

# Just Another Windows Kernel Perl Hacker

Joe Stewart

June 29, 2007

## Abstract

In this paper I will discuss the basics of the Windows serial debugging protocol, and introduce a cross-platform tool written in Perl to implement the debug protocol without requiring the windbg program.

## 1 Introduction

The Microsoft Windows kernel has included code to allow developers to debug the kernel itself since its inception, although until now, it required use of the freely-available but proprietary windbg program to take advantage of it. Since kernel debugging is best performed over a hardware connection from a second system, a serial protocol was devised to allow remote interaction with the debugging code. This serial protocol is not officially documented, but is also not terribly complex, and has been described in some detail by Albert Almeida[1]. Using this information as a basis, I was able to reverse-engineer debugging sessions using a serial-port sniffer, leading to the protocol implementation described in this paper.

### 1.1 Windows kernel debugging setup

A remote kernel debug session requires two systems - a host (running the user-land debugger) and a target (a system running in debug mode). The systems are usually connected via a null modem serial cable, although recent versions of Windows are capable of debugging over USB 2.0 or IEEE1394 connections. The low-level packet-based debug protocol is the same, however.

The target system is booted into debug mode by addition of the `/DEBUG` option to the `boot.ini` file. Additional options, such as serial port and baud rate are also configured here. When the system is booted with the `/DEBUG` option, the `KdInitSystem` subroutine handles the initialization of several variables and tables, including the global variable `KdDebuggerEnabled`, which system modules and programs can use to tell when the kernel debugging code is in play. At this point, the system is ready to handle debugging protocol packets from the host system, but continues execution normally until such an event is received.

## 1.2 windbg

windbg (pronounced “windbag”) is the software provided by Microsoft as part of the Windows DDK. It is a fairly feature-rich debugger, and provides an API for extensions that can be loaded as dynamic link libraries. However, as I prefer to use Windows as a reverse-engineering or development platform as little as possible, it seemed prudent to understand the low-level details of the debugger’s interaction with the debug code built into Windows.

## 2 Microsoft Debug Protocol

The serial debug protocol is packet-based, and uses a defined set of structures to exchange information about the system and the debugger, as well as debug commands and parameters. Packets received are replied to with an ACK packet and are checksummed, in order to deal with corruption or data loss.

### 2.1 Basic packet data exchange

There are three classes of packets used in the protocol: normal packets, control packets, and the break-in packet. Normal and control packets also contain a packet type, defining the specific function of the packet. Control packet types are:

PACKET\_TYPE\_KD\_ACKNOWLEDGE

PACKET\_TYPE\_KD\_RESEND

PACKET\_TYPE\_KD\_RESET

Normal packets may be one of:

PACKET\_TYPE\_KD\_STATE\_CHANGE32

PACKET\_TYPE\_KD\_STATE\_MANIPULATE

PACKET\_TYPE\_KD\_DEBUG\_IO

PACKET\_TYPE\_KD\_STATE\_CHANGE64

Exchange of kernel data/commands between the debugger and the target is accomplished with STATE\_MANIPULATE or STATE\_CHANGE normal packets. The flow of the protocol is maintained using the various control packet types.

A typical exchange sequence might be a break-in packet, followed by a STATE\_CHANGE packet from the target, which is ACKed by the debugger. The debugger then might send a command inside a STATE\_MANIPULATE packet, which is ACKed by the target. Any data that might result from the command would be sent back to the host inside a STATE\_MANIPULATE packet.

## 2.2 Packet headers

A packet header is constructed as shown:

Packet Leader (4 bytes)	
Packet Type (2 bytes)	Byte Count (2 bytes)
Packet ID (4 bytes)	
Checksum (4 bytes)	

The packet leader is 0x30303030 for a normal packet, and 0x69696969 for a control packet. The packet ID does not have to be incremented by the debugger, you need only ACK any packets received with the corresponding packet ID sent from the host. The checksum value is calculated by a simple sum of the payload bytes.

All packets utilize this packet header structure except for the break-in packet. It consists of a single byte, 0x62. If the break-in is successful, the target will respond with a STATE\_CHANGE packet informing the debugger that the system execution has been halted and control is being passed to the debugger.

## 2.3 Protocol functions

There are two crucial control packet types, the ACK and the RESET packet. Neither has a payload, so only the header is sent. For an ACK packet, the packet type is set to 0x0004, and a RESET packet is type 0x0006. RESET packets are used when the debugger and the target need to synchronize their operations. There is a third control packet type, RESEND, but I have not considered its use in my implementation, for reasons of keeping simplicity.

## 2.4 API functions

Using normal packets, we are able to access all of the exposed functionality of the debug API. This includes reading and writing virtual memory, physical memory or IO space, accessing kernel variables and context, setting or removing breakpoints, rebooting or resuming execution of the system, or forcing a kernel crash-dump to occur. The APIs are accessed by formatting a STATE\_MANIPULATE packet using the struct defined by \_DBGKD\_MANIPULATE\_STATE32 or \_DBGKD\_MANIPULATE\_STATE64. It in turn defines an 2-byte API number. The most commonly-used API numbers along with their corresponding names are listed below:

### **Virtual memory**

0x3130 DbgKdReadVirtualMemoryApi  
0x3131 DbgKdWriteVirtualMemoryApi  
0x3156 DbgKdSearchMemoryApi  
0x315b DbgKdFillMemoryApi  
0x315c DbgKdQueryMemoryApi

### **Physical memory**

0x313d DbgKdReadPhysicalMemoryApi  
0x313e DbgKdWritePhysicalMemoryApi

### **Control**

0x3137 DbgKdReadControlSpaceApi  
0x3138 DbgKdWriteControlSpaceApi  
0x3132 DbgKdGet Context Api  
0x3133 DbgKdSet Context Api  
0x313b DbgKdRebootApi  
0x3136 DbgKdContinueApi  
0x3149 DbgKdCauseBugCheckApi  
0x3146 DbgKdGet VersionApi  
0x3150 DbgKdSwitchProcessor  
0x3151 DbgKdPageInApi (may not exist in all API versions)  
0x3152 DbgKdReadMachineSpecificRegister  
0x3153 DbgKdWriteMachineSpecificRegister  
0x315d DbgKdSwitchPartition

## I/O

0x3139	DbgKdReadIoSpaceApi
0x3138	DbgKdWriteIoSpaceApi
0x3144	DbgKdReadIoSpaceExtendedApi
0x3145	DbgKdWriteIoSpaceExtendedApi
0x3157	DbgKdGetBusDataApi
0x3158	DbgKdSetBusDataApi

## Breakpoints

0x3134	DbgKdWriteBreakPointApi
0x3135	DbgKdRestoreBreakPointApi
0x3142	DbgKdSetInternalBreakPointApi
0x3143	DbgKdGetInternalBreakPointApi
0x3147	DbgKdWriteBreakPointExApi
0x3148	DbgKdRestoreBreakPointExApi
0x315a	DbgKdClearAllInternalBreakpointsApi

Each API number corresponds to a different payload structure containing arguments, variables or raw data. However, a detailed layout of each API structure is beyond the scope of this paper. This information can be found in the file `windbgkd.h`, which is part of the ReactOS project.

## 3 Perl framework

Many of the essential debug APIs have been implemented in a Perl framework I have developed called `windpl` (pronounced “windpill”). The source code is available from <http://www.joestewart.org/windpl/> and is GNU GPL licensed. At this time it is only procedural code. At some point it might be warranted to create a full-blown object-oriented module with better asynchronous I/O support, but as a proof-of-concept, the program does work and can be used as a simple command-line debugging console.

### 3.1 Current featureset

Some of the advanced capabilities in windbg have been implemented in the windpl framework, such as the ability to list processes and find import addresses in kernel or userspace modules. Another feature in windpl (which windbg doesn't have) is the ability to directly inject userspace threads into the system, using the Windows asynchronous procedure call API. This technique was derived from eEye's paper on kernel exploitation[2], however in our implementation it is accomplished by manipulating kernel structures only, there is no kernel-based shellcode needed. An example function in the framework can inject a Windows message box into explorer.exe as a demonstration of this technique.

### 3.2 Future development

There are numerous opportunities for extending the windpl framework to create other useful tools for hacking the Windows kernel. For instance, one could use the framework to create stealthier malware sandboxes or perform live memory forensics or rootkit detection in malware-infected systems. Because the code is freely available, with a little knowledge of Perl it should be easy to hack in additional functionality. I also expect to see the protocol implemented in other scripting languages as well, so it is doubtless that we will see windpy (windpie?) before long (perhaps with even cleaner code and a more robust I/O loop). Regardless, it should be interesting to see what other ring-0 enthusiasts are able to devise in the future with only a null modem cable and a few lines of code.

## 4 windpl command reference

`bc <address>` - clear breakpoint at address

`bl` - list breakpoints

`bp <address>` - set breakpoint at address

`break` - send break-in packet to host

`continue` - resume execution

`dw <address>` - read dword at virtual address

`eprocess <address>` - parse the eprocess block at address

`findprocessbyname <name>` - find a process from process name

getcontext - get the current thread context

getprocaddress <module name> <api name> - locate an procedure's import address

getpspcidtable - get the psp cid table

listexports <baseaddr> - list all the exports from the module at baseaddr

listmodules - list loaded modules

logical2physical <address> - convert a virtual address to a physical address

messagebox <title> <text> - inject a messagebox into explorer.exe process

parsepe - <baseaddr> give some information about the PE file at baseaddr

processcontext <pid> - show the context info for the given pid

processlist - list running processes

quit - exit the debugger

r <register>=<value> - read or set the given register

readphysicalmem <address> <length> - read physical memory

readvirtualmem <address> <length> - read virtual memory

reboot - reboot the target

reset - reset the debugger protocol stream

version - show debug API version information

writevirtualmemory <address> <data> - write bytes to virtual memory

## References

- [1] Albert Almeida. Kernel and remote debuggers, November 2003. Available from World Wide Web: <http://www.vsj.co.uk/articles/display.asp?id=265>.
- [2] Barnaby Jack. Remote windows kernel exploitation - step into the ring 0, February 2005. Available from World Wide Web: <http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf>.