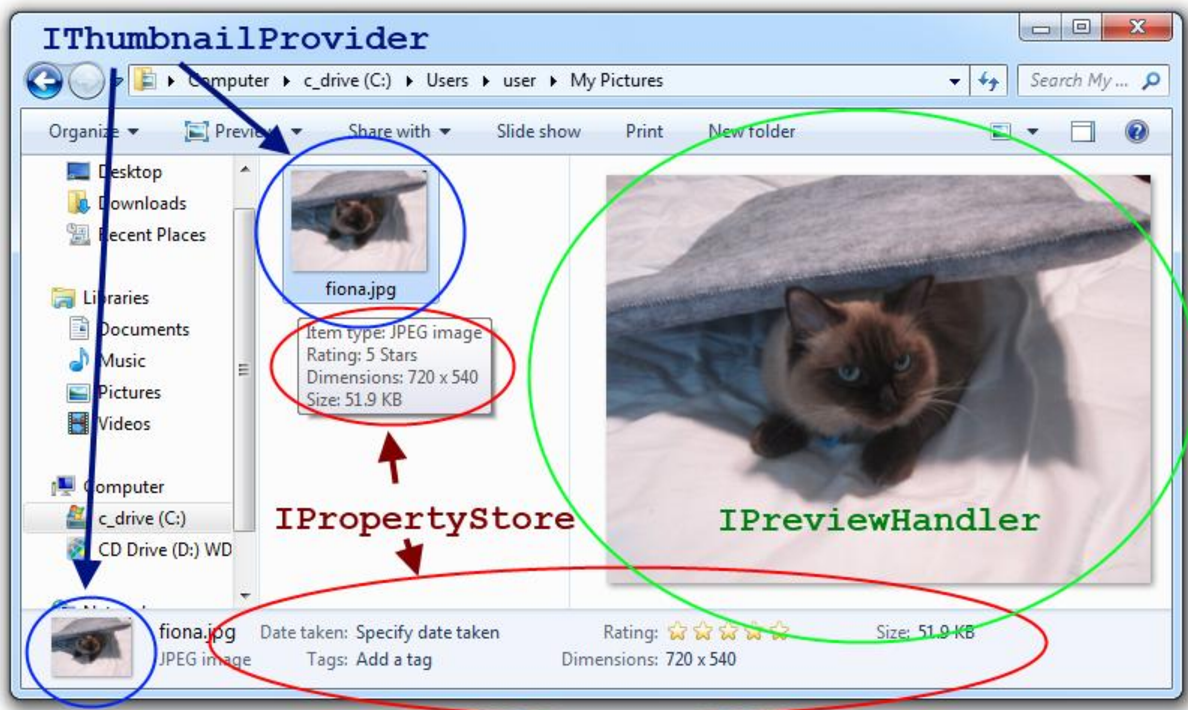

Figure 9 - Shell extension handlers

As you can see, the **IThumbnailProvider** interface displays the preview icon in both the main widow and the information pane at the bottom. **IPropertyStore** is used to obtain the information in the Infotip, which is displayed when you hover over a file with the mouse, and the information pane at the bottom. **IPreviewHandler** renders a preview of the image for the preview pane on the right. Each of the circles represents some information that Windows obtained from parsing the file.

5.1.1. Registered file types and perceived types

Files in windows are 'typed' based on the extension. In other words, Windows determines what sort of content a file contains based on the extension – the last part of a file name following the last period. For example, Windows assumes that a file ending in "`.BMP`" is a bitmap file – an image. How Windows responds to user interactions with that file is determined by the information contained in the registry. What program opens when a file is double-clicked, what options are shown on the context menu when a file is right-clicked, and what information shows up in the Infotip when the mouse is hovered over a file are all configured in the Windows registry. Shell extensions that operate on files and file types are also configured in the registry. There are several places a shell extension could be configured for a file type:

- `HKEY_CLASSES_ROOT\`*`.ext`*
- `HKEY_CLASSES_ROOT\`*`ExtProgId`*
- `HKEY_CLASSES_ROOT\SystemFileAssociations\`*`.ext`*
- `HKEY_CLASSES_ROOT\SystemFileAssociations\`*`ExtProgId`*
- `HKEY_CLASSES_ROOT\SystemFileAssociations\`*`extperceivedtype`*

`.ext` is the file extension, *`ExtProgId`* is the `ProgId` (another string representing the file type, which is referenced in the file extension entry), and *`extperceivedtype`* is the perceived type of the file. In addition to `HKEY_CLASSES_ROOT`, shell extensions can be registered under the `HKEY_CURRENT_USER\Software\Classes` key. The section for each specific handler lists which registry keys Explorer will look for handlers for certain file types.

A **perceived type** is a category that a file belongs to, such as "image" or "document" (59). A perceived type for a file extension is also configured through the registry (60), for example:

```
HKEY_CLASSES_ROOT\.jpg\PerceivedType = "image"
```

Perceived types can also be configured by `ProgId` or in the `SystemFileAssociations` section of the registry.

A major consequence of this flexibility is that it can be difficult to locate a shell extension that handles a given type or to fully enumerate all installed shell extensions.

5.1.2. Icon handlers

Icon handlers allow Windows Explorer to display a custom icon for each individual file (61). Instead of registering a file type (62) to display the same icon for each file with a given extension, Windows allows developers to implement and register an icon handler COM object that can display a different icon for each file of a given extension based on other aspects of the file, such as the file contents or metadata. For example, an icon handler registered for `.msc` files will get called for each file with a `.msc` extension in a folder to determine which icon should be displayed for each file.

The MSDN documentation states that icon handlers for file types should be registered by creating a `ShellEx\IconHandler` subkey under the entry for the `ProgId` under the `HKEY_CLASSES_ROOT` registry hive (61). However, Windows will look under several keys for the `ShellEx\IconHandler` key for a type. Here is a list of where Windows will look for the icon handler for a file ending in `.jpeg`, which has a ProgId of '`jpegfile`' and a perceived type of '`image`'. There is no icon handler registered for these files.

- `HKCU\Software\Classes\jpegfile\ShellEx\IconHandler`
- `HKCR\jpegfile\ShellEx\IconHandler`
- `HKCU\Software\Classes\SystemFileAssociations\.jpeg\ShellEx\IconHandler`
- `HKCR\SystemFileAssociations\.jpeg\ShellEx\IconHandler`
- `HKCU\Software\Classes\SystemFileAssociations\image\ShellEx\IconHandler`
- `HKCR\SystemFileAssociations\image\ShellEx\IconHandler`

The default value of `IconHandler` is the CLSID for the icon handler – a COM object implementing the **IExtractIconA** or **IExtractIconW** interface. Here's an example of the set of registry entries for the icon handler for `.msc` files:

```
HKEY_CLASSES_ROOT
  .msc = "MSCFile"

HKEY_CLASSES_ROOT
  mscfile
```

```
    shellex
      IconHandler = "{7A80E4A8-8005-11D2-BCF8-00C04F72C717}"

HKEY_CLASSES_ROOT
  CLSID
    {7A80E4A8-8005-11D2-BCF8-00C04F72C717}
      InprocServer32 = "%SystemRoot%\system32\mmcshext.dll"
```

These entries specify that the icon handler for files with a `.msc` extension are handled by the COM object with CLSID `{7A80E4A8-8005-11D2-BCF8-00C04F72C717}`, which can be found in the `mmcshext.dll` file. The `CFileSystemString::LoadHandler()` function called by `CFSFolder::_CreatePerInstanceDefExtIcon()` is the code in `shell32.dll` that locates, loads, and initializes the icon handler COM objects.

Icon handlers must implement the **IExtractIcon** interface, as well as another interface that's used to initialize the object. In Windows XP, it was the **IPersistFile** interface and `IPersistFile::Load()` was used for initialization. To initialize an icon handler that implements **IPersistFile**, the `Load()` function is called with the full path of the file to load an icon for. This function would generally store the file name as a local member of the class and return. Starting with Windows Vista, Microsoft added support for using different interfaces for initializing icon handlers – **IInitializeWithFile**, **IInitializeWithItem**, or **IInitializeWithStream** (56). In Windows 7, the code in `CFileSystemString::HandlerCreateInstance()` called by `CFileSystemString::LoadHandler()` first attempts to use **IInitializeWithStream** and if this fails then **IIniitalizeWithFile** is tried. **IPersistFile** is only called as a last resort.

After the icon handler is initialized, the shell calls `IExtractIcon::GetIconLocation()`. This function will return the path to the file that contains the icon for the requested file. If the icon is to be extracted from the file itself, the `GetIconLocation()` implementation will set the flag `GIL_NOTFILENAME` (`0x8`) for the last argument – **pwFlags**. Otherwise, the path to the file containing the icon is copied to the buffer pointed to by the **pszIconFile** argument. `GetIconLocation()` implementations need to be careful to not copy more characters than specified in the **cchMax** argument, which is typically `MAX_PATH`. Next the shell will call the `IExtractIcon::Extract()` function, passing the path and icon index returned from `GetIconLocation()`. If the `Extract()` method extracts the icon from a file itself, it will return handles to small and/or large icons in the **phiconLarge** and **phiconSmall** arguments. It's also possible for both of those handles to be `NULL` and to return `S_FALSE` from `Extract()` – in this case, the shell will pass the file name and icon index to the `PrivateExtractIconsW()` function in `user32.dll`. That function loads the file as a DLL (using `LoadLibraryExW()` and using the `LOAD_LIBRARY_AS_DATAFILE` flag) and attempts to load the icon from the file's resources.

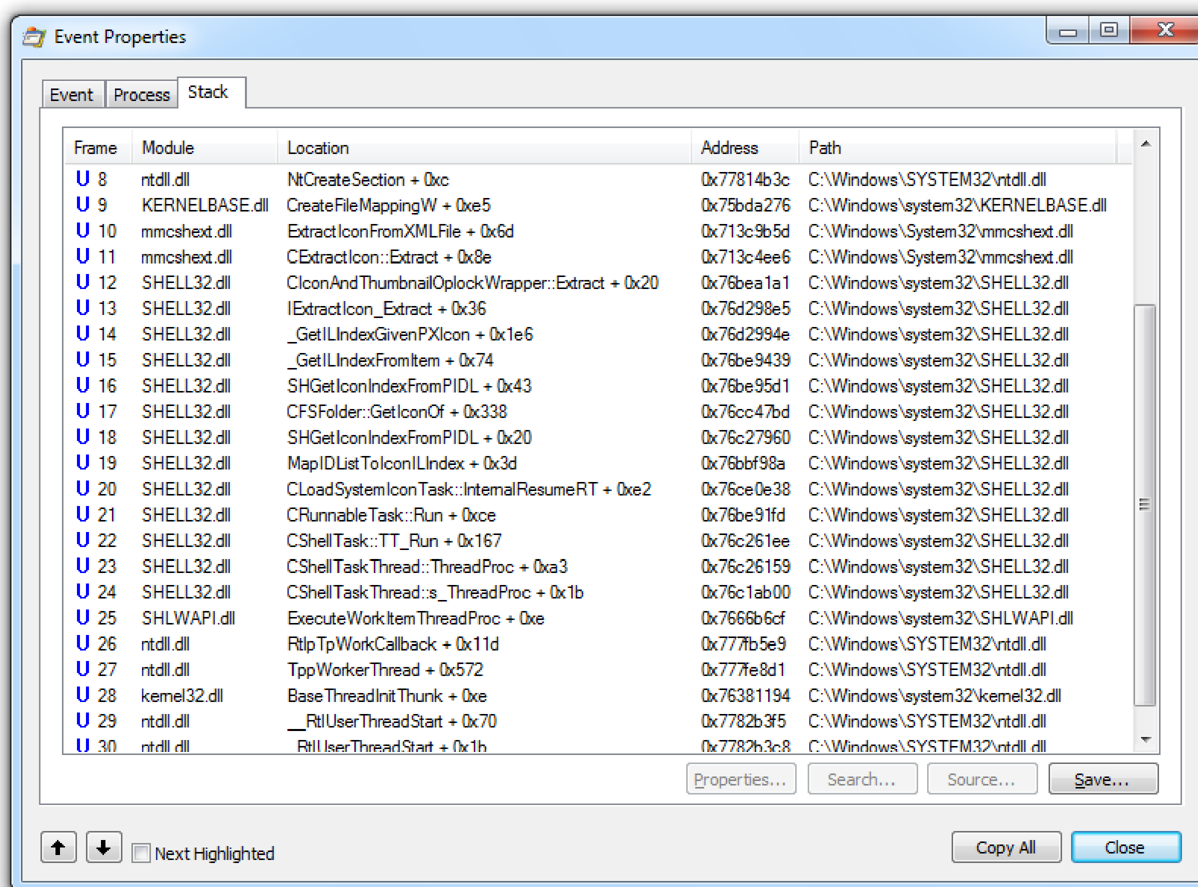

Figure 10 - Process Monitor stack trace of icon handler for .msc files

5.1.3. Thumbnail handlers

Thumbnail handlers are used to provide a small image used to represent a file (63). These image representations, or thumbnails, are used in the Windows 7 Explorer views for displaying Medium Icons, Large Icons, and Extra Large Icons. A thumbnail handler can either load a thumbnail image that was pre-rendered and stored inside of a file or generate one dynamically. Microsoft recommends that files contain pre-rendered thumbnail images for performance reasons (64), but it's also good practice for security, to avoid rendering questionable documents on untrusted removable storage devices. Embedding pre-rendered thumbnails doesn't always prevent security issues though, as Moti and Xu Hao showed at POC2010 (58) – that vulnerability was exploited with an embedded thumbnail image.

Thumbnail handlers are COM objects that implement the **IThumbnailProvider** interface, CLSID `{E357FCCD-A995-4576-B01F-234630154E96}`. Thumbnail handlers are usually registered by adding an entry for that CLSID under the `ShellEx` subkey of the extension or `ProgId` for a file. Windows will look in a few different locations in the registry for thumbnail handlers – it could be registered by extension, `ProgId`, or perceived type. Here is an example of where Explorer looks for a thumbnail handler with a file ending in `.ini`, whose `ProgId` is '`inifile`' and the perceived type is '`text`':

- `HKCU\Software\Classes\inifile\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR\inifile\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCU\Software\Classes\.ini\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR\.ini\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCU\Software\Classes\SystemFileAssociations\text\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR\SystemFileAssociations\text\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCU\Software\Classes*\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR*\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCU\Software\Classes\AllFilesystemObjects\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR\AllFilesystemObjects\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`

Additionally, if a file extension doesn't have a ProgId in the registry, these keys will be searched:

- `HKCU\Software\Classes\Unknown\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR\Unknown\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCU\Software\Classes*\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR*\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCU\Software\Classes\AllFilesystemObjects\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`
- `HKCR\AllFilesystemObjects\ShellEx\{E357FCCD-A995-4576-B01F-234630154E96}`

Here's an example of the registered thumbnail handler for AVI files:

```
HKEY_CLASSES_ROOT
  .avi
    ShellEx
      {e357fccd-a995-4576-b01f-234630154e96} = "{9DBD2C50-62AD-11D0-B806-00C04FD706EC}"

HKEY_CLASSES_ROOT
  CLSID
    {9DBD2C50-62AD-11d0-B806-00C04FD706EC}\InProcServer32 =
      "SystemRoot%\system32\shell32.dll"
```

By default thumbnail handlers will run in an isolated process (the COM Surrogate host, `dllhost.exe`), but that feature can be disabled by setting a registry value named **DisableProcessIsolation**, in the subkey for the handler's **CLSID**. The isolated process runs under the same isolation level and as the same user as `explorer.exe` itself, so any malicious code executed from a thumbnail handler will still have the same rights and permissions as the logged on user.

5.1.3.1. Thumbnail Property Handler

Many of the default thumbnail handlers included in Windows 7 make use of the Property Thumbnail Handler, CLSID `{9DBD2C50-62AD-11D0-B806-00C04FD706EC}`, located in `shell32.dll` as the **CPropertyThumbnailHandler** class.

This class is a generic thumbnail handler that makes use of the Windows Property System described below. It exposes interfaces for **IThumbnailProvider**, **IExtractImage**, and **IExtractIconW**. The main functionality of this class is in `CPropertyThumbnailHandler::_GetThumbnailInternal()`. This function attempts to load the thumbnail from three different property keys: first **PKEY_Thumbnail**, then **PKEY_ThumbnailStream**, and then **PKEY_ImageParsingName**. If found, the thumbnail is converted to the format required by `IThumbnailProvider::GetThumbnail()`, which is an **HBITMAP**.

5.1.3.2. Folder Thumbnails

When viewing a folder in Explorer in the Medium, Large, or Extra Large icon view, it will display thumbnails for folders that contain skewed images of thumbnails of up to two files within that folder. For example, if the current folder contains a folder named "`catpics`" with 3 files named "`cat1.jpg`", "`cat2.jpg`", and "`cat3.jpg`", the folder's thumbnail will show images of `cat1.jpg` and `cat2.jpg`. The implications of this are that thumbnail and icon handlers can be run on files that aren't in the directory currently being viewed, which could hide the true source of a vulnerability that gets exploited. The `CFolderThumbnail` class in `shell32.dll` is responsible for generating folder thumbnails, and the `CombineThumbnails()` function does the actual work of reading, parsing, skewing, and combining the thumbnail images.

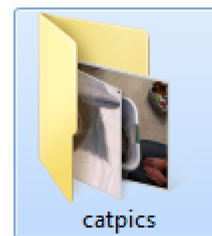

Figure 11 - Folder thumbnail

5.1.4. Image handlers

Originally, thumbnail handlers in Windows implemented the **IExtractImage** interface, which has been largely deprecated in Windows Vista in favor of **IThumbnailProvider**. There are still quite a few registered **IExtractImage** implementations in Windows 7, so it's still worth mentioning. Windows will use the **IThumbnailProvider** interface if available for the file type, but will fall back on the **IExtractImage** interface. The **IExtractImage** interface exposes two methods – `GetLocation()` and `Extract()`.

IExtractImage thumbnail providers are registered by adding an entry for the CLSID `{BB2E617C-0920-11d1-9A0B-00C04FC2D6C1}` under the `ShellEx` subkey of the file extension or `ProgId` for the file. Like other shell extension handlers, Explorer will actually look in a number of additional locations for configured thumbnail handlers. Here are the keys Explorer examines when trying to find an **IExtractImage** object for files ending in `.ini`:

- `HKCU\Software\Classes\inifile\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR\inifile\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCU\Software\Classes\.ini\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR\.ini\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCU\Software\Classes\SystemFileAssociations\text\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR\SystemFileAssociations\text\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCU\Software\Classes*\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR*\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCU\Software\Classes\AllFilesystemObjects\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR\AllFilesystemObjects\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`

Like with IThumbnailHandler, there are also additional keys that are checked if there is no ProgId for the file:

- `HKCU\Software\Classes\Unknown\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR\Unknown\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCU\Software\Classes*\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR*\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCU\Software\Classes\AllFilesystemObjects\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`
- `HKCR\AllFilesystemObjects\ShellEx\{BB2E617C-0920-11D1-9A0B-00C04FC2D6C1}`

Here are the registry entries for the registered **IExtractImage** thumbnail handler for TTF (TrueType font) files:

```
HKEY_CLASSES_ROOT
  .ttf = "ttffile"

HKEY_CLASSES_ROOT
  ttffile
    shellex
      {BB2E617C-0920-11d1-9A0B-00C04FC2D6C1} = {B8BE1E19-B9E4-4ebb-B7F6-A8FE1B3871E0}

HKEY_CLASSES_ROOT
  CLSID
    {B8BE1E19-B9E4-4ebb-B7F6-A8FE1B3871E0}
        InProcServer32 = "%SystemRoot%\system32\fontext.dll"
```

5.1.5. Preview handlers

Preview handlers are used to provide a light-weight, read-only representation of the contents of a file without actually running the file's associated application (65). Preview handlers only display a preview when a file is selected in an Explorer window.

Preview handlers implement the **IPreviewHandler** interface, as well as a few others. Preview handlers are registered by adding an entry for the CLSID `{8895B1C6-B41F-4C1C-A562-0D564250836F}` under the `ShellEx` subkey of the file extension, `ProgId`, or perceived type registry entry. Here's an example of the registration information for the preview handler for HTML files:

```
HKEY_CLASSES_ROOT
  .html = htmlfile

HKEY_CLASSES_ROOT
  htmlfile
    shellex
      {8895B1C6-B41F-4C1C-A562-0D564250836F} = "{f8b8412b-dea3-4130-b36c-5e8be73106ac}"

HKEY_CLASSES_ROOT
  CLSID
    {f8b8412b-dea3-4130-b36c-5e8be73106ac}
      InprocServer32 = "%SystemRoot%\system32\inetcomm.dll"
```

Unlike the shell extension handlers mentioned above, preview handlers always run out-of-process. This can either be an in-process COM server that runs in a surrogate host (`prevhost.exe`), or the extension can be implemented as its own COM server. Microsoft recommends developing preview handlers that run inside of `prevhost.exe`. By default, this process runs at a low integrity level to protect the system against successful exploitation of security vulnerabilities. This means that any exploit for a protected preview handler will need to exploit a separate vulnerability to increase the integrity level, such as a local privilege escalation exploit. It is possible for preview handlers to disable running in a low integrity level process by creating a DWORD value in its CLSID registry entry named

DisableLowILProcessIsolation with the value of 1. A number of preview handlers included with Windows have this feature disabled, and are listed in the Appendix.

5.1.6. Infotip handlers

Infotips are the bubbles of text that Explorer displays when the mouse is hovered over a file. There are a couple of ways that custom Infotips could be shown for a file. The most common way is to create a registry value named `Infotip` under the `ProgId` key for a file type (56). This value is a string that could either be static text that gets displayed, a reference to a string resource in a DLL, or a set of properties to display – making use of the Windows Property System described below. Here's an example of the Infotip settings for `.exe` files:

```
HKEY_CLASSES_ROOT\SystemFileAssociations\.exe\InfoTip =
"prop:System.FileDescription;System.Company;System.FileVersion;System.DateCreated;System.
Size"
```

In that example, there will be five properties displayed in the Infotip: file description, company name, file version, date created, and the file's size.

Another way to register an Infotip handler is to create and register a COM object that implements the **IQueryInfo** interface. These objects are registered as Infotip handlers by creating a `ShellEx\{00021500-0000-0000-C000-000000000046}` entry for the file type. There are only four Infotip handlers registered this way by default in Windows 7 Professional: for `.contact`, `.group`, `.lnk`, and `.url` files.

5.1.7. COM object persistence and type confusion

At BlackHat Las Vegas in 2010, Mark Dowd, Ryan Smith, and David Dewey presented a paper titled *Attacking Interoperability* (66) where they outlined several types of attacks against web browsers that could also be applicable against the Windows shell. One category of attack has to do with persistence and type confusion in COM objects. Because Windows shell extensions are implemented as COM objects and the shell expects certain types to be returned from various interface functions, type confusion vulnerabilities are possible. Shell extension handlers that read serialized COM objects from a file and return data without checking types could provide an interesting exploit opportunity.

5.1.8. Fuzzing shell extensions

Because the interface used by Windows Explorer to use shell extensions is well documented, it's relatively easy to write a fuzzer for them without having to rely on Explorer itself. A very simple fuzzer could use a collection of valid files and randomly modify them before making use of the **IExtractIcon**, **IExtractImage**, or **IThumbnailProvider** interface to load and process them. Of course, more sophisticated fuzzing techniques using code coverage analysis would allow for more thorough testing.

5.1.9. Exploiting shell extensions

Vulnerability exploit mitigation technologies such as ASLR (Address Space Layout Randomization) and DEP (Data Execution Prevention) can make successful exploitation of Windows shell extensions difficult (67). Exploitation techniques like ROP (return oriented programming) (68) can get around the protection

offered by DEP, but can be foiled by ASLR – ROP requires having certain sequences of bytes at known addresses in memory. It could be possible to brute-force the base address of a DLL for using ROP if an icon or thumbnail handler has its own exception handling in the `IExtractIcon::ExtractIcon()`, `IExtractImage::Extract()`, or `IThumbnailProvider::GetThumbnail()` functions. Explorer doesn't wrap calls to those functions (or other functions in the **IExtractIcon**, **IExtractImage**, or **IThumbnailProvider** interfaces) in exception handlers and the process calling them will crash on invalid memory access. When a thumbnail handler has process isolation disabled, this crashes the `explorer.exe` process itself. When the handler runs in an isolated process, Explorer attempts to generate the thumbnail images one at a time. If `dllhost.exe` crashes on the first thumbnail generated, it will not attempt to generate more thumbnails until the user takes action to report the error and close the process. After that, Explorer will attempt to generate the next thumbnail with a new `dllhost.exe` process, which will again crash.

Besides brute force techniques, ASLR can be defeated when a DLL file that hasn't been built with the `/DYNAMICBASE` option is loaded into memory at a fixed memory address (69). This means that if a 3rd party thumbnail handler is compiled without ASLR support, exploitation with ROP may be possible. It could also be possible to exploit a shell extension with ASLR support by causing a separate shell extension without ASLR support to be loaded into process memory at a fixed address. For example, if a thumbnail handler for `.aaa` files is built with ASLR support and the `.bbb` thumbnail handler is not, a vulnerability the `.aaa` thumbnail handler could be exploited by having `.bbb` files in the same folder. Exploitation will only be effective if the libraries are loaded in the same process – both in `explorer.exe` or both in the COM Surrogate `dllhost.exe`.

Having thumbnail handlers run in an isolated process is good for system stability because it means a bug in a thumbnailer only crashes `dllhost.exe`. It can also be good for security since `dllhost.exe` will have fewer DLLs loaded into memory than `explorer.exe`, with less of a chance that one could be found using a fixed base address. Having a shell extension run in an isolated process with low integrity – as many preview handlers do – is even better for security. In that case, even if execution of arbitrary code is achieved, the attacker would still need to find a way around the limited system access allowed by the low integrity level (70).

The vulnerability exploited by Stuxnet was able to work around the DEP and ASLR features of Windows 7 because it wasn't a memory corruption issue and Stuxnet didn't have to rely on subverting execution to run shellcode in memory – it was able to cause Explorer to load an arbitrary DLL file, resulting in code at the DLL's entry point executing.

It should be noted that while these investigations into shell extensions were intended to be exploited from a removable storage device, they could also be potentially exploitable remotely as well. Remote vectors include malicious email attachments and files placed on network shares.

5.2. Property system

The Windows Property System was implemented in Windows Vista as an extensible system for reading and storing metadata in files (71). One purpose for this was to allow the search indexer to index files

based on non-file-system properties, such as the author of an email or artist of an MP3 file. Since this metadata is stored in the file itself, instead of in the system registry or using special features of the file system, property handlers could be a source of security vulnerabilities.

Like preview handlers, property handlers are COM objects, but instead of being registered with the file extension in the registry in `HKEY_CLASSES_ROOT`, they're registered in `HKEY_CURRENT_MACHINE\Software\Microsoft\Windows\CurrentVersion\PropertySystem\PropertyHandlers`, where there's a subkey for each registered file extension. The default value of the key is the CLSID of the handler. Property handlers implement the **IPropertyStore** interface that Windows uses to enumerate, extract, and store properties.

These property handlers can be run in-process by Windows Explorer when examining the properties of a file through the context menu, viewing a folder in the Details mode, hovering over a file to view the Infotip, or selecting a file to view the information pane. An example of this is when Windows determines that a folder contains music files and the Details view will be able to show the author and song name in columns instead of just the properties of the file on disk. In Windows XP and earlier operating systems, this functionality was handled by "column handlers" implementing the **IColumnProvider** interface.

Property handlers are also used by Windows Search. By default they run in a low-privilege isolated process, `SearchFilterHost.exe`. It's possible to force a property handler to run inside of the search protocol host (`SearchProtocolHost.exe`) by setting the DWORD value **DisableProcessIsolation** to 1 in the CLSID registry entry for the property handler.

Since property handlers parse potentially untrusted files and they can be executed without user interaction, security vulnerabilities in them could be quite serious. The search indexer doesn't index files on removable storage by default, but someone could inadvertently place malicious files into an indexed directory by copying a folder from removable storage into their home directory on the PC. The search indexer could also process an email attachment in a person's inbox, allowing a security vulnerability to be exploited without the user actually opening the attachment.

One documented case of a search indexer compromising a machine by indexing a malicious file happened in 2005 – a researcher at F-Secure noticed that simply downloading a malicious WMF file with `wget` resulted in their machine being infected (72). The culprit was Google Desktop Search attempting to index image metadata for the file using Windows APIs and triggering the exploit. This type of thing is one of the major benefits of having Windows Search make use of low integrity processes and process isolation.

5.3. Folder customization

It's possible to customize the look of folders in Explorer through a number of techniques. The simplest is creating a `desktop.ini` file. This file is parsed whenever a folder is opened in Explorer and can control certain aspects of how the folder is displayed – for example, setting the icon of the folder or selecting which columns are displayed in the Details view. There have been vulnerabilities associated with this file in the past – a buffer overflow in `explorer.exe` when parsing the file (73), and a

vulnerability that allowed activating arbitrary COM objects (74). In 2005, Andrés Tarascó Acuña documented that certain fields of `desktop.ini` can contain UNC paths (75), meaning you can cause a machine to attempt to connect out to an arbitrary system when a folder is viewed in Explorer. This feature still exists in Windows 7.

5.3.1. Shell namespace extensions

Shell namespace extensions allow developers to provide an Explorer interface to data that isn't stored as files and folders on a file system. For example, the ZIP and CAB file viewers (`zipfldr.dll` and `cabview.dll`) allow the user to browse inside of a ZIP or CAB file as if they were actual folders and files on the file system. However, most namespace extensions included with Windows (such as the Games folder, Recycle Bin, the Fusion Cache viewer, etc) don't operate on a specific file or folder.

Namespace extensions are implemented as COM objects that run in-process in the Windows Explorer. When a developer wants a folder to be displayed by a namespace extension, they can either configure an entry in the registry – a virtual folder junction, or use the file system itself by using a special folder name or a `desktop.ini` file (76).

When Explorer encounters a folder with a name that ends in `.{`*`CLSID`*`}`, where *`CLSID`* is a class identifier of a shell namespace extension, it will load that namespace extension to display the contents of the folder. Another possibility is creating a `desktop.ini` file within the folder that contains a section named `[.ShellClassInfo]` with a CLSID value. In either case, the folder itself needs to have the `FILE_ATTRIBUTE_SYSTEM` attribute set. The `desktop.ini` file should have `FILE_ATTRIBUTE_HIDDEN` and `FILE_ATTRIBUTE_SYSTEM` set.

The core interface of a namespace extension is the **IShellFolder** implementation – this provides access to the items in the virtual folder (77). The important functions, the ones that generally lead to parsing of data, are:

- **EnumObjects**: returns an enumerator object used to list all items in the folder
- **CreateViewObject**: returns an IShellView object to manage the folder view
- **GetUIObjectOf**: returns extra information about UI objects such as icons.
- **GetDisplayNameOf**: returns a displayable name for an object
- **GetAttributesOf**: returns attributes for an object

Even though it's possible to create a folder pointing to a namespace extension on a removable storage device, most of the default extensions in Windows operate on data already on the system. For example, a Recycle Bin folder can be created on a USB flash drive, but when a user opens it, they see their own recycle bin and not the contents of whatever is inside of the Recycle Bin folder on the USB drive. This fact makes shell namespace extensions of very limited use for exploitation, unless one can be found that parses data within its own folder.

6. USB operation on GNU/Linux

USB has been supported on Linux since the 2.2.7 kernel (78) released in 1999. The research for this paper was based on the 32 bit version of Ubuntu Desktop 10.10 (Maverick Meerkat) running kernel 2.6.35. The information presented here may not apply to earlier or later versions of Ubuntu, or other distributions of GNU/Linux.

6.1. Core

The core of the USB driver suite, known as **usbcore**, is located in the `drivers/usb/core` directory of the Linux kernel source tree. There is a host controller driver framework (`drivers/usb/core/hcd.c`) that contains most of the code for communicating with host controllers at a high level. This is analogous to the Windows USB port driver (`usbport.sys`). Communication with the host controllers at a low level is done with code in the `drivers/usb/host` directory – `usb-uhci.c` is for **UHCI**, `usb-ehci.c` is for **EHCI**, and these are analogous to the USB host controller miniport drivers on Windows. The USB hub driver (`drivers/usb/core/hub.c`) contains the code to drive the USB hub, which contains similar functionality to the Windows driver `usbhub.sys`. The function `usb_new_device()` in `hub.c` is the code that the hub driver calls when a new device is connected. This function calls `usb_enumerate_device()` to enumerate the device, which involves reading the relevant descriptors. The device is then added to the device tree with a call to `device_add()`.

6.2. USB interface drivers

Drivers for USB devices are implemented as kernel modules. The module's initialization function (defined by the macro `module_init()`), will call `usb_register()` or `usb_register_driver()` to register itself with the USB subsystem. The registration function is passed a pointer to a **usb_driver** structure that includes a list of callback functions for various events and a list of USB IDs that the device can control – **usb_driver.id_table**. This table is a list of **usb_device_id** entries that specify the device class, subclass, and protocol, as well as information such as the vendor, product, and interface information. The USB subsystem uses this list of IDs to match a driver to a device, which happens in `usb_match_id()` in `drivers/usb/core/driver.c`.

USB interface drivers communicate with the USB host controller by using structures called URBs – USB Request Blocks, again, just like Windows, although the URB structure in Linux (named **urb**, defined in `include/linux/usb.h`) is completely different.

6.3. USB mass storage class driver

The Linux USB mass storage class driver is in `drivers/usb/storage/usb.c`. The USB subsystem will call the probe routine, `storage_probe()`, whenever it finds a new device in the mass storage class. The probe function allocates and sets up a SCSI host structure, adds the host to the SCSI subsystem, then creates a kernel thread to handle delayed SCSI device scanning. Once the SCSI subsystem is aware of the device, the USB mass storage SCSI emulation code in `drivers/usb/storage/scsiglue.c` and `protocol.c` handle communication between the SCSI subsystem and the USB subsystem. This basically involves taking SCSI SRBs (SCSI Request Blocks)

and converting them to URBs and then sending the URB to the USB hub driver. The SCSI subsystem then presents a disk block device to the system – this is the disk device that shows up in `/dev`, such as `/dev/sdb`. In older GNU/Linux systems, this is the most that would happen when inserting a USB stick – it was up to the user to mount the file system. Some people hacked together scripts using the Linux **hotplug** system (79), but **hotplug** has since been deprecated and replaced with **udev** and **D-bus** on modern GNU/Linux distributions (80).

For more information on how the Linux SCSI subsystem and the Linux file system work, read the "Anatomy of..." series of articles on the Linux kernel by M. Tim Jones that were published on the IBM **developerWorks** website. Specifically, see *Anatomy of the Linux file system* (81) and *Anatomy of the Linux SCSI subsystem* (82). Of course, the Linux kernel source code is also an excellent source of information and some parts of it are well-documented in the code.

6.4. udev, udisks, D-Bus

udev is the user-mode device manager in recent distributions of Linux and is responsible for dynamically creating and removing entries in `/dev` for devices, setting permissions on those entries, and notifying other subsystems that the entries were created. **udev** can be configured with rules that can cause other applications to run when certain devices are detected. It can also publish events to subscribers through **netlink** sockets.

D-Bus is an interprocess communication (IPC) mechanism that recent distributions of Linux use to allow applications to register for system device events.

udisks, previously known as **DeviceKit-disks**, provides a **D-Bus** interface for querying and manipulating storage devices. **udisks** uses the interface provided by **GUdev** (part of **libudev**) to subscribe to **udev** events and re-publish them to **D-Bus** subscribers.

6.5. File systems in Linux

File systems can exist in a number of forms in modern Linux distributions. Historically, file system drivers were implemented as kernel modules located in the `fs/` directory in the kernel source tree. Each of the file system drivers operate at a level between the storage device's low level device drivers and the virtual file system. The system call interface allows user mode code to talk to the virtual file system, which interacts with the individual file system drivers, which in turn interact with the storage device drivers. The list of native kernel file system drivers supported by the running kernel can be found in `/proc/filesystems`.

File system drivers can also now be implemented in user mode by using **FUSE**, Filesystem in Userspace. When a USB flash drive with the NTFS file system is inserted into an Ubuntu 10.10 machine, the **ntfs-3g** FUSE driver is loaded.

The GNOME desktop also includes a user-mode file system capability called **GVFS**, or the GNOME Virtual File System; however, this is not a Linux file system driver in the traditional sense – GVFS virtual file systems can only be accessed through the GVFS library, another interface that uses the GVFS library like GIO, or the GVFS FUSE mount point at `~/.gvfs/` . All of Nautilus's File I/O goes through GIO, so

Nautilus is able to natively use GVFS file systems. This use of GVFS allows Nautilus to browse files and directories over protocols such as DAV, FTP, and SMB.

7. GNOME and Nautilus

GNOME is the default desktop environment for Ubuntu Linux 10.10 and a few other distributions. The **GNOME** file browser application is called **Nautilus** and provides many of the features one would expect from an operating system shell.

7.1. Automatic mounting of storage devices

Instead of relying directly on **udev** rules to mount newly attached file systems, Nautilus makes use of the **GVolumeMonitor** API in GNOME's **GIO** library, which is part of **glib2**. The default implementation of **GVolumeMonitor** on Unbuntu 10.10 is part of **GVFS**, the GNOME Virtual File System, and is called **GGduVolumeMonitor**. It uses the **GduPool** interface from the **libgdu** library that communicates with **udisks** via **dbus** to get notified when new devices are connected to the PC.

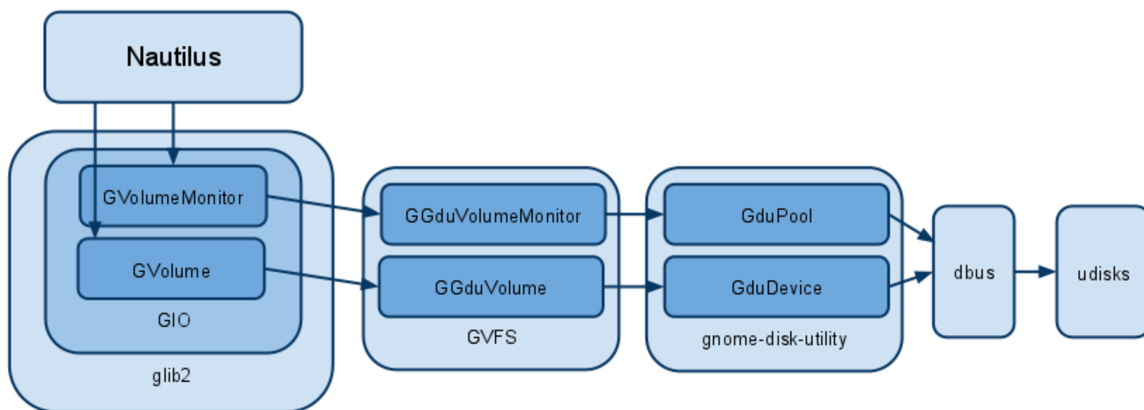

Figure 12 - Nautilus communicates with udisks to monitor and mount devices

Newly connected volumes are mounted through the **GVolume** API, which again is part of **GVFS** and called **GGduVolume** and uses the **GduDevice** interface of the **libgdu** library to mount volumes. **libgdu** uses D-Bus to send a message to **udisks** indicating the disk should be mounted. At this point, **udisks** runs the **mount** program, specifying the arguments "`-t auto`", indicating that **mount** should attempt to determine the file system type, using **libblkid**, or by trying each of the supported file systems listed in `/proc/filesystems` one at a time.

When notification is received that a new disk (or volume) is available, Nautilus checks the user's settings to decide if it should automatically mount the volume or not. This setting is stored in the **gconf** system at `/apps/nautilus/preferences/media_automount`. This setting can be retrieved from the command line using this command:

```
gconftool -g /apps/nautilus/preferences/media_automount
```

In Ubuntu 10.10, this is set to '`true`' by default. Once a file system has been mounted, Nautilus may open a new file browser window open to the root directory of the device. The setting that controls this is `/apps/nautilus/preferences/media_automount_open`, and that setting is also 'true' by default on Ubuntu 10.10.

7.2. Autorun capabilities

Nautilus supports a number of AutoPlay-like capabilities. It can start playing CDs or DVDs when they are inserted or start browsing photos when a device with photos is attached. Much like Windows, the actions are configurable.

Whether or not Nautilus does this for each type of media is controlled by these **gconf** settings in `/apps/nautilus/preferences`:

- **media_autorun_never**: If this boolean value is true, autorun is disabled.
- **media_autorun_x_content_ignore**: This string array value lists which content types that autorun should ignore ("Do Nothing").
- **media_autorun_x_content_open_folder**: This string array value lists which content types autorun should just open the folder for browsing on the dekstop.
- **media_autorun_x_content_start_app**: This value is an array of strings that determine which detected media types will start a specific application to handle them (such as an audio player).

If a media type doesn't appear in any of the **media_autorun_x_content** settings, the user is prompted with a dialog box asking them what to do.

The content type is determined by the **GContentType** API of **GIO** (through the **GMount** interface), specifically the `g_content_type_guess_for_tree()` function. This function makes use of the `/usr/share/mime/treemagic` file to determine the content type for the file system. The format of this file is part of the shared MIME-info database specifications (83) and contains a list of content mime types (`x-content/`) followed by a list of files and attributes that signify that file type. Here are two examples:

```
[50:x-content/audio-dvd]
>"AUDIO_TS/AUDIO_TS.IFO"=file
>"AUDIO_TS/AUDIO_TS.IFO;1"=file
[50:x-content/image-dcf]
>"dcim"=directory,non-empty
```

This specifies that if a volume contains the files "`AUDIO_TS/AUDIO_TS.IFO`" or "`AUDIO_TS/AUDIO_TS.IFO;1`", the content type is `x-content/audio-dvd`, which is a DVD Audio disc. If the volume contains a non-empty directory named "`dcim`", the content type is `x-content/image-dcf`, a digital camera file system.

One particularly interesting content type is `x-content/unix-software`, which is specified when the file system contains a file named `.autorun`, `autorun`, or `autorun.sh`. When one of these files is found, the system will prompt the user to run it or not. The files themselves are shell scripts, and this is analogous to having an `autorun.inf` file on Windows pointing to an executable to run. Luckily for the security of Linux desktop users everywhere, there's no option to automatically run autorun scripts when a device is inserted – this ability is specifically prohibited by the Desktop Application Autostart Specification (84).

7.3. Thumbnailers

GNOME supports the generation of thumbnail images for certain file types. Nautilus makes use of the ability to show thumbnails in the file browser when in the Icon and Details views.

Thumbnails for image files are generated internally in GNOME using the **GdkPixbuf** library to load and scale the image. GNOME relies on external thumbnailer applications to generate thumbnails for other file types, such as movies and documents. The configuration for thumbnailers is stored in **gconf** in `/desktop/gnome/thumbnailers`, and can be listed with the command "`gconftool -R /desktop/gnome/thumbnailers`". The output looks like this (but is much longer):

```
disable_all = false
/desktop/gnome/thumbnailers/audio@mpeg:
 enable = false
 command = /usr/bin/totem-video-thumbnailer -s %s %u %o
/desktop/gnome/thumbnailers/application@x-cb7:
 enable = true
 command = evince-thumbnailer -s %s %u %o
/desktop/gnome/thumbnailers/image@x-gzeps:
 enable = true
```

These settings match a file's MIME type with a program to run, with some substitution done for arguments. `%s` is the size, `%u` is the input file, and `%o` is the output file. The settings also determine if each thumbnailer is enabled or not for that MIME type. The MIME type is derived from the file extension, and the mappings from extension to MIME type are stored in XML files in `/usr/share/mime`. For example, `/usr/share/mime/image/png.xml` contains `<glob pattern="*.png">`, so files that match that pattern are reported as `image/png`.

A full list of default thumbnailers and what extensions they process can be found in the appendix. There are only three thumbnailers configured by default:

- **evince-thumbnailer** for document files
- **totem-video-thumbnailer** for audio and video files
- **gnome-thumbnail-font** for font files

Each of these programs supports a number of different file types, many times relying on 3rd party libraries. If the programs or libraries used contain security vulnerabilities, they could be used to execute arbitrary code when a file is thumbnailed – which happens when a user browses to that directory. This is particularly a problem if a user inserts an untrusted USB flash drive and opens a window to browse the drive. **evince-thumbnailer** is protected by **AppArmor**, which helps to mitigate any possible vulnerabilties, but the other two thumbnailers are not protected.

7.3.1. Exploiting thumbnailers

Exploiting a vulnerability in a thumbnailer on 32 bit Ubuntu Desktop 10.10 is somewhat easier than on 32 bit Windows 7. By default, when a new volume is mounted, Nautilus will open a file browser window to the root directory. This means that when a USB flash drive is inserted, Nautilus will start generating thumbnails for each file in the root directory of the device. This even happens even when the

screensaver is running and locked. Luckily, this feature can be disabled, but the default behavior allows easy exploitation of vulnerable thumbnail handlers. A proof-of-concept exploit was developed that can unlock the screensaver on a locked Ubuntu Desktop machine, giving the attacker full access to the desktop of the logged-on user.

Another aspect of GNOME and Nautilus that make exploitation easier is the fact that external thumbnailers are run as a separate process for each file. This means that having one thumbnailer application crash won't affect the thumbnailing of other files, making it possible to perform brute-force attacks against ASLR.

7.3.1.1. ASLR and NX

Ubuntu has non-executable memory (NX) and ASLR enabled by default; however, it's possible to brute-force addresses of libraries by creating many malicious files. If enough files are present in the root directory of a newly mounted USB device, there's a good chance that a ROP-based exploit can succeed. On my test machine, the kernel had a predilection for mapping `libc` at the higher end of the range of memory up to **0x00FFF000** in the `evince-thumbnailer` process, which is built as PIE (position-independent executable) in Ubuntu 10.10:

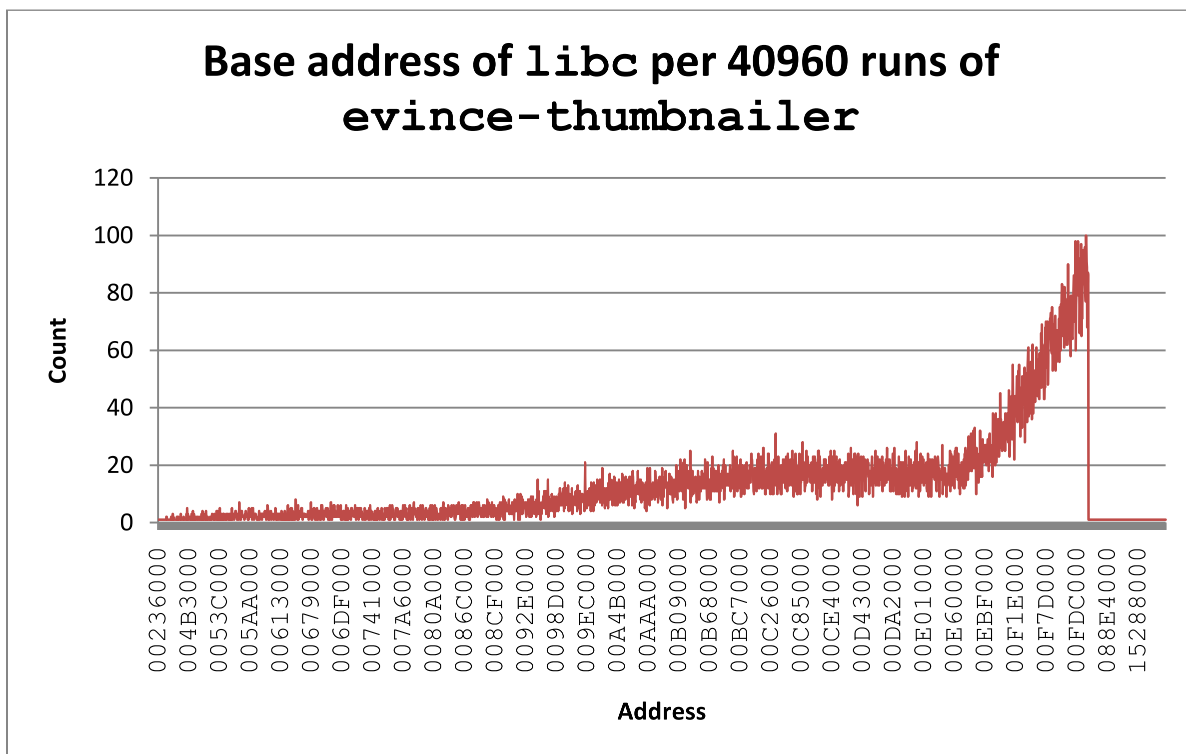

Figure 13 - Distribution of base addresses of libc

Out of 40960 runs, `libc` was loaded at the address **0x00FF8000** 100 times. In smaller runs of 4096, it was likely to hit that address, or any given address of the form **0x00FFX000**, around 10 times. That's 10 times what could be expected with 12 bits of entropy, and this makes it possible to write reliable

thumbnailer exploits that use around 410 malicious files. More research should be done to determine why the selection of addresses in this implementation of ASLR is not distributed evenly in this case.

Implementing effective ROP shellcode is another hurdle to clear for exploit developers. Writing shellcode to execute a file on the USB drive is one possibility, but likely won't work for an application protected with AppArmor.

7.3.1.2. AppArmor

AppArmor is a kernel module that allows an administrator to restrict the abilities of an application to access different aspects of a system. It can be configured to only allow an application to open certain files or prevent an application from launching another process. This is a very effective means of securing an application from exploitation – even if an exploit is successful and an attacker can execute arbitrary code, the shellcode itself is still restricted from what it can do to the system.

AppArmor is configured per-application through profiles, and the protection it offers an application depends on the profile. This means that successfully exploiting an AppArmor-protected application can require finding weaknesses in the profile or in the protection that AppArmor provides. While working on an exploit for a vulnerability that I discovered in `evince-thumbnailer` (85), I had to clear several hurdles to even trigger the vulnerability. For example, the `evince-thumbnailer` AppArmor profile only allows read access to certain files that evince supports displaying: PNG, PDF, DVI, etc. Exploitation of the vulnerability required that `evince-thumbnailer` load an additional malicious file – a font file that had a certain extension not allowed by the AppArmor profile. Creating a symlink with the font file's name that pointed to a file with a `.png` extension allowed `evince-thumbnailer` to load the malicious file, bypassing the AppArmor restriction and triggering the vulnerable code.

Other ways to get around AppArmor are to simply perform activities that aren't restricted by it. For example, there's no way to create an AppArmor profile that prohibits an application from killing a window in the X desktop. So while AppArmor can prevent shellcode from locating the screensaver process by denying access to `/proc` and `ptrace()`, the shellcode could make use of X11 libraries in memory to locate the screensaver window and kill it. Proof-of-concept code was written to do this using `XQueryTree()` to enumerate desktop windows, `XFetchName()` to query the window name and look for the screensaver, and then `XKillClient()` to kill the screensaver process.

8. Conclusion

If you weren't already convinced that USB and removable storage devices are a security threat, or you thought that disabling AutoRun features was an effective security solution, I hope the research presented here changes your mind.

Because there are many features in both Windows and Linux that parse untrusted content present on USB drives, even with minimal or no user interaction, the threat posed by these devices is greater than many people think.

Exploit mitigation technologies developed by OS vendors definitely raise the bar for exploit writers, but they aren't 100% effective.

The only sure way to prevent an intrusion or malware infection from the removable storage vector is to completely disable all removable storage devices that you don't have full physical control over. This means disabling auxiliary USB, FireWire, and eSATA ports if you can't control which devices people will plug into them.

8.1. Acknowledgements

First of all, thanks to all of the people behind the research papers, blog posts, presentations, and books that I've cited throughout this paper. There's a huge body of security research out there, and almost every new idea I come up with has already been done five years ago.

I'd also like to thank everyone who reviewed this paper for their insightful comments and scathing criticisms. Tom Cross, Matthew de Carteret, David Dewey, Joshua Drake, Robert Freeman, Shane Garrett, Darel Griffin, Herb Hintz, John Kuhn, Ben Layer, Gregory Newman, Paul Sabbanal, Natalie Spaeth, Takehiro Takahashi, Chris Valasek, Mark Yason were particularly helpful and/or scathing.

9. Appendix

9.1. USB descriptors for a mass storage class device

This is the USB descriptor output for a mass storage class device, generated by the `lsusb -v` command under Linux:

```
Bus 001 Device 003: ID 0781:5530 SanDisk Corp.
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64
  idVendor           0x0781 SanDisk Corp.
  idProduct          0x5530
  bcdDevice            1.00
  iManufacturer           1 SanDisk
  iProduct                2 Cruzer
  iSerial                 3 20060266120EDEE311F8
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           32
    bNumInterfaces          1
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0x80
      (Bus Powered)
    MaxPower              200mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           2
      bInterfaceClass         8 Mass Storage
      bInterfaceSubClass      6 SCSI
      bInterfaceProtocol     80 Bulk (Zip)
      iInterface              0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0200  1x 512 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
```

```
bEndpointAddress     0x02  EP 2 OUT
bmAttributes            2
  Transfer Type            Bulk
  Synch Type               None
  Usage Type               Data
wMaxPacketSize     0x0200  1x 512 bytes
bInterval               1
```

9.2. Default Shell Extension Handlers in Windows 7 Professional (32 bit)

9.2.1. Icon handlers

Icon Handlers implement the `IExtractIcon` interface.

File Extension	Handler Name	Handler DLL
.appref-ms	Shell Icon Handler for Application References	C:\Windows\system32\dfshim.dll
.library-ms	Library Icon Extract Extension	%SystemRoot%\System32\shdocvw.dll
.lnk	Shortcut	C:\Windows\system32\shell32.dll
.msc	ExtractIcon Class	%SystemRoot%\system32\mmcshext.dll
.pif	Shortcut	C:\Windows\system32\shell32.dll
.scf	CmdFileIcon	%SystemRoot%\system32\shell32.dll
.searchConnector-ms	SearchConnector Icon Extract Extension	%SystemRoot%\System32\shdocvw.dll
.URL	Internet Shortcut	C:\Windows\System32\ieframe.dll

9.2.2. Image handlers

Image handlers implement the `IExtractIcon` interface, but if a thumbnail handler is available for the extension (`IThumbnailProvider`), it's used instead. Many of the thumbnail handlers also have an image handler available, but the ones listed here don't have an associated thumbnail handler interface.

File Extension	Handler Name	Handler DLL
.contact	.contact shell extension handler	%CommonProgramFiles%\System\wab32.dll
.doc	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.dot	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.dwfx		%SystemRoot%\system32\XPSSHHDR.DLL
.easmx		%SystemRoot%\system32\XPSSHHDR.DLL
.edrwx		%SystemRoot%\system32\XPSSHHDR.DLL
.eprtx		%SystemRoot%\system32\XPSSHHDR.DLL
.fon	Microsoft Windows Font IExtractImage Handler	%SystemRoot%\system32\fontext.dll
.fpx	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.jtx		%SystemRoot%\system32\XPSSHHDR.DLL
.lnk	Shortcut	C:\Windows\system32\shell32.dll
.mic	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mix	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll

.mpp	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.obd	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.obt	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.otf	Microsoft Windows Font IExtractImage Handler	%SystemRoot%\system32\fontext.dll
.pfm	Microsoft Windows Font IExtractImage Handler	%SystemRoot%\system32\fontext.dll
.pot	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.ppt	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.ttc	Microsoft Windows Font IExtractImage Handler	%SystemRoot%\system32\fontext.dll
.ttf	Microsoft Windows Font IExtractImage Handler	%SystemRoot%\system32\fontext.dll
.xls	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.xlt	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.xps		%SystemRoot%\system32\XPSSHHDR.DLL

9.2.3. Thumbnail handlers

Thumbnail handlers implement the **IThumbnailProvider** interface. The items in **bold** are configured by default to not run in an isolated process – they run inside of `explorer.exe`. The behavior of items that use the "Property Thumbnail Handler" is described above.

File Extension	Handler Name	Handler DLL
.3g2	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.3gp	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.3gp2	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.3gpp	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.asf	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.avi	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.bmp	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.dib	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.dvr-ms	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.emf	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.gif	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.jfif	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.jpe	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.jpeg	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.jpg	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.library-ms	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m1v	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m2t	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m2ts	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m2v	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m4a	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m4b	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll

.m4p	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.m4v	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mod	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mov	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mp2	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mp2v	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mp3	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mp4	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mp4v	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mpe	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mpeg	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mpg	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mpv2	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.mts	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.png	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.rle	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.searchConnector-ms	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.theme	**Windows Theme Thumbnail Preview**	**%SystemRoot%\system32\themeui.dll**
.tif	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.tiff	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.ts	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.tts	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.vob	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.wav	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.wdp	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.wma	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.wmf	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll
.wmv	Property Thumbnail Handler	%SystemRoot%\system32\shell32.dll
.WTV	WTVFile Thumbnail Handler	C:\Windows\System32\sbe.dll
.wdp	Photo Thumbnail Provider	C:\Windows\system32\PhotoMetadataHandler.dll

9.2.4. Property handlers

Property handlers implement the **IPropertyStore** interface. The items below in **bold** have process isolation disabled.

File Extension	Handler Name	Handler DLL
.3gp	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.3gp2	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.3gpp	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.aac	MF ADTS Property Handler	%SystemRoot%\System32\mf.dll
.adts	MF ADTS Property Handler	%SystemRoot%\System32\mf.dll
.appref-ms	ShellLink for Application References	C:\Windows\system32\dfshim.dll

.asf	MF ASF Property Handler	%SystemRoot%\System32\mf.dll
.avi	MF AVI Property Handler	%SystemRoot%\System32\mf.dll
.bmp	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.contact	**.contact shell extension handler**	**%CommonProgramFiles%\System\wab32.dll**
.cpl		**%SystemRoot%\system32\shell32.dll**
.dib	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.dll		**%SystemRoot%\system32\shell32.dll**
.doc	Office Document Property Handler	%SystemRoot%\system32\propsys.dll
.dot	Office Document Property Handler	%SystemRoot%\system32\propsys.dll
.dvr-ms	MF ASF Property Handler	%SystemRoot%\System32\mf.dll
.dwfx	**Microsoft XPS Shell Metadata Handler**	**%SystemRoot%\system32\XPSSHHDR.DLL**
.easmx	**Microsoft XPS Shell Metadata Handler**	**%SystemRoot%\system32\XPSSHHDR.DLL**
.edrwx	**Microsoft XPS Shell Metadata Handler**	**%SystemRoot%\system32\XPSSHHDR.DLL**
.eml	**Shell Message Handler**	**%SystemRoot%\system32\inetcomm.dll**
.eprtx	**Microsoft XPS Shell Metadata Handler**	**%SystemRoot%\system32\XPSSHHDR.DLL**
.exe		**%SystemRoot%\system32\shell32.dll**
.fon		**%SystemRoot%\system32\shell32.dll**
.gif	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.group	**.group shell extension handler**	**%CommonProgramFiles%\System\wab32.dll**
.ico	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.jfif	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.jpe	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.jpeg	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.jpg	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.jtx	**Microsoft XPS Shell Metadata Handler**	**%SystemRoot%\system32\XPSSHHDR.DLL**
.label	**Property Labels**	**%SystemRoot%\System32\shdocvw.dll**
.library-ms	Library Property Store	%SystemRoot%\system32\shell32.dll
.lnk	**Shortcut**	**C:\Windows\system32\shell32.dll**
.m1v	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.m2t	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.m2ts	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.m2v	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.m4a	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.m4b	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.m4p	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.m4v	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.mod	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.mov	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.mp2	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.mp2v	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.mp3	MF MP3 Property Handler	%SystemRoot%\System32\mf.dll
.mp4	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.mp4v	MF MPEG-4 Property Handler	%SystemRoot%\System32\mf.dll
.mpe	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.mpeg	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.mpg	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.mpv2	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.msg	Office Document Property Handler	%SystemRoot%\system32\propsys.dll

.MSI	OLE DocFile Property Handler	%SystemRoot%\system32\propsys.dll
.MSM	OLE DocFile Property Handler	%SystemRoot%\system32\propsys.dll
.MSP	OLE DocFile Property Handler	%SystemRoot%\system32\propsys.dll
.MST	OLE DocFile Property Handler	%SystemRoot%\system32\propsys.dll
.mts	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.nws	**Shell Message Handler**	**%SystemRoot%\system32\inetcomm.dll**
.ocx		**%SystemRoot%\system32\shell32.dll**
.otf		**%SystemRoot%\system32\shell32.dll**
.PCP	OLE DocFile Property Handler	%SystemRoot%\system32\propsys.dll
.png	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.pot	Office Document Property Handler	%SystemRoot%\system32\propsys.dll
.ppt	Office Document Property Handler	%SystemRoot%\system32\propsys.dll
.rle	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.rll		**%SystemRoot%\system32\shell32.dll**
.search-ms	CLSID_AutoListPropertyStore	%SystemRoot%\System32\shdocvw.dll
.searchConnector-ms	Location Description Property Handler	%SystemRoot%\system32\shell32.dll
.sys		**%SystemRoot%\system32\shell32.dll**
.tif	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.tiff	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.ts	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.ttc		**%SystemRoot%\system32\shell32.dll**
.ttf		**%SystemRoot%\system32\shell32.dll**
.tts	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.url	**Internet Shortcut**	**C:\Windows\System32\ieframe.dll**
.vob	MF MPEG Property Handler	%SystemRoot%\System32\mf.dll
.wav	MF WAV Property Handler	%SystemRoot%\System32\mf.dll
.wdp	IPropertyStore Handler for Images	C:\Windows\system32\PhotoMetadataHandler.dll
.wma	MF ASF Property Handler	%SystemRoot%\System32\mf.dll
.wmv	MF ASF Property Handler	%SystemRoot%\System32\mf.dll
.wtv	**WTVFile Property Handler**	**C:\Windows\System32\sbe.dll**
.xls	Office Document Property Handler	%SystemRoot%\system32\propsys.dll
.xlt	Office Document Property Handler	%SystemRoot%\system32\propsys.dll
.xps	**Microsoft XPS Shell Metadata Handler**	**%SystemRoot%\system32\XPSSHHDR.DLL**

9.2.5. Preview handlers

Preview Handlers implement the **IPreviewHandler** interface. The items in **bold** below have the **DisableLowILProcessIsolation** setting enabled, which means they run in the same security context as the logged-on user.

File Extension	Handler Name	Handler DLL
.3g2	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.3gp	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.3gp2	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.3gpp	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.AAC	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.ADT	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**

.ADTS	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.aif	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.aifc	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.aiff	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.asf	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.asm	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.asmx	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.aspx	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.au	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.avi	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.bat	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.c	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.cmd	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.contact	**CLSID_ContactReadingPane**	**%CommonProgramFiles%\System\wab32.dll**
.cpp	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.css	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.csv	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.cxx	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.def	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.diz	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.dvr-ms	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.dwfx	**Microsoft XPS Rich Preview Handler**	**%SystemRoot%\System32\xpsrchvw.exe -IPreview**
.easmx	**Microsoft XPS Rich Preview Handler**	**%SystemRoot%\System32\xpsrchvw.exe -IPreview**
.edrwx	**Microsoft XPS Rich Preview Handler**	**%SystemRoot%\System32\xpsrchvw.exe -IPreview**
.eprtx	**Microsoft XPS Rich Preview Handler**	**%SystemRoot%\System32\xpsrchvw.exe -IPreview**
.fon	Windows Font previewer	%SystemRoot%\system32\fontext.dll
.h	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.hpp	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.hta	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.htm	CLSID_PreviewHtml	%SystemRoot%\system32\inetcomm.dll
.html	CLSID_PreviewHtml	%SystemRoot%\system32\inetcomm.dll
.hxx	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.inc	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.ini	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.java	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.jtx	**Microsoft XPS Rich Preview Handler**	**%SystemRoot%\System32\xpsrchvw.exe -IPreview**
.m1v	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.m2t	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.m2ts	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.m2v	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.m4a	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**

.m4v	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mht	CLSID_PreviewMime	%SystemRoot%\system32\inetcomm.dll
.mhtml	CLSID_PreviewMime	%SystemRoot%\system32\inetcomm.dll
.mid	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.midi	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mod	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mp2	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mp2v	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mp3	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mp4	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mp4v	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mpa	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mpe	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mpeg	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mpg	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mpv2	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.mts	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.nvr	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.otf	Windows Font previewer	%SystemRoot%\system32\fontext.dll
.pfm	Windows Font previewer	%SystemRoot%\system32\fontext.dll
.php3	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.pl	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.plg	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.ps1xml	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.reg	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.rmi	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.rtf	Windows RTF Previewer	%SystemRoot%\system32\shell32.dll
.sed	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.shtml	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.snd	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.sql	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.text	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.ts	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.tsv	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.ttc	Windows Font previewer	%SystemRoot%\system32\fontext.dll
.ttf	Windows Font previewer	%SystemRoot%\system32\fontext.dll
.tts	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.txt	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.wav	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.wm	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.wma	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**

.wmv	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.WTV	**Windows Media Player Rich Preview Handler**	**%ProgramFiles%\Windows Media Player\wmprph.exe**
.x	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.xml	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.xps	**Microsoft XPS Rich Preview Handler**	**%SystemRoot%\System32\xpsrchvw.exe -IPreview**
.xsl	Windows TXT Previewer	%SystemRoot%\system32\shell32.dll
.contact	**CLSID_ContactReadingPane**	**%CommonProgramFiles%\System\wab32.dll**
.htm	CLSID_PreviewHtml	%SystemRoot%\system32\inetcomm.dll
.html	CLSID_PreviewHtml	%SystemRoot%\system32\inetcomm.dll
.mht	CLSID_PreviewMime	%SystemRoot%\system32\inetcomm.dll
.mhtml	CLSID_PreviewMime	%SystemRoot%\system32\inetcomm.dll
.msg	**MAPI Mail Previewer**	**%SystemRoot%\system32\mssvp.dll**
.rtf	Windows RTF Previewer	%SystemRoot%\system32\shell32.dll

9.3. Default GNOME Desktop thumbnailers in Ubuntu Desktop Linux 10.10 (32 bit)

This table lists the default thumbnailer applications in Ubuntu 10.10. By default, `evince-thumbnailer` is protected with an AppArmor profile but the other thumbnailers are not.

File Extensions	Mime Type	File Description	Thumbnailer
.anim[1-9j]	video/x-anim	ANIM animation	totem-video-thumbnailer
.mp4 .m4v	video/mp4	MPEG-4 video	totem-video-thumbnailer
.m2t .m2ts .ts .mts .cpi .clpi .mpl .mpls .bdm .bdmv	video/mp2t	MPEG-2 transport stream	totem-video-thumbnailer
.asf	video/x-ms-asf	ASF video	totem-video-thumbnailer
.ogx	application/ogg	Ogg multimedia file	totem-video-thumbnailer
.shn	application/x-shorten	Shorten audio	totem-video-thumbnailer
.mxf	application/mxf	MXF video	totem-video-thumbnailer
.gvp	text/x-google-video-pointer	Google Video Pointer	totem-video-thumbnailer
.avi .divx	video/x-msvideo	AVI video	totem-video-thumbnailer
.qt .mov .moov .qtvr	video/quicktime	QuickTime video	totem-video-thumbnailer
.wmv	video/x-ms-wmv	Windows Media video	totem-video-thumbnailer
.webm	video/webm	WebM video	totem-video-thumbnailer
.wmx	video/x-ms-wmx		totem-video-thumbnailer
.ra .rm .ram	audio/x-pn-realaudio		totem-video-thumbnailer
.ogv	video/ogg	Ogg Video	totem-video-thumbnailer

.ram	application/ram	RealMedia Metafile	totem-video-thumbnailer
.mpeg .mpg .mp2 .mpe .vob	video/mpeg	MPEG video	totem-video-thumbnailer
.dv	video/dv	DV video	totem-video-thumbnailer
.mkv	video/x-matroska	Matroska video	totem-video-thumbnailer
.wpl	application/vnd.ms-wpl	WPL playlist	totem-video-thumbnailer
.fli	video/fli		totem-video-thumbnailer
.rm .rmj .rmm .rms .rmx .rmvb	application/vnd.rn-realmedia	RealMedia document	totem-video-thumbnailer
.wvx	video/x-ms-wvx		totem-video-thumbnailer
.rv .rvx	video/vnd.rn-realvideo	RealVideo document	totem-video-thumbnailer
.rp	image/vnd.rn-realpix	RealPix document	totem-video-thumbnailer
.flv	video/x-flv	Flash video	totem-video-thumbnailer
.pict .pict1 .pict2	image/x-pict	Macintosh Quickdraw/PICT drawing	totem-video-thumbnailer
.nsc	application/x-netshow-channel	Windows Media Station file	totem-video-thumbnailer
.fli .flc	video/x-flic	FLIC animation	totem-video-thumbnailer
.wm	video/x-ms-wm		totem-video-thumbnailer
.sdp	application/sdp	SDP multicast stream file	totem-video-thumbnailer
.qtl	application/x-quicktimeplayer		totem-video-thumbnailer
.3gp .3g2 .3gpp .3ga	video/3gpp	3GPP multimedia file	totem-video-thumbnailer
	application/x-matroska	Matroska stream	totem-video-thumbnailer
.nsv	video/x-nsv	NullSoft video	totem-video-thumbnailer
.viv .vivo	video/vivo	Vivo video	totem-video-thumbnailer
.pdf	application/pdf	PDF document	evince-thumbnailer
.djvu .djv	image/vnd.djvu	DjVu image	evince-thumbnailer
.pdf.bz2	application/x-bzpdf	PDF document (bzip-compressed)	evince-thumbnailer
.cbr	application/x-cbr	comic book archive	evince-thumbnailer
.cbz	application/x-cbz	comic book archive	evince-thumbnailer
.cbt	application/x-cbt	comic book archive	evince-thumbnailer
.dvi	application/x-dvi	TeX DVI document	evince-thumbnailer
.pdf.gz	application/x-gzpdf	PDF document (gzip-compressed)	evince-thumbnailer
.ps.bz2	application/x-bzpostscript	PostScript document (bzip-compressed)	evince-thumbnailer
.ps	application/postscript	PS document	evince-thumbnailer
.ps.gz	application/x-gzpostscript	PostScript document (gzip-compressed)	evince-thumbnailer
.eps.bz2 .epsi.bz2 .epsf.bz2	image/x-bzeps	EPS image (bzip-compressed)	evince-thumbnailer

.eps .epsi .epsf	image/x-eps	EPS image	evince-thumbnailer
.dvi.gz	application/x-gzdvi	TeX DVI document (gzip-compressed)	evince-thumbnailer
.dvi.bz2	application/x-bzdvi	TeX DVI document (bzip-compressed)	evince-thumbnailer
.eps.gz .epsi.gz .epsf.gz	image/x-gzeps	EPS image (gzip-compressed)	evince-thumbnailer
.cb7	application/x-cb7	comic book archive	evince-thumbnailer
.ttf .ttc	application/x-font-ttf	TrueType font	gnome-thumbnail-font
.otf	application/x-font-otf	OpenType font	gnome-thumbnail-font
.pfa .pfb .gsf	application/x-font-type1	Postscript type-1 font	gnome-thumbnail-font
.pcf .pcf.Z .pcf.gz	application/x-font-pcf	PCF font	gnome-thumbnail-font

10. Works cited

1. **O'Connor, Mary and Nelson, Tom.** Protect Your PC From Viruses. *Smart Computing.* 2002, Vol. 6, 7.

2. **Slade, Robert M.** *History of Computer Viruses.* 1992.

3. Virdem - VirusInfo. *The Virus Encyclopedia.* [Online] [Cited: Nov 27, 2010.] http://virus.wikia.com/wiki/Virdem.

4. **F-Secure.** F-Secure Virus Descriptions : Boza.A. *F-Secure.* [Online] [Cited: Nov 27, 2010.] http://www.f-secure.com/v-descs/boza.shtml.

5. —. F-Secure Virus Descriptions: Roron. *F-Secure.* [Online] F-Secure, November 6, 2002. [Cited: November 28, 2010.] http://www.f-secure.com/v-descs/roro.shtml.

6. **Turkulainen, Jarkko and Tocheva, Katrin.** F-Secure Virus Descriptions : Bacros.A. *F-Secure.* [Online] F-Secure, October 13, 2004. [Cited: November 28, 2010.] http://www.f-secure.com/v-descs/bacros_a.shtml.

7. **Microsoft.** How to disable the Autorun functionality in Windows. *Microsoft Support.* [Online] September 9, 2010. [Cited: November 30, 2010.]

8. **Dewey, David and Barrall, Darrin.** Plug and Root: The USB Key to the Kingdom. *BlackHat.* [Online] 2005. http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Barrall-Dewey.pdf.

9. 'Silly' worm targets USB sticks. *v3.co.uk.* [Online] v3.co.uk, May 4, 2007. [Cited: November 28, 2010.] http://www.v3.co.uk/vnunet/news/2189228/virus-targets-usb-sticks.

10. **Shachtman, Noah.** Under Worm Assault, Military Bans Disks, USB Drives. *Wired Magazine.* [Online] November 19, 2008. [Cited: November 29, 2010.] http://www.wired.com/dangerroom/2008/11/army-bans-usb-d/.

11. **Microsoft.** How to Test Autorun.inf Files. *Microsoft Support.* [Online] May 11, 2007. [Cited: November 29, 2010.] http://support.microsoft.com/kb/136214.

12. —. Using Hardware AutoPlay. *MSDN Library.* [Online] November 9, 2009. [Cited: November 29, 2010.]

13. **Cohen, Arik.** Improvements to AutoPlay. *Engineering Windows 7.* [Online] April 27, 2009. http://blogs.msdn.com/b/e7/archive/2009/04/27/improvements-to-autoplay.aspx.

14. **Falliere, Nicolas, Murchu, Liam O and Chien, Eric.** W32.Stuxnet Dossier. *Symantec.* [Online] http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

15. **Ferrie, Peter.** The Missing LNK. *Microsoft.* [Online] http://pferrie2.tripod.com/papers/lnk.pdf.

16. **Halderman, J. Alex, et al.** Lest We Remember: Cold Boot Attacks on Encryption Keys. [Online] 2008. http://citp.princeton.edu/pub/coldboot.pdf.

17. **Dornseif, Maximillian.** 0wned by and iPod. [Online] 2004. http://md.hudora.de/presentations/firewire/PacSec2004.pdf.

18. **Boileau, Adam.** Hit by a Bus: Physical Access Attacks with Firewire. [Online] 2006. http://www.storm.net.nz/static/files/ab_firewire_rux2k6-final.pdf.

19. **USB Implementers Forum.** USB.org Documents. *USB.org.* [Online] http://www.usb.org/developers/docs/.

20. **Wikipedia.** Universal Serial Bus. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Universal_Serial_Bus.

21. **Peacock, Craig.** USB in a NutShell. *Beyond Logic.* [Online] http://www.beyondlogic.org/usbnutshell/usb1.shtml.

22. **Intel.** Enhanced Host Controller Interface Specification for Universial Serial Bus. *Intel.* [Online] http://www.intel.com/technology/usb/download/ehci-r10.pdf.

23. **Compaq.** OpenHCI: Open Host Controller Interface Specification for USB. *Compaq.* [Online] ftp://ftp.compaq.com/pub/supportinformation/papers/hcir1_0a.pdf.

24. **Intel.** Universal Host Controller Interface (UHCI) Design guide. *Intel.* [Online] http://download.intel.com/technology/usb/UHCI11D.pdf.

25. **USB-IF.** USB Class Codes. *USB.org.* [Online] November 17, 2009. [Cited: November 30, 2010.] http://www.usb.org/developers/defined_class.

26. **Axelson, Jan.** The Mass Storage Page. *Jan Axelson's Lakeview Research.* [Online] http://www.lvr.com/mass_storage.htm.

27. **USB Implementers Forum.** Universal Serial Bus Mass Storage Class Bulk Only Transport. *USB Implementers Forum.* [Online] 1999. http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf.

28. **INCITS.** SCSI Standards Architecture. *T10 Technical Committee.* [Online] http://www.t10.org/scsi-3.htm.

29. **Axelson, Jan.** *USB Mass Storage.* s.l. : Lakeview Research, 2006.

30. **MWR InfoSecurity.** Linux USB Device Driver - Buffer Overflow. *MWR InfoSecurity.* [Online] http://labs.mwrinfosecurity.com/files/Advisories/mwri_linux-usb-buffer-overflow_2009-10-29.pdf.

31. PlayStation Jailbreak. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/PlayStation_Jailbreak.

32. **phire.** PSJailbreak Exploit Reverse Engineering. *PS3 Wiki.* [Online] https://ps3wiki.lan.st/index.php?title=PSJailbreak_Exploit_Reverse_Engineering.

33. **Peermohamed, Fizalkhan.** What is the right way to read and parse configuration descriptors? *Microsoft Windows USB Core Team Blog.* [Online] http://blogs.msdn.com/b/usbcoreblog/archive/2009/12/12/what-is-the-right-way-to-validate-and-parse-configuration-descriptors.aspx.

34. Social-Engineer Toolkit v0.6.1 Teensy USB HID Attack Vector. *SecManiac.* [Online] http://www.secmaniac.com/august-2010/social-engineer-toolkit-v0-6-1-teensy-usb-hid-attack-vector/.

35. **Jodeit, Moritz.** Evaluating Security Aspects of the Universial Serial Bus. [Online] 2009. http://www.informatik.uni-hamburg.de/SVS/archiv/slides/09-01-13-OS-Jodeit-Evaluating_Security_Aspects_of_USB.pdf.

36. **Mueller, Tobias.** Virtualized USB Fuzzing: Breaking USB for Fun and Profit. [Online] 2010. https://muelli.cryptobitch.de/paper/2010-usb-fuzzing.pdf.

37. **Microsoft.** DSF Device Simulation Framework. *Windows Hardware Developer Central.* [Online] http://www.microsoft.com/whdc/devtools/dsf.mspx.

38. **Miller, Charlie.** Fuzzing With Code Coverage By Example. [Online] 2007. http://toorcon.org/2007/talks/34/code_coverage_by_example.pdf.

39. **Microsoft.** USB Driver Stack for Windows XP and Later. *Windows Driver Kit.* [Online] October 26, 2010. [Cited: November 30, 2010.] http://msdn.microsoft.com/en-us/library/ff539311(v=VS.85).aspx.

40. —. Writing an ISR. *Windows Driver Kit.* [Online] November 21, 2010. [Cited: December 2, 2010.] http://msdn.microsoft.com/en-us/library/ff566399(VS.85).aspx.

41. —. USB Generic Parent Driver (Usbccgp.sys). *MSDN Library.* [Online] October 26, 2010. [Cited: December 2, 2010.] http://msdn.microsoft.com/en-us/library/ff539234(VS.85).aspx.

42. —. Drivers for the Supported USB Device Classes. *MSDN Library.* [Online] October 26, 2010. [Cited: December 2, 2010.] http://msdn.microsoft.com/en-us/library/ff538820(v=VS.85).aspx.

43. **Borve, Martin.** How does USB stack enumerate a device? *Microsoft Windows USB Core Team Blog.* [Online] October 30, 2009. [Cited: December 2, 2010.] http://blogs.msdn.com/b/usbcoreblog/archive/2009/10/31/how-does-usb-stack-enumerate-a-device.aspx.

44. **Peacock, Craig.** USB Descriptors. *USB in a NutShell.* [Online] September 17, 2010. [Cited: December 3, 2010.] http://www.beyondlogic.org/usbnutshell/usb5.shtml.

45. **Microsoft.** Adding a PnP Device to a Running System. *MSDN Library.* [Online] September 21, 2010. [Cited: December 3, 2010.] http://msdn.microsoft.com/en-us/library/ff540535(VS.85).aspx.

46. —. Standard USB Identifiers. *MSDN Library.* [Online] November 16, 2010. [Cited: December 2, 2010.] http://msdn.microsoft.com/en-us/library/ff553356(v=VS.85).aspx.

47. —. Storage Technologies. *Windows Hardware Developer Central.* [Online] http://www.microsoft.com/whdc/device/storage/default.mspx.

48. —. Storage Devices. *MSDN Library.* [Online] http://msdn.microsoft.com/en-us/library/ff563893(v=VS.85).aspx.

49. **OSTA.** OSTA Universal Disk Format Specifications. [Online] http://www.osta.org/specs/.

50. File Allocation Table. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/File_Allocation_Table.

51. **Microsoft.** Microsoft Extensible Firmware Initiative FAT32 File System Specification. *Windows Hardware Developer Central.* [Online] http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx.

52. ExFAT. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/ExFAT.

53. **Brantén, Bo.** Windows Driver Examples. [Online] http://www.acc.umu.se/~bosse/.

54. **Beck, Peter.** Safe Unlinking in the Kernel Pool. *Microsoft Security Research & Defense Blog.* [Online] http://blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx.

55. Downloads: DeviceTree. *OSR Online.* [Online] http://www.osronline.com/article.cfm?article=97.

56. **Microsoft.** Creating Shell Extension Handlers. *MSDN Library.* [Online] November 9, 2010. [Cited: December 10, 2010.] http://msdn.microsoft.com/en-us/library/cc144067(VS.85).aspx.

57. **Stevens, Didier.** Quickpost: /JBIG2Decode Trigger Trio. *Didier Stevens.* [Online] http://blog.didierstevens.com/2009/03/04/quickpost-jbig2decode-trigger-trio/.

58. **Hao, Xu and Joseph, Moti.** A Story about How Hackers' Heart Broken by 0-day. *POC2010.* [Online] http://www.exploit-db.com/download_pdf/15899.

59. **Microsoft.** Perceived Types. *MSDN Library.* [Online] http://msdn.microsoft.com/en-us/library/cc144150(v=vs.85).aspx.

60. —. Application Registration. *MSDN Library.* [Online] http://msdn.microsoft.com/en-us/library/ee872121(v=vs.85).aspx.

61. —. Creating Icon Handlers. *MSDN Library.* [Online] [Cited: December 10, 2010.] http://msdn.microsoft.com/en-us/library/cc144122(v=VS.85).aspx.

62. —. File Types. *MSDN Library.* [Online] [Cited: December 10, 2010.] http://msdn.microsoft.com/en-us/library/cc144148(v=VS.85).aspx.

63. —. IThumbnailProvider Interface. *MSDN Library.* [Online] [Cited: December 10, 2010.] http://msdn.microsoft.com/en-us/library/bb774614(v=VS.85).aspx.

64. —. Thumbnail Handlers. *MSDN Library.* [Online] [Cited: December 13, 2010.] http://msdn.microsoft.com/en-us/library/cc144118(v=VS.85).aspx.

65. —. Preview Handlers and Shell Preview Host. *MSDN Library.* [Online] [Cited: December 10, 2010.] http://msdn.microsoft.com/en-us/library/cc144143(VS.85).aspx.

66. **Dowd, Mark, Smith, Ryan and Dewey, David.** Attacking Interoperability. [Online] http://www.hustlelabs.com/stuff/bh2009_dowd_smith_dewey.pdf.

67. **Miller, Matt.** On the effectiveness of DEP and ASLR. *Microsoft Security Research & Defense.* [Online] http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx.

68. **Shacham, Hovav.** The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). [Online] 2007. http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf.

69. **Microsoft.** Windows ISV Software Security Defenses. *MSDN Library.* [Online] http://msdn.microsoft.com/en-us/library/bb430720.aspx.

70. —. Windows Vista Integrity Mechanism Technical Reference. *MSDN Library.* [Online] http://msdn.microsoft.com/en-us/library/bb625964.aspx.

71. —. Understanding Property Handlers. *MSDN Library.* [Online] [Cited: December 13, 2010.] http://msdn.microsoft.com/en-us/library/cc144129(v=VS.85).aspx.

72. **F-Secure.** Be careful with WMF files. *F-Secure Weblog.* [Online] December 28, 2005. http://www.f-secure.com/weblog/archives/00000753.html.

73. **Microsoft.** Unchecked Buffer in Windows Shell Could Enable System Compromise. *Microsoft Security Bulletins.* [Online] http://www.microsoft.com/technet/security/bulletin/ms03-027.mspx.

74. **Secunia.** Microsoft Windows "desktop.ini" Arbitrary File Execution Vulnerability. *McAfee.* [Online] Secunia. http://secunia.com/advisories/11633/.

75. **Acuña, Andrés Tarascó.** Exploiting Win32 Design Flaws. [Online] http://www.tarasco.org/security/Process_Injector/Win32.Design.Flaws.pdf.

76. **Microsoft.** Specifying a Namespace Extension's Location. *MSDN Library.* [Online] [Cited: 13 2010, December.] http://msdn.microsoft.com/en-us/library/cc144096(v=VS.85).aspx.

77. —. Implementing the Basic Folder Object Interfaces. *MSDN Library.* [Online] [Cited: December 13, 2010.] http://msdn.microsoft.com/en-us/library/cc144093(v=VS.85).aspx#PIDL.

78. **Linux USB Project.** Linux USB Project. [Online] http://www.linux-usb.org/.

79. **Riabitsev, Konstantin.** USB storage hotplug code. *Redhat Developers Mailing List.* [Online] https://listman.redhat.com/archives/rhl-devel-list/2003-August/msg00115.html.

80. **NTLUG.** Hotplugging, Udev, HAL and D-BUS. *NTLUG.* [Online] http://www.ntlug.org/Articles/Hotplug.

81. **Jones, M. Tim.** Anatomy of the Linux file system. *IBM developerWorks.* [Online] http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/.

82. —. Anatomy of the Linux SCSI subsystem. *IBM developerWorks.* [Online] http://www.ibm.com/developerworks/linux/library/l-scsi-subsystem/.

83. **Leonard, Thomas.** Shared MIME-info Database Specification. *X Desktop Group.* [Online] http://standards.freedesktop.org/shared-mime-info-spec/shared-mime-info-spec-latest.html.

84. **John Palmieri, et. al.** Desktop Application Autostart Specification. *X Desktop Group.* [Online] http://standards.freedesktop.org/autostart-spec/autostart-spec-latest.html.

85. **Canonical, Ltd.** USN-1035-1: Evince vulnerabilities. *Ubuntu.* [Online] http://www.ubuntu.com/usn/usn-1035-1.

86. **NirSoft.** ShellExView. *NirSoft.* [Online] [Cited: December 10, 2010.] http://www.nirsoft.net/utils/shexview.html.

87. TeensyUSB Development Board. *PJRC.* [Online] http://www.pjrc.com/teensy/.

88. **Nadel, Brian.** Kanguru Fire Flash Review. *CNET Reviews.* [Online] July 25, 2005. http://reviews.cnet.com/flash-memory-cards/kanguru-fire-flash-2gb/4505-3241_7-31229811.html.

89. **Lake, Bill.** Supercharge Your Flash Drive With eSATA. *Tom's Guide.* [Online] February 6, 2009. http://www.tomsguide.com/us/eSATA-USB-Flash,review-1195.html.

90. **Farino, Dan.** Remotely Unlock a Windows Workstation. *CodeProject.* [Online] November 2, 2006. http://www.codeproject.com/KB/system/RemoteUnlock.aspx.

91. **Microsoft.** Microsoft Security Advisory (2490606). *Microsoft Technet Security Advisories.* [Online] http://www.microsoft.com/technet/security/advisory/2490606.mspx.

11. Legal notices

- Microsoft, Windows, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
- Google is a registered trademark of Google, Inc.
- Ubuntu is a registered trademark of Canonical Ltd.
- GNOME is a registered trademark of GNOME Foundation in the U.S. and other countries.
- Linux is the registered trademark of Linux Torvalds in the U.S. and other countries.
- Apple is a registered trademark of Apple, Inc in the U.S. and other countries.
- Symantec is a registered trademark of Symantec Corporation or its affiliates in the U.S. and other countries.
- SanDisk and Cruzer are trademarks of SanDisk Corporation, registered in the United States and other countries.
- Other names may be trademarks of their respective owners.