Air Force Institute of Technology

AFIT Scholar

AFIT Documents

8-2011

Extracting Forensic Artifacts from Windows O/S Memory

James S. Okolica
Air Force Institute of Technology

Gilbert L. Peterson
Air Force Institute of Technology

Follow this and additional works at: https://scholar.afit.edu/docs

Part of the Computer Sciences Commons, and the Hardware Systems Commons

Recommended Citation
Peterson, G., & Okolica, J. (2011). Extracting Forensic Artifacts from Windows O/S Memory (AFIT/EN/TR-11-02). https://doi.org/10.21236/ADA548397

This Article is brought to you for free and open access by AFIT Scholar. It has been accepted for inclusion in AFIT Documents by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.

EXTRACTING FORENSIC ARTIFACTS FROM WINDOWS O/S MEMORY

TECHNICAL REPORT

James S. Okolica Gilbert L. Peterson

AFIT/EN/TR-11-02

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright Patterson Air Force Base, Ohio

Approved for public release; distribution unlimited

The views expressed in this technical report are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

Table of Contents

Page

List of Figures v

Abstract vi

1. Compiled Memory Analysis Tool 1

2. Using CMAT 3

2.1 Memory Dumps 3

2.2 Hypervisors 5

2.3 Parameters 5

3. Process Flow 6

3.1 cmat.c 6

3.2 cmat_registry.c 9

3.3 file_read.c 9

3.4 pdb_read.c 10

3.5 Extract File Formats 10

Appendix A. Source Code 12

A.1 cmat.c 12

A.2 cmat_registry.c 167

A.3 file_read.c 190

A.4 pdb_read.c 266

A.5 stub.c 335

A.6 cmat.h 336

A.7 cmat_registry.h 338

A.8 file_read.h 341

Page

A.9 memory_structures.h 346

A.10 pdb_read.h 350

A.11 xenaccess.h 371

Bibliography 374

List of Figures

Figure		Page
1.	CMAT Process Flow	2
2.	Top Level Process Flow	6

AFIT/EN/TR-11-02

Abstract

Memory analysis is a rapidly growing area in both digital forensics and cyber situational awareness (SA). Memory provides the most accurate snapshot of what is occurring on a computer at a moment in time. By combining it with event and network logs as well as the files present on the filesystem, an analyst can re-create much of what has occurred and is occuring on a computer. The Compiled Memory Analysis Tool (CMAT) takes either a disk image of memory from a Windows operating system or an interface into a virtual machine running a Windows operating system and extracts forensic artifacts including general system information, loaded system modules, the active processes, the files and registry keys accessed by those processes, the network connections established by the processes, the dynamic link libraries loaded by the processes, and the contents of the Windows clipboard. Operators and investigators can either take these artifacts and analyze them directly or use them as input into more complex cyber SA and digital forensics analysis tools.

EXTRACTING FORENSIC ARTIFACTS FROM WINDOWS O/S MEMORY

1. Compiled Memory Analysis Tool

Recently in the fields of cyber forensics and cyber situational awareness, there has been increased focus on memory analysis [3, 6, 12, 19, 20, 23]. Memory contains the most volatile, and generally the most recent, cyber artifacts. The active processes, current configuration information, users logged in, open network connections and the clipboard all reside in memory. Furthermore, while the operating system executables are stored on disk, while they are executing they exist in memory, making it possible for malware to change their execution without changing what is stored on disk. For all of these reasons, memory analysis is a key part of cyber situational awareness and digital forensics and there is much research [5, 6, 7, 11, 12, 13, 14, 19, 20, 23] and many tools [3, 10, 16, 18, 8].

The Compile Memory Analysis Tool (CMAT) is a self-contained memory analysis tool that analyzes a Windows O/S memory dump and extracts information about the operating system and the running processes. The memory is accessed either as a previously generated memory dump or interactively via the Xen hypervisor [24] using Xen Access [1].

As shown in Figure 1, CMAT uses signature matching to locate processes and configuration information. The process information CMAT retrieves includes the name of the process, the user who created it, the open files and registry keys, any open network connections, and any clipboard information [14]. In addition, CMAT programmatically determines the version of Windows (e.g., XP, Vista, Windows 7), whether it is 32 or 64 bit and whether physical address extensions (PAE) are enabled [12]. Finally, unlike most other memory analysis tools, CMAT achieves Windows version agnosticism by downloading program database (PDB) files from the Microsoft Symbol Server and then using these PDB files to create its signatures and retrieve the location of application specficic symbols [12].

In addition to providing a user interface for the analyst that allows process and configuration to be viewed interactively, CMAT also provides the ability to dump forensic artifacts to file. To highlight the fact that the optimal features to extract are unknown,

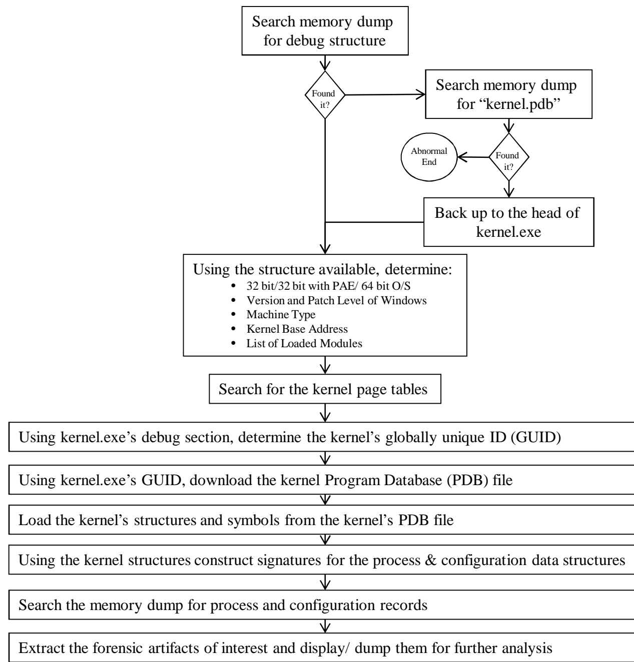

Figure 1 CMAT Process Flow

CMAT has three different dump routines. It is expected that individuals who use CMAT will likely develop their own, using the existing dump functions as templates.

The remainder of this report begins by providing details on how to use CMAT “as-is” and then describing the technical details behind CMAT. The report concludes with a complete list of the source code to aid in future modifications.

2. Using CMAT

CMAT has two distinct methods for interacting with Windows' memory. It can either interact with a memory dump or it can interact with a Xen hypervisor.

2.1 Memory Dumps

In this case, CMAT takes as input a memory dump (generated by utilities like win32dd and win64dd [21]). In this instance, the command to run cmat is:

C:\> cmat <dumpfile name>

If a swap file or pagefile is also available (copied using utilities like hobocopy [2], the command to run cmat is:

C:\> cmat <dumpfile name> -pagefile <pagefile.sys name>

Although not requiring additional parameters, if CMAT encounters memory-mapped files in the memory dump (e.g., \windows\system32\config\software), it will dynamically check if a copy of the file exists off of its current directory).

Two of the most common parameters are the **-data** parameter which allows the user to specify the directory that the memory dump, PDB files, and dump files are in or will be placed in. The default value is **.**. Observe that there is a trailing backslash. If CMAT is run on a Linux system, the **-data** parameter must always be specified and if the directory is the current directory, **./** must be specified for **-data**.

CMAT builds its signatures and locates its symbols in the memory dump through the use of program database (PDB) files that are built when the kernels, drivers, and dynamic libraries are built. These PDB files are stored on the Microsoft symbol server. The following PDB files are required by CMAT: tcpip.pdb, user32.pdb, win32k.pdb, and one of the following depending on whether physical address extensions are enabled and if there is more than one processor: ntoskrnl.pdb, ntkrnlpa.pdb, ntkrnlmp.pdb and ntkrpamp.pdb. If the user already has copies of the correct version of the pdb files, they can place them in the data directory. If not, if the user specifies the **-getpdb** parameter, CMAT will

programatically determine the current version of the PDB files and download them from the Microsoft Symbol Server. In the case of win32k.pdb and user32.pdb, it may be necessary to have the executables (win32k.sys and user32.dll respectively) located in the data directory, since in some cases, the executable were compiled without debug information.

While CMAT extracts forensic artifacts, it does not perform any analysis on the features it extracts. Instead, CMAT provides the ability to dump the features. As currently implemented, the feature file specified by the **-feature** parameter is actually a base name, as CMAT generates six files and appends the time and a numeric extension to whatever the parameter's value is. In this instance, the command to run cmat is:

```
C:\> cmat <dumpfile name> -feature <feature file>
```

There are three different memory models commonly used by Windows (32 bit, 32 bit with physical address extensions (PAE) and 64 bit). With the 32 bit memory model, 4 GB of virtual memory is addressable; with 32 bit with PAE, 64 GB of virtual memory is addressable and with 64 bit, 280 TB of virtual memory are addressable. However, since there is often significantly less physical memory, Windows provides a mechanism to convert from virtual addresses to the physical addresses in memory. These virtual to physical (VTOP) tables allow all references to memory to be in virtual addresses. The only physical address that Windows stores is the pointer to these VTOP tables. By default, CMAT provides the analyst with the ability to convert a virtual address to a physical one. However, since these tables are one way, converting from physical to virtual (PTOV) involves an exhaustive search of the leaf nodes of these tables. To enable this PTOV functionality, CMAT has a **-ptov** parameter. In this case, during initialization, CMAT uses the page tables for each process to determine all of the possible virtual addresses and processes that the physical address may be mapped to. Then, if the analyst chooses to perform this PTOV functinoality from the user interface, the response is immediate.

Finally, two parameters, **-dumpstruct** and **-dumpsymbol**, direct CMAT to dump the structures or symbols of the indicated PDB file. The format for this is:

```
C:\> cmat -dumpstruct <PDB filename> -dumpsymbol <PDB filename>
```

To dump the structures or symbols to a file, the user must redirect output.

2.2 Hypervisors

In the second instance, CMAT interacts with the Xen hypervisor through XenAccess. Most of the above parameters are still valid. The only addition is the terminating parameter **-virt_live**. The value for this parameter is the domain id of the hypervisor. In this instance, the command to run cmat is:

C:\> cmat -data <data directory> -virt_live <domain id>

Observe that since Xen is run under a Linux operating system, the **-data** parameter will always be present (if only to specify a forward slash instead of a back slash).

2.3 Parameters

In summary, the following parameters are available to CMAT:

- **-data <data directory>** - the data directory where the dump file, page files, PDB files, and features do or will reside.
- **-getpdb** - download the PDB files from the Microsoft Symbol Server.
- **-feature <feature file>** - the name of the feature file(s) to dump CMAT features into.
- **-pagefile <pagefile name>** - the name of the pagefile
- **-ptov** - load the physical to virtual tables for later user analysis
- **-dumpstruct <PDB file>** - dump the structures from the specified PDB file and then exist.
- **-dumpsymbol <PDB file>** - dump the symbols from the specified PDB file and then exist.
- **-virt_live <domain id>** - if run against a Xen hypervisor, indicates the domain id of the hypervisor of interest.

3. Process Flow

CMAT is a modularized program with four primary sections. **cmat.c** is the main program. It does the scanning of memory for initial operating system (O/S) identification, process records, configuration records, network connections, and clipboard contents. It also provides the user interface and the dump routines. **cmat_registry.c** supplements **cmat** with the routines for parsing the configuration files known as the registry. **file_read.c** includes the memory interfaces. It maps virtual addresses to physical addresses and then uses those physical addresses to access the requested memory either in the memory dump or in the hypervisor. Finally, **pdb_read.c** abstracts memory accesses by providing an O/S agnostic mechanic for accessing kernel objects.

3.1 cmat.c

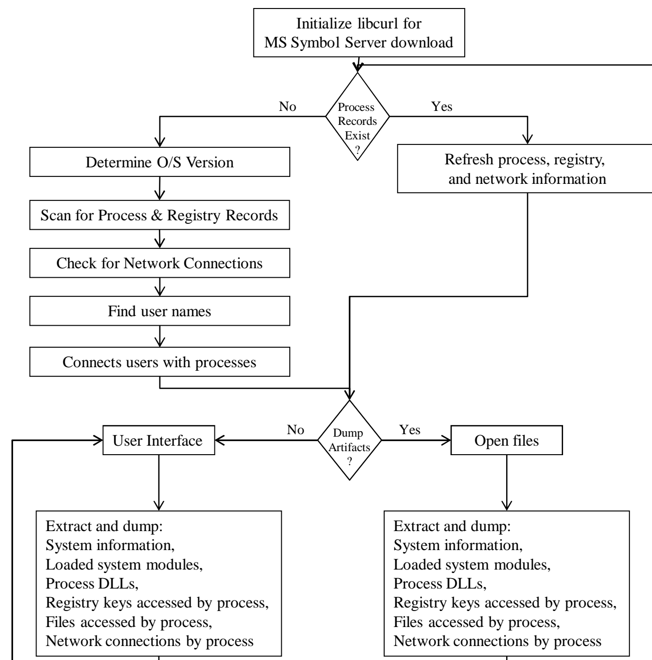

Figure 2 Top Level Process Flow

As shown in Figure 2, CMAT begins by initializing `libcurl` to interact with the Microsoft Symbol Server, parsing parameters, and then repeatedly calling **stage_1_menu** which performs the primary functions of CMAT. **stage_1_ menu** checks if there are cur-

rently any process records. If not, it first determines what the operating system is and then scans the dump for process and registry records. It then looks for active network connections and determines the user name of the user associated with each process. If there are process records and CMAT is accessing memory through the hypervisor, CMAT refreshes its information. Once, CMAT has this information it either dumps it or provides a user interface to allow the analyst to view it. CMAT then either ends if it was run with parameters to send the artifacts to a file or it repeats this process until either the user exits via the user interface, or terminates the program if it was run with parameters to send the artifacts to file for a hypervisor. The remainder of this section presents how CMAT performs these different functions but will not go into a procedure by procedure breakdown.

To identify the operating system, CMAT begins by scanning memory for the _DBGKD_DEBUG_DATA_HEADER64, _KDDEBUGGER_DATA64, and _DBGKD_GET_VERSION64 records [9]. If CMAT is able to find these records, it uses the information in them to determine what version of Windows is running and whether it is a 32 bit, 32 bit with physical address extensions (PAE) enabled, or a 64 bit implementation. It also uses the information in these records to determine the virtual address of the kernel.

If it is not able to find the debug records, it scans memory for the kernel portable executable. It does this by searching for **ntoskrnl.pdb**, **ntkrnlpa.pdb**, **ntkrnlmp.pdb** and **ntkrpamp.pdb**. When it finds one of these it assumes it is in the debug section of the kernel executable and then uses the portable executable (PE) layout [4] to backtrack to the beginning of the PE and validate its assumption. If it at the head of the PE, it then uses the information in the header to determine what version of Windows is running and whether it is a 32 bit, 32 bit with physical address extensions (PAE) enabled, or a 64 bit implementation.

Next, CMAT finds the tables used to convert virtual addresses to physical addresses. The top-level table for 32 bit without PAE is the page directory (PDE) table; for 32 bit wiht PAE, it is the page directory pointer (PDP) table, and for 64 bit, it is the page map level (PML) table. In all cases, one of the entries in these tables points back to the head of the table. By leveraging this self-referencing feature, one of these tables is found. It is then

tested to determine if the virtual address of the kernel PE maps to the correct physical. If so, this table is then used as the kernel virtual to physical (VTOP) table. If not, the process is repeated until a valid kernel VTOP table is found.

Using the VTOP tables, the kernel PE is located and a globally unique identifier (GUID) is extracted from the debug section. With this GUID, CMAT downloads the kernel's program data base (PDB) file from the Microsoft Symbol Server and extracts the symbols and structures. With these O/S-specific structures, CMATis able to construct signatures for EPROCESS and CMHIVE records. Then, using these signatures, memory is scanned and the process and registry hives are discovered and extracted. CMAT then generates a record for each process and for each hive, populating them with relevant forensic artifacts.

Next, the PE for tcpip.sys is located in memory. Then using its GUID, its PDB file is downloaded from the Microsoft Symbol Server and its symbols and structures are extracted. CMAT then uses the symbols to locate the data structures holding the open network connections and create linked lists on the previously generated process records detailing these network connections.

As the last step of data loading, the users with accounts are added to a list of users (discussed in more detail in cmat_registry.c) and then connected to the processes they created.

At this point, all initial data is loaded and the list of processes is provided to the user. If CMAT was started interactively, the user is prompted to either select an individual process to see additional information, view the system information, view the loaded system modules (found by traversing the PsLoadedModuleList found when CMAT determined the operating system), view the contents of the Windows clipboard, view a particular registry value (discussed in more detail in cmat_registry.c), convert virtual addresses to physical addresses (discussed in more detail in file_read.c) , or convert physical addresses to virtual addresses (discussed in more detail in file_read.c). If CMAT was started with the parameter to dump to batch, the system information, loaded system modules, and detailed process information are dumped into multiple different files.

Additional process data includes dynamic link libraries (DLLs) loaded, registry keys accessed, files opened, network connections, starting parameters, and clipboard contents. The DLLs loaded for the process are located in the Process Environment Block under the loader data and the registry keys and files opened are found in the object table for the process. The clipboard contents are found using the same mechanic used for finding open network connections. The difference is the driver/DLLs used (win32k.sys and user32.dll) and the symbols where the clipboard is located.

3.2 cmat_registry.c

Cmat_registry is a collection of functions used to access the Windows registry in memory. The registry is structured as a file system and so traversing involves stepping down through multiple "folders" until one arrives at the folder and item of interest. For instance, the users with an account on a box is stored in *REGISTRY/MACHINE/SOFTWARE/ Microsoft/WindowsNT/CurrentVersion/Profile-List*. Each user has keys below this for their path (**Profile Image Path**) and token id (**SID**). To retrieve the users with accounts on a system, CMAT locates the ProfileList in memory and then extracts the name for each user from the end of their path. It then connects the users with the processes by linking the SIDs and the tokens found in the process records. At each level of the file structure, there are lists of key nodes that are traversed to find a match for the string of interest (e.g., "Microsoft" or "Windows NT"). Once found there is a key that is parsed to find the memory location where the key value is found.

3.3 file_read.c

Most of file_read is concerned with translating virtual addresses to physical addresses. For details on this, refer to Russinovich's Windows Internals [15]. Beyond the virtual to physical conversion, file_read allows CMAT to function transparently with either memory dumps or a Xen hypervisor. The only functions that distinguish between them is **figetc** which retrieves a single character from the current location of memory, **fimove** which adjusts the pointer to the current location of memory, and **get_mem_size** which returns the total size of memory. The rest of file_read is either concerned with translating physical

addresses to virtual addresses or retrieving different amounts of memory (1, 2, 4, or 8 bytes or strings) either in little endian or big endian form. To convert physical addresses to virtual addresses, each processes VTOP table is loaded into a linked list. Then during initialization each of these tables is harvested to form a master table of physical to virtual addresses.

3.4 pdb_read.c

What distinguishes CMAT from many other memory analysis tools is that it is MS Windows O/S-agnostic. It achieves this through the use of PDB files [22, 17]. Over ninety percent of the memory accesses in CMAT are done through PDB calls. For instance, the head of an EPROCESS record is found during the memory scan. **cmat.c** is retrieving Directory Table Base which is the Process Environment Block of the EPROCESS record. **cmat.c** requests "_EPROCESS.Pcb. DirectoryTableBase"; **pdb_read.c** uses the kernel's PDB file and determines the offset of the process control block within the EPROCESS record and the offset of the directory table base within the PCB record; **pdb_read.c** than sends a request to **file_read.c** for the specific memory address using the head of the EPROCESS as a base.

3.5 Extract File Formats

If cmat is run with the **-feature <feature file>** flag, cmat dumps the forensic artifacts into six tab-delimited files. The feature file is slightly mis-labeled since it is only the beginning of the feature file names. For instance, **cmat -data ./ -feature dumps/xx** will dump the feature files into the dumps folder off of the current directory and each feature file will begin with the letters "xx". The suffix of the feature files begins with a 16 character date/time string of the format "YYYYMMDDHHMMSS" followed by a single digit (1 - 6) followed by ".txt". The single digit signifies the contents of the file:

- 1 - Process IDs, names, and users Data items: Process ID (8 byte int), Process name (string), User name (string)

- 2 - Network information by process Data items: Process ID (int), Local IP Address/Port (20 byte string), Remote IP Address/ Port (20 byte string), Protocol (12 byte string), 1 if the record is IPv6 and 0 otherwise.
- 3 - Loaded modules by process Data items: Process ID (int), Base DLL Name (string), Full DLL Name (string)
- 4 - Files accessed by process Data items: Process ID (8 byte int), Memory address of file (8 byte int), (xyz) where x = 'R' if readable '-' otherwise, y = 'W' if writeable, '-' otherwise, z = 'D' if deleteable, '-' otherwise).
- 5 Registry keys accessed by process Data items: Process ID (8 byte int), Memory addretss of registry key (8 byte int), Hive/Cell name (string)
- 6 System Loaded modules Data items: Driver Name (string), Memory address of driver (8 byte int)

In addition, a dump file is generated with the same date/time suffix but no digit and an extension of ".dmp". In the case of the hypervisor dumps, the feature files 1,2,3,4, and 6 are generated every time while the dump and the feature file 5 (registry keys)are only generated once every 30 minutes.

A.1 cmat.c

```
/*****************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)          *
* Developer  : Jimmy Okolica                                     *
* Date       : 15-Aug-2011                                       *
*                                                                *
*****************************************************************/

/*****************************************************************
 * Copyright 2011 James Okolica Licensed under the Educational     *
 * Community License, Version 2.0 (the "License"); you may not use *
 * this file except in compliance with the License. You may obtain *
 * a copy of the License at                                        *
 *                                                                 *
 * http://www.osedu.org/licenses/ECL-2.0                           *
 *                                                                 *
 * Unless required by applicable law or agreed to in writing,      *
 * software distributed under the License is distributed on an     *
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,     *
 * either express or implied. See the License for the specific     *
 * language governing permissions and limitations under the License.*
 *****************************************************************/

/*****************************************************************
* Source code:                                                   *
* cmat.c        : main module containing user interface          *
* (cmat.h)        searches through memory dump to determine O/S  *
*                 searches for process & registry signatures     *
*                 instantiates processes                         *
*                 searches for and instantiates network traffic  *
*                 searches for and instantiates clipboard data   *
* cmat_registry.c:instantiates registry information              *
* (cmat_registry.h)                                              *
* file_read.c   : provides a consistent interface to access the  *
* (file_read.h)   memory dump.  contains 3 sets of routines:     *
*                 memgrab_XX_YY: grabs XX bits in Big/Little Endian *
*                               using a physical memory address  *
*                 memgrabke_XX_YY: grabs XX bits in Big/Little Endn *
*                               using a virtual kernel address   *
*                 memgrabp_XX_YY: grabs XX bits in Big/Little Endn  *
*                               using a virtual process address  *
* pdb_read.c    : provides a consistent interface to access Windows *
* (pdb_read.h)    OS structures in a windows-agnostic way        *
```

```
*                 determines the GUID for a O/S and retrieves the   *
*                   appropriate PDB file                            *
*                 parses the PDB file for section locations,        *
*                    symbols, and structures                        *
*                 sits above file_read.c to make retrieving windows *
*                    structures o/s independent                     *
* memory_structures.h:  contains global structures for program      *
* stub.c            :  stub to allow compilation without xenaccess *
* (xenaccess.h)                                                     *
*                                                                   *
* External Libraries:                                               *
* libcurl.lib   : allows pdb_read to retrieve pdb files from        *
* (libcurl.dll)                   Microsoft Symbol Server           *
*                                                                   *
* Compiling cmat:                                                   *
*    compile cmat with libcurl.lib                                  *
*    run cmat with libcurl.dll in the local directory               *
*                                                                   *
* Running cmat:                                                     *
*    cmat <memory dump file> :                                      *
*                              parses memory dump file and then has *
*                              user interface to examine processes  *
*    cmat <memory dump file> <page file>                            *
*                              parses memory dump file using        *
*                              page file as needed and then has     *
*                              user interface to examine processes  *
*    cmat -feature <output file> <memory dump file> :               *
*                              parses memory dump file and then     *
*                              dumps summary process information to *
*                              output file                          *
*    cmat <memory dump file> <page file>                            *
*                              parses memory dump file using        *
*                              page file as needed and then dumps   *
*                              summary process information to output*
*                              file                                 *
*                                                                   *
*    -feature <output file>                                         *
*                   Current Features include:                       *
*                      InheritedFromUniqueProcessId, CreateTime,    *
*                      PeakVirtualSize, ReadOperationCount,         *
*                      WriteOperationCount, OtherOperationCount,    *
*                      PriorityClass, KernelTime, UserTime,         *
*                      BasePriority                                 *
*                                                                   *
* Generating a memory dump:                                         *
```

```
*     dumpit.bat : creates a copy of pagefile.sys in local directory*
*                  generates a memory dump using win32dd.exe (may be*
*                      modified to use win64dd on 64-bit machine) in*
*                      the local directory called win32_image.dmp   *
*                  creates a copy of the software config file in    *
*                      .\windows\system32\config                    *
*                      (this last step is necessary being Windows   *
*                       uses memory mapped file rather than bringing*
*                       all configuration file into memory at once) *
*    needed programs: hobocopy.exe, win32dd.exe, win32dd.sys,       *
*                     win64dd.exe, win64dd.sys                      *
*                                                                   *
*********************************************************************
*********************************************************************/

#include "cmat.h"
#include <curl/curl.h>
#include <curl/types.h>
#include <curl/easy.h>
#include <math.h>
#include <time.h>

/********************/
/* GLOBAL VARIABLES */
/********************/
short MYDEBUG =  0;   // Boolean flag to determine if debug print statements
 // should print
short TCPDEBUG = 0;   // Boolean flag for debugging load_tcpip()

FILE *feature_file;   // file ptr to the dump file
char *feature_file_name;                // Name of the dump file

short FEATURE = 0;                     // True if info is being dumped
// BEGIN -- CMAT_V
static xa_instance_t xai_live;
short LIVE = 0;
uint32_t dom = 0;
char * dom_char;
short BENCH_DEBUG = 0;
// END -- CMAT_V

short NOUDP = 0;        // True if UDP information should be included in the dump
short GET_PDB = 0;      // True if pdb files should be retrieved from the MS
   // Symbol Server
short PTOV = 0;         // True if ptov infor needs to be gen'd for physical to
```

```
                        // virtual conversion
char *DATADIRECTORY;    // The data directory where the input;s stored and
                        // output's written
short AUTO_REFRESH = 0;

process_list *proc_list = NULL;           // List of processes
thread_list *thr_list = NULL;             // List of threads -- not currently
                                          //populated
process_list *first_valid_proc = NULL;    // this is the first good process

dbgkd_debug_data_t *debug_data = NULL;    // Meta-information on dump (e.g. OS
                                          // Version, Kernel loc, etc.
win32k_t *win32k = NULL;                  // win32k symbol information
user_list_type *user_list = NULL;         // List of users with accounts on machine

time_t start_time;

pdb_type_t *kernel_pdb = NULL;
pdb_type_t *tcpip_pdb = NULL;
pdb_type_t *user32_pdb = NULL;
pdb_type_t *win32k_pdb = NULL;
/********************************************************************************/

/***********************/
/* FUNCTION PROTOTYPES */
/***********************/
void usage ( char *binname );
int stage_1_menu();
int stage_2_menu(int proc_num);

/* File IO */
int open_image_file( char *filename);
int open_feature_file( char *filename);
int cmat_dump_virtual(char *vm_id, char *file_name); // CMAT_V

/* Process Management */
void add_new_process (uint64_t eprocess_location);
void clear_lists(process_list *old_process, thread_list *old_thread,
                 user_list_type *old_users);
int load_debug_data();
int load_tcpip_data();
int load_info();
void load_user_process();
int show_system_info();
void show_proc_env(process_list *sel_proc);
```

```
void show_proc_dll(process_list *sel_proc, FILE *fptr);
uint64_t show_proc_hand(process_list *sel_proc, char *wtype, FILE *fptr);
uint64_t disp_obj_details(process_list *sel_proc, uint64_t addr, char *wtype,
                          FILE *fptr);
int show_clipboard(process_list *sel_proc);
int show_clipboardk(process_list *sel_proc);

uint32_t min_u32(uint32_t x, uint32_t y);
int isprintable(unsigned char k);
int bytecmp(unsigned char *str1, unsigned char *str2, int wlength);
char *strtoupper(char *upper);

char *get_sid(process_list *sel_proc);

int dump_memory();
int dump_memory2(uint8_t do_full);
int dump_memory3();

/* Thread Management */
void walk_thread_list (process_list *ptmp, uint32_t blink, uint32_t flink);
void add_new_thread (unsigned long ethread_location);
//void grab_ethread (process_list *ptmp, thread_list *tmp);

// BEGIN -- CMAT_V
/*List clearing funcitons*/
void clear_process_list(process_list *old_process);
void clear_thread_list(thread_list *old_thread);
void clear_user_list(user_list_type *old_user);
void clear_socket_list(socket_list_type *old_socket);
// END -- CMAT_V

/* Memory Management */
void grab_eprocess (process_list *tmp);
void kegrab_eprocess (process_list *tmp);
void grab_peb (process_list *tmp);

typedef struct session_type {

    uint32_t session_id ;
    struct session_type * next;

}
session_t;
```

```
/*********************************************************************************/

/******************************************************************************/
/* USER INTERFACE - MAIN PROCEDURES, MENU PROCEDURES, AND USAGE INFORMATION */
/******************************************************************************/

/*****************/
/* Main Procedure */
/*****************/
int main( int argc, char**argv )
{

    int i, offset;

    char *pdb_url, *wpdb_url;                // Stores the url needed to get the
                                             // MS Symbol Server to d/l pdb
    char *fname, *xname;
    char *wstring;

    /* INITIALIZE */

    // Initialize curl functions -- allows accessing websites (e.g., Microsoft
    // Symbol Server)
    curl_global_init(CURL_GLOBAL_ALL);

    if ( argc == 1) {
        usage(argv[0]); exit(1);
    }


    // if dump features, open the dump file
    FEATURE = 0;
    NOUDP = 0;
    GET_PDB = 0;
    PTOV = 0;
    DATADIRECTORY = malloc(sizeof(char) * 3);
    strcpy_s(DATADIRECTORY,3,".\\");
    DATADIRECTORY[2] = 0;
    offset = 1;
    while (offset < argc - 1) {

        if (!strcmp("-FEATURE",strtoupper(argv[offset]) ) ) {

            FEATURE = 1;
            feature_file_name = malloc(sizeof(char) * (strlen(argv[offset + 1])
```

```
                        + strlen(DATADIRECTORY) + 1));
        strcpy_s(feature_file_name,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(feature_file_name,strlen(DATADIRECTORY) +
                        strlen(argv[offset + 1]) + 1,argv[offset + 1]);
        if ( (offset + 2 == argc) || ( open_feature_file(feature_file_name ))) {
            usage(argv[0]); exit(1);
        }

        fclose(feature_file);
        offset += 2;
        continue;

    }

    if (!strcmp(strtoupper(argv[offset]),"-NOUDP")) {

        NOUDP = 1;
        offset++;
        continue;

    }

    if (!strcmp(strtoupper(argv[offset]),"-PAGEFILE")) {

        i = offset + 1;
        while ((i + 1 < argc) && (argv[i][0] != '-')) {

            fname = malloc(sizeof(char) * (strlen(argv[i]) +
                    strlen(DATADIRECTORY) + 1));
            strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
            strcat_s(fname,strlen(DATADIRECTORY) + strlen(argv[i]) +
                      1,argv[i]);
            if (open_page_file(fname)) {
                usage(argv[0]); exit(1);
            }

            i++;
            free(fname);

        }

        offset = i;
        continue;

    }
```

```
if (!strcmp(strtoupper(argv[offset]),"-GETPDB")) {

    GET_PDB = 1;
    offset++;
    continue;

}

if (!strcmp(strtoupper(argv[offset]),"-PTOV")) {

    PTOV = 1;
    offset++;
    continue;

}

if (!strcmp(strtoupper(argv[offset]),"-DATA")) {

    if (offset + 2 == argc) {
        usage(argv[0]); exit(1);
    }

    DATADIRECTORY = malloc(sizeof(char) * (strlen(argv[offset + 1]) + 1));
    strcpy_s(DATADIRECTORY, strlen(argv[offset + 1]) + 1,argv[offset + 1]);
    offset += 2;
    continue;

}

if (!strcmp(strtoupper(argv[offset]),"-DUMPSYMBOL")) {

    i = offset + 1;
    while ((i + 1 < argc) && (argv[i][0] != '-')) {

        fname = malloc(sizeof(char) * (strlen(argv[i]) +
                strlen(DATADIRECTORY) + 1));
        strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(fname,strlen(DATADIRECTORY) + strlen(argv[i]) +
                 1,argv[i]);
        if (GET_PDB) {

            pdb_url = get_pdb_url_disk(fname,PDB_RET_URL,0);
            wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
            strcpy_s(wpdb_url,strlen(pdb_url) + 1, pdb_url);
```

```
                wstring = get_pdb_url_disk(fname,PDB_RET_NAME,0);
                xname = download_pdb_file(wstring, wpdb_url, DATADIRECTORY);
                parse_pdb_file(xname,PDB_DUMP_SYM);
                free(wstring);
                free(pdb_url);
                free(wpdb_url);

            }
            else {

                parse_pdb_file(fname,PDB_DUMP_SYM);

            }

            free(fname);
            if (xname != NULL) free(xname);
            i++;

        }

        exit(0);

    }

    if (!strcmp(strtoupper(argv[offset]),"-DUMPSTRUCT")) {

        i = offset + 1;
        while ((i + 1 < argc) && (argv[i][0] != '-')) {

            fname = malloc(sizeof(char) * (strlen(argv[i]) +
                    strlen(DATADIRECTORY) + 1));
            strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
            strcat_s(fname,strlen(DATADIRECTORY) + strlen(argv[i]) + 1,argv[i]);
            if (GET_PDB) {

                pdb_url = get_pdb_url_disk(fname,PDB_RET_URL,0);
                wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
                strcpy_s(wpdb_url,strlen(pdb_url) + 1, pdb_url);
                wstring = get_pdb_url_disk(fname,PDB_RET_NAME,0);
                xname = download_pdb_file(wstring, wpdb_url,DATADIRECTORY);
                parse_pdb_file(xname,PDB_DUMP_STRUCT);
                free(wstring);
                free(pdb_url);
                free(wpdb_url);
```

```
        }
        else {

            parse_pdb_file(fname,PDB_DUMP_STRUCT);

        }

        free(fname);
        if (xname != NULL) free(xname);
        i++;

    }

    exit(0);

}


// BEGIN -- CMAT_V

//if virtualized live memory method
if (!strcmp(argv[offset],"-virt_live") ){

    LIVE = 1;
    AUTO_REFRESH = 1;
    BENCH_DEBUG = 0;
    /* this is the domain ID that we are looking at */
    dom = atoi(argv[offset + 1]);
    dom_char = argv[offset + 1];
    init_instance(dom, &xai_live);
    offset += 2;
    continue;

}


//if running repeat benchmarks in virtualized live memory method
if (!strcmp(argv[offset],"-virt_live_bench") ){

    LIVE = 1;
    BENCH_DEBUG = 1;
    /* this is the domain ID that we are looking at */
    dom_char = argv[offset + 1];
    dom = atoi(argv[offset + 1]);
    init_instance(dom, &xai_live);
```

```
            offset += 2;
            continue;

        }


        // if running in cmat benchmark repeat mode
        if (!strcmp(argv[offset],"-cmat_bench") ){

            BENCH_DEBUG = 1;
            offset++;
            continue;

        }


        // if virtualized memory dump method
        if (!strcmp(argv[offset],"-virt_dump") ){

            if (cmat_dump_virtual(argv[offset + 1], argv[offset + 2])){
                usage(argv[0]); exit(1);
            }

            offset += 3;
            continue;

        }

        // END -- CMAT_V

    }

    // Initialize data structure for meta-information (including object type table)
    debug_data = malloc(sizeof *debug_data);
    debug_data->objTypeTable = (uint64_t *) malloc(sizeof(uint64_t) * 256);
    for (i = 0; i < 255; i++) debug_data->objTypeTable[i] = 0;
    // open the image file -- file containing the memory dump
    if(!LIVE){
        // CMAT_V
        fname = malloc(sizeof(char) * (strlen(argv[offset]) +
                strlen(DATADIRECTORY) + 1));
        strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(fname,strlen(DATADIRECTORY) + strlen(argv[offset]) + 1,argv[offset]);
        if ( open_image_file(fname)) {
            usage(argv[0]); exit(1);
```

```
    }

    debug_data->MemDumpFile = fname;

}
// CMAT_V
/*
// open the page files -- file(s) containing the swap file (these may or may
// not be present)
if (argc - (offset + 2) > 0) {

    for (i = 0; i < argc - (offset + 2); i++) {

        if (open_page_file(argv[offset + 2 + i])) {
            usage(argv[0]); exit(1);
        }


    }


}

*/

printf ("\n\n\n");
/* MAIN MENU */

// run the menu... continue until it returns > 0
time(&start_time);
while (stage_1_menu()) continue;

// Clean up website access utilities
curl_global_cleanup();

/* cleanup any memory associated with the XenAccess instance */
if(LIVE){
    xa_destroy(&xai_live);
}
// CMAT_V

free(DATADIRECTORY);
free(feature_file_name);
free(debug_data);
return 0;
```

```
}


/**************************/
/* Main Menu - first level */
/**************************/
int stage_1_menu()
{

    char decision[5] = {
        0
    }
    ;
    char sva[24] = {
        0
    }
    ;
    char reg_key[200] = {
        0
    }
    ;
    char hive_name[80] = {
        0
    }
    ;
    uint64_t xva;
    int i,ctr,dummy;

    process_list *tmp;
    user_list_type *wuser;
    int proc_counter;

    ptov_ret_list *wptov;

    session_t *whead, *wptr1;
    uint8_t gotit;

    unicode_string wdrivername;              // Driver found by traversing
                                             // PsLoadedModuleList
    uint64_t wptr;                           // Pointer to traverse
                                             // PsLoadedModuleList
    uint64_t wend;                           // Endpoint for traversing
                                             // PsLoadedModuleList
    uint64_t wimage_base;                    // Linear address of image base
```

```
                                            // of driver

   uint64_t bend_address, fend_address, next_address, prev_address,
            curr_address, pid;
   uint8_t first_pass, gotone, val_change;
   uint16_t proc_count;
   uint64_t wUniqueProcessId, wToken, wObjectTable, wDTB, *wDTBA;
   uint32_t woffset;

   char *wstring;
   process_list *old, *valid_proc;
   printf ("Compiled Memory Analysis Tool Jimmy Okolica, 2009\n");


   //If this is the first pass through the function, load meta information,
//kernel structures, processes, and tcpip info
   first_pass  = 0;
   if ( proc_list == NULL ) {

       load_debug_data();        // Meta-information about dump
       load_info();                // Process and Registry information
       load_tcpip_data();        // Network traffic data
       user_list = get_users();    // List of users with accounts
       load_user_process();        // Link users with processes
       if (PTOV) ptov_init();    // For debugging purposes
       first_pass = 1;

   }


   // BEGIN -- CMAT_V
   if (LIVE) system("clear");

   printf("\n\n\n\n/***************SYSTEM INFORMATION***************/");


   //print system info here

   printf("\n\n%-25s", "Memory Source:");
   if(LIVE){

       printf("Live Virtual Machine\n");
       printf("%-25s%x\n","VM ID:", xai_live.m.xen.domain_id);
       printf("%-25s%s\n","VM Name: ", xai_live.image_type);
       printf("%-25s%d\n","Xen Version: ", xai_live.m.xen.xen_version);
```

```
    printf("%-25s%.2f MB\n","Memory Size: ", ((float)get_mem_size()/1048576) );
    printf("%-25s%s\n","HVM: ", xai_live.hvm?"ENABLED":"DISABLED");


}

else{

    printf("Memory Dump File\n");
    printf("%-25s%s\n", "File Location:", debug_data->MemDumpFile);

}


printf("%-25s%s\n", "Processor:", debug_data->MachineString);
printf("%-25s%s Ver %i.%i (build:%i)\n","Operating System:",
                debug_data->OSVersionString, debug_data->MajorOSVersion,
                debug_data->MinorOSVersion, debug_data->MinorVersion);
printf("%-25s%s\n", "PAE",(debug_data->PaeEnabled?"PAE Enabled":"PAE Disabled"));
printf("%-25s%llx\n", "Kernel Base:", debug_data->KernBase);
printf("%-25s%llx\n", "Page Directory Base:", debug_data->kDTB);
// END -- CMAT_V


if ( (AUTO_REFRESH) && (!first_pass)) {

    printf("automatically refreshing....\n");
    tmp = proc_list;
    proc_count = 0;
    while (tmp != NULL) {

        if (tmp->eprocess_in_link_list == 1) tmp->checkmark = 0;
        proc_count++;
        tmp = tmp->next;

    }

    tmp = proc_list;
    valid_proc = NULL;
    // Look for a process that's still has the same info in it for info that
    // can't change (i.e., ProcessID, Directory Table)
    // If the info has changed, the record is most likely no longer an
    // eprocess record
    while (tmp != NULL) {
```

```
    wUniqueProcessId        = get_pdb_type  (
           "_EPROCESS.UniqueProcessId",  tmp->proc_eprocess_location);
    wObjectTable            = get_pdb_type  (
           "_EPROCESS.ObjectTable",  tmp->proc_eprocess_location);
    wToken                  = get_pdb_type  (
           "_EPROCESS.Token",  tmp->proc_eprocess_location);
    wDTBA            = get_pdb_type_array  (
           "_EPROCESS.Pcb.DirectoryTableBase",  tmp->proc_eprocess_location);
    wDTB = wDTBA[0];
    free(wDTBA);
    if ( (wUniqueProcessId != tmp->proc_eprocess.UniqueProcessId) ||
                 (wObjectTable != tmp->proc_eprocess.ObjectTable) ||
                 (wToken != tmp->proc_eprocess.Token)             ||
                 (wDTB != tmp->proc_eprocess.Pcb.DirectoryTableBase) ) {

        printf("  deleting pid %i due to changed info\n",
               tmp->proc_eprocess.UniqueProcessId);
        if (tmp->next != NULL) tmp->next->prev = tmp->prev;
        if (tmp->prev != NULL) tmp->prev->next = tmp->next;
        if (proc_list == tmp) proc_list = tmp->next;
        old = tmp;
        tmp = tmp->next;
        free(old->proc_eprocess.ImageFileName2);
        free(old);
        continue;

    }

    // the process has to also be in the linked list;
    // otherwise, we can't find the other processes
    if ( (tmp->eprocess_in_link_list == 1) &&
         (tmp->proc_eprocess.UniqueProcessId != 0)) {

        tmp->checkmark = 1;
        valid_proc = tmp;

    }

    tmp = tmp->next;

}

// if you couldn't find any valid processes, you need to do a full refresh
if (valid_proc == NULL) {
```

```
    printf("Clearing old info...\n");
    //Clear out old info
    clear_process_list(proc_list);
    proc_list = NULL;
    clear_thread_list(thr_list);
    thr_list = NULL;
    clear_user_list(user_list);
    user_list = NULL;

    printf("Resetting memory_index...\n");
    if(!mem_move (0x00000000)){

        printf("error resetting memory index.\n");
        exit(1);

    }

    printf("Restarting...\n");

    //Read in new info
    load_debug_data();
    load_info();
    load_tcpip_data();
    user_list = get_users();
    load_user_process();

}

// Otherwise, we've got a valid process that's still in the process list
// Clean out the checkmarks for the existing processes
fend_address = valid_proc->proc_eprocess.ActiveProcessLinks.Blink;
bend_address = valid_proc->proc_eprocess.ActiveProcessLinks.Flink;
next_address = valid_proc->proc_eprocess.ActiveProcessLinks.Flink;
prev_address = valid_proc->proc_eprocess.ActiveProcessLinks.Blink;
ctr = 0;
woffset = (uint32_t) get_pdb_type_offset( "_EPROCESS.ActiveProcessLinks");
// Go through the EPROCESS linked list (FLINK) that the current process
// is a member of
while ((next_address != fend_address) && (next_address != 0x0)){

    curr_address = next_address - woffset;
    pid = keget_pdb_type("_EPROCESS.UniqueProcessId", curr_address);
    next_address = keget_pdb_type("_EPROCESS.ActiveProcessLinks.Flink",
                   curr_address);
    // mark any processes that are in the EPROCESS linked list as IN
```

```
    // LINKED LIST
    tmp = proc_list;
    gotone = 0;
    while (tmp != NULL) {

        if (tmp->proc_eprocess.UniqueProcessId == pid) {

            tmp->checkmark = 1;
            gotone = 1;
            break;

        }

        tmp=tmp->next;

    }

    if (gotone == 0) {
        // a process that not in CMAT's current list of processes
        printf("  adding new process %i\n",pid);
        if ((tmp = malloc(sizeof(process_list))) == NULL ) {
            (void)exit(EXIT_FAILURE);
        }

        tmp->proc_eprocess_location = curr_address;
        tmp->next = proc_list;
        tmp->prev = NULL;
        tmp->users = NULL;
        tmp->sockets = NULL;
        tmp->eprocess_in_link_list = 1;
        tmp->checkmark = 1;
        proc_list = tmp;
        if (tmp->next != NULL) tmp->next->prev = tmp;
        kegrab_eprocess(tmp);
        grab_peb(tmp);

    }

    ctr++;
    if (ctr > proc_count) break;

}

ctr = 0;
// Go through the entire EPROCESS linked list (BLINK) that the current
```

```
// process is a member of
while ((prev_address != bend_address) && (prev_address != 0x0)){

    curr_address = prev_address - woffset;
    pid = keget_pdb_type("_EPROCESS.UniqueProcessId", curr_address);
    prev_address = keget_pdb_type("_EPROCESS.ActiveProcessLinks.Blink",
                   curr_address);
    tmp = proc_list;
    gotone = 0;
    // mark any processes that are in the EPROCESS linked list as IN
    // LINKED LIST
    while (tmp != NULL) {

        if (tmp->proc_eprocess.UniqueProcessId == pid) {

            tmp->checkmark = 1;
            gotone = 1;
            break;

        }

        tmp=tmp->next;

    }

    if (gotone == 0) {
        // a process that not in CMAT's current list of processes
        printf("  adding a new process %i\n",pid);
        if ((tmp = malloc(sizeof(process_list))) == NULL ) {
            (void)exit(EXIT_FAILURE);
        }

        tmp->proc_eprocess_location = curr_address - woffset;
        tmp->next = proc_list;
        tmp->prev = NULL;
        tmp->users = NULL;
        tmp->sockets = NULL;
        tmp->eprocess_in_link_list = 1;
        tmp->checkmark = 1;
        proc_list = tmp;
        if (tmp->next != NULL) tmp->next->prev = tmp;
        kegrab_eprocess(tmp);
        grab_peb(tmp);

    }
```

```
    ctr++;
    if (ctr > proc_count) break;

}

// Now remove any processes that weren't in the list and were previously
// in the list
tmp = proc_list;
while (tmp != NULL) {

    if ((tmp->checkmark == 0) && tmp->eprocess_in_link_list &&
        (tmp->proc_eprocess.UniqueProcessId != 0) )  {

        printf("  removing defunct process %i\n",
               tmp->proc_eprocess.UniqueProcessId);
        if (tmp->next != NULL) tmp->next->prev = tmp->prev;
        if (tmp->prev != NULL) tmp->prev->next = tmp->next;
        if (proc_list == tmp) proc_list = tmp->next;
        old = tmp;
        tmp = tmp->next;
        free(old->proc_eprocess.ImageFileName2);
        free(old);
        continue;

    }

    tmp = tmp->next;

}

// Resort the processes so the lowest (first) IDs show up first
printf ("  sorting processes by PID\n");
do    {

    process_list *old1;
    process_list *old2;
    val_change = 0;
    tmp = proc_list;
    while (tmp->next != NULL) {

        if (tmp->next == NULL) break;
        if (tmp->proc_eprocess.UniqueProcessId >
            tmp->next->proc_eprocess.UniqueProcessId)
        {
```

```
            if (proc_list == tmp) proc_list = tmp->next;
            val_change++;
            old1 = tmp;
            old2 = tmp->next;


            old2->prev = old1->prev;
            if (old1->prev != 0) old1->prev->next = old2;
            old1->next = old2->next;
            if (old2->next != 0) old2->next->prev = old1;
            old2->next = old1;
            old1->prev = old2;

        }
        else {

            tmp = tmp->next;

        }


    }


  }
  while (val_change);

}

first_pass = 0;

// There are 2 ways to provide info to the user:
//   option 1: FEATURE = FALSE: standard command line user interface, user
//      selects a process and sees info about it
//   option 2: FEATURE = TRUE: send to the specified dump file, a select
//      subset of "features" for each process (probably as input into a data
//      fusion tool)
if (!FEATURE) {


  if(BENCH_DEBUG) {

    printf("\n\n");
    exit(0);
```

```
}


// Show list of users
printf("\n\nUser List \n");
printf("\tUser\t\tToken\t\t\t\t\t\t\tPath\n");
wuser = user_list;
while (wuser != NULL) {

    printf("%15s\t\t%s\t\t%s\n",wuser->Name,wuser->Token,wuser->Path);
    if (MYDEBUG) {

        printf("**stage_1_menu** \t\tToken: ");
        for (ctr = 0; ctr < 0x1c; ctr++)
           printf("%x ",((unsigned char) wuser->Sid[ctr]));
        printf("\n");

    }

    wuser = wuser->next;

}


// Show Registry hives
print_hives();

// Show Process Names and IDs
printf ("\n\nProcess List:\n");
tmp = proc_list;
proc_counter = 0;
printf (" Proc#\tSession\tStation\t\tPPID\t");
printf (" PID\tInProcList\t\t\t\t\tOwner\t\tName\n");
while (tmp != NULL)        {

    printf("%4i",proc_counter++);
    printf("\t%4i",tmp->proc_eprocess.SessionId);
    printf("\t%8lli",tmp->proc_eprocess.Win32WindowStation);
    printf("\t%4lli",tmp->proc_eprocess.InheritedFromUniqueProcessId);
    printf("\t%4lli",tmp->proc_eprocess.UniqueProcessId);
    //          printf("\t%8lli",tmp->proc_eprocess_location);
    printf("\t%s",(tmp->eprocess_in_link_list == 1)?"Yes":"No");
    printf("\t%50s",((tmp->users != NULL)?tmp->users->Name:(
                 (get_sid(tmp) == NULL)?" ":
```

```
                get_sid(tmp))));
        printf("\t\t%s \n",tmp->proc_eprocess.ImageFileName2);
        tmp = tmp->next;

    }


    printf ("Options:\n");
    if ( proc_list != NULL)
    {

        //  if (LIVE) system("clear");
        printf ("#:\tSelect a process\n");
        printf ("s:\tShow System Information\n");
        printf ("c:\tShow Clipboard Information\n");
        printf ("m:\tShow Loaded Modules\n");
        printf ("x:\t Convert Virtual  to Physical for System Space\n");
        if (PTOV) printf ("p:\t Convert Physical to Virtual for all Processes\n");
        printf ("r:\t Extract Registry Value\n");
        if (LIVE) printf ("f:\treFresh\n");

    }


    printf ("<enter>:\tQuit\n");

    fgets (decision, 5, stdin);
    dummy = strlen(decision);
    strncpy_s (decision, 5, decision, dummy);

    printf ("\n\n\n");

    if ((atoi(decision) < (proc_counter)) && (atoi(decision) > 0))
    {

        if (LIVE) system("clear");
        while(stage_2_menu(atoi(decision))) continue;

    }

    else if (*decision == 's')
    show_system_info();
    // BEGIN -- CMAT_V
    else if (*decision == 'f'){
```

```
    printf("Clearing old info...\n");
    //Clear out old info
    clear_process_list(proc_list);
    proc_list = NULL;
    clear_thread_list(thr_list);
    thr_list = NULL;
    clear_user_list(user_list);
    user_list = NULL;

    printf("Resetting memory_index...\n");
    if(!mem_move (0x00000000)){

        printf("error resetting memory index.\n");
        exit(1);

    }

    printf("Restarting...\n");

    //Read in new info
    load_debug_data();
    load_info();
    load_tcpip_data();
    user_list = get_users();
    load_user_process();

}

// END -- CMAT_V
else if (*decision == 'c') {

    tmp = proc_list;
    whead = NULL;
    while (tmp != NULL) {

        if (tmp->proc_eprocess.Win32WindowStation == 0x0) {
            tmp = tmp->next; continue;
        }

        printf(".");
        show_clipboardk(tmp);
        /* This code works with Vista and 7 (where there are session
           IDs) but not with XP (where there aren't)
        wptr1 = whead;
```

```
            gotit = 0;

            while (wptr1 != NULL) {

                if (wptr1->session_id == tmp->proc_eprocess.SessionId) {
                    gotit = 1; break;
                }

                wptr1 = wptr1->next;

            }

            if (!gotit) {

                wptr1 = malloc(sizeof(session_t));
                wptr1->next = whead;
                wptr1->session_id = tmp->proc_eprocess.SessionId;
                whead = wptr1;
                show_clipboardk(tmp);

            }

            */
            tmp = tmp->next;

        }

        printf("\n\n\n");

    }

    else if (*decision == 'm') {

        if (debug_data->x64) {

            wptr = memgrabke_64_LE(debug_data->PsLoadedModuleList);
            wend = memgrabke_64_LE(debug_data->PsLoadedModuleList + 0x8);

        }
        else {

            wptr = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList)
                 & 0x00000000ffffffff;
            wend = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList + 0x4)
                 & 0x00000000ffffffff;
```

```
    }

    // Scroll through the list until you find tcpip.sys
    while ((wptr != 0x0) && (wptr != wend)) {

        // Retrieve PE header information for the module
        if (debug_data->x64) {

            //_MODULE_ENTRY.ImageBase
            wimage_base = memgrabke_64_LE(wptr+0x30);
            // _MODULE_ENTRY.DriverName
            wdrivername.Length = memgrabke_16_LE(wptr + 0x58);
            wdrivername.MaximumLength = wdrivername.Length;
            wdrivername.Buffer = memgrabke_64_LE(wptr + 0x60);

        }
        else {

            //_MODULE_ENTRY.ImageBase
            wimage_base = memgrabke_32_LE(wptr+0x18);
            // _MODULE_ENTRY.DriverName
            wdrivername.Length = memgrabke_16_LE(wptr + 0x2c);
            wdrivername.MaximumLength = wdrivername.Length;
            wdrivername.Buffer = memgrabke_32_LE(wptr + 0x30);

        }

        wstring = memgrabke_unicode_LE_space (wdrivername);
        printf("Driver Name: %s\tImage Base: %x\n",wstring,wimage_base);
        free(wstring);
        // Iterate to the next driver
        if (debug_data->x64)
        wptr = memgrabke_64_LE(wptr + 0x00);
        else
        wptr = memgrabke_32_LE(wptr + 0x00);

    }


}

else if (*decision == 'r') {

    strcpy_s( hive_name, 2, " ");
```

```
    strcpy_s( reg_key, 2, " ");
    while (strlen(hive_name) != 0) {

        printf("Fully qualified hive\n");
        gets_s (hive_name,80);
        printf("Registry Key\n");
        gets_s (reg_key,200);
        get_values(hive_name,reg_key,"");

    }


}

else if (*decision == 'x') {

    while (1) {

        printf("Virtual Address\n");
        gets_s (sva,24);
        xva = 0x0;
        for (i=0; i < 24; i++) {

            if (sva[i] == 0) break;
            if ( (sva[i] >= 'A') && (sva[i] <= 'F') )
            xva = (xva << 4) + (sva[i] - 'A' + 10);
            else if ( (sva[i] >= 'a') && (sva[i] <= 'f') )
            xva = (xva << 4) + (sva[i] - 'a' + 10);
            else if ( (sva[i] >= '0') && (sva[i] <= '9') )
            xva = (xva << 4) + (sva[i] - '0');
            else {

                printf("%i %i\n",i,sva[i]);
                xva = 0; break;

            }


        }

        if (xva == 0) break;
        printf("VA: %llx     Phys: %llx\n",xva,kevtop(xva));

    }
```

```
}

else if ( (*decision == 'p') && PTOV) {

    while (1) {

        printf("Physical Address: ");
        gets_s (sva,24);
        xva = 0x0;
        for (i=0; i < 24; i++) {

            if (sva[i] == 0) break;
            if ( (sva[i] >= 'A') && (sva[i] <= 'F') )
            xva = (xva << 4) + (sva[i] - 'A' + 10);
            else if ( (sva[i] >= 'a') && (sva[i] <= 'f') )
            xva = (xva << 4) + (sva[i] - 'a' + 10);
            else if ( (sva[i] >= '0') && (sva[i] <= '9') )
            xva = (xva << 4) + (sva[i] - '0');
            else {

                printf("%i %i\n",i,sva[i]);
                xva = 0; break;

            }


        }

        if (xva == 0) break;
        wptov = ptov(xva);
        if ((wptov == NULL) || (wptov->vaddress == 0))
           printf(" Virtual Address: 0\n");
        else {

            printf("Virtual Address (Process): \t");
            while (wptov != NULL) {

                printf("%llx (%4lli)\t",wptov->vaddress,wptov->pid);
                wptov = wptov->next;

            }

            printf("\n");
```

```
                }


            }


        }

        else
        return 0; // we're done
        return 1; // Keep looping

    }

    else {

        dump_memory3();
        return (LIVE); // we're done

    }


}


/***********************************/
/* Per Process menu - second level */
/***********************************/
int stage_2_menu(int proc_num)
{

    process_list *sel_proc = proc_list;
    socket_list_type *wsocket;
    char decision[5] = {
        0
    }
    ;
    char sva[24] = {
        0
    }
    ;
    uint64_t xva;
    int i;

    for (i=0; (signed int) i < proc_num; i++) sel_proc = sel_proc->next;
```

```
printf ("%ld: %s selected:\n",  (long) sel_proc->proc_eprocess.UniqueProcessId,
         sel_proc->proc_eprocess.ImageFileName2);
printf ("1\tDisplay Process Environment Information\n");
printf ("2\tDisplay all DLLs loaded\n");
printf ("3\tDisplay all Files accessed\n");
printf ("4\tDisplay all Registry Keys accessed\n");
printf ("5\tDisplay all Sockets opened\n");
printf ("c:\tShow Clipboard Information\n");
printf ("\nx:\t Convert Virtual  to Physical\n");
printf ("<enter>:\tquit\n");

fgets (decision, 5, stdin);
strncpy_s (decision, 5, decision, strlen(decision));

switch (*decision)
{

    case '1':
    show_proc_env (sel_proc);
    break;
    case '2':
    show_proc_dll (sel_proc,NULL);
    break;
    case '3':
    show_proc_hand (sel_proc,"file",NULL);
    break;
    case '4':
    show_proc_hand (sel_proc,"key",NULL);
    break;
    case '5':
    printf("\n\nLocal Address        Remote Address        Protocol    IPv4/v6\n");
    wsocket = sel_proc->sockets;
    while (wsocket != NULL) {

        printf("%20s %20s %12s %11s\n",wsocket->LocalPort,
               wsocket->RemotePort, wsocket->Protocol, wsocket->Ipv6);
        wsocket = wsocket->next;

    }

    printf("\n\n");
    break;
    case 'c':
    show_clipboard(sel_proc);
```

```
    break;
    case 'x':
    while (1) {

        printf("Virtual Address\n");
        gets_s (sva,24);
        xva = 0x0;
        for (i=0; i < 24; i++) {

            if (sva[i] == 0) break;
            if ( (sva[i] >= 'A') && (sva[i] <= 'F') )
            xva = (xva << 4) + (sva[i] - 'A' + 10);
            else if ( (sva[i] >= 'a') && (sva[i] <= 'f') )
            xva = (xva << 4) + (sva[i] - 'a' + 10);
            else if ( (sva[i] >= '0') && (sva[i] <= '9') )
            xva = (xva << 4) + (sva[i] - '0');
            else {

                printf("%i %i\n",i,sva[i]);
                xva = 0; break;

            }


        }

        if (xva == 0) break;
        printf("VA: %llx     Phys: %llx\n",xva,
               vtop(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
               xva));

    }

    break;
    default:
    return 0;
    break;

}


if ((atoi(decision) < (8)) && (atoi(decision) > 0)) {

}
```

```
    else return 0;

    return 1;

}

/******************************/
/* Display the usage statement */
/******************************/
void usage ( char *binname ) {

    printf ("Usage:\n\n");
    printf ("%s [-DATA <data directory>] [-GETPDB] [-PAGEFILE <pagefile(s)>] ",
            binname);
    printf ("[-PTOV] [-DUMPSTRUCT <filename.pdb>] [-DUMPSYM <filename.pdb>] ");
    printf ("[-NOUDP] [-FEATURE <output file>] <memory dump file> \n\n");
    printf ("%s [-DATA <data directory>] [-GETPDB] [-FEATURE <output file>] ",
            binname);
    printf ("-virt_live <domain id> \n\n");
    printf("\t-DATA <data directory> \t\tDirectory where the files are (this ");
    printf("must be before any filenames are specified); default is .\\\n");
    printf("\t                       \t\tnote the trailing backslash; this is ");
   printf("required\n");
    printf("\t                       \t\twhen running under Linux, this must be ");
    printf("used with a forward slash\n");
    printf("\t-GETPDB\t\t\t\tDownload PDB files from Microsoft Symbol Server (if ");
    printf("not set, pdb files must be in the data directory)\n");
    printf("\t-PAGEFILE\t\t\tNames of the page files(s)\n");
    printf("\t-PTOV\t\t\t\tEnable converting physical addresses to virtual ");
    printf("(linear) addresses\n");
    printf("\t-DUMPSTRUCT <pdb file>\t\tDisplay the structures from the PDB file ");
    printf("and then exit\n");
    printf("\t-DUMPSTRUCT <pdb file>\t\tDisplay the symbols from the PDB file ");
    printf("and then exit\n");
    printf("\t-NOUDP\t\t\t\tDo not extract UDP information\n");
    printf("\t-FEATURE <output file> \t\tSend each process' features to ");
    printf("<output file>.txt \n");
    printf(" \t\t\t\t\t  Feature Files: \n");
    printf(" \t\t\t\t\t     name+DateTime_1\t\tProcess ID, Name, and User\n");
    printf(" \t\t\t\t\t     name+DateTime_2\t\tNetwork Connections by ");
    printf("Process\n");
    printf(" \t\t\t\t\t     name+DateTime_3\t\tLoaded Modules by Process\n");
    printf(" \t\t\t\t\t     name+DateTime_4\t\tOpen Files by Process\n");
    printf(" \t\t\t\t\t     name+DateTime_5\t\tRegistry Keys by Process ");
    printf("(currently removed due to speed considerations)\n");
```

```
    printf(" \t\t\t\t\t     name+DateTime_6\t\tLoaded Modules\n");
    printf("\n");
    printf ("Default: %s <memory dump file> \n\n\n",binname);

}


/**********************/
/* Open the dump file */
/**********************/
int open_feature_file( char *filename) {

    if (fopen_s(&feature_file, filename, "wb" ) ) {

        printf ("The feature file %s can't be opened. Sorry.\n", filename);
        return 1;

    }

    else {
        return 0;
    }


}


//Create a memory dump from a virtual machine ID

int cmat_dump_virtual(char *vm_id, char *file_name) {


    #ifdef ENABLE_XEN

    xa_instance_t xai;
    char *filename = NULL;
    FILE *f = NULL;
    unsigned char *memory = NULL;
    uint32_t offset = 0;
    uint32_t address = 0;

    system("clear");

    printf("\n\nObtaining a memory dump from VM ID: %s. Please wait...\n\n", vm_id);
```

```
/* this is the domain ID that we are looking at */
uint32_t dom = atoi(vm_id);

/* this is the file name to write the memory image to */
filename = strndup(file_name, 200);

/* initialize the xen access library */
if (xa_init_vm_id_cmat(dom, &xai, XA_FAILSOFT) == XA_FAILURE){

    perror("failed to init XenAccess library");
    goto error_exit;

}


/* open the file for writing */
if ((f = fopen(filename, "w+")) == NULL){

    perror("failed to open file for writing");
    goto error_exit;

}


/* assuming that we are looking at xen domain, and not image file */
while (address < xai.m.xen.size){


    /* access the memory */
    memory = xa_access_pa(&xai, address, &offset, PROT_READ);

    /* write memory to file */
    if (memory){

        /* memory mapped, just write to file */
        size_t written = fwrite(memory, 1, xai.page_size, f);
        if (written != xai.page_size){

            perror("failed to write memory to file");
            goto error_exit;

        }

        munmap(memory, xai.page_size);
```

```
    }

    else{

        /* memory not mapped, write zeros to maintain offset */
        unsigned char *zeros = malloc(xai.page_size);
        memset(zeros, 0, xai.page_size);
        size_t written = fwrite(zeros, 1, xai.page_size, f);
        if (written != xai.page_size){

            perror("failed to write zeros to file");
            goto error_exit;

        }

        free(zeros);

    }


    /* move on to the next page */
    address += xai.page_size;

}


error_exit:
if (memory){
    munmap(memory, xai.page_size);
}

if (f){
    fclose(f);
}


/* cleanup any memory associated with the XenAccess instance */
xa_destroy(&xai);

printf("Memory dump file created.\n\n");

#endif /* ENABLE XEN*/

return 0;
```

```
}


/********************************************************************************/

/*******************************/
/* LOAD METADATA FOR MEMORY DUMP */
/*******************************/

/* find KDDEBUGGER__DATA64 and some other cool stuff */
/*       including the version of the memory dump and when physical address
          extensions is enabled */

int load_debug_data() {


    uint32_t data_from_memdump = 0;          // Placeholder for pattern
                                             // matching the memory dump
    uint32_t data_from_memdump2 = 0;         // Placeholder for pattern
                                             // matching the memory dump
    uint64_t data_from_memdump8;             // Placeholder for pattern
                                             // matching the memory dump

    uint64_t wloc = 0;                       // Pointer into memory dump
    uint64_t end_of_file;                    // Size of the memory dump
    int percent_count = 0;                   // Progress bar

    uint64_t wstart, skipback, back_pos;     // Iterators used to move around
                                             // known data structures

    uint32_t wKernBase, wPsLoadedModuleList; // Used to find the
                                             // DBGKD_GET_VERSION64
                                             // data structure after locating
                                             // KDDEBUGGER_DATA64

    uint8_t done;                            // Used to flag when second pass
                                             // is done when parsing a
                                             // file w/o the KDDEBUGGER_DATA64 &
                                             // to extract ObjTypTable
    uint32_t tst1, tst2, tst3, tst4;         // Placeholders to check name of
                                             // PDB file and to check
                                             // self-reference page directory
                                             // tables
    uint32_t woffset;                        // Used to check if x64 when
                                             // parsing a file without the
```

```
                                           // KDDEBUGGER_DATA64 structure
uint16_t wmagic;                           // Used to check if x64 when
                                           // parsing a file without
                                           // the KDDEBUGGER_DATA64 structure

char *pdb_url, *wpdb_url;                  // Stores the url needed to get
                                           // the Microsoft Symbol Server
                                           // to download kernel pdb file

uint64_t pe_offset, hdr_base;              // Used to extract info (e.g., OS
                                           // Version) from the kernel PE

uint32_t data_segment, woffsetx, wptrx;    // extract Object Type Table from
                                           // kernel data segment
uint64_t wlocx, whead;                     // extract Object Type Table from
                                           // kernel data segment
uint8_t widx;                              // extract Object Type Table from
                                           // kernel data segment

uint16_t nbr_sections;                     // The number of sections in the
                                           // PE file

uint64_t toffset;                         // offset from Object Type list
                                          // header info
                                          // to Object Type info

uint64_t mem_pointer;

char osversion[9];
char * test_section;
unicode_string wtypename;
char *xname;

char *wstring;
int i;


printf ("\nSearching for debug data in memory dump\n");

// go to the beginning of the file
mem_move(0);

// Iterate through the memory dump looking looking for KDDEBUGGER_DATA64 structure
end_of_file = get_mem_size();
```

```
while (wloc < end_of_file) {


    // Progress measurement
    if ( ((percent_count * 5) == 100) && ((wloc / (end_of_file/100)) >=
          (percent_count * 5) )) {

        fprintf(stderr,"%02ld%%\n", (wloc / (end_of_file/100)) );
        percent_count++;

    }

    else if ( (wloc / (end_of_file/100)) >= (percent_count * 5) ) {

        fprintf(stderr,"%02ld%%--", (wloc / (end_of_file/100)) );
        percent_count++;

    }


    // Pull the next 8 bytes from memory dump
    data_from_memdump = memgrab_32_BE(0); wloc += 4;
    data_from_memdump2 = memgrab_32_LE(0); wloc += 4;

    // The reason the following code isn't only using pdb format is because
    // only the beginning part of the structure
    // is in the pdb.  The remainder (including the tag info used above and
    // the variable PaeEnabled)
    // can be found in WINDBG\sdk\inc\wdbgexts.h shipped with WINDBG
    // Reference used: http://www.rootkit.com/newsread.php?newsid=153

    // If we find the KDDEBUGGER_DATA64 structure
    if (    (data_from_memdump == 0x4b444247) && (data_from_memdump2 < 0x0500)) {
        //        ULONG OwnerTag = KDBG     ULONG Size < 0x1000
        // Check/set the PAE Enabled flag
        debug_data->PaeEnabled =         memgrab_16_LE(wloc + 0x1e);
        if (debug_data->PaeEnabled) {

            set_nopae(0);
            debug_data->KernBase = memgrab_64_LE(wloc + 0x00);
            debug_data->PsLoadedModuleList = memgrab_64_LE(wloc + 0x30);
            debug_data->ObpTypeObjectType = memgrab_64_LE(wloc + 0x88);
            wKernBase = memgrab_32_LE(wloc + 0x00);
            wPsLoadedModuleList = memgrab_32_LE(wloc + 0x30);
```

```
}
else {

    set_nopae(1);
    debug_data->KernBase = memgrab_32_LE(wloc + 0x00) &
              0x00000000ffffffff;
    debug_data->PsLoadedModuleList = memgrab_32_LE(wloc + 0x38) &
              0x00000000ffffffff;
    debug_data->ObpTypeObjectType = memgrab_32_LE(wloc + 0x90) &
              0x00000000ffffffff;
    wKernBase = memgrab_32_LE(wloc + 0x00);
    wPsLoadedModuleList = memgrab_32_LE(wloc + 0x38);

}


// Look for the DBGKD_GET_VERSION64 data structure -- it should
// either be right before or right after KDDEBUGGER_DATA64 structure
debug_data->MachineType = 0x0;
wstart = wloc - 0x100;
while ( ((memgrab_64_LE(wstart + 0x10) != debug_data->KernBase) ||
(memgrab_64_LE(wstart + 0x18) != debug_data->PsLoadedModuleList)
) &&
((memgrab_32_LE(wstart + 0x10) != wKernBase) ||
(memgrab_32_LE(wstart + 0x18) != wPsLoadedModuleList)
) &&
wstart < wloc - 0x20) {

    wstart = wstart + 0x04;

}

if ( ((memgrab_64_LE(wstart + 0x10) == debug_data->KernBase) &&
      (memgrab_64_LE(wstart + 0x18) == debug_data->PsLoadedModuleList)) ||
     ((memgrab_32_LE(wstart + 0x10) == wKernBase) &&
            (memgrab_32_LE(wstart + 0x18) == wPsLoadedModuleList)) ){

    debug_data->MajorVersion =        memgrab_16_LE(wstart + 0x00);
    debug_data->MinorVersion =        memgrab_16_LE(wstart + 0x02);
    debug_data->MachineType =        memgrab_16_LE(wstart + 0x08);

}

else {
```

```
    wstart = wloc + 0x260;
    while ( ((memgrab_64_LE(wstart + 0x10) != debug_data->KernBase)
         || (memgrab_64_LE(wstart + 0x18) !=
             debug_data->PsLoadedModuleList))
    && wstart < wloc + 0x1000) {

        wstart = wstart + 0x04;

    }

    if ((memgrab_64_LE(wstart + 0x10) == debug_data->KernBase) &&
        (memgrab_64_LE(wstart + 0x18) ==
           debug_data->PsLoadedModuleList)) {

        debug_data->MajorVersion =         memgrab_16_LE(wstart + 0x00);
        debug_data->MinorVersion =         memgrab_16_LE(wstart + 0x02);
        debug_data->MachineType =         memgrab_16_LE(wstart + 0x08);

    }

    else {

        printf("Unable to find debug data!\n");

    }


}


// Provide a string representation of the Machine Type (from the
// Microsoft PECOFF Spec)
debug_data->MachineString = (char *) malloc( 13 * sizeof(char));
debug_data->MachineString[0] = ((int) (debug_data->MachineType /
    0x10)) + 0x30;
debug_data->MachineString[1] = ((int) (debug_data->MachineType %
    0x10)) + 0x30;
debug_data->MachineString[2] = 0;
if (debug_data->MachineType == 0x0)
                   debug_data->MachineString = "UNKNOWN";
if (debug_data->MachineType == 0x1d3)
                   debug_data->MachineString = "AM33";
if (debug_data->MachineType == 0x8664)
                   debug_data->MachineString = "AMD64";
if (debug_data->MachineType == 0x1c0)
```

```
                    debug_data->MachineString = "ARM";
if (debug_data->MachineType == 0xebc)
                    debug_data->MachineString = "EBC";
if (debug_data->MachineType == 0x14c)
                    debug_data->MachineString = "I386";
if (debug_data->MachineType == 0x200)
                    debug_data->MachineString = "IA64";
if (debug_data->MachineType == 0x9041)
                    debug_data->MachineString = "M32R";
if (debug_data->MachineType == 0x266)
                    debug_data->MachineString = "MIPS16";
if (debug_data->MachineType == 0x366)
                    debug_data->MachineString = "MIPSFPU";
if (debug_data->MachineType == 0x466)
                    debug_data->MachineString = "MIPSFPU16";
if (debug_data->MachineType == 0x1f0)
                    debug_data->MachineString = "POWERPC";
if (debug_data->MachineType == 0x1f1)
                    debug_data->MachineString = "POWERPCFP";
if (debug_data->MachineType == 0x166)
                    debug_data->MachineString = "R4000";
if (debug_data->MachineType == 0x1a2)
                    debug_data->MachineString = "SH3";
if (debug_data->MachineType == 0x1a3)
                    debug_data->MachineString = "SH3DSP";
if (debug_data->MachineType == 0x1a6)
                    debug_data->MachineString = "SH4";
if (debug_data->MachineType == 0x1a8)
                    debug_data->MachineString = "SH5";
if (debug_data->MachineType == 0x1c2)
                   debug_data->MachineString = "THUMB";
if (debug_data->MachineType == 0x169)
                   debug_data->MachineString = "WCEMIPSV2";

// If the machine type is 0x8664, set the x64 flag
debug_data->x64 = (debug_data->MachineType == 0x8664);
set_x64(debug_data->x64);
setreg_x64(debug_data->x64);
set_pdb_x64(debug_data->x64);

break;


}
```

```
}


// If KDDEBUGGER_DATA64 isn't in the memory dump, go through the memory dump
// again and look for the kernel executable
if (wloc >= end_of_file) {

    printf("\n KDBG record not present.  Initiating second pass\n");
    wloc = 0;
    percent_count = 0;
    done = 0;
    while (!done && (wloc < end_of_file)) {

        // Progress indicator
        if (((percent_count * 5) == 100) && ((wloc / (end_of_file/100)) >=
              (percent_count * 5))) {

            fprintf(stderr,"%02ld%%\n", (wloc / (end_of_file/100)) );
            percent_count++;

        }

        else if ( (wloc / (end_of_file/100)) >= (percent_count * 5) ) {

            fprintf(stderr,"%02ld%%--", (wloc / (end_of_file/100)) );
            percent_count++;

        }


        // Pull the next 8 bytes from memory dump
        data_from_memdump8 = memgrab_64_LE(wloc);
        data_from_memdump2 = memgrab_32_BE(0);

        // If you find a pdb file, check if the base name matches one of the
        // known kernel executable names
        if (data_from_memdump2 == 0x2e706462) {
            // ".pdb"
            for (i = 0; i < 5; i++) {

                // looking for "krnl" anywhere in the 8 bytes that make up
                // the name of the PDB file (remember it's reversed)
                // could be in positions 8,7,6,5   7,6,5,4   6,5,4,3
                // 5,4,3,2    4,3,2,1
```

```
uint8_t tst1 = (uint8_t) (data_from_memdump8 & 0xff);
uint8_t tst2 = (uint8_t) ((data_from_memdump8 & 0xff00) >>8);
uint8_t tst3 = (uint8_t) ((data_from_memdump8 & 0xff0000) >>16);
uint8_t tst4 = (uint8_t) ((data_from_memdump8 & 0xff000000) >>24);

// if a valid kernel PDB filename is found, check if it
// really is in the middle of a debug section
if ( (tst1 == 'k') && (tst2 == 'r') && (tst3 == 'n') &&
     (tst4 == 'l')) {

    // Determine if PAE is enabled based on the name of the
    // kernel executable
    debug_data->PaeEnabled =  ((data_from_memdump8 &
                               0xff00000000ULL) >>32 == 'p'
           && (data_from_memdump8 & 0xff0000000000ULL) >>40
              == 'a');

    // Now go back 16M (0x1000000) and start looking for the
    //  beginning of the PE
    // (assuming we are currently in the Debug section)
    skipback = 0;
    back_pos = wloc & 0xfffffffffffff000ULL;
    while (skipback <= 0x1000000) {

        data_from_memdump8 = memgrab_64_BE(back_pos -
                               skipback + 0x4e);
        // If we find the beginning of the PE, we'll see
        // "This program cannot
        // be run in DOS mode" 0x4E into the PE
        if (data_from_memdump8 == 0x546869732070726fULL) {
            // "This pro"gram cannot be run in DOS mode
            printf("Found the head at %llx\n",
                     back_pos - skipback);
            // Make sure the virtual Kernel base address
            // (KernBase) is zero
            debug_data->KernBase = 0;

            // Populate the physical Kernel base address
            // (pKernBase)
            debug_data->pKernBase = back_pos - skipback;

            // Check if it's x64 by using the Magic# from
            // the Microsoft PECOFF Spec
            woffset = memgrab_32_LE(back_pos - skipback + 0x3c);
            wmagic = memgrab_16_LE(back_pos - skipback +
```

```
         woffset + 0x18);
debug_data->x64 = (wmagic == 0x20b);
set_x64(debug_data->x64);
setreg_x64(debug_data->x64);
set_pdb_x64(debug_data->x64);
if (debug_data->x64) debug_data->PaeEnabled = 1;
if (debug_data->PaeEnabled)
set_nopae(0);
else
set_nopae(1);

// Now set the other meta-information
debug_data->MajorVersion =
            memgrab_16_LE(back_pos - skipback +
            woffset + 0x18 + 0x28);
debug_data->MinorVersion =
            memgrab_16_LE(back_pos - skipback +
            woffset + 0x18 + 0x2a);
debug_data->MachineType =
            memgrab_16_LE(back_pos - skipback +
            woffset);
debug_data->MachineString =
            (char *) malloc( 13 * sizeof(char));
debug_data->MachineString[0] =
            ((int) (debug_data->MachineType /
            0x10)) + 0x30;
debug_data->MachineString[1] =
            ((int) (debug_data->MachineType %
             0x10)) + 0x30;
debug_data->MachineString[2] = 0;
if (debug_data->MachineType == 0x0)
   debug_data->MachineString = "UNKNOWN";
if (debug_data->MachineType == 0x1d3)
   debug_data->MachineString = "AM33";
if (debug_data->MachineType == 0x8664)
   debug_data->MachineString = "AMD64";
if (debug_data->MachineType == 0x1c0)
   debug_data->MachineString = "ARM";
if (debug_data->MachineType == 0xebc)
   debug_data->MachineString = "EBC";
if (debug_data->MachineType == 0x14c)
   debug_data->MachineString = "I386";
if (debug_data->MachineType == 0x200)
   debug_data->MachineString = "IA64";
if (debug_data->MachineType == 0x9041)
```

```
                debug_data->MachineString = "M32R";
            if (debug_data->MachineType == 0x266)
                debug_data->MachineString = "MIPS16";
            if (debug_data->MachineType == 0x366)
                debug_data->MachineString = "MIPSFPU";
            if (debug_data->MachineType == 0x466)
                debug_data->MachineString = "MIPSFPU16";
            if (debug_data->MachineType == 0x1f0)
                debug_data->MachineString = "POWERPC";
            if (debug_data->MachineType == 0x1f1)
                debug_data->MachineString = "POWERPCFP";
            if (debug_data->MachineType == 0x166)
                debug_data->MachineString = "R4000";
            if (debug_data->MachineType == 0x1a2)
                debug_data->MachineString = "SH3";
            if (debug_data->MachineType == 0x1a3)
                debug_data->MachineString = "SH3DSP";
            if (debug_data->MachineType == 0x1a6)
                debug_data->MachineString = "SH4";
            if (debug_data->MachineType == 0x1a8)
                debug_data->MachineString = "SH5";
            if (debug_data->MachineType == 0x1c2)
                debug_data->MachineString = "THUMB";
            if (debug_data->MachineType == 0x169)
                debug_data->MachineString = "WCEMIPSV2";
            printf("PAE = %x    KernelBase = %llx    x64 = %x   ",
                debug_data->PaeEnabled,
                debug_data->pKernBase,debug_data->x64);
            printf("MachineType = %x    Machine = %s MajorV = %x  ",
                debug_data->MachineType,
                debug_data->MachineString,
                debug_data->MajorVersion);

            printf("MinorV = %x\n",debug_data->MinorVersion);
            done = 1;
            break;

        }

        skipback +=0x1000;

    }


}
```

```
                        else
                        data_from_memdump8 = data_from_memdump8>>8;

                    }


                }

                wloc += 4;

            }


        }

        else {

            debug_data->KDBG_loc = wloc - 8;

            printf("\nKernBase = %llx     ObpTypeObjectType = %llx ",
    debug_data->KernBase,debug_data->ObpTypeObjectType);
            printf("PsLoadedModuleList = %llx     PAE = %x     wloc = %llx 64bit = %x\n",
                 debug_data->PsLoadedModuleList,
                 debug_data->PaeEnabled,wloc,debug_data->x64);

        }


        // Look for the Page Directory Tables... each version (32bit w/o PAE, 32bit
        //  w/ PAE, 64bit) has some sort of self-referencing trick
        wloc = 0;
        percent_count = 0;
        end_of_file = get_mem_size();
        while (wloc < end_of_file) {

            // Progress Bar
            if ( ((percent_count * 5) == 100) && ((wloc / (end_of_file/100)) >=
                 (percent_count * 5) )) {

                fprintf(stderr,"%02ld%%\n", (wloc / (end_of_file/100)) );
                percent_count++;

            }

            else if ( (wloc / (end_of_file/100)) >= (percent_count * 5) ) {
```

```
    fprintf(stderr,"%02ld%%--", (wloc / (end_of_file/100)) );
    percent_count++;

}


// If it's 64bit the entry 0x68 bytes from the beginning of the Page Map
// Level table points back
//  to the head of the table
if (debug_data->x64) {

    tst1 = memgrab_32_LE(wloc + 0xf68);
    tst2 = memgrab_32_LE(wloc + 0xf6c);
    if ( ((tst1 & 0xFFFFF000) == wloc) && ((tst2 & 0xFFFFF000) == 0x0)
              && ( (tst1 & 0x1)
              == 0x1) && ( (vtop(wloc,debug_data->KernBase) != 0) ||
              (debug_data->KernBase == 0))
    ) {

        if (MYDEBUG) printf("Got a candidate: %x\n",wloc);
        debug_data->kDTB = wloc;
        set_kDTB(debug_data->kDTB);
        printf("\n");
        break;

    }

    wloc += 0x1000;

}


// If it's 32bit w/ PAE, the fourth entry in the Page Directory Pointer
// table points back to the head of the table
else if (debug_data->PaeEnabled) {

    tst1 = memgrab_32_LE(wloc + 0x00) & 0xFFFFF000;
    if (tst1 == 0x0) {
        wloc += 8; continue;
    }

    tst2 = memgrab_32_LE(wloc + 0x08) & 0xFFFFF000;
    if ( (tst2 == 0x0) || (tst2 == tst1) ) {
        wloc += 8; continue;
```

```
}

tst3 = memgrab_32_LE(wloc + 0x10) & 0xFFFFF000;
if ( (tst3 == 0x0) || (tst3 == tst2) || (tst3 == tst1)) {
    wloc += 8; continue;
}

tst4 = memgrab_32_LE(wloc + 0x18) & 0xFFFFF000;
if ( (tst4 == 0x0) || (tst4 == tst3) || (tst4 == tst2) || (tst4 == tst1)) {
    wloc += 8; continue;
}

if (tst4 == wloc) {
    wloc += 8; continue;
}

if (tst4 > end_of_file) {
    wloc += 8; continue;
}

if ( (memgrab_32_LE(tst4) & 0xFFFFF000) == tst1) {

    if ( (memgrab_32_LE(tst4 + 0x08) & 0xFFFFF000) == tst2) {

        if ( (memgrab_32_LE(tst4 + 0x10) & 0xFFFFF000) == tst3) {

            if ( ((memgrab_32_LE(tst4 + 0x18) & 0xFFFFF000) == tst4)
            && ( (vtop(wloc,debug_data->KernBase) != 0) ||
                  (debug_data->KernBase == 0)) ) {

                if (MYDEBUG) printf("Got a candidate: %x\n",wloc);
                debug_data->kDTB = wloc;
                set_kDTB(debug_data->kDTB);
                printf("\n");
                break;

            }


        }


    }
```

```
        }

        wloc += 0x08;

    }


    // If it's 32bit w/o PAE, the entry 0xc00 from the beginning of the Page
    // Directory table points back to the head of the table
    else {

        mem_pointer = memgrab_32_LE(wloc + 0xc00);
        if ( ((mem_pointer & 0xFFFFF000) == wloc) && (
              (mem_pointer & 0x1) == 0x1)
        && ( (vtop(wloc,debug_data->KernBase) != 0) ||
             (debug_data->KernBase == 0)) ) {

            if (MYDEBUG) printf("Got a candidate: %x\n",wloc);
            debug_data->kDTB = wloc;
            set_kDTB(debug_data->kDTB);
            printf("\n");
            break;

        }

        wloc += 0x1000;

    }


}



// Now it's time to get the data structures for the kernel

// Case I: KDDEBUGGER_DATA64 present and we've got a virtual address

// The Object Types are stored in a doubly linked list.  There is a certain
// amount of header  information followed by
// the object type information.  TOFFSET stores the offset from the start of
// the header information (where the linked list is)
// to the beginning of the object type information (which coincidentally has
// another linked list that is not used)
```

```
// not sure if 32bit should be 28 or 38 or if it depends on Vista versus 7
toffset = (debug_data->x64?0x50:0x28);

if (debug_data->KernBase != 0x0) {

    if (kernel_pdb == NULL) {

        if (1) printf("in Kernbase != 0\n");
        // Build the url to query the Microsoft Symbol Server
        pdb_url = get_pdb_url(debug_data->KernBase,
                    PDB_RET_URL,debug_data->x64,debug_data->kDTB);
        if (MYDEBUG) printf("url = %s\n",pdb_url);
        wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
        strcpy_s(wpdb_url,strlen(pdb_url) + 1, pdb_url);
        wstring = get_pdb_url(debug_data->KernBase,PDB_RET_NAME,
                      debug_data->x64,debug_data->kDTB);
        xname = malloc(sizeof(char) * (strlen(wstring) +
                       strlen(DATADIRECTORY) + 1));
        strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(xname,strlen(DATADIRECTORY) + strlen(wstring) + 1,wstring);
        // Download the correct pdb file and extract the information from it
        if (MYDEBUG) printf("getting ready to call download_pdb_file\n");
        if (GET_PDB) xname = download_pdb_file(wstring, wpdb_url, DATADIRECTORY);
        printf("xname = %s\n",xname);
        if (MYDEBUG) printf("getting ready to call parse_pdb_file\n");
        kernel_pdb = parse_pdb_file(xname,PDB_DEFAULT);
        set_ntoskrnl_pdb(kernel_pdb);
        free(wstring);
        free(wpdb_url);
        free(pdb_url);
        free(xname);

    }


    // Extract the OS Version
    pe_offset =  memgrabke_32_LE(debug_data->KernBase + 0x3c);
    hdr_base = debug_data->KernBase + pe_offset + 0x18;
    debug_data->MajorOSVersion = memgrabke_16_LE(hdr_base + 0x28 );
    debug_data->MinorOSVersion = memgrabke_16_LE(hdr_base + 0x2a );
    // If the object type is no longer stored in the object_header (e.g.,
    // Windows 7),
    if (MYDEBUG) printf("getting ready to working on object types\n");
    if (!(get_pdb_fld_exists("_OBJECT_HEADER.Type"))) {
```

```
        // Get the location of the ObpObjectType symbol
        data_segment = get_pdb_section(kernel_pdb,".data",5)->virtual_address;
        nbr_sections = memgrabke_16_LE(hdr_base + 0x02);
        for (i = 0; i < nbr_sections; i++) {

            test_section = memgrabke_stringn(hdr_base +
                           (debug_data->x64?0xf0:0xe0) +
                           i * 0x28 + 0,8);
            if (!strcmp(test_section,".data")) {

                data_segment = memgrabke_32_LE(hdr_base +
                               (debug_data->x64?0xf0:0xe0) +
                               i * 0x28 + 0x0c);

            }

            free(test_section);

        }

        if (debug_data->x64)
        woffsetx = get_pdb_symbol(kernel_pdb,"ObpTypeObjectType",17)->offset;
        else
        woffsetx = get_pdb_symbol(kernel_pdb,"_ObpTypeObjectType",18)->offset;
        wptrx = get_pdb_symbol_loc(kernel_pdb,woffsetx,data_segment);
        // Go to the location of the ObpObjectType symbol and get the
        // starting address of ObjTypTable
        wlocx = debug_data->KernBase + (((uint64_t) wptrx) & 0x00000000ffffffff);
        if (debug_data->x64)
        whead = memgrabke_64_LE(wlocx) - toffset;
        else
        whead = ((uint64_t) (memgrabke_32_LE(wlocx) - toffset)) &
                 (0x00000000ffffffff);

    }


}

// Case II: KDDEBUGGER_DATA64 not present and we've got a physical address
else {

    if (kernel_pdb == NULL) {

        if (1) printf("in Kernbase == 0\n");
```

```
    // Build the url to query the Microsoft Symbol Server
    pdb_url = pget_pdb_url(debug_data->pKernBase, PDB_RET_URL,debug_data->x64);
    wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
    strcpy_s(wpdb_url,strlen(pdb_url) + 1, pdb_url);
    wstring = pget_pdb_url(debug_data->pKernBase,
              PDB_RET_NAME,debug_data->x64);
    xname = malloc(sizeof(char) * (strlen(wstring) +
            strlen(DATADIRECTORY) + 1));
    strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
    strcat_s(xname,strlen(DATADIRECTORY) + strlen(wstring) + 1,wstring);
    // Download the correct pdb file and extract the information from it
    if (GET_PDB) xname = download_pdb_file(wstring, wpdb_url, DATADIRECTORY);
    kernel_pdb = parse_pdb_file(xname, PDB_DEFAULT);
    set_ntoskrnl_pdb(kernel_pdb);
    free(wstring);
    free(wpdb_url);
    free(pdb_url);
    free(xname);

}


// Extract the OS Version
pe_offset =  memgrab_32_LE(debug_data->pKernBase + 0x3c);
hdr_base = debug_data->pKernBase + pe_offset + 0x18;
debug_data->MajorOSVersion = memgrab_16_LE(hdr_base + 0x28 );
debug_data->MinorOSVersion = memgrab_16_LE(hdr_base + 0x2a );

data_segment = get_pdb_section(kernel_pdb,".data",5)->virtual_address;
nbr_sections = memgrabke_16_LE(hdr_base + 0x02);
for (i = 0; i < nbr_sections; i++) {

    test_section = memgrabke_stringn(hdr_base +
                   (debug_data->x64?0xf0:0xe0) + i * 0x28 + 0,8);
    if (!strcmp(test_section,".data")) {

        data_segment = memgrabke_32_LE(hdr_base +
                       (debug_data->x64?0xf0:0xe0) +
                       i * 0x28 + 0x0c);

    }

    free(test_section);

}
```

```
      // Get the location of the PsLoadedModuleList symbol
      woffsetx = get_pdb_symbol(kernel_pdb,"PsLoadedModuleList",18)->offset;
      wptrx = get_pdb_symbol_loc(kernel_pdb,woffsetx,data_segment);
      wlocx = debug_data->pKernBase + (((uint64_t) wptrx) & 0x00000000ffffffff);
      // Go to the location of the PsLoadedModuleList symbol and get the
      // virtual address of the Loaded Modules List
      if (debug_data->x64)
        debug_data->PsLoadedModuleList = memgrab_64_LE(wlocx);
      else
        debug_data->PsLoadedModuleList = ((uint64_t) memgrab_32_LE(wlocx)) &
                    0x00000000ffffffff;

      // Get the location of the ObpObjectType symbol
      woffsetx = get_pdb_symbol(kernel_pdb,"ObpTypeObjectType",17)->offset;
      wptrx = get_pdb_symbol_loc(kernel_pdb,woffsetx,data_segment);
      // Go to the location of the ObpObjectType symbol and get the starting
      //  address of the
      // Object Type Table
      wlocx = debug_data->pKernBase + (((uint64_t) wptrx) &
             0x00000000ffffffff);
      if (debug_data->x64)
        whead = memgrab_64_LE(wlocx) - toffset;
      else
        whead = (memgrab_32_LE(wlocx) - toffset) & 0x00000000ffffffff;

  }


  // If the object type is no longer stored in the object_header (e.g.,
  // Windows 7), build the Object Type Table in the meta-data
  if (!(get_pdb_fld_exists("_OBJECT_HEADER.Type"))) {

if (debug_data->ObpTypeObjectType != 0)
         whead = debug_data->ObpTypeObjectType;
      mem_pointer = whead;
      done = 0;
      i = 0;
      while (!done) {

          if (debug_data->x64)
            widx = (uint8_t) keget_pdb_type( "_OBJECT_TYPE.Index", mem_pointer
                 + toffset);
          else
            widx = (uint8_t) keget_pdb_type( "_OBJECT_TYPE.Index", mem_pointer
```

```
       + toffset);
if ((widx > 0) && (widx < 255)) {

    if (debug_data->x64)
      debug_data->objTypeTable[widx] = mem_pointer + toffset;
    else
      debug_data->objTypeTable[widx] = mem_pointer + toffset;

}

else {
    if (i > 0) break;
    i = 1;
    if (debug_data->x64)
      printf("**Unknown object type index** %llx (phys: %llx)\n",
                  mem_pointer + toffset,kevtop(mem_pointer +
                  toffset));
    else
      printf("**Unknown object type index** %llx (phys: %llx)\n",
                 mem_pointer + toffset,kevtop(mem_pointer +
                 toffset));
    if (MYDEBUG) {

        if (debug_data->x64) {

            wtypename.Length        = (uint16_t) keget_pdb_type(
                  "_OBJECT_TYPE.Name.Length", mem_pointer +
                  toffset);
            wtypename.MaximumLength = (uint16_t) keget_pdb_type(
                  "_OBJECT_TYPE.Name.MaximumLength", mem_pointer  +
                  toffset);
            wtypename.Buffer        = keget_pdb_type(
                  "_OBJECT_TYPE.Name.Buffer", mem_pointer  +
                  toffset);
            if ((wtypename.Length < 40) && (wtypename.Length > 3)) {

                wstring = memgrabke_unicode_LE(wtypename);
                printf("Name = %s    Buffer = %llx (phys: %llx)\n",
                   wstring,wtypename.Buffer,
                   kevtop(wtypename.Buffer));
                free(wstring);

            }

            else
```

```
                    printf("Buffer = %llx (phys: %llx)\n",
                        wtypename.Buffer,kevtop(wtypename.Buffer));

                }
                else {

                    wtypename.Length        = (uint16_t) keget_pdb_type(
                         "_OBJECT_TYPE.Name.Length", mem_pointer + toffset);
                    wtypename.MaximumLength = (uint16_t) keget_pdb_type(
                         "_OBJECT_TYPE.Name.MaximumLength", mem_pointer  + toffset);
                    wtypename.Buffer        = keget_pdb_type(
                         "_OBJECT_TYPE.Name.Buffer", mem_pointer  + toffset);
                    if ((wtypename.Length < 40) && (wtypename.Length > 3)) {

                        wstring = memgrabke_unicode_LE(wtypename);
                        printf("Name = %s    Buffer = %llx (phys: %llx)\n",
                                 wstring,wtypename.Buffer,
                                 kevtop(wtypename.Buffer));
                        free(wstring);

                    }

                    else
                    printf("Buffer = %llx (phys: %llx)\n",
                           wtypename.Buffer,kevtop(wtypename.Buffer));

                }


            }


        }

        if (debug_data->x64)
          mem_pointer = memgrabke_64_LE(mem_pointer);
        else
          mem_pointer = ((uint64_t) memgrabke_32_LE(mem_pointer)) &
                        0x00000000ffffffff;
        done = (mem_pointer == whead);

    }


}
```

```
    osversion[0] = 0;
    if ( (debug_data->MajorOSVersion == 4) && (debug_data->MinorOSVersion == 0))
                    strcpy_s(osversion, 9, "Win95");
    if ( (debug_data->MajorOSVersion == 4) && (debug_data->MinorOSVersion == 10))
                    strcpy_s(osversion, 9, "Win98");
    if ( (debug_data->MajorOSVersion == 4) && (debug_data->MinorOSVersion == 90))
                    strcpy_s(osversion, 9, "WinME");
    if ( (debug_data->MajorOSVersion == 3) && (debug_data->MinorOSVersion == 51))
                    strcpy_s(osversion, 9, "WinNT351");
    if ( (debug_data->MajorOSVersion == 4) && (debug_data->MinorOSVersion == 0) )
                    strcpy_s(osversion, 9, "WinNT4");
    if ( (debug_data->MajorOSVersion == 5) && (debug_data->MinorOSVersion == 0) )
                    strcpy_s(osversion, 9, "Win2000");
    if ( (debug_data->MajorOSVersion == 5) && (debug_data->MinorOSVersion == 1) )
                    strcpy_s(osversion, 9, "WinXP");
    if ( (debug_data->MajorOSVersion == 5) && (debug_data->MinorOSVersion == 2) )
                    strcpy_s(osversion, 9, "Win2003");
    if ( (debug_data->MajorOSVersion == 6) && (debug_data->MinorOSVersion == 0) )
                    strcpy_s(osversion, 9, "Vista");
    if ( (debug_data->MajorOSVersion == 6) && (debug_data->MinorOSVersion == 1) )
                    strcpy_s(osversion, 9, "Windows7");
    debug_data->OSVersionString = malloc(sizeof(char) * 9);
    strcpy_s(debug_data->OSVersionString, 9, osversion);
    printf("\nKDBG (%llx) ",debug_data->KDBG_loc);
    printf("\t%s  ",debug_data->MachineString);
    printf("%s ",osversion);
    printf("Ver %i.%i (build: %i) ",
            debug_data->MajorOSVersion,
            debug_data->MinorOSVersion,debug_data->MinorVersion);
    printf("\tKernel Base = %llx ",debug_data->KernBase);
    printf("(PageDirectoryBase = %llx) \n\n",debug_data->kDTB);
    return 0;

}


/*******************************************************************************/

/************************************/
/* FIND PROCESS AND REGISTRY RECORDS */
/************************************/

/**********************************************/
```

```
/* Search the memory dump for processes and hives */
/*************************************************/
int load_info() {

    uint32_t data_from_memdump = 0;  // Placeholder for pattern matching memory dump
    uint32_t data_from_memdump2 = 0; // Placeholder for pattern matching memory dump
    uint32_t data_from_memdump3 = 0; // Placeholder for pattern matching memory dump
    uint16_t data_from_memdump4 = 0; // Placeholder for pattern matching memory dump

    uint32_t wsize;         // Size of _DISPATCHER_HEADER; used for constructing
                            // _EPROCESS pattern matching string
    uint32_t *proc_string;  // To date (2K,Srv03,XP,Vista,7) Type = 03 for processes;
                            // used for constructing _EPROCESS pattern matching string
    uint32_t *wproc_string; // Size of _KPROCESS, used for constructing _EPROCESS
                            // pattern matching string
    uint32_t woffset;       // Offset of the hive linked list from the top of _CMHIVE;
                            // used to get to the top when traversing the linked list
    uint32_t phive_head = 0xbee0bee0u;   // Pattern match for hives is 0xbee0bee0
    uint32_t hive_pattern = 0xe0bee0beu; // Pattern match for hives is 0xbee0bee0

    uint64_t wloc = 0;           // Pointer into memory dump
    uint64_t end_of_file;        // Size of the memory dump
    int percent_count = 0;       // Progress bar

    hive_list_type *whive_list;  // Head of the hive list
    uint64_t fstart_address;     // The location of the next hive in the linked list;
                                 // used for adding new hives to the hive list
    hive_list_type *htmp, *hhtmp, *wtail;    // Used to traverse the the hive list
    uint64_t new_hive;      // Used to add new hives found by traversing
                            // the CMHIVE linked list
    short addit;            // True if a new hive was found to be added

    int val_change;         // Used to sort processes by PID (while loop
                            // continues until no valid change occurs)
    uint32_t proc_count;    // Count of # of processes -- used to handle
                            // broken linked lists when determining hidden processes
    process_list *whead;    // Iterator process list; used to find hidden processes
    int gotone;             // Used to remove duplicate instances of the
                            // same process (same PID, PPID, and executable)
    process_list *tmp2, *old; // iterator through process list; used to remove
                            //  duplicate instances of the same process

    uint32_t ctr1;          // Used for labeling hives
    uint16_t wname_length;  // Used for labeling hives
    char *xname, *xname1, *xname2;  // Folder name at a certain level of the hive,
```

```
                                // Used for labeling hives
uint32_t *num_subkeys, *num_subkeys1;    // #Subkeys (folders) at the next
                                         // level of the hive,
                                         // Used for labeling hives
uint32_t *slist_cell, *slist_cell1;      // Location of subkey (folder) list
                                         // at next
                                         // level of the hive, Used for
                                         // labeling hives
unsigned int cell_offset1, cell_offset2; // Location of a particular subkey;
                                         // Used for labeling hives
uint64_t mem_pointer, mem_pointer1, mem_pointer2, mem_pointer3;
                                         // Location of a particular
                                         // subkey; Used for labeling hives
uint64_t whive_location;                 // Pointer to a hive record in the
                                         // hive list; Used for labeling hives

process_list *tmp;
thread_list     *ttmp;
uint32_t ctr, i;

whive_list = NULL;

// Determine how big the memory dump is
end_of_file = get_mem_size();
if (end_of_file < 100)
{

    printf("Houston we have a problem... image file is too small\n");
    return 1;

}


printf ("\nSearching for processes and hives in memory dump\n");

// Construct the pattern for the EPROCESS RECORD
// _DISPATCHER_HEADER is at the beginning of _KPROCESS which is at the
// beginning of _EPROCESS so it's a good string to use
wsize = (uint32_t) get_pdb_type_sizeof( "_DISPATCHER_HEADER");
wsize = (uint32_t) ( (uint32_t) wsize / 4 ) + (wsize % 4?1:0);
// This assumes that ProcessType is 0x03.  This has been checked using ke.h
// and procobj.c
//proc_string = malloc(sizeof(uint32_t) * wsize);
//wproc_string = malloc(sizeof(uint32_t) * wsize);
proc_string = get_pdb_type_value( "_DISPATCHER_HEADER.Type",0x03);
```

```
wproc_string = get_pdb_type_value( "_DISPATCHER_HEADER.Size",((uint32_t)
                  get_pdb_type_sizeof("_KPROCESS") / 0x04) );
for (i = 0; i < wsize; i++) {
    proc_string[i] += wproc_string[i];
}


woffset = (uint32_t) get_pdb_type_offset( "_CMHIVE.HiveList");

// Traverse the memory dump looking for processes and hives
mem_move(0);
while (wloc < end_of_file)
{

    // Progress bar
    if ( ((percent_count * 5) == 100) && ((wloc / (end_of_file/100)) >=
         (percent_count * 5) )) {

        fprintf(stderr,"%02ld%%\n", ( (wloc / (end_of_file/100)) ) );
        percent_count++;

    }

    else if ( (wloc / (end_of_file/100)) >= (percent_count * 5) ) {

        fprintf(stderr,"%02ld%%--", ( (wloc / (end_of_file/100)) ) );
        percent_count++;

    }


    // Get the next 8 bytes to pattern match against
    data_from_memdump = memgrab_32_BE(0); wloc += 4;
    data_from_memdump2 = memgrab_32_BE(0); wloc += 4;

    // Note: this assumes that TYPE and SIZE are stored in the first 8 bytes
   //        of _DISPATCHER_HEADER
    //       and that DISPATCHER_HEADER is the first field in _KPROCESS
    // If you find a process, populate the process list with its location
    if ((data_from_memdump == proc_string[0]) &&
        (data_from_memdump2 == proc_string[1]))
    {

        if (MYDEBUG) printf("**load_info** found Process %llx\n",wloc - 8);
        if ((tmp = malloc(sizeof(process_list))) == NULL ) {
```

```
            (void)exit(EXIT_FAILURE);
        }

        tmp->proc_eprocess_location = wloc - 8;
        tmp->next = proc_list;
        tmp->prev = NULL;
        tmp->users = NULL;
        tmp->sockets = NULL;
        tmp->eprocess_in_link_list = 0;
        proc_list = tmp;
        if (tmp->next != NULL) tmp->next->prev = tmp;

    }


    // If you find a hive, populate the hive list with the location of the
    // hive in its Forward Link linked list of hives
    if ((data_from_memdump == hive_pattern) )
    {

        if (!debug_data->x64)
          fstart_address = memgrab_32_LE(wloc - 8 + woffset) - woffset;
        else
          fstart_address = memgrab_64_LE(wloc - 8 + woffset) - woffset;
        // Add hive to link list (only populates hive location in image file)
        // Note: this is needed because you just did a memgrab which moved
        // the file pointer
        mem_move(wloc);
        whive_list = add_new_hive(whive_list, fstart_address);

    }


}


// Traverse the linked list of hives and instantiate each one
printf("Instantiating Hives \n");
htmp = whive_list;
wtail = NULL;
while (htmp != NULL) {

    hhtmp = htmp->next;
    if (grab_hive(htmp))
    wtail = htmp;
```

```
    else
    if (wtail == NULL)
    whive_list = hhtmp;
    htmp = hhtmp;

}

free(proc_string);
free(wproc_string);

// Go through the linked list of hives (in the CMHIVE structure) and see if
// any were missed
printf("Looking for more hives \n");
while (wtail != NULL) {


    new_hive = wtail->Hive.HiveList.Blink - woffset;
    addit = 1;
    htmp = whive_list;
    while (htmp != NULL) {

        if (htmp->hive_location == new_hive) {

            addit = 0;
            break;

        }

        htmp = htmp->next;

    }


    if (addit) {

        data_from_memdump = memgrabke_32_LE(new_hive);
        if (data_from_memdump == phive_head) {

           // Add hive to link list (only populates hive location in image file)
            whive_list = add_new_hive(whive_list, new_hive);
            grab_hive(whive_list);

        }
```

```
    }


    new_hive = wtail->Hive.HiveList.Flink -
               get_pdb_type_offset( "_CMHIVE.HiveList.Flink");
    addit = 1;
    htmp = whive_list;
    while (htmp != NULL) {

        if (htmp->hive_location == new_hive) {

            addit = 0;
            break;

        }

        htmp = htmp->next;

    }


    if (addit) {

        data_from_memdump = memgrabke_32_LE(new_hive);

        if (data_from_memdump == phive_head) {

           // Add hive  to link list (only populates hive location in image file)
            whive_list = add_new_hive(whive_list, new_hive);
            grab_hive(whive_list);

        }


    }


    wtail = wtail->prev;

}


// For each hive, find it's name by being to go through the hive structure
printf("Labeling hives\n");
htmp = whive_list;
```

```
while (htmp != NULL) {


    // Find the name of this hive
    mem_pointer = get_cell_loc(htmp,htmp->Hive.HHive.BaseBlock.RootCell);
    wname_length = (int) keget_pdb_type  ( "_CM_KEY_NODE.NameLength",
                   mem_pointer);
    xname =  keget_pdb_type_stringn  ( "_CM_KEY_NODE.Name", mem_pointer,
                   wname_length);

    // If this is the root hive
    if (!strcmp(xname,"REGISTRY")) {

        htmp->name = (char *) malloc( (strlen(xname) + 1) * sizeof(char));
        strcpy_s(htmp->name, strlen(xname) + 1, xname);
        // Get the sublists (although they're arrays, only the first entry
        //appears to be used)
        num_subkeys = (uint32_t *) keget_pdb_type_array(
                       "_CM_KEY_NODE.SubKeyCounts",
                       mem_pointer); // This better be 2  (MACH, USER)
        slist_cell = (uint32_t *) keget_pdb_type_array(
                      "_CM_KEY_NODE.SubKeyLists", mem_pointer);
        mem_pointer = get_cell_loc(htmp,slist_cell[0]); // this better point
                                                        // to an lf data structure
        for (ctr = 0; ctr < num_subkeys[0]; ctr++) {


            // Unfortunately, it does not appear that the structure of the
            // lf record is in ntoskrnl_pdb
            // it begins like CM_KEY_INDEX; however, the List array is
            // composed of 2 4 byte fields
            // - 0x00 offset to field, 0x04 first 4 bytes of name
            cell_offset1 = memgrabke_32_LE(mem_pointer + 0x4 + (0x8 * ctr));
            mem_pointer1 = get_cell_loc(htmp,cell_offset1); // this better point
                                                    // to an nk data structure

            // the lf record should point to a Key Node record (one for
            // MACHINE and one for USERS)
            wname_length = (uint16_t) keget_pdb_type  (
                           "_CM_KEY_NODE.NameLength", mem_pointer1);
            xname1 =  keget_pdb_type_stringn  ( "_CM_KEY_NODE.Name",
                           mem_pointer1, wname_length);

            // now get the lf records for the registries under either
            // MACHINE or USERS
```

```
num_subkeys1 = (uint32_t *) keget_pdb_type_array(
                          "_CM_KEY_NODE.SubKeyCounts",
        mem_pointer1); // This should be 5 for MACH and 7 for USER
slist_cell1 = (uint32_t *) keget_pdb_type_array(
              "_CM_KEY_NODE.SubKeyLists", mem_pointer1);
mem_pointer2 = get_cell_loc(htmp,slist_cell1[0]); // this better point
                                          // to an lf data structure

// now get the hives
for (ctr1 = 0;ctr1 < num_subkeys1[0]; ctr1++) {


    cell_offset2 = memgrabke_16_LE(mem_pointer2 + 0x4 + (0x8 * ctr1));
    mem_pointer3 = get_cell_loc(htmp,cell_offset2); // this better
                               // point to an nk data structure

    wname_length = (uint16_t) keget_pdb_type  (
                        "_CM_KEY_NODE.NameLength",
             mem_pointer3);
    xname2 =  keget_pdb_type_stringn  ( "_CM_KEY_NODE.Name",
             mem_pointer3,
             wname_length);

    whive_location = keget_pdb_type  (
              "_CM_KEY_NODE.ChildHiveReference.KeyHive",
                    mem_pointer3);
    hhtmp = whive_list;
    while (hhtmp != NULL) {

        if (hhtmp->hive_location == whive_location) {

            hhtmp->name = (char *) malloc( (strlen(xname) +
                      strlen(xname1) +
                      strlen(xname2) + 3) * sizeof(char));
            strcpy_s(hhtmp->name, strlen(xname) + strlen(xname1)
                      + strlen(xname2) + 3, xname);
            strcat_s(hhtmp->name,strlen(xname) + strlen(xname1)
                      + strlen(xname2) + 3, "/");
            strcat_s(hhtmp->name,strlen(xname) + strlen(xname1)
                      + strlen(xname2) + 3, xname1);
            strcat_s(hhtmp->name,strlen(xname) + strlen(xname1)
                      + strlen(xname2) + 3, "/");
            strcat_s(hhtmp->name,strlen(xname) + strlen(xname1)
                      + strlen(xname2) + 3, xname2);
            break;
```

```
                        }

                        hhtmp = hhtmp->next;

                    }

                    free(xname2);

                }

                free(xname1);
                free(num_subkeys1);
                free(slist_cell1);

            }

            free(num_subkeys);
            free(slist_cell);
            break;

        }

        free(xname);
        htmp = htmp->next;

    }

    // Make the hive list globally known
    set_hive_list(whive_list);


    // Traverse the linked list of processes and instantiate each one
    printf ("Enumerating process structures.\n");
    ctr = 0;
    tmp = proc_list;
    proc_count = 0;
    while (tmp != NULL)
    {
        grab_eprocess (tmp);
        // If the record is bogus (e.g., the Page Directory Table can't have a
        // physical address so low), get rid of it
        if (tmp->proc_eprocess.Pcb.DirectoryTableBase < 0x1000) {

            if (tmp->next != NULL) tmp->next->prev = tmp->prev;
```

```
        if (tmp->prev != NULL) tmp->prev->next = tmp->next;
        if (proc_list == tmp) proc_list = tmp->next;
        old = tmp;
        tmp = tmp->next;
        free(old);
        continue;

    }

    grab_peb(tmp);
    tmp = tmp->next;
    proc_count++;

}



// Resort the processes so the lowest (first) IDs show up first
printf ("Sorting processes by PID\n");
do     {

    process_list *old1;
    process_list *old2;
    tmp = proc_list;
    val_change = 0;
    while (tmp->next != NULL) {

        if (tmp->next == NULL) break;
        if (tmp->proc_eprocess.UniqueProcessId >
            tmp->next->proc_eprocess.UniqueProcessId)
        {

            if (proc_list == tmp) proc_list = tmp->next;
            val_change++;
            old1 = tmp;
            old2 = tmp->next;


            old2->prev = old1->prev;
            if (old1->prev != 0) old1->prev->next = old2;
            old1->next = old2->next;
            if (old2->next != 0) old2->next->prev = old1;
            old2->next = old1;
            old1->prev = old2;
```

```
        }
        else {

            tmp = tmp->next;

        }


    }


}
while (val_change);

// The goal of the next section is to determine which of the found processes
// are in an EPROCESS  linked list.
// While in theory, there should be only one EPROCESS linked list, in
// reality there may be more than one (e.g., due to
// processes continuing in memory from previous executions).
// So to handle this, we go through the EPROCESS linked list for each found
// process and look for  all of the found processes.
// While this is wasteful (since at most there will only be two or three
// linked lists and the  rest of the searches are redundant), it is
// complete.
printf ("Checking for detached/ hidden processes\n");
woffset = (uint32_t) get_pdb_type_offset( "_EPROCESS.ActiveProcessLinks");
{

    uint64_t        bend_address;
    uint64_t        fend_address;
    uint64_t        next_address;
    uint64_t        prev_address;
    uint64_t        pid;

    whead = proc_list;

    // For each process that was found
    while (whead != NULL) {

        fend_address = whead->proc_eprocess.ActiveProcessLinks.Blink;
        bend_address = whead->proc_eprocess.ActiveProcessLinks.Flink;
        next_address = whead->proc_eprocess.ActiveProcessLinks.Flink;
        prev_address = whead->proc_eprocess.ActiveProcessLinks.Blink;
        ctr = 0;
        // Go through the entire EPROCESS linked list (FLINK) that the
```

```
// current process is a member of
while ((next_address != fend_address) && (next_address != 0x0)){

    pid = pget_pdb_type("_EPROCESS.UniqueProcessId", next_address -
            woffset,
            whead->proc_eprocess.Pcb.DirectoryTableBase);
    next_address = pget_pdb_type("_EPROCESS.ActiveProcessLinks.Flink",
            next_address - woffset,
            whead->proc_eprocess.Pcb.DirectoryTableBase);
    tmp = proc_list;
    // mark any processes that are in the EPROCESS linked list as
    // IN LINKED LIST
    while (tmp->next != NULL) {

        if (tmp->proc_eprocess.UniqueProcessId == pid) {

            tmp->eprocess_in_link_list = 1;
            break;

        }

        tmp=tmp->next;

    }

    ctr++;
    if (ctr > proc_count) break;

}

ctr = 0;
// Go through the entire EPROCESS linked list (BLINK) that the
// current process is a member of
while ((prev_address != bend_address) && (prev_address != 0x0)){

    pid = pget_pdb_type("_EPROCESS.UniqueProcessId", prev_address -
           woffset,
           whead->proc_eprocess.Pcb.DirectoryTableBase);
    next_address = pget_pdb_type("_EPROCESS.ActiveProcessLinks.Blink",
           prev_address - woffset,
           whead->proc_eprocess.Pcb.DirectoryTableBase);
    tmp = proc_list;
    // mark any processes that are in the EPROCESS linked list as
    // IN LINKED LIST
    while (tmp->next != NULL) {
```

```
            if (tmp->proc_eprocess.UniqueProcessId == pid) {

                tmp->eprocess_in_link_list = 1;
                break;

            }

            tmp=tmp->next;

        }

        ctr++;
        if (ctr > proc_count) break;

    }


    whead=whead->next;

}


// Removing duplicates (i.e., same executable, same PID, and same parent PID)
printf ("Cleaning up\n");
tmp = proc_list;
while (tmp != NULL)
{

    if (!tmp->eprocess_in_link_list) {

        tmp2 = proc_list;
        gotone = 0;
        while (tmp2 != NULL) {

            if ( (tmp->proc_eprocess.UniqueProcessId ==
                   tmp2->proc_eprocess.UniqueProcessId) &&
                  (tmp->proc_eprocess.InheritedFromUniqueProcessId ==
                   tmp2->proc_eprocess.InheritedFromUniqueProcessId) &&
                   (!strcmp(tmp->proc_eprocess.ImageFileName2,
                    tmp2->proc_eprocess.ImageFileName2)) &&
                   (tmp2->eprocess_in_link_list) ) {

                if (tmp->prev != NULL)
                tmp->prev->next = tmp->next;
```

```
                if (tmp->next != NULL)
                tmp->next->prev = tmp->prev;
                old = tmp;
                tmp = tmp->next;
                free(old->proc_eprocess.ImageFileName2);
                free(old);
                gotone = 1;
                break;

            }

            tmp2 = tmp2->next;

        }

        if (!gotone)
        tmp = tmp->next;

    }

    else
    tmp = tmp->next;

  }



}


// At some point, I'll turn this back on
if (0) {

  printf("Searching for all threads.\n");
  tmp=proc_list;ctr = 0;
  printf("Searching process");
  while (tmp != NULL) {

    printf("--%i",ctr++);
    walk_thread_list(tmp,(uint32_t)
           tmp->proc_eprocess.ThreadListHead.Blink, (uint32_t)
           tmp->proc_eprocess.ThreadListHead.Flink);
    ttmp = thr_list;
    while (ttmp != NULL) {
```

```
                uint32_t        thread_pid;
                process_list    *ptmp;
                //grab_ethread(tmp,ttmp);
                // 0x9c is the location of UniqueProcessID in EPROCESS in WIN2K
                thread_pid = memgrabp_32_LE(tmp->proc_eprocess.Pcb.DirectoryTableBase,
                     (ttmp->thr_ethread.ThreadsProcess + 0x9c));
                ptmp = proc_list;
                while (ptmp != NULL) {

                    if ((ptmp->proc_eprocess.UniqueProcessId == thread_pid) &&
                         (ptmp->eprocess_in_link_list > 0))    {

                        ttmp->process = ptmp;
                        break;

                    }

                    ptmp=ptmp->next;

                }

                ttmp=ttmp->next;

            }

            tmp=tmp->next;

        }

        printf("\n");

    }


    return 0;

}


/********************************************************************************/

/*********************************/
/* INSTANIATE PROCESS INFORMATION */
/*********************************/
```

```
/************************/
/* Instantiate a process */
/************************/
void grab_eprocess (process_list *tmp)
{

    uint64_t * wDirectoryTableBase;
    uint64_t wsession_ptr;

    tmp->proc_eprocess.Pcb.KernelTime               = get_pdb_type  (
              "_EPROCESS.Pcb.KernelTime", tmp->proc_eprocess_location);
    tmp->proc_eprocess.Pcb.UserTime                 = get_pdb_type  (
              "_EPROCESS.Pcb.UserTime", tmp->proc_eprocess_location);
    tmp->proc_eprocess.Pcb.BasePriority             = (signed char) get_pdb_type
              ( "_EPROCESS.Pcb.BasePriority",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.CreateTime.LowPart           = (uint32_t) get_pdb_type  (
              "_EPROCESS.CreateTime.LowPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.CreateTime.HighPart          = (int32_t) get_pdb_type  (
              "_EPROCESS.CreateTime.HighPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.UniqueProcessId              = get_pdb_type  (
              "_EPROCESS.UniqueProcessId",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.PeakVirtualSize              = get_pdb_type  (
              "_EPROCESS.PeakVirtualSize",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ObjectTable                  = get_pdb_type  (
              "_EPROCESS.ObjectTable",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.Token                        = get_pdb_type  (
              "_EPROCESS.Token",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ImageFileName2               = get_pdb_type_string  (
              "_EPROCESS.ImageFileName",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ReadOperationCount.LowPart   = (uint32_t) get_pdb_type (
              "_EPROCESS.ReadOperationCount.LowPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ReadOperationCount.HighPart  = (int32_t) get_pdb_type  (
              "_EPROCESS.ReadOperationCount.HighPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.WriteOperationCount.LowPart  = (uint32_t) get_pdb_type  (
              "_EPROCESS.WriteOperationCount.LowPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.WriteOperationCount.HighPart = (int32_t) get_pdb_type  (
              "_EPROCESS.WriteOperationCount.HighPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.OtherOperationCount.LowPart  = (uint32_t) get_pdb_type  (
              "_EPROCESS.OtherOperationCount.LowPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.OtherOperationCount.HighPart = (int32_t) get_pdb_type  (
              "_EPROCESS.OtherOperationCount.HighPart",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.InheritedFromUniqueProcessId = get_pdb_type  (
              "_EPROCESS.InheritedFromUniqueProcessId",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.DeviceMap                    = get_pdb_type  (
              "_EPROCESS.DeviceMap",  tmp->proc_eprocess_location);
```

```
    tmp->proc_eprocess.ActiveProcessLinks.Flink     = get_pdb_type  (
              "_EPROCESS.ActiveProcessLinks.Flink",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ActiveProcessLinks.Blink     = get_pdb_type  (
              "_EPROCESS.ActiveProcessLinks.Blink",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ThreadListHead.Flink         = get_pdb_type  (
              "_EPROCESS.ThreadListHead.Flink",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.ThreadListHead.Blink         = get_pdb_type  (
              "_EPROCESS.ThreadListHead.Blink",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.PriorityClass                = (unsigned char) get_pdb_type  (
              "_EPROCESS.PriorityClass",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.Peb                          = get_pdb_type  (
              "_EPROCESS.Peb",  tmp->proc_eprocess_location);
    wDirectoryTableBase                                = get_pdb_type_array  (
              "_EPROCESS.Pcb.DirectoryTableBase",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.Pcb.DirectoryTableBase = wDirectoryTableBase[0];
    wsession_ptr                                    = get_pdb_type  (
              "_EPROCESS.Session", tmp->proc_eprocess_location);
    if (tmp->proc_eprocess.Pcb.DirectoryTableBase != 0)
    tmp->proc_eprocess.SessionId                    = (uint32_t) pget_pdb_type
          ( "_MM_SESSION_SPACE.SessionId", wsession_ptr,
              tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_eprocess.Win32WindowStation            = get_pdb_type  (
              "_EPROCESS.Win32WindowStation",tmp->proc_eprocess_location);
    add_ptov_dtb(tmp->proc_eprocess.UniqueProcessId,
                tmp->proc_eprocess.Pcb.DirectoryTableBase);
    free(wDirectoryTableBase);

}


void kegrab_eprocess (process_list *tmp)
{

    uint64_t * wDirectoryTableBase;
    uint64_t wsession_ptr;

    tmp->proc_eprocess.Pcb.KernelTime               = keget_pdb_type  (
         "_EPROCESS.Pcb.KernelTime", tmp->proc_eprocess_location);
    tmp->proc_eprocess.Pcb.UserTime                 = keget_pdb_type  (
         "_EPROCESS.Pcb.UserTime", tmp->proc_eprocess_location);
    tmp->proc_eprocess.Pcb.BasePriority             = (signed char)
keget_pdb_type   (
         "_EPROCESS.Pcb.BasePriority",  tmp->proc_eprocess_location);
    tmp->proc_eprocess.CreateTime.LowPart           = (uint32_t) keget_pdb_type
         ( "_EPROCESS.CreateTime.LowPart",  tmp->proc_eprocess_location);
```

```
tmp->proc_eprocess.CreateTime.HighPart          = (int32_t) keget_pdb_type
     ( "_EPROCESS.CreateTime.HighPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.UniqueProcessId              = keget_pdb_type  (
     "_EPROCESS.UniqueProcessId",  tmp->proc_eprocess_location);
tmp->proc_eprocess.PeakVirtualSize              = keget_pdb_type  (
     "_EPROCESS.PeakVirtualSize",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ObjectTable                  = keget_pdb_type  (
     "_EPROCESS.ObjectTable",  tmp->proc_eprocess_location);
tmp->proc_eprocess.Token                        = keget_pdb_type  (
     "_EPROCESS.Token",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ImageFileName2               = keget_pdb_type_string  (
     "_EPROCESS.ImageFileName",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ReadOperationCount.LowPart   = (uint32_t) keget_pdb_type
     ( "_EPROCESS.ReadOperationCount.LowPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ReadOperationCount.HighPart  = (int32_t) keget_pdb_type
     ( "_EPROCESS.ReadOperationCount.HighPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.WriteOperationCount.LowPart  = (uint32_t) keget_pdb_type
     ( "_EPROCESS.WriteOperationCount.LowPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.WriteOperationCount.HighPart = (int32_t) keget_pdb_type
     ( "_EPROCESS.WriteOperationCount.HighPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.OtherOperationCount.LowPart  = (uint32_t) keget_pdb_type
     ( "_EPROCESS.OtherOperationCount.LowPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.OtherOperationCount.HighPart = (int32_t) keget_pdb_type
     ( "_EPROCESS.OtherOperationCount.HighPart",  tmp->proc_eprocess_location);
tmp->proc_eprocess.InheritedFromUniqueProcessId = keget_pdb_type  (
     "_EPROCESS.InheritedFromUniqueProcessId",  tmp->proc_eprocess_location);
tmp->proc_eprocess.DeviceMap                    = keget_pdb_type  (
     "_EPROCESS.DeviceMap",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ActiveProcessLinks.Flink     = keget_pdb_type  (
     "_EPROCESS.ActiveProcessLinks.Flink",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ActiveProcessLinks.Blink     = keget_pdb_type  (
     "_EPROCESS.ActiveProcessLinks.Blink",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ThreadListHead.Flink         = keget_pdb_type  (
     "_EPROCESS.ThreadListHead.Flink",  tmp->proc_eprocess_location);
tmp->proc_eprocess.ThreadListHead.Blink         = keget_pdb_type  (
     "_EPROCESS.ThreadListHead.Blink",  tmp->proc_eprocess_location);
tmp->proc_eprocess.PriorityClass                = (unsigned char)
       keget_pdb_type  ( "_EPROCESS.PriorityClass",  tmp->proc_eprocess_location);
tmp->proc_eprocess.Peb                          = keget_pdb_type  (
     "_EPROCESS.Peb",  tmp->proc_eprocess_location);
wDirectoryTableBase                                = keget_pdb_type_array  (
     "_EPROCESS.Pcb.DirectoryTableBase",  tmp->proc_eprocess_location);
tmp->proc_eprocess.Pcb.DirectoryTableBase = wDirectoryTableBase[0];
wsession_ptr                                    = keget_pdb_type  (
     "_EPROCESS.Session", tmp->proc_eprocess_location);
```

```
    if (tmp->proc_eprocess.Pcb.DirectoryTableBase != 0)
    tmp->proc_eprocess.SessionId                    = (uint32_t) pget_pdb_type
         ( "_MM_SESSION_SPACE.SessionId", wsession_ptr,
         tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_eprocess.Win32WindowStation           = keget_pdb_type  (
         "_EPROCESS.Win32WindowStation",tmp->proc_eprocess_location);
    add_ptov_dtb(tmp->proc_eprocess.UniqueProcessId,
                 tmp->proc_eprocess.Pcb.DirectoryTableBase);
    free(wDirectoryTableBase);

}


/*****************************************/
/* Instantiate a process environment bloc */
/*****************************************/
void grab_peb (process_list *tmp)
{

    tmp->proc_peb.Ldr                = pget_pdb_type  (
         "_PEB.Ldr", tmp->proc_eprocess.Peb,
         tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.ProcessParameters  = (uint32_t) pget_pdb_type  (
         "_PEB.ProcessParameters",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.OSMajorVersion     = (uint32_t) pget_pdb_type  (
         "_PEB.OSMajorVersion",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.OSMinorVersion     = (uint32_t) pget_pdb_type  (
         "_PEB.OSMinorVersion",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.OSPlatformID       = (uint32_t) pget_pdb_type  (
         "_PEB.OSPlatformId",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.OSCSDVersion       = (uint16_t) pget_pdb_type  (
         "_PEB.OSCSDVersion",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.OSBuildNumber      = (uint16_t) pget_pdb_type  (
         "_PEB.OSBuildNumber",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    tmp->proc_peb.NumberOfProcessors = (uint32_t) pget_pdb_type  (
         "_PEB.NumberOfProcessors",
         tmp->proc_eprocess.Peb, tmp->proc_eprocess.Pcb.DirectoryTableBase);

}
```

```
/******************************************************************************/

/************************************/
/* FIND AND INSTANTIATE THE THREADS */ /* -- some day, I'll actually use this */
/************************************/
void walk_thread_list (process_list *ptmp, uint32_t blink, uint32_t flink)
{

    unsigned int forward_stop_addr = blink;
    unsigned int backward_stop_addr = flink;
    unsigned int curr_flink = flink;
    unsigned int curr_blink = blink;

    while ((curr_flink != 0x0)&&( curr_flink != forward_stop_addr))
    {

        add_new_thread(curr_flink - 0x240);
        add_new_thread(curr_flink - 0x1a4); // ??
        curr_flink = memgrabp_32_LE(ptmp->proc_eprocess.Pcb.DirectoryTableBase,
                     curr_flink);
        if (curr_flink == backward_stop_addr) break;

    }


    while ((curr_blink != 0x0)&&(curr_blink != backward_stop_addr))
    {

        add_new_thread(curr_blink - 0x240);
        add_new_thread(curr_blink - 0x1a4);
        // Since BLink points to the top of the ListEntry, you have to offset
        // past FLink to get to the next BLink
        curr_blink = memgrabp_32_LE(ptmp->proc_eprocess.Pcb.DirectoryTableBase,
                      (curr_blink + 0x4));
        if (curr_blink == forward_stop_addr) break;

    }


}

void add_new_thread (unsigned long ethread_location)
{
```

```
    thread_list *tmp;

    if ((tmp = malloc(sizeof(thread_list))) == NULL) {

        (void)exit(EXIT_FAILURE);

    }

    tmp->thread_list_location = ethread_location;
    tmp->next = thr_list;
    tmp->prev = NULL;
    thr_list = tmp;
    if (tmp->next != NULL) tmp->next->prev = tmp;

}


/******************************************************************************/

/*****************************************/
/* FIND NETWORK INFORMATION FOR PROCESSES */
/*****************************************/

/****************************/
/* Find Network Information */
/****************************/
int load_tcpip_data() {


    unicode_string wdrivername; // Driver found by traversing PsLoadedModuleList
    uint64_t wptr;              // Pointer to traverse PsLoadedModuleList
    uint64_t wend;              // Endpoint for traversing PsLoadedModuleList
    uint64_t wimage_base;       // Linear address of image base of driver

    char *pdb_url, *wpdb_url, *fname;  // URL needed for Ms Symbol Server to
                                       // retrieve tcpip.pdb
    pdb_symbol_t *wwptr;       // Pointer to the symbol_t struct for symbol of interest;

    uint32_t pe_offset;       // PE offset in the PE file
    uint32_t nbr_sections;    // Number of sections for the PE
    uint64_t hdr_base;        // RVA of the PE header
    uint32_t data_segment;    // RVA of the .data segment
    uint32_t rdata_segment;   // RVA of the .rdata segment
```

```
uint64_t addr_base;       // Pointer to entry in addrtable and tcbtable
uint64_t old_addr_base;   // Reminder of where we've been

uint8_t localIP1, localIP2, localIP3, localIP4; // Each bit of the IP address
uint8_t remotIP1, remotIP2, remotIP3, remotIP4; // Each bit of the IP address
uint16_t Protocol;                         // Protocol
uint16_t LocalPort,RemotePort;             // Local and Remote Ports
uint32_t pid;                             // Process ID
uint32_t RemoteAddress, LocalAddress;     // Local and Remote IP addresses (4 bytes)
uint8_t Ipv6;                             // IPv6 true/false

// Windows Vista and Windows 7 symbols
uint8_t isValid;
uint32_t PartitionTable_offset;   // unadjusted offset from beginning of section
uint32_t PartitionTable_offset_ptr; // adjusted RVA
uint64_t PartitionTable;          // linear address of table of TCP
                                  // established connections
uint32_t PartitionCount_offset;   // unadjusted offset from beginning of section
uint32_t PartitionCount_offset_ptr; // adjusted RVA
uint64_t PartitionCount;         // linear address of count of TCP
                                 // established connections
uint32_t TcpPortPool_offset;     // unadjusted offset from beginning of section
uint32_t TcpPortPool_offset_ptr; // adjusted RVA
uint64_t TcpPortPool;            // linear address of pool of TCP listeners
uint32_t UdpPortPool_offset;     // unadjusted offset from beginning of section
uint32_t UdpPortPool_offset_ptr; // adjusted RVA
uint64_t UdpPortPool;            // linear address of pool of UDP endpoints

// Windows XP symbols
uint32_t addrtable_offset;       // unadjusted offset from beginning of section
uint32_t addrtable_offset_ptr;   // adjusted RVA
uint64_t addrtable_address;      // linear address
uint64_t addrtable_size;         // unadjusted offset from beginning of section
                                 //  AND linear address
uint32_t addrtable_size2;        // unadjusted offset from beginning of section
                                 // AND linear address
uint32_t addrtable_size_ptr;     // adjusted RVA
uint32_t tcbtable_offset;        // unadjusted offset from beginning of section
uint32_t tcbtable_offset_ptr;   // adjusted RVA
uint64_t tcbtable_address;      // linear address
uint32_t tcbtable_size2;        // unadjusted offset from beginning of section
                                // AND linear address
uint64_t tcbtable_size;         // unadjusted offset from beginning of section
                                // AND linear address
uint32_t tcbtable_size_ptr;     // adjusted RVA
```

```
uint64_t PTable;               // Location of Partition Table
uint64_t HashTable;            // Hashtable of Partition Table
uint64_t TableEntries;         // Number of entries in Partition Table;
uint64_t Entry;                // An individual entry in the hash table
                               // (a pointer to an entry);
uint64_t IpInfo;               // The IpInfo structure pointed to by the hash table

uint64_t BitMapPtr;            // Pointer to the bitmap
uint64_t BitMapSize;           // Number of bits in the bitmap
uint64_t SIdx;                 // Special case to skip over the first bit
                               // of the bitmap
uint64_t idx;                  // Position in the bitmap
uint64_t pagenbr;              // the high byte of the bitmap index --
                               // points to the page where the IPInfo
                               // pointers are
uint64_t offset;               // the low byte of the bitmap index -- points
                               // to the offset in the page where the
                               // IPInfo pointers are

socket_list_type *wsocket;     // Structure to connect to the process
                               // info to store the network connection info
process_list *tmp;             // Pointer to traverse the process list
                               // to add in the network connection info

uint64_t wloc, tmp1, tmp2, tmp3, tmp4, j, k, p;
uint32_t i;
uint8_t tst1;
char * test_section;
char wtmp[25];
char *wstring, *wstring2;
char *xname;

printf("\nRetrieving process socket/port information\n");

// Get the list of loaded system modules
if (debug_data->x64) {

    wptr = memgrabke_64_LE(debug_data->PsLoadedModuleList);
    wend = memgrabke_64_LE(debug_data->PsLoadedModuleList + 0x8);

}
else {

    wptr = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList)
```

```
                  & 0x00000000ffffffff;
    wend = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList + 0x4)
                  & 0x00000000ffffffff;

}

// Scroll through the list until you find tcpip.sys
while ((wptr != 0x0) && (wptr != wend)) {


    // Retrieve PE header information for the module
    if (debug_data->x64) {

        wimage_base = memgrabke_64_LE(wptr+0x30);
        wdrivername.Length = memgrabke_16_LE(wptr + 0x58);
        wdrivername.MaximumLength = wdrivername.Length;
        wdrivername.Buffer = memgrabke_64_LE(wptr + 0x60);

    }
    else {

        wimage_base = ((uint64_t) memgrabke_32_LE(wptr+0x18))
                    & 0x00000000ffffffff;
        wdrivername.Length = memgrabke_16_LE(wptr + 0x2c);
        wdrivername.MaximumLength = wdrivername.Length;
        wdrivername.Buffer = ((uint64_t) memgrabke_32_LE(wptr + 0x30))
                    & 0x00000000ffffffff;

    }


    // If you've found tcpip.sys
    wstring = memgrabke_unicode_LE_space(wdrivername);
    if (!strcmp("tcpip.sys",wstring)) {


        // Retrieve the symbol information from the PDB file
        // First, load the pdb information
        if (tcpip_pdb == NULL) {

            pdb_url = get_pdb_url(wimage_base, PDB_RET_URL,
                                  debug_data->x64,debug_data->kDTB);

            if (pdb_url == NULL) {
```

```
    fname = malloc(sizeof(char) * (strlen("TCPIP.SYS") +
              strlen(DATADIRECTORY) + 1));
    strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
    strcat_s(fname,strlen(DATADIRECTORY) + strlen("TCPIP.SYS")
                 + 1,"TCPIP.SYS");
    pdb_url = get_pdb_url_disk(fname,PDB_RET_URL,debug_data->x64);
    wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
    strcpy_s(wpdb_url, strlen(pdb_url) + 1, pdb_url);
    wstring2 = get_pdb_url_disk(fname,PDB_RET_NAME,debug_data->x64);
    xname = malloc(sizeof(char) * (strlen(wstring2) +
             strlen(DATADIRECTORY) + 1));
    strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
    strcat_s(xname,strlen(DATADIRECTORY) + strlen(wstring2) +
         1,wstring2);
    if (GET_PDB) xname = download_pdb_file(wstring2, wpdb_url,
         DATADIRECTORY);
    tcpip_pdb = parse_pdb_file(xname, PDB_DEFAULT);
    free(wstring2);
    free(xname);
    free(fname);
    free(wpdb_url);

}
else {

    wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
    strcpy_s(wpdb_url, strlen(pdb_url) + 1, pdb_url);
    wstring2 = get_pdb_url(wimage_base,PDB_RET_NAME,
               debug_data->x64,debug_data->kDTB);
    xname = malloc(sizeof(char) * (strlen(wstring2) +
              strlen(DATADIRECTORY) + 1));
    strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
    strcat_s(xname,strlen(DATADIRECTORY) + strlen(wstring2) +
              1,wstring2);
    if (GET_PDB)
      xname = download_pdb_file(wstring2, wpdb_url, DATADIRECTORY);
    tcpip_pdb = parse_pdb_file(xname, PDB_DEFAULT);
    free(wstring2);
    free(xname);
    free(wpdb_url);

}

free(pdb_url);
```

```
}


// Second, find where the different sections start
pe_offset =  memgrabke_32_LE(wimage_base + 0x3c);
hdr_base = wimage_base + pe_offset + 0x18;
/*
for (i = 0; i < nbr_sections; i++) {

    test_section = memgrabke_stringn(hdr_base +
                (debug_data->x64?0xf0:0xe0) +
                i * 0x28 + 0,8);
    if (!strcmp(test_section,".data")) {

        data_segment = memgrabke_32_LE(hdr_base +
                (debug_data->x64?0xf0:0xe0) +
                 i * 0x28 + 0x0c);

    }

    if (!strcmp(test_section,".rdata")) {

        rdata_segment = memgrabke_32_LE(hdr_base +
                 (debug_data->x64?0xf0:0xe0) +
                 i * 0x28 + 0x0c);

    }


}

printf("data_segment = %x    rdata_segment = %x\n",
        data_segment,rdata_segment);
*/
data_segment = get_pdb_section(tcpip_pdb,".data",5)->virtual_address;
rdata_segment = get_pdb_section(tcpip_pdb,".rdata",6)->virtual_address;
nbr_sections = memgrabke_16_LE(hdr_base + 0x02);
if (1) {

    printf("HERE MajorOsVersion = %x  MinorOsVersion = %x\n",
           debug_data->MajorOSVersion,debug_data->MinorOSVersion);
    printf("data_segment = %x    rdata_segment = %x\n",
           data_segment,rdata_segment);

}
```

```
// If this is an XP or Win2003 memory dump
if (debug_data->MajorOSVersion == 5) {


    // First get the established TCP connections
    wwptr = get_pdb_symbol(tcpip_pdb,"_AddrObjTable",13);
    if (wwptr == 0) wwptr = get_pdb_symbol(tcpip_pdb,"AddrObjTable",12);
    if (wwptr != NULL) {

        addrtable_offset = wwptr->offset; // Unadjusted offset
        addrtable_offset_ptr = get_pdb_symbol_loc(tcpip_pdb,
                    addrtable_offset,data_segment); // Adjusted RVA

    }
    else {

        addrtable_offset = 0;
        addrtable_offset_ptr = 0;

    }

    wwptr = get_pdb_symbol(tcpip_pdb,"_AddrObjTableSize",17);
    if (wwptr == NULL) wwptr = get_pdb_symbol(tcpip_pdb,
                  "AddrObjTableSize",16);
    if (wwptr != NULL) {

        addrtable_size2       = wwptr->offset; // Unadjusted offset
        addrtable_size_ptr   = get_pdb_symbol_loc(tcpip_pdb,
                    addrtable_size2,data_segment); // Adjusted RVA

    }
    else {

        addrtable_size2       = 0;
        addrtable_size_ptr   = 0;

    }

    wwptr                 = get_pdb_symbol(tcpip_pdb,"_TCBTable",9);
    if (wwptr == NULL) wwptr = get_pdb_symbol(tcpip_pdb,
                              "TCBTable",8);
    if (wwptr != NULL) {
```

```
    tcbtable_offset      = wwptr->offset; // Unadjusted offset
    tcbtable_offset_ptr  = get_pdb_symbol_loc(tcpip_pdb,
                 tcbtable_offset,data_segment); // Adjusted RVA

}
else {

    tcbtable_offset         = 0;
    tcbtable_offset_ptr  = 0;

}

wwptr                 = get_pdb_symbol(tcpip_pdb,
                   "_MaxHashTableSize",17);
if (wwptr == NULL) wwptr = get_pdb_symbol(tcpip_pdb,
                         "MaxHashTableSize",16);
if (wwptr != NULL) {

    tcbtable_size2        = wwptr->offset; // Unadjusted offset
    tcbtable_size_ptr    = get_pdb_symbol_loc(tcpip_pdb,
                  tcbtable_size2,data_segment); // Adjusted RVA
    printf("tcb_table_size offset = %x   ptr = %x\n",
            tcbtable_size2,tcbtable_size_ptr);

}

else {

    tcbtable_size2         = 0;
    tcbtable_size_ptr    = 0;

}



 // Retrieve the symbol values for addrtable_size and
// addrtable_address
 wloc = wimage_base + (((uint64_t) addrtable_size_ptr)
      & 0x00000000ffffffff);
 addrtable_size = ((uint64_t) memgrabke_32_LE(wloc) )
               & 0x00000000ffffffff;
 wloc = wimage_base + (((uint64_t) addrtable_offset_ptr)
      & 0x00000000ffffffff);
 if (debug_data->x64)
   addrtable_address = memgrabke_64_LE(wloc);
```

```
else
  addrtable_address = ((uint64_t) memgrabke_32_LE(wloc))
                      & 0x00000000ffffffff;

// Print and store the established TCP connections
addrtable_size &= 0x00000000ffffffff;
printf("\nSockets    (size = %i)\n\n",addrtable_size);
if (addrtable_size > 0x1000) addrtable_size = 0;
for (i = 1; i < addrtable_size; i++) {

    if (debug_data->x64)
      addr_base = memgrabke_64_LE(addrtable_address + (i - 1) * 8);
    else
      addr_base = ((uint64_t) memgrabke_32_LE(addrtable_address
                + i * 4)) & 0x00000000ffffffff;
    while (addr_base != 0x0) {

        old_addr_base = addr_base;
        if (debug_data->x64) {

            localIP1   = memgrabke_8_LE(addr_base + 0x58);
            localIP2   = memgrabke_8_LE(addr_base + 0x59);
            localIP3   = memgrabke_8_LE(addr_base + 0x5a);
            localIP4   = memgrabke_8_LE(addr_base + 0x5b);
            LocalPort = memgrabke_16_BE(addr_base + 0x5c);
            Protocol  = memgrabke_16_LE(addr_base + 0x5e);
            pid       = memgrabke_32_LE(addr_base + 0x238);

        }
        else {

            if (debug_data->MinorOSVersion == 1) {
                // WinXP
                localIP1   = memgrabke_8_LE(addr_base + 0x2c);
                localIP2   = memgrabke_8_LE(addr_base + 0x2d);
                localIP3   = memgrabke_8_LE(addr_base + 0x2e);
                localIP4   = memgrabke_8_LE(addr_base + 0x2f);
                LocalPort = memgrabke_16_BE(addr_base + 0x34);
                Protocol  = memgrabke_16_LE(addr_base + 0x32);
                pid       = memgrabke_32_LE(addr_base + 0x148);

            }
            else {
                // Win03 (Should go back and do this for Windows
                // 2000 and for 64bit Win03 -- maybe later
```

```
        localIP1   = memgrabke_8_LE(addr_base + 0x30);
        localIP2   = memgrabke_8_LE(addr_base + 0x31);
        localIP3   = memgrabke_8_LE(addr_base + 0x32);
        localIP4   = memgrabke_8_LE(addr_base + 0x33);
        LocalPort = memgrabke_16_BE(addr_base + 0x34);
        Protocol  = memgrabke_16_LE(addr_base + 0x36);
        pid       = memgrabke_32_LE(addr_base + 0x14c);

    }


}


if ( (localIP1 != 0x0) || (localIP2 != 0x0) ||
     (localIP3 != 0x0) || (localIP4 != 0x0) ||
     (LocalPort != 0x0)) {

    // Print the connection information
    printf("AOTABLE   Addr: %llx (phys: %llx) \tPID %4i\t",
       addr_base,kevtop(addr_base),pid);

    printf("Local Addr = %3i.%3i.%3i.%3i:%6i\tProtocol: %i\n",
       localIP1,localIP2,localIP3,localIP4,LocalPort,Protocol);
    // Associate the information with the correct
    // process and store it in the process record
    tmp = proc_list;
    while ( (tmp != NULL) &&
            (tmp->proc_eprocess.UniqueProcessId != pid) )
           tmp = tmp->next;
    if (tmp != NULL) {

        if ((wsocket = malloc(sizeof *wsocket)) == NULL ) {

            (void)exit(EXIT_FAILURE);

        }

        wsocket->LocalPort = malloc(sizeof(char) * 25);
        sprintf_s(wtmp,25,"%i.%i.%i.%i:%i",
             localIP1,localIP2,localIP3,localIP4,LocalPort);
        strcpy_s(wsocket->LocalPort,25,wtmp);
        wsocket->RemotePort = malloc(sizeof(char) * 25);
        wsocket->Protocol = malloc(sizeof(char) * 13);
        wsocket->RemotePort = "*.*";
```

```
            switch (Protocol) {

                case 6:
                strcpy_s(wsocket->Protocol, 13, "TCP");
                wsocket->RemotePort = "0.0.0.0:0";
                break;
                case 17:
                strcpy_s(wsocket->Protocol, 13, "UDP");
                break;
                case 47:
                strcpy_s(wsocket->Protocol, 13, "GRE");
                break;
                default:
                strcpy_s(wsocket->Protocol, 13, "Other");
                break;

            }

            wsocket->Ipv6 = malloc(sizeof(char) * 5);
            strcpy_s(wsocket->Ipv6, 5, "IPv4");
            wsocket->next = tmp->sockets;
            wsocket->prev = NULL;
            tmp->sockets = wsocket;
            if (wsocket->next != NULL)
              wsocket->next->prev = wsocket;

        }


    }

    while ((addr_base <= old_addr_base) && (addr_base != 0)) {

        if (debug_data->x64)
          addr_base = memgrabke_64_LE(addr_base);
        else
          addr_base = ((uint64_t) memgrabke_32_LE(addr_base))
                    & 0x00000000ffffffff;

    }


}
```

```
}


// Retrieve the symbol values for tcbtable_size and tcbtabl_address
wloc = wimage_base + (((uint64_t) tcbtable_size_ptr)
     & 0x00000000ffffffff);
tcbtable_size = ((uint64_t) memgrabke_32_LE(wloc) )
              & 0x00000000ffffffff;
wloc = wimage_base + (((uint64_t) tcbtable_offset_ptr)
     & 0x00000000ffffffff);

if (debug_data->x64)
  tcbtable_address = memgrabke_64_LE(wimage_base +
                     tcbtable_offset_ptr);
else
  tcbtable_address = ((uint64_t) memgrabke_32_LE(wimage_base
                   + tcbtable_offset_ptr)) & 0x00000000ffffffff;

// Print and store the TCP listeners and UDP endpoints
tcbtable_size &= 0x00000000ffffffff;
printf("\nConnections (Size = %i\n\n",tcbtable_size);
if (tcbtable_size > 0x1000) tcbtable_size = 0;
for (i = 0; i < tcbtable_size; i++) {

    if (debug_data->x64)
      addr_base = memgrabke_64_LE(tcbtable_address + i * 8);
    else
      addr_base = ((uint64_t) memgrabke_32_LE(tcbtable_address
                + i * 4)) & 0x00000000ffffffff;
    while (addr_base != 0x0) {

        remotIP1   = memgrabke_8_LE(addr_base + 0x0c);
        remotIP2   = memgrabke_8_LE(addr_base + 0x0d);
        remotIP3   = memgrabke_8_LE(addr_base + 0x0e);
        remotIP4   = memgrabke_8_LE(addr_base + 0x0f);
        localIP1   = memgrabke_8_LE(addr_base + 0x10);
        localIP2   = memgrabke_8_LE(addr_base + 0x11);
        localIP3   = memgrabke_8_LE(addr_base + 0x12);
        localIP4   = memgrabke_8_LE(addr_base + 0x13);
        RemotePort = memgrabke_16_BE(addr_base + 0x14);
        LocalPort = memgrabke_16_BE(addr_base + 0x16);
        pid       = memgrabke_32_LE(addr_base + 0x18);
        if ( (localIP1 != 0x0) || (localIP2 != 0x0) ||
             (localIP3 != 0x0) ||
               (localIP4 != 0x0) || (LocalPort != 0x0)) {
```

```
                            // Print the information
                            printf("TCBTABLE    Addr: %llx (phys: %llx) \tPID %4i",
addr_base,kevtop(addr_base),pid);
                            printf("\tLocal Address = %3i.%3i.%3i.%3i:%6i\t",
                                localIP1,localIP2,localIP3,localIP4,LocalPort);
                            printf("Remote Address = %3i.%3i.%3i.%3i:%6i\n",
                                remotIP1,remotIP2,remotIP3,remotIP4,RemotePort);
                            // Find the appropriate process and save the
                            // inofrmation in an EPROCESS record
                            tmp = proc_list;
                            while ( (tmp != NULL) &&
                                    (tmp->proc_eprocess.UniqueProcessId != pid) )
                                 tmp = tmp->next;
                            if (tmp != NULL) {

                                if ((wsocket = malloc(sizeof *wsocket)) == NULL ) {

                                    (void)exit(EXIT_FAILURE);

                                }

                                wsocket->LocalPort = malloc(sizeof(char) * 25);
                                 sprintf_s(wtmp,25,"%i.%i.%i.%i:%i",
                                      localIP1,localIP2,localIP3,localIP4,LocalPort);
                                strcpy_s(wsocket->LocalPort,25,wtmp);
                                wsocket->RemotePort = malloc(sizeof(char) * 25);
                                sprintf_s(wtmp,25,"%i.%i.%i.%i:%i",
                                       remotIP1,remotIP2,remotIP3,remotIP4,RemotePort);
                                strcpy_s(wsocket->RemotePort,25,wtmp);
                                wsocket->Protocol = malloc(sizeof(char) * 13);
                                strcpy_s(wsocket->Protocol, 13, "TCP");
                                wsocket->Ipv6 = malloc(sizeof(char) * 5);
                                strcpy_s(wsocket->Ipv6, 5, "IPv4");
                                wsocket->next = tmp->sockets;
                                wsocket->prev = NULL;
                                tmp->sockets = wsocket;
                                if (wsocket->next != NULL)
                                  wsocket->next->prev = wsocket;

                            }


                        }
```

```
            if (debug_data->x64)
              addr_base = memgrabke_64_LE(addr_base);
            else
              addr_base = ((uint64_t) memgrabke_32_LE(addr_base))
                          & 0x00000000ffffffff;

        }


    }


}


// If this is a Win7 or Vista dump
else if (debug_data->MajorOSVersion == 6) {


    // Retrieve the symbols
    wwptr = get_pdb_symbol(tcpip_pdb,"PartitionCount",14);
    if (wwptr == NULL)
      wwptr = get_pdb_symbol(tcpip_pdb,"_PartitionCount",15);
    if (TCPDEBUG) printf("wwptr = %x\n",wwptr);
    if (wwptr != NULL) {

        PartitionCount_offset = wwptr->offset;
        PartitionCount_offset_ptr = get_pdb_symbol_loc(tcpip_pdb,
                       PartitionCount_offset, data_segment);
        wloc = wimage_base + (((uint64_t) PartitionCount_offset_ptr)
             & 0x00000000ffffffff);

    }
    else {

        PartitionCount_offset = 0;
        PartitionCount_offset_ptr = 0;

    }

    PartitionCount = wimage_base + (((uint64_t) PartitionCount_offset_ptr)
                   & 0x00000000ffffffff);
    wwptr = get_pdb_symbol(tcpip_pdb,"PartitionTable",14);
    if (wwptr == NULL)
      wwptr = get_pdb_symbol(tcpip_pdb,"_PartitionTable",15);
```

```
if (wwptr != NULL) {

    PartitionTable_offset = wwptr->offset;
    PartitionTable_offset_ptr = get_pdb_symbol_loc(tcpip_pdb,
                     PartitionTable_offset, data_segment);
    wloc = wimage_base + (((uint64_t) PartitionTable_offset_ptr)
         & 0x00000000ffffffff);

}
else {

    PartitionTable_offset = 0;
    PartitionTable_offset_ptr = 0;

}

PartitionTable = wimage_base + (((uint64_t) PartitionTable_offset_ptr)
               & 0x00000000ffffffff);
if (debug_data->x64)
     tmp1 = memgrabke_64_LE(PartitionTable);
else
     tmp1 = ((uint64_t) memgrabke_32_LE(PartitionTable)) &
          0x00000000ffffffff;

// For each partition
for (p = 0; p < 1; p++) {

    //for (p = 0; p < PartitionCount; p++) {

        // Retrieve the hash table
        if (debug_data->x64) {

            PTable = memgrabke_64_LE(tmp1 +
                   (debug_data->MinorOSVersion==0?0x40:0x78)
                       * p + 0x08 );
            HashTable = memgrabke_64_LE(PTable +
                    (debug_data->MinorOSVersion==0?0x38:0x20));
            TableEntries = memgrabke_64_LE(PTable + 0x08);

        }

        else {

            PTable = memgrabke_32_LE(tmp1 +
                (debug_data->MinorOSVersion==0?0x48
```

```
       :0x78) * p + 0x04 );
    HashTable = memgrabke_32_LE(PTable +
         (debug_data->MinorOSVersion==0?0x30:0x20));
    TableEntries = memgrabke_32_LE(PTable + 0x08) ;

}


// For each entry in the hash table
for (j = 0; j < TableEntries; j++) {

    if (debug_data->x64)
      Entry = HashTable + j * 0x10;
    else
      Entry = HashTable + j * 0x08;
    if (debug_data->x64)
      IpInfo = memgrabke_64_LE(Entry);
    else
      IpInfo = ((uint64_t) memgrabke_32_LE(Entry))
             & 0x00000000ffffffff;
    // For each non-zero entry in the hash table
    while ((IpInfo != Entry) && (IpInfo != 0)) {

        isValid = 1;
        if (isValid) {

            // Retrieve the established TCP connection info
            if (debug_data->x64) {

                tmp1 = memgrabke_64_LE(IpInfo - 0x08);
                tmp2 = memgrabke_64_LE(tmp1);
                tmp3 = memgrabke_64_LE(tmp2 + 0x10);
                tmp4 = memgrabke_64_LE(tmp3);
                LocalAddress = memgrabke_32_BE(tmp4);

            }
            else {

                tmp1 = ((uint64_t)
                       memgrabke_32_LE(IpInfo - 0x04))
                      & 0x00000000ffffffff;
                tmp2 = ((uint64_t)
                       memgrabke_32_LE(tmp1))
                      & 0x00000000ffffffff;
                tmp3 = ((uint64_t)
```

```
            memgrabke_32_LE(tmp2 + 0x0c))
           & 0x00000000ffffffff;
    tmp4 = ((uint64_t)
            memgrabke_32_LE(tmp3))
           & 0x00000000ffffffff;
    LocalAddress = memgrabke_32_BE(tmp4);

}

if (debug_data->x64)
  LocalPort = memgrabke_16_BE(IpInfo +
    (debug_data->MinorOSVersion==0?0x2c:0x44) );
else
  LocalPort = memgrabke_16_BE(IpInfo +
       (debug_data->MinorOSVersion==0?0x18:0x24) );
localIP1   = memgrabke_8_LE(tmp4 + 0x00);
localIP2   = memgrabke_8_LE(tmp4 + 0x01);
localIP3   = memgrabke_8_LE(tmp4 + 0x02);
localIP4   = memgrabke_8_LE(tmp4 + 0x03);
if (debug_data->x64)
  tmp2 = memgrabke_64_LE(tmp1 + 0x10);
else
  tmp2 = memgrabke_32_LE(tmp1 + 0x08);
RemoteAddress = memgrabke_32_BE(tmp2);
if (debug_data->x64)
  RemotePort = memgrabke_16_BE(IpInfo +
    (debug_data->MinorOSVersion==0?0x2e:0x46) );
else
  RemotePort = memgrabke_16_BE(IpInfo +
    (debug_data->MinorOSVersion==0?0x1a:0x26) );
remotIP1   = memgrabke_8_LE(tmp2 + 0x00);
remotIP2   = memgrabke_8_LE(tmp2 + 0x01);
remotIP3   = memgrabke_8_LE(tmp2 + 0x02);
remotIP4   = memgrabke_8_LE(tmp2 + 0x03);
if (debug_data->x64) {

    tmp1 = memgrabke_64_LE(IpInfo - 0x10);
    tmp2 = memgrabke_32_LE(tmp1 + 0x14);

}
else {

    tmp1 = memgrabke_32_LE(IpInfo - 0x08);
    tmp2 = memgrabke_32_LE(tmp1 + 0x0c);
```

```
}

Ipv6 = ((tmp2 == 0x17)?1:0);
if (debug_data->x64)
  pid = (uint32_t) keget_pdb_type(
       "_EPROCESS.UniqueProcessId",
       memgrabke_64_LE(IpInfo +
      (debug_data->MinorOSVersion==0?0x1e0
      :0x210)));
else
  pid = (uint32_t) keget_pdb_type(
       "_EPROCESS.UniqueProcessId",
      ((uint64_t) memgrabke_32_LE(IpInfo +
          (debug_data->MinorOSVersion==
           0?0x14c:0x160)))
          & 0x00000000ffffffff );

// Print and store the information
printf("Process ID: %i   Local IP = %i.%i.%i.%i:%i",
  pid,localIP1,localIP2,localIP3,
  localIP4,LocalPort);
printf("  Remote IP = %i.%i.%i.%i:%i   Ipv6 = %i\n",
  remotIP1,remotIP2,remotIP3,remotIP4,
  RemotePort,Ipv6);
if (pid != 0) {

    tmp = proc_list;
    while ( (tmp != NULL) &&
          (tmp->proc_eprocess.UniqueProcessId !=
    pid) )
        tmp = tmp->next;
    if (tmp != NULL) {

        if ((wsocket = malloc(sizeof *wsocket))
            == NULL ) {

            (void)exit(EXIT_FAILURE);

        }

        wsocket->LocalPort =
           malloc(sizeof(char) * 25);
        sprintf_s(wtmp,25,
           "%i.%i.%i.%i:%i",
            localIP1,localIP2,localIP3,
```

```
                        localIP4,LocalPort);
                    strcpy_s(wsocket->LocalPort,25,wtmp);
                    wsocket->RemotePort =
                        malloc(sizeof(char) * 25);
                    sprintf_s(wtmp,25,
                        "%i.%i.%i.%i:%i",
                          remotIP1,remotIP2,remotIP3,
                          remotIP4,RemotePort);
strcpy_s(wsocket->RemotePort,25,wtmp);
                    wsocket->Protocol =
                         malloc(sizeof(char) * 13);
                    strcpy_s(wsocket->Protocol, 13, "TCP");
                    wsocket->Ipv6 = malloc(sizeof(char) * 5);
                    strcpy_s(wsocket->Ipv6, 5,
                         (Ipv6?"IPv6":"IPv4"));
                    wsocket->next = tmp->sockets;
                    wsocket->prev = NULL;
                    tmp->sockets = wsocket;
                    if (wsocket->next != NULL)
                      wsocket->next->prev = wsocket;

                }


            }

            // Iterate to the next IPInfo record
            if (debug_data->x64)
              IpInfo = memgrabke_64_LE(IpInfo);
            else
              IpInfo = memgrabke_32_LE(IpInfo);

        }


    }


}


// Retrieve the symbols
```

```
wwptr = get_pdb_symbol(tcpip_pdb,"TcpPortPool",11);
if (wwptr == 0)
  wwptr = get_pdb_symbol(tcpip_pdb,"_TcpPortPool",12);
if (TCPDEBUG) printf("wwptr = %x\n",wwptr);
if (wwptr != NULL) {

    TcpPortPool_offset = wwptr->offset;
    TcpPortPool_offset_ptr = get_pdb_symbol_loc(tcpip_pdb,
            TcpPortPool_offset,data_segment);

}
else {

    TcpPortPool_offset = 0;
    TcpPortPool_offset_ptr = 0;

}

TcpPortPool = wimage_base + (((uint64_t) TcpPortPool_offset_ptr)
             & 0x00000000ffffffff);
if (debug_data->x64)
    TcpPortPool = memgrabke_64_LE(TcpPortPool);
else
    TcpPortPool = ((uint64_t) memgrabke_32_LE(TcpPortPool))
                & 0x00000000ffffffff;
if (TCPDEBUG) printf("TcpPortPool %llx\n",TcpPortPool);

// Retrieve the bit map
if (debug_data->x64) {

    BitMapSize = memgrabke_64_LE(TcpPortPool + 0x90);
    BitMapPtr = memgrabke_64_LE(TcpPortPool + 0x98);

}
else {

    BitMapSize = ( (uint64_t) memgrabke_32_LE(TcpPortPool + 0x50))
              & 0x00000000ffffffff;
    BitMapPtr = ( (uint64_t) memgrabke_32_LE(TcpPortPool + 0x54))
             & 0x00000000ffffffff;

}


// Iterate through the bitmap -- the bitmap is made up of a
```

```
// series of bytes.
// The first byte is the first eight bits.
// The lowest bit in the byte is the first bit.
SIdx = 1; // Used to ship the first bit in the first byte
for (k = 0; k < BitMapSize / 8; k++) {

    tst1 = memgrabke_8_LE(BitMapPtr + k);
    for (j = SIdx; j < 8; j++) {

        // Each time you find a bit that is set.
        if ((tst1 >> j) & 0x01) {

            // The bitmap is large enough that it will take
            // up to two bytes to
            //  describe the bit
            // The first (high) byte is the page number.
            // The second (low)
            // byte is the offset in the page
            idx = k * 8 + j;
            pagenbr = (idx >>8) & 0xff;
            offset = idx & 0xff;
            if (debug_data->x64)
              tmp1 = memgrabke_64_LE(TcpPortPool + 0xA0 +
                     pagenbr * 8);
            else
              tmp1 = ((uint64_t) memgrabke_32_LE(TcpPortPool
                   + 0x58 + pagenbr * 4))
                   & 0x00000000ffffffff;
            if (TCPDEBUG) printf("tmp1 = %llx\n",tmp1);
            if (debug_data->x64)
              tmp2 = memgrabke_64_LE(tmp1 + 0x20);
            else
              tmp2 = ((uint64_t) memgrabke_32_LE(tmp1 + 0x14))
                   & 0x00000000ffffffff;
            if (TCPDEBUG) printf("tmp2 = %llx\n",tmp2);
            if (debug_data->x64)
              IpInfo = memgrabke_64_LE(tmp2 + (offset <<4)
                     + 0x08) & 0xfffffffffffffffcULL;
            else
              IpInfo = ((uint64_t) memgrabke_32_LE(tmp2
                     + (offset <<3) + 0x04))
                     & 0x00000000fffffffc;
            // Once you've got the first IPInfo structure
            // (there is a linked list
            // at the beginning of it that let's you
```

```
// traverse mulitple IPInfo records that are
// connected (somehow!?),
// retrieve the network connection information
while (IpInfo != 0) {

    if (debug_data->x64)
      tmp4 = memgrabke_64_LE(IpInfo - 0x18);
    else
      tmp4 = memgrabke_32_LE(IpInfo - 0x0c);
    if (debug_data->x64) {

        LocalPort = memgrabke_16_BE(IpInfo - 0x06);
        RemotePort = memgrabke_16_BE(
                  memgrabke_64_LE(IpInfo - 0x10)
                   + 0x14);

    }

    else {

        LocalPort = memgrabke_16_BE(IpInfo - 0x02);
        tmp1 = ((uint64_t)
              memgrabke_32_LE(IpInfo - 0x08))
             & 0x00000000ffffffff;
        RemotePort = memgrabke_16_BE(
             (((uint64_t)
             memgrabke_32_LE(IpInfo - 0x08))
               & 0x00000000ffffffff)
               + 0x0c);

    }

    if (tmp4 == 0) {

        LocalAddress = localIP1 = localIP2 = 0;
        localIP3 = localIP4 = 0;

    }
    else {

        if (debug_data->x64) {

            tmp1 = memgrabke_64_LE(tmp4);
            tmp2 = memgrabke_64_LE(tmp1);
            LocalAddress = memgrabke_32_BE(tmp2);
```

```
    }
    else {

        tmp1 = ((uint64_t)
            memgrabke_32_LE(tmp4 + 0x0c))
            & 0x00000000ffffffff;
        tmp2 = ((uint64_t)
             memgrabke_32_LE(tmp1 + 0x04))
            & 0x00000000ffffffff;
        LocalAddress = memgrabke_32_BE(tmp2);

    }

    localIP1   = memgrabke_8_LE(tmp2 + 0x00);
    localIP2   = memgrabke_8_LE(tmp2 + 0x01);
    localIP3   = memgrabke_8_LE(tmp2 + 0x02);
    localIP4   = memgrabke_8_LE(tmp2 + 0x03);

}

tmp1 = memgrabke_64_LE(IpInfo - 0x10);
tmp2 = memgrabke_32_LE(tmp1 + 0x14);
Ipv6 = ((tmp2 == 0x17)?1:0);
pid = 0;
if (debug_data->x64)
  pid = (uint32_t) keget_pdb_type(
       "_EPROCESS.UniqueProcessId",
       memgrabke_64_LE(IpInfo - 0x48));
else
  pid = (uint32_t) keget_pdb_type(
      "_EPROCESS.UniqueProcessId",
      ((uint64_t) memgrabke_32_LE(IpInfo - 0x28))
       & 0x00000000ffffffff );
// Print and store the information
printf("Process ID: %i       ",pid);
printf("Local IP = %i.%i.%i.%i",
  localIP1,localIP2,localIP3,localIP4);
printf(":%i    Remote Port = %i   IPv6 = %i\n",
  LocalPort,RemotePort,Ipv6);
if (pid != 0) {

    tmp = proc_list;
    while ( (tmp != NULL) &&
       (tmp->proc_eprocess.UniqueProcessId
```

```
                        != pid) )
                       tmp = tmp->next;
                    if (tmp != NULL) {

                        if ((wsocket = malloc(sizeof *wsocket))
                             == NULL ) {

                            (void)exit(EXIT_FAILURE);

                        }

                        wsocket->LocalPort =
                            malloc(sizeof(char) * 25);
                        sprintf_s(wtmp,
                            25,"%i.%i.%i.%i:%i",
                            localIP1,localIP2,localIP3,
                            localIP4,LocalPort);
                        strcpy_s(wsocket->LocalPort,25,wtmp);
                        wsocket->RemotePort =
                            malloc(sizeof(char) * 25);
                        sprintf_s(wtmp,
                            25,"*:%i",RemotePort);
strcpy_s(wsocket->RemotePort,25,wtmp);
                        wsocket->Protocol =
                            malloc(sizeof(char) * 13);
                        strcpy_s(wsocket->Protocol, 13, "TCP");
                        wsocket->Ipv6 = malloc(sizeof(char) * 5);
                        strcpy_s(wsocket->Ipv6, 5,
                            (Ipv6?"IPv6":"IPv4"));
                        wsocket->next = tmp->sockets;
                        wsocket->prev = NULL;
                        tmp->sockets = wsocket;
                        if (wsocket->next != NULL)
                          wsocket->next->prev = wsocket;

                    }


                }

                if (debug_data->x64)
                  IpInfo = memgrabke_64_LE(IpInfo);
                else
                  IpInfo = ((uint64_t)
                          memgrabke_32_LE(IpInfo))
```

```
                              & 0x00000000ffffffff;

                        }


                    }


                }

                SIdx = 0;

            }


            if (!NOUDP) {

                // Retrieve the symbols
                wwptr = get_pdb_symbol(tcpip_pdb,"UdpPortPool",11);
                if (wwptr == 0)
                  wwptr = get_pdb_symbol(tcpip_pdb,"_UdpPortPool",12);
                if (TCPDEBUG) printf("wwptr = %x\n",wwptr);
                if (wwptr != NULL) {

                    UdpPortPool_offset = wwptr->offset;
                    UdpPortPool_offset_ptr = get_pdb_symbol_loc(
                        tcpip_pdb,UdpPortPool_offset,data_segment);

                }
                else {

                    UdpPortPool_offset = 0;
                    UdpPortPool_offset_ptr = 0;

                }

                UdpPortPool = wimage_base + (((uint64_t)
                         UdpPortPool_offset_ptr) & 0x00000000ffffffff);
                if (debug_data->x64)
                  UdpPortPool = memgrabke_64_LE(UdpPortPool);
                else
                  UdpPortPool = ((uint64_t) memgrabke_32_LE(UdpPortPool))
                              & 0x00000000ffffffff;

                // Retrieve the bit map
```

```
if (debug_data->x64) {

    BitMapSize = memgrabke_64_LE(UdpPortPool + 0x90);
    BitMapPtr = memgrabke_64_LE(UdpPortPool + 0x98);

}
else {

    BitMapSize = ( (uint64_t) memgrabke_32_LE(
               UdpPortPool + 0x50))
               & 0x00000000ffffffff;
    BitMapPtr = ( (uint64_t) memgrabke_32_LE(UdpPortPool
              + 0x54)) & 0x00000000ffffffff;

}


// Iterate through the bitmap -- the bitmap is made up
// of a series of bytes.
// The first byte is the first eight bits.
//      The lowest bit in the byte is the first bit.
SIdx = 1; // Used to ship the first bit in the first byte
for (k = 0; k < BitMapSize / 8; k++) {

    tst1 = memgrabke_8_LE(BitMapPtr + k);
    for (j = SIdx; j < 8; j++) {

        // Each time you find a bit that is set.
        if ((tst1 >> j) & 0x01) {

            // The bitmap is large enough that it will
            // take up to two bytes to
            // describe the bit
            // The first (high) byte is the page number.
            // The second (low) byte
            // is the offset in the page
            idx = k * 8 + j;
            pagenbr = (idx >>8) & 0xff;
            offset = idx & 0xff;
            if (debug_data->x64)
              tmp1 = memgrabke_64_LE(UdpPortPool + 0xA0
                   + pagenbr * 8);
            else
              tmp1 = ((uint64_t)
                  memgrabke_32_LE(UdpPortPool + 0x58 +
```

```
      pagenbr * 4)) & 0x00000000ffffffff;
if (TCPDEBUG) printf("tmp1 = %llx\n",tmp1);
if (debug_data->x64)
  tmp2 = memgrabke_64_LE(tmp1 + 0x20);
else
  tmp2 = ((uint64_t) memgrabke_32_LE(tmp1 + 0x14))
     & 0x00000000ffffffff;
if (TCPDEBUG) printf("tmp2 = %llx\n",tmp2);
if (debug_data->x64)
  IpInfo = memgrabke_64_LE(tmp2 +
       (offset <<4) + 0x08)
       & 0xfffffffffffffffcULL;
else
  IpInfo = ((uint64_t) memgrabke_32_LE(tmp2
       + (offset <<3) + 0x04))
       & 0x00000000fffffffc;
// Once you've got the first IPInfo
// structure (there is a
// linked list at the beginning of it that
// let's you
// traverse mulitple IPInfo records that are
// connected (somehow!?),
// retrieve the network connection
// information
while (IpInfo != 0) {

    if (debug_data->x64)
      tmp4 = memgrabke_64_LE(IpInfo - 0x28);
    else
      tmp4 = memgrabke_32_LE(IpInfo - 0x14);
    if (debug_data->x64) {

        LocalPort = memgrabke_16_BE(IpInfo - 0x08);
        RemotePort = memgrabke_16_BE(
                 memgrabke_64_LE(IpInfo
                 - 0x10) + 0x14);

    }

    else {

        LocalPort = memgrabke_16_BE(IpInfo - 0x08);
        RemotePort = memgrabke_16_BE(
          (((uint64_t) memgrabke_32_LE(
          IpInfo - 0x14)) &
```

```
         0x00000000ffffffff)
         + 0x0c);

}

if (tmp4 == 0) {

  LocalAddress = localIP1 = localIP2 = 0;
  localIP3 = localIP4 = 0;

}
else {

    if (debug_data->x64) {

        tmp1 = memgrabke_64_LE(tmp4 + 0x10);
        tmp2 = memgrabke_64_LE(tmp1);
        LocalAddress = memgrabke_32_BE(tmp2);

    }
    else {

        tmp1 = ((uint64_t) memgrabke_32_LE(
              tmp4 + 0x0c))
             & 0x00000000ffffffff;
        tmp2 = ((uint64_t) memgrabke_32_LE(
              tmp1 + 0x04))
             & 0x00000000ffffffff;
        LocalAddress = memgrabke_32_BE(tmp2);

    }

    localIP1   = memgrabke_8_LE(tmp2 + 0x00);
    localIP2   = memgrabke_8_LE(tmp2 + 0x01);
    localIP3   = memgrabke_8_LE(tmp2 + 0x02);
    localIP4   = memgrabke_8_LE(tmp2 + 0x03);

}

tmp1 = memgrabke_64_LE(IpInfo - 0x68);
tmp2 = memgrabke_32_LE(tmp1 + 0x14);
Ipv6 = ((tmp2 == 0x17)?1:0);
pid = 0;
if (debug_data->x64)
  pid = (uint32_t) keget_pdb_type(
```

```
        "_EPROCESS.UniqueProcessId",
        memgrabke_64_LE(IpInfo - 0x60));
else
  pid = (uint32_t) keget_pdb_type(
         "_EPROCESS.UniqueProcessId",
        ((uint64_t) memgrabke_32_LE(IpInfo
    - 0x34)) & 0x00000000ffffffff );
// Print and store the information
printf("Process ID: %i       ",pid);
printf("Local IP = %i.%i.%i.%i",
  localIP1,localIP2,localIP3,localIP4);
printf(":%i     IPv6 = %i\n",
  LocalPort,Ipv6);
if (pid != 0) {

    tmp = proc_list;
    while ( (tmp != NULL) &&
        (tmp->proc_eprocess.UniqueProcessId
        != pid) ) tmp = tmp->next;
    if (tmp != NULL) {

        if ((wsocket = malloc(sizeof *wsocket))
            == NULL ) {

            (void)exit(EXIT_FAILURE);

        }

        wsocket->LocalPort =
           malloc(sizeof(char) * 25);
        sprintf_s(wtmp,25,
           "%i.%i.%i.%i:%i",
           localIP1,localIP2,localIP3,
           localIP4,LocalPort);
        strcpy_s(wsocket->LocalPort,25,wtmp);
        wsocket->RemotePort =
           malloc(sizeof(char) * 25);
        sprintf_s(wtmp,
           25,"%s","*");
strcpy_s(wsocket->RemotePort,25,wtmp);
        wsocket->Protocol =
           malloc(sizeof(char) * 13);
        strcpy_s(wsocket->Protocol, 13, "UDP");
        wsocket->Ipv6 =
           malloc(sizeof(char) * 5);
```

```
                                    strcpy_s(wsocket->Ipv6, 5,
                                       (Ipv6?"IPv6":"IPv4"));
                                    wsocket->next = tmp->sockets;
                                    wsocket->prev = NULL;
                                    tmp->sockets = wsocket;
                                    if (wsocket->next != NULL)
                                        wsocket->next->prev = wsocket;

                                }


                            }

                            if (debug_data->x64)
                              IpInfo = memgrabke_64_LE(IpInfo);
                            else
                              IpInfo = ((uint64_t)
                                   memgrabke_32_LE(IpInfo))
                                   & 0x00000000ffffffff;

                        }


                    }


                }

                SIdx = 0;

            }


        }


    }


}

free(wstring);
// Iterate to the next driver
if (debug_data->x64)
wptr = memgrabke_64_LE(wptr + 0x00);
```

```
        else
        wptr = memgrabke_32_LE(wptr + 0x00);

    }


    return 0;

}


/********************************************************************************/

/****************************/
/* LINK USERS WITH PROCESSES */
/****************************/

/****************************/
/* Link users with processes */
/****************************/
void load_user_process()
{

    process_list *tmp;              // iterator for processes
    user_list_type *wuser, *wwuser; // iterators through the list of users
    uint64_t wpointer;              // pointer to SID and ATTRIBUTES for a token
    uint64_t token_address;         // virtual address of token associated
                                    //   with a process
    unsigned char * wtoken_sid;     // string representation of token
    uint64_t ptr_array;             // list of users/groups associated with
                                    //   a particular token
    unsigned int ptr_count;         // count of users/groups associated with
                                    //   a particular token

    uint32_t ctr, addit,i, cctr;
    ascii_string wname;


    // For each process
    tmp = proc_list;
    while (tmp != NULL)        {

        // Get the virtual address of the token for that process
        token_address = (debug_data->x64?tmp->proc_eprocess.Token &
                        0xFFFFFFFFFFFFFFF0ULL:
```

```
                tmp->proc_eprocess.Token & 0xFFFFFFFFFFFFFFF8ULL);

        // Get the Users/Groups the token is associated with
        ptr_array = pget_pdb_type  ( "_TOKEN.UserAndGroups", token_address,
                     tmp->proc_eprocess.Pcb.DirectoryTableBase);
        ptr_count = (uint32_t) (pget_pdb_type  ( "_TOKEN.UserAndGroupCount",
                     token_address, tmp->proc_eprocess.Pcb.DirectoryTableBase)
                      & 0x00000000ffffffff);

        if (ptr_count < 100) {
            // If it isn't, we've got a problem
            for (ctr = 0; ctr < ptr_count; ctr ++) {

                // Here's where it gets fun -- we need to parse the SID and
                // Attributes
                wpointer = pget_pdb_type  ( "_SID_AND_ATTRIBUTES.Sid",
                             ptr_array + ctr *
                             get_pdb_type_sizeof("_SID_AND_ATTRIBUTES"),
                             tmp->proc_eprocess.Pcb.DirectoryTableBase);
                wname.Length = 0x1c;
                wname.MaximumLength = wname.Length;
                wname.Buffer = wpointer;
                // Get the token string
                wtoken_sid = memgrabp_string_LE (wname,
                               tmp->proc_eprocess.Pcb.DirectoryTableBase);
                if (MYDEBUG) {

                    printf("\t\tToken: ");
                    for (cctr = 0; cctr < 0x1c; cctr++)
                       printf("%x ",((unsigned char) wtoken_sid[cctr]));
                    printf("\n");

                }


                // Check if the token id matches any of the user tokens
                wuser = user_list;
                while (wuser != NULL) {

                    if (MYDEBUG) {

                        printf("\t\t    User: ");
                        for (cctr = 0; cctr < 0x1c; cctr++)
                              printf("%x ",((unsigned char) wuser->Sid[cctr]));
                        printf("\n");
```

```
    }

    if ((wuser->Sid[0] == wtoken_sid[0]) &&
        (wuser->Sid[1] == wtoken_sid[1])) {

        if (!bytecmp(wuser->Sid,wtoken_sid,2 + 6 + 4 *
            wuser->Sid[1]))
        break;
        else
        wuser = wuser->next;

    }

    else
    wuser = wuser->next;

}

free(wtoken_sid);

// If you've found a match, add it to the list of users
// associated with this process
if (wuser != NULL) {

    addit = (tmp->users == NULL);
    if (!addit)    {

        wwuser = tmp->users;
        while ( (wwuser != NULL) &&
                (strcmp(wwuser->Name,wuser->Name)) )
             wwuser = wwuser->next;
        addit = (wwuser == NULL);

    }

    if (addit) {

        if ((wwuser = malloc(sizeof(user_list_type))) ==
             NULL ) {

            (void)exit(EXIT_FAILURE);

        }
```

```
                        wwuser->Token = malloc(sizeof(char) *
                                        (strlen(wuser->Token) + 1));
                        strcpy_s(wwuser->Token, strlen(wuser->Token) + 1,
                                         wuser->Token);
                        wwuser->SidSize = wuser->SidSize;
                        wwuser->Sid = malloc(sizeof(uint8_t) *
                                       wuser->SidSize);
                        for (i = 0; i < wwuser->SidSize; i++)
                          wwuser->Sid[i] = wuser->Sid[i];
                        wwuser->Path = malloc(sizeof(char) *
                                       (strlen(wuser->Path) + 1));
                        strcpy_s(wwuser->Path, strlen(wuser->Path) + 1,
                                    wuser->Path);
                        wwuser->Name = malloc(sizeof(char) *
                                   (strlen(wuser->Name) + 1));
                        strcpy_s(wwuser->Name, strlen(wuser->Name) + 1,
                                   wuser->Name);
                        wwuser->next = tmp->users;
                        wwuser->prev = NULL;
                        tmp->users = wwuser;
                        if (wwuser->next != NULL)
                           wwuser->next->prev = wwuser;

                    }


                }


            }


        }

        tmp = tmp->next;

    }


}


/******************************************************************************/
```

```
/************************************/
/* DUMP FEATURES FOR FURTHER ANALYSIS */
/************************************/

int dump_memory() {
    // For Joe Erskine
    int percent_count = 0;
    uint32_t data_from_memdump = 0;
    uint64_t big_parent = 1;
    uint64_t big_vsize = 1;
    uint32_t big_time_low = 1;
    uint64_t big_time_high = 1;
    uint32_t big_read_low = 1;
    uint64_t big_read_high = 1;
    uint32_t big_write_low = 1;
    uint64_t big_write_high = 1;
    uint32_t big_other_low = 1;
    uint64_t big_other_high = 1;
    uint32_t big_priority = 1;
    uint64_t big_kernel = 1;
    uint64_t big_user = 1;
    unsigned char big_base = 1;

    process_list *tmp;
    int user_ctr;
    user_list_type *wuser;

    tmp = proc_list;
    while (tmp != NULL)        {

        if (tmp->eprocess_in_link_list == 1) {

            if (tmp->proc_eprocess.InheritedFromUniqueProcessId > big_parent)
            big_parent = tmp->proc_eprocess.InheritedFromUniqueProcessId;
            if (tmp->proc_eprocess.PeakVirtualSize > big_vsize)
            big_vsize = tmp->proc_eprocess.PeakVirtualSize;
            if (tmp->proc_eprocess.CreateTime.LowPart > big_time_low)
            big_time_low = tmp->proc_eprocess.CreateTime.LowPart;
            if (tmp->proc_eprocess.CreateTime.HighPart > big_time_high)
            big_time_high = tmp->proc_eprocess.CreateTime.HighPart;
            if (tmp->proc_eprocess.ReadOperationCount.LowPart > big_read_low)
            big_read_low = tmp->proc_eprocess.ReadOperationCount.LowPart;
            if (tmp->proc_eprocess.ReadOperationCount.HighPart > big_read_high)
            big_read_high = tmp->proc_eprocess.ReadOperationCount.HighPart;
            if (tmp->proc_eprocess.WriteOperationCount.LowPart > big_write_low)
```

```
            big_write_low = tmp->proc_eprocess.WriteOperationCount.LowPart;
            if (tmp->proc_eprocess.WriteOperationCount.HighPart > big_write_high)
            big_write_high = tmp->proc_eprocess.WriteOperationCount.HighPart;
            if (tmp->proc_eprocess.WriteOperationCount.LowPart > big_other_low)
            big_other_low = tmp->proc_eprocess.OtherOperationCount.LowPart;
            if (tmp->proc_eprocess.WriteOperationCount.HighPart > big_other_high)
            big_other_high = tmp->proc_eprocess.OtherOperationCount.HighPart;
            if (tmp->proc_eprocess.PriorityClass > big_priority)
            big_priority = tmp->proc_eprocess.PriorityClass;
            if (tmp->proc_eprocess.Pcb.KernelTime > big_kernel)
            big_kernel = tmp->proc_eprocess.Pcb.KernelTime;
            if (tmp->proc_eprocess.Pcb.UserTime > big_user)
            big_user = tmp->proc_eprocess.Pcb.UserTime;
            if (tmp->proc_eprocess.Pcb.BasePriority > big_base)
            big_base = tmp->proc_eprocess.Pcb.BasePriority;

        }

        tmp = tmp->next;

    }

    tmp = proc_list;
    while (tmp != NULL)        {

        if (tmp->eprocess_in_link_list == 1) {


            user_ctr = 0;
            if (tmp->users != NULL) {

                wuser = user_list;
                while (wuser != NULL) {

                    wuser = wuser->next;
                    user_ctr++;
                    if (!strcmp(wuser->Name,tmp->users->Name))
                    break;

                }


            }

            if (tmp->users == NULL) user_ctr = 0;
```

```
            fprintf (feature_file,"%f %f %f %f %f %f %f %f %f %f %f %f %f %f %i\n",
            ((double) tmp->proc_eprocess.InheritedFromUniqueProcessId /
                    (double) big_parent),
            ((double) tmp->proc_eprocess.PeakVirtualSize /
                    (double) big_vsize),
            ((double) tmp->proc_eprocess.CreateTime.HighPart /
                    (double) big_time_high),
            ((double) tmp->proc_eprocess.CreateTime.LowPart /
                    (double) big_time_low),
            ((double) tmp->proc_eprocess.ReadOperationCount.HighPart /
                    (double) big_read_high),
            ((double) tmp->proc_eprocess.ReadOperationCount.LowPart /
                    (double) big_read_low),
            ((double) tmp->proc_eprocess.WriteOperationCount.HighPart /
                    (double) big_write_high),
            ((double) tmp->proc_eprocess.WriteOperationCount.LowPart /
                    (double) big_write_low),
            ((double) tmp->proc_eprocess.OtherOperationCount.HighPart /
                    (double) big_other_high),
            ((double) tmp->proc_eprocess.OtherOperationCount.LowPart /
                    (double) big_other_low),
            ((double) tmp->proc_eprocess.PriorityClass /
                    (double) big_priority),
            ((double) tmp->proc_eprocess.Pcb.KernelTime /
                    (double) big_kernel),
            ((double) tmp->proc_eprocess.Pcb.UserTime / (double) big_user),
            ((double) tmp->proc_eprocess.Pcb.BasePriority /
                    (double) big_base),
            user_ctr
            );



        }

        tmp = tmp->next;

    }


    return 0;

}
```

```
int dump_memory3() {
    // For ACE Hackfest
    time_t current;
    uint32_t seconds;

    time (&current);
    seconds = (uint32_t) difftime(current,start_time);
    dump_memory2 (seconds % 1800 <= 20);

    return 0;

}


int dump_memory2(uint8_t do_full) {
    // For Jenny Ji
    unicode_string wdrivername; // Driver found by traversing PsLoadedModuleList
    uint64_t wptr;              // Pointer to traverse PsLoadedModuleList
    uint64_t wend;              // Endpoint for traversing PsLoadedModuleList
    uint64_t wimage_base;      // Linear address of image base of driver

    socket_list_type *wsocket;

    process_list *tmp;

    char *fname;

    char *tstring, *wstring, *wstring2;
    int i;
    time_t wtime;
    struct tm *timeinfo;
    tstring = malloc( sizeof(char) * 17);
    for (i = 0; i < 17; i++) tstring[i] = 0;
    time(&wtime);
    timeinfo = localtime(&wtime);
    sprintf_s(tstring,17,"_%04i%02i%02i%02i%02i%02i_",timeinfo->tm_year + 1900,
              timeinfo->tm_mon + 1,timeinfo->tm_mday,
              timeinfo->tm_hour,timeinfo->tm_min,timeinfo->tm_sec);
    printf("string =  %s\n",tstring);

    // File 1: Process ID, Name, User ID
    time(&wtime);
    printf("Dumping Process IDs, Names, and Users");
    fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
```

```
for (i = 0; i < strlen(feature_file_name) + 6 + 18; i++) fname[i] = 0;
strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
strcat_s(fname,strlen(feature_file_name) + 18,tstring);
strcat_s(fname,strlen(feature_file_name) + 18 + 6,"1.txt");
//fclose(feature_file);
open_feature_file(fname);
tmp = proc_list;
while (tmp != NULL)        {

    wstring2 = get_sid(tmp);
    fprintf (feature_file,"%lli\t%s\t%s \n",
    tmp->proc_eprocess.UniqueProcessId,
    tmp->proc_eprocess.ImageFileName2,
    ((tmp->users != NULL)?tmp->users->Name:( (wstring2 == NULL)?" ":wstring2))
    );
    free(wstring2);
    tmp = tmp->next;
    printf(".");

}

printf("%.2lfs",difftime(time(NULL),wtime));
free(fname);

// File 2: Process ID, Local Port, Remote Port, Protocol, IPv6
time(&wtime);
printf("\nDumping Network Information");
fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
strcat_s(fname,strlen(feature_file_name) + 18,tstring);
strcat_s(fname,strlen(feature_file_name) + 6 + 18,"2.txt");
fclose(feature_file);
open_feature_file(fname);
tmp = proc_list;
while (tmp != NULL)        {

    wsocket = tmp->sockets;
    while (wsocket != NULL) {

        fprintf(feature_file,"%i %20s %20s %12s %11s\n",
              tmp->proc_eprocess.UniqueProcessId,
              wsocket->LocalPort, wsocket->RemotePort,
              wsocket->Protocol, wsocket->Ipv6);
        wsocket = wsocket->next;
```

```
    }

    tmp = tmp->next;
    printf(".");

}

printf("%.2lfs",difftime(time(NULL),wtime));
free(fname);

// File 3: Process ID, DLL Name
time(&wtime);
printf("\nDumping Loaded Modules by Process");
fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
strcat_s(fname,strlen(feature_file_name) + 18,tstring);
strcat_s(fname,strlen(feature_file_name) + 6 + 18,"3.txt");
fclose(feature_file);
open_feature_file(fname);
tmp = proc_list;
while (tmp != NULL)        {

    show_proc_dll (tmp,feature_file);
    tmp = tmp->next;
    printf(".");

}

printf("%.2lfs",difftime(time(NULL),wtime));
free(fname);

// File 4: Process ID, opened file names
time(&wtime);
printf("\nDumping Opened Files");
fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
strcat_s(fname,strlen(feature_file_name) + 18,tstring);
strcat_s(fname,strlen(feature_file_name) + 6 + 18,"4.txt");
fclose(feature_file);
open_feature_file(fname);
tmp = proc_list;
while (tmp != NULL)        {

    show_proc_hand (tmp,"file",feature_file);
    tmp = tmp->next;
```

```
    printf(".");

}

printf("%.2lfs",difftime(time(NULL),wtime));
free(fname);

// File 5: Process ID, opened registry keys
if (do_full) {

    time(&wtime);
    printf("\nDumping Registry Keys");
    fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
    strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
    strcat_s(fname,strlen(feature_file_name) + 18,tstring);
    strcat_s(fname,strlen(feature_file_name) + 6 + 18,"5.txt");
    fclose(feature_file);
    open_feature_file(fname);
    tmp = proc_list;
    while (tmp != NULL)        {

        show_proc_hand (tmp,"key",feature_file);
        tmp = tmp->next;
        printf(".");

    }

    printf("%.2lfs",difftime(time(NULL),wtime));
    free(fname);

}


// File 6: Loaded Modules
time(&wtime);
printf("\nDumping Loaded Modules");
fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
strcat_s(fname,strlen(feature_file_name) + 18,tstring);
strcat_s(fname,strlen(feature_file_name) + 6 + 18,"6.txt");
fclose(feature_file);
open_feature_file(fname);
if (debug_data->x64) {

    wptr = memgrabke_64_LE(debug_data->PsLoadedModuleList);
```

```
    wend = memgrabke_64_LE(debug_data->PsLoadedModuleList + 0x8);

}
else {

    wptr = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList)
         & 0x00000000ffffffff;
    wend = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList
         + 0x4) & 0x00000000ffffffff;

}

while ((wptr != 0x0) && (wptr != wend)) {

    // Retrieve PE header information for the module
    if (debug_data->x64) {

        wimage_base = memgrabke_64_LE(wptr+0x30);
        wdrivername.Length = memgrabke_16_LE(wptr + 0x58);
        wdrivername.MaximumLength = wdrivername.Length;
        wdrivername.Buffer = memgrabke_64_LE(wptr + 0x60);

    }
    else {

        wimage_base = memgrabke_32_LE(wptr+0x18);
        wdrivername.Length = memgrabke_16_LE(wptr + 0x2c);
        wdrivername.MaximumLength = wdrivername.Length;
        wdrivername.Buffer = memgrabke_32_LE(wptr + 0x30);

    }

    wstring = memgrabke_unicode_LE_space(wdrivername);
    fprintf(feature_file,"Driver Name: %s\tImage Base: %x\n",
              wstring,wimage_base);
    free(wstring);
    printf(".");
    // Iterate to the next driver
    if (debug_data->x64)
    wptr = memgrabke_64_LE(wptr + 0x00);
    else
    wptr = memgrabke_32_LE(wptr + 0x00);

}
```

```
    printf("%.2lfs",difftime(time(NULL),wtime));
    fclose(feature_file);
    free(fname);

    if (do_full) {

        time(&wtime);
        printf("\nDumping Memory");
        fname = malloc(sizeof(char) * (strlen(feature_file_name) + 6 + 18));
        strcpy_s(fname,strlen(feature_file_name) + 1,feature_file_name);
        strcat_s(fname,strlen(feature_file_name) + 18,tstring);
        strcat_s(fname,strlen(feature_file_name) + 5 + 18,".dmp");
        cmat_dump_virtual(dom_char,fname);
        printf("%.2lfs",difftime(time(NULL),wtime));
        free(fname);

    }


    printf("\n");
    free(tstring);

    return 0;

}


/********************************************************************************/

/*******************************************/
/* DISPLAY SYSTEM AND PROCESS INFORMATION */
/*******************************************/

/*********************************************/
/* Show the objects that a process accesses */
/*********************************************/
uint64_t show_proc_hand(process_list *sel_proc, char *wtype, FILE *fptr)
{

    uint64_t handle_table_entry_pointer, object_pointer,
             handle_table_page_pointer, mem_pointer,
                     wTableCode, indirection, wchk;
    uint32_t wHandleCount;
    unsigned int i;
```

```
// Get the location of the handle table (the last 3 bits are 000
// (the actual values represent indirection)
wTableCode = pget_pdb_type  ( "_HANDLE_TABLE.TableCode",
            sel_proc->proc_eprocess.ObjectTable,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
// Get the # of handles the process has opened
wHandleCount = (uint32_t) pget_pdb_type( "_HANDLE_TABLE.HandleCount",
            sel_proc->proc_eprocess.ObjectTable,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
if (wHandleCount > 0x10000) return 0;

// Get the level of indirection
indirection = (wTableCode & 3);

switch (indirection) {

    case 0:
    // Iterate through the Handle Table, making a call to
    // disp_obj_details for each entry
    handle_table_entry_pointer = (wTableCode & 0xFFFFFFFFFFFFFFFCULL);
    for (i = 0; i < wHandleCount; i++ ) {

        object_pointer = pget_pdb_type  ( "_HANDLE_TABLE_ENTRY.Object",
                handle_table_entry_pointer + (i * (uint32_t)
                (get_pdb_type_sizeof("_HANDLE_TABLE_ENTRY")
                & 0x00000000ffffffff)),
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        object_pointer = object_pointer & 0xFFFFFFFFFFFFFFF8ULL;
        wchk = disp_obj_details(sel_proc,object_pointer,wtype,fptr);
        if ((!(strcmp(wtype,"clipboard"))) && wchk != 0 ) return wchk;

    }

    break;
    case 1:
    // With a level of indirection, there is a table of tables;
    // so, the plan is to go through the top level table and then
    // iterate through each table
    // pointer in it
    mem_pointer = (wTableCode & 0xFFFFFFFFFFFFFFFCULL);
    // Unfortunately there isn't a data structure that would let me use
    // the PDB functions so we've got the if...then...else
    if (debug_data->x64)
      handle_table_page_pointer = memgrabp_64_LE(
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
```

```
        mem_pointer);
else
  handle_table_page_pointer = ((uint64_t) memgrabp_32_LE(
         sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
        mem_pointer)) & 0x00000000FFFFFFFF;

// Iterate through each entry in the handle table pointer table
while (handle_table_page_pointer != 0x0) {

    handle_table_entry_pointer = (handle_table_page_pointer
                               & 0xFFFFFFFFFFFFFFFCULL);

    // Iterate through each entry in the handle table, making a call to
    // disp_obj_detail for each entry
    for (i = 0; i < ( (wHandleCount < 512) ? wHandleCount : 512); i++ ) {


        object_pointer = pget_pdb_type  ( "_HANDLE_TABLE_ENTRY.Object",
               handle_table_entry_pointer + (i *
               get_pdb_type_sizeof("_HANDLE_TABLE_ENTRY") ),
                 sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        object_pointer = object_pointer & 0xFFFFFFFFFFFFFFF8ULL;
        if (object_pointer != 0) {

            wchk = disp_obj_details(sel_proc,object_pointer,wtype,fptr);
            if ((!(strcmp(wtype,"clipboard"))) && wchk != 0 ) return wchk;

        }



    }


    // keep track of the number of handles we've already seen so we
    // know the max# remaining
    wHandleCount -= i;

    // get the next handle table pointer entry
    if (debug_data->x64)
      handle_table_page_pointer =
           memgrabp_64_LE(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
               handle_table_page_pointer + 4);
    else
      handle_table_page_pointer = ((uint64_t)
```

```
                    memgrabp_32_LE(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                                handle_table_page_pointer + 4)) & 0x00000000FFFFFFFF;

        }

        break;
        case 2:
        // This is for the second level of indirection... just haven't
        // figured out how to do this yet 'cause there aren't any examples
        break;

    }

    if (((strcmp(wtype,"clipboard"))) && (fptr == NULL)) printf("\n\n\n");
    return 0;

}



/***********************************/
/* Show the details about an object */
/***********************************/
uint64_t disp_obj_details(process_list *sel_proc, uint64_t addr,
                          char *wtype, FILE *fptr)
{

    uint64_t mem_pointer = addr;
    uint64_t obj_pointer = addr + get_pdb_type_offset( "_OBJECT_HEADER.Body");
    uint8_t wSharedRead, wSharedWrite, wSharedDelete, wtypeindex;
    unicode_string wtypename,wFileName;
    key_object      *wkey;
    char *wstring, *wstring2;

    // Case I: Object Type exists in the Object Header structure
    if (get_pdb_fld_exists("_OBJECT_HEADER.Type")) {

        // mem_pointer points to the object type
        mem_pointer             = pget_pdb_type( "_OBJECT_HEADER.Type",
                   addr, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        if (MYDEBUG) printf("**disp_obj_details** Type Pointer exists\n");

        // Case II: Object Type isn't in the Object Header structure (e.g.,
        //  Windows 7); the only thing there is an index to the Object Type Table
```

```
}
else {

    // get the index into the table
    wtypeindex              = (uint8_t) pget_pdb_type(
                   "_OBJECT_HEADER.TypeIndex", addr,
                   sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    // pull the pointer to the object type our the of the Object Type
    // Table populated in the meta-data
    mem_pointer = debug_data->objTypeTable[wtypeindex];

}


wtypename.Length        = (uint16_t) pget_pdb_type(
               "_OBJECT_TYPE.Name.Length", mem_pointer,
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wtypename.MaximumLength = (uint16_t) pget_pdb_type(
              "_OBJECT_TYPE.Name.MaximumLength", mem_pointer,
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wtypename.Buffer        = pget_pdb_type( "_OBJECT_TYPE.Name.Buffer",
              mem_pointer,
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

// If the length of the object type doesn't make sense, something went
//  wrong -- get out of here
if ((wtypename.Length > 40) || (wtypename.Length < 3)) return 0;

// Now depending on the parameter passed in, provide relevant
// information for the requested type
wstring = memgrabp_unicode_LE(wtypename,
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
if  (!(strcmp(wstring,"File") ) && !(strcmp(wtype,"file")) ) // FILES
{

    wSharedRead     = (uint8_t) pget_pdb_type  ( "_FILE_OBJECT.SharedRead",
            obj_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wSharedWrite    = (uint8_t) pget_pdb_type  ( "_FILE_OBJECT.SharedWrite",
            obj_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wSharedDelete   = (uint8_t) pget_pdb_type  ( "_FILE_OBJECT.SharedDelete",
            obj_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFileName.Length = (uint16_t) pget_pdb_type(
            "_FILE_OBJECT.FileName.Length", obj_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFileName.MaximumLength = (uint16_t) pget_pdb_type(
```

```
                "_FILE_OBJECT.FileName.MaximumLength", obj_pointer,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        wFileName.Buffer        = pget_pdb_type(
                "_FILE_OBJECT.FileName.Buffer", obj_pointer,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

        wstring2 = memgrabp_unicode_LE(wFileName,
                    sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        if (fptr == NULL)
          printf ("%llx\t\t (%s%s%s)  (File) %s \n",
                obj_pointer, (wSharedRead?"R":"-"),
                (wSharedWrite?"W":"-"), (wSharedDelete?"D":"-"),
                ((wFileName.Buffer != 0)?wstring2:" "));
        else
          fprintf (fptr,"%lli %llx\t\t (%s%s%s)  (File)\t %s \n",
                      sel_proc->proc_eprocess.UniqueProcessId,obj_pointer,
                      (wSharedRead?"R":"-"),(wSharedWrite?"W":"-"),
                      (wSharedDelete?"D":"-"),
                      ((wFileName.Buffer != 0)?wstring:" "));
        free(wstring2);
        free(wstring);
        return 1;

    }

    else if  ((!(strcmp(wstring,"Key"))) &&
                 (!(strcmp(wtype,"key")))) // REGISTRY
    {

        wkey = get_key_object(sel_proc,obj_pointer);
        if (wkey != NULL) {

            if (fptr == NULL)
              printf ("Key %llx\t\t%s/%s\n",
                     obj_pointer,wkey->hive_name,wkey->cell.name);
            else
              fprintf (fptr,"%lli %llx\t\t%s/%s\n"
                     ,sel_proc->proc_eprocess.UniqueProcessId,
                    obj_pointer, wkey->hive_name,wkey->cell.name);
            free(wkey->cell.name);
            free(wkey);
            free(wstring);
            return 1;

        }
```

```
    free(wkey->cell.name);
    free(wkey);

}

else if ((!(strcmp(wstring,"Section"))) && (!(strcmp(wtype,"file"))))
{

    mem_pointer = pget_pdb_type  ( "_SECTION_OBJECT.Segment",
            obj_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    mem_pointer = pget_pdb_type  ( "_SEGMENT_OBJECT.BaseAddress",
             mem_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    mem_pointer = pget_pdb_type  ( "_CONTROL_AREA.FilePointer",
             mem_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wSharedRead = (uint8_t) pget_pdb_type  ( "_FILE_OBJECT.SharedRead",
             mem_pointer,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wSharedWrite = (uint8_t) pget_pdb_type  ( "_FILE_OBJECT.SharedWrite",
             mem_pointer,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wSharedDelete = (uint8_t) pget_pdb_type  ( "_FILE_OBJECT.SharedDelete",
             mem_pointer,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFileName.Length        = (uint16_t) pget_pdb_type( "_FILE_OBJECT.FileName.L
             mem_pointer,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFileName.MaximumLength = (uint16_t) pget_pdb_type(
             "_FILE_OBJECT.FileName.MaximumLength", mem_pointer,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFileName.Buffer        = pget_pdb_type(
             "_FILE_OBJECT.FileName.Buffer", mem_pointer,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

    wstring2 = memgrabp_unicode_LE(wFileName,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    if (fptr == NULL)
      printf ("%llx\t\t (%s%s%s)  (Section) %s \n",obj_pointer,
            (wSharedRead?"R":"-"),(wSharedWrite?"W":"-"),
          (wSharedDelete?"D":"-"),(wFileName.Buffer != 0?wstring2:" "));
    else
      fprintf (fptr,"%lli %llx\t\t (%s%s%s)  (Section)\t %s \n",
             sel_proc->proc_eprocess.UniqueProcessId,obj_pointer,
             (wSharedRead?"R":"-"),(wSharedWrite?"W":"-"),
             (wSharedDelete?"D":"-"),
```

```
                (wFileName.Buffer != 0?wstring2:" "));
        free(wstring2);
        free(wstring);
        return 1;

    }

    else if ( (strcmp(wstring,"Key")) && (strcmp(wstring,"File")) &&
            !(strcmp(wtype,"other")) ) // EVERYTHING ELSE
    {

        printf ("%15s:\t %llx\n", wstring,obj_pointer);
        free(wstring);
        return 1;

    }

    else if ((!(strcmp(wstring,"WindowStation"))) &&
             (!(strcmp(wtype,"clipboard")))) // Clipboard
    {

        if (MYDEBUG) printf ("%15s:\t %llx\n", wstring,obj_pointer);
        free(wstring);
        return obj_pointer;

    }

    else if (MYDEBUG)
    printf ("%15s:\t %llx (phys: %llx)\n",
           wstring,obj_pointer,
           vtop(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
           obj_pointer));
    free(wstring);
    return 0;

}


/*************************************/
/* Show the DLLs loaded for a process */
/*************************************/
void show_proc_dll(process_list *sel_proc, FILE *fptr)
{

    uint64_t whead, mem_pointer, wInLoadOrderModuleList;
```

```
unicode_string wFullDllName, wBaseDllName;
char *wstring2, *wstring3;

// The DLLs are stored in the load data off the Process Environment
// Block (PEB) off the EPROCESS record
whead = pget_pdb_type  ( "_PEB_LDR_DATA.InLoadOrderModuleList.Flink",
          sel_proc->proc_peb.Ldr,
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
mem_pointer = whead;

if (fptr == NULL) printf("\n\n\n");
while (mem_pointer > 0) {

    wInLoadOrderModuleList       = pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.InLoadOrderLinks.Flink", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFullDllName.Length             = (uint16_t) pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.FullDllName.Length", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFullDllName.MaximumLength     = (uint16_t) pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.FullDllName.MaximumLength", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wFullDllName.Buffer             = pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.FullDllName.Buffer", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wBaseDllName.Length             = (uint16_t) pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.BaseDllName.Length", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wBaseDllName.MaximumLength     = (uint16_t) pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.BaseDllName.MaximumLength", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wBaseDllName.Buffer             = pget_pdb_type  (
            "_LDR_DATA_TABLE_ENTRY.BaseDllName.Buffer", mem_pointer,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

    wstring2 = memgrabp_unicode_LE(wFullDllName,
                       sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wstring3 = memgrabp_unicode_LE(wBaseDllName,
                       sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    if ((wBaseDllName.Length > 0) && (isprintable(memgrabp_8_LE(
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
          wBaseDllName.Buffer))))
    if (fptr == NULL)
      printf("Base Dll Name: %-16s\tFull Name: %s\n", wstring3, wstring2);
    else
```

```
            fprintf(fptr,"%lli\t%s\t%s\n",
                 sel_proc->proc_eprocess.UniqueProcessId, wstring3, wstring2);
        free(wstring2);
        free(wstring3);

        if (wInLoadOrderModuleList == whead)
        break;
        else
        mem_pointer = wInLoadOrderModuleList;

    }

    if (fptr == NULL) printf("\n\n\n");

}


/************************************/
/* Show the parameters for a process  */
/************************************/
void show_proc_env (process_list *sel_proc)
{

    unicode_string wImagePathName, wCommandLine, wWindowTitle, wDesktopInfo,
                   wShellInfo, wRuntimeData, wDllPath;
    char * wstring;

    // This stuff is in the Run Time Library Parameters off the Process
    // Environment Block (PEB) off the EPROCESS record
    wImagePathName.Length          = (uint16_t) pget_pdb_type  (
            "_RTL_USER_PROCESS_PARAMETERS.ImagePathName.Length",
            sel_proc->proc_peb.ProcessParameters,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wImagePathName.MaximumLength  = (uint16_t) pget_pdb_type  (
            "_RTL_USER_PROCESS_PARAMETERS.ImagePathName.MaximumLength",
            sel_proc->proc_peb.ProcessParameters,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wImagePathName.Buffer          = pget_pdb_type  (
            "_RTL_USER_PROCESS_PARAMETERS.ImagePathName.Buffer",
            sel_proc->proc_peb.ProcessParameters,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wCommandLine.Length              = (uint16_t) pget_pdb_type  (
            "_RTL_USER_PROCESS_PARAMETERS.CommandLine.Length",
            sel_proc->proc_peb.ProcessParameters,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
```

```
wCommandLine.MaximumLength      = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.CommandLine.MaximumLength",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wCommandLine.Buffer              = pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.CommandLine.Buffer",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wWindowTitle.Length              = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.WindowTitle.Length",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wWindowTitle.MaximumLength      = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.WindowTitle.MaximumLength",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wWindowTitle.Buffer              = pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.WindowTitle.Buffer",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wDesktopInfo.Length              = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.DesktopInfo.Length",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wDesktopInfo.MaximumLength      = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.DesktopInfo.MaximumLength",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wDesktopInfo.Buffer              = pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.DesktopInfo.Buffer",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wShellInfo.Length                = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.ShellInfo.Length",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wShellInfo.MaximumLength      = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.ShellInfo.MaximumLength",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wShellInfo.Buffer              = pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.ShellInfo.Buffer",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wRuntimeData.Length              = (uint16_t) pget_pdb_type  (
```

```
        "_RTL_USER_PROCESS_PARAMETERS.RuntimeData.Length",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wRuntimeData.MaximumLength      = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.RuntimeData.MaximumLength",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wRuntimeData.Buffer              = pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.RuntimeData.Buffer",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wDllPath.Length                  = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.DllPath.Length",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wDllPath.MaximumLength          = (uint16_t) pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.DllPath.MaximumLength",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wDllPath.Buffer                  = pget_pdb_type  (
        "_RTL_USER_PROCESS_PARAMETERS.DllPath.Buffer",
        sel_proc->proc_peb.ProcessParameters,
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

printf("\n\n\n");
printf("Process Environment Information:\n");
wstring = memgrabp_unicode_LE(wImagePathName,
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
printf("\tExecutable File:\t%s\n", wstring);
free(wstring);
wstring = memgrabp_unicode_LE(wCommandLine,
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
printf("\tCommand Line:\t%s\n", wstring);
free(wstring);
wstring = memgrabp_unicode_LE(wWindowTitle,
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
printf("\tWindow Title:\t%s\n", wstring);
free(wstring);
wstring = memgrabp_unicode_LE(wDesktopInfo,
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
printf("\tDesktop Info:\t%s\n", wstring);
free(wstring);
wstring = memgrabp_unicode_LE(wShellInfo,
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
printf("\tShell Info:\t%s\n", wstring);
```

```
    free(wstring);
    wstring = memgrabp_unicode_LE(wRuntimeData,
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    printf("\tRuntime Data:\t%s\n", wstring);
    free(wstring);
    wstring = memgrabp_unicode_LE(wDllPath,
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    printf("\tDll Path:\t%s\n", wstring);
    free(wstring);
    printf("\n\n\n");

}


/******************************/
/* Show the System Information */
/******************************/
int show_system_info ()
{

    process_list *tmp = proc_list;
    char osversion[10];
    printf ("\n\nShowing System Information\n\n\n");

    // Start at the first process and keep going until you find one with
    // valid system information
    while (tmp != NULL)
    {

        if ( (tmp->proc_peb.OSMajorVersion == 4)
                && (tmp->proc_peb.OSMinorVersion == 0)
                && (tmp->proc_peb.OSPlatformID == 1))
              strcpy_s(osversion, 10, "Win95");
        if ( (tmp->proc_peb.OSMajorVersion == 4)
                && (tmp->proc_peb.OSMinorVersion == 10)
                && (tmp->proc_peb.OSPlatformID == 1))
              strcpy_s(osversion, 10, "Win98");
        if ( (tmp->proc_peb.OSMajorVersion == 4)
                && (tmp->proc_peb.OSMinorVersion == 90)
                && (tmp->proc_peb.OSPlatformID == 1))
              strcpy_s(osversion, 10, "WinME");
        if ( (tmp->proc_peb.OSMajorVersion == 3)
                && (tmp->proc_peb.OSMinorVersion == 51)
                && (tmp->proc_peb.OSPlatformID == 2))
              strcpy_s(osversion, 10, "WinNT351");
```

```
if ( (tmp->proc_peb.OSMajorVersion == 4)
            && (tmp->proc_peb.OSMinorVersion == 0)
            && (tmp->proc_peb.OSPlatformID == 2))
      strcpy_s(osversion, 10, "WinNT4");
if ( (tmp->proc_peb.OSMajorVersion == 5)
            && (tmp->proc_peb.OSMinorVersion == 0)
            && (tmp->proc_peb.OSPlatformID == 2))
      strcpy_s(osversion, 10, "Win2000");
if ( (tmp->proc_peb.OSMajorVersion == 5)
            && (tmp->proc_peb.OSMinorVersion == 1)
            && (tmp->proc_peb.OSPlatformID == 2))
      strcpy_s(osversion, 10, "WinXP");
if ( (tmp->proc_peb.OSMajorVersion == 5)
             && (tmp->proc_peb.OSMinorVersion == 2)
             && (tmp->proc_peb.OSPlatformID == 2))
      strcpy_s(osversion, 10, "Win2003");
if ( (tmp->proc_peb.OSMajorVersion == 6)
             && (tmp->proc_peb.OSMinorVersion == 0)
             && (tmp->proc_peb.OSPlatformID == 2))
      strcpy_s(osversion, 10, "Vista");
if ( tmp->proc_peb.OSPlatformID == 3)
      strcpy_s(osversion, 10, "WinCE");
if ( tmp->proc_peb.OSPlatformID == 0)
      strcpy_s(osversion, 10, "Win32s");

if ((tmp->proc_peb.OSMajorVersion > 3)
               && (tmp->proc_peb.OSMajorVersion < 7)) {

    printf("OS:\t%s\tVersion %i.%i SP %i \t",
            osversion,tmp->proc_peb.OSMajorVersion,
            tmp->proc_peb.OSMinorVersion,
           ((tmp->proc_peb.OSCSDVersion >> 8) & 0xFF));
    printf("build:\t%i\n", tmp->proc_peb.OSBuildNumber);
    printf("Number of processors: %i\n",
             tmp->proc_peb.NumberOfProcessors);

    printf("Machine Type: %s \t%s \nKDBG (phys) = %x ",
           debug_data->MachineString,
           (debug_data->PaeEnabled?"PAE Enabled":"PAE Disabled"),
           debug_data->KDBG_loc);
    printf("\t Kernel PageDirectoryBase (phys) = %x \t",
           debug_data->kDTB);
    printf(" Kernel PE Image (virt) = %x\n\n",
           debug_data->KernBase);
    break;
```

```
        }

        tmp = tmp->next;

    }

    return 0;

}


/***********************************************/
/* Show the OS devices associated with a process */
/***********************************************/
void list_os_devices(process_list *sel_proc)
{

    #define hash_bucket_count 37
    uint32_t mem_pointer;
    unsigned int hash_buckets[hash_bucket_count];
    unsigned int i;
    typedef struct dev_list_type {

        struct dev_list_type *next;
        struct dev_list_type *prev;
        char *dev_name;

    }
    dev_list;
    dev_list    *dev_top = NULL;

    // Grab the pointer from the EPROCESS structure
    mem_pointer = (uint32_t) sel_proc->proc_eprocess.DeviceMap;

    // Read pointer to Object Directory
    mem_pointer = memgrabp_32_LE(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                   mem_pointer);
    // Mem_pointer now points to _OBJECT_DIRECTORY

    // Grab the hash bucket items from the Object Directory
    for (i=0; i < hash_bucket_count; i++)
            hash_buckets[i] = memgrabp_32_LE(
                         sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                        (mem_pointer + (4 * i)));
```

```
// Loop through hash buckets
for (i=0; i < hash_bucket_count; i++)
{

    unsigned int next_obj_address, object_address;
    object_symbolic_link sym_link;

    mem_pointer = hash_buckets[i];

    while (1)
    {

        dev_list *tmp = malloc(sizeof *tmp);
        //Grab a _OBJECT_DIRECTORY_ENTRY
        next_obj_address =
               memgrabp_32_LE(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
               mem_pointer);
        object_address =
               memgrabp_32_LE(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
               (mem_pointer + 4));


        //The object_address is a _OBJECT_SYMBOLIC_LINK
        sym_link.CreationTime.LowPart = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address);
        sym_link.CreationTime.HighPart = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x4);
        sym_link.LinkTarget.Length = memgrabp_16_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x8);
        sym_link.LinkTarget.MaximumLength = memgrabp_16_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0xa);
        sym_link.LinkTarget.Buffer = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0xc);
        sym_link.LinkTargetRemaining.Length = memgrabp_16_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x10);
        sym_link.LinkTargetRemaining.MaximumLength = memgrabp_16_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x12);
```

```
        sym_link.LinkTargetRemaining.Buffer = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x14);
        sym_link.LinkTargetObject = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x18);
        sym_link.DosDeviceDriveIndex = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                  object_address + 0x1c);


        tmp->dev_name = memgrabp_unicode_LE(sym_link.LinkTarget,
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        tmp->next = dev_top;
        tmp->prev = NULL;
        if (tmp->next != NULL) tmp->next->prev = tmp;
        dev_top = tmp;

        if (next_obj_address > 0) mem_pointer = next_obj_address;
        else break;

    }


}

{

    int change = 0;
    dev_list *tmp = dev_top;
    dev_list *tmp2 = dev_top;

    do
    {

        dev_list *old1;
        dev_list *old2;
        tmp = dev_top;
        change = 0;
        while (tmp->next != NULL)
        {

            if (tmp->next == NULL) break;
            if (strcmp (tmp->dev_name, tmp->next->dev_name) > 0)
            {
```

```
            change++;
            if (dev_top == tmp) dev_top = tmp->next;
            old1 = tmp;
            old2 = tmp->next;

            old2->prev = old1->prev;
            if (old1->prev != 0) old1->prev->next = old2;
            old1->next = old2->next;
            if (old2->next != 0) old2->next->prev = old1;
            old2->next = old1;
            old1->prev = old2;

        }

        else tmp = tmp->next;

    }


}
while (change);
tmp = dev_top;
while (1)
{

    if (tmp->next != NULL)
    {

        if (!(strcmp(tmp->dev_name,tmp->next->dev_name)))
        {

            tmp = tmp->next;
            continue;

        }

        if (strlen(tmp->dev_name) > 3) printf("\t%s\n",tmp->dev_name);
        tmp = tmp->next;

    }

    else
    {
```

```
                if (strlen(tmp->dev_name) > 3) printf("\t%s\n",tmp->dev_name);
                break;

            }


        }

        tmp = dev_top;
        while (tmp->next !=NULL) tmp = tmp->next;

        while (tmp->prev !=NULL)
        {

            tmp2 = tmp;
            tmp = tmp->prev;
            free(tmp2);

        }


        printf ("\n\n\n");

    }


}


/******************************************************************************/
/***************************/
/* MISCELLANEOUS UTILITIES */
/***************************/

/******************************************/
/* Return a process' token id as a string */
/******************************************/
char * get_sid(process_list *tmp) {


    unsigned char * wtoken_sid;
    char * ret_string;
    ascii_string wname;
    uint32_t ptr_count, wvalue;
    uint64_t token_address, ptr_array, wpointer;
```

```
    int i,j;
    char wtmp[70];

    if (debug_data->x64)
    token_address = tmp->proc_eprocess.Token & 0xFFFFFFFFFFFFFFF0ULL;
    else
    token_address = tmp->proc_eprocess.Token & 0xFFFFFFFFFFFFFFF8ULL;
    ptr_array = pget_pdb_type  ( "_TOKEN.UserAndGroups", token_address,
                tmp->proc_eprocess.Pcb.DirectoryTableBase);
    if (MYDEBUG) printf("**get_sid *_sid = %llx\n",ptr_array);
    ptr_count = (uint32_t) pget_pdb_type  ( "_TOKEN.UserAndGroupCount",
                token_address, tmp->proc_eprocess.Pcb.DirectoryTableBase);
    if ((ptr_count > 100) || (ptr_count = 0)) return NULL;

    wpointer = pget_pdb_type  ( "_SID_AND_ATTRIBUTES.Sid", ptr_array,
                tmp->proc_eprocess.Pcb.DirectoryTableBase);
    wname.Length = 0x1c;
    wname.MaximumLength = wname.Length;
    wname.Buffer = wpointer;
    wtoken_sid = memgrabp_string_LE (wname,
                   tmp->proc_eprocess.Pcb.DirectoryTableBase);
    if (MYDEBUG) printf("**get_sid     token_sid = %s\n",wtoken_sid);
    ret_string = malloc( sizeof(char) * 70);
    for (i = 0; i < 70; i++) ret_string[i] = 0;
    sprintf_s(wtmp,70,"%i-%i",wtoken_sid[0],wtoken_sid[1]);
    strcpy_s(ret_string,70,wtmp);
    for (i = 0; i < min_u32((69 - strlen(wtmp)) / 4, wtoken_sid[1]); i++) {

        wvalue = 0;
        for (j = 0; j < 4; j++) {

            wvalue += wtoken_sid[8 + i * 4 + j] * (int) pow(0x100,j);

        }

        sprintf_s(wtmp,70,"%s-%u",wtmp,wvalue);

    }
    strcpy_s(ret_string,70,wtmp);
    free(wtoken_sid);
    return ret_string;

}
```

```
/*******************************************/
/* Do a byte for byte compare of two strings */
/*******************************************/
int bytecmp(unsigned char *str1, unsigned char *str2, int wlength)
{

    int i;
    for (i = 0; i < wlength; i++)
    if (str1[i] != str2[i]) return i;
    return 0;

}


/********************************/
/* Convert a string to uppercase */
/********************************/
char * strtoupper (char *lower) {

    uint32_t i;
    char *upper;
    upper = malloc(sizeof(char) * (strlen(lower) + 1));
    for (i = 0; i < strlen(lower); i++)    upper[i] = toupper(lower[i]);
    upper[strlen(lower)] = 0;
    return upper;

}


/********************************************************/
/* Check if a character is in the ASCII range of printable */
/********************************************************/
int isprintable(unsigned char k)
{

    if ( (k >= 0x20) && (k <= 0x7e) ) return 1;
    else return 0;

}


/*************************************/
/* Return the minimum of two integers */
/*************************************/
uint32_t min_u32(uint32_t x, uint32_t y) {
```

```
  if (x < y) return x; else return y;
}

/*************************/
/* Show Clipboard contents */
/*************************/
int show_clipboard(process_list *sel_proc)
{

    unsigned int whead, mem_pointer, wInLoadOrderModuleList;
    unicode_string wBaseDllName;
    uint32_t i, ctr;
    char *test_dll;
    char *wclip_string;

    char *pdb_url, *wpdb_url;   // URL for MS Symbol Server to retrieve tcpip.pdb
    pdb_symbol_t *wwptr;        // Pointer to the symbol_t structure for the symbol
    char *fname, *xname, *test_section;

    uint16_t nbr_sections;
    uint32_t pe_offset, data_segment;
    uint32_t gphn_offset, gphn_offset_ptr;
    uint64_t wimage_base, hdr_base;
    uint64_t gphn, wgphn;
    uint16_t wformat;
    uint64_t whandle, wmemloc;
    uint64_t wsectionptr;
    whead = pget_pdb_type_uint32  ( "_PEB_LDR_DATA.InLoadOrderModuleList.Flink",
            sel_proc->proc_peb.Ldr,
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    mem_pointer = whead;
    while (mem_pointer > 0) {

        wInLoadOrderModuleList       = pget_pdb_type_uint32  (
                 "_LDR_DATA_TABLE_ENTRY.InLoadOrderLinks.FLink", mem_pointer,
                 sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        wBaseDllName.Length             = pget_pdb_type_uint16  (
                 "_LDR_DATA_TABLE_ENTRY.BaseDllName.Length", mem_pointer,
                 sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        wBaseDllName.MaximumLength     = pget_pdb_type_uint16  (
                  "_LDR_DATA_TABLE_ENTRY.BaseDllName.MaximumLength", mem_pointer,
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        wBaseDllName.Buffer             = pget_pdb_type_uint32  (
                  "_LDR_DATA_TABLE_ENTRY.BaseDllName.Buffer", mem_pointer,
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
```

```
wsectionptr = pget_pdb_type(
          "_LDR_DATA_TABLE_ENTRY.SectionPointer", mem_pointer,
          sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
test_dll = memgrabp_unicode_LE(wBaseDllName,
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
for    (i = 0; i < strlen(test_dll);i++)
    test_dll[i] = toupper(test_dll[i]);
if (MYDEBUG) printf("checking... %s\n",test_dll);
if (!strcmp(test_dll,"USER32.DLL")) {

    wimage_base = pget_pdb_type_uint32  ( "_LDR_DATA_TABLE_ENTRY.DllBase",
              mem_pointer, sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    // Retrieve the symbol information from the PDB file
    // First, load the pdb information
    if (user32_pdb == NULL) {

        pdb_url = get_pdb_url(wimage_base, PDB_RET_URL,debug_data->x64,
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        if (pdb_url == NULL) {

            fname = malloc(sizeof(char) * (strlen("USER32.DLL") +
                       strlen(DATADIRECTORY) + 1));
            strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
            strcat_s(fname,strlen(DATADIRECTORY) +
                      strlen("USER32.DLL") + 1,"USER32.DLL");
            pdb_url = get_pdb_url_disk(fname,PDB_RET_URL,debug_data->x64);
            wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
            strcpy_s(wpdb_url, strlen(pdb_url) + 1, pdb_url);
            xname = malloc(sizeof(char) * (strlen(get_pdb_url_disk(fname,
                 PDB_RET_NAME
                  ,debug_data->x64)) + strlen(DATADIRECTORY) + 1));
            strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
            strcat_s(xname,strlen(DATADIRECTORY) +
              strlen(get_pdb_url_disk(fname,
              PDB_RET_NAME,debug_data->x64)) +
              1,get_pdb_url_disk(fname,
              PDB_RET_NAME,debug_data->x64));
            if (GET_PDB)
               xname = download_pdb_file(
                       get_pdb_url_disk(fname,PDB_RET_NAME,
                       debug_data->x64), wpdb_url, DATADIRECTORY);
            user32_pdb = parse_pdb_file(xname, PDB_DEFAULT);
            free(fname);
            free(wpdb_url);
            free(xname);
```

```
    }
    else {

        wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
        strcpy_s(wpdb_url, strlen(pdb_url) + 1, pdb_url);
        xname = malloc(sizeof(char) * (strlen(get_pdb_url(wimage_base,
                PDB_RET_NAME,debug_data->x64,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase)) +
                 strlen(DATADIRECTORY) + 1));
        strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(xname,strlen(DATADIRECTORY) +
            strlen(get_pdb_url(wimage_base,PDB_RET_NAME,debug_data->x64,
           sel_proc->proc_eprocess.Pcb.DirectoryTableBase)) + 1,
           get_pdb_url(wimage_base,PDB_RET_NAME,debug_data->x64,
           sel_proc->proc_eprocess.Pcb.DirectoryTableBase));
        if (GET_PDB)
            xname = download_pdb_file(get_pdb_url(
                wimage_base,PDB_RET_NAME,
                debug_data->x64,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase),
                wpdb_url, DATADIRECTORY);
        user32_pdb = parse_pdb_file(xname, PDB_DEFAULT);
        free(wpdb_url);
        free(xname);

    }


}

free(pdb_url);
// Second, find where the different sections start
pe_offset =  memgrabp_32_LE(
         sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
         wimage_base + 0x3c);
hdr_base = wimage_base + pe_offset + 0x18;
data_segment         = get_pdb_section(user32_pdb,
                        ".data",5)->virtual_address;

// First get the gphn symbol
wwptr = get_pdb_symbol(user32_pdb,"gphn",4);
if (wwptr == 0) wwptr = get_pdb_symbol(user32_pdb,"_gphn",5);

if (wwptr != NULL) {
```

```
    gphn_offset = wwptr->offset; // Unadjusted offset in the section
    gphn_offset_ptr = get_pdb_symbol_loc(
                 user32_pdb,gphn_offset,data_segment);//Adj RVA

}
else {

    gphn_offset = 0;
    gphn_offset_ptr = 0;

}

gphn = wimage_base + (((uint64_t) gphn_offset_ptr)
     & 0x00000000ffffffff);

if (debug_data->x64)
  gphn = memgrabp_64_LE(
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase,gphn);
else
  gphn = ((uint64_t) memgrabp_32_LE(
        sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
        gphn)) & 0x00000000ffffffff;
wgphn = gphn;
ctr = 0;
while (wgphn != 0) {

    if (debug_data->x64) {

        whandle = memgrabp_64_LE(
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
              wgphn + 0x18);
        wformat = memgrabp_16_LE(
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
              wgphn + 0x08);
        wmemloc = memgrabp_64_LE(
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
              whandle);
        wclip_string = memgrabp_unicode_LE_0 (wmemloc,
               sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

        if (strlen(wclip_string) > 0) {
          printf("\nSession = %i   Process: %lli  %s    ",
                  sel_proc->proc_eprocess.SessionId,
                  sel_proc->proc_eprocess.UniqueProcessId,
```

```
                                sel_proc->proc_eprocess.ImageFileName2);
printf("clipboard (format %i) = %s\n",
                                 wformat,wclip_string);
 }
                      else {

                          wclip_string = memgrabp_unicode_LE_0 (whandle,
                                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
                          if (strlen(wclip_string) > 0) {
                           printf("\nSession = %i      Process: %lli  %s       ",
                               sel_proc->proc_eprocess.SessionId,
                               sel_proc->proc_eprocess.UniqueProcessId,
                               sel_proc->proc_eprocess.ImageFileName2);
                           printf("indirection clipboard (format %i) = %s\n",
                                wformat,wclip_string);
       }
                          else {
                            printf("\nSession = %i ",
                                  sel_proc->proc_eprocess.SessionId);
                            printf("clipboard (fmt %x) handle = %llx (phys: %llx) ",
                                  wformat,whandle,
                                 vtop(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                                  whandle));
                            printf("memloc = %llx (phys: %llx)\n",
                                  wmemloc,
                                  vtop(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                                  wmemloc));
                         }
                      }

                      free(wclip_string);

                  }
                  else {

                      whandle = memgrabp_32_LE(
                                sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                                wgphn + 0x0c);
                      wformat = memgrabp_16_LE(
                                sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                                wgphn + 0x04);
                      wmemloc = memgrabp_32_LE(
                             sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                             whandle);
                      wclip_string = memgrabp_unicode_LE_0 (whandle,
```

```
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

            if (strlen(wclip_string) > 0) {
              printf("\nSession = %i  Process: %lli  %s   clipboard ",
                     sel_proc->proc_eprocess.SessionId,
                     sel_proc->proc_eprocess.UniqueProcessId,
                     sel_proc->proc_eprocess.ImageFileName2);
              printf("(format %i) = %s\n",wformat,wclip_string);
            }
            else
              wclip_string = memgrabp_unicode_LE_0 (wmemloc,
                 sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
            if (strlen(wclip_string) > 0) {
              printf("\nSession = %i      Process: %lli  %s       clipboard"
                 sel_proc->proc_eprocess.SessionId,
                 sel_proc->proc_eprocess.UniqueProcessId,
                 sel_proc->proc_eprocess.ImageFileName2);
              printf(" (format %i) = %s\n",wformat,wclip_string);

            } else {
              printf("\nSession = %i  Process: %lli  %s  clipboard",
                sel_proc->proc_eprocess.SessionId,
                sel_proc->proc_eprocess.UniqueProcessId,
                sel_proc->proc_eprocess.ImageFileName2);
              printf(" (format %x) handle = %llx (phys: %llx)\n",
                wformat,wmemloc,
                vtop(sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                wmemloc));
            }
            free(wclip_string);

        }

        // Format = 0xd = Ascii

        wgphn = memgrabp_64_LE(
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase,wgphn);
        ctr++;
        if (ctr > 20) break;

    }

    return 1;

}
```

```
        free(test_dll);
        if (wInLoadOrderModuleList == whead)
        break;
        else
        mem_pointer = wInLoadOrderModuleList;

    }

    return 0;

}


/***************************/
/* Show Clipboard contents */
/***************************/
int show_clipboardk(process_list *sel_proc)
{

    uint32_t i, gotone;
    char *wclip_string;

    uint32_t gSharedInfo_offset, gSharedInfo_offset_ptr;
    uint64_t gSharedInfo;
    uint64_t wformat_table, chk_format, rec_size, memloc;
    uint32_t wformat_count, wformat;
    uint64_t whandle, low16, high16;

    uint64_t wWindowStation;    // Process' WindowStation; used for clipboard

    unicode_string wdrivername; // Driver found by traversing PsLoadedModuleList
    uint64_t wptr;              // Pointer to traverse PsLoadedModuleList
    uint64_t wend;              // Endpoint for traversing PsLoadedModuleList
    uint64_t wimage_base;       // Linear address of image base of driver

    char *pdb_url, *wpdb_url;   // URL needed for MS Symbol Server to retrieve pdb
    pdb_symbol_t *wwptr;        // Pointer to the symbol_t structure for the symbol
    char *xname, *fname;
    char * test_section;
    char *wstring;

    uint32_t pe_offset;         // PE offset in the PE file
    uint32_t nbr_sections;      // Number of sections for the PE
    uint64_t hdr_base;          // RVA of the PE header
```

```
uint32_t data_segment;      // RVA of the .data segment
uint32_t rdata_segment;     // RVA of the .rdata segment

// Get the list of loaded system modules
if (debug_data->x64) {

    wptr = memgrabke_64_LE(debug_data->PsLoadedModuleList);
    wend = memgrabke_64_LE(debug_data->PsLoadedModuleList + 0x8);

}
else {

    wptr = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList)
         & 0x00000000ffffffff;
    wend = (uint64_t) memgrabke_32_LE(debug_data->PsLoadedModuleList
         + 0x4) & 0x00000000ffffffff;

}

// Scroll through the list until you find tcpip.sys
while ((wptr != 0x0) && (wptr != wend)) {


    // Retrieve PE header information for the module
    if (debug_data->x64) {

        wimage_base = memgrabke_64_LE(wptr+0x30);
        wdrivername.Length = memgrabke_16_LE(wptr + 0x58);
        wdrivername.MaximumLength = wdrivername.Length;
        wdrivername.Buffer = memgrabke_64_LE(wptr + 0x60);

    }
    else {

        wimage_base = memgrabke_32_LE(wptr+0x18);
        wdrivername.Length = memgrabke_16_LE(wptr + 0x2c);
        wdrivername.MaximumLength = wdrivername.Length;
        wdrivername.Buffer = memgrabke_32_LE(wptr + 0x30);

    }


    // If you've found tcpip.sys
    wstring = memgrabke_unicode_LE_space (wdrivername);
    if (!strcmp("win32k.sys",wstring)) {
```

```
// Retrieve the symbol information from the PDB file
// First, load the pdb information
if (win32k_pdb == NULL) {

    pdb_url = get_pdb_url(wimage_base,
         PDB_RET_URL,debug_data->x64,debug_data->kDTB);
    if (pdb_url == NULL) {

        fname = malloc(sizeof(char) * (strlen("WIN32K.SYS") +
               strlen(DATADIRECTORY) + 1));
        strcpy_s(fname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(fname,strlen(DATADIRECTORY) +
               strlen("WIN32K.SYS") + 1,"WIN32K.SYS");
        pdb_url = get_pdb_url_disk(fname,PDB_RET_URL,
               debug_data->x64);
        wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
        strcpy_s(wpdb_url, strlen(pdb_url) + 1, pdb_url);
        xname = malloc(sizeof(char) *
                 (strlen(get_pdb_url_disk(fname,
                 PDB_RET_NAME,debug_data->x64)) +
                 strlen(DATADIRECTORY) + 1));
        strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(xname,strlen(DATADIRECTORY) +
             strlen(get_pdb_url_disk(fname,PDB_RET_NAME,
             debug_data->x64)) +
            1, get_pdb_url_disk(fname,
               PDB_RET_NAME,debug_data->x64));
        if (GET_PDB)
              xname = download_pdb_file( get_pdb_url_disk(
                 fname,PDB_RET_NAME,
                debug_data->x64), wpdb_url, DATADIRECTORY);
        win32k_pdb = parse_pdb_file(xname, PDB_DEFAULT);
        free(fname);
        free(wpdb_url);
        free(xname);

    }
    else {

        wpdb_url = malloc(sizeof(char) * (strlen(pdb_url) + 1));
        strcpy_s(wpdb_url, strlen(pdb_url) + 1, pdb_url);
        xname = malloc(sizeof(char) *
               (strlen(get_pdb_url(wimage_base,PDB_RET_NAME,
```

```
                debug_data->x64,debug_data->kDTB)) +
                strlen(DATADIRECTORY) + 1));
        strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
        strcat_s(xname,strlen(DATADIRECTORY) +
                    strlen(get_pdb_url(wimage_base,
                    PDB_RET_NAME,debug_data->x64,
                    debug_data->kDTB)) + 1,get_pdb_url(
                     wimage_base,PDB_RET_NAME,
                    debug_data->x64,debug_data->kDTB));
        if (GET_PDB)
              xname = download_pdb_file(get_pdb_url(
                     wimage_base,PDB_RET_NAME,
                     debug_data->x64,debug_data->kDTB),
                     wpdb_url, DATADIRECTORY);
        win32k_pdb = parse_pdb_file(xname, PDB_DEFAULT);
        free(wpdb_url);
        free(xname);

    }


}


// Second, find where the different sections start
pe_offset =  memgrabke_32_LE(wimage_base + 0x3c);
hdr_base = wimage_base + pe_offset + 0x18;
data_segment         = get_pdb_section(
        win32k_pdb,".data",5)->virtual_address;
rdata_segment         = get_pdb_section(
        win32k_pdb,".rdata",6)->virtual_address;

// First get the gphn symbol
wwptr = get_pdb_symbol(win32k_pdb,"gSharedInfo",11);
if (wwptr == 0)
  wwptr = get_pdb_symbol(win32k_pdb,"_gSharedInfo",12);

if (wwptr != NULL) {

    gSharedInfo_offset = wwptr->offset; // Unadjusted offset
    gSharedInfo_offset_ptr = get_pdb_symbol_loc(win32k_pdb,
             gSharedInfo_offset, data_segment); // Adjusted RVA

}
else {
```

```
    gSharedInfo_offset = 0;
    gSharedInfo_offset_ptr = 0;

}

gSharedInfo = wimage_base + (((uint64_t) gSharedInfo_offset_ptr)
            & 0x00000000ffffffff);

wWindowStation = show_proc_hand (sel_proc,"clipboard",NULL);

if (debug_data->x64) {

    wformat_table = memgrabp_64_LE(
              sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
              wWindowStation + 0x58);
    wformat_count = memgrabp_32_LE(
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
             wWindowStation + 0x60);

}
else {

    wformat_table = (uint64_t) memgrabp_32_LE(
               sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
               wWindowStation + 0x2c)  & 0x00000000ffffffff;
    wformat_count = memgrabp_32_LE(
               sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
               wWindowStation + 0x30);

}

gotone = 0;
for (i = 0; i < wformat_count; i++) {

    if (debug_data->x64)
    chk_format = wformat_table + i * 0x18;
    else
    chk_format = wformat_table + i * 0x10;
    wformat = memgrabp_32_LE(
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                     chk_format);
    if ( (wformat == 0x00d) || (wformat == 0x001) ) {
        // text or unicode
        if (debug_data->x64)
```

```
          whandle = memgrabp_64_LE(
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                wformat_table + i * 0x18 + 0x08);
        else
          whandle = (uint64_t) memgrabp_32_LE(
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                wformat_table + i * 0x10 + 0x04)
                & 0x00000000ffffffff;
        gotone = 1;
        break;

    }

    else
      printf("Kernel Session = %i      Other format = %i\n",
          sel_proc->proc_eprocess.SessionId,wformat);

}

if (gotone) {

    if (1) printf("Handle = %llx\n",whandle);
    low16 = whandle & 0x000000000000ffff;
    high16 = (whandle >> 0x10) & 0x000000000000ffff;
    if (debug_data->x64) {

        if (((debug_data->MajorOSVersion == 6) &&
              (debug_data->MinorOSVersion == 0)) ||
             (debug_data->MajorOSVersion == 5)){
            // Vista/ XP/ 2003
            memloc = memgrabp_64_LE(
                 sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                 gSharedInfo + 0x08) + (low16 * 3 * 0x08);

        }
        else {

            rec_size = memgrabp_64_LE(
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                gSharedInfo + 0x10);
            memloc = memgrabp_64_LE(
                 sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
                 gSharedInfo + 0x08) + (low16 * rec_size);

        }
```

```
}
else {

    if (((debug_data->MajorOSVersion == 6) &&
                (debug_data->MinorOSVersion == 0))
            || (debug_data->MajorOSVersion == 5)){
        // Vista/ XP
        memloc = (uint64_t) memgrabp_32_LE(
            sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
            gSharedInfo + 0x04) + (low16 * 3 * 0x04)
            & 0x00000000ffffffff;

    }

    else {

        rec_size = (uint64_t) memgrabp_32_LE(
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
             gSharedInfo + 0x08) & 0x00000000ffffffff;
        memloc = (uint64_t) memgrabp_32_LE(
             sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
             gSharedInfo + 0x04) + (low16 * rec_size)
             & 0x00000000ffffffff;

    }


}

if (debug_data->x64)
  memloc = memgrabp_64_LE(
       sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
       memloc) + 0x14;
else
  memloc = ( (uint64_t) memgrabp_32_LE(
       sel_proc->proc_eprocess.Pcb.DirectoryTableBase,
       memloc) & 0x00000000ffffffff) + 0x0c;
wclip_string = memgrabp_unicode_LE_0 (memloc,
       sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

if (strlen(wclip_string) > 0)
  printf("Kernel Session = %i      %s\n",
           sel_proc->proc_eprocess.SessionId,wclip_string);
```

```
                // Format = 0xd = Ascii
                free(wclip_string);

            }

            return 1;

        }

        free(wstring);
        // Iterate to the next driver
        if (debug_data->x64)
        wptr = memgrabke_64_LE(wptr + 0x00);
        else
        wptr = memgrabke_32_LE(wptr + 0x00);

    }

    return 0;

}



/*****************************************************************************/
/* Clear process, thread, user, socket lists  - Used for refresh */


void clear_process_list(process_list *old_process){


    process_list *tmp;

    if(old_process == NULL)
    return;


    if(MYDEBUG) printf("Moving to front of process list...\n");
    //moves to front of process list
    while(old_process->prev != NULL){

        old_process = old_process->prev;

    }
```

```
    if(MYDEBUG) printf("Removing processes from list...\n");
    //removes process from list until end of list is reached
    while(old_process != NULL){

        tmp = old_process->next;
        free(old_process->proc_eprocess.ImageFileName2);
        free(old_process);
        old_process = tmp;

    }

    old_process = NULL;

}


void clear_thread_list(thread_list *old_thread){


    thread_list *tmp;

    if(old_thread == NULL)
    return;

    while(old_thread->prev != NULL){

        old_thread = old_thread->prev;

    }


    //removes thread from list until end of list is reached
    while(old_thread != NULL){

        tmp = old_thread->next;
        free(old_thread);
        old_thread = tmp;

    }


}
```

```
void clear_user_list(user_list_type *old_user){


    user_list_type *tmp;

    if(old_user == NULL)
    return;

    //moves to front of user list
    while(old_user->prev != NULL){

        old_user = old_user->prev;

    }


    //removes user from list until end of list is reached
    while(old_user != NULL){

        tmp = old_user->next;
        free(old_user->Token);
        free(old_user->Path);
        free(old_user->Sid);
        free(old_user->Name);
        free(old_user);
        old_user = tmp;
        if(old_user != NULL)
        old_user->prev = NULL;

    }



}


void clear_socket_list(socket_list_type *old_socket){


    //moves to front of socket list
    while(old_socket->prev != NULL){

        old_socket = old_socket->prev;

    }
```

```
        //removes socket from list until end of list is reached
        while(old_socket != NULL){

            free(old_socket->LocalPort);
            free(old_socket->RemotePort);
            old_socket = old_socket->next;
            free(old_socket->prev);
            old_socket->prev = NULL;
            old_socket = old_socket->next;

        }


    }
```

A.2 cmat_registry.c

```
/*********************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)             *
* Developer  : Jimmy Okolica                                        *
* Date       : 15-Aug-2011                                          *
*                                                                   *
*********************************************************************/

/*****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/*****************************************************************************/

//#ifdef DEBUG
#define DEBUG_registry 0
//#endif
```

```
#ifndef CMAT_REGISTRY
#include "cmat_registry.h"
#endif

/********************/
/* GLOBAL VARIABLES */
/********************/

hive_list_type *hive_list = NULL;  // List of hives
short reg_x64 = 0;                   // True if 64-bit memory dump

/*****************************************************************************/

/*******************************/
/* SETTING OF GLOBAL VARIABLES */
/*******************************/

/************************************/
/* Set pointer to the list of hives */
/************************************/
void set_hive_list(hive_list_type *wtmp) {

    hive_list = wtmp;

}


/*********************************/
/* Set whether 64-bit is enabled */
/*********************************/
void setreg_x64(short new_value) {

    reg_x64 = new_value;

}


/******************************************************************************/

/****************************************/
/* FIND & RETRIEVE A SPECIFIC KEY VALUE */
/****************************************/

/*****************************/
/* Find a specific key value */
```

```
/****************************/
uint64_t get_cell_loc(hive_list_type *tmp, uint32_t keycell) {


    uint64_t hive_ptr;
    uint64_t *directories;

    pdb_array_attr_t wattr;

    // Disassemble the cell value into its constituents parts
    uint8_t SV = (uint8_t) ((keycell      & 0x80000000) >> 31);
    uint16_t DI = (uint16_t) ((keycell     & 0x7FE00000) >> 21);
    uint16_t TI = (uint16_t) ((keycell     & 0x001FF000) >> 12);
    uint16_t offset = (uint16_t) ((keycell & 0x00000FFF) );

    uint64_t cell_loc = 0;

    // First find the use the storage map to get the directory map
    /* Storage[i].Map is a ptr to the directory map */
    /* note: the first storage entry is the stable hive and the second storage
             entry is the volatile hive */
    hive_ptr = tmp->Hive.HHive.Storage[SV].Map;
    if (DEBUG_registry) printf("**get_cell_loc**  hive_ptr = %llx\n",hive_ptr);

    // Next use the directory to find the offset into the hive
    /* once you're at the hive directory, use DI as the offset to find the right
        hive_table */
    directories = keget_pdb_type_array( "_HMAP_DIRECTORY.Directory", hive_ptr);
    hive_ptr = directories[DI];

    // Finally get the location of the cell
    if (hive_ptr != 0) {

        wattr = get_pdb_type_array_attr ( "_HMAP_TABLE.Table");
        cell_loc = keget_pdb_type( "_HMAP_ENTRY.BlockAddress", hive_ptr +
                          wattr.base_offset
                        + wattr.size * TI) +
                          offset + 0x4; // 0x4 lets us skip over the size

    }

    free(directories);
    return cell_loc;

}
```

```
/**********************************************************************/
/* Search through a Registry folder and return the location of the match */
/**********************************************************************/
uint64_t get_key(hive_list_type *tmp, unsigned int sublist,
                  unsigned int num_subkeys, char *value)
{

    uint64_t mem_pointer, ret_pointer;
    uint32_t cell_offset;
    int ctr = 0;
    // BEGIN -- CMAT_V
    int compare;
    int cell_index = 0;
    int old_index;
    int L = 0;
    int R = num_subkeys;
    // END -- CMAT_V
    uint16_t wname_length;
    char * wname;
    wname = NULL;

    // Get the memory location of the sublist (parse the sublist cell code)
    mem_pointer = get_cell_loc(tmp,sublist);


    // BEGIN - CMAT-V
    cell_index = (R-L)/2;
    old_index = 0;
    //begin binary search

    if (DEBUG_registry) printf("Looking for %s...", value);

    // Scroll through each of the keys in the list
    while (cell_index >= 0) {


        cell_index = L + cell_index;
        if (DEBUG_registry) printf("jumping to %d\n", cell_index);

        //for (ctr = 0; ctr < num_subkeys; ctr++) {

            // END - CMAT-V
```

```
// Unfortunately, it does not appear that the structure of the lf
// record is in ntoskrnl_pdb
// it begins like CM_KEY_INDEX; however, the List array is composed
// of 2 4 byte fields -
// 0x00 offset to field, 0x04 first 4 bytes of name

// Find the cell location of the next key
//cell_offset = memgrabke_32_LE(mem_pointer + 0x4 + (0x8 * ctr));

cell_offset = memgrabke_32_LE(mem_pointer + 0x4 + (0x8 * cell_index));
if (cell_offset != 0) {

    // translate the cell location into an actual memory address
    ret_pointer = get_cell_loc(tmp,cell_offset);

    // get the name of the key
    wname_length = (uint16_t) (keget_pdb_type  ( "_CM_KEY_NODE.NameLength",
             ret_pointer) & 0x000000000000ffff);
    wname =  keget_pdb_type_stringn  ( "_CM_KEY_NODE.Name",
             ret_pointer, wname_length);

    // compare the key to the input value and if it's a match,
    // return it
    // BEGIN -- CMAT_V
    //            if (!strcmp(wname,value)) {

        //                if (DEBUG_registry) printf("...got it\n");
        //                return ret_pointer;
        //

    //            if (DEBUG_registry) printf("...nope\n");
    //


compare = strcmp(wname,value);
free(wname);
if(compare == 0) {

    if (DEBUG_registry) printf("...got it. cell_index = %d\n",cell_index);
    return ret_pointer;

}


else if(compare > 0){
```

```
                if (DEBUG_registry) printf(" too far down...");
                R = cell_index;

            }


            else{

                if (DEBUG_registry) printf(" further down...");
                L = cell_index;

            }


            if (DEBUG_registry) printf("...nope\n");
            // END -- CMAT_V

        }
        //endif  // CMAT-V
        // BEGIN -- CMAT_V
        cell_index = ((R-L)/2);
        if (old_index == cell_index) {

            if (ctr == 0) cell_index = 0; else cell_index = 1;
            ctr++;

        }

        if (ctr > 2) return 0;
        old_index = cell_index;
        // END -- CMAT_V


    }

    return 0;

}


/*********************************************/
/* Print out the location of all of the hives */
/*********************************************/
void print_hives() {
```

```
    hive_list_type * htmp;

    printf ("\n\nHive List:\n");
    htmp = hive_list;
    printf (" Hive#\t\t Name\n");
    while (htmp != NULL)        {

        printf ("%llx \t %s\n",htmp->hive_location,(htmp->name != NULL?htmp->name:" "));
        htmp = htmp->next;

    }


}


/*******************************************/
/* Add the hive to the linked list of hives */
/*******************************************/
hive_list_type * add_new_hive (hive_list_type * hive_head, uint64_t hive_location)
{

    hive_list_type *tmp;

    if ((tmp = malloc(sizeof(hive_list_type))) == NULL ) {

        (void)exit(EXIT_FAILURE);

    }

    tmp->hive_location = hive_location;
    tmp->next = hive_head;
    tmp->prev = NULL;
    tmp->name = malloc(sizeof(char));
    tmp->name[0] = 0;
    if (tmp->next != NULL) tmp->next->prev = tmp;

    return tmp;

}


/***********************/
/* Instantiate the hive */
```

```
/***********************/
uint8_t grab_hive (hive_list_type *tmp)
{

    uint32_t i;
    pdb_array_attr_t wattr;

    // the root location of the hive
    tmp->Hive.HHive.BaseBlockPtr = keget_pdb_type  ( "_CMHIVE.Hive.BaseBlock",
                tmp->hive_location);

    // Check if this is a false positive; if it is, get rid of it
    if (tmp->Hive.HHive.BaseBlockPtr == 0x0) {
        // False Positive
        if (tmp->next != NULL) tmp->next->prev = tmp->prev;
        if (tmp->prev != NULL) tmp->prev->next = tmp->next;
        free(tmp);
        return 0;

    }


    // Get the linked list keys
    tmp->Hive.HiveList.Blink     = keget_pdb_type  ( "_CMHIVE.HiveList.Blink",
                 tmp->hive_location);
    tmp->Hive.HiveList.Flink     = keget_pdb_type  ( "_CMHIVE.HiveList.Flink",
                 tmp->hive_location);

    // The location of the storage area and map that parses out the cell locations
    wattr = get_pdb_type_array_attr ( "_CMHIVE.Hive.Storage");
    for (i = 0; i < wattr.elements; i++)
    tmp->Hive.HHive.Storage[i].Map = keget_pdb_type  ( "_DUAL.Map",
                  tmp->hive_location + wattr.base_offset + wattr.size * i);

    // The location of the root cell in the hive
    tmp->Hive.HHive.BaseBlock.RootCell = (uint32_t) keget_pdb_type  (
                          "_HBASE_BLOCK.RootCell",
                          tmp->Hive.HHive.BaseBlockPtr);


    return 1;

}
```

```
/*******************************************/
/* Extract the users from the SOFTWARE hive */
/*******************************************/
user_list_type * get_users()
{

    user_list_type *user_list;
    hive_list_type *tmp;
    user_list_type *wuser;
    ascii_string wname;
    unicode_string uni_name;
    uint16_t wname_length;
    uint32_t ctr, ctr2,ictr;
    char *keyname, *pname;

    uint32_t cell_offset, *num_subkeys, *slist_cell, vlist_cell, num_vkeys;
    uint64_t mem_pointer, profile_pointer, val_pointer, wpointer, keyvalue_pointer;

    printf ("\nFinding users in memory dump\n");

    tmp = hive_list;
    user_list = NULL;

    // First find the SOFTWARE hive
    while ((tmp != NULL) && (strcmp(tmp->name,"REGISTRY/MACHINE/SOFTWARE")))
          tmp = tmp->next;
    if (tmp != NULL) {


        // Find the root cell in the SOFTWARE hive
        mem_pointer = get_cell_loc(tmp,tmp->Hive.HHive.BaseBlock.RootCell);

        // Find the "Microsoft" key in the SOFTWARE hive
        num_subkeys = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyCounts",
                      mem_pointer);
        slist_cell = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyLists",
                      mem_pointer);
        mem_pointer = get_key(tmp, slist_cell[0], num_subkeys[0], "Microsoft");
        free(num_subkeys);
        free(slist_cell);

        // Find the "Windows NT" key in the SOFTWARE/Microsoft hive
        num_subkeys = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyCounts",
                     mem_pointer);
        slist_cell = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyLists",
```

```
            mem_pointer);
mem_pointer = get_key(tmp, slist_cell[0], num_subkeys[0], "Windows NT");
free(num_subkeys);
free(slist_cell);

// Find the "CurrentVersion" key in the SOFTWARE/Microsoft/Windows NT hive
num_subkeys = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyCounts",
        mem_pointer);
slist_cell = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyLists",
        mem_pointer);
mem_pointer = get_key(tmp, slist_cell[0], num_subkeys[0], "CurrentVersion");
free(num_subkeys);
free(slist_cell);

// Find the "ProfileList" key in the SOFTWARE/Microsoft/Windows
// NT/CurrentVersion hive
num_subkeys = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyCounts",
        mem_pointer);
slist_cell = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyLists",
        mem_pointer);
mem_pointer = get_key(tmp, slist_cell[0], num_subkeys[0], "ProfileList");
free(num_subkeys);
free(slist_cell);

// Now parse through the entries in SOFTWARE/Microsoft/Windows
// NT/CurrentVersion/ProfileList --
// these are the users with accounts on the box
num_subkeys = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyCounts",
     mem_pointer);
slist_cell = keget_pdb_type_uint32array( "_CM_KEY_NODE.SubKeyLists",
     mem_pointer);
mem_pointer = get_cell_loc(tmp,slist_cell[0]);
for (ctr = 0; ctr < num_subkeys[0]; ctr++) {

    if ((wuser = malloc(sizeof(user_list_type))) == NULL ) {

        (void)exit(EXIT_FAILURE);

    }

    // Unfortunately, it does not appear that the structure of the lf
    // record is in ntoskrnl_pdb
    // it begins like CM_KEY_INDEX; however, the List array is composed
    // of 2 4 byte fields -
    // 0x00 offset to field, 0x04 first 4 bytes of name
```

```
// Get the cell location
cell_offset = memgrabke_32_LE(mem_pointer + 0x4 + (0x8 * ctr));
// Translate into a memory address
profile_pointer = get_cell_loc(tmp,cell_offset);
// Go to that spot and extract the token id for this profile
wname_length = keget_pdb_type_uint16  ( "_CM_KEY_NODE.NameLength",
     profile_pointer);
wuser->Token =  keget_pdb_type_stringn  ( "_CM_KEY_NODE.Name",
     profile_pointer, wname_length);
wuser->Sid = malloc( sizeof(char));
wuser->Sid[0] = 0;
wuser->Name = malloc( sizeof(char));
wuser->Name[0] = 0;
wuser->Path = malloc( sizeof(char));
wuser->Path[0] = 0;
wuser->SidSize = 0;
if (DEBUG_registry) printf("**get_users** Token: %s\n",wuser->Token);

// Each token is a folder with entries underneath it;
// go into that folder and extract the path name for that token
vlist_cell = (uint32_t) keget_pdb_type  (
       "_CM_KEY_NODE.ValueList.List", profile_pointer);
num_vkeys = (uint32_t) keget_pdb_type  (
       "_CM_KEY_NODE.ValueList.Count", profile_pointer);
val_pointer = get_cell_loc(tmp,vlist_cell);
// For each entry in SOFTWARE/Microsoft/Windows
// NT/CurrentVersion/ProfileList/**token**
for (ctr2 = 0; ctr2 < num_vkeys; ctr2++) {

    cell_offset = memgrabke_32_LE(val_pointer + (0x4 * ctr2));
    keyvalue_pointer = get_cell_loc(tmp,cell_offset);
    // Get the name of the entry
    wname_length = (uint16_t) keget_pdb_type  (
                  "_CM_KEY_VALUE.NameLength",
                 keyvalue_pointer);
    keyname = keget_pdb_type_stringn  ( "_CM_KEY_VALUE.Name",
                 keyvalue_pointer, wname_length);
    // If it's the SID, extract the value
    if (!strcmp("Sid",keyname)) {
        cell_offset = (uint32_t) keget_pdb_type  (
               "_CM_KEY_VALUE.Data", keyvalue_pointer );
        wpointer = get_cell_loc(tmp,cell_offset);
        wuser->SidSize = (uint32_t) keget_pdb_type  (
               "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
        wname.Length = (uint16_t) wuser->SidSize;
```

```
            wname.MaximumLength = wname.Length;
            wname.Buffer = wpointer;
            wuser->Sid = memgrabke_string_LE (wname);


        }

        // If it's the ProfileImagePath, extract it; then store both the
        // path name and the
        // (hopefully) user name (which should be the last entry in the
        // path)
        if (!strcmp("ProfileImagePath",keyname)) {

            cell_offset = (uint32_t) keget_pdb_type  ( "_CM_KEY_VALUE.Data",
                       keyvalue_pointer);
            wpointer = get_cell_loc(tmp, cell_offset);
            uni_name.Length = (uint16_t) keget_pdb_type  (
                "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
            uni_name.MaximumLength = uni_name.Length;
            uni_name.Buffer = wpointer;
            wuser->Path = memgrabke_unicode_LE (uni_name);
            pname = strrchr(wuser->Path,'\\');
            if (pname != NULL) {

                pname++;
                wuser->Name = malloc(sizeof(char) *
                        (strlen(wuser->Path) - (pname - wuser->Path) + 2));
                strcpy_s(wuser->Name,strlen(wuser->Path) - (pname -
                         wuser->Path) + 2,pname);

            }


        }

        free(keyname);

    }

    wuser->next = user_list;
    wuser->prev = NULL;
    user_list = wuser;
    if (wuser->next != NULL) wuser->next->prev = wuser;

}
```

```
        free(num_subkeys);
        free(slist_cell);

    }

    return user_list;


}


/*********************************************************************************/
/* Extract a value from an entry in an arbitrary registry key in an arbitrary hive */
/*********************************************************************************/
char * get_values(char * whive_name, char * wkey, char * wentry)
{

    hive_list_type *tmp;
    char *wwkey, *wsep, *keyname, *wret_string, *valname;
    unicode_string uni_name;
    uint16_t ctr,wname_length, wlength;

    uint32_t cell_offset, *num_subkeys, *slist_cell, vlist_cell, num_vkeys, wtype;
    uint64_t mem_pointer, val_pointer, wpointer, wpointer2, keyvalue_pointer;

    int i;
    uint8_t wval8;
    uint32_t wval32;
    uint64_t wval64;
    char * wtypename;

    tmp = hive_list;
    wwkey = NULL;

    // First find the requested hive (e.g., //REGISTRY/.MACHINE/SYSTEM)
    while ((tmp != NULL) && (strcmp(tmp->name,whive_name))) tmp = tmp->next;
    if (tmp != NULL) {

        // Get the value of the base block
        mem_pointer = get_cell_loc(tmp,tmp->Hive.HHive.BaseBlock.RootCell);
        while ( (wkey != NULL) && (strlen(wkey) != 0)) {

            // get the next element of the requested key
            wsep = strchr(wkey,'/');
```

```
    if (wsep != NULL) {

        if (wwkey != NULL) free(wwkey);
        wlength = wsep - wkey;
        wwkey = malloc(sizeof(char) * (wlength + 1));
        strncpy_s(wwkey,wlength + 1,wkey,wlength);
        wwkey[wlength] = 0;
        wkey = wsep + 1;

    }

    else {

        wwkey = wkey;
        wkey = NULL;

    }

    // Search through the sublist for the requested key
    num_subkeys = (uint32_t *) keget_pdb_type_array(
           "_CM_KEY_NODE.SubKeyCounts", mem_pointer);
    slist_cell = (uint32_t *) keget_pdb_type_array(
           "_CM_KEY_NODE.SubKeyLists", mem_pointer);
    mem_pointer = get_key(tmp, slist_cell[0], num_subkeys[0], wwkey);
    free(num_subkeys);
    free(slist_cell);
    free(wwkey);

}



// At the bottom, you've got values or another sublist (or both)

// First we'll look at the sublist
if (strlen(wentry) == 0) {

    num_subkeys = (uint32_t *) keget_pdb_type_array(
        "_CM_KEY_NODE.SubKeyCounts", mem_pointer);
    slist_cell = (uint32_t *) keget_pdb_type_array(
        "_CM_KEY_NODE.SubKeyLists", mem_pointer);
    wpointer = get_cell_loc(tmp,slist_cell[0]);
    for (ctr = 0; ctr < num_subkeys[0]; ctr++) {

        cell_offset = memgrabke_32_LE(wpointer + 0x4 + (0x8 * ctr));
```

```
        if (cell_offset != 0) {

            wpointer2 = get_cell_loc(tmp,cell_offset);
            wname_length = (uint16_t) (keget_pdb_type  (
                           "_CM_KEY_NODE.NameLength",
                          wpointer2) & 0x000000000000ffff);
            keyname =  keget_pdb_type_stringn  ( "_CM_KEY_NODE.Name",
                           wpointer2, wname_length);
            printf("    Sublist        %s\n",keyname);
            free(keyname);

        }


    }

    free(num_subkeys);
    free(slist_cell);

}


// Now, let's look at the values
vlist_cell = (uint32_t) keget_pdb_type  ( "_CM_KEY_NODE.ValueList.List",
         mem_pointer);
num_vkeys = (uint32_t) keget_pdb_type  ( "_CM_KEY_NODE.ValueList.Count",
         mem_pointer);
val_pointer = get_cell_loc(tmp,vlist_cell);

// Now, parse through the values in the requested keys folder until you
// find the entry of interest
for (ctr = 0; ctr < num_vkeys; ctr++) {

    cell_offset = memgrabke_32_LE(val_pointer + (0x4 * ctr));
    keyvalue_pointer = get_cell_loc(tmp,cell_offset);
    wname_length = (uint16_t) keget_pdb_type  (
             "_CM_KEY_VALUE.NameLength", keyvalue_pointer);
    keyname = keget_pdb_type_stringn  ( "_CM_KEY_VALUE.Name",
             keyvalue_pointer, wname_length);
    // Once you get a match, return the value
    if (strlen(wentry) > 0) {

        if (!strcmp(keyname,wentry)) {

            cell_offset = (uint32_t) keget_pdb_type  (
```

```
                "_CM_KEY_VALUE.Data", keyvalue_pointer);
            wpointer = get_cell_loc(tmp,cell_offset);
            uni_name.Length = (uint16_t) keget_pdb_type  (
                "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
            uni_name.MaximumLength = (uint16_t) wname_length;
            uni_name.Buffer = wpointer;
            wret_string = malloc(sizeof(char) * (uni_name.Length + 1));
            wret_string = memgrabke_unicode_LE (uni_name);
            free(keyname);
            return wret_string;

        }


    }

    else {

        wtype = (uint32_t) keget_pdb_type(
               "_CM_KEY_VALUE.Type",keyvalue_pointer);
        cell_offset = (uint32_t) keget_pdb_type  ( "_CM_KEY_VALUE.Data",
               keyvalue_pointer);
        wlength = (uint16_t) keget_pdb_type  (
               "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
        wpointer = get_cell_loc(tmp,cell_offset);
        switch (wtype) {

            case 0:
            wtypename = "REG_NONE"; // Unstructured (probably binary) data
            printf("    Value        %s        %s        ",keyname,wtypename);
            if (wlength <= 4) wpointer = keyvalue_pointer +
                 get_pdb_type_offset( "_CM_KEY_VALUE.Data");
            for (i = 0; i < wlength; i++) {

                wval8 = memgrabke_8_LE(wpointer + i * 1);
                printf("%2x ",wval8);

            }

            printf("\n");
            break;
            case 1:
            wtypename = "REG_SZ"; // fixed length string
            uni_name.Length = (uint16_t) keget_pdb_type  (
                        "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
```

```
uni_name.MaximumLength = (uint16_t) wname_length;
uni_name.Buffer = wpointer;
valname = malloc(sizeof(char) * (uni_name.Length + 1));
valname = memgrabke_unicode_LE (uni_name);
printf("    Value        %s        %s        %s \n",
           keyname,wtypename,valname);
break;
case 2:
wtypename = "REG_EXPAND_SZ"; // variable length string
uni_name.Length = (uint16_t) keget_pdb_type  (
            "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
uni_name.MaximumLength = (uint16_t) wname_length;
uni_name.Buffer = wpointer;
valname = malloc(sizeof(char) * (uni_name.Length + 1));
valname = memgrabke_unicode_LE (uni_name);
printf("    Value        %s        %s        %s\n",
            keyname,wtypename,valname);
break;
case 3:
wtypename = "REG_BINARY"; // binary data
printf("    Value        %s        %s        ",
             keyname,wtypename);
if (wlength <= 4) wpointer = keyvalue_pointer +
       get_pdb_type_offset( "_CM_KEY_VALUE.Data");
for (i = 0; i < wlength; i++) {

    wval8 = memgrabke_8_LE(wpointer + i * 1);
    printf("%2x ",wval8);

}

printf("\n");
break;
case 4:
wtypename = "REG_DWORD"; // 32-bit Little Endian
printf("    Value        %s        %s        ",
            keyname,wtypename);
if (wlength <= 4) wpointer = keyvalue_pointer +
      get_pdb_type_offset( "_CM_KEY_VALUE.Data");
for (i = 0; i < wlength; i += 4) {

    wval32 = memgrabke_32_LE(wpointer + i * 4);
    printf("%2x ",wval32);

}
```

```
printf("\n");
break;
case 5:
wtypename = "REG_DWORD_BIG_ENDIAN"; //32-bit Big Endian
printf("    Value       %s       %s       ",
             keyname,wtypename);
if (wlength <= 4) wpointer = keyvalue_pointer +
             get_pdb_type_offset( "_CM_KEY_VALUE.Data");
for (i = 0; i < wlength; i += 4) {

    wval32 = memgrabke_32_BE(wpointer + i * 4);
    printf("%2x ",wval32);

}

printf("\n");
break;
case 6:
wtypename = "REG_LINK"; // Symbolic link, stored as a string
uni_name.Length = (uint16_t) keget_pdb_type  (
           "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
uni_name.MaximumLength = (uint16_t) wname_length;
uni_name.Buffer = wpointer;
valname = malloc(sizeof(char) * (uni_name.Length + 1));
valname = memgrabke_unicode_LE (uni_name);
printf("    Value       %s       %s       %s \n",
     keyname,wtypename,valname);
break;
case 7:
wtypename = "REG_MULTI_SZ"; // Multiple strings, each
    // null-terminated,
    // list is null-terminated after last string null-termination
uni_name.Length = (uint16_t) keget_pdb_type  (
    "_CM_KEY_VALUE.DataLength", keyvalue_pointer);
uni_name.MaximumLength = (uint16_t) wname_length;
uni_name.Buffer = wpointer;
valname = malloc(sizeof(char) * (uni_name.Length + 1));
valname = memgrabke_unicode_LE (uni_name);
for (i = 0; i < (uni_name.Length / 2); i++)
  if (valname[i] == 0) valname[i] = 0x20;
printf("    Value       %s       %s       %s\n",
      keyname,wtypename,valname);
break;
case 8:
```

```
wtypename = "REG_RESOURCE_LIST"; // Series of nested arrays
                                 // to store a resource
                                 // used by a device driver
printf("    Value        %s        %s        ",
      keyname,wtypename);
if (wlength <= 4) wpointer = keyvalue_pointer +
       get_pdb_type_offset( "_CM_KEY_VALUE.Data");
for (i = 0; i < wlength; i++) {

    wval8 = memgrabke_8_LE(wpointer + i * 1);
    printf("%2x ",wval8);

}

printf("\n");
break;
case 9:
wtypename = "REG_FULL_RESOURCE_DESCRIPTOR";
           // Series of nested arrays
           // to store a resource used
           // by a physical hardware device
printf("    Value        %s        %s        ",
         keyname,wtypename);
if (wlength <= 4) wpointer = keyvalue_pointer +
     get_pdb_type_offset( "_CM_KEY_VALUE.Data");
for (i = 0; i < wlength; i++) {

    wval8 = memgrabke_8_LE(wpointer + i * 1);
    printf("%2x ",wval8);

}

printf("\n");
break;
case 10:
wtypename = "REG_RESOURCE_REQUIREMENTS_LIST";
              // Series of nested arrays to store a
              // resource used by a device driver to
              // store a list of possible hardware resources
printf("    Value        %s        %s        ",
      keyname,wtypename);
if (wlength <= 4) wpointer = keyvalue_pointer +
          get_pdb_type_offset( "_CM_KEY_VALUE.Data");
for (i = 0; i < wlength; i++) {
```

```
                        wval8 = memgrabke_8_LE(wpointer + i * 1);
                        printf("%2x ",wval8);

                    }

                    printf("\n");
                    break;
                    case 11:
                    wtypename = "REG_QWORD"; // 64-bit little endian
                    printf("    Value        %s        %s        ",
                           keyname,wtypename);
                    if (wlength <= 4) wpointer = keyvalue_pointer +
                          get_pdb_type_offset( "_CM_KEY_VALUE.Data");
                    for (i = 0; i < wlength; i += 8) {

                        wval64 = memgrabke_64_LE(wpointer + i * 8);
                        printf("%2I64x ",wval64);

                    }

                    printf("\n");
                    break;
                    default:
                    printf("    Value        %s        Unknown Type: %x\n",
                              keyname,wtype);

                }

                free(keyname);

            }


        }


    }


    wret_string = malloc(1);
    wret_string[0] = 0;
    return wret_string;

}
```

```
/******************************************/
/* Extract information about a registry key */
/******************************************/
key_object * get_key_object (process_list *sel_proc, uint64_t addr)
{

    uint64_t mem_pointer = addr;
    uint32_t wparent;
    uint32_t wsignature, wflags;
    char  *basename, *extension;
    uint16_t wname_length;
    hive_list_type *tmp;

    hive_list_type *hive_ptr = NULL;

    key_object *wkey = malloc(sizeof(key_object));
    wkey->cell.name = (char *) malloc(2 * sizeof(char));
    wkey->cell.name[0] = 0;
    wkey->cell.name[1] = 0;
    wkey->hive_name = (char *) malloc(2 * sizeof(char));
    wkey->hive_name[0] = 0;
    wkey->hive_name[1] = 0;



    // retrieve the pointer to the registry key -- the size of the key will vary
    // for 32 versus 64 bit
    if (reg_x64) {

        wkey->unknown1 = memgrabp_64_LE(
                   sel_proc->proc_eprocess.Pcb.DirectoryTableBase,mem_pointer);
        mem_pointer +=8;
        wkey->key_pointer = memgrabp_64_LE(
                   sel_proc->proc_eprocess.Pcb.DirectoryTableBase,mem_pointer);

    }

    else {

        wkey->unknown1 = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,mem_pointer);
        mem_pointer +=4;
        wkey->key_pointer = memgrabp_32_LE(
                  sel_proc->proc_eprocess.Pcb.DirectoryTableBase,mem_pointer);
```

```
}

mem_pointer = wkey->key_pointer;


// Store the hive this key comes from and the cell location
wkey->KeyHive = pget_pdb_type( "_CM_KEY_CONTROL_BLOCK.KeyHive", mem_pointer,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
wkey->KeyCell = (uint32_t) pget_pdb_type( "_CM_KEY_CONTROL_BLOCK.KeyCell",
                mem_pointer,
                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);

// Get the hive of the interest
hive_ptr = hive_list;
while (hive_ptr != NULL) {

    if (hive_ptr->hive_location == wkey->KeyHive) break;
    hive_ptr = hive_ptr->next;

}

if (hive_ptr == NULL) return wkey;

// Translate the cell location into a memory address
wkey->cell.address = get_cell_loc(hive_ptr,wkey->KeyCell);


// just check to make sure cell.address isn't zero starting
wparent = (uint32_t) wkey->cell.address;
mem_pointer = wkey->cell.address;
while (wparent != 0x0) {

    wsignature = (uint32_t) pget_pdb_type  ( "_CM_KEY_NODE.Signature",
                    mem_pointer,
                   sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    wflags = (uint32_t) pget_pdb_type  ( "_CM_KEY_NODE.Flags", mem_pointer,
                   sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
    if ((wsignature == 0x6b6e) && (wflags != 0x2c)) {
        //nk    _CM_KEY_NODE  0x2c=root hive
        wname_length = (uint16_t) pget_pdb_type  (
                    "_CM_KEY_NODE.NameLength", mem_pointer,
                   sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        extension =  pget_pdb_type_stringn  ( "_CM_KEY_NODE.Name", mem_pointer,
                    sel_proc->proc_eprocess.Pcb.DirectoryTableBase, wname_length);
```

```
        basename = wkey->cell.name;
        wkey->cell.name = (char *) malloc( (strlen(basename) +
                            strlen(extension) + 2) * sizeof(char));
        strcpy_s(wkey->cell.name, strlen(basename) + strlen(extension) + 2,
               extension);
        strcat_s(wkey->cell.name, strlen(basename) + strlen(extension) + 2,
               "/");
        strcat_s(wkey->cell.name, strlen(basename) + strlen(extension) + 2,
               basename);
        free(extension);

    }

    if (wflags == 0x2c) {

        wparent = 0x0;

    }

    else {

        wparent = (uint32_t) pget_pdb_type  ( "_CM_KEY_NODE.Parent", mem_pointer,
                                sel_proc->proc_eprocess.Pcb.DirectoryTableBase);
        if (wparent != 0x0)    mem_pointer = get_cell_loc(hive_ptr,wparent);

    }


}


tmp = hive_list;
while (tmp != NULL) {

    if (tmp->hive_location == wkey->KeyHive) {

        wkey->hive_name = malloc(sizeof(char) * (strlen(tmp->name) + 1));
        strcpy_s(wkey->hive_name,strlen(tmp->name) + 1,tmp->name);
        break;

    }

    tmp = tmp->next;

}
```

```
    return wkey;

}
```

A.3 file_read.c

```
/****************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)         *
* Developer  : Jimmy Okolica                                    *
* Date       : 15-Aug-2011                                      *
*                                                               *
****************************************************************/

/*****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/*****************************************************************************/

#ifndef FILE_READ
#include "file_read.h"
#endif

/********************/
/* GLOBAL VARIABLES */
/********************/
short DEBUGIT = 0;                         // DEBUG variable

short NO_PAE = 0;                         // PAE disabled/enabled
short x64 = 0;                            // 64-bit enabled/disabled
uint64_t kDTB = 0;                        // the Page Directory Table base for
                                          // the Kernel
FILE *image_file;                         // Handle to the memory dump file

// BEGIN -- CMAT_V
```

```
unsigned char *memory = NULL;
uint32_t memory_index = 0x0;
uint32_t prev_memory_index = 0x0;
uint32_t offset = 0;

xa_instance_t *local_instance;
// END -- CMAT-V

pagefile_list *pagefiles = NULL;        // Pointer to list of page files
missingfile_list *missingfiles = NULL;
largefile_list *largefiles = NULL;      // Pointer to temporary files used for
                                        // a memory dump greater than 2K
ptov_list *ptovs = NULL;
dtb_list *ptov_dtb = NULL;

/******************************************************************************/

/**************************/
/* MISCELLANEOUS UTILITIES */
/**************************/

// BEGIN -- CMAT_V
void init_instance(uint32_t domain_num, xa_instance_t *instance){


    local_instance = instance;

    /* initialize the xen access library */
    if (xa_init_vm_id_cmat(domain_num, local_instance, XA_FAILSOFT) == XA_FAILURE){

        perror("failed to init XenAccess library");
        exit(1);

    }



}

// END -- CMAT_V

/**********************************************/
/* Sets the flags for a 32-bit Page Table Entry */
/**********************************************/
void map_pte(uint32_t ptetemp, hardware_pte_x86 *pte)
```

```
{

    pte->Valid = (uint8_t) (ptetemp & VALID);
    pte->Write = (uint8_t) ((ptetemp & WRITE) >> 1);
    pte->Owner = (uint8_t) ((ptetemp & OWNER) >> 2);
    pte->WriteThrough = (uint8_t) ((ptetemp & WRITETHROUGH) >> 3);
    pte->CacheDisable = (uint8_t) ((ptetemp & CACHEDISABLE) >> 4);
    pte->Accessed = (uint8_t) ((ptetemp & ACCESSED) >> 5);
    pte->Dirty = (uint8_t) ((ptetemp & DIRTY) >> 6);
    pte->LargePage = (uint8_t) ((ptetemp & LARGEPAGE) >> 7);
    pte->Global = (uint8_t) ((ptetemp & GLOBAL) >> 8);
    pte->CopyOnWrite = (uint8_t) ((ptetemp & COPYONWRITE) >> 9);
    pte->Prototype = (uint8_t) ((ptetemp & PROTOTYPE) >> 10);
    pte->Transition = (uint8_t) ((ptetemp & RESERVED)>> 11);
    pte->PageFrameNumber = (ptetemp & PAGEFRAMENUMBER) & 0xFFFFF000;

}


/************************************************/
/* Sets the flags for a 64-bit Page Table Entry */
/************************************************/
void map_pte64(uint32_t ptetemp_high, uint32_t ptetemp_low, hardware_pte_x86 *pte)
{

    pte->Valid = (uint8_t) (ptetemp_low & VALID);
    pte->Write = (uint8_t) ((ptetemp_low & WRITE) >> 1);
    pte->Owner = (uint8_t) ((ptetemp_low & OWNER) >> 2);
    pte->WriteThrough = (uint8_t) ((ptetemp_low & WRITETHROUGH) >> 3);
    pte->CacheDisable = (uint8_t) ((ptetemp_low & CACHEDISABLE) >> 4);
    pte->Accessed = (uint8_t) ((ptetemp_low & ACCESSED) >> 5);
    pte->Dirty = (uint8_t) ((ptetemp_low & DIRTY) >> 6);
    pte->LargePage = (uint8_t) ((ptetemp_low & LARGEPAGE) >> 7);
    pte->Global = (uint8_t) ((ptetemp_low & GLOBAL) >> 8);
    pte->CopyOnWrite = (uint8_t) ((ptetemp_low & COPYONWRITE) >> 9);
    pte->Prototype = (uint8_t) ((ptetemp_low & PROTOTYPE) >> 10);
    pte->Transition = (uint8_t) ((ptetemp_low & RESERVED)>> 11);
    pte->PageFrameNumber64_high = (ptetemp_high & 0x0000FFFF);
    pte->PageFrameNumber64_low = (ptetemp_low   & 0xFFFFF000);

}


/*****************************************************************/
/* Sets whether this memory dump has PAE enabled/disabled - Public */
```

```
/********************************************************************/
void set_nopae(short new_value) {

    NO_PAE = new_value;

}


/*************************************************************/
/* Sets whether this memory dump is 32bit or 64bit - Public */
/*************************************************************/
void set_x64(short new_value) {

    x64 = new_value;

}


/***************************************************************/
/* Sets where the Kernel's Page Directory Table Base - Public */
/***************************************************************/
void set_kDTB(uint64_t wDTB) {

    kDTB = wDTB;

}


/****************************************************/
/* Break a memory dump into files <= 2G -- Private */ //
/****************************************************/
int mapfile(FILE * wfile) {

    // file_seg is in file_read.i and is the size of each temporary file...
    // currently it is 1G
    uint8_t * buffer;
    uint16_t wcount;
    uint16_t buf_size = 0x1000;  // Copying 4K pages to the temp files

    int x = 0;

    largefile_list *tmp; // This structure is for the memory dump, page files,
                         // and memory mapped files
    uint64_t mem_bb = 0;
    uint64_t wfile_seg = (((uint64_t) file_seg) & 0x00000000ffffffff);
```

```
// Create a new entry in largefiles -- every memory dump will have at least
//                                       one entry in largefile
buffer = malloc(0x1000 * sizeof(uint8_t));
tmp = malloc(sizeof(largefile_list));




tmp->next = largefiles;
largefiles = tmp;
tmp->file_ptr = wfile;
tmp->mem_2g = 0;
tmp->mem_ptrs[0] = tmp->file_ptr;

// Go to the spot in the memory dump file where you want to make the first break
fseek(tmp->file_ptr,file_seg - 1,SEEK_SET);
fgetc(tmp->file_ptr);
// If you're at the end (i.e. no breaks necessary), you're done -- leave
if (feof(tmp->file_ptr)) return 0;

printf("Segmenting Memory Dump...");
// mem_bb = low 32bits of the file pointer's location in the memory dump --
// starting out this should be 0
mem_bb = ((uint64_t) ftell(tmp->file_ptr)) & 0x00000000ffffffff;
while (!feof(tmp->file_ptr)) {
    // While there's still more to read
    // If we've reached the next break point, create new temp file for next
    // slice and keep going
    if (mem_bb % wfile_seg == 0) {

        printf("X...");
        tmp->mem_2g++;  // increment # of tempfiles
        tmpfile_s(&tmp->mem_ptrs[tmp->mem_2g] ); // every 2G, create a new
                                                 // temporary file;
        if (tmp->mem_ptrs[tmp->mem_2g] != 0) x = 0; else x = 1;
        fseek(tmp->mem_ptrs[tmp->mem_2g],0,SEEK_SET); // go to the beginning
                                                 // of the new temp file

    }

    // Read and write the next bufsize to the temporary file
    wcount = fread(buffer,1,buf_size,tmp->file_ptr);
    fwrite(buffer,1,wcount,tmp->mem_ptrs[tmp->mem_2g]);
    mem_bb += wcount;
```

```
    }

    printf("\n");
    return 0;

}


/*********************************/
/* Open the memory file -- Public */
/*********************************/
int open_image_file( char *filename) {

    char *wbyte = malloc(sizeof(char));

    if ( fopen_s(&image_file, filename, "rb") ) {

        printf ("The image file %s can't be opened. Sorry.\n", filename);
        return 1;

    }

    else {

        return mapfile(image_file);

    }


}


/*******************************/
/* Open the swap file -- Public */
/*******************************/
int open_page_file( char *filename) {

    pagefile_list *wpagefile;
    wpagefile = malloc(sizeof(pagefile_list));
    wpagefile->next = pagefiles;
    pagefiles = wpagefile;
    if (fopen_s(&wpagefile->file_ptr, filename, "rb" ) ) {

        printf ("The page file %s can't be opened. Sorry.\n", filename);
```

```
        return 1;

    }

    else {

        wpagefile->name = malloc(sizeof(char) * (strlen(filename) + 1));
        strcpy_s(wpagefile->name,strlen(filename) + 1,filename);
        return mapfile(wpagefile->file_ptr);

    }


}


/*************************************/
/* Open memory-mapped files -- Public */
/*************************************/
FILE * open_map_file(char *filename) {

    missingfile_list *tmp; // Keep track of memory mapped files that we don't
                           // have the files for

    if (strlen(filename) == 0) return NULL;
    tmp = missingfiles;
    while (tmp != NULL) {

        if (!strcmp(tmp->name,filename))
        return tmp->file_ptr;
        else
        tmp = tmp->next;

    }

    tmp = malloc(sizeof(missingfile_list));
    tmp->next = missingfiles;
    missingfiles = tmp;
    tmp->name = malloc(sizeof(char) * (strlen(filename) + 1));
    strcpy_s(tmp->name,strlen(filename) + 1,filename);
    fopen_s(&tmp->file_ptr, filename, "rb" );
    if (tmp->file_ptr == NULL ) {

        printf ("Missing mapped file %s %s\n", filename,tmp->name);
        return NULL;
```

```
    }

    else {

        mapfile(tmp->file_ptr);
        return tmp->file_ptr;

    }


}


/**********************************************************************/
/* Retrieve page file needed (note: Windows can have multiple swap files) */
/**********************************************************************/
FILE * get_pagefile(int pagefile_nbr) {

    pagefile_list * wtmp;
    wtmp = pagefiles;

    while ( (pagefile_nbr-- > 0) && (wtmp != NULL)) wtmp = wtmp->next;
    if (wtmp == NULL) // this should never happen;
    return NULL;
    else
    return wtmp->file_ptr;

}


/***************************************************************************/

/*******************************************/
/* READING VIRTUAL MEMORY plus VTOP utility */
/*******************************************/

/****************************************************************/
/* Wrapper for reading a byte from a virtual address -- Private */
/****************************************************************/
unsigned char *read_proc_mem (uint64_t DTB, uint64_t start_address,
                              unsigned int length, uint8_t wis_proto)
{

    return read_proc_mem_helper(DTB,start_address,length,wis_proto,GET_DATA);
```

```
}


/***********************************************************************/
/* Convert a virtual address to a physical address for a process -- Public */
/***********************************************************************/
uint64_t vtop (uint64_t DTB, uint64_t start_address)
{

    int i;
    unsigned char *result;
    uint64_t ret_val = 0;
    result = read_proc_mem_helper(DTB,start_address,9,NO_PROTO,GET_VTOP);
    for (i = 0; i < 8; i++) {

        ret_val = (ret_val << 8) + result[i];

    }

    if (result[8] != 0) {
        free(result); return 0;
    }
    // Pagefile or Mapfile
    free(result);
    return ret_val; // Later we may want to return if it's a pagefile or mapfile
                    // (stored in the ninth byte)

}


/************************************************************************/
/* Convert a virtual address to a physical address for kernel process- Public */
/************************************************************************/
uint64_t kevtop (uint64_t start_address)
{

    int i;
    unsigned char *result;
    uint64_t ret_val = 0;
    result = read_proc_mem_helper(kDTB,start_address,9,NO_PROTO,GET_VTOP);
    for (i = 0; i < 8; i++) {

        ret_val = (ret_val << 8) + result[i];
```

```
    }

    if (result[8] != 0) {
        free(result); return 0;
    }
    // Pagefile or Mapfile
    free(result);
    return ret_val; // Later we may want to return if it's a pagefile or mapfile
                    // (stored in the ninth byte)

}


/**************************************************************/
/* Convert a physical address to a virtual address -- Public */
/**************************************************************/
void ptov_init ()
{

    uint64_t PDM, PDP, PDI, PTI, sPFN, ePFN, sVA, eVA;
    hardware_pte_x86 pdm,pdp,pde,pte;
    uint32_t pdetmp, ptetmp, pdmtmp64_high, pdmtmp64_low, pdptmp64_high,
             pdptmp64_low, pdetmp64_high, pdetmp64_low, ptetmp64_high, ptetmp64_low;
    uint64_t i,j,k, m;
    dtb_list *wdtb;
    ptov_list *wptov;

    wdtb = ptov_dtb;
    ptovs = NULL;
    printf("Initializing ptov utility\n");
    while (wdtb != NULL) {

        printf(".");
        mem_move(0);
        wptov = malloc(sizeof(ptov_list));
        wptov->next = ptovs;
        wptov->pid = wdtb->pid;
        ptovs = wptov;
        wptov->vaddress = malloc(sizeof(uint64_t) * (get_mem_size() >> 12) );
        for (i = 0; i < (get_mem_size() >> 12); i++) wptov->vaddress[i] = 0;
        if (x64) {

            //            printf("PID = %lli Dtb = %llx\n",wptov->pid,wdtb->dtb);
            for (i = 0; i < 0x200;i++) {
```

```
PDM = i << 39;
pdmtmp64_low = figrab_32_LE(image_file, wdtb->dtb + (i<<3));
pdmtmp64_high = figrab_32_LE(image_file, wdtb->dtb + (i<<3) + 4);
map_pte64(pdmtmp64_high,pdmtmp64_low, &pdm);
if (pdm.Valid) {
    // not handling virtual address swapped to disk
    for (j = 0; j < 0x200;j++) {

        PDP = j << 30;
        pdptmp64_low = figrab_32_LE(image_file,
                       (pdm.PageFrameNumber64_high << 32) +
                        pdm.PageFrameNumber64_low  + (j<<3));
        pdptmp64_high = figrab_32_LE(image_file,
                       (pdm.PageFrameNumber64_high << 32) +
                        pdm.PageFrameNumber64_low  + (j<<3) + 4);
        map_pte64(pdptmp64_high,pdptmp64_low, &pdp);
        if (pdp.Valid) {
            // not handling virtual address swapped to disk
            for (k = 0; k < 0x200; k++) {

                PDI = k << 21;
                pdetmp64_low = figrab_32_LE(image_file,
                        (pdp.PageFrameNumber64_high << 32) +
                        pdp.PageFrameNumber64_low  + (k<<3));
                pdetmp64_high = figrab_32_LE(image_file,
                       (pdp.PageFrameNumber64_high << 32) +
                       pdp.PageFrameNumber64_low  + (k<<3) + 4);
                map_pte64(pdetmp64_high,pdetmp64_low, &pde);
                if ((pde.Valid) && (pde.LargePage)) {

                    sPFN = (pde.PageFrameNumber64_high << 32) +
                           pde.PageFrameNumber64_low;
                    ePFN = (pde.PageFrameNumber64_high << 32) +
                           pde.PageFrameNumber64_low + 0x1FFFFF;
                    sVA = PDM + PDP + PDI;
                    eVA = PDI + 0x1FFFFF;
                    if (sPFN <= get_mem_size())
                         for (m = 0; m < 0x200; m++)
                             wptov->vaddress[(sPFN >>12) + m] =
                                       sVA + m * 0x1000;


                }

                else if (pde.Valid) {
```

```
                    for (m=0; m < 0x200; m++) {

                        PTI = m << 12;
                        ptetmp64_low = figrab_32_LE(image_file,
                             (pde.PageFrameNumber64_high << 32)
                           + pde.PageFrameNumber64_low  + (m<<3));
                        ptetmp64_high = figrab_32_LE(image_file,
                             (pde.PageFrameNumber64_high << 32)
                           + pde.PageFrameNumber64_low  + (m<<3) + 4);
                        map_pte64(ptetmp64_high,ptetmp64_low, &pte);
                        if (pte.Valid) {
                            // not handling virtual addressed
                            // swapped to disk or
                            // prototype pages
                            sPFN = (pte.PageFrameNumber64_high << 32)
                                  + pte.PageFrameNumber64_low;
                            ePFN = (pte.PageFrameNumber64_high << 32)
                                 + pte.PageFrameNumber64_low + 0xFFF;
                            sVA = PDM + PDP + PDI + PTI;
                            eVA = PDI + PTI + 0xFFF;

                            if (sPFN <= get_mem_size())
                                wptov->vaddress[sPFN >>12] = sVA;


                        }


                    }


                }


            }


        }


    }


}
```

```
    }


}

else if (!NO_PAE) {

    for (i = 0; i < 0x4;i++) {

        PDP = i << 30;
        pdptmp64_low = figrab_32_LE(image_file, wdtb->dtb + (i<<3));
        pdptmp64_high = figrab_32_LE(image_file, wdtb->dtb + (i<<3) + 4);
        map_pte64(pdptmp64_high,pdptmp64_low, &pdp);
        if (pdp.Valid) {
            // not handling virtual address swapped to disk
            for (j = 0; j < 0x200; j++) {

                PDI = j << 21;
                pdetmp64_low = figrab_32_LE(image_file,
                       (pdp.PageFrameNumber64_high << 32) +
                       pdp.PageFrameNumber64_low + (j<<3));
                pdetmp64_high = figrab_32_LE(image_file,
                      (pdp.PageFrameNumber64_high << 32) +
                      pdp.PageFrameNumber64_low + (j<<3) + 4);
                map_pte64(pdetmp64_high,pdetmp64_low, &pde);
                if ((pde.Valid) && (pde.LargePage)) {

                    sPFN = (pde.PageFrameNumber64_high << 32) +
                           pde.PageFrameNumber64_low;
                    ePFN = (pde.PageFrameNumber64_high << 32) +
                           pde.PageFrameNumber64_low + 0x1FFFFF;
                    sVA = PDP + PDI;
                    eVA = PDI + 0x1FFFFF;
                    if (sPFN <= get_mem_size())
                      for (k = 0; k < 0x200; k++)
                        wptov->vaddress[(sPFN>>12) + k] = sVA + k * 0x1000;

                }

                else if (pde.Valid) {

                    for (k=0; k < 0x200; k++) {
```

```
                        PTI = k << 12;
                        ptetmp64_low = figrab_32_LE(image_file,
                              (pde.PageFrameNumber64_high << 32) +
                              pde.PageFrameNumber64_low + (k<<3));
                        ptetmp64_high = figrab_32_LE(image_file,
                              (pde.PageFrameNumber64_high << 32) +
                              pde.PageFrameNumber64_low + (k<<3) + 4);
                        map_pte64(ptetmp64_high,ptetmp64_low, &pte);
                        if (pte.Valid) {
                            // not handling virtual addressed swapped to disk
                            // or prototype pages
                            sPFN = (pte.PageFrameNumber64_high << 32) +
                                    pte.PageFrameNumber64_low;
                            ePFN = (pte.PageFrameNumber64_high << 32) +
                                    pte.PageFrameNumber64_low + 0xFFF;
                            sVA = PDP + PDI + PTI;
                            eVA = PDI + PTI + 0xFFF;
                            if (sPFN <= get_mem_size())
                              wptov->vaddress[sPFN>>12] = sVA;

                        }


                    }


                }


            }


        }


    }


}

else {

    for (i = 0; i < 0x400;i++) {

        PDI = i << 22;
```

```
pdetmp = figrab_32_LE(image_file, wdtb->dtb + (i<<2));
map_pte(pdetmp, &pde);
if (pde.Valid) {
    // not handling virtual address swapped to disk
    if (pde.LargePage) {

        sPFN = pde.PageFrameNumber;
        ePFN = pde.PageFrameNumber + 0x3FFFFF;
        sVA = PDI;
        eVA = PDI + 0x3FFFFF;
        if (sPFN <= get_mem_size())
          for (j = 0; j < 0x400; j++)
            wptov->vaddress[(sPFN >>12) + j] = sVA + j * 0x400;

    }

    else {

        for (j=0; j < 0x400; j++) {

            PTI = j << 12;
            ptetmp = figrab_32_LE(image_file,
                        pde.PageFrameNumber + (j<<2));
            map_pte(ptetmp, &pte);
            if (pte.Valid) {
                // not handling virtual addressed swapped to
                // disk or prototype pages
                sPFN = pte.PageFrameNumber;
                ePFN = pte.PageFrameNumber + 0xFFF;
                sVA = PDI + PTI;
                eVA = PDI + PTI + 0xFFF;
                if (sPFN <= get_mem_size())
                  wptov->vaddress[sPFN >>12] = sVA;

            }


        }


    }


}
```

```
            }


        }

        wdtb = wdtb->next;

    }

    printf("\n");

}


void add_ptov_dtb( uint64_t pid, uint64_t dtb) {

    dtb_list *wdtb;
    wdtb = malloc(sizeof(dtb_list));
    wdtb->next = ptov_dtb;
    ptov_dtb = wdtb;
    wdtb->pid = pid;
    wdtb->dtb = dtb;
    //    printf("PID = %llx   DTB = %llx\n",wdtb->pid,wdtb->dtb);

}


ptov_ret_list *ptov(uint64_t physaddr) {

    uint32_t windex;
    uint64_t wretval,chkit;
    ptov_list *wptov;
    ptov_ret_list *retlist,*retentry;

    retlist = NULL;
    windex = (uint32_t) (physaddr >>12);  // Limits memory addressed to 4TB
    wptov = ptovs;
    while (wptov != NULL) {

        wretval = wptov->vaddress[windex];
        wretval = wretval & 0x0000FFFFFFFFFFFFULL; // The high 20 bits should
                                     // all either be 1 (system) or 0 (user)
        chkit = wretval >> 47;
        if (chkit) wretval = wretval | 0xffff000000000000ULL;
```

```
        if (wretval != 0) {

            retentry = malloc(sizeof(ptov_ret_list));
            retentry->next = retlist;
            retlist = retentry;
            retentry->pid = 0;
            retentry->vaddress = wretval | (physaddr & 0x0000000000000fffULL);
            //            if (chkit) return retlist; // System Process
            retentry->pid = wptov->pid;

        }

        wptov = wptov->next;

    }

    return retlist;

}



/*********************************************************/
/* Read a byte from an inputted virtual address -- Private */
/*********************************************************/
unsigned char *read_proc_mem_helper (uint64_t DTB, uint64_t start_address,
 unsigned int length, uint8_t wis_proto, uint8_t do_what)
{

    uint8_t DEBUGIT;
    FILE *page_file;
    FILE *map_file;
    char *wname;

    unsigned int i, bytes_to_grab;
    unsigned char *result;
    uint32_t PDM, PDI, PTI, BI, PDP;
    uint64_t proto_index, proto_pte, wbase_proto, wbase_sector, wsector,
             pagefile_offset, wptr;
    hardware_pte_x86 pte;
    uint32_t ptetmp, ptetmp64_high, ptetmp64_low;
    uint32_t pagefile;
    uint32_t shift_amt;
```

```
unsigned int res_index = 0;        // index within result keeps stat across
                                   // multiple loops
unicode_string wFileName;

uint32_t  last_pte;
DEBUGIT = (start_address == 0x776613FC);
if (DEBUGIT) printf("DEBUGIT = %x\n",DEBUGIT);
result = malloc(sizeof(unsigned char) * length);
PDM = 0; PDP = 0; PDI = 0; PTI = 0; BI = 0;
for (i=0; i < length; i++) result[i] = 0;
if (x64) {

    PDM = (uint32_t) ((start_address >> 39) & 0x1FF) << 3;
    PDP = (uint32_t) ((start_address >> 30) & 0x1FF) << 3;
    PDI = (uint32_t) ((start_address >> 21) & 0x1FF) << 3;
    PTI = (uint32_t) ((start_address >> 12) & 0x1FF) << 3;
    BI  = (uint32_t)  (start_address & 0xFFF);

}

else if (!NO_PAE) {

    PDP = (uint32_t) (start_address >> 30) << 3;
    PDI = (uint32_t) ((start_address >> 21) & 0x1FF) << 3;
    PTI = (uint32_t) ((start_address >> 12) & 0x1FF) << 3;
    BI  = (uint32_t)  (start_address & 0xFFF);

}

else {

    PDI = (uint32_t) (start_address >> 22) << 2;
    PTI = (uint32_t) ((start_address >> 12) & 0x3FF) << 2;
    BI  = (uint32_t) (start_address & 0xFFF);

}

// For each byte(s) to be read
while (length > 0) {


    // IF we're using x64, we need to do 3 lookups -- one for the page map level,
    // a second for the directory pointer table and a third for the page directory
    if (x64) {
```

```
    ptetmp64_low = figrab_32_LE(image_file, DTB + PDM);
    ptetmp64_high = figrab_32_LE(image_file, DTB + PDM + 4);
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);
    if (!(pte.Valid) ) return result; // This should never happen

    ptetmp64_low = figrab_32_LE(image_file, (pte.PageFrameNumber64_high
                    << 32) + pte.PageFrameNumber64_low + PDP);
    ptetmp64_high = figrab_32_LE(image_file, (pte.PageFrameNumber64_high
                    << 32) + pte.PageFrameNumber64_low + PDP + 4);
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);
    if (!(pte.Valid) ) return result; // This should never happen

    ptetmp64_low = figrab_32_LE(image_file, (pte.PageFrameNumber64_high
                     << 32) + pte.PageFrameNumber64_low + PDI);
    ptetmp64_high = figrab_32_LE(image_file, (pte.PageFrameNumber64_high
                    << 32) + pte.PageFrameNumber64_low + PDI + 4);
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);

}

// IF PAE is enabled, we need to do 2 lookups -- one for the page
//  directory pointer table and a second for the page directory
else if (!NO_PAE) {

    ptetmp64_low = figrab_32_LE(image_file, DTB + PDP);
    ptetmp64_high = figrab_32_LE(image_file, DTB + PDP + 4);
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);
    if (!(pte.Valid) ) return result; // This should never happen

    ptetmp64_low = figrab_32_LE(image_file, (pte.PageFrameNumber64_high
                   << 32) + pte.PageFrameNumber64_low + PDI);
    ptetmp64_high = figrab_32_LE(image_file, (pte.PageFrameNumber64_high
                   << 32) + pte.PageFrameNumber64_low + PDI + 4);
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);

}

// If PAE is not enabled, we only need to do one lookup
else {

    ptetmp = figrab_32_LE(image_file, DTB + PDI);
    map_pte(ptetmp, &pte);

}
```

```
// We are now looking at the Page Directory Entry

// If we've got an invalid page
if (!(pte.Valid) && (!pte.Transition || pte.Prototype) ) {


    if ( pte.Prototype == 1)
    return result; // This should never happen 'cause PDEs should never
                   // be prototypes
    else {
        //page file

        // We figure out which pagefile and what the offset is
        if (!NO_PAE) {
            // PAE is enabled with 64bit
            pagefile = (ptetmp64_low & 0x0000001E) >> 1;
            pagefile_offset = (ptetmp64_low & 0xFFFFF000);

        }

        else {

            pagefile = (ptetmp & 0x0000001E) >> 1;
            pagefile_offset = (ptetmp & 0xFFFFF000);

        }

        if (pte.LargePage)
        BI += ( (!NO_PAE) ? (PTI << 9) : (PTI << 10));

        // If it's a demand zero page, we send back zeros
        if ((pagefile == 0) && (pagefile_offset == 0)) // Demand Zero Page
        return result;

        // Otherwise we've got to go out to the pagefile
        else {


            // if the PDE says it's a large page, we're going to be getting
            // the page frame from the page file
            if (pte.LargePage) {

                if (!NO_PAE)
                  bytes_to_grab = (length > (2097152 - BI)) ? (2097152 -
```

```
                 BI) : length; /*2M */
    else
      bytes_to_grab = (length > (4194304 - BI)) ? (4194304 -
                 BI) : length; /*4M */
    page_file = get_pagefile(pagefile);
    if (page_file == NULL) {

        return result;

    }

    if (do_what == GET_VTOP) {

        result[0] = (pagefile_offset + BI) >> 56;
        result[1] = ((pagefile_offset + BI) >> 48)
                  & 0x00000000000000FF;
        result[2] = ((pagefile_offset + BI) >> 40)
                  & 0x00000000000000FF;
        result[3] = ((pagefile_offset + BI) >> 32)
                  & 0x00000000000000FF;
        result[4] = ((pagefile_offset + BI) >> 24)
                  & 0x00000000000000FF;
        result[5] = ((pagefile_offset + BI) >> 16)
                  & 0x00000000000000FF;
        result[6] = ((pagefile_offset + BI) >> 8)
                  & 0x00000000000000FF;
        result[7] = (pagefile_offset + BI)
                  & 0x00000000000000FF;
        result[8] = IS_PAGEFILE;
        return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
         result[res_index++] = figrab_8_LE(page_file,
                                   pagefile_offset + BI + i);
    length -= bytes_to_grab;
    continue;

}


// otherwise, we get the PTE from the page file and then go
// look for the page frame
else {
```

```
// we get the PTE from the page file
page_file = get_pagefile(pagefile);
if (page_file == NULL) {

    return result;

}

if (!NO_PAE) {
    // PAE is enabled with 64bit
    ptetmp64_low = figrab_32_LE(page_file,
                   pagefile_offset + PTI);
    ptetmp64_high = figrab_32_LE(page_file,
                    pagefile_offset + PTI + 4);
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);

}

else {

    ptetmp = figrab_32_LE(page_file, pagefile_offset + PTI);
    map_pte(ptetmp, &pte);

}

bytes_to_grab = (length > (4096 - BI)) ? (4096 - BI) :
                length;    /* 4k */

// We check if the PTE is valid -- if not, we may be
// going back to page file
if (!(pte.Valid) && (!pte.Transition || pte.Prototype)) {

    if (pte.Prototype) {
        // prototype page
        if (!NO_PAE) {
            // PAE is enabled with 64bit
            // If PAE is enabled, the high 32 bits
            // contain the virtual
            // address of the prototype PTE
            proto_index = ptetmp64_high;
            ptetmp64_low = memgrabp_32_LE(DTB,
                           ptetmp64_high);
            ptetmp64_high = memgrabp_32_LE(DTB,
                           ptetmp64_high + 4);
```

```
    map_pte64(ptetmp64_high,ptetmp64_low, &pte);
  // PAE is enabled, the pointer to the
  // subsection that
  // references the mapped file is in the high 32 bits
    proto_pte = ptetmp64_high;
    shift_amt = 3;

}

else {

    proto_index = 0xe1000000 + ( ( (ptetmp >> 2)
               & 0x3FFFFe00) +
               ( (ptetmp << 1) & 0x000001fc) );
    ptetmp = memgrabp_32_LE(DTB, proto_index);
    map_pte(ptetmp, &pte);
    proto_pte = ptetmp;
    shift_amt = 2;

}

if (!(pte.Valid) && (pte.Prototype)) {
    // if the prototype bit is set, this refers
    // to a page in a mapped file
    // unfortunately, there's nothing we can do,
    // since the
    // physical file may be on some other system
    wptr        = keget_pdb_type  (
                    "_SUBSECTION.ControlArea",
                    proto_pte);
    wptr        = keget_pdb_type  (
                    "_CONTROL_AREA.FilePointer",
                    wptr);
    wFileName.Length        = (uint16_t)
          keget_pdb_type
          ( "_FILE_OBJECT.FileName.Length", wptr);
    wFileName.MaximumLength = (uint16_t)
          keget_pdb_type
          ("_FILE_OBJECT.FileName.MaximumLength",
          wptr);
    wFileName.Buffer        = keget_pdb_type
           ( "_FILE_OBJECT.FileName.Buffer", wptr) + 2;
    wbase_proto  = keget_pdb_type
           ( "_SUBSECTION.SubsectionBase", proto_pte);
    wbase_sector = keget_pdb_type
```

```
       ( "_SUBSECTION.StartingSector", proto_pte);
wsector = (wbase_sector + ( (proto_index -
          wbase_proto) >> shift_amt )) <<12;
if (wFileName.Length <= 1)
  return result;
wname = malloc(sizeof(char) *
        (wFileName.Length + 1));
wname = memgrabp_unicode_LE(wFileName,DTB);
map_file = open_map_file(wname);
if (map_file != NULL) {

    if (do_what == GET_VTOP) {

        result[0] = (wsector + BI) >> 56;
        result[1] = ((wsector + BI) >> 48)
                  & 0x00000000000000FF;
        result[2] = ((wsector + BI) >> 40)
                  & 0x00000000000000FF;
        result[3] = ((wsector + BI) >> 32)
                  & 0x00000000000000FF;
        result[4] = ((wsector + BI) >> 24)
                  & 0x00000000000000FF;
        result[5] = ((wsector + BI) >> 16)
                  & 0x00000000000000FF;
        result[6] = ((wsector + BI) >> 8)
                  & 0x00000000000000FF;
        result[7] = (wsector + BI)
                  & 0x00000000000000FF;
        result[8] = IS_MAPFILE;
        return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
       result[res_index++] = figrab_8_LE(map_file,
             wsector + BI + i);
    length -= bytes_to_grab;
    continue;

}

else {

    return result;
```

```
    }


}

else if (!(pte.Valid) && (!pte.Transition)) {
    // page file
    pagefile = (uint32_t) (proto_pte & 0x0000001E) >>1;
    pagefile_offset = (proto_pte & 0xFFFFF000);
    page_file = get_pagefile(pagefile);
    if (page_file == NULL) {

        return result;

    }

    if (do_what == GET_VTOP) {

      result[0] = (pagefile_offset + BI) >> 56;
      result[1] = ((pagefile_offset + BI) >> 48)
                & 0x00000000000000FF;
      result[2] = ((pagefile_offset + BI) >> 40)
                & 0x00000000000000FF;
      result[3] = ((pagefile_offset + BI) >> 32)
                & 0x00000000000000FF;
      result[4] = ((pagefile_offset + BI) >> 24)
                & 0x00000000000000FF;
      result[5] = ((pagefile_offset + BI) >> 16)
                & 0x00000000000000FF;
      result[6] = ((pagefile_offset + BI) >> 8)
                & 0x00000000000000FF;
      result[7] = (pagefile_offset + BI)
                & 0x00000000000000FF;
      result[8] = IS_PAGEFILE;
      return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
       result[res_index++] = figrab_8_LE(page_file,
            pagefile_offset + BI + i);
    length -= bytes_to_grab;
    continue;

}
```

```
    else {

        if (do_what == GET_VTOP) {

          result[0] = 0;
          result[1] = 0;
          result[2] = 0;
          result[3] = 0;
          result[4] = ((pte.PageFrameNumber + BI) >> 24)
                    & 0x00000000000000FF;
          result[5] = ((pte.PageFrameNumber + BI) >> 16)
                    & 0x00000000000000FF;
          result[6] = ((pte.PageFrameNumber + BI) >> 8)
                    & 0x00000000000000FF;
          result[7] = (pte.PageFrameNumber + BI)
                    & 0x00000000000000FF;
          result[8] = 0;
          return result;

        }

        for (i = 0; i < bytes_to_grab; i++)
            result[res_index++] = figrab_8_LE(image_file,
              pte.PageFrameNumber + BI + i);
        length -= bytes_to_grab;
        continue;

    }


}

else {
    //page file
    if (!NO_PAE) {
        // PAE is enabled with 64bit
        pagefile = (ptetmp64_low & 0x0000001E) >> 1;
        pagefile_offset = (ptetmp64_low & 0xFFFFF000);

    }

    else {

        pagefile = (ptetmp & 0x0000001E) >>1;
```

```
    pagefile_offset = (ptetmp & 0xFFFFF000);

}

if ((pagefile == 0) && (pagefile_offset == 0))
    // Demand Zero Page
    return result;
else {

    page_file = get_pagefile(pagefile);
    if (page_file == NULL) {

        return result;

    }

    if (do_what == GET_VTOP) {

        result[0] = (pagefile_offset + BI) >> 56;
        result[1] = ((pagefile_offset + BI) >> 48)
                  & 0x00000000000000FF;
        result[2] = ((pagefile_offset + BI) >> 40)
                  & 0x00000000000000FF;
        result[3] = ((pagefile_offset + BI) >> 32)
                  & 0x00000000000000FF;
        result[4] = ((pagefile_offset + BI) >> 24)
                  & 0x00000000000000FF;
        result[5] = ((pagefile_offset + BI) >> 16)
                  & 0x00000000000000FF;
        result[6] = ((pagefile_offset + BI) >> 8)
                  & 0x00000000000000FF;
        result[7] = (pagefile_offset + BI)
                  & 0x00000000000000FF;
        result[8] = IS_PAGEFILE;
        return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
       result[res_index++] = figrab_8_LE(page_file,
             pagefile_offset + BI + i);
    length -= bytes_to_grab;
    continue;

}
```

```
    }

}

// This time it's a valid page; we'll get it from the
// memory dump file
else {

    if (!NO_PAE) {

        if (do_what == GET_VTOP) {

          result[0] = ((pte.PageFrameNumber64_high << 32)
               + pte.PageFrameNumber64_low + BI) >> 56;
          result[1] = (((pte.PageFrameNumber64_high << 32)
               + pte.PageFrameNumber64_low + BI) >> 48)
               & 0x00000000000000FF;
          result[2] = (((pte.PageFrameNumber64_high << 32)
               + pte.PageFrameNumber64_low + BI) >> 40)
               & 0x00000000000000FF;
          result[3] = (((pte.PageFrameNumber64_high << 32)
              + pte.PageFrameNumber64_low + BI) >> 32)
              & 0x00000000000000FF;
          result[4] = (((pte.PageFrameNumber64_high << 32)
             + pte.PageFrameNumber64_low + BI) >> 24)
             & 0x00000000000000FF;
          result[5] = (((pte.PageFrameNumber64_high << 32)
            + pte.PageFrameNumber64_low + BI) >> 16)
            & 0x00000000000000FF;
          result[6] = (((pte.PageFrameNumber64_high << 32)
            + pte.PageFrameNumber64_low + BI) >> 8)
            & 0x00000000000000FF;
          result[7] = ((pte.PageFrameNumber64_high << 32)
            + pte.PageFrameNumber64_low + BI)
            & 0x00000000000000FF;
            result[8] = 0;
            return result;

        }

        for (i = 0; i < bytes_to_grab; i++)
```

```
                    result[res_index++] =
                     figrab_8_LE(image_file,
                        (pte.PageFrameNumber64_high << 32) +
                        pte.PageFrameNumber64_low + BI + i);

            }

            else {

                if (do_what == GET_VTOP) {

                    result[0] = 0;
                    result[1] = 0;
                    result[2] = 0;
                    result[3] = 0;
                    result[4] = ((pte.PageFrameNumber + BI) >> 24)
                              & 0x00000000000000FF;
                    result[5] = ((pte.PageFrameNumber + BI) >> 16)
                              & 0x00000000000000FF;
                    result[6] = ((pte.PageFrameNumber + BI) >> 8)
                              & 0x00000000000000FF;
                    result[7] = (pte.PageFrameNumber + BI)
                              & 0x00000000000000FF;
                    result[8] = 0;
                    return result;

                }

                for (i = 0; i < bytes_to_grab; i++)
                    result[res_index++] =
                      figrab_8_LE(image_file,
                         pte.PageFrameNumber + BI + i);

            }

            length -= bytes_to_grab;

        }


    }


}
```

```
    }


}


// Otherwise the PTE is valid -- let's go to the next level
else {


    // if it's a large page, all we need to do is get the page frame
    // from memory and we're done
    if (pte.LargePage) {

        if (!NO_PAE) {
            // PAE is enabled with 64bit
            bytes_to_grab = (length > (2097152 - BI)) ? (2097152 - BI) :
                            length;        /* Page is 2M large */
            BI += (PTI << 9);
            pte.PageFrameNumber64_low = (pte.PageFrameNumber64_low
                                        & 0xFFE00000);
            if (do_what == GET_VTOP) {

                result[0] = ((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 56;
                result[1] = (((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 48)
                            & 0x00000000000000FF;
                result[2] = (((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 40)
                           & 0x00000000000000FF;
                result[3] = (((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 32)
                          & 0x00000000000000FF;
                result[4] = (((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 24)
                          & 0x00000000000000FF;
                result[5] = (((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 16)
                          & 0x00000000000000FF;
                result[6] = (((pte.PageFrameNumber64_high << 32) +
                             pte.PageFrameNumber64_low + BI) >> 8)
                          & 0x00000000000000FF;
                result[7] = ((pte.PageFrameNumber64_high << 32) +
```

```
                    pte.PageFrameNumber64_low + BI)
                 & 0x00000000000000FF;
        result[8] = 0;
        return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
       result[res_index++] = figrab_8_LE(image_file,
         (pte.PageFrameNumber64_high << 32)
        + pte.PageFrameNumber64_low + BI + i);

}

else {

    bytes_to_grab = (length > (4194304 - BI)) ? (4194304 - BI) :
                    length;  /* 4M */
    BI += (PTI << 10);
    if (do_what == GET_VTOP) {

        result[0] = 0;
        result[1] = 0;
        result[2] = 0;
        result[3] = 0;
        result[4] = ((pte.PageFrameNumber + BI) >> 24)
                  & 0x00000000000000FF;
        result[5] = ((pte.PageFrameNumber + BI) >> 16)
                  & 0x00000000000000FF;
        result[6] = ((pte.PageFrameNumber + BI) >> 8)
                  & 0x00000000000000FF;
        result[7] = (pte.PageFrameNumber + BI)
                  & 0x00000000000000FF;
        result[8] = 0;
        return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
      result[res_index++] = figrab_8_LE(image_file,
                    pte.PageFrameNumber + BI + i);

}

length -= bytes_to_grab;
```

```
        continue;

    }


    // otherwise, we've got one more level to traverse
    else  {

        bytes_to_grab = (length > (4096 - BI)) ? (4096 - BI)
                        : length;        /* 4k */
        if (!NO_PAE) {
            // PAE is enabled with 64bit
            ptetmp64_low = figrab_32_LE(image_file,
                            (pte.PageFrameNumber64_high << 32)
                          + pte.PageFrameNumber64_low + PTI);
            ptetmp64_high = figrab_32_LE(image_file,
                            (pte.PageFrameNumber64_high << 32)
                          + pte.PageFrameNumber64_low + PTI + 4);
            map_pte64(ptetmp64_high,ptetmp64_low, &pte);
            ptetmp = ptetmp64_low; // just for the debug print statement

        }

        else {

            ptetmp = figrab_32_LE(image_file, pte.PageFrameNumber + PTI);
            map_pte(ptetmp, &pte);

        }


        // If we've got an invalid page frame, we've got to go find it
        if (!(pte.Valid) && (!pte.Transition || pte.Prototype)) {

            if (pte.Prototype) {
                // prototype page
                if (!NO_PAE) {
                    // PAE is enabled with 64bit
                    // If PAE is enabled, the high 32 bits contain the
                    // virtual address of the prototype PTE
                    proto_index = ptetmp64_high;
                    ptetmp64_low = memgrabp_32_LE(DTB,ptetmp64_high);
                    ptetmp64_high = memgrabp_32_LE(DTB,ptetmp64_high + 4);
                    map_pte64(ptetmp64_high,ptetmp64_low, &pte);
                    proto_pte = ptetmp64_high;
```

```
            //  PAE is enabled, the pointer to the
            // subsection that references the
            // mapped file is in the high 32 bits
    shift_amt = 3;

}

else {

    proto_index = 0xe1000000 + ( (
             (ptetmp >> 2) & 0x3FFFFe00) +
             ( (ptetmp << 1) & 0x000001fc) );
    ptetmp = memgrabp_32_LE(DTB, proto_index);
    map_pte(ptetmp, &pte);
    proto_pte = ptetmp;
    shift_amt = 2;

}

if (!(pte.Valid) && (pte.Prototype)) {
    // if the prototype bit is set, this refers to a
    // page in a mapped file
    // unfortunately, there's nothing we can do, since
    // the physical file may be on some other system
    wptr        = keget_pdb_type  (
                "_SUBSECTION.ControlArea", proto_pte);
    wptr        = keget_pdb_type  (
                "_CONTROL_AREA.FilePointer", wptr);
    last_pte    = (uint32_t) keget_pdb_type  (
                "_SUBSECTION.u",proto_pte);
    wFileName.Length        = (uint16_t) keget_pdb_type
                ( "_FILE_OBJECT.FileName.Length", wptr);
    wFileName.MaximumLength = (uint16_t) keget_pdb_type
                ( "_FILE_OBJECT.FileName.MaximumLength", wptr);
    wFileName.Buffer        = keget_pdb_type  (
                  "_FILE_OBJECT.FileName.Buffer", wptr) + 2;
                  // want a relative path
    wbase_proto  = keget_pdb_type  (
                   "_SUBSECTION.SubsectionBase", proto_pte);
    wbase_sector = keget_pdb_type  (
                   "_SUBSECTION.StartingSector", proto_pte);
    wsector = (wbase_sector + ( (proto_index - wbase_proto)
             >> shift_amt )) <<12;
    if (wFileName.Length <= 1)
    return result;
```

```
wname = malloc(sizeof(char) * (wFileName.Length + 1));
wname = memgrabp_unicode_LE(wFileName,DTB);
map_file = open_map_file(wname);
free(wname);
if (map_file != NULL) {

    for (i = 0; i < bytes_to_grab; i++) {

        if (do_what == GET_VTOP) {

            result[0] = (wsector + BI) >> 56;
            result[1] = ((wsector + BI) >> 48)
                      & 0x00000000000000FF;
            result[2] = ((wsector + BI) >> 40)
                      & 0x00000000000000FF;
            result[3] = ((wsector + BI) >> 32)
                      & 0x00000000000000FF;
            result[4] = ((wsector + BI) >> 24)
                      & 0x00000000000000FF;
            result[5] = ((wsector + BI) >> 16)
                      & 0x00000000000000FF;
            result[6] = ((wsector + BI) >> 8)
                      & 0x00000000000000FF;
            result[7] = (wsector + BI)
                      & 0x00000000000000FF;
            result[8] = IS_MAPFILE;
            return result;

        }

        result[res_index++] = figrab_8_LE(map_file,
                                wsector + BI + i);
        if (DEBUGIT) printf("%x ",result[res_index - 1]);

    }

    if (DEBUGIT) printf("\n");
    length -= bytes_to_grab;
    continue;

}

else {

    return result;
```

```
        }


    }

    else if (!(pte.Valid) && (!pte.Transition)) {
        // page file
        pagefile = (uint32_t) (proto_pte & 0x0000001E) >>1;
        pagefile_offset = (proto_pte & 0xFFFFF000);
        page_file = get_pagefile(pagefile);
        if (page_file == NULL) {

            return result;

        }

        if (do_what == GET_VTOP) {

            result[0] = (pagefile_offset + BI) >> 56;
            result[1] = ((pagefile_offset + BI) >> 48)
                      & 0x00000000000000FF;
            result[2] = ((pagefile_offset + BI) >> 40)
                      & 0x00000000000000FF;
            result[3] = ((pagefile_offset + BI) >> 32)
                      & 0x00000000000000FF;
            result[4] = ((pagefile_offset + BI) >> 24)
                      & 0x00000000000000FF;
            result[5] = ((pagefile_offset + BI) >> 16)
                      & 0x00000000000000FF;
            result[6] = ((pagefile_offset + BI) >> 8)
                      & 0x00000000000000FF;
            result[7] = (pagefile_offset + BI)
                      & 0x00000000000000FF;
            result[8] = IS_PAGEFILE;
            return result;

        }

        for (i = 0; i < bytes_to_grab; i++)
          result[res_index++] = figrab_8_LE(page_file,
                                  pagefile_offset + BI + i);
        length -= bytes_to_grab;
        continue;
```

```
    }

    else {

        if (do_what == GET_VTOP) {

            result[0] = 0;
            result[1] = 0;
            result[2] = 0;
            result[3] = 0;
            result[4] = ((pte.PageFrameNumber + BI) >> 24)
                      & 0x00000000000000FF;
            result[5] = ((pte.PageFrameNumber + BI) >> 16)
                      & 0x00000000000000FF;
            result[6] = ((pte.PageFrameNumber + BI) >> 8)
                      & 0x00000000000000FF;
            result[7] = (pte.PageFrameNumber + BI)
                      & 0x00000000000000FF;
            result[8] = 0;
            return result;

        }

        for (i = 0; i < bytes_to_grab; i++)
          result[res_index++] = figrab_8_LE(image_file,
                                  pte.PageFrameNumber + BI + i);
        length -= bytes_to_grab;
        continue;

    }


}

else {
    //page file
    if (!NO_PAE) {
        // PAE is enabled with 64bit
        pagefile = (ptetmp64_low & 0x0000001E) >> 1;
        pagefile_offset = (ptetmp64_low & 0xFFFFF000);

    }

    else {
```

```
    pagefile = (ptetmp & 0x0000001E) >>1;
    pagefile_offset = (ptetmp & 0xFFFFF000);

}

if ((pagefile == 0) && (pagefile_offset == 0))
    // Demand Zero Page
    return result;
else {

    page_file = get_pagefile(pagefile);
    if (page_file == NULL) {

        return result;

    }

    if (do_what == GET_VTOP) {

        result[0] = (pagefile_offset + BI) >> 56;
        result[1] = ((pagefile_offset + BI) >> 48)
                  & 0x00000000000000FF;
        result[2] = ((pagefile_offset + BI) >> 40)
                  & 0x00000000000000FF;
        result[3] = ((pagefile_offset + BI) >> 32)
                  & 0x00000000000000FF;
        result[4] = ((pagefile_offset + BI) >> 24)
                  & 0x00000000000000FF;
        result[5] = ((pagefile_offset + BI) >> 16)
                  & 0x00000000000000FF;
        result[6] = ((pagefile_offset + BI) >> 8)
                  & 0x00000000000000FF;
        result[7] = (pagefile_offset + BI)
                  & 0x00000000000000FF;
        result[8] = IS_PAGEFILE;
        return result;

    }

    for (i = 0; i < bytes_to_grab; i++)
      result[res_index++] = figrab_8_LE(page_file,
                              pagefile_offset + BI + i);
    length -= bytes_to_grab;
    continue;
```

```
        }

    }

}

// Otherwise, it's a valid page -- we just need to get it from
// the memory dump
else {

    if (!NO_PAE) {

        if (do_what == GET_VTOP) {

            result[0] = ((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 56;
            result[1] = (((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 48)
                      & 0x00000000000000FF;
            result[2] = (((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 40)
                      & 0x00000000000000FF;
            result[3] = (((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 32)
                      & 0x00000000000000FF;
            result[4] = (((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 24)
                      & 0x00000000000000FF;
            result[5] = (((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 16)
                      & 0x00000000000000FF;
            result[6] = (((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI) >> 8)
                      & 0x00000000000000FF;
            result[7] = ((pte.PageFrameNumber64_high << 32)
                      + pte.PageFrameNumber64_low + BI)
                      & 0x00000000000000FF;
            result[8] = 0;
            return result;

        }
```

```
            for (i = 0; i < bytes_to_grab; i++)
              result[res_index++] = figrab_8_LE(image_file,
                         (pte.PageFrameNumber64_high << 32)
                         + pte.PageFrameNumber64_low + BI + i);
            length -= bytes_to_grab;
            continue;

        }

        else {

            if (do_what == GET_VTOP) {

                result[0] = 0;
                result[1] = 0;
                result[2] = 0;
                result[3] = 0;
                result[4] = ((pte.PageFrameNumber + BI) >> 24)
                          & 0x00000000000000FF;
                result[5] = ((pte.PageFrameNumber + BI) >> 16)
                          & 0x00000000000000FF;
                result[6] = ((pte.PageFrameNumber + BI) >> 8)
                          & 0x00000000000000FF;
                result[7] = (pte.PageFrameNumber + BI)
                          & 0x00000000000000FF;
                result[8] = 0;
                return result;

            }

            for (i = 0; i < bytes_to_grab; i++)
              result[res_index++] = figrab_8_LE(image_file,
                                pte.PageFrameNumber + BI + i);
            length -= bytes_to_grab;
            continue;

        }


    }


}
```

```
        }

    }

    return result;
}

/******************************************************************************/

/*****************************************************/
/* MOVING AND GETTING A BYTE FROM MEMORY -- Private */
/*****************************************************/

/*************************************************************/
/* Moves to a particular physical location in memory -- Private */
/*************************************************************/
int fimove (FILE *wfile, uint64_t wloc) {

    // BEGIN -- CMAT_V
    if(wfile == NULL){

        memory_index = (uint32_t) (wloc & 0x00000000ffffffff);
        //printf("Moved to %lx. Leaving FIMOVE\n",memory_index);
        return 1;

    }


    else{

        // END -- CMAT_V
        uint64_t mem_bb = 0;
        largefile_list *tmp;

        tmp = largefiles;
        while ((tmp->file_ptr != wfile) && (tmp != NULL) ) tmp = tmp->next;
        if (tmp == NULL) return 0;

        tmp->cur_index = (uint32_t) (wloc / (((uint64_t) file_seg)
                       & 0x00000000ffffffff) );
        tmp->cur_pos =   (uint32_t) (wloc % (((uint64_t) file_seg)
```

```
                    & 0x00000000ffffffff) );

        if (tmp->cur_index > tmp->mem_2g) return 0;
        fseek(tmp->mem_ptrs[tmp->cur_index],tmp->cur_pos,SEEK_SET);
        return (!feof(tmp->mem_ptrs[tmp->cur_index]));

    }
    // CMAT_V

}


/*******************************************************/
/* Get a byte from where the file pointer is -- Private */
/*******************************************************/
uint8_t figetc (FILE *wfile) {

    uint8_t data_grab = 0;

    // BEGIN -- CMAT_V
    if (wfile == NULL){


        if(memory_index > local_instance->m.xen.size) {

            return 1;

        }

        if(memory==NULL){

            //first pass through map memory
            memory = xa_access_pa(local_instance, memory_index, &offset, PROT_READ);

        }


        else if(memory_index < prev_memory_index){

            if(prev_memory_index - offset > memory_index){

                if(memory!=NULL)
                munmap(memory, local_instance->page_size);
                /* map the page from machine memory */
                memory = xa_access_pa(local_instance, memory_index, &offset, PROT_READ);
```

```
    }

    else{

        offset = offset - (prev_memory_index - memory_index);

    }


}

else {

    if(memory_index-prev_memory_index+offset >=
         local_instance->page_size){

        if(memory!=NULL)
        munmap(memory, local_instance->page_size);
        /* map the page from machine memory */
        memory = xa_access_pa(local_instance, memory_index, &offset, PROT_READ);

    }

    else{

        offset = offset + (memory_index-prev_memory_index);

    }


}



/* write memory to file */
if (memory){

    /* memory mapped, just get byte*/
    data_grab = memory[offset];

}

else{
```

```
            /* memory not mapped, write zeros */
            data_grab = 0;

        }

        prev_memory_index = memory_index;
        memory_index += 1;    //increment to next byte in memory by default


    }
    //endif wfile=NULL

    else{

        // END -- CMAT_V
        largefile_list *tmp = largefiles;
        while ((tmp != NULL) && (tmp->file_ptr != wfile)) tmp = tmp->next;
        if (tmp == NULL) return 0;

        data_grab = fgetc(tmp->mem_ptrs[tmp->cur_index]);
        tmp->cur_pos++;
        if (tmp->cur_pos > file_seg) printf("darn it!!\n");
        if (tmp->cur_pos == file_seg ) {

            tmp->cur_index++;
            tmp->cur_pos = 0;
            fseek(tmp->mem_ptrs[tmp->cur_index],tmp->cur_pos,SEEK_SET);

        }


    }
    // CMAT_V
    return data_grab;

}


/*********************************************************/
/* Determine whether the file pointer is at EOF -- Private */
/*********************************************************/
int fieof (FILE *wfile) {

    largefile_list *tmp = largefiles;
    while ((tmp != NULL) && (tmp->file_ptr != wfile)) tmp = tmp->next;
```

```
    if (tmp == NULL) return 1;
    return feof(tmp->mem_ptrs[tmp->cur_index]);

}


/*****************************************************************************/

/****************************************************************************/
/* DOES AN OPTIONAL PHYSICAL SEEK AND THEN GRABS BYTE(S) FROM A MEMORY DUMP */
/* FILE IN LITTLE ENDIAN ORDER -- Private                                   */
/****************************************************************************/

/***************/
/* Grab 1 byte */
/***************/
uint8_t figrab_8_LE(FILE *wfile, uint64_t wloc)
{

    uint8_t data_grab = 0;
    if (wloc != 0)
    if (!fimove(wfile,wloc)) return 0;
    data_grab = figetc(wfile);
    return data_grab;

}


/****************/
/* Grab 4 bytes */
/****************/
uint32_t figrab_32_LE(FILE *wfile, uint64_t wloc)
{

    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint32_t byte4 = 0;

    if (wloc != 0)
    if (!fimove(wfile,wloc)) return 0;
    byte1 = figetc(wfile);
    byte2 = figetc(wfile);
    byte3 = figetc(wfile);
    byte4 = figetc(wfile);
```

```
    byte4 = byte4 << 8;
    byte4 += byte3;
    byte4 = byte4 << 8;
    byte4 += byte2;
    byte4 = byte4 << 8;
    byte4 += byte1;

    return byte4;

}


/****************/
/* Grab 8 bytes */
/****************/
uint64_t figrab_64_LE(FILE *wfile, uint64_t wloc)
{

    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
    uint8_t byte7 = 0;
    uint64_t byte8 = 0;

    if (wloc != 0)
    if (!fimove(wfile,wloc)) return 0;
    byte1 = figetc(wfile);
    byte2 = figetc(wfile);
    byte3 = figetc(wfile);
    byte4 = figetc(wfile);
    byte5 = figetc(wfile);
    byte6 = figetc(wfile);
    byte7 = figetc(wfile);
    byte8 = figetc(wfile);

    byte8 = byte8 << 8;
    byte8 += byte7;
    byte8 = byte8 << 8;
    byte8 += byte6;
    byte8 = byte8 << 8;
    byte8 += byte5;
```

```
    byte8 = byte8 << 8;
    byte8 += byte4;
    byte8 = byte8 << 8;
    byte8 += byte3;
    byte8 = byte8 << 8;
    byte8 += byte2;
    byte8 = byte8 << 8;
    byte8 += byte1;

    return byte8;

}


/****************************************************************************/

/****************************************************************************/
/* MOVING TO A PHYSICAL LOCATION IN MEMORY. GETMEMORY SIZE AND EOF -- Public */
/****************************************************************************/

/********************************************/
/* Moves to a particular location in memory */
/********************************************/
int mem_move (uint64_t wloc) {

    return fimove(image_file,wloc);

}


/*******************/
/* Check if at EOF */
/*******************/
int mem_eof() {

    return fieof(image_file);

}


/*****************************/
/* Return the size of memory */
/*****************************/
uint64_t get_mem_size() {
```

```
    uint64_t mem_size;
    largefile_list *tmp = largefiles;

    // BEGIN -- CMAT_V
    if(image_file == NULL){

        mem_size = local_instance->m.xen.size;

    }

    else{

        // END -- CMAT_V
        while ((tmp != NULL) && (tmp->file_ptr != image_file)) tmp = tmp->next;
        if (tmp == NULL) return 0;

        fseek(tmp->mem_ptrs[tmp->mem_2g],0,SEEK_END);
        mem_size = ( ((uint64_t) file_seg)  & 0x00000000ffffffff);
        mem_size = mem_size    * ( ((uint64_t) tmp->mem_2g)
                 & 0x00000000ffffffff);
        mem_size = mem_size + (((uint64_t) ftell(tmp->mem_ptrs[tmp->mem_2g]))
                 & 0x00000000ffffffff);
        fseek(tmp->mem_ptrs[tmp->cur_index],tmp->cur_pos,SEEK_SET);

    }
    // CMAT_V

    return mem_size;


}


/*******************************************************************************/

/******************************************************************************/
/* DOES AN OPTIONAL PHYSICAL SEEK AND THEN GRABS BYTE(S) FROM A MEMORY DUMP */
/* FILE IN LITTLE ENDIAN ORDER -- Public                                    */
/******************************************************************************/

/***************/
/* Grab 1 byte */
/***************/
uint8_t memgrab_8_LE(uint64_t wloc)
{
```

```
    uint8_t data_grab = 0;
    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    data_grab = figetc(image_file);
    return data_grab;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t memgrab_16_LE(uint64_t wloc)
{

    uint8_t byte1 = 0;
    uint16_t byte2 = 0;
    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);
    byte2 = byte2 << 8;
    byte2 += byte1;

    return byte2;

}


/****************/
/* Grab 4 bytes */
/****************/
uint32_t memgrab_32_LE(uint64_t wloc)
{

    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint32_t byte4 = 0;

    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);
```

```
    byte3 = figetc(image_file);
    byte4 = figetc(image_file);

    byte4 = byte4 << 8;
    byte4 += byte3;
    byte4 = byte4 << 8;
    byte4 += byte2;
    byte4 = byte4 << 8;
    byte4 += byte1;

    return byte4;

}


/****************/
/* Grab 8 bytes */
/****************/
uint64_t memgrab_64_LE(uint64_t wloc)
{

    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
    uint8_t byte7 = 0;
    uint64_t byte8 = 0;

    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);
    byte3 = figetc(image_file);
    byte4 = figetc(image_file);
    byte5 = figetc(image_file);
    byte6 = figetc(image_file);
    byte7 = figetc(image_file);
    byte8 = figetc(image_file);

    byte8 = byte8 << 8;
    byte8 += byte7;
    byte8 = byte8 << 8;
    byte8 += byte6;
```

```
    byte8 = byte8 << 8;
    byte8 += byte5;
    byte8 = byte8 << 8;
    byte8 += byte4;
    byte8 = byte8 << 8;
    byte8 += byte3;
    byte8 = byte8 << 8;
    byte8 += byte2;
    byte8 = byte8 << 8;
    byte8 += byte1;

    return byte8;

}


/****************/
/* Grab 6 bytes */
/****************/
uint64_t memgrab_48_LE(uint64_t wloc)
{

    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
    uint8_t byte7 = 0;
    uint64_t byte8 = 0;

    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);
    byte3 = figetc(image_file);
    byte4 = figetc(image_file);
    byte5 = figetc(image_file);
    byte6 = figetc(image_file);

    byte8 = byte8 << 8;
    byte8 += byte7;
    byte8 = byte8 << 8;
    byte8 += byte6;
    byte8 = byte8 << 8;
```

```
    byte8 += byte5;
    byte8 = byte8 << 8;
    byte8 += byte4;
    byte8 = byte8 << 8;
    byte8 += byte3;
    byte8 = byte8 << 8;
    byte8 += byte2;
    byte8 = byte8 << 8;
    byte8 += byte1;

    return byte8;

}


/************************/
/* Grab a Unicode String */
/************************/
char * memgrab_unicode_LE(unicode_string ustr)
{

    int i;
    char *uni_str = malloc(sizeof(char) * (ustr.Length + 1));

    for (i=0; i < ustr.Length + 1; i++) uni_str[i] = 0;

    for (i=0; i < ustr.Length; i++)
    {

        uni_str[i] = memgrab_8_LE(ustr.Buffer + i * 2);

    }


    return uni_str;

}


/***********************/
/* Grab an ASCII String */
/***********************/
char * memgrab_string_LE (ascii_string str)
{
```

```
    int i;
    char *rstr = malloc(sizeof(char) * (str.Length + 1));

    for (i=0; i < str.Length + 1; i++) rstr[i] = 0;

    for (i=0; i < str.Length; i++)
    {

        rstr[i] = memgrab_8_LE(str.Buffer + i);

    }

    return rstr;

}


/****************************************************************/
/* Grab an ASCII String that is guaranteed to be NULL TERMINATED */
/****************************************************************/
// If not null-terminated it retrieves the first 50 characters
char * memgrab_string0(uint64_t address)
{

    int i, wlength;
    char *rstr;
    uint8_t tmp;

    for (wlength=0; wlength < 50; wlength++)
    {

        tmp = memgrab_8_LE(address + wlength);
        if (tmp == 0) break;

    }

    rstr = malloc(sizeof(char) * (wlength + 1));

    for (i=0; i < wlength + 1; i++) rstr[i] = 0;

    for (i=0; i < wlength; i++)
    {

        rstr[i] = memgrab_8_LE(address + i);
```

```
    }

    return rstr;


}


/******************************************************************************/

/****************************************************************************/
/* DOES AN OPTIONAL PHYSICAL SEEK AND THEN GRABS BYTE(S) FROM A MEMORY DUMP */
/* FILE IN BIG ENDIAN ORDER -- Public                                       */
/****************************************************************************/

/***************/
/* Grab 1 byte */
/***************/
uint8_t memgrab_8_BE(uint64_t wloc)
{

    uint8_t data_grab = 0;
    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    data_grab = figetc(image_file);
    return data_grab;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t memgrab_16_BE(uint64_t wloc)
{

    uint16_t byte1 = 0;
    uint8_t byte2 = 0;

    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);

    byte1 = byte1 << 8;
```

```
    byte1 += byte2;

    return byte1;

}


/****************/
/* Grab 4 bytes */
/****************/
uint32_t memgrab_32_BE(uint64_t wloc)
{

    uint32_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;

    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);
    byte3 = figetc(image_file);
    byte4 = figetc(image_file);

    byte1 = byte1 << 8;
    byte1 += byte2;
    byte1 = byte1 << 8;
    byte1 += byte3;
    byte1 = byte1 << 8;
    byte1 += byte4;

    return byte1;

}


/****************/
/* Grab 8 bytes */
/****************/
uint64_t memgrab_64_BE(uint64_t wloc)
{

    uint64_t byte1 = 0;
    uint8_t byte2 = 0;
```

```
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
    uint8_t byte7 = 0;
    uint8_t byte8 = 0;

    if (wloc != 0)
    if (!fimove(image_file,wloc)) return 0;
    byte1 = figetc(image_file);
    byte2 = figetc(image_file);
    byte3 = figetc(image_file);
    byte4 = figetc(image_file);
    byte5 = figetc(image_file);
    byte6 = figetc(image_file);
    byte7 = figetc(image_file);
    byte8 = figetc(image_file);

    byte1 = byte1 << 8;
    byte1 += byte2;
    byte1 = byte1 << 8;
    byte1 += byte3;
    byte1 = byte1 << 8;
    byte1 += byte4;
    byte1 = byte1 << 8;
    byte1 += byte5;
    byte1 = byte1 << 8;
    byte1 += byte6;
    byte1 = byte1 << 8;
    byte1 += byte7;
    byte1 = byte1 << 8;
    byte1 += byte8;

    return byte1;

}


/******************************************************************************/

/******************************************************************************/
/* GRABS BYTE(S) FROM MEMORY IN LITTLE ENDIAN ORDER USING A REQUESTED VIRTUAL */
/* ADDRESS FOR A PROCESS -- Public                                            */
/******************************************************************************/
```

```
/***************/
/* Grab 1 byte */
/***************/
uint8_t memgrabp_8_LE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint8_t data_grab = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO );
    data_grab = (uint8_t) * tmp;
    free (tmp);
    return data_grab;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t memgrabp_16_LE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint8_t byte1 = 0;
    uint16_t byte2 = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    byte2 = byte2 << 8;
    byte2 += byte1;

    return byte2;


}


/****************/
/* Grab 4 bytes */
/****************/
uint32_t memgrabp_32_LE(uint64_t DTB, uint64_t address)
{
```

```
    unsigned char * tmp;
    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint32_t byte4 = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);

    byte4 = byte4 << 8;
    byte4 += byte3;
    byte4 = byte4 << 8;
    byte4 += byte2;
    byte4 = byte4 << 8;
    byte4 += byte1;

    return byte4;


}


/****************/
/* Grab 6 bytes */
/****************/
uint64_t memgrabp_48_LE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
```

```
    uint8_t byte7 = 0;
    uint64_t byte8 = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+4, 1,NO_PROTO);
    byte5 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+5, 1,NO_PROTO);
    byte6 = (uint8_t) * tmp;
    free (tmp);

    byte8 = byte8 << 8;
    byte8 += byte7;
    byte8 = byte8 << 8;
    byte8 += byte6;
    byte8 = byte8 << 8;
    byte8 += byte5;
    byte8 = byte8 << 8;
    byte8 += byte4;
    byte8 = byte8 << 8;
    byte8 += byte3;
    byte8 = byte8 << 8;
    byte8 += byte2;
    byte8 = byte8 << 8;
    byte8 += byte1;

    return byte8;


}


/****************/
/* Grab 8 bytes */
/****************/
```

```
uint64_t memgrabp_64_LE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
    uint8_t byte7 = 0;
    uint64_t byte8 = 0;

    tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+4, 1,NO_PROTO);
    byte5 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+5, 1,NO_PROTO);
    byte6 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+6, 1,NO_PROTO);
    byte7 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+7, 1,NO_PROTO);
    byte8 = (uint8_t) * tmp;
    free (tmp);

    byte8 = byte8 << 8;
    byte8 += byte7;
    byte8 = byte8 << 8;
    byte8 += byte6;
    byte8 = byte8 << 8;
    byte8 += byte5;
    byte8 = byte8 << 8;
```

```
    byte8 += byte4;
    byte8 = byte8 << 8;
    byte8 += byte3;
    byte8 = byte8 << 8;
    byte8 += byte2;
    byte8 = byte8 << 8;
    byte8 += byte1;

    return byte8;


}


/************************/
/* Grab a Unicode String */
/************************/
char * memgrabp_unicode_LE (unicode_string ustr, uint64_t DTB)
{

    int i;
    char *uni_str = malloc(sizeof(char) * (ustr.Length + 1));

    for (i=0; i < ustr.Length + 1; i++) uni_str[i] = 0;

    for (i=0; i < ustr.Length; i+=2)
    {

        uni_str[i/2] = memgrabp_8_LE(DTB, ustr.Buffer + i);
        /*uni_str[i] = pgrab_8_LE(image_file, sel_proc, ustr.Buffer + (1 + i));*/

    }


    return uni_str;

}


/***********************************************/
/* Grab a Unicode String -- replace 0s with Spaces */
/***********************************************/
char * memgrabp_unicode_LE_space (unicode_string ustr, uint64_t DTB)
{
```

```
    int i;
    char *uni_str = malloc(sizeof(char) * (ustr.Length + 1));

    for (i=0; i < ustr.Length + 1; i++) uni_str[i] = 0;

    for (i=0; i < ustr.Length; i+=2) {

        uni_str[i/2] = memgrabp_8_LE(DTB, ustr.Buffer + i);
        if (uni_str[i/2] == 0) uni_str[i/2] = 0x20;

    }


    return uni_str;

}


/*********************************************************************/
/* Grab a Unicode String that is guaranteed to be terminated with NULL */
/*********************************************************************/
char * memgrabp_unicode_LE_0 (uint64_t address, uint64_t DTB)
{

    int i, wlength;
    char *uni_str;
    uint8_t tmp;

    for (wlength=0; wlength < 100; wlength+=2)
    {

        tmp = memgrabp_8_LE(DTB,address + wlength);
        if (tmp == 0) break;

    }

    uni_str = malloc(sizeof(char) * (wlength + 1));


    for (i=0; i < wlength + 1; i++) uni_str[i] = 0;

    for (i=0; i < wlength; i+=2) {

        uni_str[i/2] = memgrabp_8_LE(DTB, address + i);
```

```
    }

    return uni_str;

}


/************************/
/* Grab an ASCII String */
/************************/
char * memgrabp_string_LE (ascii_string str, uint64_t DTB)
{

    int i;
    char *rstr = malloc(sizeof(char) * (str.Length + 1));

    for (i=0; i < str.Length + 1; i++) rstr[i] = 0;

    for (i=0; i < str.Length; i++)
    {

        rstr[i] = memgrabp_8_LE(DTB, str.Buffer + i);

    }

    return rstr;

}


/*************************************************************/
/* Grab an ASCII String that is guaranteed to be NULL TERMINATED */
/*************************************************************/
// If not null-terminated it retrieves the first 50 characters
char * memgrabp_string0(uint64_t address, uint64_t DTB)
{

    int i, wlength;
    char *rstr;
    uint8_t tmp;

    for (wlength=0; wlength < 50; wlength++)
    {

        tmp = memgrabp_8_LE(DTB,address + wlength);
```

```
        if (tmp == 0) break;

    }

    rstr = malloc(sizeof(char) * (wlength + 1));

    for (i=0; i < wlength + 1; i++) rstr[i] = 0;

    for (i=0; i < wlength; i++)
    {

        rstr[i] = memgrabp_8_LE(DTB,address + i);

    }

    return rstr;


}


char * memgrabp_stringn(uint64_t address, int wlength, uint64_t DTB)
{

    int i;
    char *rstr;

    rstr = malloc(sizeof(char) * (wlength + 1));

    for (i=0; i < wlength + 1; i++) rstr[i] = 0;

    for (i=0; i < wlength; i++)
    {

        rstr[i] = memgrabp_8_LE(DTB,address + i);

    }

    return rstr;


}


/******************************************************************************/
```

```
/*****************************************************************************/
/* GRABS BYTE(S) FROM MEMORY IN LITTLE ENDIAN ORDER USING A REQUESTED VIRTUAL */
/* ADDRESS FOR A KERNEL -- Public                                             */
/*****************************************************************************/

/***************/
/* Grab 1 byte */
/***************/
uint8_t memgrabke_8_LE(uint64_t address)
{

    unsigned char * tmp;
    uint8_t data_grab = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO );
    data_grab = (uint8_t) * tmp;
    free (tmp);
    return data_grab;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t memgrabke_16_LE(uint64_t address)
{

    unsigned char * tmp;
    uint8_t byte1 = 0;
    uint16_t byte2 = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    byte2 = byte2 << 8;
    byte2 += byte1;

    return byte2;


}
```

```
/****************/
/* Grab 4 bytes */
/****************/
uint32_t memgrabke_32_LE(uint64_t address)
{

    unsigned char * tmp;
    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint32_t byte4 = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);

    byte4 = byte4 << 8;
    byte4 += byte3;
    byte4 = byte4 << 8;
    byte4 += byte2;
    byte4 = byte4 << 8;
    byte4 += byte1;

    return byte4;


}


/****************/
/* Grab 8 bytes */
/****************/
uint64_t memgrabke_64_LE(uint64_t address)
{

    unsigned char * tmp;
```

```
uint8_t byte1 = 0;
uint8_t byte2 = 0;
uint8_t byte3 = 0;
uint8_t byte4 = 0;
uint8_t byte5 = 0;
uint8_t byte6 = 0;
uint8_t byte7 = 0;
uint64_t byte8 = 0;
tmp = read_proc_mem (kDTB, address, 1,NO_PROTO);
byte1 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+1, 1,NO_PROTO);
byte2 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+2, 1,NO_PROTO);
byte3 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+3, 1,NO_PROTO);
byte4 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+4, 1,NO_PROTO);
byte5 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+5, 1,NO_PROTO);
byte6 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+6, 1,NO_PROTO);
byte7 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (kDTB, address+7, 1,NO_PROTO);
byte8 = (uint8_t) * tmp;
free (tmp);

byte8 = byte8 << 8;
byte8 += byte7;
byte8 = byte8 << 8;
byte8 += byte6;
byte8 = byte8 << 8;
byte8 += byte5;
byte8 = byte8 << 8;
byte8 += byte4;
byte8 = byte8 << 8;
byte8 += byte3;
byte8 = byte8 << 8;
byte8 += byte2;
```

```
    byte8 = byte8 << 8;
    byte8 += byte1;

    return byte8;

}


/************************/
/* Grab a Unicode String */
/************************/
char * memgrabke_unicode_LE (unicode_string ustr)
{

    int i;
    char *uni_str = malloc(sizeof(char) * (ustr.Length + 1));

    for (i=0; i < ustr.Length + 1; i++) uni_str[i] = 0;

    for (i=0; i < ustr.Length; i+=2)
    {

        uni_str[i/2] = memgrabke_8_LE(ustr.Buffer + i);
        /*uni_str[i] = pgrab_8_LE(image_file, sel_proc, ustr.Buffer + (1 + i));*/

    }


    return uni_str;

}


/**********************************************/
/* Grab a Unicode String -- replace 0s with Spaces */
/**********************************************/
char * memgrabke_unicode_LE_space (unicode_string ustr)
{

    int i;
    char *uni_str = malloc(sizeof(char) * (ustr.Length + 1));

    for (i=0; i < ustr.Length + 1; i++) uni_str[i] = 0;

    for (i=0; i < ustr.Length; i+=2)
```

```
    {

        uni_str[i/2] = memgrabke_8_LE(ustr.Buffer + i);
        if (uni_str[i/2] == 0) uni_str[i/2] = 0x20;
        /*uni_str[i] = pgrab_8_LE(image_file, sel_proc, ustr.Buffer + (1 + i));*/

    }


    return uni_str;

}


/*********************************************************************/
/* Grab a Unicode String that is guaranteed to be terminated with NULL */
/*********************************************************************/
char * memgrabke_unicode_LE_0 (uint64_t address)
{

    int i, wlength;
    char *uni_str;
    uint8_t tmp;

    for (wlength=0; wlength < 100; wlength+=2)
    {

        tmp = memgrabke_8_LE(address + wlength);
        if (tmp == 0) break;

    }

    uni_str = malloc(sizeof(char) * (wlength + 1));


    for (i=0; i < wlength + 1; i++) uni_str[i] = 0;

    for (i=0; i < wlength; i+=2) {

        uni_str[i/2] = memgrabke_8_LE(address + i);

    }

    return uni_str;
```

```
}


/************************/
/* Grab an ASCII String */
/************************/
char * memgrabke_string_LE (ascii_string str)
{

    int i;
    char *rstr = malloc(sizeof(char) * (str.Length + 1));

    for (i=0; i < str.Length + 1; i++) rstr[i] = 0;

    for (i=0; i < str.Length; i++)
    {

        rstr[i] = memgrabke_8_LE(str.Buffer + i);

    }

    return rstr;

}


/***************************************************************/
/* Grab an ASCII String that is guaranteed to be NULL TERMINATED */
/***************************************************************/
// If not null-terminated it retrieves the first 50 characters
char * memgrabke_string0(uint64_t address)
{

    int i, wlength;
    char *rstr;
    uint8_t tmp;

    for (wlength=0; wlength < 50; wlength++)
    {

        tmp = memgrabke_8_LE(address + wlength);
        if (tmp == 0) break;

    }
```

```
    rstr = malloc(sizeof(char) * (wlength + 1));

    for (i=0; i < wlength + 1; i++) rstr[i] = 0;

    for (i=0; i < wlength; i++)
    {

        rstr[i] = memgrabke_8_LE(address + i);

    }

    return rstr;


}


char * memgrabke_stringn(uint64_t address, int wlength)
{

    int i;
    char *rstr;

    rstr = malloc(sizeof(char) * (wlength + 1));

    for (i=0; i < wlength + 1; i++) rstr[i] = 0;

    for (i=0; i < wlength; i++)
    {

        rstr[i] = memgrabke_8_LE(address + i);

    }

    return rstr;


}

/*****************************************************************************/

/****************************************************************************/
/* GRABS BYTE(S) FROM MEMORY IN BIG ENDIAN ORDER USING A REQUESTED VIRTUAL */
/* ADDRESS FOR A PROCESS -- Public                                          */
```

```
/*****************************************************************************/

/***************/
/* Grab 1 byte */
/***************/
uint8_t memgrabp_8_BE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint8_t data_grab = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO );
    data_grab = (uint8_t) * tmp;
    free (tmp);
    return data_grab;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t memgrabp_16_BE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint16_t byte1 = 0;
    uint8_t byte2 = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);

    byte1 = byte1 << 8;
    byte1 += byte2;

    return byte1;


}


/****************/
/* Grab 4 bytes */
```

```
/****************/
uint32_t memgrabp_32_BE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint32_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (DTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);

    byte1 = byte1 << 8;
    byte1 += byte2;
    byte1 = byte1 << 8;
    byte1 += byte3;
    byte1 = byte1 << 8;
    byte1 += byte4;

    return byte1;


}


/****************/
/* Grab 8 bytes */
/****************/
uint64_t memgrabp_64_BE(uint64_t DTB, uint64_t address)
{

    unsigned char * tmp;
    uint64_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
```

```
uint8_t byte4 = 0;
uint8_t byte5 = 0;
uint8_t byte6 = 0;
uint8_t byte7 = 0;
uint8_t byte8 = 0;
tmp = read_proc_mem (DTB, address, 1,NO_PROTO);
byte1 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+1, 1,NO_PROTO);
byte2 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+2, 1,NO_PROTO);
byte3 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+3, 1,NO_PROTO);
byte4 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+4, 1,NO_PROTO);
byte5 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+5, 1,NO_PROTO);
byte6 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+6, 1,NO_PROTO);
byte7 = (uint8_t) * tmp;
free (tmp);
tmp = read_proc_mem (DTB, address+7, 1,NO_PROTO);
byte8 = (uint8_t) * tmp;
free (tmp);

byte1 = byte1 << 8;
byte1 += byte2;
byte1 = byte1 << 8;
byte1 += byte3;
byte1 = byte1 << 8;
byte1 += byte4;
byte1 = byte1 << 8;
byte1 += byte5;
byte1 = byte1 << 8;
byte1 += byte6;
byte1 = byte1 << 8;
byte1 += byte7;
byte1 = byte1 << 8;
byte1 += byte8;
```

```
    return byte1;


}


/*****************************************************************************/

/****************************************************************************/
/* GRABS BYTE(S) FROM MEMORY IN BIG ENDIAN ORDER USING A REQUESTED VIRTUAL */
/* ADDRESS FOR THE KERNEL -- Public                                         */
/****************************************************************************/

/***************/
/* Grab 1 byte */
/***************/
uint8_t memgrabke_8_BE(uint64_t address)
{

    unsigned char * tmp;
    uint8_t data_grab = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO );
    data_grab = (uint8_t) * tmp;
    free (tmp);
    return data_grab;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t memgrabke_16_BE(uint64_t address)
{

    unsigned char * tmp;
    uint16_t byte1 = 0;
    uint8_t byte2 = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
```

```
    byte1 = byte1 << 8;
    byte1 += byte2;

    return byte1;


}


/****************/
/* Grab 4 bytes */
/****************/
uint32_t memgrabke_32_BE(uint64_t address)
{

    unsigned char * tmp;
    uint32_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);

    byte1 = byte1 << 8;
    byte1 += byte2;
    byte1 = byte1 << 8;
    byte1 += byte3;
    byte1 = byte1 << 8;
    byte1 += byte4;

    return byte1;


}
```

```
/****************/
/* Grab 8 bytes */
/****************/
uint64_t memgrabke_64_BE(uint64_t address)
{

    unsigned char * tmp;
    uint64_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;
    uint8_t byte5 = 0;
    uint8_t byte6 = 0;
    uint8_t byte7 = 0;
    uint8_t byte8 = 0;
    tmp = read_proc_mem (kDTB, address, 1,NO_PROTO);
    byte1 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+1, 1,NO_PROTO);
    byte2 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+2, 1,NO_PROTO);
    byte3 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+3, 1,NO_PROTO);
    byte4 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+4, 1,NO_PROTO);
    byte5 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+5, 1,NO_PROTO);
    byte6 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+6, 1,NO_PROTO);
    byte7 = (uint8_t) * tmp;
    free (tmp);
    tmp = read_proc_mem (kDTB, address+7, 1,NO_PROTO);
    byte8 = (uint8_t) * tmp;
    free (tmp);

    byte1 = byte1 << 8;
    byte1 += byte2;
    byte1 = byte1 << 8;
    byte1 += byte3;
```

```
    byte1 = byte1 << 8;
    byte1 += byte4;
    byte1 = byte1 << 8;
    byte1 += byte5;
    byte1 = byte1 << 8;
    byte1 += byte6;
    byte1 = byte1 << 8;
    byte1 += byte7;
    byte1 = byte1 << 8;
    byte1 += byte8;

    return byte1;


}
```

A.4 pdb_read.c

```
/*******************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)            *
* Developer  : Jimmy Okolica                                       *
* Date       : 15-Aug-2011                                         *
*                                                                  *
*******************************************************************/

/****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community    */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at     */
/*                                                                          */
/* http://www.osedu.org/licenses/ECL-2.0                                    */
/*                                                                          */
/* Unless required by applicable law or agreed to in writing,               */
/* software distributed under the License is distributed on an              */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,             */
/* either express or implied. See the License for the specific              */
/* language governing permissions and limitations under the License.        */
/****************************************************************************/

#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
```

```
#include <stdlib.h>
#include <stddef.h>
//#include <process.h>

#ifndef PDB
#include "pdb_read.h"
#endif

#ifndef FILE_READ
#include "file_read.h"
#endif

#include "math.h"

/********************/
/* GLOBAL VARIABLES */
/********************/
uint8_t DEBUG_PDB=0;
uint8_t DEBUG_INIT=0;
uint8_t DEBUG_LOAD=0;
uint8_t symbol_loc_stream;
short pdb_x64 = 0;

pdb_structures_t *ntoskrnl_pdb;

/*****************************************************************************/

/************************/
/* INITIALIZATION STUFF */
/************************/

/**************************************************/
/* Set a pointer to the kernel metadata structure */
/**************************************************/
void set_ntoskrnl_pdb (pdb_type_t *wtmp) {

    ntoskrnl_pdb = &wtmp->structures;

}


/******************************************/
/* Set whether the dump is 32bit or 64bit */
/******************************************/
void set_pdb_x64(short new_value) {
```

```
    pdb_x64 = new_value;

}


/**************************************************/
/* Set a pointer to the kernel metadata structure */
/**************************************************/
uint8_t have_ntoskrnl_pdb () {

    return (ntoskrnl_pdb != NULL);

}


/*****************************************************************************/

/******************************************************************/
/* RETRIEVE THE PROGRAM DATABASE FROM THE MICROSOFT SYMBOL SERVER */
/******************************************************************/

/*******************************/
/* Helper function for libcurl */
/*******************************/
static size_t write_data(void *ptr, size_t size, size_t nmemb, void *stream) {

    int written = fwrite(ptr, size, nmemb, (FILE *)stream);
    return written;

}


/************************************************************************************/
/* Determine the URL to get the PDB file -- using the virtual address of the kernel */
/************************************************************************************/
char * get_pdb_url(uint64_t wimage_base, int pdb_parm, int is_x64, uint64_t wDTB) {

    ascii_string pdbname;
    char guid[41];
    char * pdb_info;
    uint64_t debug_table,hdr_base, pe_offset, debug_data;
    uint32_t age, guid1,guid4,guid5, debug_signature;
    uint16_t guid2, guid3;
    char *pdb_name, *pdb_name2, *wtmp;
```

```
if (DEBUG_INIT) printf("get_pdb_url image at %llx\n",wimage_base);

// Retrieve the location where the PE offset starts
pe_offset =  (uint64_t) memgrabp_32_LE(wDTB,wimage_base + 0x3c)
          & 0x00000000ffffffff;
if (pe_offset == 0) return NULL;

// Calculate where the PE Header starts
hdr_base = wimage_base + pe_offset + 0x18;
// Find the virtual address of the debug table
debug_table = wimage_base + ((uint64_t) memgrabp_32_LE(wDTB,hdr_base +
              (is_x64?0xa0:0x90)) & 0x00000000ffffffff);

// Check the type of debug data; if it isn't CodeView, where out of luck -
// this shouldn't ever happen
if (memgrabp_32_LE(wDTB,debug_table + 0x0c) != 0x2) {
    // Debug Type = CodeView (used for PDB)
    //       printf("Houston we have a problem.. debug type isn't set up
    // for PDBs\n");
    return NULL;

}


// Get the address of the debug data
debug_data = wimage_base + (memgrabp_32_LE(wDTB,debug_table + 0x14)
           & 0x00000000ffffffff);
if (DEBUG_INIT) printf("debug data at %llx\n",debug_data);

// Check the debug signature -- depending on what version of codeview we've got,
// we reconstruct the GUID and Age differently
debug_signature = memgrabp_32_BE(wDTB,debug_data);
switch (debug_signature) {

    case 0x52534453: // This is current version of the PE files and all I've
                     // had experience
    if (pdb_parm == PDB_RET_NAME) // If all we're looking for is the name,
                                  // return it
    return memgrabp_string0(debug_data + 0x18,wDTB);
    // Otherwise, retrieve the globally unique identifiers and the age
    guid1 = memgrabp_32_LE(wDTB,debug_data + 0x04);
    guid2 = memgrabp_16_LE(wDTB,debug_data + 0x08);
    guid3 = memgrabp_16_LE(wDTB,debug_data + 0x0a);
    guid4 = memgrabp_32_BE(wDTB,debug_data + 0x0c);
```

```
guid5 = memgrabp_32_BE(wDTB,debug_data + 0x10);
age = memgrabp_32_LE(wDTB,debug_data + 0x14);
// Construct the "GUID" we'll use to query the symbol server
sprintf_s(guid,41,"%08x%04x%04x%08x%08x%x",guid1,guid2,guid3,guid4,guid5,age);
// Now get the name of the PDB file
/*
pdbname.Buffer =  debug_data + 0x18;
pdbname.Length = 0;
while (memgrabke_8_LE(pdbname.Buffer + pdbname.Length) != 0)
pdbname.Length++;
pdbname.MaximumLength = pdbname.Length;
pdb_name = memgrabke_string_LE (pdbname);
// Now make a second copy but change .pdb to .pd_ (needed for the url)
pdb_name2 = malloc(sizeof(char) * (strlen(pdb_name) + 1));
strcpy_s(pdb_name2,strlen(pdb_name) + 1,pdb_name);
pdb_name2[pdbname.Length - 1] = '_';
*/
pdb_info = malloc(sizeof(char) * 200);
wtmp = malloc(sizeof(char) * 200);
sprintf_s(wtmp,200,"%s",guid);
strcpy_s(pdb_info,200,wtmp);
free(wtmp);

break;
case 0x4e423130: // below formatting hasn't been tested because I
                 // haven't got an example to test
printf("New Debug Type: %x\n",debug_signature);
// Otherwise, retrieve the globally unique identifiers and the age
guid1 = memgrabp_32_LE(wDTB,debug_data + 0x08);
guid4 = memgrabp_32_LE(wDTB,debug_data + 0x0c);
age = memgrabp_32_LE(wDTB,debug_data + 0x10);
sprintf_s(guid,41,"%08x%08x%08x",guid1,guid4,age);
// Now get the name of the PDB file
pdbname.Buffer =  debug_data + 0x14;
pdbname.Length = 0;
while (memgrabp_8_LE(wDTB,pdbname.Buffer + pdbname.Length) != 0)
pdbname.Length++;
pdbname.MaximumLength = pdbname.Length;
pdb_name = memgrabp_string_LE (pdbname,wDTB); //MACHINE or USERS
// Now make a second copy but change .pdb to .pd_ (needed for the url)
/*
pdb_name2 = malloc(sizeof(char) * (strlen(pdb_name) + 1) );
strcpy_s(pdb_name2,strlen(pdb_name) + 1,pdb_name);
pdb_name2[pdbname.Length - 1] = '_';
*/
```

```
        pdb_info = malloc(sizeof(char) * 200);
        wtmp = malloc(sizeof(char) * 200);
        sprintf_s(wtmp,200,"%s",guid);
        strcpy_s(pdb_info,200,wtmp);
        free(wtmp);
        break;
        default:
        printf("New Debug Type: %x\n",debug_signature);
        pdb_info = NULL;
        break;

    }

    return pdb_info;

}


/*************************************************************************************/
/* Determine the URL to get the PDB file -- using the physical address of the kernel */
/*************************************************************************************/
char * pget_pdb_url(uint64_t wimage_base, int pdb_parm, int is_x64) {

    ascii_string pdbname;
    char guid[41];
    char * pdb_info;
    uint64_t debug_table,hdr_base, pe_offset, debug_data;
    uint32_t age, guid1,guid4,guid5, debug_signature;
    uint16_t guid2, guid3;
    char *pdb_name, *pdb_name2, *wtmp;

    if (DEBUG_INIT) printf("pget_pdb_url  image at %llx\n",wimage_base);
    // Retrieve the location where the PE offset starts
    pe_offset =  (uint64_t) memgrab_32_LE(wimage_base + 0x3c) & 0x00000000ffffffff;
    // Calculate where the PE Header starts
    hdr_base = wimage_base + pe_offset + 0x18;
    // Find the virtual address of the debug table
    debug_table = wimage_base + ((uint64_t) memgrab_32_LE(hdr_base +
                  (is_x64?0xa0:0x90)) & 0x00000000ffffffff);
        // Check the type of debug data; if it isn't CodeView, where out of luck
        // - this shouldn't ever happen
    if (memgrab_32_LE(debug_table + 0x0c) != 0x2) {
        // Debug Type = CodeView (used for PDB)
        return NULL;
```

```
}


// Get the address of the debug data
debug_data = wimage_base + (memgrab_32_LE(debug_table + 0x14)
          & 0x00000000ffffffff);
if (DEBUG_INIT) printf("debug data at %llx\n",debug_data);

// Check the debug signature -- depending on what version of codeview we've got,
// we reconstruct the GUID and Age differently
debug_signature = memgrab_32_BE(debug_data);
switch (debug_signature) {

    case 0x52534453: // Current version of the PE files and all I've had
                     // experience with testing
    if (pdb_parm == PDB_RET_NAME) // If all we're looking for is the name,
                                  // return it
    return memgrab_string0(debug_data + 0x18);
    // Otherwise, retrieve the globally unique identifiers and the age
    guid1 = memgrab_32_LE(debug_data + 0x04);
    guid2 = memgrab_16_LE(debug_data + 0x08);
    guid3 = memgrab_16_LE(debug_data + 0x0a);
    guid4 = memgrab_32_BE(debug_data + 0x0c);
    guid5 = memgrab_32_BE(debug_data + 0x10);
    age = memgrab_32_LE(debug_data + 0x14);
    // Construct the "GUID" we'll use to query the symbol server
    sprintf_s(guid,41,"%08x%04x%04x%08x%08x%x",guid1,guid2,guid3,guid4,guid5,age);
    /*
    // Now get the name of the PDB file
    pdbname.Buffer =  debug_data + 0x18;
    pdbname.Length = 0;
    while (memgrab_8_LE(pdbname.Buffer + pdbname.Length) != 0)
    pdbname.Length++;
    pdbname.MaximumLength = pdbname.Length;
    pdb_name = memgrab_string_LE (pdbname); //MACHINE or USERS
    // Now make a second copy but change .pdb to .pd_ (needed for the url)
    pdb_name2 = malloc(sizeof(char) * (strlen(pdb_name) + 1));
    strcpy_s(pdb_name2,strlen(pdb_name) + 1,pdb_name);
    pdb_name2[pdbname.Length - 1] = '_';
    */
    pdb_info = malloc(sizeof(char) * 200);
    wtmp = malloc(sizeof(char) * 200);
    sprintf_s(wtmp,200,"%s",guid);
    strcpy_s(pdb_info,200,wtmp);
    free(wtmp);
```

```
        break;
        case 0x4e423130: // below formatting hasn't been tested because I
                         // haven't got an example to test
        printf("New Debug Type: %x\n",debug_signature);
        // Otherwise, retrieve the globally unique identifiers and the age
        guid1 = memgrab_32_LE(debug_data + 0x08);
        guid4 = memgrab_32_LE(debug_data + 0x0c);
        age = memgrab_32_LE(debug_data + 0x10);
        // Construct the "GUID" we'll use to query the symbol server
        sprintf_s(guid,41,"%08x%08x%08x",guid1,guid4,age);
        // Now get the name of the PDB file
        pdbname.Buffer =  debug_data + 0x14;
        pdbname.Length = 0;
        while (memgrab_8_LE(pdbname.Buffer + pdbname.Length) != 0)
        pdbname.Length++;
        pdbname.MaximumLength = pdbname.Length;
        pdb_name = memgrab_string_LE (pdbname); //MACHINE or USERS
        // Now make a second copy but change .pdb to .pd_ (needed for the url)
        /*
        pdb_name2 = malloc(sizeof(char) * (strlen(pdb_name) + 1) );
        strcpy_s(pdb_name2,strlen(pdb_name) + 1,pdb_name);
        pdb_name2[pdbname.Length - 1] = '_';
        */
        pdb_info = malloc(sizeof(char) * 200);
        wtmp = malloc(sizeof(char) * 200);
        sprintf_s(wtmp,200,"%s",guid);
        strcpy_s(pdb_info,200,wtmp);
        free(wtmp);
        break;
        default:
        printf("New Debug Type: %x\n",debug_signature);
        break;

    }

    return pdb_info;

}


/*********************************************************************************/
/* Determine the URL to get the PDB file -- using the virtual address of the kernel */
/*********************************************************************************/
char * get_pdb_url_disk(char *file_name, int pdb_parm, int is_x64) {
```

```
ascii_string pdbname;
char guid[41];
char * pdb_info;
uint32_t debug_table,hdr_base, pe_offset, debug_data;
uint32_t age, guid1,guid4,guid5, debug_signature;
uint16_t guid2, guid3;
char *pdb_name, *pdb_name2, *wtmp;
FILE *wfile;
uint32_t i, nbr_sections, vsize, vaddr, rsize, raddr;

fopen_s(&wfile, file_name, "rb");
rewind(wfile);

// Retrieve the location where the PE offset starts
fseek(wfile,0x3c,SEEK_SET);
pe_offset =  fgrab_32_LE(wfile);
// Calculate where the PE Header starts
hdr_base = pe_offset + 0x18;
// Find the virtual address of the debug table
fseek(wfile,hdr_base + (is_x64?0xa0:0x90),SEEK_SET);
debug_table = fgrab_32_LE(wfile);

fseek(wfile,hdr_base + 0x02,SEEK_SET);
nbr_sections = fgrab_32_LE(wfile);
for (i = 0; i < nbr_sections; i++) {

    fseek(wfile,hdr_base + (is_x64?0xf0:0xe0) + i * 0x28 + 8, SEEK_SET);
    vsize = fgrab_32_LE(wfile);
    vaddr = fgrab_32_LE(wfile);
    rsize = fgrab_32_LE(wfile);
    raddr = fgrab_32_LE(wfile);
    if ((debug_table > vaddr) && (debug_table < vaddr + vsize)) {

        debug_table = debug_table - vaddr + raddr;
        break;

    }


}


//    printf("revised debug_table = %x\n",debug_table);
```

```
// Check the type of debug data; if it isn't CodeView, where out of luck -
// this shouldn't ever happen
fseek(wfile,debug_table + 0x0c,SEEK_SET);
if (fgrab_32_LE(wfile) != 0x2) {
    // Debug Type = CodeView (used for PDB)
    //        fclose(wfile);
    return NULL;

}


// Get the address of the debug data
fseek(wfile,debug_table + 0x14,SEEK_SET);
debug_data = fgrab_32_LE(wfile);

fseek(wfile,hdr_base + 0x02,SEEK_SET);
nbr_sections = fgrab_32_LE(wfile);
for (i = 0; i < nbr_sections; i++) {

    fseek(wfile,hdr_base + (is_x64?0xf0:0xe0) + i * 0x28 + 8, SEEK_SET);
    vsize = fgrab_32_LE(wfile);
    vaddr = fgrab_32_LE(wfile);
    rsize = fgrab_32_LE(wfile);
    raddr = fgrab_32_LE(wfile);
    if ((debug_data > vaddr) && (debug_data < vaddr + vsize)) {

        debug_data = debug_data - vaddr + raddr;
        break;

    }


}


// Check the debug signature -- depending on what version of codeview we've got,
// we reconstruct the GUID and Age differently
fseek(wfile,debug_data,SEEK_SET);
debug_signature = fgrab_32_BE(wfile);
switch (debug_signature) {

    case 0x52534453: // Current version of the PE files and all I've had
                     // experience with testing
    if (pdb_parm == PDB_RET_NAME) {
        // If all we're looking for is the name, return it
```

```
    fseek(wfile,debug_data + 0x18,SEEK_SET);
    return fgrab_string0(wfile);

}

// Otherwise, retrieve the globally unique identifiers and the age
fseek(wfile,debug_data + 0x04,SEEK_SET);
guid1 = fgrab_32_LE(wfile);
guid2 = fgrab_16_LE(wfile);
guid3 = fgrab_16_LE(wfile);
guid4 = fgrab_32_BE(wfile);
guid5 = fgrab_32_BE(wfile);
age = fgrab_32_LE(wfile);
// Construct the "GUID" we'll use to query the symbol server
sprintf_s(guid,41,"%08x%04x%04x%08x%08x%x",guid1,guid2,guid3,guid4,guid5,age);
// Now get the name of the PDB file
/*
pdbname.Buffer =  debug_data + 0x18;
pdbname.Length = 0;
while (memgrabke_8_LE(pdbname.Buffer + pdbname.Length) != 0)
pdbname.Length++;
pdbname.MaximumLength = pdbname.Length;
pdb_name = memgrabke_string_LE (pdbname);
// Now make a second copy but change .pdb to .pd_ (needed for the url)
pdb_name2 = malloc(sizeof(char) * (strlen(pdb_name) + 1));
strcpy_s(pdb_name2,strlen(pdb_name) + 1,pdb_name);
pdb_name2[pdbname.Length - 1] = '_';
*/
pdb_info = malloc(sizeof(char) * 200);
wtmp = malloc(sizeof(char) * 200);
sprintf_s(wtmp,200,"%s",guid);
strcpy_s(pdb_info,200,wtmp);
free(wtmp);

break;
case 0x4e423130: // Below formatting hasn't been tested because I
                 // haven't got an example to test
printf("New Debug Type: %x\n",debug_signature);
// Otherwise, retrieve the globally unique identifiers and the age
fseek(wfile,debug_data + 0x08,SEEK_SET);
guid1 = fgrab_32_LE(wfile);
guid4 = fgrab_32_LE(wfile);
age = fgrab_32_LE(wfile);
sprintf_s(guid,41,"%08x%08x%08x",guid1,guid4,age);
// Now get the name of the PDB file
```

```
        pdbname.Buffer =  debug_data + 0x14;
        pdbname.Length = 0;
        fseek(wfile,(uint32_t) pdbname.Buffer,SEEK_SET);
        while (fgrab_8_LE(wfile) != 0)
        pdbname.Length++;
        pdbname.MaximumLength = pdbname.Length;
        pdb_name = fgrab_string_LE (wfile,pdbname); //MACHINE or USERS
        // Now make a second copy but change .pdb to .pd_ (needed for the url)
        /*
        pdb_name2 = malloc(sizeof(char) * (strlen(pdb_name) + 1) );
        strcpy_s(pdb_name2,strlen(pdb_name) + 1,pdb_name);
        pdb_name2[pdbname.Length - 1] = '_';
        */
        pdb_info = malloc(sizeof(char) * 200);
        wtmp = malloc(sizeof(char) * 200);
        sprintf_s(wtmp,200,"%s",guid);
        strcpy_s(pdb_info,200,wtmp);
        free(wtmp);
        break;
        default:
        printf("New Debug Type: %x\n",debug_signature);
        break;

    }

    fclose(wfile);
    return pdb_info;

}


/*******************************************************************************/

/*************************************/
/* LOAD THE METADATA FROM THE PDB FILE */
/*************************************/

/************************************/
/* Download the PDB file and parse it */
/************************************/
char * download_pdb_file (char *wfile_name, char *wpdb_url, char *DATADIRECTORY) {

    // Reference: http://www.informit.com/articles/article.aspx?p=22685

    CURL *curl;
```

```
FILE *data_file;
CURLcode res;
uint32_t i;
/*
pdb_type_t *pdb;
uint32_t wunknown, wnbr_pages,i,j;
uint8_t data_from_memdump1;
uint16_t rec_size;
uint8_t section_stream,symbol_stream,structure_stream;
char * chk_section;
uint32_t chk_struct;
uint16_t sym_type;
*/
char *wfile_name2;    // The PDB name with the .pdb extension changed to .pd_
char *xpdb_url;        // The URL line needed to download the PDB file
char *xcmd;            // The command needed to decompress the PDB file
char *xname, *xname2;

xname = malloc(sizeof(char) * (strlen(wfile_name ) + strlen(DATADIRECTORY) + 1));
strcpy_s(xname,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
strcat_s(xname,strlen(DATADIRECTORY) + strlen(wfile_name) + 1,wfile_name);

xpdb_url = malloc(sizeof(char) * 200);
xcmd = malloc(sizeof(char) * 200);
for (i = 0; i < 200; i++) xcmd[i] = 0;
wfile_name2 = (char *) malloc(sizeof(char) * (strlen(wfile_name) + 1));
strcpy_s(wfile_name2,strlen(wfile_name) + 1,wfile_name);
wfile_name2[strlen(wfile_name) - 1] = '_'; // the PDB name with the .pdb
                                           // extension change to .pd_

xname2 = malloc(sizeof(char) * (strlen(wfile_name2 ) + strlen(DATADIRECTORY) + 1));
strcpy_s(xname2,strlen(DATADIRECTORY) + 1,DATADIRECTORY);
strcat_s(xname2,strlen(DATADIRECTORY) + strlen(wfile_name2) + 1,
                    wfile_name2);

printf("Loading %s with GUID %s into %s\n",wfile_name,wpdb_url,xname);

// Download the PDB file
curl = curl_easy_init();
//    if ((curl) && (strcmp(wfile_name,wpdb_url))) {

    if (curl) {

        fopen_s(&data_file, xname2,"wb");
        curl_easy_setopt(curl,
```

```
            CURLOPT_USERAGENT,"Microsoft-Symbol-Server/6.6.0007.5");
    sprintf_s(xpdb_url,200,
            "http://msdl.microsoft.com/download/symbols/%s/%s/%s",
            wfile_name,wpdb_url,wfile_name2);
    curl_easy_setopt(curl, CURLOPT_URL, xpdb_url);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, data_file);
    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);
    res = curl_easy_perform(curl);

    /* always cleanup */
    curl_easy_cleanup(curl);
    if(CURLE_OK != res) {

        /* we failed */
        fprintf(stderr, "curl told us %d\n", res);

    }

    fclose(data_file);
    // Decompress the PDB file
    //        sprintf(xcmd,"cabextract %s >/dev/null 2>/dev/null",wfile_name2);
    sprintf_s(xcmd,200,"expand %s %s >nul 2>nul",xname2,xname); // Windows
    printf("%s\n",xcmd);
    system(xcmd);
    free(xcmd);
    free(xname2);
    free(xpdb_url);
    free(wfile_name2);
    return xname;

  }

  else
  return NULL;

}


/*********************/
/* Parse the PDB file */
/*********************/
pdb_type_t * parse_pdb_file (char *wfile_name, int parm) {
```

```
pdb_type_t *pdb;
uint32_t wunknown, wnbr_pages,i,j,k;
uint8_t data_from_memdump1;
uint16_t rec_size;
uint8_t section_stream,symbol_stream,structure_stream;
char * chk_section;
uint32_t chk_struct;
uint16_t sym_type;

uint32_t wreloc, wdata, wcurrent_page;


// Begin to build the PDB structure
pdb = malloc(sizeof *pdb);
pdb->file_name = wfile_name;

// Open the PDB file
if (DEBUG_INIT) printf("filename = %s\n",pdb->file_name);
fopen_s(&pdb->pdb_file.file_ptr, pdb->file_name,"rb");
rewind (pdb->pdb_file.file_ptr);

// Retrieve the header information
// The beginning of the file is a eof-terminated header string that
// includes the compiler and the PDB version
while ( (data_from_memdump1 = fgrab_8_LE(pdb->pdb_file.file_ptr)) != 0x1a);
pdb->magic_number = fgrab_32_BE(pdb->pdb_file.file_ptr);
fseek(pdb->pdb_file.file_ptr,0x20,SEEK_SET);
pdb->bytes_per_page = fgrab_32_LE(pdb->pdb_file.file_ptr);
pdb->start_page = fgrab_32_LE(pdb->pdb_file.file_ptr);
pdb->nbr_pages = fgrab_32_LE(pdb->pdb_file.file_ptr);
pdb->root_size = fgrab_32_LE(pdb->pdb_file.file_ptr);
wunknown = fgrab_32_LE(pdb->pdb_file.file_ptr);
pdb->root_ptr = fgrab_32_LE(pdb->pdb_file.file_ptr);

// Goto the root stream (the root directory of the file system)
// and get the # of streams and their size
pdb_goto_stream(pdb,0,0); // Stream 0 (root) Page 0
pdb->stream_count = pdb_grab_32_LE(pdb);
pdb->stream_size = malloc(sizeof(uint32_t) * pdb->stream_count);
for (i = 0; i < pdb->stream_count; i++) {

    pdb->stream_size[i] = pdb_grab_32_LE(pdb);

}
```

```
// Now get the pages associated with each stream
pdb->stream_ptr = malloc(sizeof(uint32_t) * pdb->stream_count);
for (i = 0; i < pdb->stream_count; i++) {

    if (pdb->stream_size[i] > 0)
      pdb->stream_ptr[i] = (pdb->pdb_file.current_page * pdb->bytes_per_page)
                         + pdb->pdb_file.byte_count;
    else
      pdb->stream_ptr[i] = 0;
    wnbr_pages = (int) ((double) pdb->stream_size[i] /
                        (double) pdb->bytes_per_page);
    if ( (pdb->stream_size[i] % pdb->bytes_per_page) != 0) wnbr_pages++;
    for (j = 0; j < wnbr_pages; j++)  {

        pdb_grab_32_LE(pdb);

    }


}

if (DEBUG_LOAD) printf("Stream Count = %i\n",pdb->stream_count);

section_stream = 0;
structure_stream = 0;
symbol_stream = 0;

// Scroll through each stream (since we don't know which stream contains
// what info)
for (i = 0; i < pdb->stream_count; i++) {

    pdb_goto_stream(pdb,i + 1,0);

    // Check to see if it's the section stream
    chk_section = pdb_grab_string(pdb, 6);
    if ( (!strncmp(chk_section,".data",5) || !strncmp(chk_section,".text",5))
              && section_stream == 0)
      section_stream = (uint8_t) (i + 1);

    // Check if it's the structure stream
    pdb_goto_stream(pdb,i + 1,0);
    pdb_grab_32_LE(pdb);
    chk_struct = pdb_grab_32_LE(pdb);
    if ( (chk_struct == 0x38) && (structure_stream == 0))
```

```
    structure_stream = (uint8_t) (i + 1);

    // Check if it's the symbol stream
    pdb_goto_stream(pdb,i + 1,0);
    rec_size = pdb_grab_16_LE(pdb);
    sym_type = pdb_grab_16_LE(pdb);
    if ( ((rec_size < 0x100) && (sym_type == 0x110e)) && (symbol_stream == 0))
    symbol_stream = (uint8_t) (i + 1);

}

symbol_loc_stream = section_stream + 1;
// Symbols
// Go through the symbols once to count the number of symbols
pdb_goto_stream(pdb, symbol_stream, 0); // Stream 17, Page 0
pdb->symbols.count = 0;
while (!pdb_eof_stream(pdb)) {

    rec_size = pdb_grab_16_LE(pdb);
    if (rec_size == 0x0) break;
    pdb->symbols.count = pdb->symbols.count + 1;
    for (i = 0; i < rec_size; i++) pdb_grab_8_LE(pdb);

}

// Allocate space for all of the symbols
pdb->symbols.table = malloc(sizeof(pdb_symbol_t) * pdb->symbols.count);
pdb_goto_stream(pdb, symbol_stream, 0); // Stream 17, Page 0
if (DEBUG_LOAD) printf("Symbols:\n");
if (parm == PDB_DUMP_SYM) printf("Symbols:\n");
// Load the symbol information
for (i = 0; i < pdb->symbols.count; i++) {

    rec_size = pdb_grab_16_LE(pdb);
    if (rec_size == 0x0) {

        pdb->symbols.table[i].name     = malloc(sizeof(char) * 2);
        for (k = 0; k < 2; k++) pdb->symbols.table[i].name[k] = 0;
        pdb->symbols.table[i].namelen  = 0;
        continue;

    }

    pdb->symbols.table[i].unknown1 = pdb_grab_16_LE(pdb); //?
    pdb->symbols.table[i].unknown2 = pdb_grab_32_LE(pdb); //?
```

```
    pdb->symbols.table[i].offset   = pdb_grab_32_LE(pdb);
    pdb->symbols.table[i].type     = pdb_grab_16_LE(pdb);
    pdb->symbols.table[i].namelen  = rec_size - 0x0c;
    pdb->symbols.table[i].name   = malloc(sizeof(char) *
                                (pdb->symbols.table[i].namelen + 1));
    for (j = 0; j < pdb->symbols.table[i].namelen; j++)
    pdb->symbols.table[i].name[j] = pdb_grab_8_LE(pdb);
    if (parm == PDB_DUMP_SYM)
      printf("%40s\tOffset: %x\tType: %x\tUnknown:%x\tUnknown:%x\n",
                     pdb->symbols.table[i].name,
                     pdb->symbols.table[i].offset,
                     pdb->symbols.table[i].type,
                     pdb->symbols.table[i].unknown1,
                     pdb->symbols.table[i].unknown2);

}


// Sections
pdb_goto_stream(pdb, section_stream, 0); // Stream 11, Page 0
pdb->sections.count = 0;
// First count the number of sections
while (!pdb_eof_stream(pdb)) {

    wunknown = pdb_grab_32_LE(pdb);
    if (wunknown == 0x0) break;
    for (i = 0; i < 9; i++) pdb_grab_32_LE(pdb);
    pdb->sections.count++;

}

// Allocate space for the sections
pdb->sections.table = malloc(sizeof(pdb_section_t) * pdb->sections.count);
pdb_goto_stream(pdb, section_stream, 0); // Stream 11, Page 0
// Load section information
for (i = 0; i < pdb->sections.count; i++) {

    pdb->sections.table[i].name = malloc(sizeof(char) * 9);
    for (j = 0; j < 8; j++)
      pdb->sections.table[i].name[j] = pdb_grab_8_LE(pdb);
    pdb->sections.table[i].virtual_size = pdb_grab_32_LE(pdb);
    pdb->sections.table[i].virtual_address = pdb_grab_32_LE(pdb);
    pdb->sections.table[i].raw_size = pdb_grab_32_LE(pdb);
    pdb->sections.table[i].raw_ptr = pdb_grab_32_LE(pdb);
    pdb->sections.table[i].relocation_ptr = pdb_grab_32_LE(pdb);
```

```
    pdb->sections.table[i].line_ptr = pdb_grab_32_LE(pdb);
    pdb->sections.table[i].relocation_count = pdb_grab_16_LE(pdb);
    pdb->sections.table[i].line_count = pdb_grab_16_LE(pdb);
    pdb->sections.table[i].characteristics = pdb_grab_32_LE(pdb);

}


// Structures
pdb_goto_stream(pdb, structure_stream, 0); // Stream 3, Page 0
pdb->structures.version = pdb_grab_32_LE(pdb);
pdb->structures.header_size = pdb_grab_32_LE(pdb);
pdb->structures.base_index = pdb_grab_32_LE(pdb);
pdb->structures.count = 0;
// First count the number of structures
for (i = 0; i < pdb->structures.header_size - 0xc; i++) pdb_grab_8_LE(pdb);
while (!pdb_eof_stream(pdb)) {

    rec_size = pdb_grab_16_LE(pdb);
    if (rec_size == 0x0) break;
    pdb->structures.count = pdb->structures.count + 1;
    for (i = 0; i < rec_size; i++) pdb_grab_8_LE(pdb);

}

// Allocate space for all of the structures
pdb->structures.table = malloc(sizeof(pdb_structure_t) *
                        pdb->structures.count);
pdb_goto_stream(pdb, structure_stream, 0); // Stream 3 Page 0
for (i = 0; i < pdb->structures.header_size; i++) pdb_grab_8_LE(pdb);
if (DEBUG_LOAD) printf("Structures:\n");
if (parm == PDB_DUMP_STRUCT) printf("Structures:\n");
// Load the structure information
for (i = 0; i < pdb->structures.count; i++) {

    rec_size = pdb_grab_16_LE(pdb);
    pdb->structures.table[i] = get_pdb_struct(pdb,rec_size);
    pdb->structures.table[i].index = pdb->structures.base_index + i;
    if (parm == PDB_DUMP_STRUCT)
      print_pdb_struct(pdb->structures.table, &pdb->structures.table[i],
                       pdb->structures.base_index, 0);

}

//    fclose(pdb->pdb_file.file_ptr);
```

```
    return pdb;

}


/*****************************************************************************/

/*******************************/
/* READS DATA FROM THE PDB FILE */
/*******************************/
// assumes a given read won't span pages within a stream -- needs to be made
// more robust later
// Struct keeps track of current current stream, current page, bytes read in
// page, and bytes count (read) in stream

/* file movement */

void pdb_goto_stream(pdb_type_t *pdb,uint32_t stream_nbr, uint32_t page_nbr) {

    if (stream_nbr == 0) // Root
    fseek(pdb->pdb_file.file_ptr, pdb->root_ptr * pdb->bytes_per_page
                 + page_nbr * 4, SEEK_SET);
    else
    fseek(pdb->pdb_file.file_ptr, pdb->stream_ptr[stream_nbr - 1] + page_nbr
                 * 4, SEEK_SET);
    pdb->pdb_file.current_stream = stream_nbr;
    pdb->pdb_file.current_page = fgrab_32_LE(pdb->pdb_file.file_ptr);
    pdb->pdb_file.offset = page_nbr;
    fseek(pdb->pdb_file.file_ptr, pdb->pdb_file.current_page *
                 pdb->bytes_per_page, SEEK_SET);
    pdb->pdb_file.byte_count = 0;
    if (page_nbr == 0)
    pdb->pdb_file.bytes_read = 0;

}



pdb_file_t pdb_set_point(pdb_type_t *pdb) {

    pdb_file_t wpdb_file;

    wpdb_file.byte_count = pdb->pdb_file.byte_count;
    wpdb_file.bytes_read = pdb->pdb_file.bytes_read;
```

```
    wpdb_file.current_stream = pdb->pdb_file.current_stream;
    wpdb_file.current_page = pdb->pdb_file.current_page;
    wpdb_file.offset = pdb->pdb_file.offset;

    return wpdb_file;

}


void pdb_goto_point(pdb_type_t *pdb, pdb_file_t wpdb_file) {


    pdb->pdb_file.byte_count = wpdb_file.byte_count;
    pdb->pdb_file.bytes_read = wpdb_file.bytes_read;
    pdb->pdb_file.current_stream = wpdb_file.current_stream;
    pdb->pdb_file.current_page = wpdb_file.current_page;
    pdb->pdb_file.offset = wpdb_file.offset;
    fseek(pdb->pdb_file.file_ptr, pdb->pdb_file.current_page *
               pdb->bytes_per_page + pdb->pdb_file.byte_count, SEEK_SET);

}


int pdb_eof_stream(pdb_type_t *pdb) {

    if (pdb->pdb_file.current_stream == 0)
    return (pdb->pdb_file.bytes_read >= pdb->root_size);
    else
    return (pdb->pdb_file.bytes_read >=
            pdb->stream_size[pdb->pdb_file.current_stream - 1]);

}


/**************************************************/
/* Returns the bytes read in the current stream */
/**************************************************/
uint32_t pdb_get_bytes_read(pdb_type_t *pdb) {

    return pdb->pdb_file.bytes_read;

}


/****************/
```

```
/* Grab 4 bytes */
/****************/
uint32_t pdb_grab_32_LE(pdb_type_t *pdb) {

    uint32_t retval;

    if (feof(pdb->pdb_file.file_ptr)) return 0;
    retval = fgrab_32_LE(pdb->pdb_file.file_ptr);
    pdb->pdb_file.byte_count += 4;
    pdb->pdb_file.bytes_read += 4;
    if (pdb->pdb_file.byte_count >= pdb->bytes_per_page) {

        pdb_goto_stream(pdb,pdb->pdb_file.current_stream,pdb->pdb_file.offset + 1);

    }


    return retval;

}


/****************/
/* Grab 2 bytes */
/****************/
uint16_t pdb_grab_16_LE(pdb_type_t *pdb) {

    uint16_t retval;

    retval = fgrab_16_LE(pdb->pdb_file.file_ptr);
    pdb->pdb_file.byte_count += 2;
    pdb->pdb_file.bytes_read += 2;
    if (pdb->pdb_file.byte_count >= pdb->bytes_per_page) {

        pdb_goto_stream(pdb,pdb->pdb_file.current_stream,pdb->pdb_file.offset + 1);

    }


    return retval;

}


/***************/
```

```
/* Grab 1 byte */
/***************/
uint8_t pdb_grab_8_LE(pdb_type_t *pdb) {

    uint8_t retval;

    retval = fgrab_8_LE(pdb->pdb_file.file_ptr);
    pdb->pdb_file.byte_count += 1;
    pdb->pdb_file.bytes_read += 1;
    if (pdb->pdb_file.byte_count >= pdb->bytes_per_page) {

        pdb_goto_stream(pdb,pdb->pdb_file.current_stream,pdb->pdb_file.offset + 1);

    }


    return retval;

}


/*****************/
/* Grab a String  */
/*****************/
char * pdb_grab_string(pdb_type_t *pdb, uint32_t wsize) {

    pdb_file_t wpdb_file;
    char *wname;
    uint32_t j;
    uint8_t x;
    uint32_t xsize;
    xsize = wsize;

    wpdb_file = pdb_set_point(pdb);

    // If no size was given, find the first 0 byte to figure out the size
    // Note: If the the first byte is zero, that means the string is size 1
    if (xsize == 0) {

        while (x = pdb_grab_8_LE(pdb)) wsize++;
        if (wsize == 0) wsize = 1;
        wsize++;

    }
```

```
    // Retrieve the string
    wname = malloc(sizeof(char) * (wsize + 1));
    pdb_goto_point(pdb,wpdb_file);
    for (j = 0; j < wsize; j++) {

        wname[j] = pdb_grab_8_LE(pdb);

    }


    // If the size wasn't specified, there may be space fillers (f1,f2,f3)
    // at the end so move over that
    if (xsize == 0) {

        wpdb_file = pdb_set_point(pdb);
        x = pdb_grab_8_LE(pdb);
        while ( (x == 0xf1) || (x == 0xf2) || (x == 0xf3)) {

            wpdb_file = pdb_set_point(pdb);
            x = pdb_grab_8_LE(pdb);

        }

        pdb_goto_point(pdb,wpdb_file);

    }


    return wname;

}



/************************************/
/* Grab 1 byte from a physical file */
/************************************/
uint8_t fgrab_8_LE(FILE *image_file)
{

    uint8_t data_grab = 0;
    fread (&data_grab, 1, 1, image_file);
    return data_grab;
```

```
}


/****************************************************/
/* Grab 4 bytes from a physical file in Little Endian */
/****************************************************/
uint32_t fgrab_32_LE(FILE *image_file)
{

    uint8_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint32_t byte4 = 0;

    fread(&byte1, 1, 1, image_file);
    fread(&byte2, 1, 1, image_file);
    fread(&byte3, 1, 1, image_file);
    fread(&byte4, 1, 1, image_file);

    byte4 = byte4 << 8;
    byte4 += byte3;
    byte4 = byte4 << 8;
    byte4 += byte2;
    byte4 = byte4 << 8;
    byte4 += byte1;

    return byte4;

}


/****************************************************/
/* Grab 2 bytes from a physical file in Little Endian */
/****************************************************/
uint16_t fgrab_16_LE(FILE *image_file)
{

    uint8_t byte1 = 0;
    uint16_t byte2 = 0;
    fread(&byte1, 1, 1, image_file);
    fread(&byte2, 1, 1, image_file);
    byte2 = byte2 << 8;
    byte2 += byte1;

    return byte2;
```

```
}


/**************************************************/
/* Grab 4 bytes from a physical file in Big Endian */
/**************************************************/
uint32_t fgrab_32_BE(FILE *image_file)
{

    uint32_t byte1 = 0;
    uint8_t byte2 = 0;
    uint8_t byte3 = 0;
    uint8_t byte4 = 0;

    fread(&byte1, 1, 1, image_file);
    fread(&byte2, 1, 1, image_file);
    fread(&byte3, 1, 1, image_file);
    fread(&byte4, 1, 1, image_file);

    byte1 = byte1 << 8;
    byte1 += byte2;
    byte1 = byte1 << 8;
    byte1 += byte3;
    byte1 = byte1 << 8;
    byte1 += byte4;

    return byte1;

}


/***********************/
/* Grab an ASCII String */
/***********************/
char * fgrab_string_LE (FILE* wfile, ascii_string str)
{

    int i;
    char *rstr = malloc(sizeof(char) * (str.Length + 1));

    for (i=0; i < str.Length + 1; i++) rstr[i] = 0;

    fseek(wfile,(uint32_t) str.Buffer,SEEK_SET);
    for (i=0; i < str.Length; i++)
```

```
    {

        rstr[i] = fgrab_8_LE(wfile);

    }

    return rstr;

}


/************************************************************/
/* Grab an ASCII String that is guaranteed to be NULL TERMINATED */
/************************************************************/
// If not null-terminated it retrieves the first 50 characters
char * fgrab_string0(FILE *wfile)
{

    int i, wlength;
    char *rstr;
    uint8_t tmp;
    uint32_t cur_pos;
    cur_pos = ftell(wfile);
    for (wlength=0; wlength < 50; wlength++)
    {

        tmp = fgrab_8_LE(wfile);
        if (tmp == 0) break;

    }

    fseek(wfile,cur_pos,SEEK_SET);
    rstr = malloc(sizeof(char) * (wlength + 1));

    for (i=0; i < wlength + 1; i++) rstr[i] = 0;

    for (i=0; i < wlength; i++)
    {

        rstr[i] = fgrab_8_LE(wfile);

    }

    return rstr;
```

```
}


/*****************************************************************************/

/******************************/
/* RETRIEVE A REQUESTED SECTION */
/******************************/
pdb_section_t * get_pdb_section(pdb_type_t *pdb, char *wname, uint32_t wlength) {


    uint8_t i;
    char chkit[8];

    for (i = 0; i < 8; i++) chkit[i] = 0;
    if (wlength > 8) wlength = 8;
    for (i = 0; i < wlength; i++) chkit[i] = wname[i];

    for (i = 0; i < pdb->sections.count; i++) {

        if (!strncmp(pdb->sections.table[i].name,chkit,8))
        return &pdb->sections.table[i];

    }

    return NULL;

}


/*****************************************************************************/

/******************************/
/* RETRIEVE A REQUESTED SYMBOL */
/******************************/

/******************************/
/* Retrieve a requested symbol */
/******************************/
pdb_symbol_t * get_pdb_symbol(pdb_type_t *pdb, char *wname, uint32_t wlength) {


    uint32_t i;
```

```
        for (i = 0; i < pdb->symbols.count; i++) {

            if (pdb->symbols.table[i].namelen >= wlength) {

                if (!strncmp(pdb->symbols.table[i].name,wname,wlength) &&
                    (pdb->symbols.table[i].name[wlength] == 0) ) {

                    return &pdb->symbols.table[i];

                }


            }


        }

        return NULL;

    }


    /******************************************************************************/
/* Adjust location of symbol to reflect where it was placed during compilation*/
/******************************************************************************/
// If no relocation is found, just add the location of the "data" (e.g., .data,
// .idata, .rdata, ...) section to the offset
    uint32_t get_pdb_symbol_loc(pdb_type_t *pdb, uint32_t woffset, uint32_t
                                wdata_section) {

        uint32_t wreloc,wdata;
        int  i;

        pdb_goto_stream(pdb, symbol_loc_stream, 0); // Stream 12, Page 0
        i = 0;
        while (!pdb_eof_stream(pdb)) {

            wreloc = pdb_grab_32_LE(pdb);
            wdata = pdb_grab_32_LE(pdb);
            if (wdata == (woffset + wdata_section)) {

                return wreloc;

            }
```

```
    }

    return wdata_section + woffset;

}


/******************************************************************************/
/**********************************/
/* RETRIEVE A STRUCTURE INFORMATION */
/**********************************/
/* all records are of the basic format [0x00 rec_size 0x02 data] */
/* rec_size is retrieved prior to calling this function          */

/*********************************************/
/* Retrieve a single structure from the PDB file */
/*********************************************/
// This is used up above in load_pdb_file
pdb_structure_t get_pdb_struct(pdb_type_t *pdb, uint16_t rec_size) {

    pdb_structure_t wfld,wwfld;
    pdb_file_t wpdb_file;
    uint32_t x,j;

    wfld.index = 0;
    wfld.data_type = pdb_grab_16_LE(pdb);
    wfld.element_count = 0;
    wfld.properties = 0;
    wfld.derived = 0;
    wfld.vshape = 0;
    wfld.size = 0;
    wfld.name = NULL;
    wfld.underlying_type = 0;
    wfld.pointer_array = 0;
    wfld.offset = 0;
    wfld.field_index = 0;
    wfld.arguments = NULL;
    wfld.retval_type = 0;
    wfld.call_type = 0;
    wfld.value = 0;
    wfld.index_type = 0;
    wfld.field_count = 0;
    wfld.field_list = NULL;
```

```
switch (wfld.data_type) {

    case LF_FIELDLIST    : // field list -- does not handle nested field lists
                           //            -- don't think these occur
    wpdb_file = pdb_set_point(pdb);
    x = pdb_get_bytes_read(pdb) - 0x2;
    wfld.field_count = 0;
    while ( (pdb_get_bytes_read(pdb) - x) < rec_size) {

        wwfld = get_pdb_struct(pdb,0);
        free(wwfld.name);
        wfld.field_count = wfld.field_count + 1;

    }

    pdb_goto_point(pdb,wpdb_file);
    wfld.field_list = malloc(sizeof(pdb_structure_t) * wfld.field_count);
    for (j = 0; j < wfld.field_count; j++) {

        wfld.field_list[j] = get_pdb_struct(pdb,0);

    }

    break;
    case LF_STRUCTURE            : // structure
    wfld.element_count = pdb_grab_16_LE(pdb);
    wfld.properties = pdb_grab_16_LE(pdb);
    wfld.field_index = pdb_grab_32_LE(pdb);
    wfld.derived = pdb_grab_32_LE(pdb);
    wfld.vshape = pdb_grab_32_LE(pdb);
    wfld.size = pdb_grab_16_LE(pdb);
    if (wfld.size == 0x8004) {

        wfld.size = pdb_grab_32_LE(pdb);
        if (rec_size > 0)
        wfld.name = pdb_grab_string(pdb,rec_size - 0x18);
        else
        wfld.name = pdb_grab_string(pdb,0);

    }

    else {

        if (rec_size > 0)
        wfld.name = pdb_grab_string(pdb,rec_size - 0x14);
```

```
    else
    wfld.name = pdb_grab_string(pdb,0);

}

break;
case LF_POINTER              : // pointer
wfld.underlying_type = pdb_grab_32_LE(pdb);
wfld.pointer_array = pdb_grab_32_LE(pdb);
break;
case LF_MEMBER               : //member
wfld.properties = pdb_grab_16_LE(pdb);
wfld.underlying_type = pdb_grab_32_LE(pdb);
wfld.offset = pdb_grab_16_LE(pdb);
if (wfld.offset == 0x8004)
wfld.offset = pdb_grab_32_LE(pdb);
wfld.name = pdb_grab_string(pdb,0);
break;
case LF_NESTTYPE              : //nesttype
wfld.properties = pdb_grab_16_LE(pdb);
wfld.underlying_type = pdb_grab_32_LE(pdb);
wfld.name = pdb_grab_string(pdb,0);
break;
case LF_MODIFIER             : // modifier?
pdb_grab_32_LE(pdb);
pdb_grab_16_LE(pdb);
pdb_grab_16_LE(pdb);
break;
case LF_UNION                : // union
wfld.element_count = pdb_grab_16_LE(pdb);
wfld.properties = pdb_grab_16_LE(pdb);
wfld.field_index = pdb_grab_32_LE(pdb);
wfld.size = pdb_grab_16_LE(pdb);
if (wfld.size == 0x8004) {
    // Size is more than 2 bytes long
    wfld.size = pdb_grab_32_LE(pdb);
    if (rec_size > 0)
    wfld.name = pdb_grab_string(pdb,rec_size - 0x10);
    else
    wfld.name = pdb_grab_string(pdb,0);

}

else {
```

```
    if (rec_size > 0)
    wfld.name = pdb_grab_string(pdb,rec_size - 0x0c);
    else
    wfld.name = pdb_grab_string(pdb,0);

}

break;
case LF_ARGLIST              : // arg_list
wfld.element_count = pdb_grab_32_LE(pdb);
wfld.arguments = malloc(sizeof(uint32_t) * wfld.element_count);
for (j = 0; j < wfld.element_count; j++)
wfld.arguments[j] = pdb_grab_32_LE(pdb);
break;
case LF_PROCEDURE            : // procedure
wfld.retval_type = pdb_grab_32_LE(pdb);
wfld.call_type = pdb_grab_8_LE(pdb);
pdb_grab_8_LE(pdb);
wfld.element_count = pdb_grab_16_LE(pdb);
wfld.field_index = pdb_grab_32_LE(pdb);
break;
case LF_ENUM                 : //enumerated list
wfld.element_count = pdb_grab_16_LE(pdb);
wfld.properties = pdb_grab_16_LE(pdb);
wfld.underlying_type = pdb_grab_32_LE(pdb);
wfld.field_index = pdb_grab_32_LE(pdb);
if (rec_size > 0)
wfld.name = pdb_grab_string(pdb,rec_size - 0x0e);
else
wfld.name = pdb_grab_string(pdb,0);
break;
case LF_ENUMERATE            : // enumerated item
wfld.properties = pdb_grab_16_LE(pdb);
wfld.value = pdb_grab_16_LE(pdb);
if (wfld.value == 0x8004)
wfld.value = pdb_grab_32_LE(pdb);
wfld.name = pdb_grab_string(pdb,0);
break;
case LF_ARRAY                : // array
wfld.underlying_type = pdb_grab_32_LE(pdb);
wfld.index_type = pdb_grab_32_LE(pdb);
wfld.size = pdb_grab_16_LE(pdb);
if (wfld.size == 0x8004) // Size is more than 2 bytes long
wfld.size = pdb_grab_32_LE(pdb);
pdb_grab_16_LE(pdb);
```

```
        break;
        case LF_BITFIELD            : // bitfield
        wfld.underlying_type = pdb_grab_32_LE(pdb);
        wfld.size = pdb_grab_8_LE(pdb);
        wfld.offset = pdb_grab_8_LE(pdb);
        pdb_grab_16_LE(pdb);
        break;

    }

    return wfld;

}


/*****************************************************/
/* Given a data_type value, translate it into a string */
/*****************************************************/
/* the data type may be primitive (e.g. int, char, ...)  or             */
/* the data type may be another variable (or pointer to another variable */
char * get_data_type(pdb_structure_t *pdb_table, uint32_t wdata_type,
                     uint32_t base_idx, int pdb_parm) {

    char *wname, *wwname;
    char wtmp[13];
    int i;
    wwname = NULL;
    wname = NULL;
    if (wdata_type >= base_idx) {

        if (pdb_table[wdata_type - base_idx].name == NULL) {

            if (pdb_table[wdata_type - base_idx].data_type == LF_POINTER) {

                // If it's a pointer, get the data type it points to
                wwname = get_data_type(pdb_table, pdb_table[wdata_type -
                        base_idx].underlying_type, base_idx, PDB_DEFAULT);
                if (pdb_parm == PDB_RET_BASE) {

                    wname = malloc(sizeof(char) * (strlen(wwname) + 1));
                    if (pdb_parm == PDB_DEFAULT) {

                        wname[0] = 42; wname[1] = 0; // ascii 42 = *
                        strcat_s(wname,strlen(wwname) + 1,wwname);
```

```
            }

            else
            strcpy_s(wname,strlen(wwname) + 1,wwname);

        }

        else {

            wname = malloc(sizeof(char) * (strlen(wwname) + 2));
            if (pdb_parm == PDB_DEFAULT) {

                wname[0] = 42; wname[1] = 0; // ascii 42 = *
                strcat_s(wname,strlen(wwname) + 2,wwname);

            }

            else
            strcpy_s(wname,strlen(wwname) + 1,wwname);

        }

        free(wwname);
        return wname;

    }

    else if (pdb_table[wdata_type - base_idx].data_type == LF_ARRAY) {

        // If it's an array, get the data type of the array elements
        wwname = get_data_type(pdb_table, pdb_table[wdata_type -
                    base_idx].underlying_type, base_idx, PDB_DEFAULT);
        if (pdb_parm == PDB_RET_BASE) {

            wname = malloc(sizeof(char) * (strlen(wwname) + 1));
            strcpy_s(wname,strlen(wwname) + 1,wwname);

        }

        else {

            wname = malloc(sizeof(char) * (strlen(wwname) + 3));
            for (i = 0; i < strlen(wwname); i++) wname[i] = wwname[i];
            wname[strlen(wwname)] = 0;
            if (pdb_parm == PDB_DEFAULT) {
```

```
                    wname[strlen(wwname)] = '[';
                    wname[strlen(wwname) + 1] = ']';
                    wname[strlen(wwname) + 2] = 0;

                }


            }

            free(wwname);
            return wname;

        }

        else {

            if (DEBUG_PDB) printf("**get_data_type** Something else\n");
            wname = malloc(sizeof(char) * 7);
            sprintf_s(wtmp,7,"%x",wdata_type);
            strcpy_s(wname,7,wtmp);
            return wname;

        }


    }

    else {

        if (DEBUG_PDB) printf("**get_data_type** Named Struct\n");
        // Just return the name of the Struct
        wname = malloc(sizeof(char) * (strlen(pdb_table[wdata_type -
                             base_idx].name) + 1));
        strcpy_s(wname,strlen(pdb_table[wdata_type - base_idx].name) + 1,
                 pdb_table[wdata_type - base_idx].name);
        return wname;

    }


}

else {
```

```
            if (DEBUG_PDB) printf("**get_data_type** Primitive\n");
            wname = malloc(sizeof(char) * 13);
            for (i = 0; i < 13; i++) wname[i] = 0;
switch (wdata_type) {

            case T_NOTYPE : strcpy_s(wname,8,"No Type"); break;
            case T_ABS: wname = strcpy_s(wname,4,"Abs"); break;
            case T_SEGMENT: strcpy_s(wname,5,"Void"); break;
            case T_HRESULT: strcpy_s(wname,8,"HResult"); break;
            case T_32PHRESULT: strcpy_s(wname, 11, "32PHResult"); break;
            case T_64PHRESULT: strcpy_s(wname, 11, "64PHResult"); break;
            case T_PVOID: strcpy_s(wname,6,"PVoid"); break;
            case T_PFVOID: strcpy_s(wname,7, "PFVoid"); break;
            case T_PHVOID: strcpy_s(wname,7, "PHVoid"); break;
            case T_32PVOID: strcpy_s(wname,8, "32PVoid"); break;
            case T_32PFVOID: strcpy_s(wname, 9, "32PFVoid"); break;
            case T_64PVOID: strcpy_s(wname,8, "64PVoid"); break;
            case T_CURRENCY: strcpy_s(wname,9, "Currency"); break;
            case T_NBASICSTR: strcpy_s(wname, 9, "BasicStr"); break;
            case T_FBASICSTR: strcpy_s(wname, 10, "FBasicStr"); break;
            case T_NOTTRANS: strcpy_s(wname, 9, "NotTrans"); break;
            case T_BIT: strcpy_s(wname,4, "Bit"); break;
            case T_PASCHAR: strcpy_s(wname,8, "PasChar"); break;
            case T_CHAR: strcpy_s(wname,5, "Char"); break;
            case T_PCHAR: strcpy_s(wname,6, "PChar"); break;
            case T_PFCHAR: strcpy_s(wname,7, "PFChar"); break;
            case T_PHCHAR: strcpy_s(wname,7, "PHChar"); break;
            case T_32PCHAR: strcpy_s(wname, 8, "32PChar"); break;
            case T_32PFCHAR: strcpy_s(wname, 9, "32PFChar"); break;
            case T_64PCHAR: strcpy_s(wname,8, "64PChar"); break;
            case T_UCHAR: strcpy_s(wname,6, "UChar"); break;
            case T_RCHAR: strcpy_s(wname,6, "RChar"); break;
            case T_WCHAR: strcpy_s(wname,6, "WChar"); break;
            case T_INT1: strcpy_s(wname,5, "Int1"); break;
            case T_UINT1: strcpy_s(wname,6, "UInt1"); break;
            case T_SHORT: strcpy_s(wname,6, "Short"); break;
            case T_USHORT: strcpy_s(wname,7, "UShort"); break;
            case T_INT2: strcpy_s(wname,5, "Int2"); break;
            case T_UINT2: strcpy_s(wname,6, "UInt2"); break;
            case T_ULONG: strcpy_s(wname,6, "ULong"); break;
            case T_LONG: strcpy_s(wname,5, "Long"); break;
            case T_INT4: strcpy_s(wname,5, "Int4"); break;
            case T_UINT4: strcpy_s(wname,6, "UInt4"); break;
            case T_QUAD: strcpy_s(wname,5, "Quad"); break;
            case T_UQUAD: strcpy_s(wname,6, "UQuad"); break;
```

```
        case T_INT8: strcpy_s(wname,5, "Int8"); break;
        case T_UINT8: strcpy_s(wname,6, "UInt8"); break;
        case T_OCT: strcpy_s(wname,4, "Oct"); break;
        case T_UOCT: strcpy_s(wname,5, "UOct"); break;
        case T_INT16: strcpy_s(wname,6, "Int16"); break;
        case T_UINT16: strcpy_s(wname, 7, "UInt16"); break;
        case T_REAL32: strcpy_s(wname,7, "Real32"); break;
        case T_REAL48: strcpy_s(wname,7, "Real48"); break;
        case T_REAL64: strcpy_s(wname,7, "Real64"); break;
        case T_REAL80: strcpy_s(wname,7, "Real80"); break;
        case T_CPLX32: strcpy_s(wname,10, "Complex32"); break;
        case T_CPLX64: strcpy_s(wname,10, "Complex64"); break;
        case T_CPLX80: strcpy_s(wname,10, "Complex80"); break;
        case T_CPLX128: strcpy_s(wname,11, "Complex128"); break;
        case T_BOOL08: strcpy_s(wname, 10, "Boolean08"); break;
        case T_BOOL16: strcpy_s(wname,10,  "Boolean16"); break;
        case T_BOOL32: strcpy_s(wname, 10, "Boolean32"); break;
        case T_BOOL64: strcpy_s(wname, 10, "Boolean64"); break;
        case T_NCVPTR: strcpy_s(wname, 7, "NCVPtr"); break;
        default:
            sprintf_s(wtmp,13,"%x",pdb_table[wdata_type - base_idx]);
            strcpy_s(wname,13,wtmp);
            break;

        }


    }


    return wname;

}


/**********************************************************/
/* Given a pdb_type record, print out information about it */
/**********************************************************/
void print_pdb_struct(pdb_structure_t *pdb_table, pdb_structure_t * wfld,
                       uint32_t base_idx, uint8_t indent) {

    uint32_t j;

    if (indent) printf("     ");
    switch (wfld->data_type) {
```

```
case LF_FIELDLIST:     // field list
printf("%6x \t Field List (%i entries)\n",wfld->index,wfld->field_count);
for (j = 0; j < wfld->field_count; j++) {

    print_pdb_struct(pdb_table,&wfld->field_list[j],base_idx,1);

}

break;
case LF_STRUCTURE            : // structure
case LF_UNION:
printf("%6x %20s\t Structure (%i entries, %i bytes)\tBased on %s\n",
    wfld->index,wfld->name,wfld->element_count,wfld->size,
    get_data_type(pdb_table,wfld->field_index,base_idx,PDB_DEFAULT));
break;
case LF_POINTER              : // pointer
printf("%6x \t *%s\n", wfld->index,get_data_type(pdb_table,
                       wfld->underlying_type,base_idx,PDB_DEFAULT));
break;
case LF_MEMBER               : //member
printf("%6x \t %3i\t%30s\t%s\n", wfld->index,wfld->offset,
       get_data_type(pdb_table,wfld->underlying_type,
                     base_idx,PDB_DEFAULT),wfld->name);
break;
case LF_MODIFIER             : // modifier?
printf("%6x MODIFIER\n",wfld->index);
break;
case LF_ARGLIST              : // arg_list
printf("%6x\tArgument List \t(" ,wfld->index);
for (j = 0; j < wfld->element_count; j++)  {

    printf("%s",get_data_type(pdb_table,
                wfld->arguments[j],base_idx,PDB_DEFAULT)) ;
    if (j + 1 <  wfld->element_count) printf(", ");

}

printf(")\n");
break;
case LF_PROCEDURE            : // procedure
printf("%6x\t Procedure\t return %s using %x arguments\n",
            wfld->index, get_data_type(pdb_table,wfld->retval_type,
            base_idx,PDB_DEFAULT), wfld->field_index);
break;
```

```
        case LF_ENUM                 : //enumerated list
        printf("%6x\t %20s \t Enumerated List of %i elements\t Data Type = %s \t",
                 wfld->index,wfld->name,wfld->element_count,
                 get_data_type(pdb_table,
                        wfld->underlying_type,base_idx,PDB_DEFAULT));
        printf(" Elements = %x\n", wfld->field_index);
        break;
        case LF_ENUMERATE            : // enumerated item
        printf("%6x\t Value: %i\tName: %s\n",
                    wfld->index,wfld->value,wfld->name);
        break;
        case LF_ARRAY                : // array
        printf("%6x\t Array of %s\t Index Type = %s\t Size (in hex bytes) = %x\n",
                     wfld->index,get_data_type(pdb_table,
                             wfld->underlying_type,base_idx,PDB_DEFAULT),
                     get_data_type(pdb_table,
                             wfld->index_type,base_idx,PDB_DEFAULT),
                     wfld->size);
        break;
        case LF_BITFIELD:
        printf("%6x\t Bitfield of %s\t Length = %i \t Bit Offset of Bit 0 = %i\n",
               wfld->index,get_data_type(pdb_table,
                     wfld->underlying_type,base_idx,PDB_DEFAULT),
               wfld->size,wfld->offset);
        break;
        default:
        printf("%6x Unknown \tType: %x\t%s\n",
                    wfld->index,wfld->data_type,wfld->name);

    }


}


/******************************************************************************/
/****************************************************/
/* RETRIEVE INFORMATION ABOUT PDB ELEMENTS -- Private */
/****************************************************/

/************************************************************/
/* Return the number of bytes for the provided data structure */
/************************************************************/
//     for arrays, this returns the total size (size of a single element X
//      number of elements)
```

```
uint32_t get_pdb_type_sizeof2(pdb_structures_t *pdb_struct, char *fldname) {

    uint32_t i;

    // First check if it's a primirive
    if (!strcmp("Void",fldname)) return 4;
    if (!strcmp("32PHResult",fldname)) return 4;
    if (!strcmp("64PHResult",fldname)) return 8;
    if (!strcmp("PVoid",fldname)) return 4;
    if (!strcmp("PFVoid",fldname)) return 4;
    if (!strcmp("PHVoid",fldname)) return 4;
    if (!strcmp("PHVoid",fldname)) return 4;
    if (!strcmp("32PVoid",fldname)) return 4;
    if (!strcmp("32PFVoid",fldname)) return 4;
    if (!strcmp("64PVoid",fldname)) return 8;
    if (!strcmp("PasChar",fldname)) return 1;
    if (!strcmp("Char",fldname)) return 1;
    if (!strcmp("PFChar",fldname)) return 4;
    if (!strcmp("PHChar",fldname)) return 4;
    if (!strcmp("32PChar",fldname)) return 4;
    if (!strcmp("32PFChar",fldname)) return 4;
    if (!strcmp("64PChar",fldname)) return 8;
    if (!strcmp("UChar",fldname)) return 1;
    if (!strcmp("RChar",fldname)) return 1;
    if (!strcmp("WChar",fldname)) return 1;
    if (!strcmp("Int1",fldname)) return 1;
    if (!strcmp("UInt1",fldname)) return 1;
    if (!strcmp("Short",fldname)) return 2;
    if (!strcmp("Short",fldname)) return 2;
    if (!strcmp("Int2",fldname)) return 2;
    if (!strcmp("Uint2",fldname)) return 2;
    if (!strcmp("ULong",fldname)) return 4;
    if (!strcmp("Long",fldname)) return 4;
    if (!strcmp("Int4",fldname)) return 4;
    if (!strcmp("UInt4",fldname)) return 4;
    if (!strcmp("Quad",fldname)) return 8;
    if (!strcmp("UQuad",fldname)) return 8;
    if (!strcmp("Int8",fldname)) return 8;
    if (!strcmp("UInt8",fldname)) return 8;
    if (!strcmp("Oct",fldname)) return 8;
    if (!strcmp("UOct",fldname)) return 8;
    if (!strcmp("Int16",fldname)) return 16;
    if (!strcmp("UInt16",fldname)) return 16;
    if (!strcmp("Real32",fldname)) return 32;
    if (!strcmp("Real48",fldname)) return 48;
```

```
if (!strcmp("Real64",fldname)) return 64;
if (!strcmp("Real80",fldname)) return 80;
if (!strcmp("Complex32",fldname)) return 32;
if (!strcmp("Complex64",fldname)) return 64;
if (!strcmp("Complex80",fldname)) return 80;
if (!strcmp("Complex128",fldname)) return 128;

// If it's a pointer, return either 8 bytes or 4 bytes (depending on 64 bit)
if (fldname[0] == 0x2a) return (pdb_x64?8:4); // first character is a *
                                              // (i.e., pointer)

// Search through the sructures to find the correct one
for (i = 0; i < pdb_struct->count; i++) {

    if (pdb_struct->table[i].name == NULL) continue;
    // When you find the right one, return the size
    if (!strcmp(fldname,pdb_struct->table[i].name)) {

        if ( (pdb_struct->table[i].data_type == LF_STRUCTURE) ||
             (pdb_struct->table[i].data_type == LF_UNION) ) {

            if (pdb_struct->table[i].size == 0) continue;
            return pdb_struct->table[i].size;

        }

        else if (pdb_struct->table[i].data_type == LF_ARRAY) {

            return pdb_struct->table[i].size;

        }

        else {

            return 0; // This should never happen

        }


    }


}

if (DEBUG_PDB) printf("**pdb_type_sizeof2** No match %s \n",fldname);
```

```
        return 0; // This should never happen

    }


    /***************************************************************************/ /*
/*    by using the offset parameter, the next (+1) or previous (-1) member's  */
/*     offset can be determined, allowing the calling function to calculate   */
/*        the size of the member                                              */
    uint64_t get_pdb_type_offset2(pdb_structures_t *pdb_struct, char *fldname,
                                  char *subfldname, int woffset) {
        // Offset is normally 0 but may be 1 or -1 to get size
        uint32_t i, j, windex;

        // Search through the list to find the correct entry
        for (i = 0; i < pdb_struct->count; i++) {

            if (pdb_struct->table[i].name == NULL) continue;
            // When you find the correct entry
            if (!strcmp(fldname,pdb_struct->table[i].name)) {


                // If it's a structure or a union
                if ( (pdb_struct->table[i].data_type == LF_STRUCTURE) ||
                     (pdb_struct->table[i].data_type == LF_UNION) ) {

                    if (pdb_struct->table[i].size == 0) continue;

                    // Find the field list
                    // This better point to  a field_list
                    windex = pdb_struct->table[i].field_index - pdb_struct->base_index;
                    if (pdb_struct->table[windex].data_type != LF_FIELDLIST) {
                        // This should never happen
                        return pdb_struct->table[i + woffset].offset;

                    }

                    // Read through the field list
                    for (j = 0; j < pdb_struct->table[windex].field_count; j++) {
                        // these should be LF_MEMBER
                        // Once you find the field name
                        if (!strcmp(subfldname,
                              pdb_struct->table[windex].field_list[j].name)) {

                            // If we're looking for the next field (to do a size
```

```
            // calculation) and we're at the end of the file,
            // use the size of the structure
            if (j + woffset >=
                 pdb_struct->table[windex].field_count) {

                return (uint64_t)
                      get_pdb_type_sizeof2(pdb_struct, fldname)
                            & 0x00000000ffffffff;

            }

            // Otherwise, return the offset
            else {

                return pdb_struct->table[windex].field_list[j +
                        woffset].offset;

            }


        }


    }



}

// If it's anything else, we're confused -- this shouldn't happen
// (if it's a member return the offset 'cause we have it, otherwise
// return 0)
else if (pdb_struct->table[i].data_type == LF_MEMBER) {

    // This should never happen
    return pdb_struct->table[i + woffset].offset;


}

else {

    return 0; // This should never happen

}
```

```
        }


    }

    if (DEBUG_PDB)
      printf("**pdb_type_offset2** No match %s %s\n",fldname,subfldname);
    return 0; // This should never happen

}


/*******************************************************/
/* Returns the underlying type of a member in a structure */
/*******************************************************/
char * get_pdb_type_utype(pdb_structures_t *pdb_struct, char *fldname,
                           char *subfldname) {

    uint32_t i, j, windex;
    char *wname;
    // Search for the appropriate entry
    for (i = 0; i < pdb_struct->count; i++) {

        if (pdb_struct->table[i].name == NULL) continue;
        // When you find the entry
        if (!strcmp(fldname,pdb_struct->table[i].name)) {

            // If it's a structure or a union
            if ( (pdb_struct->table[i].data_type == LF_STRUCTURE) ||
                 (pdb_struct->table[i].data_type == LF_UNION) ) {

                if (pdb_struct->table[i].size == 0) continue;
                // Find the field list
                // This better point to  a field_list
                windex = pdb_struct->table[i].field_index -
                         pdb_struct->base_index;
                if (pdb_struct->table[windex].data_type != LF_FIELDLIST) {

                    wname = get_data_type(pdb_struct->table,
                              pdb_struct->table[windex].underlying_type,
                              pdb_struct->base_index,PDB_RET_BASE);
                    return wname; // This should never happen
```

```
                }

                // Read through the field list
                for (j = 0; j < pdb_struct->table[windex].field_count; j++) {
                    // thse should be LF_MEMBER
                    // Once you find the right field
                    if (!strcmp(subfldname,
                           pdb_struct->table[windex].field_list[j].name)) {

                        // Return the underlying type
                        wname = get_data_type(pdb_struct->table,
                          pdb_struct->table[windex].field_list[j].underlying_type,
                          pdb_struct->base_index,PDB_RET_BASE);
                        return wname;

                    }


                }



            }

            else {

                return NULL; // This should never happen

            }


        }


    }

    return NULL; // This should never happen

}


/***************************************************/
/* Main helper function for all public PDB functions */
/***************************************************/
// The name is inputted as XXX.YYYY or possibly XXX.YYY.ZZZ.etc.
```

```
uint64_t get_pdb_type_helper(pdb_structures_t *pdb_struct, char *name,
                             int ret_parm) {


    char *parsestring, *dotpos, *fldname, *subfldname, *wfldname, *orig_ptr;
    int i;
    uint64_t fld_offset, wfld_offset, ret_size;
    int64_t x; // x CAN BE NEGATIVE (if x is part of a union,
               // the field after it may be less than it!!)

    parsestring = malloc(sizeof(char) * (strlen(name) + 1));
    orig_ptr = parsestring;
    strcpy_s(parsestring, strlen(name) + 1,name);
    wfldname = NULL;
    fldname = NULL;
    subfldname = NULL;
    // First, split off the top level name and the first (and possibly only)
    // sublevel name
    dotpos = strchr(parsestring,'.');
    // Get the fldname
    if (dotpos != NULL) {

        fldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
        for (i = 0; i < (dotpos - parsestring + 1); i++) fldname[i] = 0;
        strncpy_s(fldname, dotpos - parsestring + 2, parsestring, dotpos -
                  parsestring);
        parsestring = &dotpos[1];

    }

    else {

        fldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
        strcpy_s(fldname, strlen(parsestring) + 1, parsestring);
        parsestring = NULL;

    }

    // Get the subfldname
    if (parsestring != NULL) {

        dotpos = strchr(parsestring,'.');
        if (dotpos != NULL) {

            subfldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
```

```
        for (i = 0; i < (dotpos - parsestring + 1); i++) subfldname[i] = 0;
        strncpy_s(subfldname, dotpos - parsestring + 2, parsestring,
                  dotpos - parsestring);
        parsestring = &dotpos[1];

    }

    else {

        subfldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
        strcpy_s(subfldname, strlen(parsestring) + 1, parsestring);
        parsestring = NULL;

    }


}

else {

    free(orig_ptr);
    free(fldname);
    free(subfldname);
    return 0;

}


// Get the location of the subfield in the main field's structure
wfld_offset = get_pdb_type_offset2(pdb_struct, fldname, subfldname, 0);
fld_offset = wfld_offset;
// Now go all the way down through multiple subfields to find exactly
// where the rightmost subfield in the input string is located
while (parsestring != NULL) {
    // multiple structures in the input stream
    wfldname = get_pdb_type_utype(pdb_struct, fldname, subfldname);
    free(fldname);
    free(subfldname);
    fldname = wfldname;
    dotpos = strchr(parsestring,'.');
    if (dotpos != NULL) {

        subfldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
        for (i = 0; i < (dotpos - parsestring + 1); i++) subfldname[i] = 0;
        strncpy_s(subfldname, dotpos - parsestring + 2, parsestring,
```

```
                dotpos - parsestring);
            parsestring = &dotpos[1];

        }

        else {

            subfldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
            strcpy_s(subfldname, strlen(parsestring) + 1, parsestring);
            parsestring = NULL;

        }

        wfld_offset = get_pdb_type_offset2(pdb_struct, fldname, subfldname, 0);
        fld_offset += wfld_offset;

    }


    // Now figure out what the request is for and send back the info
    switch (ret_parm) {

        // PDB_RET_SIZE: We want the size of the subfield (in the case of
        //                the array, we want
        //               the total size of the array)
        //               so we check where the next field starts and
        //               subtract them.
        //               The reason for the FOR loop is that union fields
        //               are all at the
        //               same location, so we want to move past them
        //               The guess is 40 is enough to get past a LF_UNION
        case PDB_RET_SIZE: {

            for (i = 1; i < 40; i++) {
                // NEED TO CLEAN THIS UP TO ONLY GO TO END OF STRUCTURE.
                // NEEDED TO HANDLE LF_UNION
                x = get_pdb_type_offset2(pdb_struct, fldname, subfldname, i)
                  - wfld_offset;
                if (x > 0) {

                    free(orig_ptr);
                    free(fldname);
                    free(subfldname);
                    return x;
```

```
            }


        }

        free(orig_ptr);
        free(fldname);
        free(subfldname);
        return 0;

    }

    // PDB_RET_SIZEOF: In this case we want the size of an element of
    //                 this type (probably an array)
    //                 so we're going to first get the underlying type
    //                 and then get the size of it
    case PDB_RET_SIZEOF:
    wfldname = get_pdb_type_utype(pdb_struct, fldname, subfldname);
    ret_size = (uint64_t) get_pdb_type_sizeof2(pdb_struct, wfldname)
             & 0x00000000ffffffff;
    free(orig_ptr);
    free(fldname);
    free(wfldname);
    free(subfldname);
    return ret_size;
    // PDB_DEFAULT: the original reason for this function, tell me what
    //              the offset is for the rightmost subfield
    //              from the main field
    case PDB_DEFAULT:
    free(orig_ptr);
    free(fldname);
    free(subfldname);
    return fld_offset;
    default:
    free(orig_ptr);
    free(fldname);
    free(subfldname);
    return fld_offset;

}


}
```

```
/******************************************************************************/

/**********************************************************************/
/* RETRIEVE DATA FROM A MEMORY DUMP USING PDB TYPE INFORMATION -- PUBLIC */
/**********************************************************************/

/*******************************************************/
/* Retrieve a field given the physical address in memory */
/*******************************************************/
uint64_t get_pdb_type(char *name, uint64_t base_loc) {

    uint64_t fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    uint64_t fld_size   = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);

    switch (fld_size) {

        case 1: return ((uint64_t) memgrab_8_LE(base_loc + fld_offset) )
                    & 0x00000000000000ffULL;
        case 2: return ((uint64_t) memgrab_16_LE(base_loc + fld_offset))
                    & 0x000000000000ffffULL;
        case 4: return ((uint64_t) memgrab_32_LE(base_loc + fld_offset))
                    & 0x00000000ffffffffULL;
        case 6: return ((uint64_t) memgrab_48_LE(base_loc + fld_offset))
                    & 0x0000ffffffffffffULL;
        case 8: return ((uint64_t) memgrab_64_LE(base_loc + fld_offset))
                    & 0xffffffffffffffffULL;
        default: {
            printf("**get_pdb_type  field: %s     unknown size: %llx\n",
                    name,fld_size);
            return 0;
        }


    }


}


/*******************************************************************/
/* Retrieve a field given the virtual address in memory for a process */
/*******************************************************************/
uint64_t pget_pdb_type(char *name, uint64_t base_loc, uint64_t DTB) {

    uint64_t fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
```

```
    uint64_t fld_size   = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
    switch (fld_size) {

        case 1: return ((uint64_t) memgrabp_8_LE(DTB, base_loc + fld_offset) )
                     & 0x00000000000000ffULL;
        case 2: return ((uint64_t) memgrabp_16_LE(DTB, base_loc + fld_offset))
                     & 0x000000000000ffffULL;
        case 4: return ((uint64_t) memgrabp_32_LE(DTB, base_loc + fld_offset))
                     & 0x00000000ffffffffULL;
        case 6: return ((uint64_t) memgrabp_48_LE(DTB, base_loc + fld_offset))
                     & 0x0000ffffffffffffULL;
        case 8: return ((uint64_t) memgrabp_64_LE(DTB, base_loc + fld_offset))
                     & 0xffffffffffffffffULL;
        default: {
            printf("**pget_pdb_type  field: %s     unknown size: %llx\n",
                    name,fld_size);
            return 0;
        }


    }


}


/**************************************************************************/
/* Retrieve a field given the virtual address in memory of a kernel process */
/**************************************************************************/
uint64_t keget_pdb_type(char *name, uint64_t base_loc) {

    uint64_t fld_offset, fld_size;
    fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    fld_size   = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
    switch (fld_size) {

        case 1: return ((uint64_t) memgrabke_8_LE(base_loc + fld_offset))
                     & 0x00000000000000ffULL;
        case 2: return ((uint64_t) memgrabke_16_LE(base_loc + fld_offset))
                     & 0x000000000000ffffULL;
        case 4: return ((uint64_t) memgrabke_32_LE(base_loc + fld_offset))
                     & 0x00000000ffffffffULL;
        case 6: return ((uint64_t) memgrabke_64_LE(base_loc + fld_offset))
                     & 0x0000ffffffffffffULL;
        case 8: return ((uint64_t) memgrabke_64_LE(base_loc + fld_offset))
```

```
                    & 0xffffffffffffffffULL;
        default: {
            printf("**keget_pdb_type  field: %s     unknown size: %llx\n",
                    name,fld_size); return 0;
        }


    }


}


/*********************************************************/
/* Retrieve an array given the physical address in memory */
/*********************************************************/
uint64_t * get_pdb_type_array(char *name, uint64_t base_loc) {


    int i;
    pdb_array_attr_t wattr;
    uint64_t fld_offset, fld_size;
    uint64_t * retval = 0;
    fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    fld_size = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
    wattr = get_pdb_type_array_attr (name);
    retval = malloc(sizeof(uint64_t) * (wattr.elements + 1));

    for (i=0; i < (fld_size / wattr.size); i++)
    switch (wattr.size) {

        case 1: {
            retval[i] = ((uint64_t) memgrab_8_LE(base_loc + fld_offset + i
                      * wattr.size)) & 0x00000000000000ffULL;
            continue;
        }

        case 2: {
            retval[i] = ((uint64_t) memgrab_16_LE(base_loc + fld_offset + i
                      * wattr.size)) & 0x000000000000ffffULL;
            continue;
        }

        case 4: {
            retval[i] = ((uint64_t) memgrab_32_LE(base_loc + fld_offset + i
```

```
                    * wattr.size)) & 0x00000000ffffffffULL;
            continue;
        }

        case 8: {
            retval[i] = ((uint64_t) memgrab_64_LE(base_loc + fld_offset + i
                    * wattr.size)) & 0xffffffffffffffffULL;
            continue;
        }

        default: {
            printf("**get_pdb_type_array  field: %s     unknown size: %llx\n",
                   name,fld_size);
            return 0;
        }


    }

    return retval;

}


/*****************************************************************************/
/* Retrieve an array given the virtual address in memory of a kernel process */
/*****************************************************************************/
uint64_t * keget_pdb_type_array(char *name, uint64_t base_loc) {


    uint64_t fld_offset, fld_size, *retval;
    pdb_array_attr_t wattr;
    int i;

    fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    fld_size = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
    wattr = get_pdb_type_array_attr (name);
    retval = malloc(sizeof(uint64_t) * (wattr.elements + 1));

    for (i=0; i < (fld_size / wattr.size); i++) {

        switch (wattr.size) {

            case 1: {
                retval[i] = ((uint64_t) memgrabke_8_LE(base_loc + fld_offset + i
```

```
                    * wattr.size) ) & 0x00000000000000ffULL;
                continue;
            }

            case 2: {
                retval[i] = ((uint64_t) memgrabke_16_LE(base_loc + fld_offset + i
                    * wattr.size)) & 0x000000000000ffffULL;
                continue;
            }

            case 4: {
                retval[i] = ((uint64_t) memgrabke_32_LE(base_loc + fld_offset + i
                    * wattr.size)) & 0x00000000ffffffffULL;
                continue;
            }

            case 8: {
                retval[i] = ((uint64_t) memgrabke_64_LE(base_loc + fld_offset + i
                    * wattr.size)) & 0xffffffffffffffffULL;
                continue;
            }

            default: {
                printf("**keget_pdb_type_array**  field: %s     unknown size: %x\n",
                       name,wattr.size);
                return 0;
            }


        }


    }

    return retval;

}


/*********************************************************************/
/* Retrieve a field given the virtual address in memory for a process */
/*********************************************************************/
uint64_t * pget_pdb_type_array(char *name, uint64_t base_loc, uint64_t DTB) {
```

```
    uint64_t fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    uint64_t fld_size = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
    pdb_array_attr_t wattr = get_pdb_type_array_attr (name);
    uint64_t *retval = malloc(sizeof(uint64_t) * (wattr.elements + 1));
    int i;

    for (i=0; i < (fld_size / wattr.size); i++)
    switch (wattr.size) {

        case 1: {
            retval[i] = ((uint64_t) memgrabp_8_LE(DTB, base_loc + fld_offset + i
                     * wattr.size) ) & 0x00000000000000ffULL;
            continue;
        }

        case 2: {
            retval[i] = ((uint64_t) memgrabp_16_LE(DTB, base_loc + fld_offset + i
                     * wattr.size)) & 0x000000000000ffffULL;
            continue;
        }

        case 4: {
            retval[i] = ((uint64_t) memgrabp_32_LE(DTB, base_loc + fld_offset + i
                     * wattr.size)) & 0x00000000ffffffffULL;
            continue;
        }

        case 8: {
            retval[i] = ((uint64_t) memgrabp_64_LE(DTB, base_loc + fld_offset + i
                     * wattr.size)) & 0xffffffffffffffffULL;
            continue;
        }

        default: {
            printf("**get_pdb_type_array  field: %s     unknown size: %llx\n",
                    name,fld_size);
            return 0;
        }


    }

    return retval;

}
```

```
    /*************************************************************************/
/* Retrieve ascii string given physical address in memory without known size */
/*************************************************************************/
    char * get_pdb_type_string(char *name, uint64_t base_loc) {


        uint32_t fld_offset,wsize;
        char * retstring;
        uint32_t i;

        fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
        wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);

        retstring = malloc(sizeof(char) * (wsize + 1));

        for (i=0; i < wsize + 1; i++)
        retstring[i] = memgrab_8_LE(base_loc + fld_offset + i);

        return retstring;

    }


    /*************************************************************************/
/* Retrieve ascii string given virtual address for process without known size */
/*************************************************************************/
    char * pget_pdb_type_string(char *name, uint64_t base_loc, uint64_t DTB) {


        uint32_t fld_offset,wsize;
        char * retstring;
        uint32_t i;

        fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
        wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);

        retstring = malloc(sizeof(char) * (wsize + 1));

        for (i=0; i < (wsize + 1); i++)
        retstring[i] = memgrabp_8_LE(DTB, base_loc + fld_offset + i);

        return retstring;
```

```
    }


    /**************************************************************************/
/* Retrieve ascii string given the virtual address for a kernel process w/o a */
/* known size                                                                 */
/******************************************************************************/
    char * keget_pdb_type_string(char *name, uint64_t base_loc) {


        uint32_t fld_offset,wsize;
        char * retstring;
        uint32_t i;

        fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
        wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);

        retstring = malloc(sizeof(char) * (wsize + 1));

        for (i=0; i < (wsize + 1); i++)
        retstring[i] = memgrabke_8_LE(base_loc + fld_offset + i);

        return retstring;

    }



/******************************************************************************/
/* Retrieve an ascii string given the virtual address in memory for a process */
/* with a known size                                                          */
/******************************************************************************/
    char * pget_pdb_type_stringn(char *name, uint64_t base_loc, uint64_t DTB,
                                 uint16_t wlength) {


        uint32_t fld_offset;
        char * retstring;
        uint16_t i;

        fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);

        retstring = malloc(sizeof(char) * (wlength + 2));

        for (i=0; i < (wlength + 1); i++) retstring[i] = 0;
```

```
        for (i=0; i < (wlength); i++)
        retstring[i] = memgrabp_8_LE(DTB, base_loc + fld_offset + i);
        retstring[wlength] = 0;

        return retstring;

    }



/************************************************************************/
/* Retrieve an ascii string given the virtual address in memory for a kernel  */
/* process with a known size                                                   */
/************************************************************************/
    char * keget_pdb_type_stringn(char *name, uint64_t base_loc, uint16_t wlength) {


        uint64_t fld_offset;
        char * retstring;
        int i;

        fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);

        retstring = malloc(sizeof(char) * (wlength + 2));

        for (i=0; i < (wlength + 1); i++) retstring[i] = 0;

        for (i=0; i < (wlength); i++)
        retstring[i] = memgrabke_8_LE(base_loc + fld_offset + i);
        retstring[wlength] = 0;

        return retstring;

    }



/************************************************************************/
/* Retrieve a unicode string given the virtual address in memory for a process*/
/* with a known size                                                           */
/************************************************************************/
    char * pget_pdb_type_unicoden(char *name, uint64_t base_loc, uint64_t DTB, uint16_t
```

```
    uint32_t fld_offset;
    char * retstring;
    uint16_t i;

    fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);

    retstring = malloc(sizeof(char) * (wlength / 2) + 2);

    for (i=0; i < (wlength + 1); i++) retstring[i / 2] = 0;

    for (i=0; i < (wlength + 1); i+=2)
    retstring[i / 2] = memgrabp_8_LE(DTB, base_loc + fld_offset + i);

    return retstring;

}



/****************************************************************************/
/* Retrieve a unicode string given the virtual address in memory for a kernel */
/* process with a known size                                                */
/****************************************************************************/
  char * keget_pdb_type_unicoden(char *name, uint64_t base_loc, uint16_t wlength) {


    uint32_t fld_offset;
    char * retstring;
    uint16_t i;

    fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);

    retstring = malloc(sizeof(char) * (wlength / 2) + 2);

    for (i=0; i < (wlength + 1); i++) retstring[i / 2] = 0;

    for (i=0; i < (wlength + 1); i+=2)
    retstring[i / 2] = memgrabke_8_LE(base_loc + fld_offset + i);

    return retstring;

}
```

```
/*************************************************************************/
/* Retrieve the attributes of an array -- it's initial offset, how big each */
/* element is, and how many elements there are                              */
/*************************************************************************/
    pdb_array_attr_t get_pdb_type_array_attr(char *name) {


        pdb_array_attr_t wattributes;
        uint32_t wsize;

        wattributes.base_offset = (uint32_t)
                        get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
        wattributes.size        = (uint32_t)
                        get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZEOF);
        wsize                   = (uint32_t)
                        get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
        if (wattributes.size == 0)
          wattributes.elements = 0;
        else
          wattributes.elements =  wsize / wattributes.size;
        return wattributes;

    }


    /******************************************/
    /* Returns the size of the requested field */
    /******************************************/
    uint64_t get_pdb_type_sizeof(char *fldname) {

        return (uint64_t) get_pdb_type_sizeof2(ntoskrnl_pdb, fldname)
                        & 0x00000000ffffffff;

    }



/*************************************************************************/
/* Returns offset of the requested field from the eginning of the structure */
/*************************************************************************/
    uint64_t get_pdb_type_offset(char *name) {

        return get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);

    }
```

```
/****************************************************************************/
/* Create a string that contains the passed value in the right place for the */
/* passed field                                                              */
/****************************************************************************/
    // Note: If the size of the fieldname being stuffed with a value is more
    // than 32bits, the 32bits sent will repeat
    uint32_t * get_pdb_type_value(char *name, uint32_t wvalue) {


        char *parsestring, *dotpos, *fldname;
        uint32_t * ret_val;
        uint8_t * ret_bytes;
        int ii;
        uint32_t i,j;
        uint32_t wlongs, woffset, wsize;
        uint32_t wbytes;

        parsestring = malloc(sizeof(char) * (strlen(name) + 1) );
        strcpy_s(parsestring, strlen(name) + 1,name);

        if (DEBUG_PDB) printf("**get_pdb_type_value** name = %s  \n",name);
        // Get the size of the overall structure
        dotpos = strchr(parsestring,'.');
        if (dotpos != NULL) {

            fldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
            for (ii = 0; ii < (dotpos - parsestring + 1); ii++) fldname[ii] = 0;
            strncpy_s(fldname, dotpos - parsestring + 2, parsestring, dotpos
                        - parsestring);

        }

        else {

            fldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
            strcpy_s(fldname, strlen(parsestring) + 1,parsestring);

        }

        wbytes = get_pdb_type_sizeof2(ntoskrnl_pdb,fldname);

        // Initialize the return value
```

```
    ret_bytes = malloc (sizeof(uint8_t) * (wbytes + 1));
    for (i = 0; i < wbytes; i++) ret_bytes[i] = 0;

    // Figure out exactly where the passed value belongs and put it there
    woffset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
    for (i = woffset; i < woffset + wsize; i += 4) {

        ret_bytes[i] = wvalue % 0x100;
        ret_bytes[i + 1] = (uint8_t) (wvalue % 0x10000 - ret_bytes[i]);
        ret_bytes[i + 2] = (uint8_t) (wvalue % 0x1000000 - ret_bytes[i + 1]
                           - ret_bytes[i]);
        ret_bytes[i + 3] = ((int) (wvalue / 0x1000000));

    }


    // Convert the string into an array of uint32
    wlongs = (wbytes / 4) + (wbytes % 4?1:0);
    ret_val = malloc( sizeof(uint32_t) * wlongs);
    for (i = 0; i < wlongs; i++) {

        ret_val[i] = 0;
        for (j = 0; j < 4; j++)
        ret_val[i] += (uint32_t) (ret_bytes[i * 4 + j] * pow(256,(3 - j) ));

    }

    free(parsestring);
    free(fldname);

    return ret_val;

}


// Check if a field exists */
uint8_t get_pdb_fld_exists(char *name) {


    char *parsestring, *dotpos, *fldname, *subfldname, *wfldname, *ostring;
    int i;
    uint8_t x;

    parsestring = malloc(sizeof(char) * (strlen(name) + 1));
```

```
ostring = parsestring;
strcpy_s(parsestring, strlen(name) + 1,name);
fldname = NULL;
subfldname = NULL;
wfldname = NULL;

// Get the structure
dotpos = strchr(parsestring,'.');
if (dotpos != NULL) {

    fldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
    for (i = 0; i < (dotpos - parsestring + 1); i++) fldname[i] = 0;
    strncpy_s(fldname, dotpos - parsestring + 2, parsestring, dotpos
               - parsestring);
    parsestring = &dotpos[1];

}

else {

    fldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
    strcpy_s(fldname, strlen(parsestring) + 1,parsestring);
    parsestring = NULL;

}


// Get the field in the structure
if (parsestring != NULL) {

    dotpos = strchr(parsestring,'.');
    if (dotpos != NULL) {

        subfldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
        for (i = 0; i < (dotpos - parsestring + 1); i++) subfldname[i] = 0;
        strncpy_s(subfldname, dotpos - parsestring + 2, parsestring,
                   dotpos - parsestring);
        parsestring = &dotpos[1];

    }

    else {

        subfldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
        strcpy_s(subfldname, strlen(parsestring) + 1,parsestring);
```

```
        parsestring = NULL;

    }


}

// If there's no subfield, return the size of the entire structure
else {

    x = (uint8_t) ((get_pdb_type_sizeof2(ntoskrnl_pdb, fldname)));
    // Based on what was passed, we're good
    free(fldname);
    free(subfldname);
    free(ostring);
    // Based on what was passed, we're good
    return x;

}


// If there's mroe subfields, get to the bottom of the passed field
while (parsestring != NULL) {
    // multiple structures in the input stream
    wfldname = get_pdb_type_utype(ntoskrnl_pdb, fldname, subfldname);
    free(fldname);
    free(subfldname);
    fldname = wfldname;
    if (fldname == NULL) {
        free(ostring);return 0;
    }

    dotpos = strchr(parsestring,'.');
    if (dotpos != NULL) {

        subfldname = malloc(sizeof(char) * (dotpos - parsestring + 2));
        for (i = 0; i < (dotpos - parsestring + 1); i++) subfldname[i] = 0;
        strncpy_s(subfldname, dotpos - parsestring + 2, parsestring,
                  dotpos - parsestring);
        parsestring = &dotpos[1];

    }

    else {
```

```
            subfldname = malloc(sizeof(char) * (strlen(parsestring) + 1));
            strcpy_s(subfldname, strlen(parsestring) + 1,parsestring);
            parsestring = NULL;

        }


    }


    // Get the underlying type for the subfield
    wfldname = get_pdb_type_utype(ntoskrnl_pdb, fldname, subfldname);
    free(fldname);
    free(subfldname);
    fldname = wfldname;
    if (fldname == NULL) {

        free(wfldname);
        free(ostring);
        return 0; // The subfield couldn't be found

    }

     // Based on what was passed, we're good
    x = (uint8_t) (get_pdb_type_sizeof2(ntoskrnl_pdb, fldname));
    free(wfldname);
    free(ostring);
    return x; // Based on what was passed, we're good

}


/*************************************************************/
/* The following code is byte-specific versions of the above code */
/*************************************************************/
uint32_t get_pdb_type_uint32(char *name, uint64_t base_loc) {

    uint64_t fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    return memgrab_32_LE(base_loc + fld_offset);

}


uint32_t pget_pdb_type_uint32(char *name, uint64_t base_loc, uint64_t DTB) {
```

```
    return memgrabp_32_LE(DTB,base_loc + get_pdb_type_helper(ntoskrnl_pdb,name,0) );

}


uint32_t keget_pdb_type_uint32(char *name, uint64_t base_loc) {

    return memgrabke_32_LE(base_loc + get_pdb_type_helper(ntoskrnl_pdb,name,0) );

}


uint16_t get_pdb_type_uint16(char *name, uint64_t base_loc) {


    uint64_t fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    return memgrab_16_LE(base_loc + fld_offset);

}


uint16_t pget_pdb_type_uint16(char *name, uint64_t base_loc, uint64_t DTB) {

    return memgrabp_16_LE(DTB,base_loc + get_pdb_type_helper(ntoskrnl_pdb,name,0));

}


uint16_t keget_pdb_type_uint16(char *name, uint64_t base_loc) {

    return memgrabke_16_LE(base_loc + get_pdb_type_helper(ntoskrnl_pdb,name,0));

}


uint8_t get_pdb_type_uint8(char *name, uint64_t base_loc) {


    uint64_t fld_offset = get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    return memgrab_8_LE(base_loc + fld_offset);

}


uint8_t pget_pdb_type_uint8(char *name, uint64_t base_loc, uint64_t DTB) {
```

```
    return memgrabp_8_LE(DTB,base_loc + get_pdb_type_helper(ntoskrnl_pdb,name,0));

}


uint8_t keget_pdb_type_uint8(char *name, uint64_t base_loc) {

    return memgrabke_8_LE(base_loc + get_pdb_type_helper(ntoskrnl_pdb,name,0));

}


uint32_t * get_pdb_type_uint32array(char *name, uint64_t base_loc) {


    uint32_t fld_offset,wsize;
    uint32_t * retval;
    uint32_t i;

    fld_offset = (uint32_t)
                 get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);

    retval = malloc(sizeof(uint32_t) * (wsize + 1));

    for (i=0; i < (wsize / 4); i++)
    retval[i] = memgrab_32_LE(base_loc + fld_offset + i);
    return retval;

}


uint32_t * pget_pdb_type_uint32array(char *name, uint64_t base_loc,
                                      uint64_t DTB) {


    uint32_t fld_offset,wsize;
    uint32_t * retval;
    uint32_t i;

    fld_offset = (uint32_t)
                       get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);
```

```
    retval = malloc(sizeof(uint32_t) * (wsize + 1));

    for (i=0; i < (wsize / 4); i++)
    retval[i] = memgrabp_32_LE(DTB, base_loc + fld_offset + i * 4);
    return retval;

}


uint32_t * keget_pdb_type_uint32array(char *name, uint64_t base_loc) {


    uint32_t fld_offset,wsize;
    uint32_t * retval;
    uint32_t i;

    fld_offset = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_DEFAULT);
    wsize = (uint32_t) get_pdb_type_helper(ntoskrnl_pdb,name,PDB_RET_SIZE);

    retval = malloc(sizeof(uint32_t) * (wsize + 1));

    for (i=0; i < (wsize / 4); i++)
    retval[i] = memgrabke_32_LE(base_loc + fld_offset + i * 4);
    return retval;

}


/****************************************************************************/
/**************************/
/* MISCELLANEOUS UTILITIES */
/**************************/

char * hexit(char * winput) {

    char *woutput, *wtmp;
    uint32_t i;
    if (winput[0] == 0) {

        woutput = malloc(sizeof(char) * 2);
        strcpy_s(woutput,3,"00");
        return woutput;

    }
```

```
        woutput = malloc(sizeof(char) * (strlen(winput) * 2));
        wtmp = malloc(sizeof(char) * (strlen(winput) * 2));
        for (i = 0; i < strlen(winput) * 2; i++) woutput[i] = 0;
        for (i = 0; i < strlen(winput); i++) {

            sprintf_s(wtmp,strlen(winput) * 2,"%s %x",wtmp,winput[i]);

        }
        strcpy_s(woutput, strlen(wtmp) + 1, wtmp);
        free(wtmp);
        return woutput;

    }
```

A.5 stub.c

```
/*******************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)            *
* Developer  : Jimmy Okolica                                        *
* Date       : 15-Aug-2011                                          *
*                                                                   *
*******************************************************************/

/*****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                          */
/* http://www.osedu.org/licenses/ECL-2.0                                    */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/*****************************************************************************/
```

```
#ifndef FILE_READ
#include "file_read.h"
#endif

#ifndef XEN
#include "xenaccess.h"
#endif

void xa_destroy(xa_instance_t *dummy) {}
int xa_init_vm_id_cmat(uint32_t dummy1, xa_instance_t *dummy2, int dummy3)
                {return 1;}
unsigned char* xa_access_pa(xa_instance_t *dummy1, uint32_t dummy2,
                uint32_t *dummy3, int dummy4)
                { char* x; x = malloc(sizeof(char)); return x;}
void munmap(unsigned char* dummy1, uint32_t dummy2) {}
```

A.6 cmat.h

```
/*********************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)             *
* Developer  : Jimmy Okolica                                        *
* Date       : 15-Aug-2011                                          *
*                                                                   *
*********************************************************************/

/*****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/*****************************************************************************/

#if defined WIN32

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
```

```
typedef unsigned long uint32_t;
typedef unsigned __int64 uint64_t;

typedef signed char int8_t;
typedef signed short int16_t;
typedef signed long int32_t;
typedef signed __int64 int64_t;

#include <process.h>
#include <ctype.h>
//typedef bool _Bool;

#else

#include <stdint.h>

#endif

// BEGIN -- CMAT_V
#ifdef WIN32
#ifndef XENH
#include <xenaccess.h>
#endif
#else
#include <xenaccess/xenaccess.h>
#endif
// END -- CMAT_V

/* Standard Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#ifndef FILE_READ
#include "file_read.h"
#endif

#ifndef PDB
#include "pdb_read.h"
#endif

#ifndef CMAT_REGISTRY
#include "cmat_registry.h"
```

```
#endif

typedef struct dbgkd_debug_data_type {
   uint16_t   MajorVersion ;
   uint16_t   MinorVersion ;
   uint16_t   MachineType ;
   uint64_t   KernBase ;
   uint64_t   pKernBase ;
   uint64_t   PsLoadedModuleList;
   uint64_t    ObpTypeObjectType;
   uint64_t *  objTypeTable;
   uint16_t   PaeEnabled ;
   char*      MachineString;
   uint16_t   MajorOSVersion;
   uint16_t   MinorOSVersion;
   uint64_t    kDTB; // a valid(?) kernel page directory table base
   uint64_t   KDBG_loc;
   uint16_t   x64;
   char*      OSVersionString; // CMAT-V
   char*      MemDumpFile;    // CMAT-V

} dbgkd_debug_data_t;

typedef struct win32k_type {
   pdb_type_t *pdb;
   uint32_t data_segment;
   uint32_t rdata_segment;
   uint64_t image_base;
} win32k_t;
```

A.7 cmat_registry.h

```
/*******************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)           *
* Developer  : Jimmy Okolica                                       *
* Date       : 15-Aug-2011                                         *
*                                                                  *
********************************************************************/

/****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
```

```
/*                                                                          */
/* Unless required by applicable law or agreed to in writing,               */
/* software distributed under the License is distributed on an              */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,             */
/* either express or implied. See the License for the specific              */
/* language governing permissions and limitations under the License.        */
/****************************************************************************/

#define CMAT_REGISTRY 1
#if defined WIN32

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned __int64 uint64_t;

typedef signed char int8_t;
typedef signed short int16_t;
typedef signed long int32_t;
typedef signed __int64 int64_t;

#include <process.h>
//typedef bool _Bool;

#else

#include <stdint.h>

#endif

/* Standard Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef MEMORY_STRUCTURES
#include "memory_structures.h"
#endif

#ifndef PDB
#include "pdb_read.h"
#endif

#ifndef FILE_READ
```

```
#include "file_read.h"
#endif

/*****************************************************************************/
/* Hive Structures */

typedef struct cell_info_type {
   uint64_t address;
   char * name;
} cell_info;

typedef struct key_object_type {
   // The first two entries are part of the key object.
   // I couldn't find the structure so had to guess that it is 0x008 bytes long
   uint64_t         unknown1;
   uint64_t         key_pointer;
   // The following entries are the CM_KEY_CONTROL_BLOCK structure pointed to by
   // key_pointer
   uint64_t         KeyHive;
   uint32_t         KeyCell;
   cell_info         cell;
   char *            hive_name;
} key_object;

typedef struct base_block_type {
  uint32_t    RootCell;
} hbase_block;

typedef struct dual_type {
  uint64_t       Map;
} dual;

typedef struct hhive_type {
    uint64_t    BaseBlockPtr;
   hbase_block   BaseBlock;
    dual      Storage[2];
} hhive;

typedef struct hive_type {
  hhive HHive;
  list_entry    HiveList;
  uint32_t       FileObject;
} hive_t;

typedef struct hivelist_type {
```

```
        struct hivelist_type *prev;
        struct hivelist_type *next;
      uint8_t   hive_in_link_list;
      hive_t Hive;
      char* name;
        uint64_t hive_location;
} hive_list_type;


/* Function Prototypes */

hive_list_type * add_new_hive (hive_list_type * hive_head, uint64_t hive_location);
uint8_t grab_hive (hive_list_type *tmp);
uint64_t get_key(hive_list_type *tmp, unsigned int sublist,
                 unsigned int num_subkeys, char *value);
user_list_type * get_users();
int load_hive_list();
uint64_t get_cell_loc(hive_list_type *tmp, uint32_t keycell);
key_object * get_key_object (process_list *sel_proc, uint64_t addr);
char * get_values(char * whive_name, char * wkey, char * wentry);
void print_hives();
void set_hive_list(hive_list_type *wtmp);
void setreg_x64(short new_value);
```

A.8 file_read.h

```
/*********************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)             *
* Developer  : Jimmy Okolica                                        *
* Date       : 15-Aug-2011                                          *
*                                                                   *
**********************************************************************/

/*****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
```

```
/*****************************************************************************/

#define FILE_READ

#if defined WIN32

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned __int64 uint64_t;

typedef signed char int8_t;
typedef signed short int16_t;
typedef signed long int32_t;
typedef signed __int64 int64_t;

//typedef bool _Bool;

#else

#include <stdint.h>

#endif

/* Standard Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// BEGIN -- CMAT_V
#ifdef WIN32
#ifndef XENH
#include <xenaccess.h>
#endif
#else
#include <xenaccess/xenaccess.h>
#endif// END -- CMAT_V

#ifndef PDB
#include "pdb_read.h"
#endif

#ifndef MEMORY_STRUCTURES
#include "memory_structures.h"
```

```
#endif

#ifndef CMAT_REGISTRY
#include "cmat_registry.h"
#endif


static uint32_t file_seg = 0x70000000;
//static uint32_t file_seg = 0x20000000;

typedef struct hardware_pte_x86_type {
   uint8_t            Valid;
   uint8_t            Write;
   uint8_t            Owner;
   uint8_t            WriteThrough;
   uint8_t            CacheDisable;
   uint8_t            Accessed;
   uint8_t            Dirty;
   uint8_t            LargePage;
   uint8_t            Global;
   uint8_t            CopyOnWrite;
   uint8_t            Prototype;
   uint8_t            Transition;
   uint32_t          PageFrameNumber;
   uint64_t          PageFrameNumber64;
   uint64_t          PageFrameNumber64_low;
   uint64_t          PageFrameNumber64_high;
} hardware_pte_x86;

enum { VALID = 1, WRITE = 2, OWNER = 4, WRITETHROUGH = 8, CACHEDISABLE = 16,
       ACCESSED = 32, DIRTY = 64, LARGEPAGE = 128, GLOBAL = 256, COPYONWRITE = 512,
       PROTOTYPE = 1024, RESERVED = 2048, PAGEFRAMENUMBER = 0xFFFFF000};

typedef enum _FILE_READ_PARMS_ENUM
{
   IS_PROTO = 1,
   NO_PROTO = 0,
   GET_VTOP = 3,
   GET_DATA = 4,
   IS_PAGEFILE = 5,
   IS_MAPFILE = 6,
} FILE_READ_PARMS_ENUM;

typedef struct dtb_list_type {
   struct dtb_list_type      *next;
```

```
   uint64_t dtb;
   uint64_t pid;
} dtb_list;

typedef struct ptov_list_type {
   struct ptov_list_type      *next;
   uint64_t *vaddress;
   uint64_t pid;
} ptov_list;

typedef struct ptov_ret_list_type {
   struct ptov_ret_list_type      *next;
   uint64_t vaddress;
   uint64_t pid;
} ptov_ret_list;

typedef struct pagefile_list_type {
   struct pagefile_list_type      *next;
   FILE * file_ptr;
   char * name;
   int found;
} pagefile_list;

typedef struct missingfile_list_type {
   struct missingfile_list_type      *next;
   FILE * file_ptr;
   char * name;
} missingfile_list;

typedef struct largefile_list_type {
   struct largefile_list_type      *next;
   FILE * file_ptr;
   FILE * mem_ptrs[50];
   uint32_t mem_2g;
   uint32_t cur_index;
   uint32_t cur_pos;
} largefile_list;

uint64_t get_mem_size();
void set_nopae(short new_value);
void set_x64(short new_value);
void set_ntoskrnl(pdb_type_t *wntoskrnl);
void set_kDTB(uint64_t wDTB);

int open_image_file( char *filename);
```

```
int open_page_file( char *filename);
FILE * get_pagefile(int pagefile_nbr);
void init_instance(uint32_t domain_num, xa_instance_t *instance);

void ptov_init();
ptov_ret_list *ptov(uint64_t physaddr);

void map_pte(uint32_t ptetemp, hardware_pte_x86 *pte);
void map_pte64(uint32_t ptetemp_high, uint32_t ptetemp_low, hardware_pte_x86 *pte);

unsigned char *read_proc_mem (uint64_t DTB, uint64_t start_address,
                              unsigned int length, uint8_t wis_proto);
unsigned char *read_proc_mem_helper (uint64_t DTB, uint64_t start_address,
           unsigned int length, uint8_t wis_proto, uint8_t do_what);
uint64_t vtop(uint64_t DTB, uint64_t start_address);
uint64_t kevtop(uint64_t start_address);

uint8_t figrab_8_LE(FILE *wfile, uint64_t pos);
uint32_t figrab_32_LE(FILE *wfile, uint64_t pos);

void set_instance(xa_instance_t instance); // CMAT_V

int fimove(FILE *wfile, uint64_t pos);
uint8_t firead(FILE *wfile);
int fieof(FILE *wfile);

int mem_move(uint64_t pos);
int mem_eof();
int mapfile(FILE *wfile);
void add_ptov_dtb( uint64_t pid, uint64_t dtb);

uint8_t  memgrab_8_LE (uint64_t pos);
uint16_t memgrab_16_LE(uint64_t pos);
uint32_t memgrab_32_LE(uint64_t pos);
uint64_t memgrab_48_LE(uint64_t pos);
uint64_t memgrab_64_LE(uint64_t pos);
char *   memgrab_unicode_LE (unicode_string ustr);
char *   memgrab_string_LE (ascii_string str);
char *   memgrab_string0(uint64_t address) ;

uint8_t  memgrab_8_BE (uint64_t pos);
uint16_t memgrab_16_BE(uint64_t pos);
uint32_t memgrab_32_BE(uint64_t pos);
uint64_t memgrab_64_BE(uint64_t pos);
```

```
uint8_t  memgrabp_8_LE (uint64_t DTB, uint64_t address);
uint16_t memgrabp_16_LE(uint64_t DTB, uint64_t address);
uint32_t memgrabp_32_LE(uint64_t DTB, uint64_t address);
uint64_t memgrabp_48_LE(uint64_t DTB, uint64_t address);
uint64_t memgrabp_64_LE(uint64_t DTB, uint64_t address);
char *   memgrabp_string_LE (ascii_string str, uint64_t DTB);
char *   memgrabp_string0(uint64_t address, uint64_t DTB);
char *   memgrabp_stringn(uint64_t address, int wlength, uint64_t DTB);
char *   memgrabp_unicode_LE (unicode_string ustr, uint64_t DTB);
char *   memgrabp_unicode_LE_space (unicode_string ustr, uint64_t DTB);
char *   memgrabp_unicode_LE_0 (uint64_t address, uint64_t DTB);

uint8_t memgrabp_8_BE(uint64_t DTB, uint64_t address);
uint16_t memgrabp_16_BE(uint64_t DTB, uint64_t address);
uint32_t memgrabp_32_BE(uint64_t DTB, uint64_t address);
uint64_t memgrabp_64_BE(uint64_t DTB, uint64_t address);

uint8_t  memgrabke_8_LE (uint64_t address);
uint16_t memgrabke_16_LE(uint64_t address);
uint32_t memgrabke_32_LE(uint64_t address);
uint64_t memgrabke_64_LE(uint64_t address);

uint8_t  memgrabke_8_BE (uint64_t address);
uint16_t memgrabke_16_BE(uint64_t address);
uint32_t memgrabke_32_BE(uint64_t address);
uint64_t memgrabke_64_BE(uint64_t address);
char *   memgrabke_string_LE (ascii_string str);
char *   memgrabke_unicode_LE (unicode_string ustr);
char *   memgrabke_unicode_LE_space (unicode_string ustr);
char *   memgrabke_unicode_LE_0 (uint64_t address);
char *   memgrabke_string0(uint64_t address) ;
char *   memgrabke_stringn(uint64_t address, int wlength) ;
```

A.9 memory_structures.h

```
/******************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)           *
* Developer  : Jimmy Okolica                                      *
* Date       : 15-Aug-2011                                        *
*                                                                 *
******************************************************************/

/****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community    */
/* License, Version 2.0 (the "License"); you may not use this file except in */
```

```
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/*****************************************************************************/

#define MEMORY_STRUCTURES
#if defined WIN32
#define LE
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned __int64 uint64_t;
//typedef unsigned long long uint64_t;

typedef signed char int8_t;
typedef signed short int16_t;
typedef signed long int32_t;
typedef signed __int64 int64_t;

//typedef bool _Bool;

#else

#include <stdint.h>

#define strcpy_s(dest, nbr, src) strcpy(dest, src)
#define strcat_s(dest, nbr, src) strcat(dest, src)
#define strncpy_s(dest, nbr, src, cnt) strncpy(dest, src, cnt)
#define gets_s(buf, sze) gets(buf)
#define fopen_s(fle, fname, mde) ( ! (*fle = fopen(fname, mde)) )
#define tmpfile_s(fle) *fle = tmpfile()
#define sprintf_s(buf, sze, fmt, ...) sprintf(buf, fmt, __VA_ARGS__)
#endif

//static uint32_t hive_mid3  = 0x434d3130u;

/******************************************************************************/
/* basic Windows OS structures */
```

```
typedef struct list_entry_type {
   uint64_t Flink;
   uint64_t Blink;
} list_entry;


typedef struct large_integer_type {
   uint32_t   LowPart;
   int32_t      HighPart;
} large_integer;

typedef struct unicode_string_type {
   uint16_t         Length;
   uint16_t         MaximumLength;
   uint64_t         Buffer;
} unicode_string;

typedef struct ascii_string_type {
   uint16_t         Length;
   uint16_t         MaximumLength;
   uint64_t         Buffer;
} ascii_string;

typedef struct userlist_type {
        struct userlist_type *prev;
        struct userlist_type *next;
      char * Token;
      char * Path;
      unsigned char * Sid;
      uint32_t SidSize;
      char * Name;
} user_list_type;

/******************************************************************************/
/* Process Structures */

typedef struct ethread_type {
   uint64_t         ThreadsProcess; // pointer _EPROCESS
} ethread;

typedef struct peb_type {
   uint64_t         ImageBaseAddress;
   uint64_t         Ldr; // pointer to _PEB_LDR_DATA
   uint64_t         ProcessParameters; // pointer to _RTL_USE_PROCESS_PARAMETERS
   uint32_t         NumberOfProcessors;
```

```
   uint32_t         OSMajorVersion;
   uint32_t         OSMinorVersion;
   uint16_t         OSBuildNumber;
   uint16_t         OSCSDVersion;
   uint32_t         OSPlatformID;
} peb;

typedef struct object_symbolic_link_type {
   large_integer   CreationTime;
   unicode_string   LinkTarget;
   unicode_string   LinkTargetRemaining;
   uint64_t      LinkTargetObject;
   uint64_t      DosDeviceDriveIndex;
} object_symbolic_link;

typedef struct kprocess_type {
   uint64_t         DirectoryTableBase;
   uint64_t         DirectoryTableBase_2; //For KPROCESS has DiectoryTableBase
                                          // [2] Uint4B ... we'll see what it means!
   uint64_t         KernelTime;
   uint64_t         UserTime;
   list_entry         ThreadListHead;
   signed char         BasePriority;
} kprocess;

typedef struct eprocess_type {
   kprocess         Pcb;
   large_integer      CreateTime;
   uint64_t         Win32WindowStation;
   uint64_t         UniqueProcessId;
   uint64_t         InheritedFromUniqueProcessId;
   list_entry         ActiveProcessLinks;
   uint64_t         PeakVirtualSize;
   uint64_t         DebugPort;
   uint64_t         ObjectTable;
   uint64_t         Token;
   uint64_t         DeviceMap;
   char *            ImageFileName2;
   list_entry         ThreadListHead;
   uint64_t         Peb;
   large_integer      ReadOperationCount;
   large_integer      WriteOperationCount;
   large_integer      OtherOperationCount;
   unsigned char      PriorityClass;
   uint32_t         SessionId;
```

```
} eprocess;

/*******************************************************************************/
// linked lists for processes, threads, users, hives (and anything else that
// comes along)

typedef struct socketlist_type {
        struct socketlist_type *prev;
        struct socketlist_type *next;
      char * LocalPort;
      char * RemotePort;
      char * Ipv6 ;
      unsigned char * Protocol;
} socket_list_type;


typedef struct process_list_type {
   struct process_list_type   *prev;
   struct process_list_type   *next;
   uint64_t               proc_eprocess_location;
   eprocess               proc_eprocess;
   uint8_t                  eprocess_in_link_list;
   peb                     proc_peb;
   user_list_type            *users;
   socket_list_type         *sockets;
        uint8_t                        checkmark;
} process_list;

typedef struct thread_list_type {
   struct thread_list_type      *prev;
   struct thread_list_type      *next;
   uint64_t               thread_list_location;
   ethread                  thr_ethread;
   process_list            *process;
} thread_list;
```

A.10 pdb_read.h

```
/*********************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)             *
* Developer  : Jimmy Okolica                                        *
* Date       : 15-Aug-2011                                          *
*                                                                   *
*********************************************************************/
```

```
/*****************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an               */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/*****************************************************************************/

#define PDB
#if defined WIN32

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned __int64 uint64_t;
typedef __int64 int64_t;

typedef signed char int8_t;
typedef signed short int16_t;
typedef signed long int32_t;
typedef signed __int64 int64_t;

#include <process.h>
//typedef bool _Bool;

#else

#include <stdint.h>
#include <unistd.h>

#endif

/* Standard Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef MEMORY_STRUCTURES
```

```
#include "memory_structures.h"
#endif

#include <curl/curl.h>
#include <curl/types.h>
#include <curl/easy.h>

typedef enum _PDB_PARMS_ENUM
    {
   PDB_DEFAULT    = 0,
   PDB_RET_SIZE   = 1,
   PDB_RET_SIZEOF = 2,
   PDB_RET_BASE   = 3,
   PDB_RET_URL    = 4,
   PDB_RET_NAME   = 5,
   PDB_DUMP_SYM   = 6,
   PDB_DUMP_STRUCT = 7,
    }
    PDB_PARMS_ENUM;

typedef struct pdb_file_type {
   FILE *      file_ptr;
   uint32_t   offset;
   uint32_t   byte_count;
   uint32_t   bytes_read;
   uint32_t   current_page;
   uint32_t   current_stream;
} pdb_file_t;

typedef struct pdb_symbol_type {
   uint16_t unknown1; //?
   uint32_t unknown2; //?
   uint32_t offset;
   uint16_t type;
   uint16_t namelen;
   char * name;
} pdb_symbol_t;

typedef struct pdb_symbols_type {
   uint32_t count;
   pdb_symbol_t * table;
} pdb_symbols_t;

typedef struct pdb_section_type {
   char * name;
```

```
   uint32_t virtual_size;
   uint32_t virtual_address;
   uint32_t raw_size;
   uint32_t raw_ptr;
   uint32_t relocation_ptr;
   uint32_t line_ptr;
   uint32_t relocation_count;
   uint32_t line_count;
   uint32_t characteristics;
} pdb_section_t;

typedef struct pdb_sections_type {
   uint32_t count;
   pdb_section_t *table;
} pdb_sections_t;

typedef struct pdb_structure_type {
   uint32_t index;
   uint32_t data_type;
   uint32_t underlying_type;
   uint32_t element_count;
   uint16_t properties;
   uint32_t derived;
   uint32_t vshape;
   uint32_t size;
   char * name;
   uint32_t pointer_array;
   uint32_t offset;
   uint32_t field_index;
   uint32_t *  arguments;
   uint32_t retval_type;
   uint32_t call_type;
   uint32_t value;
   uint32_t index_type;
   uint32_t field_count;
    struct pdb_structure_type *field_list;
} pdb_structure_t;

typedef struct pdb_structures_type {
   uint32_t count;
   uint32_t version;
   uint32_t header_size;
   uint32_t base_index;
   pdb_structure_t * table;
} pdb_structures_t;
```

```
typedef struct pdb_type {
   char *      file_name;
   char *      pdb_url;
   pdb_file_t   pdb_file;
   uint32_t   magic_number;
   uint32_t   bytes_per_page;
   uint32_t   start_page;
   uint32_t   nbr_pages;
   uint32_t   root_size;
   uint32_t   root_ptr;
   uint32_t   stream_count;
   uint32_t *   stream_size;
   uint32_t *   stream_ptr;
   pdb_symbols_t symbols;
   pdb_sections_t sections;
   pdb_structures_t structures;
} pdb_type_t;

typedef struct pdb_array_attr_type {
   uint32_t   base_offset;  // how far the array is from the beginning of the structure
   uint32_t   size;        // how large one element of the array is
   uint32_t   elements;     // how many elements the array holds
} pdb_array_attr_t;

/* Function Prototypes */
uint8_t    pdb_grab_8_LE(pdb_type_t *pdb);
uint16_t   pdb_grab_16_LE(pdb_type_t *pdb);
uint32_t   pdb_grab_32_LE(pdb_type_t *pdb);
char *     pdb_grab_string(pdb_type_t *pdb, uint32_t wsize);
void       pdb_goto_stream(pdb_type_t *pdb,uint32_t stream_nbr, uint32_t page_nbr);
int        pdb_eof_stream(pdb_type_t *pdb);
pdb_file_t pdb_set_point(pdb_type_t *pdb);
void       pdb_goto_point(pdb_type_t *pdb, pdb_file_t wpdb_file);
uint32_t   pdb_get_bytes_read(pdb_type_t *pdb);

char *    download_pdb_file (char *wfile_name, char *wpdb_url, char *datadirectory);
pdb_type_t *    parse_pdb_file (char *wfile_name, int parm);
pdb_section_t * get_pdb_section(pdb_type_t *wpdb, char *name, uint32_t length);
pdb_symbol_t *  get_pdb_symbol (pdb_type_t *wpdb, char *name, uint32_t length);
uint32_t        get_pdb_symbol_loc(pdb_type_t *wpdb, uint32_t offset,
                      uint32_t wdata_section);
pdb_structure_t get_pdb_struct(pdb_type_t *pdb, uint16_t rec_size);

char *      get_data_type(pdb_structure_t *pdb_table, uint32_t wdata_type,
```

```
                          uint32_t base_idx, int pdb_parm);
void      print_pdb_struct(pdb_structure_t *pdb_table, pdb_structure_t * wfld,
                          uint32_t base_idx, uint8_t indent);

char * hexit (char * winput);

void set_ntoskrnl_pdb(pdb_type_t *wpdb);
void set_pdb_x64(short new_value);
void set_dump_symbols(char *fname);
void set_dump_structs(char *fname);
uint8_t have_ntoskrnl_pdb();

uint64_t get_pdb_type_offset2(pdb_structures_t *pdb_struct, char *fldname,
                               char *subfldname, int woffset);
char * get_pdb_type_utype(pdb_structures_t *pdb_struct, char *fldname,
                          char *subfldname);
uint64_t get_pdb_type_helper(pdb_structures_t *pdb_struct, char *name, int do_size);

uint32_t get_pdb_type_uint32(char *name, uint64_t base_loc);
uint16_t get_pdb_type_uint16(char *name, uint64_t base_loc);
uint8_t get_pdb_type_uint8(char *name, uint64_t base_loc);
uint32_t * get_pdb_type_uint32array(char *name, uint64_t base_loc);
char * get_pdb_type_string(char *name, uint64_t base_loc);

uint32_t pget_pdb_type_uint32(char *name, uint64_t base_loc, uint64_t DTB);
uint16_t pget_pdb_type_uint16(char *name, uint64_t base_loc, uint64_t DTB);
uint8_t pget_pdb_type_uint8(char *name, uint64_t base_loc, uint64_t DTB);
uint32_t * pget_pdb_type_uint32array(char *name, uint64_t base_loc, uint64_t DTB);
char * pget_pdb_type_string(char *name, uint64_t base_loc, uint64_t DTB);
char * pget_pdb_type_unicode(char *name,  uint64_t base_loc, uint64_t DTB);
char * pget_pdb_type_unicoden(char *name, uint64_t base_loc, uint64_t DTB,
                               uint16_t wlength);
char * pget_pdb_type_stringn(char *name, uint64_t base_loc, uint64_t DTB,
                               uint16_t wlength);

uint32_t keget_pdb_type_uint32(char *name, uint64_t base_loc);
uint16_t keget_pdb_type_uint16(char *name, uint64_t base_loc);
uint8_t keget_pdb_type_uint8(char *name, uint64_t base_loc);
uint32_t * keget_pdb_type_uint32array(char *name, uint64_t base_loc);
char * keget_pdb_type_string(char *name, uint64_t base_loc);
char * keget_pdb_type_unicode(char *name, uint64_t base_loc);
char * keget_pdb_type_unicoden(char *name, uint64_t base_loc, uint16_t wlength);
char * keget_pdb_type_stringn(char *name, uint64_t base_loc, uint16_t wlength);

uint64_t get_pdb_type(char *name, uint64_t base_loc);
```

```
uint64_t pget_pdb_type(char *name, uint64_t base_loc, uint64_t DTB);
uint64_t keget_pdb_type(char *name, uint64_t base_loc);
uint64_t * get_pdb_type_array(char *name, uint64_t base_loc);
uint64_t * pget_pdb_type_array(char *name, uint64_t base_loc, uint64_t DTB);
uint64_t * keget_pdb_type_array(char *name, uint64_t base_loc);

pdb_array_attr_t get_pdb_type_array_attr(char *name);
uint64_t get_pdb_type_sizeof(char *fldname);
uint64_t get_pdb_type_offset(char *fldname);
uint32_t * get_pdb_type_value(char *name, uint32_t value);
char * get_pdb_url(uint64_t wimage_base, int pdb_parm, int is_x64, uint64_t kDTB);
char * pget_pdb_url(uint64_t wimage_base, int pdb_parm, int is_x64);
char * get_pdb_url_disk(char *file_name, int pdb_parm, int is_x64);

uint8_t fgrab_8_LE(FILE *image_file);
uint16_t fgrab_16_LE(FILE *image_file);
uint32_t fgrab_32_LE(FILE *image_file);
uint32_t fgrab_32_BE(FILE *image_file);
char * fgrab_string_LE (FILE *image_file, ascii_string str);
char * fgrab_string0(FILE *wfile);

uint8_t get_pdb_fld_exists(char *name);

// pdb_info.h
// 06-25-2006 Sven B. Schreiber
// sbs@orgon.com

// ==================================================================
// PDB INTERFACE VERSIONS
// ==================================================================

#define PDBIntv41         920924
#define PDBIntv50       19960502
#define PDBIntv50a      19970116
#define PDBIntv60       19970116
#define PDBIntv61       19980914
#define PDBIntv69       19990511
#define PDBIntv70Dep    20000406
#define PDBIntv70       20001102

#define PDBIntv         PDBIntv70

#define PDBIntvAlt      PDBIntv50
#define PDBIntvAlt2     PDBIntv60
#define PDBIntvAlt3     PDBIntv69
```

```
// ====================================================================
// PDB IMPLEMENTATION VERSIONS
// ====================================================================

#define PDBImpvVC2      19941610
#define PDBImpvVC4      19950623
#define PDBImpvVC41     19950814
#define PDBImpvVC50     19960307
#define PDBImpvVC98     19970604
#define PDBImpvVC70Dep  19990604
#define PDBImpvVC70     20000404

#define PDBImpv         PDBImpvVC70

// ====================================================================
// DBI IMPLEMENTATION VERSIONS
// ====================================================================

#define DBIImpvV41        930803
#define DBIImpvV50      19960307
#define DBIImpvV60      19970606
#define DBIImpvV70      19990903

#define DBIImpv         DBIImpvV70

// ====================================================================
// BASIC TYPES
// ====================================================================

typedef enum _TYPE_ENUM_e
    {
    T_NOTYPE                = 0x00000000,
    T_ABS                   = 0x00000001,
    T_SEGMENT               = 0x00000002,
    T_VOID                  = 0x00000003,

    T_HRESULT               = 0x00000008,
    T_32PHRESULT            = 0x00000408,
    T_64PHRESULT            = 0x00000608,

    T_PVOID                 = 0x00000103,
    T_PFVOID                = 0x00000203,
    T_PHVOID                = 0x00000303,
    T_32PVOID               = 0x00000403,
```

```
T_32PFVOID              = 0x00000503,
T_64PVOID               = 0x00000603,

T_CURRENCY              = 0x00000004,
T_NBASICSTR             = 0x00000005,
T_FBASICSTR             = 0x00000006,
T_NOTTRANS              = 0x00000007,
T_BIT                   = 0x00000060,
T_PASCHAR               = 0x00000061,

T_CHAR                  = 0x00000010,
T_PCHAR                 = 0x00000110,
T_PFCHAR                = 0x00000210,
T_PHCHAR                = 0x00000310,
T_32PCHAR               = 0x00000410,
T_32PFCHAR              = 0x00000510,
T_64PCHAR               = 0x00000610,

T_UCHAR                 = 0x00000020,
T_PUCHAR                = 0x00000120,
T_PFUCHAR               = 0x00000220,
T_PHUCHAR               = 0x00000320,
T_32PUCHAR              = 0x00000420,
T_32PFUCHAR             = 0x00000520,
T_64PUCHAR              = 0x00000620,

T_RCHAR                 = 0x00000070,
T_PRCHAR                = 0x00000170,
T_PFRCHAR               = 0x00000270,
T_PHRCHAR               = 0x00000370,
T_32PRCHAR              = 0x00000470,
T_32PFRCHAR             = 0x00000570,
T_64PRCHAR              = 0x00000670,

T_WCHAR                 = 0x00000071,
T_PWCHAR                = 0x00000171,
T_PFWCHAR               = 0x00000271,
T_PHWCHAR               = 0x00000371,
T_32PWCHAR              = 0x00000471,
T_32PFWCHAR             = 0x00000571,
T_64PWCHAR              = 0x00000671,

T_INT1                  = 0x00000068,
T_PINT1                 = 0x00000168,
T_PFINT1                = 0x00000268,
```

```
T_PHINT1                 = 0x00000368,
T_32PINT1                = 0x00000468,
T_32PFINT1               = 0x00000568,
T_64PINT1                = 0x00000668,

T_UINT1                  = 0x00000069,
T_PUINT1                 = 0x00000169,
T_PFUINT1                = 0x00000269,
T_PHUINT1                = 0x00000369,
T_32PUINT1               = 0x00000469,
T_32PFUINT1              = 0x00000569,
T_64PUINT1               = 0x00000669,

T_SHORT                  = 0x00000011,
T_PSHORT                 = 0x00000111,
T_PFSHORT                = 0x00000211,
T_PHSHORT                = 0x00000311,
T_32PSHORT               = 0x00000411,
T_32PFSHORT              = 0x00000511,
T_64PSHORT               = 0x00000611,

T_USHORT                 = 0x00000021,
T_PUSHORT                = 0x00000121,
T_PFUSHORT               = 0x00000221,
T_PHUSHORT               = 0x00000321,
T_32PUSHORT              = 0x00000421,
T_32PFUSHORT             = 0x00000521,
T_64PUSHORT              = 0x00000621,

T_INT2                   = 0x00000072,
T_PINT2                  = 0x00000172,
T_PFINT2                 = 0x00000272,
T_PHINT2                 = 0x00000372,
T_32PINT2                = 0x00000472,
T_32PFINT2               = 0x00000572,
T_64PINT2                = 0x00000672,

T_UINT2                  = 0x00000073,
T_PUINT2                 = 0x00000173,
T_PFUINT2                = 0x00000273,
T_PHUINT2                = 0x00000373,
T_32PUINT2               = 0x00000473,
T_32PFUINT2              = 0x00000573,
T_64PUINT2               = 0x00000673,
```

```
T_LONG                  = 0x00000012,
T_PLONG                 = 0x00000112,
T_PFLONG                = 0x00000212,
T_PHLONG                = 0x00000312,
T_32PLONG               = 0x00000412,
T_32PFLONG              = 0x00000512,
T_64PLONG               = 0x00000612,

T_ULONG                 = 0x00000022,
T_PULONG                = 0x00000122,
T_PFULONG               = 0x00000222,
T_PHULONG               = 0x00000322,
T_32PULONG              = 0x00000422,
T_32PFULONG             = 0x00000522,
T_64PULONG              = 0x00000622,

T_INT4                  = 0x00000074,
T_PINT4                 = 0x00000174,
T_PFINT4                = 0x00000274,
T_PHINT4                = 0x00000374,
T_32PINT4               = 0x00000474,
T_32PFINT4              = 0x00000574,
T_64PINT4               = 0x00000674,

T_UINT4                 = 0x00000075,
T_PUINT4                = 0x00000175,
T_PFUINT4               = 0x00000275,
T_PHUINT4               = 0x00000375,
T_32PUINT4              = 0x00000475,
T_32PFUINT4             = 0x00000575,
T_64PUINT4              = 0x00000675,

T_QUAD                  = 0x00000013,
T_PQUAD                 = 0x00000113,
T_PFQUAD                = 0x00000213,
T_PHQUAD                = 0x00000313,
T_32PQUAD               = 0x00000413,
T_32PFQUAD              = 0x00000513,
T_64PQUAD               = 0x00000613,

T_UQUAD                 = 0x00000023,
T_PUQUAD                = 0x00000123,
T_PFUQUAD               = 0x00000223,
T_PHUQUAD               = 0x00000323,
T_32PUQUAD              = 0x00000423,
```

```
T_32PFUQUAD           = 0x00000523,
T_64PUQUAD            = 0x00000623,

T_INT8                = 0x00000076,
T_PINT8               = 0x00000176,
T_PFINT8              = 0x00000276,
T_PHINT8              = 0x00000376,
T_32PINT8             = 0x00000476,
T_32PFINT8            = 0x00000576,
T_64PINT8             = 0x00000676,

T_UINT8               = 0x00000077,
T_PUINT8              = 0x00000177,
T_PFUINT8             = 0x00000277,
T_PHUINT8             = 0x00000377,
T_32PUINT8            = 0x00000477,
T_32PFUINT8           = 0x00000577,
T_64PUINT8            = 0x00000677,

T_OCT                 = 0x00000014,
T_POCT                = 0x00000114,
T_PFOCT               = 0x00000214,
T_PHOCT               = 0x00000314,
T_32POCT              = 0x00000414,
T_32PFOCT             = 0x00000514,
T_64POCT              = 0x00000614,

T_UOCT                = 0x00000024,
T_PUOCT               = 0x00000124,
T_PFUOCT              = 0x00000224,
T_PHUOCT              = 0x00000324,
T_32PUOCT             = 0x00000424,
T_32PFUOCT            = 0x00000524,
T_64PUOCT             = 0x00000624,

T_INT16               = 0x00000078,
T_PINT16              = 0x00000178,
T_PFINT16             = 0x00000278,
T_PHINT16             = 0x00000378,
T_32PINT16            = 0x00000478,
T_32PFINT16           = 0x00000578,
T_64PINT16            = 0x00000678,

T_UINT16              = 0x00000079,
T_PUINT16             = 0x00000179,
```

```
T_PFUINT16              = 0x00000279,
T_PHUINT16              = 0x00000379,
T_32PUINT16             = 0x00000479,
T_32PFUINT16            = 0x00000579,
T_64PUINT16             = 0x00000679,

T_REAL32                = 0x00000040,
T_PREAL32               = 0x00000140,
T_PFREAL32              = 0x00000240,
T_PHREAL32              = 0x00000340,
T_32PREAL32             = 0x00000440,
T_32PFREAL32            = 0x00000540,
T_64PREAL32             = 0x00000640,

T_REAL48                = 0x00000044,
T_PREAL48               = 0x00000144,
T_PFREAL48              = 0x00000244,
T_PHREAL48              = 0x00000344,
T_32PREAL48             = 0x00000444,
T_32PFREAL48            = 0x00000544,
T_64PREAL48             = 0x00000644,

T_REAL64                = 0x00000041,
T_PREAL64               = 0x00000141,
T_PFREAL64              = 0x00000241,
T_PHREAL64              = 0x00000341,
T_32PREAL64             = 0x00000441,
T_32PFREAL64            = 0x00000541,
T_64PREAL64             = 0x00000641,

T_REAL80                = 0x00000042,
T_PREAL80               = 0x00000142,
T_PFREAL80              = 0x00000242,
T_PHREAL80              = 0x00000342,
T_32PREAL80             = 0x00000442,
T_32PFREAL80            = 0x00000542,
T_64PREAL80             = 0x00000642,

T_REAL128               = 0x00000043,
T_PREAL128              = 0x00000143,
T_PFREAL128             = 0x00000243,
T_PHREAL128             = 0x00000343,
T_32PREAL128            = 0x00000443,
T_32PFREAL128           = 0x00000543,
T_64PREAL128            = 0x00000643,
```

```
T_CPLX32              = 0x00000050,
T_PCPLX32             = 0x00000150,
T_PFCPLX32            = 0x00000250,
T_PHCPLX32            = 0x00000350,
T_32PCPLX32           = 0x00000450,
T_32PFCPLX32          = 0x00000550,
T_64PCPLX32           = 0x00000650,

T_CPLX64              = 0x00000051,
T_PCPLX64             = 0x00000151,
T_PFCPLX64            = 0x00000251,
T_PHCPLX64            = 0x00000351,
T_32PCPLX64           = 0x00000451,
T_32PFCPLX64          = 0x00000551,
T_64PCPLX64           = 0x00000651,

T_CPLX80              = 0x00000052,
T_PCPLX80             = 0x00000152,
T_PFCPLX80            = 0x00000252,
T_PHCPLX80            = 0x00000352,
T_32PCPLX80           = 0x00000452,
T_32PFCPLX80          = 0x00000552,
T_64PCPLX80           = 0x00000652,

T_CPLX128             = 0x00000053,
T_PCPLX128            = 0x00000153,
T_PFCPLX128           = 0x00000253,
T_PHCPLX128           = 0x00000353,
T_32PCPLX128          = 0x00000453,
T_32PFCPLX128         = 0x00000553,
T_64PCPLX128          = 0x00000653,

T_BOOL08              = 0x00000030,
T_PBOOL08             = 0x00000130,
T_PFBOOL08            = 0x00000230,
T_PHBOOL08            = 0x00000330,
T_32PBOOL08           = 0x00000430,
T_32PFBOOL08          = 0x00000530,
T_64PBOOL08           = 0x00000630,

T_BOOL16              = 0x00000031,
T_PBOOL16             = 0x00000131,
T_PFBOOL16            = 0x00000231,
T_PHBOOL16            = 0x00000331,
```

```
    T_32PBOOL16             = 0x00000431,
    T_32PFBOOL16            = 0x00000531,
    T_64PBOOL16             = 0x00000631,

    T_BOOL32                = 0x00000032,
    T_PBOOL32               = 0x00000132,
    T_PFBOOL32              = 0x00000232,
    T_PHBOOL32              = 0x00000332,
    T_32PBOOL32             = 0x00000432,
    T_32PFBOOL32            = 0x00000532,
    T_64PBOOL32             = 0x00000632,

    T_BOOL64                = 0x00000033,
    T_PBOOL64               = 0x00000133,
    T_PFBOOL64              = 0x00000233,
    T_PHBOOL64              = 0x00000333,
    T_32PBOOL64             = 0x00000433,
    T_32PFBOOL64            = 0x00000533,
    T_64PBOOL64             = 0x00000633,

    T_NCVPTR                = 0x000001F0,
    T_FCVPTR                = 0x000002F0,
    T_HCVPTR                = 0x000003F0,
    T_32NCVPTR              = 0x000004F0,
    T_32FCVPTR              = 0x000005F0,
    T_64NCVPTR              = 0x000006F0,
    }
    TYPE_ENUM_e, *PTYPE_ENUM_e, **PPTYPE_ENUM_e;

// ==================================================================
// TYPE INFO RECORD TAGS
// ==================================================================

typedef enum _LEAF_ENUM_e
    {
    LF_MODIFIER_16t         = 0x00000001,
    LF_POINTER_16t          = 0x00000002,
    LF_ARRAY_16t            = 0x00000003,
    LF_CLASS_16t            = 0x00000004,
    LF_STRUCTURE_16t        = 0x00000005,
    LF_UNION_16t            = 0x00000006,
    LF_ENUM_16t             = 0x00000007,
    LF_PROCEDURE_16t        = 0x00000008,
    LF_MFUNCTION_16t        = 0x00000009,
    LF_VTSHAPE              = 0x0000000A,
```

```
LF_COBOL0_16t         = 0x0000000B,
LF_COBOL1             = 0x0000000C,
LF_BARRAY_16t         = 0x0000000D,
LF_LABEL              = 0x0000000E,
LF_NULL               = 0x0000000F,
LF_NOTTRAN            = 0x00000010,
LF_DIMARRAY_16t       = 0x00000011,
LF_VFTPATH_16t        = 0x00000012,
LF_PRECOMP_16t        = 0x00000013,
LF_ENDPRECOMP         = 0x00000014,
LF_OEM_16t            = 0x00000015,
LF_TYPESERVER_ST      = 0x00000016,

LF_SKIP_16t           = 0x00000200,
LF_ARGLIST_16t        = 0x00000201,
LF_DEFARG_16t         = 0x00000202,
LF_LIST               = 0x00000203,
LF_FIELDLIST_16t      = 0x00000204,
LF_DERIVED_16t        = 0x00000205,
LF_BITFIELD_16t       = 0x00000206,
LF_METHODLIST_16t     = 0x00000207,
LF_DIMCONU_16t        = 0x00000208,
LF_DIMCONLU_16t       = 0x00000209,
LF_DIMVARU_16t        = 0x0000020A,
LF_DIMVARLU_16t       = 0x0000020B,
LF_REFSYM             = 0x0000020C,

LF_BCLASS_16t         = 0x00000400,
LF_VBCLASS_16t        = 0x00000401,
LF_IVBCLASS_16t       = 0x00000402,
LF_ENUMERATE_ST       = 0x00000403,
LF_FRIENDFCN_16t      = 0x00000404,
LF_INDEX_16t          = 0x00000405,
LF_MEMBER_16t         = 0x00000406,
LF_STMEMBER_16t       = 0x00000407,
LF_METHOD_16t         = 0x00000408,
LF_NESTTYPE_16t       = 0x00000409,
LF_VFUNCTAB_16t       = 0x0000040A,
LF_FRIENDCLS_16t      = 0x0000040B,
LF_ONEMETHOD_16t      = 0x0000040C,
LF_VFUNCOFF_16t       = 0x0000040D,

LF_TI16_MAX           = 0x00001000,
LF_MODIFIER           = 0x00001001,
LF_POINTER            = 0x00001002,
```

```
LF_ARRAY_ST           = 0x00001003,
LF_CLASS_ST           = 0x00001004,
LF_STRUCTURE_ST       = 0x00001005,
LF_UNION_ST           = 0x00001006,
LF_ENUM_ST            = 0x00001007,
LF_PROCEDURE          = 0x00001008,
LF_MFUNCTION          = 0x00001009,
LF_COBOL0             = 0x0000100A,
LF_BARRAY             = 0x0000100B,
LF_DIMARRAY_ST        = 0x0000100C,
LF_VFTPATH            = 0x0000100D,
LF_PRECOMP_ST         = 0x0000100E,
LF_OEM                = 0x0000100F,
LF_ALIAS_ST           = 0x00001010,
LF_OEM2               = 0x00001011,

LF_SKIP               = 0x00001200,
LF_ARGLIST            = 0x00001201,
LF_DEFARG_ST          = 0x00001202,
LF_FIELDLIST          = 0x00001203,
LF_DERIVED            = 0x00001204,
LF_BITFIELD           = 0x00001205,
LF_METHODLIST         = 0x00001206,
LF_DIMCONU            = 0x00001207,
LF_DIMCONLU           = 0x00001208,
LF_DIMVARU            = 0x00001209,
LF_DIMVARLU           = 0x0000120A,

LF_BCLASS             = 0x00001400,
LF_VBCLASS            = 0x00001401,
LF_IVBCLASS           = 0x00001402,
LF_FRIENDFCN_ST       = 0x00001403,
LF_INDEX              = 0x00001404,
LF_MEMBER_ST          = 0x00001405,
LF_STMEMBER_ST        = 0x00001406,
LF_METHOD_ST          = 0x00001407,
LF_NESTTYPE_ST        = 0x00001408,
LF_VFUNCTAB           = 0x00001409,
LF_FRIENDCLS          = 0x0000140A,
LF_ONEMETHOD_ST       = 0x0000140B,
LF_VFUNCOFF           = 0x0000140C,
LF_NESTTYPEEX_ST      = 0x0000140D,
LF_MEMBERMODIFY_ST    = 0x0000140E,
LF_MANAGED_ST         = 0x0000140F,
```

```
LF_ST_MAX               = 0x00001500,
LF_TYPESERVER           = 0x00001501,
LF_ENUMERATE            = 0x00001502,
LF_ARRAY                = 0x00001503,
LF_CLASS                = 0x00001504,
LF_STRUCTURE            = 0x00001505,
LF_UNION                = 0x00001506,
LF_ENUM                 = 0x00001507,
LF_DIMARRAY             = 0x00001508,
LF_PRECOMP              = 0x00001509,
LF_ALIAS                = 0x0000150A,
LF_DEFARG               = 0x0000150B,
LF_FRIENDFCN            = 0x0000150C,
LF_MEMBER               = 0x0000150D,
LF_STMEMBER             = 0x0000150E,
LF_METHOD               = 0x0000150F,
LF_NESTTYPE             = 0x00001510,
LF_ONEMETHOD            = 0x00001511,
LF_NESTTYPEEX           = 0x00001512,
LF_MEMBERMODIFY         = 0x00001513,
LF_MANAGED              = 0x00001514,
LF_TYPESERVER2          = 0x00001515,

LF_NUMERIC              = 0x00008000,
LF_CHAR                 = 0x00008000,
LF_SHORT                = 0x00008001,
LF_USHORT               = 0x00008002,
LF_LONG                 = 0x00008003,
LF_ULONG                = 0x00008004,
LF_REAL32               = 0x00008005,
LF_REAL64               = 0x00008006,
LF_REAL80               = 0x00008007,
LF_REAL128              = 0x00008008,
LF_QUADWORD             = 0x00008009,
LF_UQUADWORD            = 0x0000800A,
LF_REAL48               = 0x0000800B,
LF_COMPLEX32            = 0x0000800C,
LF_COMPLEX64            = 0x0000800D,
LF_COMPLEX80            = 0x0000800E,
LF_COMPLEX128           = 0x0000800F,
LF_VARSTRING            = 0x00008010,
LF_OCTWORD              = 0x00008017,
LF_UOCTWORD             = 0x00008018,
LF_DECIMAL              = 0x00008019,
LF_DATE                 = 0x0000801A,
```

```
    LF_UTF8STRING           = 0x0000801B,

    LF_PAD0                 = 0x000000F0,
    LF_PAD1                 = 0x000000F1,
    LF_PAD2                 = 0x000000F2,
    LF_PAD3                 = 0x000000F3,
    LF_PAD4                 = 0x000000F4,
    LF_PAD5                 = 0x000000F5,
    LF_PAD6                 = 0x000000F6,
    LF_PAD7                 = 0x000000F7,
    LF_PAD8                 = 0x000000F8,
    LF_PAD9                 = 0x000000F9,
    LF_PAD10                = 0x000000FA,
    LF_PAD11                = 0x000000FB,
    LF_PAD12                = 0x000000FC,
    LF_PAD13                = 0x000000FD,
    LF_PAD14                = 0x000000FE,
    LF_PAD15                = 0x000000FF,
    }
    LEAF_ENUM_e, *PLEAF_ENUM_e, **PPLEAF_ENUM_e;

// =================================================================
// CALLING CONVENTIONS
// =================================================================

typedef enum _CV_call_e
    {
    CV_CALL_NEAR_C          = 0x00000000,
    CV_CALL_FAR_C           = 0x00000001,
    CV_CALL_NEAR_PASCAL     = 0x00000002,
    CV_CALL_FAR_PASCAL      = 0x00000003,
    CV_CALL_NEAR_FAST       = 0x00000004,
    CV_CALL_FAR_FAST        = 0x00000005,
    CV_CALL_SKIPPED         = 0x00000006,
    CV_CALL_NEAR_STD        = 0x00000007,
    CV_CALL_FAR_STD         = 0x00000008,
    CV_CALL_NEAR_SYS        = 0x00000009,
    CV_CALL_FAR_SYS         = 0x0000000A,
    CV_CALL_THISCALL        = 0x0000000B,
    CV_CALL_MIPSCALL        = 0x0000000C,
    CV_CALL_GENERIC         = 0x0000000D,
    CV_CALL_ALPHACALL       = 0x0000000E,
    CV_CALL_PPCCALL         = 0x0000000F,
    CV_CALL_SHCALL          = 0x00000010,
    CV_CALL_ARMCALL         = 0x00000011,
```

```
    CV_CALL_AM33CALL        = 0x00000012,
    CV_CALL_TRICALL         = 0x00000013,
    CV_CALL_SH5CALL         = 0x00000014,
    CV_CALL_M32RCALL        = 0x00000015,
    CV_CALL_RESERVED        = 0x00000016,
    }
    CV_call_e, *PCV_call_e, **PPCV_call_e;

// ==================================================================
// POINTER TYPES
// ==================================================================

typedef enum _CV_ptrtype_e
    {
    CV_PTR_NEAR             = 0x00000000,
    CV_PTR_FAR              = 0x00000001,
    CV_PTR_HUGE             = 0x00000002,
    CV_PTR_BASE_SEG         = 0x00000003,
    CV_PTR_BASE_VAL         = 0x00000004,
    CV_PTR_BASE_SEGVAL      = 0x00000005,
    CV_PTR_BASE_ADDR        = 0x00000006,
    CV_PTR_BASE_SEGADDR     = 0x00000007,
    CV_PTR_BASE_TYPE        = 0x00000008,
    CV_PTR_BASE_SELF        = 0x00000009,
    CV_PTR_NEAR32           = 0x0000000A,
    CV_PTR_FAR32            = 0x0000000B,
    CV_PTR_64               = 0x0000000C,
    CV_PTR_UNUSEDPTR        = 0x0000000D,
    }
    CV_ptrtype_e, *PCV_ptrtype_e, **PPCV_ptrtype_e;

// ==================================================================
// POINTER MODES
// ==================================================================

typedef enum _CV_ptrmode_e
    {
    CV_PTR_MODE_PTR         = 0x00000000,
    CV_PTR_MODE_REF         = 0x00000001,
    CV_PTR_MODE_PMEM        = 0x00000002,
    CV_PTR_MODE_PMFUNC      = 0x00000003,
    CV_PTR_MODE_RESERVED    = 0x00000004,
    }
    CV_ptrmode_e, *PCV_ptrmode_e, **PPCV_ptrmode_e;
```

```
// ====================================================================
// ACCESS PROTECTION MODES
// ====================================================================

typedef enum _CV_access_e
    {
    CV_private             = 0x00000001,
    CV_protected           = 0x00000002,
    CV_public              = 0x00000003,
    }
    CV_access_e, *PCV_access_e, **PPCV_access_e;

// ====================================================================
// METHOD PROPERTIES
// ====================================================================


// --------------------------------------------------------------------

//typedef struct _CV_prop_t
//         {
// /*000.0*/ WORD packed   : 1;
// /*000.1*/ WORD ctor     : 1;
// /*000.2*/ WORD ovlops   : 1;
// /*000.3*/ WORD isnested : 1;
// /*000.4*/ WORD cnested  : 1;
// /*000.5*/ WORD opassign : 1;
// /*000.6*/ WORD opcast   : 1;
// /*000.7*/ WORD fwdref   : 1;
// /*001.0*/ WORD scoped   : 1;
// /*001.1*/ WORD reserved : 7;
// /*002*/ }
//          CV_prop_t, *PCV_prop_t, **PPCV_prop_t;

// #define CV_prop_t_ sizeof (CV_prop_t)

// --------------------------------------------------------------------

// typedef struct _CV_fldattr_t
//         {
// /*000.0*/ WORD access      : 2; // CV_access_e
// /*000.2*/ WORD mprop       : 3; // CV_methodprop_e
// /*000.5*/ WORD pseudo      : 1;
// /*000.6*/ WORD noinherit   : 1;
// /*000.7*/ WORD noconstruct : 1;
```

```
// /*001.0*/ WORD compgenx    : 1;
// /*001.1*/ WORD unused      : 7;
// /*002*/ }
//        CV_fldattr_t, *PCV_fldattr_t, **PPCV_fldattr_t;

//#define CV_fldattr_t_ sizeof (CV_fldattr_t)
```

A.11 xenaccess.h

```
/*****************************************************************
* Description: Compiled Memory Analysis Tool (CMAT.exe)          *
* Developer  : Jimmy Okolica                                     *
* Date       : 15-Aug-2011                                       *
*                                                                *
*****************************************************************/

/***************************************************************************/
/* Copyright 2011 James Okolica Licensed under the Educational Community     */
/* License, Version 2.0 (the "License"); you may not use this file except in */
/* compliance with the License. You may obtain a copy of the License at      */
/*                                                                           */
/* http://www.osedu.org/licenses/ECL-2.0                                     */
/*                                                                           */
/* Unless required by applicable law or agreed to in writing,                */
/* software distributed under the License is distributed on an              */
/* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,              */
/* either express or implied. See the License for the specific               */
/* language governing permissions and limitations under the License.         */
/***************************************************************************/

#define XEN
#define XENH
#define PROT_READ 1 // The value assigned is random 'cause I don't know the real
                    // vlaue.  The constant is defined in sys/mman.h


#include <stdio.h>
#include <stdlib.h>
#define XA_SUCCESS 0
#define XA_FAILURE -1
#define XA_FAILHARD 0
#define XA_FAILSOFT 1

struct xa_cache_entry{
    time_t last_used;
```

```
    char *symbol_name;
    uint32_t virt_address;
    uint32_t mach_address;
    int pid;
    struct xa_cache_entry *next;
    struct xa_cache_entry *prev;
};
typedef struct xa_cache_entry* xa_cache_entry_t;

struct xa_pid_cache_entry{
    time_t last_used;
    int pid;
    uint32_t pgd;
    struct xa_pid_cache_entry *next;
    struct xa_pid_cache_entry *prev;
};
typedef struct xa_pid_cache_entry* xa_pid_cache_entry_t;

/**
 * @brief XenAccess instance.
 *
 * This struct holds all of the relavent information for an instance of
 * XenAccess.  Each time a new domain is accessed, a new instance must
 * be created using the xa_init function.  When you are done with an instance,
 * its resources can be freed using the xa_destroy function.
 */
typedef struct xa_instance{
    uint32_t mode;          /**< file or xen VM data source */
    uint32_t error_mode;    /**< XA_FAILHARD or XA_FAILSOFT */
    char *sysmap;           /**< system map file for domain's running kernel */
    char *image_type;       /**< image type that we are accessing */
    uint32_t page_offset;   /**< page offset for this instance */
    uint32_t page_shift;    /**< page shift for last mapped page */
    uint32_t page_size;     /**< page size for last mapped page */
    uint32_t kpgd;          /**< kernel page global directory */
    uint32_t init_task;     /**< address of task struct for init */
    int os_type;            /**< type of os: XA_OS_LINUX, etc */
    int hvm;                /**< nonzero if HVM memory image */
    int pae;                /**< nonzero if PAE is enabled */
    int pse;                /**< nonzero if PSE is enabled */
    uint32_t cr3;           /**< value in the CR3 register */
    xa_cache_entry_t cache_head;         /**< head of the address cache list */
    xa_cache_entry_t cache_tail;         /**< tail of the address cache list */
    int current_cache_size;              /**< size of the address cache list */
    xa_pid_cache_entry_t pid_cache_head; /**< head of the pid cache list */
```

```
    xa_pid_cache_entry_t pid_cache_tail; /**< tail of the pid cache list */
    int current_pid_cache_size;          /**< size of the pid cache list */
    union{
        struct linux_instance{
            int tasks_offset;    /**< task_struct->tasks */
            int mm_offset;       /**< task_struct->mm */
            int pid_offset;      /**< task_struct->pid */
            int pgd_offset;      /**< mm_struct->pgd */
            int addr_offset;     /**< mm_struct->start_code */
        } linux_instance;
        struct windows_instance{
            uint32_t ntoskrnl;   /**< base phys address for ntoskrnl image */
            int tasks_offset;    /**< EPROCESS->ActiveProcessLinks */
            int pdbase_offset;   /**< EPROCESS->Pcb.DirectoryTableBase */
            int pid_offset;      /**< EPROCESS->UniqueProcessId */
            int peb_offset;      /**< EPROCESS->Peb */
            int iba_offset;      /**< EPROCESS->Peb.ImageBaseAddress */
            int ph_offset;       /**< EPROCESS->Peb.ProcessHeap */
        } windows_instance;
    } os;
    union{
        struct xen{
            int xc_handle;       /**< handle to xenctrl library (libxc) */
            uint32_t domain_id;  /**< domid that we are accessing */
            int xen_version;     /**< version of Xen libxa is running on */
         //   xc_dominfo_t info;   /**< libxc info: domid, ssidref, stats, etc */
            uint32_t size;       /**< total size of domain's memory */
            unsigned long *live_pfn_to_mfn_table;
            unsigned long nr_pfns;
        } xen;
        struct wfile{
            FILE *fhandle;       /**< handle to the memory image file */
            uint32_t size;       /**< total size of file, in bytes */
        } file;
    } m;
} xa_instance_t;

void xa_destroy(xa_instance_t *dummy);
int xa_init_vm_id_cmat(uint32_t dummy1, xa_instance_t *dummy2, int dummy3);
unsigned char* xa_access_pa(xa_instance_t *dummy1, uint32_t dummy2,
                              uint32_t *dummy3, int dummy4);
void munmap(unsigned char* dummy1, uint32_t);
```

Bibliography

1. Access, Xen, "XenAccess Documentation." xen.org/products/xenhyp.html, Last accessed 15-Aug-11.
2. Andera, C., "hobocopy," 2006. http://www.pluralsight-training.net/community/blogs/craig/archive/2006/09/20/38362.aspx.
3. Betz, C., "memparser," 2005.
4. Central, Microsoft Windows Hardware Developer, "Microsoft Portable Executable and Common Object File Format Specification."
5. D. Dodge, B. Mullins, G. Peterson and J. Okolica. "Simulating Windows-Based Cyber Attacks Using Live Virtual Machine Introspection." *Proceedings of the Summer Computer Simulation Conference (SCS10)*. 550–555. 2010.
6. Dolan-Gavitt, B. "Forensic Analysis of the Windows Registry in Memory." *Proceedings of the 2008 Digital Forensic Research Workshop (DFRWS)*. 26–32. 2008.
7. Dolan-Gavitt, B., "Push the Red Button Linking Processes to Users," 2008.
8. Framework, The Volatility, "Volatile memory artifact extraction utility framework." accessed 15-April-2011.
9. Ionescu, A., "Getting Kernel Variables from KdVersionBlock, Part2."
10. Mandiant, "Memoryze." http://www.mandiant.com/software/memoryze.htm, Accessed August 15, 2009.
11. Okolica, J. and G. Peterson. "A Compiled Memory Analysis Tool." *Advances in Digital Forensics VI* edited by Kam-Pui Chow and Sujeet Shenoi, 195–204, Springer, 2008.
12. Okolica, J. and G. Peterson. "Windows Operating Systems Agnostic Memory Analysis." *Proceedings of the 2010 Digital Forensic Research Workshop (DFRWS)*. 48–56. 2010.
13. Okolica, J. and G. Peterson. "Extracting the Windows Clipboard from Physical Memory," *Digital Investigations Journal*, 118–124 (2011).
14. Okolica, J. and G. Peterson. "Windows Driver Memory Analysis: A Reverse Engineering Methodology," *Computers and Security* (2011).
15. Russinovich, M., "Sysinternals Suite." Accessed August 15, 2009.
16. Russinovich, M. and D. Solomon. *Microsoft Windows Internals, 4th Edition*. Microsoft Press, 2005.
17. Schreiber, S. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Addison Wesley, 2001.
18. Schuster, A., "PTfinder," 2006.
19. Schuster, A. "Searching for Processes and Threads in Microsoft Windows Memory Dumps." *Proceedings of the 2006 Digital Forensic Research Workshop (DFRWS)*. 10–16. 2006.

20. Stevens, R. and E. Casey. “Extracting Windows Command Line Details From Physical Memory.” *Proceedings of the 2010 Digital Forensic Research Workshop (DFRWS)*. 57–63. 2010.

21. Suiche, M., “Win32dd.” Accessed August 15, 2009.

22. Support, Microsoft, “Description of .PDB and of the .DBG files.”

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

24. Xen, “Xen Hypervisor - Leading Open Source Hypervisor for Servers.” xen.org/products/xenhyp.html, Last accessed 15-Aug-11.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
30-08-2011	**Technical Report**	

4. TITLE AND SUBTITLE

Extracting Forensic Artifacts from Windows O/S Memory

5a. CONTRACT NUMBER

5b. GRANT NUMBER
11G258

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
James S. Okolica and Gilbert L. Peterson

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
AFIT/ENG
2950 Hobson Way
Wright-Patterson AFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/EN/TR-11-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/RIG
26 Electronic Parkway
Rome, NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Memory analysis is a rapidly growing area in both digital forensics and cyber situational awareness (SA). Memory provides the most accurate snapshot of what is occurring on a computer at a moment in time. By combining it with event and network logs as well as the files present on the filesystem, an analyst can re-create much of what has occurred and is occuring on a computer. The Compiled Memory Analysis Tool (CMAT) takes either a disk image of memory from a Windows operating system or an interface into a virtual machine running a Windows operating system and extracts forensic artifacts including general system information, loaded system modules, the active processes, the files and registry keys accessed by those processes, the network connections established by the processes, the dynamic link libraries loaded by the processes, and the contents of the Windows clipboard. Operators and investigators can either take these artifacts and analyze them directly or use them as input into more complex cyber SA and digital forensics analysis tools.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
REPORT	**ABSTRACT**	**c. THIS PAGE**		381	Gilbert L. Peterson
					19b. TELEPHONE NUMBER *(Include area code)* (937)255-3636x4281

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18