Windows Driver Memory Analysis: A Reverse Engineering Methodology

James S. Okolica* and Gilbert L. Peterson

Air Force Institute of Technology
Graduate School of Engineering and Management
Department of Electrical and Computer Engineering
AFIT/ENG
2950 Hobson Way
Wright Patterson AFB, OH 45433
phone: 937-255-3636 x7255, x4281
fax: 937-904-7979
email: jokolica@afit.edu, gpeterso@afit.edu

Abstract

In a digital forensics examination, the capture and analysis of volatile data provides significant information on the state of the computer at the time of seizure. Memory analysis is a premier method of discovering volatile digital forensic information. While much work has been done in extracting forensic artifacts from Windows kernel structures, less focus has been paid to extracting information from Windows drivers. There are two reasons for this: (1) source code for one version of the Windows kernel (but not associated drivers) is available for educational use and (2) drivers are generally called asynchronously and contain no exported functions. Therefore, finding the handful of driver functions of interest out of the thousands of candidates makes reverse code engineering problematic at best. Developing a methodology to minimize the effort of analyzing these drivers, finding the functions of interest, and extracting the data structures of interest is highly desirable. This paper provides two contributions. First, it describes a general methodology for reverse code engineering of Windows drivers memory structures. Second it applies the methodology to **tcpip.sys**, a Windows driver that controls network connectivity. The result is the extraction from **tcpip.sys** of the data structures needed to determine current network connections and listeners from the 32 and 64 bit versions of Windows Vista and Windows 7.

Keywords: Digital Forensics, Reverse Engineering, Direct Kernel Object

Manipulation (DKOM), tcpip.sys, Windows 7, Windows Vista.
2000 MSC: 60, 490

1. Introduction

There are several benefits of supplementing hard drive media analysis with memory analysis. Since programs load into memory prior to their instructions being sent to the processor, looking at programs in memory increases the probability of seeing the actual code that executes rather than code obfuscated in some way. Since system configuration information is loaded into memory where it may be changed, looking at the configuration in memory increases the probability of seeing the actual system configuration rather than what the configuration was before potentially being modified. Finally, since operating system structures, like current processes, objects in use, and network connections, reside in memory, memory is the only place to find and analyze this information [23]. The first step in memory analysis is extracting the O/S structures, such as process lists and registry entries as well as network activity. Fortunately,several authors have documented the structures and location of these artifacts in Windows XP [26, 1, 7, 31] and one or more versions of Linux [3, 10, 16]. Unfortunately, the specifics of these structures vary between operating system versions. While [23] overcomes this for the Windows NT family of operating system kernel structures (i.e., processes and registry entries), it remains an issue for non-kernel artifacts, including network connections, which are stored in device drivers.

Device drivers provide an I/O interface for a particular type of device [24]. They are structured differently from portable executables (PE) that have a single entry point where execution begins and then flows linearly through code. Instead, functions in the driver are called from external programs directly. As a result, reverse code engineers cannot start at the beginning and work their way down. Instead, they must first discover which functions are relevant for analysis. **Tcpip.sys** is the primary driver for managing network connectivity within the Windows NT family of operating systems [13]. By reversing the structures within **tcpip.sys**, memory forensic tools can locate these structures within a Windows memory dump and extract the network connections that existed when the memory was captured.

While **tcpip.sys** is specific to the Windows NT family of operating systems, similar drivers exist for other families of operating systems including

Linux. Initial thoughts may be that the open-source nature of Linux would make finding these structures easier. While this is true, it neglects to consider the number of implementations and versions of Linux. So, although finding the structures of interest for a specific implementation and version of Linux may indeed be easier, generalizing the location of these structures across all versions of Linux to produce a single tool that can find these structures in an arbitrary Linux memory dump is not feasible.

There are two primary methods for reverse code engineering (RCE) [8]. The first, offline code analysis, deciphers what the code does by manually reading the code. The second, live code analysis, uses both the code and its behavior to gain a better understanding of what the code does. In addition, there are two levels of RCE, user-level and kernel-level. User-level RCE restricts itself to user-level programs that do not have access to modifying the kernel or its structures, while kernel-level RCE restricts itself to kernel-level programs that access and/or modify the kernel and its structures. Since **tcpip.sys** is a system driver, reversing it requires kernel-level debugging.

This paper proposes a methodology for the reversing of **tcpip.sys** to extract the structures that enumerate the TCP and UDP network connections, listeners, and endpoints that exist on a target machine. Section 2 places the methodology within the framework of related work. It begins by discussing Windows memory forensic tools and their inability to extract non-kernel artifacts due to lack of exported structures. Section 2 then goes on to discuss reverse code engineering techniques including kernel-level debugging. Section 3 then provides a detailed explanation of the methodology breaking down into its constituent parts. The authors then implement the methodology for Windows 7 64 bit to extract the memory structures. Section 4 shows the results of these tests. A brute force, exhaustive search of **tcpip.sys** would potentially require looking at 4,974 different functions. By using the methodology discussed in this paper, analysts were able to reduce this initially to 30 functions and then to further reduce those 30 functions to 4. The end result is several weeks of reverse code engineering of **tcpip.sys** to arrive at a collection of offsets that allow extraction of network connections for forensic analysis. Finally, Section 5 discusses limitations of the methodology and proposes future extensions.

2. Background

2.1. Memory Forensics

Historically, computer forensics focused on file systems, i.e., the files, programs, configurations and log files installed on a target machine. While this information is valuable, it overlooks the architecture of computers. Before the computer executes a program, displays a file, or logs an activity, it first loads the information into memory. Memory forensics examines the information captured from memory at the time the computer is seized. Forensic memory analysis starts with collecting the memory from the target machine followed by parsing the memory dump into meaningful artifacts.

There are several tools for collecting memory from a machine including windd [28] and Memoryze [20]. The important step is to use these tools to capture memory *before* the computer is shut off or rebooted. Although some forensic artifacts may remain after a reboot [26] or even after power is turned off [12, 9] after power is turned off, to maximize the retrieval of forensic artifacts, memory dumps need to be done prior to rebooting or disconnecting the machine. These memory capture tools generally have options to either store the entire memory image on the local hard disk or send it over a network connection to another machine. When possible, saving the memory image on another machine is preferable [19] because it minimizes changes to the seized computer's hard disk. Recently, Beverly et al [2] observed that many of the forensic artifacts found in memory are routinely stored to disk when computers go into hibernation. Given the frequency that users put their laptops into hibernation, there is a good chance that a seized computer may have a hibernation file. While the headers of these compressed hibernation files are destroyed, each page of the compressed hibernation file is also marked and may be found intact. Beverly carved these residual files on disk to find network activity that had occurred in the past. However, it should be possible to extend his work and retrieve a wide range of forensic artifacts from the old hibernation files of computers (particularly laptops, cell phones, PDAs, and tablets).

After the memory has been captured, the next step is to extract forensic artifacts from it. Several tools for parsing Windows XP memory dumps include Schuster's **Ptfinder** [26, 27], Betz's **Memparser** [1], and Walter's **Volatility** [31]. The purpose of these tools is to identify key operating system structures and extract a subset of the data in them. Some of the structures found in the Windows kernel include processes, threads, and configurations (i.e., registry hives).

Unfortunately, what the above tools have in common is a limitation to a particular operating system (and in many cases, service pack). CMAT [23] provides some additional flexibility by parsing much of the same information out of a memory dump from any of the Windows NT family of operating systems (including Windows XP, Vista, and 7). As shown in Figure 1, CMAT begins by searching through memory for a debugger data structure; once it finds this data structure, it uses the information in it to determine if the memory dump is 32 bit, 32 bit with physical address extensions (PAE) or 64 bit. At the same time, it determines the version of Windows, the kernel's base address, the address of the loaded module list. If it was unable to find the debugger structure, it then searches for the string kernel.exe (or some variant of it); CMAT then backtracks to the beginning of kernel.exe (implicitly finding the kernel's base address) and uses the information in the PE header to determine the above information. With this information in hand, CMAT sequentially searches through the memory dump looking for top-level page directory tables (i.e., page directory map tables for 64 bit operating systems, page directory pointer tables for 32 bit operating systems with PAE enabled, and page directory tables for 32 bit operating systems). Finally, with all of this information, CMAT is able to extract the globally unique identifier from the kernel's debug section and download the kernel's program database (PDB) file from the Microsoft Symbol Server. It then uses the structures and symbols found in the PDB file to create signatures for searching for process and configuration files in the memory dump. Once it has found these structures, it extracts the forensic artifacts of interest.

While CMAT does provide more flexibility than the other parsing tools, it is limited to operating system structures, i.e., structures found in and exported by **ntoskrnl.exe**. One of the key pieces of forensic data, open network connections, does not reside within **ntoskrnl.exe**. Instead, it resides within the driver *tcpip.sys*. Furthermore, while **ntoskrnl.exe** makes many of its structures available to external programs, *tcpip.sys* does not. This is captured in the differences between **ntoskrnl.exe**'s program database (PDB) file and *tcpip.sys*'s PDB file. **Ntoskrnl.exe** has approximately 5,000 structures in its PDB file while *tcpip.sys* has none (*tcpip.sys* has 7,336 symbols and **ntoskrnl.exe** has 18,640 symbols). As a result, in order to bring structure to the data section of *tcpip.sys*, its structures must be reversed.

What is needed is a methodology for quickly reversing *tcpip.sys* from an arbitrary Windows O/S to extract the symbols and data structures needed to provide the network connections. The methodology can then be applied to any of the forensic tools (e.g., CMAT [23] or Volatility [31]) to allow them to handle new operating systems.

2.2. Reverse Code Engineering

Reverse Engineering was defined in 1990 by Chikofsky and Cross as "the process of analyzing a subject system to (i) identify its system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction" [4]. In the case of binary executables, if the software's developers have not consented to this, the question of legality is often raised. While every situation is different, the general consensus is if the code is not duplicated (i.e., the reverser doesn't take the reversed code and put it into a competing product) and if the public benefits from the effort (e.g., through additional functionality not otherwise available), reverse code engineering is legal [8].

In the past twenty years, there have been three primary focus areas in reverse code engineering (RCE). The first area is the analysis of large-scale systems for maintenance, enhancement and porting to new systems [21, 14, 30]. In the case of large systems, code analysis is only one element of RCE. Equally important is an understanding of the user requirements, system architecture, engineering constraints, and design tradeoffs [22]. The second area is on developing tools for better static analysis of source code, including tools to handle object oriented programs [6, 18, 17] and Unified Modeling Language (UML) [15]. Finally, the third area is reversing malware. While there has been a tremendous amount of effort dedicated to reversing specific pieces of malware (e.g., Conficker, Stuxnet, etc.), the efforts have typically been performed in an ad hoc manner. According to Muller et al, "the process must become more mature and repeatable, and more of its elements need to be supported by automated tools" [22].

While reverse engineering of malware often means reversing binary code, there has been little published on reversing other types of binary code, particularly drivers for the Windows operating systems. Chipounov [5] has developed a tool, RevNIC, that allows drivers to be automatically ported from one operating system to another and has successfully tested the tool against four small Windows NIC device drivers(pcntpci5.sys, rtl8139.sys, lan9000.sys, and rtl8029.sys). While not directly applicable to the problem at hand, Guha and Mukherjee [11] have reversed TCP/IP source programs for UNIX systems using slicing. Unfortunately, the source code for Windows drivers is not available. An important distinction between malware and Windows drivers is that when Microsoft compiles their programs, they create a program database (PDB) file which stores debug information. They then publish this PDB file on their symbol server so that debuggers can download it on the fly to aid with debugging. The consequence of this is that different reversing techniques are possible with Windows drivers than with malware.

2.3. Kernel-Level Debugging

While user-level debugging can be performed straightforwardly, using several different disassemblers and debuggers, kernel-level debugging is more involved. First, an analyst cannot debug the kernel of the operating system he is using. Instead, the analyst must instantiate an instance of the operating system and then debug it from another instance. This can be done either with two machines connected through a null modem or through the use of virtual machines. In addition, the number of debuggers that may be used decreases, especially when working with 64 bit operating systems, to either WinDbg or KDbg.

First, a virtual machine is installed; in this case, VMWare was chosen. Next, an instance of Windows 7 is loaded into the virtual machine. Third, a virtual serial connection is created between the guest operating system and the host operating system with the guest O/S being the server and the host O/S being the client. In some cases, an additional piece of software, e.g., VirtualKD [29], is used to provide the interface. In this case, **vmmon.exe** is run on the host machine and then the virtual machine is started. WinDbg starts automatically, connects to the virtual machine and kernel-level debugging begins.

3. Methodology

As shown in Figure 2, before attempting to understand the data structures within a Windows' driver, some preliminary reconnaissance is advisable. There are several methods to gain information. User applications that interact with the driver provide the analyst with relevant names of the functions and symbol names in the user application. The analyst can then look in the driver for symbols and functions with similar names. Offline code analysis (OCA) gives an overview and allows the analyst to look for symbols or functions with potentially relevant names. Finally, seeding and harvesting data enables an analyst to seed known values into the data structures of interest and then extract the locations in memory where those data structures reside (e.g., opening Internet Explorer and going to a known website will place its IP address in the data structure of interest in **tcpip.sys**).

Once these preliminary steps are performed, the analyst has three leads for setting breakpoints to perform live code analysis: symbols of interest, functions of interest, and memory addresses of interest. After setting breakpoints for each of these, the analyst then begins live code analysis to winnow down the results to a handful of likely functions. The final step is then to reverse the functions of interest to extract the structure of the data structures of interest.

3.1. User Application Analysis

When faced with a large, opaque driver, the first step in gaining insight into it is to perform live code analysis on a small, user-level program that uses the driver. Such an analysis is significantly faster and will provide symbol and function names of interest. For instance, if the user application has a function called "GetExtendedTcpTable" which eventually contains the data of interest, there may be a function or symbol in the driver called either TcpTable or ExtendedTcpTable that should be investigated.

Tcpvcon.exe [25] is a user-level program developed by Mark Russinovich which shows the current TCP and UDP connections. Using **tcpvcon.exe** as a starting point provides insight into the **tcpip.sys** symbols, functions, and entry points. Reversing **tcpvcon.exe** is fairly straightforward. Like most application portable executables (PEs), it begins with a *main* function. *Main2* begins by loading a constant called "ShowAllEndpoints" and then makes external calls to "GetExtendedTcpTable", "GetTcpTable" and "GetUdpTable". When these functions are traced, function calls are made from them including calls like "nsi_NsiEnumerateObjects- AllParameters". While it is not possible to follow the calls into **tcpip.sys** due to programs interacting with drivers through traps, the names of these functions and symbols provides a starting point for offline code analysis.

3.2. Offline Code Analysis

The most intuitive approach for understanding a program is to read the code, especially given some previous understanding. For example, in Windows XP, the data structures that hold the TCP/IP and UDP connections are

ADDROBJTABLE and TCBTABLE. Unfortunately, neither of these symbols exist in the Windows 7 version of **tcpip.sys** nor are there any similarly named symbols. However, there are several symbols with promising names (e.g., TcpPortPool, UdpPortPool, InetSockAddrStorage, IOCtlDispatchTable, TcpCcbObject, and TcpInetTransport) that can have breakpoints set on them. In addition, when the symbols and functions discovered during User Application Analysis are searched for, several groups of promising functions appear, e.g., UdpEnumerateAllEndpoints. Breakpoints are set on the potential functions and symbols of interest in the hopes that one of them is called during live code analysis.

3.3. Seeding and Harvesting Data

While OCA and user applications provide likely symbol and function names of interest, seeding and harvesting data provides potential memory addresses of interest (Figure 3). First the analyst generates some unique data (e.g., by establishing a TCP/IP connection to a known IP address). Next, the analyst generates a memory dump and then transfers it to the host O/S; he then suspends the guest O/S so the state doesn't change. Finally, the analyst locates the data in the memory dump, translates the physical address into a system virtual address. The analyst now has virtual memory addresses to set breakpoints on. Observe that the seeded data does shows up in the memory dump in the process space for the user application (in this case, Internet Explorer) that generated it; however, by noting that these physical addresses don't have any corresponding valid virtual system addresses, they are eliminated.

3.4. Live Code Analysis

The analyst then sets breakpoints on the functions of interest, the memory locations of interest, and, as much as possible to debugger limitations, the symbols of interest. The guest O/S is then restarted and the user application re-run, triggering a debugger breakpoint. The resultant call stack shows both the relevant driver function and how it was called. When this process is applied to **tcpip.sys**, several potential functions are discovered. Furthermore, after a quick look at the disassembled code, several of these functions are stubs that call others in the list. At this point, additional breakpoint are set and the user application run again.

When live code analysis is performed on a 64-bit Windows 7 virtual machine, first the memory locations that hold the IP address is located within the **tcpip.sys** function Ipv4EnumerateAllPaths. Judging by the name, this certainly seems like it may be the correct function since the name suggests its purpose is to enumerate all of something. The first named symbol in the function is Ipv4Global which also seems promising. It appears to be composed of a linked list of IP "compartments" which house relevant IP information. Within each compartment is a hash table composed of IP information entries. The hash table is implemented as a series of linked lists. Each linked list either has a forward and backward link that points to itself (in which case there is no entry) or a forward link that points to an IP information entry. This entry begins with a linked list that eventually points back to the hash table. Unfortunately, while the IP information entry holds both the local and remote addresses, it does not hold information on the process ID that created the connection (or if it does, the author was unable to find it). A second function found by tracing down through **tcpvcon.exe** was "nsi_NsiEnumerateObjectsAllParameters".

Since both nsi_NsiEnumerateObjectsAllParameters and Ipv4Enumerate-AllPaths have the word "Enumerate" in common, functions in **tcpip.sys** with the word enumerate in their name are searched for. Going back through a list of all functions in **tcpip.sys** (retrieved using CMAT and the tcpip.pdb file), finds approximately 30 functions. While this is a large number of functions, they break down into four categories, InetEnumerateXXX, IpEnumerateXXX, TcpEnumerateXXX, and UdpEnumerateXXX. Furthermore, if the functions are restricted to the Tcp and Udp functions, the result is UdpEnumerateAllEndpoints, TcpEnumerateAllPortPropertyObjects, TcpEnumerateBasicConnections, TcpEnumerateListeningConnections, TcpE numerateConnections, TcpEnu- merateAllConnections, UdpEnumerateEndpoints, TcpEnumerateListeners, and TcpEnumerateConnectionType. After examining the disassembled code, several of these functions are stubs that call others in the list. At this point, placing breakpoints at the start of each of the functions and running **tcpvcon.exe** and **netstat.exe** in the guest operating system provides an entry point for live code analysis.

3.5. Tcpip.sys Functions that Describe the data Structures

Using the above approach, Reversing TcpEnumerateConnections, TcpEnumerateListeners, and UdpEnumerateEndpoints provides sufficient details to describe the data structures needed to report on the TCP and UDP connections and listeners in Window 7.

TcpEnumerateConnections enumerates all established TCP connections, i.e. it enumerates connections where there is both a local and a remote IP address. Figure 4 shows the function flow. First the function begins a loop that repeats for the number of partitions (found in *PartitionCount*). Within this loop is a second loop that uses Microsoft's runtime library functions to enumerate a hash table. First, the function acquire a spin lock and raises the interrupt level to Deferred Procedure Calls (DPC). It then retrieves the connection information, starting with the local and remote addresses and ports, followed by some flags and the process that owns the connection. It then releases the spin lock and finds the next element in the hash table. Once the table is enumerated, the outer loop than repeats for the next partition table.

There are two symbols used in this function that are important for retrieving network connections, *PartitionTable* and *PartitionCount*. *PartitionTable* is a pointer to the overall structure that holds the connections and listeners. This structure begins with an array of partition tables. The symbol *PartitionCount* tracks how many elements there are in the array. Each partition is 64 bytes. The first element of each partition table is a pointer to a hash table. The hash table is a structure with several elements. The elements at offset 0x08 is the maximum number of elements in the hash table while the element at offset 0x20 is a pointer to the first element in the hash table (Figure 5). The hash table structure itself is fairly straightforward. Each element is a doubly linked list (forward link followed by backward link). If the forward link points to itself than the hash table entry is empty. If it doesn't, then it points to the first entry in a linked list of network connections/listener information, henceforth called *IpInfo*.

The local port is at offset +0x44 of *IpInfo* and the Remote Port is at offset +0x46. The local IP address is two levels deep (Figure 5). There is a pointer at offset -0x08 which points to a structure which contains the IP addresses, henceforth called *IpAddressInfo*. In this structure at offset 0x00 is a pointer to another structure (structure A). Within this structure at offset 0x10 there is a pointer to a third structure (structure B). Within this structure at offset 0x00, there is a pointer to the local IP address. The remote address is less deep. At offset 0x10 of *IpAddressInfo*, there is a pointer to the remote address. The process ID associated with this IP information is at offset +0x210. Finally, to determine whether this is IPv4 or IPv6, the pointer at -0x10 is used. It points to another structure which has a pointer at offset 0x14 which points to a value. If this value is 0x17, the information relates to IPv6; otherwise it relates to IPv4 (Figure 5).

These two functions are very similar, both in flow (Figure 6) and in the type of data structures used (Figure 7). They begin by enumerating a pool of ports (either UDP ports or TCP ports). They then check to see if there is a local address (just for TCP listeners) and then retrieve the remote address, port, owning process id, and some flags. They then repeat for the next entry in the port pool.

These function enumerates all TCP listeners and UDP endpoints, i.e. it enumerates connections where there is at most a local IP address. There are two symbols used in these function that are important for retrieving network listeners and UDP endpoints, $TcpPortPool$ and $UdpPortPool$. The two variables are pointers to a pool of available ports. Each pool has two fields, $BitMapPtr$ and $BitMapSize$. The bitmap size is the number of bits in the bitmap. The way the bitmap is arranged is the lowest bit of the first byte is in position 0. The second lowest bit of the first byte is in position 1. The lowest bit of the second byte is in position 8 and so on. Those bits that are set indicate ports that are in use. $BitMapSize$ and $BitMapPtr$ are at offsets $0x90$ and $0x98$ respectively for both pools. To find the location of the port, left shift the bit number by 8. This results in a number that is up to two bytes long. The high byte is the page number and the low byte is the offset. The pointers to the pages begin at offset $0xA0$. A pointer to the list of pointers to the $IpInfo$ record is at offset $0x20$ of the page. The $IpInfo$ record pointer is then found by taking the offset, left shifting by 4, adding 0x08 to the result (Figure 7) and then zeroing out the low 2 bits. The local port is at offset - 0x06 for TCP and at offset - 0x08 for UDP. The local address is again reached via indirection. If there is a valid pointer at offset - 0x18 (offset - 0x28 for UDP), the structure that it points to has a pointer to a structure at offset 0x00 (offset 0x10 for UDP)which has a pointer to the local address at offset 0x00. The remote port and Ipv4/Ipv6 determination is also reached via indirection by the pointer at offset - 0x10 (- 0x68 for UDP) of the $IpInfo$ record and then retrieving the remote port by finding the pointer at offset + 0x14 of the new structure (Figure 7). Finally, a pointer to the process ID is at offset -0x48.

4. Analysis of Results

Using the methodology discussed in this paper, the authors reduced the 5,000 functions in **tcpip.sys** to three functions. Due to one author's unfamiliarity with reverse code engineering, reversing the three functions and finding the locations of the forensic artifacts of interest still took approximately three weeks. However, when considers that this works out to approximately one function per week, reversing all 5,000 functions would have taken this novice author 100 years to complete.

As an added bonus, once the initial reverse code engineering was completed on Windows Vista 32-bit, extending this work to include Windows Vista 64-bit and 32-bit as well as Windows 7 32-bit only added an additional six hours of effort. All of these versions were similar enough that the only effort involved was locating the functions described above and adjusting the CMAT code to match the offset values in the code. The details of these locations are shown in Figures 8, 9, and 10.

With the methodology described, updating CMAT to handle new versions of tcpip.sys consists of reviewing TcpEnumerateConnections, TcpEnumerateListeners, and UdpEnumerateEndpoints and update the values in Figures 8, 9 and 10. Although these figures appear complex, the implementation is straightforward. First the location of the symbols named at the top of the Figures are retrieved from tcpip.pdb. Then the variables in the figures are filled in from top to bottom (including iterating through the partition table, hash table, bit maps and the IPInfo linked lists) resulting in the network connections being retrieved. However, this does assume that while the fields in the data structure may change names or move around, the underlying data structures (i.e., hash tables and bitmaps) won't. The tests do show that this process works on both 32-bit and 64-bit versions of the operating systems.

The results were verified by running *netstat.exe* on the virtual machine immediately after the memory dump was performed and comparing the results. In all cases, CMAT and netstat produced the same results.

5. Conclusions and Future Work

As forensic analysts become more experienced with the types of data they can extract from memory dumps, they will continue to ask for more forensic artifacts. One such artifact is open network connections at the time the suspect computer is seized. Software engineers are then faced with the task of locating the drivers that create and store these artifacts, determining the functions within these drivers that read and/or write these artifacts and then finally, reversing these functions. The authors, faced with the need to reverse **tcpip.sys** to discover the location of network connections used the methodology discussed in this paper. The result was a quick reduction of the almost 5,000 functions in **tcpip.sys** to a manageable three functions. These three functions were then quickly reversed and the locations of the data structures of interest were found.

The primary contribution of this paper is developing a process that enables efficient reversing of Windows drivers and dynamic link libraries. Since they do not have a single entry point, and in the case of drivers, may have no exported functions, locating the data structures of interest is a hard problem. Possibly as a consequence of this, there has not been a lot of effort focused on extracting forensic artifacts from drivers. Unfortunately, there are a large number of forensic artifacts that reside in drivers or dynamic link libraries (DLLs). Network connections are one of them. Others include print spooler information, graphic processing until information and clipboard contents. Applying this methodology to extract these forensic artifacts would be very useful future work.

6. Acknowledgements

This work is supported by the Air Force Research Lab/ Information Grid Division (AFRL/RIG) and the Air Force Office of Scientific Research/ Dynamics and Control (AFOSR/RSL) project 2311/FX. The views expressed in this paper are those of the authors and do not represent the views or policies of AFRL/RIG, AFOSR/RSL, the United States Air Force, or the Department of Defense.

References

[1] C. Betz, memparser, http://sourceforge.net/projects/memparser, 2005.

[2] R. Beverly, S. Garfinkel, and G. Caldwellt, Forensic Carving of Network Packets and Associate Data Structures, *Proceedings of the 2011 Digital Forensic Research Workshop (DFRWS)*, Accepted for Publication, 2011.

[3] M. Burdach Digital Forensics of the Physical Memory http://forensic.secure.net/pdf/mburdach_digital_forensics_of_physical_memory.pdf, 2005. Last accessed 8-Jun-2011.

[4] E. Chikofsky and J. Cross Reverse Engineering and Design Recovery: A Taxonomy *IEEE Software.*, 7 (1) pp. 13–17, 1990.

[5] V. Chipounov and G. Candea Reverse Engineering of Binary Device Drivers with RevNIC *Proceedings of the 5th European conference on Computer Systems.*, pp. 167–180, 2010.

[6] A. van Deursen and T. Kuipers Identifying Objects using Cluster and Concept Analysis *Proceedings of the 21st International Conference on Software Engineering.*, pp. 246–255, 1999.

[7] B. Dolan-Gavitt, Forensic Analysis of the Windows Registry in Memory, *Proceedings of the 2008 Digital Forensic Research Workshop (DFRWS)*, pp. 26–32, 2008.

[8] E. Eilam *Reversing - Secrets of Reverse Engineering*, Wiley Publishing, 2005.

[9] W. Enck, K. Butler, T. Richardson, P. McDaniel, A. Smith Defending Against Attacks on Main Memory Persistence *Computer Security Applications Conference, 2008*, pp. 65–74, 2008.

[10] Y. Gao and T. Cao Linux Memory Forensics: Searching for Processes http://cms.ieee.org/IEEE_Edit/IEEE/iportals/publications/rights/plagiarism/PALpapers 2010. Last accessed 8-Jun-2011.

[11] B. Guha and B. Mukherjee Network Security via Reverse Engineering of TCP Code: Vulnerability Analysis and Proposed Solutions *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer Societies, Networking: the Next Generation.*, pp. 603–610, 1996.

[12] J.A. Halderman, S.D. Schoen, N. Heninger, W. Clarkson, W. Paul, J.A. Calandrino, A.J. Feldman, J. Appelbaum, and E.W. Felten Lest We Remember: Cold Boot Attacks on Encryption Keys *Proceedings of the 17th Conference on Sec urity symposium*, pp. 45–60, 2008.

[13] G. Hoglund, and J. Butler *Rootkits: Subverting the Windows Kernel*, Addison-Wesley, 2006.

[14] R. Keller, R. Schauer, S. Robitaille, and P. Page Pattern-Based Reverse Engineering of Design Components *Proceedings of the 21st Internal Conference on Software Engineering.*, pp. 226–235, 1999.

[15] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, A. Zundorf A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering *Proceedings of the Ninth Working Conference on Reverse Engineering.*, pp. 22–32, 2002.

[16] P. Movall, W. Nelson, and S Wetzstein Linux Physical Memory Analysis *Usenix Annual Technical Conference, April 2005.*, 2005.

[17] M. Lanza and S. Ducasse Polymetric Views - A Lightweight Visual Approach to Reverse Engineering *IEEE Transactions on Software Engineering.*, 29 (9) pp. 782–795, 2003.

[18] G. diLucca, A. Fasolino, F. Pace, P. Tramontana, U. De Carlini WARE: a tool for the Reverse Engineering of Web Applications *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering.*, pp. 241–250, 2002.

[19] K. Mandia, C. Prosise, and M. Pepe, *Incident Response & Computer Forensics, 2ed*, McGraw-Hill/Osborne, 2003.

[20] Mandiant, Memoryze, http://www.mandiant.com/software/ memoryze.htm, Accessed August 15, 2009.

[21] H. Muller, M. Orgun, S. Tilley, and J. Uhl A Reverse Engineering Approach to Subsytem Structure Identification Research and Practice, 5(4) pp. 181–204, 1993.

[22] H. Muller, J. Jahnke, D. Smith, M. Storey, S. Tilley, K. Wong Reverse Engineering: A Roadmap *Proceedings of the Conference on the Future of Software Engineering*, pp. 47–60, 2000.

[23] J. Okolica, and G. Peterson Windows Operating Systems Agnostic Memory Analysis *Proceedings of the 2010 Digital Forensic Research Workshop (DFRWS)*, pp. 48–56, 2006.

[24] M. Russinovich, and D. Solomon, *Microsoft Windows Internals, 5th Edition*, Microsoft Press, 2009.

[25] M. Russinovich, Sysinternals Suite, http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx, Accessed August 15, 2010.

[26] A. Schuster, Searching for Processes and Threads in Microsoft Windows Memory Dumps, *Proceedings of the 2006 Digital Forensic Research Workshop (DFRWS)*, pp. 10–16, 2006.

[27] A. Schuster, PTfinder, http://computer.forensikblog.de/en/2006/03/ptfinder_0_2_00.html, March 2, 2006.

[28] M. Suiche, Windd, http://www.msuiche.net//windd, Accessed March 2, 2010.

[29] Sysprogs, VirtualKD, http://virtualkd.sysprogs.org, Accessed March 2, 2010.

[30] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis Design Pattern Detection Using Similarity Scoring *IEEE Transactions on Software Engineering.*, 32 (11) pp. 896-909, 2006.

[31] A. Walters and N. Petroni, Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process, *Blackhat Hat DC 2007*, www.blackhat.com/presentations/bh-dc.../bh-dc-07-Walters-WP.pdf, 2007.

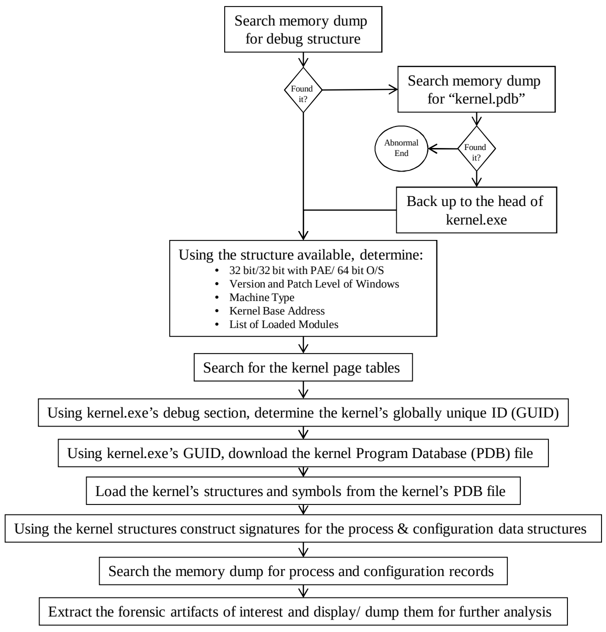

Search memory dump for debug structure
Found it?
Search memory dump for "kernel.pdb"
Abnormal End
Found it?
Back up to the head of kernel.exe
Using the structure available, determine:
• 32 bit/32 bit with PAE/ 64 bit O/S
• Version and Patch Level of Windows
• Machine Type
• Kernel Base Address
• List of Loaded Modules
Search for the kernel page tables
Using kernel.exe's debug section, determine the kernel's globally unique ID (GUID)
Using kernel.exe's GUID, download the kernel Program Database (PDB) file
Load the kernel's structures and symbols from the kernel's PDB file
Using the kernel structures construct signatures for the process & configuration data structures
Search the memory dump for process and configuration records
Extract the forensic artifacts of interest and display/ dump them for further analysis

Figure 1: CMAT's Process Flow

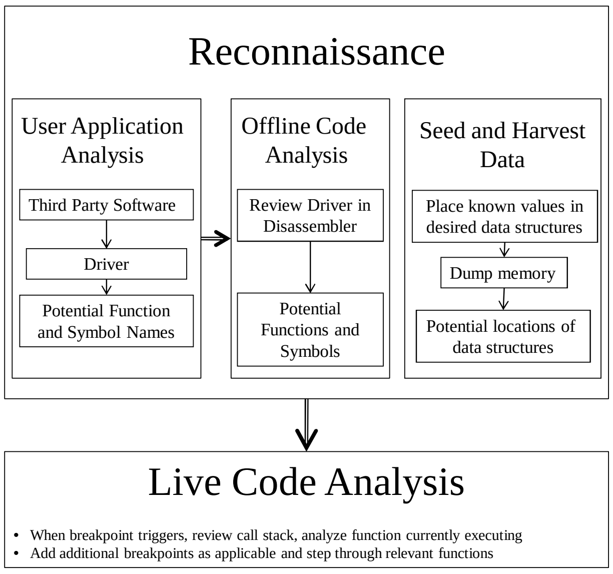
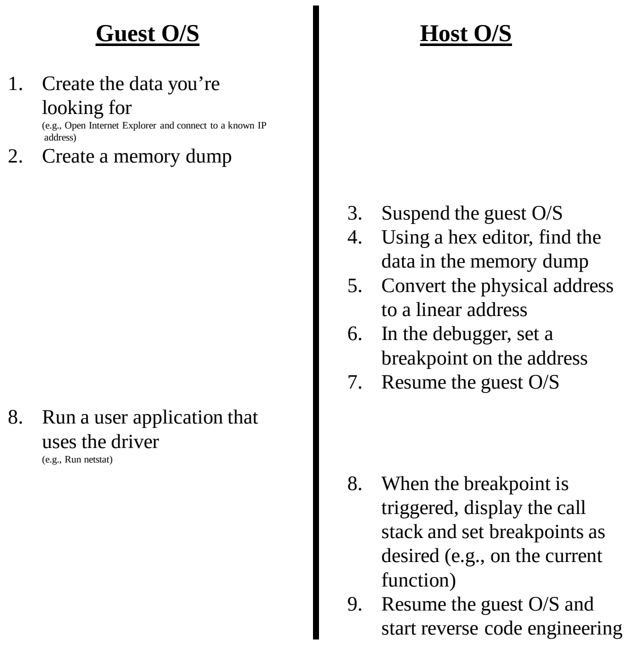

Reconnaissance
User Application Analysis
Third Party Software
Driver
Potential Function and Symbol Names
Offline Code Analysis
Review Driver in Disassembler
Potential Functions and Symbols
Seed and Harvest Data
Place known values in desired data structures
Dump memory
Potential locations of data structures
Live Code Analysis
When breakpoint triggers, review call stack, analyze function currently executing
Add additional breakpoints as applicable and step through relevant functions

Figure 2: A Methodology for Reversing Drivers.



Guest O/S

1. Create the data you're looking for
   (e.g., Open Internet Explorer and connect to a known IP address)
2. Create a memory dump

Host O/S

3. Suspend the guest O/S
4. Using a hex editor, find the data in the memory dump
5. Convert the physical address to a linear address
6. In the debugger, set a breakpoint on the address
7. Resume the guest O/S

Guest O/S

8. Run a user application that uses the driver
   (e.g., Run netstat)

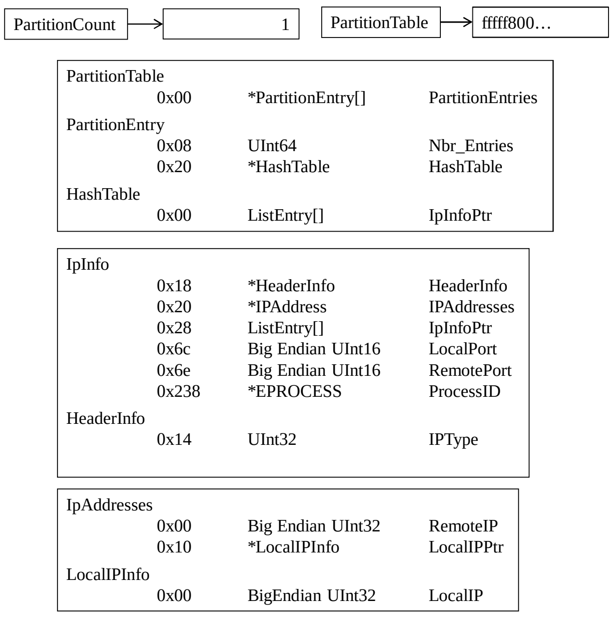
Host O/S

8. When the breakpoint is triggered, display the call stack and set breakpoints as desired (e.g., on the current function)
9. Resume the guest O/S and start reverse code engineering

Figure 3: Seeding and Harvesting Data.

```
Lock TcpInetTransport
For each partition:
  For each entry in the hash table:
    Acquire SpinLock
    Raise Interrupt Level to DPC
    Retrieve Local Addresses & Ports
    Retrieve Remote Addresses & Ports
    Retrieve Flags
    Retrieve Owning Process Handle
    Drop Interrupt Level
    Release SpinLock
```

Figure 4: TCPEnumerateConnections Flow Diagram.



PartitionTable			
	0x00	*PartitionEntry[]	PartitionEntries
PartitionEntry			
	0x08	UInt64	Nbr_Entries
	0x20	*HashTable	HashTable
HashTable			
	0x00	ListEntry[]	IpInfoPtr

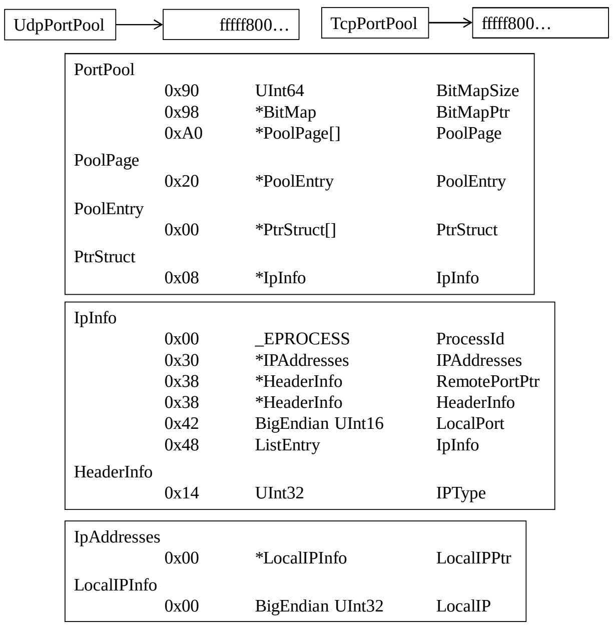
IpInfo			
	0x18	*HeaderInfo	HeaderInfo
	0x20	*IPAddress	IPAddresses
	0x28	ListEntry[]	IpInfoPtr
	0x6c	Big Endian UInt16	LocalPort
	0x6e	Big Endian UInt16	RemotePort
	0x238	*EPROCESS	ProcessID
HeaderInfo			
	0x14	UInt32	IPType

IpAddresses			
	0x00	Big Endian UInt32	RemoteIP
	0x10	*LocalIPInfo	LocalIPPtr
LocalIPInfo			
	0x00	BigEndian UInt32	LocalIP

Figure 5: TCP Connections Data Structures.

```
For each entry in the bitmap
     (port pool):
 If local address, retrieve it
 Retrieve remote address & port
 Retrieve Owning Process Handle
 Retrieve Flags
```

Figure 6: TCP Listeners and UDP Endpoints Process Flow.

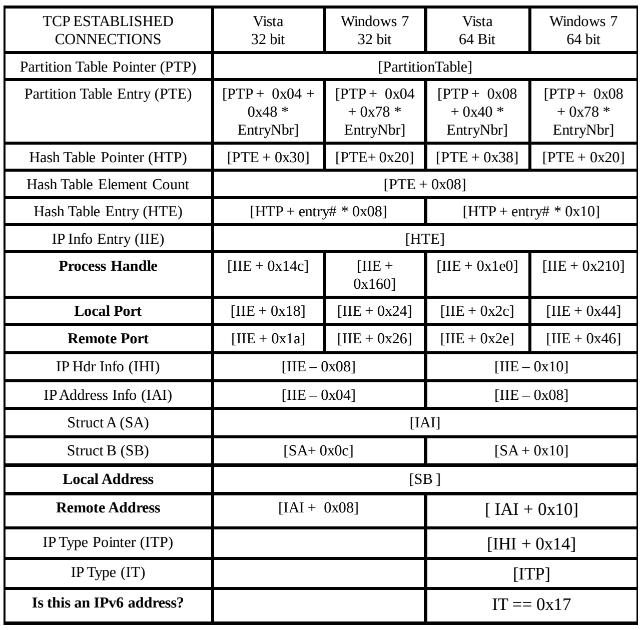


PortPool			
	0x90	UInt64	BitMapSize
	0x98	*BitMap	BitMapPtr
	0xA0	*PoolPage[]	PoolPage
PoolPage			
	0x20	*PoolEntry	PoolEntry
PoolEntry			
	0x00	*PtrStruct[]	PtrStruct
PtrStruct			
	0x08	*IpInfo	IpInfo

IpInfo			
	0x00	_EPROCESS	ProcessId
	0x30	*IPAddresses	IPAddresses
	0x38	*HeaderInfo	RemotePortPtr
	0x38	*HeaderInfo	HeaderInfo
	0x42	BigEndian UInt16	LocalPort
	0x48	ListEntry	IpInfo
HeaderInfo			
	0x14	UInt32	IPType

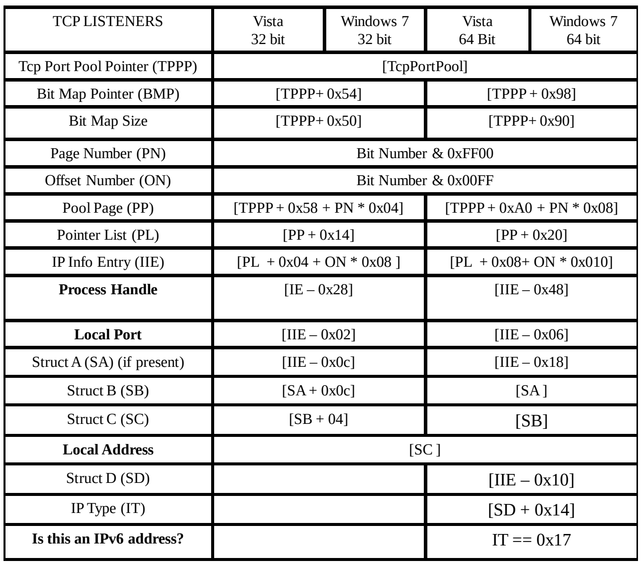
IpAddresses			
	0x00	*LocalIPInfo	LocalIPPtr
LocalIPInfo			
	0x00	BigEndian UInt32	LocalIP

Figure 7: TCP Listeners and UDP Connections Data Structures.



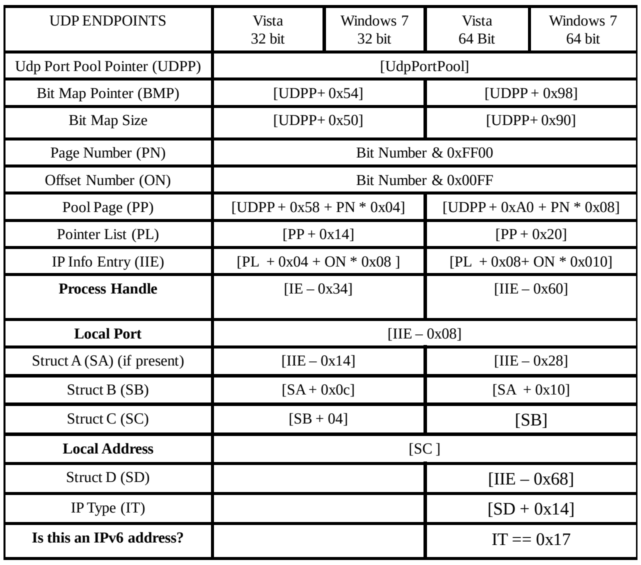
TCP ESTABLISHED CONNECTIONS	Vista 32 bit	Windows 7 32 bit	Vista 64 Bit	Windows 7 64 bit
Partition Table Pointer (PTP)	[PartitionTable]			
Partition Table Entry (PTE)	[PTP + 0x04 + 0x48 * EntryNbr]	[PTP + 0x04 + 0x78 * EntryNbr]	[PTP + 0x08 + 0x40 * EntryNbr]	[PTP + 0x08 + 0x78 * EntryNbr]
Hash Table Pointer (HTP)	[PTE + 0x30]	[PTE+ 0x20]	[PTE + 0x38]	[PTE + 0x20]
Hash Table Element Count	[PTE + 0x08]			
Hash Table Entry (HTE)	[HTP + entry# * 0x08]		[HTP + entry# * 0x10]	
IP Info Entry (IIE)	[HTE]			
Process Handle	[IIE + 0x14c]	[IIE + 0x160]	[IIE + 0x1e0]	[IIE + 0x210]
Local Port	[IIE + 0x18]	[IIE + 0x24]	[IIE + 0x2c]	[IIE + 0x44]
Remote Port	[IIE + 0x1a]	[IIE + 0x26]	[IIE + 0x2e]	[IIE + 0x46]
IP Hdr Info (IHI)	[IIE – 0x08]		[IIE – 0x10]	
IP Address Info (IAI)	[IIE – 0x04]		[IIE – 0x08]	
Struct A (SA)	[IAI]			
Struct B (SB)	[SA+ 0x0c]		[SA + 0x10]	
Local Address	[SB]			
Remote Address	[IAI + 0x08]		[IAI + 0x10]	
IP Type Pointer (ITP)			[IHI + 0x14]	
IP Type (IT)			[ITP]	
Is this an IPv6 address?			IT == 0x17	

Figure 8: TCP Established Connections.



TCP LISTENERS	Vista 32 bit	Windows 7 32 bit	Vista 64 Bit	Windows 7 64 bit
Tcp Port Pool Pointer (TPPP)	[TcpPortPool]			
Bit Map Pointer (BMP)	[TPPP+ 0x54]		[TPPP + 0x98]	
Bit Map Size	[TPPP+ 0x50]		[TPPP+ 0x90]	
Page Number (PN)	Bit Number & 0xFF00			
Offset Number (ON)	Bit Number & 0x00FF			
Pool Page (PP)	[TPPP + 0x58 + PN * 0x04]		[TPPP + 0xA0 + PN * 0x08]	
Pointer List (PL)	[PP + 0x14]		[PP + 0x20]	
IP Info Entry (IIE)	[PL + 0x04 + ON * 0x08]		[PL + 0x08+ ON * 0x010]	
Process Handle	[IE – 0x28]		[IIE – 0x48]	
Local Port	[IIE – 0x02]		[IIE – 0x06]	
Struct A (SA) (if present)	[IIE – 0x0c]		[IIE – 0x18]	
Struct B (SB)	[SA + 0x0c]		[SA]	
Struct C (SC)	[SB + 04]		[SB]	
Local Address	[SC]			
Struct D (SD)			[IIE – 0x10]	
IP Type (IT)			[SD + 0x14]	
Is this an IPv6 address?			IT == 0x17	

Figure 9: TCP Listeners.



UDP ENDPOINTS	Vista 32 bit	Windows 7 32 bit	Vista 64 Bit	Windows 7 64 bit
Udp Port Pool Pointer (UDPP)	[UdpPortPool]			
Bit Map Pointer (BMP)	[UDPP+ 0x54]		[UDPP + 0x98]	
Bit Map Size	[UDPP+ 0x50]		[UDPP+ 0x90]	
Page Number (PN)	Bit Number & 0xFF00			
Offset Number (ON)	Bit Number & 0x00FF			
Pool Page (PP)	[UDPP + 0x58 + PN * 0x04]		[UDPP + 0xA0 + PN * 0x08]	
Pointer List (PL)	[PP + 0x14]		[PP + 0x20]	
IP Info Entry (IIE)	[PL + 0x04 + ON * 0x08]		[PL + 0x08+ ON * 0x010]	
Process Handle	[IE – 0x34]		[IIE – 0x60]	
Local Port	[IIE – 0x08]			
Struct A (SA) (if present)	[IIE – 0x14]		[IIE – 0x28]	
Struct B (SB)	[SA + 0x0c]		[SA + 0x10]	
Struct C (SC)	[SB + 04]		[SB]	
Local Address	[SC]			
Struct D (SD)			[IIE – 0x68]	
IP Type (IT)			[SD + 0x14]	
Is this an IPv6 address?			IT == 0x17	

Figure 10: UDP Endpoints.