Malware Analysis

Kernel Debugging

COMPILED BY: DHARMESH DAVE | ASST. PROF. | NATIONAL FORENSIC SCIENCES UNIVERSITY

Drivers and Kernel Code

- Before we begin debugging malicious kernel code, you need to understand **how kernel code works**, why malware writers use it, and some of the unique challenges it presents.
- Drivers are difficult to analyze because they **load into memory, stay resident**, and respond to requests from applications. This is further complicated because applications do not directly interact with kernel drivers.
- Instead, they access **device objects**, which send requests to particular devices. Devices are not necessarily physical hardware components; the driver creates and destroys devices, which can be accessed from user space.

Drivers and Kernel Code

- For example, consider a USB flash drive. A driver on the system handles USB flash drives, but an application does not make requests directly to that driver; it makes **requests to a specific device object instead**.
- When the user plugs the USB flash drive into the computer, Windows creates the **"F: drive" device object** for that drive. An application can now make requests to the F: drive, which ultimately will be sent to the driver for USB flash drives.
- The same driver might handle requests for a second USB flash drive, but applications would access it through a different device object such as the G: drive.

Drivers and Kernel Code

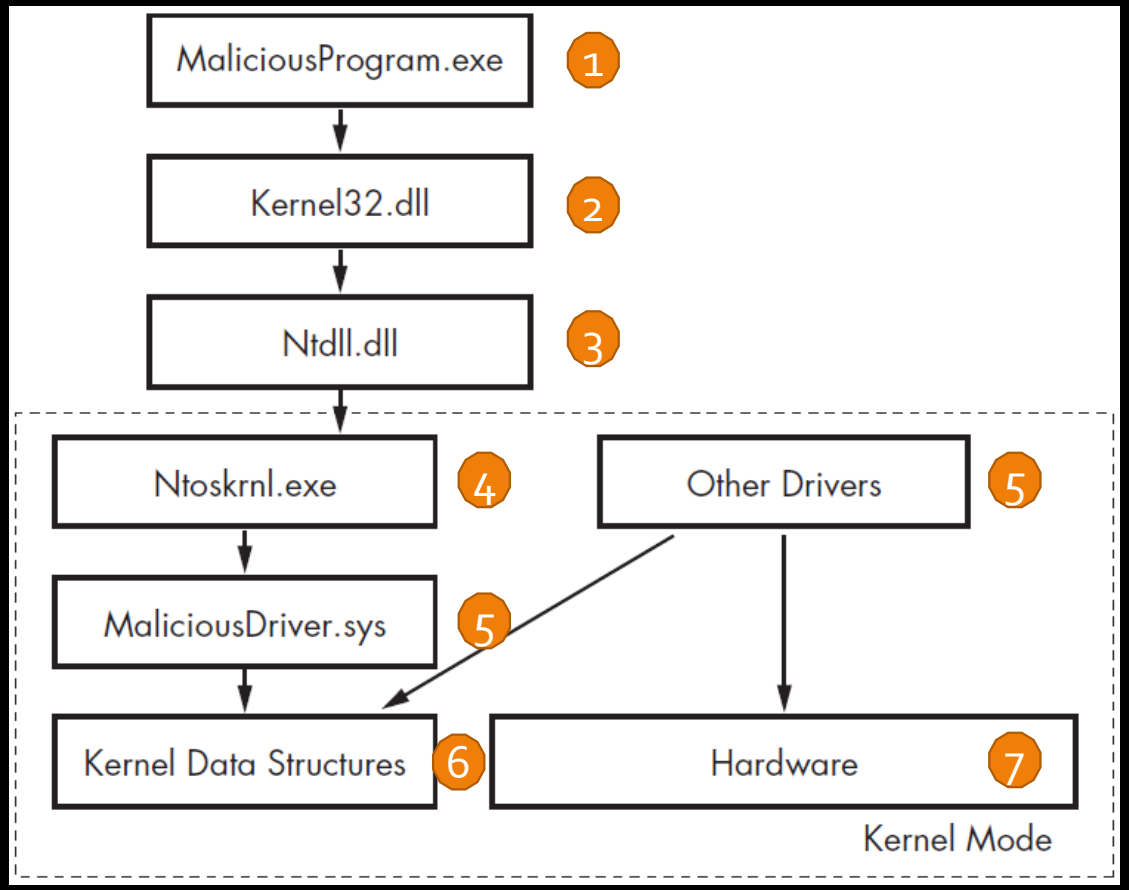
- In order for this system to work properly, **drivers must be loaded into the kernel**, just as DLLs are loaded into processes. When a driver is first loaded, **its DriverEntry procedure is called**, similar to DLLMain for DLLs.
- Unlike DLLs, which expose functionality through the export table, drivers must register the address for callback functions, which will be called when a user-space software component requests a service.
- The registration happens in the DriverEntry routine. Windows creates a driver object structure, which is passed to the DriverEntry routine.
- The DriverEntry routine then creates a **device that can be accessed from user space**, and the user-space application interacts with the driver by sending requests to that device.

Drivers and Kernel Code

- Consider a **read request** from a program in **user space**. This request will eventually be routed to a driver that manages the hardware that stores the data to be read.
- The user-mode application **first obtains a file handle to this device**, and then calls ReadFile on that handle.
- The kernel will process the ReadFile request, and eventually invoke the driver's callback function responsible for handling read I/O requests.
- The most commonly encountered request for a **malicious kernel component is DeviceIoControl**, which is a generic request from a user-space module to a device managed by a driver.

Drivers and Kernel Code

- Requests originate from a user-mode program and eventually reach the kernel.



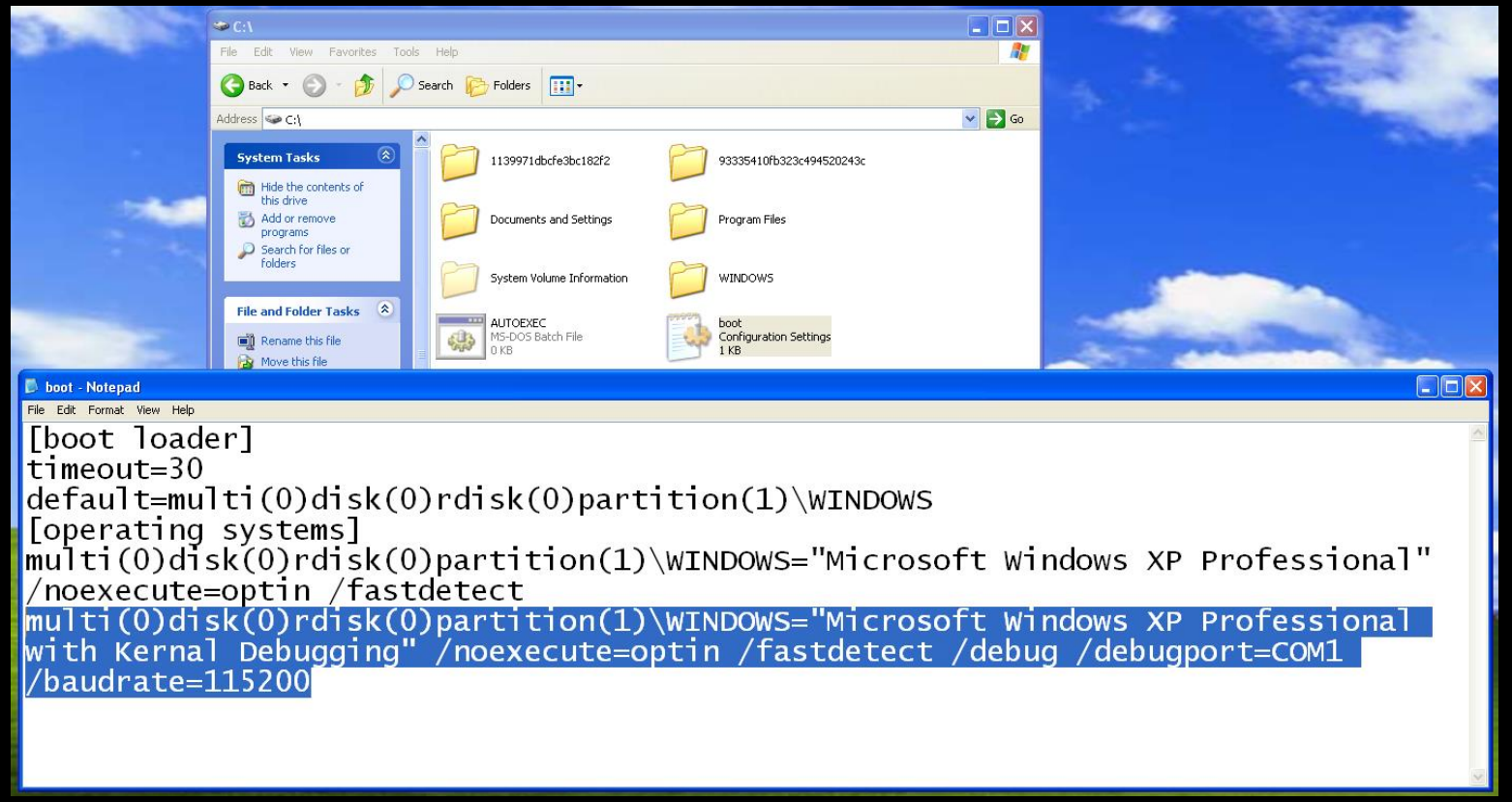
Malicious Drivers

- Malicious drivers generally **do not usually control hardware**; instead, they **interact** with the main Windows kernel components, **ntoskrnl.exe** and **hal.dll**.
- The ntoskrnl.exe component has the code for the core OS functions.
- The hal.dll has the code for interacting with the main hardware components.

Setting Up Kernel Debugging

- Debugging in the kernel is more complicated than debugging a user-space program because when the **kernel is being debugged, the OS is frozen, and it's impossible to run a debugger.**
- Therefore, the most common way to debug the kernel is with two physical systems or with Virtual Machine.
- Unlike user-mode debugging, **kernel debugging** requires a certain amount of initial setup enable kernel debugging, configure VM to enable a **virtual serial port between the virtual machine and the host**, and configure WinDbg on the host machine.
- set up the virtual machine by editing the normally hidden **C:\boot.ini** file. *(On Windows 10 the boot.ini file has been replaced with Boot Configuration Data (BCD))*

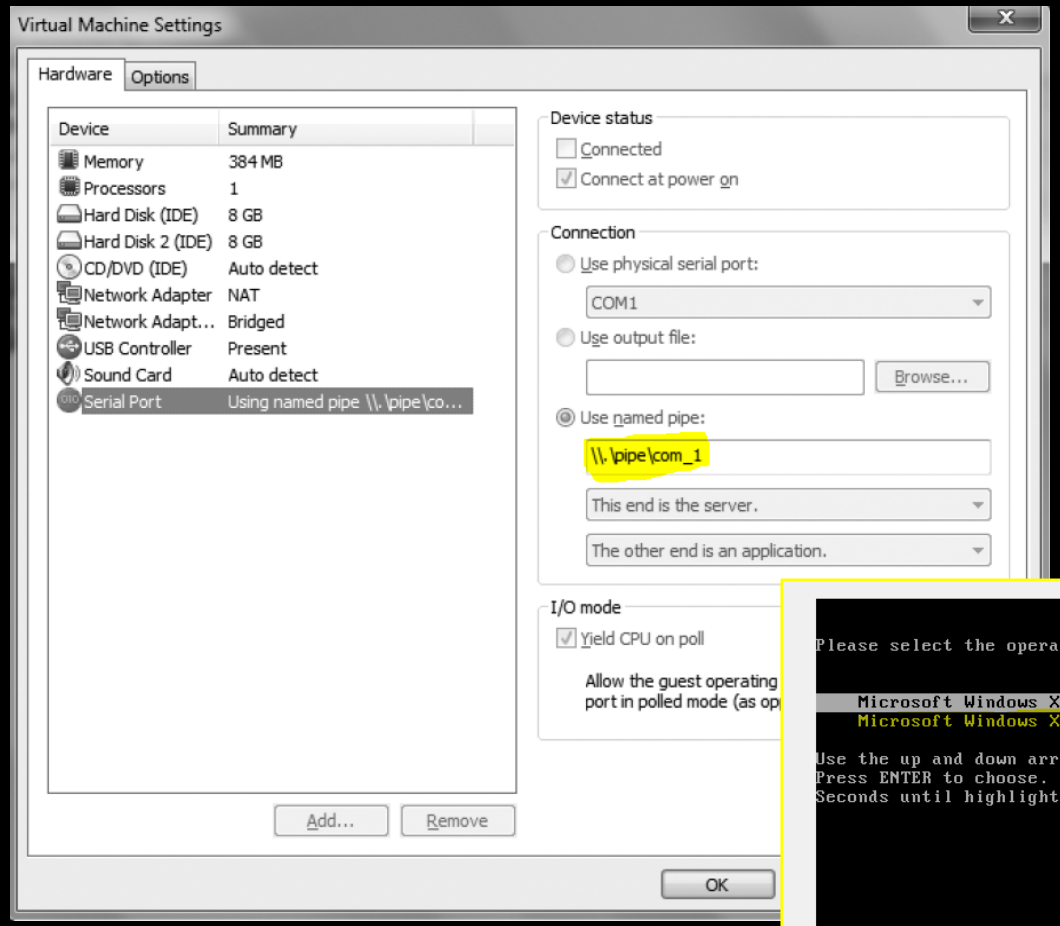
Setting Up Kernel Debugging

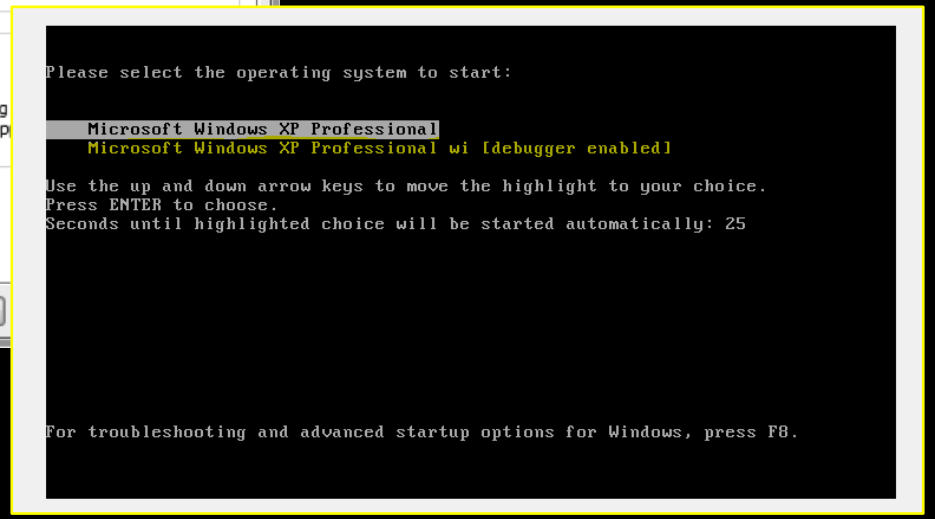


The **/debug flag enables kernel debugging**, the **/debugport=COM1 tells the OS which port will connect** the debugged machine to the debugging machine, and the **baudrate=115200 specifies the speed of the connection**. In our case, we'll be using a virtual COM port created by VM.

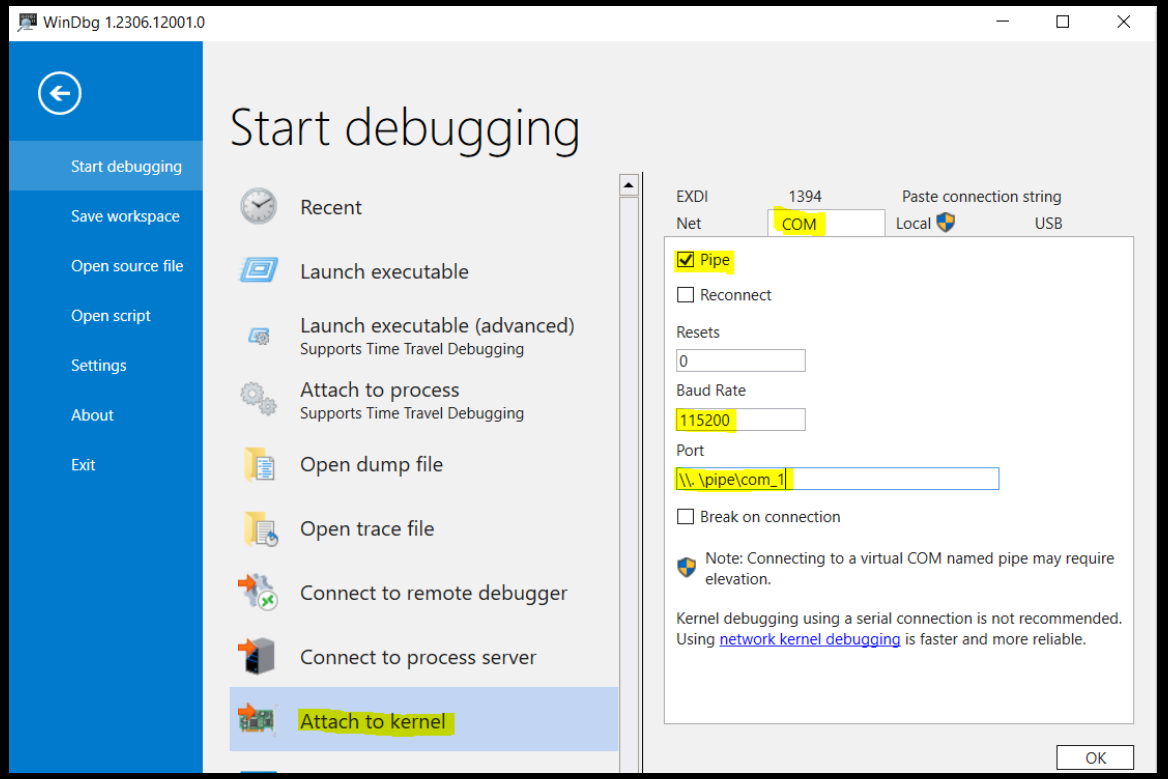
Setting Up Kernel Debugging

- Let's Configure VMware to create a virtual connection between the virtual machine and the host OS.





Setting Up Kernel Debugging

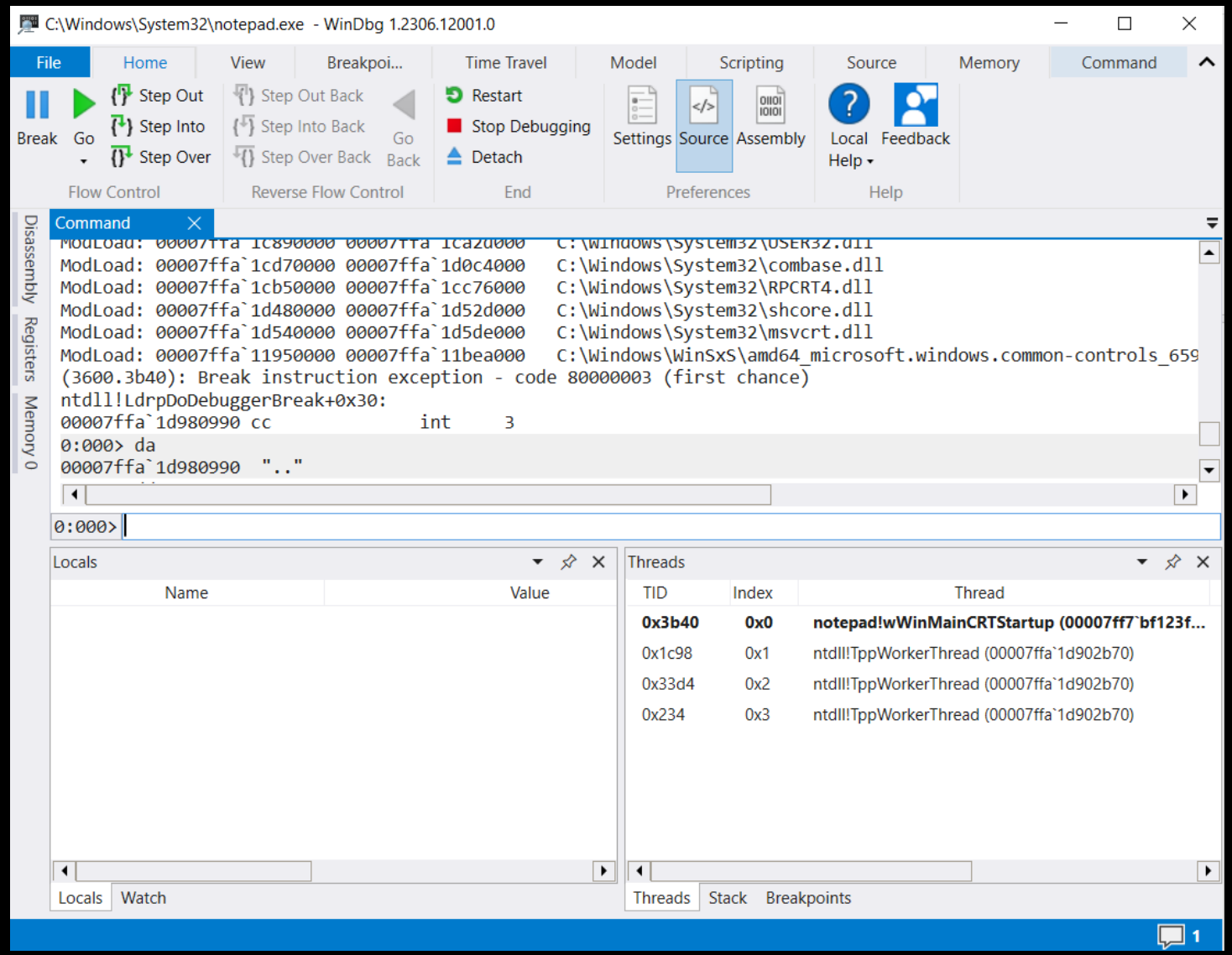


- If the virtual machine is running, the debugger should connect within a few seconds. If it is not running, the debugger will wait until the OS boots, and then connect during the boot process.

WinDBG Kernel Debugging

- WinDbg's memory window supports memory browsing directly from the command line. The d command is used to read locations in memory such as program data or the stack.
- **Syntax:** dx addressToRead (x is the option given below)
- da Reads from memory and displays it as ASCII text
- du Reads from memory and displays it as Unicode text
- dd Reads from memory and displays it as 32-bit double words
- The e command is used in the same way to change memory values.
- **Syntax:** ex addressToWrite dataToWrite (*x* values are the same values used by the d*x* commands)
- The bp command is used to set basic breakpoints in WinDbg.

WinDBG Kernel Debugging



References

- Practical Malware Analysis by Michael Sikorski