```c
#include <stdio.h>
#include "winsock2.h"
#include <windows.h>
#include <conio.h>
#include <tchar.h>
#include <winioctl.h>
#include <winternl.h>

#define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)

#pragma comment(lib,"ws2_32.lib")
#pragma warning(disable:4996)
#pragma comment(lib,"ntdll.lib")

readHexandCharStream(UNICODE_STRING );
InitializeSniffer(long long int);
void StartSniffing(SOCKET , long long int);
void ProcessPacket(char*, int);
void PrintTcpPacket(char*, int);
void PrintUdpPacket(char *, int);
void PrintIcmpPacket(char*, int);
void PrintIpHeader(char*);
void PrintData(char*, int);
int getInt();
void Display_Graphics(int);
void ScanLogicalDisk_DisplayDisk(TCHAR[]);
void ScanPhysicalDisk_DisplayDisk(HANDLE, UNICODE_STRING);
int ScanMemory_DisplayMemory(char, unsigned int, UNICODE_STRING);
DWORD WINAPI Processing_Thread();
VOID WINAPI SetConsoleColors(WORD);
BOOL SetPrivilege(HANDLE, LPCTSTR, BOOL);

HANDLE eventHnd;
int stopRequested = 0;

FILE *logfile;
int tcp = 0, udp = 0, icmp = 0, others = 0, igmp = 0, total = 0, i, j;
struct sockaddr_in source, dest;
char hex[2];

typedef struct _EVENTLOGHEADER {
    ULONG HeaderSize;
    ULONG Signature;
    ULONG MajorVersion;
    ULONG MinorVersion;
    ULONG StartOffset;
    ULONG EndOffset;
    ULONG CurrentRecordNumber;
    ULONG OldestRecordNumber;
    ULONG MaxSize;
    ULONG Flags;
    ULONG Retention;
```

```
53      ULONG EndHeaderSize;
54  } EVENTLOGHEADER, *PEVENTLOGHEADER;
55
56  typedef struct ip_hdr
57  {
58      unsigned char ip_header_len : 4;
59      unsigned char ip_version : 4;
60      unsigned char ip_tos;
61      unsigned short ip_total_length;
62      unsigned short ip_id;
63
64      unsigned char ip_frag_offset : 5;
65
66      unsigned char ip_more_fragment : 1;
67      unsigned char ip_dont_fragment : 1;
68      unsigned char ip_reserved_zero : 1;
69
70      unsigned char ip_frag_offset1;
71
72      unsigned char ip_ttl;
73      unsigned char ip_protocol;
74      unsigned short ip_checksum;
75      unsigned int ip_srcaddr;
76      unsigned int ip_destaddr;
77  } IPV4_HDR;
78
79  typedef struct udp_hdr
80  {
81      unsigned short source_port;
82      unsigned short dest_port;
83      unsigned short udp_length;
84      unsigned short udp_checksum;
85  } UDP_HDR;
86
87  typedef struct tcp_header
88  {
89      unsigned short source_port;
90      unsigned short dest_port;
91      unsigned int sequence;
92      unsigned int acknowledge;
93
94      unsigned char ns : 1;
95      unsigned char reserved_part1 : 3;
96      unsigned char data_offset : 4;
97
98      unsigned char fin : 1;
99      unsigned char syn : 1;
100     unsigned char rst : 1;
101     unsigned char psh : 1;
102     unsigned char ack : 1;
103     unsigned char urg : 1;
104
```

```
105     unsigned char ecn : 1;
106     unsigned char cwr : 1;
107
108     unsigned short window;
109     unsigned short checksum;
110     unsigned short urgent_pointer;
111 } TCP_HDR;
112
113 typedef struct icmp_hdr
114 {
115     BYTE type;
116     BYTE code;
117     USHORT checksum;
118     USHORT id;
119     USHORT seq;
120 } ICMP_HDR;
121
122 /**struct ipv6_header
123 {
124 unsigned int
125 version : 4,
126 traffic_class : 8,
127 flow_label : 20;
128 unsigned short int length;
129 unsigned char  next_header;
130 unsigned char  hop_limit;
131 struct in6_addr src;
132 struct in6_addr dst;
133
134 };**/
135
136 typedef struct _SYSTEM_PROCESS_INFO
137 {
138     ULONG                   NextEntryOffset;
139     ULONG                   NumberOfThreads;
140     LARGE_INTEGER           Reserved[3];
141     LARGE_INTEGER           CreateTime;
142     LARGE_INTEGER           UserTime;
143     LARGE_INTEGER           KernelTime;
144     UNICODE_STRING          ImageName;
145     ULONG                   BasePriority;
146     HANDLE                  ProcessId;
147     HANDLE                  InheritedFromProcessId;
148 }SYSTEM_PROCESS_INFO, *PSYSTEM_PROCESS_INFO;
149
150 IPV4_HDR *iphdr;
151 TCP_HDR *tcpheader;
152 UDP_HDR *udpheader;
153 ICMP_HDR *icmpheader;
154
155 BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)
156 {
```

```
157     TOKEN_PRIVILEGES tp;
158     LUID luid;
159
160     if (!LookupPrivilegeValue(NULL, lpszPrivilege, &luid))
161     {
162         //printf("LookupPrivilegeValue error: %u\n", GetLastError());
163         return FALSE;
164     }
165
166     tp.PrivilegeCount = 1;
167     tp.Privileges[0].Luid = luid;
168     if (bEnablePrivilege)
169         tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
170     else
171         tp.Privileges[0].Attributes = 0;
172
173
174     if (!AdjustTokenPrivileges(
175         hToken,
176         FALSE,
177         &tp,
178         sizeof(TOKEN_PRIVILEGES),
179         (PTOKEN_PRIVILEGES)NULL,
180         (PDWORD)NULL))
181     {
182         printf("AdjustTokenPrivileges error: %u\n", GetLastError());
183         return FALSE;
184     }
185
186     if (GetLastError() == ERROR_NOT_ALL_ASSIGNED)
187
188     {
189         printf("The process does not have the required privilege. \n");
190         return FALSE;
191     }
192
193     return TRUE;
194 }
195
196 VOID WINAPI SetConsoleColors(WORD attribs) {
197
198     HANDLE hOutput = GetStdHandle(STD_OUTPUT_HANDLE);
199
200     CONSOLE_SCREEN_BUFFER_INFOEX cbi;
201     cbi.cbSize = sizeof(CONSOLE_SCREEN_BUFFER_INFOEX);
202     GetConsoleScreenBufferInfoEx(hOutput, &cbi);
203     cbi.wAttributes = attribs;
204     SetConsoleScreenBufferInfoEx(hOutput, &cbi);
205
206 }
207
208 DWORD WINAPI Processing_Thread() {
```

```
209
210     printf("\n\nOperation is being performed. Please wait.....................");
211
212     do {
213
214         // WaitForSingleObject(eventHnd, INFINITE);
215
216         if (stopRequested)
217             return;
218
219         printf("|");
220         printf("\b");
221         Sleep(100);
222         fflush(stdout);
223         printf("/");
224         printf("\b");
225         Sleep(100);
226         fflush(stdout);
227         printf("-");
228         printf("\b");
229         Sleep(100);
230         fflush(stdout);
231         printf("\\");
232         printf("\b");
233         Sleep(100);
234         fflush(stdout);
235
236     } while (1);
237
238
239 }
240
241 int ScanMemory_DisplayMemory(char option, unsigned int pid, UNICODE_STRING pName)
242 {
243
244     MEMORY_BASIC_INFORMATION meminfo;
245     unsigned char  *addr = 0, *addr1 = 0;
246
247
248     FILE *f;
249
250     if (option == '0') {
251         f = fopen("ProcDump.txt", "a");
252     }
253     else if(option == '1') {
254         f = fopen("MemDump.txt", "a");
255     }
256     else {
257         f = fopen("ErrDump.txt", "a");
258     }
```

```
259
260
261     HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
262     HANDLE lProc = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, pid);
263     eventHnd = CreateEvent(NULL, 0, 0, NULL);
264     stopRequested = 0;
265     HANDLE thread = CreateThread(NULL, 0, Processing_Thread, NULL, 0, NULL);
266     DWORD error = GetLastError();
267
268     HANDLE fProc;
269
270     int t_run = -1;
271
272     if (SetEvent(eventHnd)) {
273
274         t_run = 1;
275
276     }
277
278
279     if (hProc || lProc)
280     {
281
282
283         //(!hProc && lProc) ? printf("lproc") : printf("hProc"); //Testing
              Condition
284         fProc = (!hProc && lProc) ? lProc : hProc;
285
286         while (1) {
287             if ((VirtualQueryEx(fProc, addr1, &meminfo, sizeof(meminfo))) == 0)
288             {
289                 break;
290             }
291
292
293             if (meminfo.State == MEM_COMMIT)
294             {
295                 static unsigned char display_buffer[1024 * 128];
296                 SIZE_T bytes_left;
297                 SIZE_T total_read;
298                 SIZE_T bytes_to_read;
299                 SIZE_T bytes_read;
300
301                     addr = (unsigned char*)meminfo.BaseAddress;
302
303                     //printf("Process Name : %ws\r\n", pName.Buffer);
304                     fprintf(f, "Process Name : %ws\r\n", pName.Buffer);
305
306                     //printf("Base Address : 0x%08x\r\n", addr);
307                     fprintf(f, "Base Address : 0x%08x\r\n", addr);
308
309                     bytes_left = meminfo.RegionSize;
```

```
310
311                     //printf("Region Size : %d\r\n", bytes_left);
312                     fprintf(f, "Region Size : %d\r\n", bytes_left);
313
314                     total_read = 0;
315
316                     //printf("Contents : \r\n");
317                     fprintf(f, "Contents : \r\n");
318
319                     while (bytes_left)
320                     {
321                         bytes_to_read = (bytes_left > sizeof(display_buffer)) ? sizeof(display_buffer) : bytes_left;
322                         ReadProcessMemory(hProc, addr + total_read, display_buffer, bytes_to_read, &bytes_read);
323                         if (bytes_read != bytes_to_read) break;
324
325                         int j = 0;
326
327                         fprintf(f, "-------------------------------------------------HEX CODE-------------------------------------------------");
328                         fprintf(f, "\n");
329
330                         for (j = 0; j < bytes_to_read; j++)
331                         {
332                             fprintf(f, "%02x ", display_buffer[j]);
333                         }
334
335                         fprintf(f, "\r\n\n");
336
337                         fprintf(f, "-------------------------------------------------CHARACTER STREAM-------------------------------------------------");
338                         fprintf(f, "\n");
339
340                         for (j = 0; j < bytes_to_read; j++)
341                         {
342                             if ((display_buffer[j] > 31) && (display_buffer[j] < 127)) {
343                                 //printf("%c ", display_buffer[j]);
344                                 fprintf(f, "%c", display_buffer[j]);
345                             }
346                             else {
347                                 //printf(".");
348                                 fprintf(f, ".");
349                             }
350                         }
351                         //printf("\r\n");
352                         fprintf(f, "\r\n");
353
354                         bytes_left -= bytes_read;
```

```
355                         total_read += bytes_read;
356                     }
357                 }
358             addr1 = (unsigned char*)meminfo.BaseAddress + meminfo.RegionSize;
359         }
360
361         CloseHandle(fProc);
362
363     }
364
365     else {
366
367         printf("\nFailed to open process - error - %d\r\n", error);
368
369     }
370
371     if (t_run == 1) {
372
373         stopRequested = 1;
374         SetEvent(eventHnd);
375         WaitForSingleObject(thread, 5000);
376         CloseHandle(thread);
377         CloseHandle(eventHnd);
378         printf(" \b\b  ");
379         fflush(stdout);
380         printf("\n");
381         t_run = -1;
382     }
383
384     fclose(f);
385     return 0;
386 }
387
388 void ScanPhysicalDisk_DisplayDisk(HANDLE disk, UNICODE_STRING diskx) {
389
390     eventHnd = CreateEvent(NULL, 0, 0, NULL);
391     stopRequested = 0;
392     HANDLE thread = CreateThread(NULL, 0, Processing_Thread, NULL, 0, NULL);
393
394     int t_run = -1;
395
396     if (SetEvent(eventHnd)) {
397
398         t_run = 1;
399
400     }
401
402     FILE *f;
403
404     f = fopen("DiskDump.txt", "a");
405
406     DWORD br = 0;
```

```
407     DISK_GEOMETRY dg;
408
409     DeviceIoControl(disk, IOCTL_DISK_GET_DRIVE_GEOMETRY, 0, 0, &dg, sizeof(dg),
          &br, 0);
410
411     int bufsize = dg.BytesPerSector;
412     unsigned char *buf = malloc(bufsize);
413
414     fprintf(f, "Disk Name : %ws\r\n", diskx.Buffer);
415
416
417     while (ReadFile(disk, buf, bufsize, &br, NULL))
418     {
419         if (!br)
420             break;
421
422         fprintf(f, "\n\n\r");
423         fprintf(f, "----------------------------------------------------HEX
            CODE----------------------------------------------------");
424         fprintf(f, "\n");
425
426         for (int bsize = 0; bsize < bufsize; bsize++) {
427             fprintf(f, "%02x ", buf[bsize]);
428         }
429
430         fprintf(f, "\n\n\r");
431         fprintf(f, "----------------------------------------------------CHARACTER
            STREAM----------------------------------------------------");
432         fprintf(f, "\n");
433
434         for (int bsize = 0; bsize < bufsize; bsize++) {
435             if (buf[bsize] > 31 && buf[bsize] < 127) {
436                 fprintf(f, "%c", buf[bsize]);
437             }
438             else {
439                 fprintf(f, ".", buf[bsize]);
440             }
441         }
442
443
444     }
445
446     fclose(f);
447     printf("\n");
448
449     if (t_run == 1) {
450
451         stopRequested = 1;
452         SetEvent(eventHnd);
453         WaitForSingleObject(thread, 5000);
454         CloseHandle(thread);
455         CloseHandle(eventHnd);
```

```
456         printf(" \b\b  ");
457         fflush(stdout);
458         printf("\n");
459         t_run = -1;
460     }
461
462 }
463
464
465 void ScanLogicalDisk_DisplayDisk(TCHAR driveString[]) {
466
467     eventHnd = CreateEvent(NULL, 0, 0, NULL);
468     stopRequested = 0;
469     HANDLE thread = CreateThread(NULL, 0, Processing_Thread, NULL, 0, NULL);
470
471     int t_run = -1;
472
473     if (SetEvent(eventHnd)) {
474
475         t_run = 1;
476
477     }
478
479     FILE *f;
480
481     f = fopen("DriveDump.txt", "a");
482     //printf("%ws", driveString);
483
484     HANDLE handle = CreateFile(driveString, GENERIC_READ, FILE_SHARE_READ |
      FILE_SHARE_WRITE, NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING, NULL);
485
486     DWORD br = 0;
487     DISK_GEOMETRY dg;
488
489     DeviceIoControl(handle, IOCTL_DISK_GET_DRIVE_GEOMETRY, 0, 0, &dg, sizeof
      (dg), &br, 0);
490
491     int bufsize = dg.BytesPerSector;
492     unsigned char *buf = malloc(bufsize);
493
494     fprintf(f, "Drive Name : %ws\r\n", driveString);
495
496
497     while (ReadFile(handle, buf, bufsize, &br, NULL))
498     {
499         if (!br)
500             break;
501
502         fprintf(f, "\n\n\r");
503         fprintf(f, "-----------------------------------------------HEX
          CODE-----------------------------------------------");
504         fprintf(f, "\n");
```

```
505
506         for (int bsize = 0; bsize < bufsize; bsize++) {
507             fprintf(f, "%02x ", buf[bsize]);
508         }
509
510         fprintf(f, "\n\n\r");
511         fprintf(f, "-------------------------------------------------CHARACTER
              STREAM-------------------------------------------------");
512         fprintf(f, "\n");
513
514         for (int bsize = 0; bsize < bufsize; bsize++) {
515             if (buf[bsize] > 31 && buf[bsize] < 127) {
516                 fprintf(f, "%c", buf[bsize]);
517             }
518             else {
519                 fprintf(f, ".", buf[bsize]);
520             }
521         }
522
523     }
524
525     fprintf(f, "\n\r");
526
527     printf("\n");
528     fclose(f);
529     CloseHandle(handle);
530
531     if (t_run == 1) {
532
533         stopRequested = 1;
534         SetEvent(eventHnd);
535         WaitForSingleObject(thread, 5000);
536         CloseHandle(thread);
537         CloseHandle(eventHnd);
538         printf(" \b\b  ");
539         fflush(stdout);
540         printf("\n");
541
542     }
543 }
544
545 void Display_Graphics(int priv) {
546
547     system("cls");
548
549     printf("                  __________           _____ _____   _____
          _ _    _ _\n");
550     printf("                 |_   _| ___ \\   ___    |  ___/  __ \\ |_   _|
          | | |  (_) |\n");
551     printf("                   | | | |_/ /  ( _ )   | |__ | /  \\/   | | ___
        ___ | | | ___| |_\n");
552     printf("                   | | |    /   / _ \\/\\ |  __|| |       | |/ _ \
```

```
          \ / _ \\| | |/ / | __|\n");
553       printf("                  _| |_| |\\ \\  | (_>  < | |___| \\__/\\   | | (_)
            | (_) | |   <| | |_\n");
554       printf("                  \\___/\\_| \\_|  \\___/\\/ \\____/ \\____/   \\_/\
            \___/ \\___/|_|_|\\_\\_|\\__|\n");
555       printf("\n");
556       printf("                                DEVELOPED BY : SAPTARSHI LAHA (RIK)
            \n");
557       if (priv == 1) {
558           printf("                                    DEBUG PRIVILEGE STATUS :
              GRANTED\n");
559       }
560       else if (priv == -1) {
561           printf("                                    DEBUG PRIVILEGE STATUS :
              DENIED\n");
562       }
563       printf("\n\n");
564       printf("0. Single Process Memory Dump\n");
565       printf("1. Full System Memory Dump (Consumes a LOT of Space!)\n");
566       printf("2. Logical Drive Analysis (Consumes a LOT of Space!)\n");
567       printf("3. Physical Drive Analysis (Consumes a LOT of Space!)\n");
568       printf("4. Application Port Scanning\n");
569       printf("5. Network Analysis (Packet Sniffing)\n");
570       printf("6. Handle Analysis - File\n");//TO BE CREATED!!!!!!
571       printf("7. Handle Analysis - Process\n");//TO BE CREATED!!!!!
572       printf("8. Windows Event Log Analysis\n"); //TO BE CREATED!!!!
573       printf("9. Log Reader (Dump Files Generated By Toolkit)\n");// Editing &
            Beautification Left
574       printf("A. Windows File Encryption\n");// TO BE CREATED!!!!!
575       printf("B. Forensic Wipe - Logical Drive (Launch From Another Logical Drive)
            \n");//TO BE CREATED!!!
576       printf("C. Forensic Wipe - Physical Drive (Launch From Another Physical
            Drive)\n");//TO BE CREATED!!!
577       printf("Q. Exit\n\n");
578   }
579
580   int getInt()
581   {
582       int n = 0;
583       char buffer[128];
584       fgets(buffer, sizeof(buffer), stdin);
585       n = atoi(buffer);
586       return (n > 0) ? n : -1;
587   }
588
589   void PrintData(char* data, int Size)
590   {
591       char a, line[17], c;
592       int j;
593
594       for (i = 0; i < Size; i++)
595       {
```

```
596             c = data[i];
597
598             fprintf(logfile, " %.2x", (unsigned char)c);
599
600             a = (c >= 32 && c <= 128) ? (unsigned char)c : '.';
601
602             line[i % 16] = a;
603
604             if ((i != 0 && (i + 1) % 16 == 0) || i == Size - 1)
605             {
606                 line[i % 16 + 1] = '\0';
607
608                 fprintf(logfile, "          ");
609
610                 for (j = strlen(line); j < 16; j++)
611                 {
612                     fprintf(logfile, "   ");
613                 }
614
615                 fprintf(logfile, "%s \n", line);
616             }
617         }
618
619         fprintf(logfile, "\n");
620     }
621
622     void PrintIpHeader(char* Buffer)
623     {
624         unsigned short iphdrlen;
625
626         iphdr = (IPV4_HDR *)Buffer;
627         iphdrlen = iphdr->ip_header_len * 4;
628
629         memset(&source, 0, sizeof(source));
630         source.sin_addr.s_addr = iphdr->ip_srcaddr;
631
632         memset(&dest, 0, sizeof(dest));
633         dest.sin_addr.s_addr = iphdr->ip_destaddr;
634
635         fprintf(logfile, "\n");
636         fprintf(logfile, "IP Header\n");
637         fprintf(logfile, " |-IP Version : %d\n", (unsigned int)iphdr->ip_version);
638         fprintf(logfile, " |-IP Header Length : %d DWORDS or %d Bytes\n", (unsigned
              int)iphdr->ip_header_len, ((unsigned int)(iphdr->ip_header_len)) * 4);
639         fprintf(logfile, " |-Type Of Service : %d\n", (unsigned int)iphdr->ip_tos);
640         fprintf(logfile, " |-IP Total Length : %d Bytes(Size of Packet)\n", ntohs
              (iphdr->ip_total_length));
641         fprintf(logfile, " |-Identification : %d\n", ntohs(iphdr->ip_id));
642         fprintf(logfile, " |-Reserved ZERO Field : %d\n", (unsigned int)iphdr-
              >ip_reserved_zero);
643         fprintf(logfile, " |-Dont Fragment Field : %d\n", (unsigned int)iphdr-
              >ip_dont_fragment);
```

```
644     fprintf(logfile, " |-More Fragment Field : %d\n", (unsigned int)iphdr->ip_more_fragment);
645     fprintf(logfile, " |-TTL : %d\n", (unsigned int)iphdr->ip_ttl);
646     fprintf(logfile, " |-Protocol : %d\n", (unsigned int)iphdr->ip_protocol);
647     fprintf(logfile, " |-Checksum : %d\n", ntohs(iphdr->ip_checksum));
648     fprintf(logfile, " |-Source IP : %s\n", inet_ntoa(source.sin_addr));
649     fprintf(logfile, " |-Destination IP : %s\n", inet_ntoa(dest.sin_addr));
650 }
651
652 void PrintIcmpPacket(char* Buffer, int Size)
653 {
654     unsigned short iphdrlen;
655
656     iphdr = (IPV4_HDR *)Buffer;
657     iphdrlen = iphdr->ip_header_len * 4;
658
659     icmpheader = (ICMP_HDR*)(Buffer + iphdrlen);
660
661     fprintf(logfile, "\n\n**********************ICMP Packet*************************\n");
662     PrintIpHeader(Buffer);
663
664     fprintf(logfile, "\n");
665
666     fprintf(logfile, "ICMP Header\n");
667     fprintf(logfile, " |-Type : %d", (unsigned int)(icmpheader->type));
668
669     if ((unsigned int)(icmpheader->type) == 11)
670     {
671         fprintf(logfile, " (TTL Expired)\n");
672     }
673     else if ((unsigned int)(icmpheader->type) == 0)
674     {
675         fprintf(logfile, " (ICMP Echo Reply)\n");
676     }
677
678     fprintf(logfile, " |-Code : %d\n", (unsigned int)(icmpheader->code));
679     fprintf(logfile, " |-Checksum : %d\n", ntohs(icmpheader->checksum));
680     fprintf(logfile, " |-ID : %d\n", ntohs(icmpheader->id));
681     fprintf(logfile, " |-Sequence : %d\n", ntohs(icmpheader->seq));
682     fprintf(logfile, "\n");
683
684     fprintf(logfile, "IP Header\n");
685     PrintData(Buffer, iphdrlen);
686
687     fprintf(logfile, "UDP Header\n");
688     PrintData(Buffer + iphdrlen, sizeof(ICMP_HDR));
689
690     fprintf(logfile, "Data Payload\n");
691     PrintData(Buffer + iphdrlen + sizeof(ICMP_HDR), (Size - sizeof(ICMP_HDR) - iphdr->ip_header_len * 4));
692
```

```
693     fprintf(logfile, "\n----------------------------------------End Of
          Packet----------------------------------------");
694 }
695
696 void PrintUdpPacket(char *Buffer, int Size)
697 {
698     unsigned short iphdrlen;
699
700     iphdr = (IPV4_HDR *)Buffer;
701     iphdrlen = iphdr->ip_header_len * 4;
702
703     udpheader = (UDP_HDR *)(Buffer + iphdrlen);
704
705     fprintf(logfile, "\n\n**********************UDP
          Packet*************************\n");
706
707     PrintIpHeader(Buffer);
708
709     fprintf(logfile, "\nUDP Header\n");
710     fprintf(logfile, " |-Source Port : %d\n", ntohs(udpheader->source_port));
711     fprintf(logfile, " |-Destination Port : %d\n", ntohs(udpheader->dest_port));
712     fprintf(logfile, " |-UDP Length : %d\n", ntohs(udpheader->udp_length));
713     fprintf(logfile, " |-UDP Checksum : %d\n", ntohs(udpheader->udp_checksum));
714
715     fprintf(logfile, "\n");
716     fprintf(logfile, "IP Header\n");
717
718     PrintData(Buffer, iphdrlen);
719
720     fprintf(logfile, "UDP Header\n");
721
722     PrintData(Buffer + iphdrlen, sizeof(UDP_HDR));
723
724     fprintf(logfile, "Data Payload\n");
725
726     PrintData(Buffer + iphdrlen + sizeof(UDP_HDR), (Size - sizeof(UDP_HDR) -
          iphdr->ip_header_len * 4));
727
728     fprintf(logfile, "\n----------------------------------------End Of
          Packet----------------------------------------");
729 }
730
731 void PrintTcpPacket(char* Buffer, int Size)
732 {
733     unsigned short iphdrlen;
734
735     iphdr = (IPV4_HDR *)Buffer;
736     iphdrlen = iphdr->ip_header_len * 4;
737
738     tcpheader = (TCP_HDR*)(Buffer + iphdrlen);
739
740     fprintf(logfile, "\n\n**********************TCP
```

```
          Packet*************************\n");
741
742       PrintIpHeader(Buffer);
743
744       fprintf(logfile, "\n");
745       fprintf(logfile, "TCP Header\n");
746       fprintf(logfile, " |-Source Port : %u\n", ntohs(tcpheader->source_port));
747       fprintf(logfile, " |-Destination Port : %u\n", ntohs(tcpheader->dest_port));
748       fprintf(logfile, " |-Sequence Number : %u\n", ntohl(tcpheader->sequence));
749       fprintf(logfile, " |-Acknowledge Number : %u\n", ntohl(tcpheader-
            >acknowledge));
750       fprintf(logfile, " |-Header Length : %d DWORDS or %d BYTES\n", (unsigned
            int)tcpheader->data_offset, (unsigned int)tcpheader->data_offset * 4);
751       fprintf(logfile, " |-CWR Flag : %d\n", (unsigned int)tcpheader->cwr);
752       fprintf(logfile, " |-ECN Flag : %d\n", (unsigned int)tcpheader->ecn);
753       fprintf(logfile, " |-Urgent Flag : %d\n", (unsigned int)tcpheader->urg);
754       fprintf(logfile, " |-Acknowledgement Flag : %d\n", (unsigned int)tcpheader-
            >ack);
755       fprintf(logfile, " |-Push Flag : %d\n", (unsigned int)tcpheader->psh);
756       fprintf(logfile, " |-Reset Flag : %d\n", (unsigned int)tcpheader->rst);
757       fprintf(logfile, " |-Synchronise Flag : %d\n", (unsigned int)tcpheader-
            >syn);
758       fprintf(logfile, " |-Finish Flag : %d\n", (unsigned int)tcpheader->fin);
759       fprintf(logfile, " |-Window : %d\n", ntohs(tcpheader->window));
760       fprintf(logfile, " |-Checksum : %d\n", ntohs(tcpheader->checksum));
761       fprintf(logfile, " |-Urgent Pointer : %d\n", tcpheader->urgent_pointer);
762       fprintf(logfile, "\n");
763       fprintf(logfile, " DATA Dump ");
764       fprintf(logfile, "\n");
765
766       fprintf(logfile, "IP Header\n");
767       PrintData(Buffer, iphdrlen);
768
769       fprintf(logfile, "TCP Header\n");
770       PrintData(Buffer + iphdrlen, tcpheader->data_offset * 4);
771
772       fprintf(logfile, "Data Payload\n");
773       PrintData(Buffer + iphdrlen + tcpheader->data_offset * 4, (Size - tcpheader-
            >data_offset * 4 - iphdr->ip_header_len * 4));
774
775       fprintf(logfile, "\n-----------------------------------------End Of
            Packet-----------------------------------------");
776   }
777
778   void ProcessPacket(char* Buffer, int Size)
779   {
780       iphdr = (IPV4_HDR *)Buffer;
781       ++total;
782
783       switch (iphdr->ip_protocol)
784       {
785       case 1:
```

```
786         ++icmp;
787         PrintIcmpPacket(Buffer, Size);
788         break;
789
790     case 2:
791         ++igmp;
792         break;
793
794     case 6:
795         ++tcp;
796         PrintTcpPacket(Buffer, Size);
797         break;
798
799     case 17:
800         ++udp;
801         PrintUdpPacket(Buffer, Size);
802         break;
803
804     default:
805         ++others;
806         break;
807     }
808     printf("TCP : %d UDP : %d ICMP : %d IGMP : %d Others : %d Total : %d\r",
      tcp, udp, icmp, igmp, others, total);
809 }
810
811 void StartSniffing(SOCKET sniffer, long long int number)
812 {
813     int psniff = 0;
814     char *Buffer = (char *)malloc(65536);
815     int mangobyte;
816
817     do
818     {
819         mangobyte = recvfrom(sniffer, Buffer, 65536, 0, 0, 0);
820
821         if (mangobyte > 0)
822         {
823             ProcessPacket(Buffer, mangobyte);
824             psniff++;
825         }
826         else
827         {
828             printf("Receiving failed.\n");
829         }
830
831         if (psniff >= number)
832             break;
833
834     } while (mangobyte > 0);
835
836     free(Buffer);
```

```
837 }
838
839 readHexandCharStream(UNICODE_STRING fileName)
840 {
841
842     char toBeEnteredByTheUser;
843     unsigned long long int HexPointers[10000];
844     unsigned long long int CharacterPointer[10000];
845     char HexString[] =  "-------------------------------------------------HEX
        CODE-------------------------------------------------" ;
846     char CharacterString[] =  "------------------------------------------------
        CHARACTER STREAM-------------------------------------------------" ;
847
848     system("mode 650, 650");
849
850     FILE *f;
851
852     f = fopen("ProcDump.txt", "r");
853
854     if (f) {
855
856         unsigned long long int pos = 0, iter = 0;
857         int matchnumberH = 0, matchnumberC = 0;
858         char c;
859         while ((c = getc(f)) != EOF) {
860
861             pos++;
862
863             if (c == HexString[matchnumberH]) {
864                 if (matchnumberH <= 106) {
865                     matchnumberH++;
866                     if (matchnumberH == 106) {
867                         HexPointers[iter] = pos - 106;
868                         matchnumberH = 0;
869                     }
870                 }
871             }
872             else {
873                 matchnumberH = 0;
874             }
875
876
877
878             if (c == CharacterString[matchnumberC]) {
879                 if (matchnumberC <= 114) {
880                     matchnumberC++;
881
882                     if (matchnumberC == 114) {
883                         CharacterPointer[iter] = pos - 115;
884                         iter++;
885                         matchnumberC = 0;
886                     }
```

```
                }
            }
            else {
                matchnumberC = 0;
            }

            //printf("H : %d\n", matchnumberH);
            //printf("C : %d\n", matchnumberC);

        }

        /**for (int counter = 0; counter < 10000; counter++) {
            printf("HP: %d\t", HexPointers[counter]);
        }
        for (int counter = 0; counter < 10000; counter++) {
            printf("CP: %d\t", CharacterPointer[counter]);
        }**/

    }
    else {

        system("mode 120, 35");
        return -1;
    }

    char buffer[50000];
    for (int counter = 0; counter < 50000; counter++) {
        buffer[counter] = NULL;
    }
    fseek(f, HexPointers[0], SEEK_SET);
    unsigned long long int bytes_to_read = CharacterPointer[0] - HexPointers[0];
    fread(buffer, sizeof(char), bytes_to_read, f);
    printf("%s", buffer);
    fseek(f, CharacterPointer[0], SEEK_SET);

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

    bytes_to_read = (HexPointers[1] - CharacterPointer[0]) + 3;
    for (int counter = 0; counter < 50000; counter++) {
        buffer[counter] = NULL;
    }
    fread(buffer, sizeof(char), bytes_to_read, f);
    printf("%s", buffer);

    system("pause");
    system("mode 120, 35");
    return 0;


}
```

```
939  int InitializeSniffer(long long int number)
940  {
941
942      SOCKET sniffer;
943      struct in_addr addr;
944      int in;
945
946      char hostname[100];
947      struct hostent *local;
948      WSADATA wsa;
949
950      logfile = fopen("SniffDump.txt", "a");
951
952      if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
953      {
954          printf("Winsock Startup Failed.\n");
955          return 1;
956      }
957
958      sniffer = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
959      if (sniffer == INVALID_SOCKET)
960      {
961          printf("Failed to Create Raw Socket.\n");
962          return 1;
963      }
964
965      if (gethostname(hostname, sizeof(hostname)) == SOCKET_ERROR)
966      {
967          printf("Error : %d", WSAGetLastError());
968          return 1;
969      }
970      printf("Host name : %s \n", hostname);
971
972      local = gethostbyname(hostname);
973      printf("\nAvailable Network Interfaces : \n");
974      if (local == NULL)
975      {
976          printf("Error : %d.\n", WSAGetLastError());
977          return 1;
978      }
979
980      for (i = 0; local->h_addr_list[i] != 0; ++i)
981      {
982          memcpy(&addr, local->h_addr_list[i], sizeof(struct in_addr));
983          printf("Interface Number : %d Address : %s\n", (i + 1), inet_ntoa
              (addr));
984      }
985
986      printf("Enter the interface number you would like to sniff : ");
987
988      in = getInt();
989      in--;
```

```
990
991     memset(&dest, 0, sizeof(dest));
992     memcpy(&dest.sin_addr.s_addr, local->h_addr_list[in], sizeof
          (dest.sin_addr.s_addr));
993     dest.sin_family = AF_INET;
994     dest.sin_port = 0;
995
996     if (bind(sniffer, (struct sockaddr *)&dest, sizeof(dest)) == SOCKET_ERROR)
997     {
998         printf("Binding (%s) failed.\n", inet_ntoa(addr));
999         return 1;
1000    }
1001
1002    j = 1;
1003
1004    if (WSAIoctl(sniffer, SIO_RCVALL, &j, sizeof(j), 0, 0, (LPDWORD)&in, 0, 0)
        == SOCKET_ERROR)
1005    {
1006        printf("IOCTL Windows Sniffing failed.\n");
1007        return 1;
1008    }
1009
1010    printf("\nStarted Sniffing\n");
1011    printf("Packet Capture Statistics...\n");
1012    StartSniffing(sniffer, number);
1013    closesocket(sniffer);
1014    WSACleanup();
1015
1016    return 0;
1017
1018 }
1019
1020 int main() {
1021
1022
1023    system("mode 120, 35");
1024
1025    int privilege = 1;
1026
1027    SetConsoleColors(BACKGROUND_GREEN | BACKGROUND_BLUE | FOREGROUND_RED |
        FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_INTENSITY);
1028    SetConsoleTitle(_T("Incident Response & Evidence Collection Toolkit"));
1029
1030    HANDLE hProc = GetCurrentProcess();
1031
1032    HANDLE hToken = NULL;
1033    if (!OpenProcessToken(hProc, TOKEN_ADJUST_PRIVILEGES, &hToken)) {
1034        //printf("Failed to open access token\n\n");
1035        privilege = -1;
1036    }
1037
1038    if (!SetPrivilege(hToken, SE_DEBUG_NAME, TRUE)) {
```

```
1039          //printf("Failed to set debug privilege\n\n");
1040          privilege = -1;
1041      }
1042
1043      NTSTATUS status;
1044      PVOID buffer;
1045      PSYSTEM_PROCESS_INFO spi;
1046
1047
1048      buffer = VirtualAlloc(NULL, 1024 * 1024, MEM_COMMIT | MEM_RESERVE,
            PAGE_READWRITE);
1049
1050      if (!buffer)
1051      {
1052          printf("Error: Unable to allocate memory for process list (%d)\n\n",
              GetLastError());
1053          return -1;
1054      }
1055
1056      char option = -1;
1057      int pid = -1, i, x;
1058      int con = -1;
1059      uintptr_t h[1024 * 30];
1060      UNICODE_STRING pName[1024 * 30];
1061
1062      Display_Graphics(privilege);
1063
1064
1065
1066      while (1) {
1067
1068          option = -1;
1069          spi = (PSYSTEM_PROCESS_INFO)buffer;
1070
1071          if (!NT_SUCCESS(status = NtQuerySystemInformation
              (SystemProcessInformation, spi, 1024 * 1024, NULL)))
1072          {
1073              printf("Error: Unable to query process list (%#x)\n\n", status);
1074
1075              VirtualFree(buffer, 0, MEM_RELEASE);
1076              return -1;
1077          }
1078
1079          if (con != 1) { printf("IR&ECToolkit@Root>"); }
1080
1081          con = -1;
1082          option = getch();
1083
1084          i = 0;
1085
1086          if (option == 13) {
1087              printf("\nIR&ECToolkit@Root>");
```

```
1088            con = 1;
1089        }
1090        else if (option == 0) {
1091            con = 1;
1092            //NULL TERMINATING STRING
1093        }
1094        else if (option == '0') {
1095            printf("\n");
1096            while (spi->NextEntryOffset)
1097            {
1098                h[i] = spi->ProcessId;
1099                pName[i].Buffer = spi->ImageName.Buffer;
1100                printf("Process name: %ws | Process ID: %d\n", spi->ImageName.Buffer, spi->ProcessId);
1101                spi = (PSYSTEM_PROCESS_INFO)((LPBYTE)spi + spi->NextEntryOffset);
1102                i++;
1103            }
1104
1105            x = i;
1106
1107            printf("\nEnter Process ID :");
1108
1109
1110            pid = getInt();
1111
1112            printf("\n");
1113
1114            UNICODE_STRING pNameToBePassed;
1115            pNameToBePassed.Buffer = NULL;
1116
1117            for (i = 0; i < x; i++) {
1118                if (h[i] == pid) {
1119                    pNameToBePassed.Buffer = pName[i].Buffer;
1120                }
1121            }
1122
1123            if (pNameToBePassed.Buffer != NULL) {
1124                printf("%ws\n", pNameToBePassed.Buffer);
1125                ScanMemory_DisplayMemory(option, pid, pNameToBePassed);
1126                printf("\n\nOperation Completed.\n");
1127                system("pause");
1128                Display_Graphics(privilege);
1129            }
1130            else {
1131                printf("\nInvalid PID. Please Try Again.\n");
1132                system("pause");
1133                Display_Graphics(privilege);
1134            }
1135
1136        }
1137
```

```
1138          else if (option == '1') {
1139
1140              i = 0;
1141              printf("\n");
1142
1143              while (spi->NextEntryOffset)
1144              {
1145                  h[i] = spi->ProcessId;
1146                  pName[i].Buffer = spi->ImageName.Buffer;
1147                  printf("Process name: %ws | Process ID: %d\n", spi-
                      >ImageName.Buffer, spi->ProcessId);
1148                  spi = (PSYSTEM_PROCESS_INFO)((LPBYTE)spi + spi-
                      >NextEntryOffset);
1149                  i++;
1150              }
1151
1152              x = i;
1153
1154              UNICODE_STRING pNameToBePassed;
1155              pNameToBePassed.Buffer = NULL;
1156
1157              for (i = 0; i < x; i++) {
1158
1159                  pNameToBePassed.Buffer = pName[i].Buffer;
1160                  printf("\n%d.Process Name : %ws\n", (i + 1),
                      pNameToBePassed.Buffer);
1161                  pid = h[i];
1162                  ScanMemory_DisplayMemory(option, pid, pNameToBePassed);
1163                  printf("Operation Completed Status :  \t %d out of %d\n\n", (i +
                         1), (x + 1));
1164              }
1165
1166              printf("\n\nOperation Completed.\n");
1167              system("pause");
1168              Display_Graphics(privilege);
1169
1170          }
1171
1172          else if (option == '2') {
1173
1174              int loopdrives;
1175              DWORD drives = GetLogicalDrives();
1176              int drivenum[26];
1177              int drivecount = 0;
1178
1179              for (loopdrives = 0; loopdrives < 26; loopdrives++) {
1180
1181                  drivenum[loopdrives] = 0;
1182
1183              }
1184
1185
```

```
1186
1187            printf("\n");
1188            printf("Logical Volumes : \n");
1189            char Drive1[] = { ("A:\\") };
1190            TCHAR Drive2[] = L"\\\\.\\A:";
1191
1192            for (loopdrives = 0; loopdrives < 26; loopdrives++)
1193            {
1194                if (drives & (1 << loopdrives))
1195                {
1196                    drivecount = drivecount + 1;
1197                    Drive1[0] = ('A') + loopdrives;
1198                    printf("%d. %s\n", drivecount, Drive1);
1199                    drivenum[loopdrives] = loopdrives;
1200                }
1201
1202            }
1203
1204            printf("Enter Logical Drive Number :");
1205
1206            int getLogicalDriveNumber = 0;
1207            getLogicalDriveNumber = getInt();
1208
1209            printf("\n");
1210
1211            if (getLogicalDriveNumber > 0 && getLogicalDriveNumber <=
                  drivecount) {
1212
1213                int countdrive = 0;
1214                int driveloop;
1215
1216                for (driveloop = 0; driveloop < 26; driveloop++) {
1217
1218                    if (drivenum[driveloop] != 0) {
1219
1220                        Drive2[4] = ('A') + drivenum[driveloop];
1221                        countdrive = countdrive + 1;
1222
1223                        if (countdrive == getLogicalDriveNumber) {
1224
1225                            ScanLogicalDisk_DisplayDisk(Drive2);
1226                            printf("\n\nOperation Completed.\n");
1227                            system("pause");
1228                            Display_Graphics(privilege);
1229                            break;
1230
1231                        }
1232
1233                    }
1234
1235                }
1236
```

```
1237            }
1238            else {
1239
1240                printf("\nInvalid Drive. Please Try Again.\n");
1241                system("pause");
1242                Display_Graphics(privilege);
1243
1244            }
1245
1246        }
1247        else if (option == '3') {
1248
1249            printf("\n");
1250            printf("Physical Volumes : \n");
1251
1252            HANDLE device[100];
1253            TCHAR strPathFinal[100][20];
1254
1255            for (int clear1 = 0; clear1 < 100; clear1++) {
1256                for (int clear2 = 0; clear2 < 20; clear2++) {
1257                    strPathFinal[clear1][clear2] = NULL;
1258                }
1259            }
1260
1261            int diskloop;
1262            int diskcounter = 0;
1263
1264            for (diskloop = 0; diskloop < 100; diskloop++) {
1265
1266                TCHAR strPath0[] = L"\\\\.\\PhysicalDrive0";
1267                TCHAR strPath1[] = L"\\\\.\\PhysicalDrive10";
1268                TCHAR strPath2[] = L"\\\\.\\PhysicalDrive20";
1269                TCHAR strPath3[] = L"\\\\.\\PhysicalDrive30";
1270                TCHAR strPath4[] = L"\\\\.\\PhysicalDrive40";
1271                TCHAR strPath5[] = L"\\\\.\\PhysicalDrive50";
1272                TCHAR strPath6[] = L"\\\\.\\PhysicalDrive60";
1273                TCHAR strPath7[] = L"\\\\.\\PhysicalDrive70";
1274                TCHAR strPath8[] = L"\\\\.\\PhysicalDrive80";
1275                TCHAR strPath9[] = L"\\\\.\\PhysicalDrive90";
1276
1277
1278                if (diskloop >= 0 && diskloop < 10) {
1279                    strPath0[17] = ('0') + diskloop;
1280                    device[diskloop] = CreateFile(strPath0, GENERIC_READ,
                        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                        FILE_FLAG_NO_BUFFERING, NULL);
1281
1282                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1283                    {
1284                        diskcounter = diskcounter + 1;
1285                        wcscpy(strPathFinal[diskloop], strPath0);
1286                        printf("%d. %ws\n", diskcounter, strPathFinal
```

```
                    [diskloop]);
1287                }
1288            }
1289            else if (diskloop >= 10 && diskloop < 20) {
1290                strPath1[18] = ('0') + (diskloop - 10);
1291                device[diskloop] = CreateFile(strPath1, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
1292
1293                if (device[diskloop] != INVALID_HANDLE_VALUE)
1294                {
1295                    diskcounter = diskcounter + 1;
1296                    wcscpy(strPathFinal[diskloop], strPath1);
1297                    printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
1298                }
1299            }
1300            else if (diskloop >= 20 && diskloop < 30) {
1301                strPath2[18] = ('0') + (diskloop - 20);
1302                device[diskloop] = CreateFile(strPath2, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
1303
1304                if (device[diskloop] != INVALID_HANDLE_VALUE)
1305                {
1306                    diskcounter = diskcounter + 1;
1307                    wcscpy(strPathFinal[diskloop], strPath2);
1308                    printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
1309                }
1310
1311            }
1312            else if (diskloop >= 30 && diskloop < 40) {
1313                strPath3[18] = ('0') + (diskloop - 30);
1314                device[diskloop] = CreateFile(strPath3, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
1315
1316                if (device[diskloop] != INVALID_HANDLE_VALUE)
1317                {
1318                    diskcounter = diskcounter + 1;
1319                    wcscpy(strPathFinal[diskloop], strPath3);
1320                    printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
1321                }
1322
1323            }
1324            else if (diskloop >= 40 && diskloop < 50) {
1325                strPath4[18] = ('0') + (diskloop - 40);
1326                device[diskloop] = CreateFile(strPath4, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
```

```
1327
1328                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1329                    {
1330                        diskcounter = diskcounter + 1;
1331                        wcscpy(strPathFinal[diskloop], strPath4);
1332                        printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
1333                    }
1334
1335                }
1336                else if (diskloop >= 50 && diskloop < 60) {
1337                    strPath5[18] = ('0') + (diskloop - 50);
1338                    device[diskloop] = CreateFile(strPath5, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
1339
1340                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1341                    {
1342                        diskcounter = diskcounter + 1;
1343                        wcscpy(strPathFinal[diskloop], strPath5);
1344                        printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
1345                    }
1346
1347                }
1348                else if (diskloop >= 60 && diskloop < 70) {
1349                    strPath6[18] = ('0') + (diskloop - 60);
1350                    device[diskloop] = CreateFile(strPath6, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
1351
1352                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1353                    {
1354                        diskcounter = diskcounter + 1;
1355                        wcscpy(strPathFinal[diskloop], strPath6);
1356                        printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
1357                    }
1358
1359                }
1360                else if (diskloop >= 70 && diskloop < 80) {
1361                    strPath7[18] = ('0') + (diskloop - 70);
1362                    device[diskloop] = CreateFile(strPath7, GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                    FILE_FLAG_NO_BUFFERING, NULL);
1363
1364                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1365                    {
1366                        diskcounter = diskcounter + 1;
1367                        wcscpy(strPathFinal[diskloop], strPath7);
1368                        printf("%d. %ws\n", diskcounter, strPathFinal
                    [diskloop]);
```

```
1369                    }
1370
1371                }
1372                else if (diskloop >= 80 && diskloop < 90) {
1373                    strPath8[18] = ('0') + (diskloop - 80);
1374                    device[diskloop] = CreateFile(strPath8, GENERIC_READ,
                        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                        FILE_FLAG_NO_BUFFERING, NULL);
1375
1376                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1377                    {
1378                        diskcounter = diskcounter + 1;
1379                        wcscpy(strPathFinal[diskloop], strPath8);
1380                        printf("%d. %ws\n", diskcounter, strPathFinal
                        [diskloop]);
1381                    }
1382
1383                }
1384                else if (diskloop >= 90 && diskloop < 100) {
1385                    strPath9[18] = ('0') + (diskloop - 90);
1386                    device[diskloop] = CreateFile(strPath9, GENERIC_READ,
                        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                        FILE_FLAG_NO_BUFFERING, NULL);
1387
1388                    if (device[diskloop] != INVALID_HANDLE_VALUE)
1389                    {
1390                        diskcounter = diskcounter + 1;
1391                        wcscpy(strPathFinal[diskloop], strPath9);
1392                        printf("%d. %ws\n", diskcounter, strPathFinal
                        [diskloop]);
1393                    }
1394
1395                }
1396
1397            }
1398
1399            printf("Enter Physical Drive : ");
1400            int getDisk = getInt();
1401            printf("\n");
1402            int validcounter = 0;
1403
1404            if (getDisk > 0 && getDisk <= diskcounter) {
1405
1406                for (diskloop = 0; diskloop < 100; diskloop++) {
1407
1408                    if (device[diskloop] != INVALID_HANDLE_VALUE) {
1409
1410                        validcounter = validcounter + 1;
1411                        if (validcounter == getDisk) {
1412                            UNICODE_STRING dStr;
1413                            dStr.Buffer = strPathFinal[diskloop];
1414                            ScanPhysicalDisk_DisplayDisk(device[diskloop],
```

```
                        dStr);
1415
1416                            for (diskloop = 0; diskloop < 100; diskloop++) {
1417
1418                                if (device[diskloop] != INVALID_HANDLE_VALUE) {
1419
1420                                    CloseHandle(device[diskloop]);
1421
1422                                }
1423
1424                            }
1425
1426                            printf("\n\nOperation Completed.\n");
1427                            system("pause");
1428                            Display_Graphics(privilege);
1429                            break;
1430                        }
1431
1432                    }
1433
1434                }
1435            }
1436            else {
1437
1438                printf("\nInvalid Drive. Please Try Again.\n");
1439                system("pause");
1440                Display_Graphics(privilege);
1441
1442            }
1443
1444        }
1445        else if (option == '4') {
1446
1447            FILE *f;
1448
1449
1450            system("netstat -a -b -ano >> NetworkStats.txt");
1451            f = fopen("NetworkStats.txt", "a");
1452            fprintf(f, "\n\n-------------------------------------------------    ↵
                  EOF---------------------------------------------------------- ↵
                  --\n\n\n");
1453            printf("\n\nOperation Completed.\n");
1454            system("pause");
1455            Display_Graphics(privilege);
1456
1457            fclose(f);
1458
1459        }
1460        else if (option == '5') {
1461
1462          long long int psniff = -1;
1463
```

```
1464            while (1) {
1465                printf("\nEnter Number of Packets to Sniff (Minimum 1) : ");
1466                psniff = getInt();
1467                if (psniff >= 1)
1468                    break;
1469            }
1470
1471            printf("\n");
1472            int sniffer_result = InitializeSniffer(psniff);
1473
1474            if (sniffer_result == 0) {
1475                printf("\n\nOperation Completed.\n");
1476                system("pause");
1477                Display_Graphics(privilege);
1478            }
1479            else {
1480                printf("\n\nOperation Could Not Be Completed.\n");
1481                system("pause");
1482                Display_Graphics(privilege);
1483            }
1484
1485        }
1486        else if (option == '6') {
1487
1488
1489        }
1490        else if (option == '7') {
1491
1492
1493        }
1494        else if (option == '8') {
1495
1496
1497        }
1498        else if (option == '9') {
1499
1500            BOOL areThereTxtFiles = TRUE;
1501            int txtFiles = 0;
1502
1503
1504            wchar_t* file = L"*.txt";
1505            WIN32_FIND_DATA FindFileData;
1506
1507            HANDLE hFind;
1508            TCHAR txtFileName[100][265];
1509
1510            hFind = FindFirstFile(file, &FindFileData);
1511
1512            if (hFind != INVALID_HANDLE_VALUE) {
1513
1514                wcscpy(txtFileName[txtFiles], FindFileData.cFileName);
1515                txtFiles++;
```

```
1516
1517                while ((areThereTxtFiles = FindNextFile(hFind, &FindFileData))
                      == TRUE) {
1518
1519                    wcscpy(txtFileName[txtFiles], FindFileData.cFileName);
1520                    txtFiles++;
1521
1522                }
1523            }
1524
1525            printf("\nTotal Number Of TXT Files :  %d\n", txtFiles);
1526
1527            int whichFile;
1528            UNICODE_STRING unicodeTxtFileName;
1529
1530            while (1) {
1531
1532                for (int filenum = 0; filenum < txtFiles; filenum++) {
1533                    printf("%d. %ws\n", (filenum + 1), txtFileName[filenum]);
1534                }
1535
1536                printf("Select The File To Be Opened : ");
1537                whichFile = getInt();
1538                if (whichFile > 0 && whichFile <= txtFiles) {
1539                    break;
1540                }
1541                else {
1542                    printf("Invalid Input. Please Try Again.\n\n");
1543                }
1544            }
1545            printf("\n");
1546
1547            unicodeTxtFileName.Buffer = txtFileName[whichFile - 1];
1548
1549            int readStatus = readHexandCharStream(unicodeTxtFileName);
1550
1551            printf("\n\nOperation Completed.\n");
1552            system("pause");
1553            Display_Graphics(privilege);
1554
1555        }
1556        else if (option == 'Q' || option == 'q') {
1557            break;
1558        }
1559        else {
1560            printf("\nInvalid Input. Please Try Again.\n");
1561            system("pause");
1562            Display_Graphics(privilege);
1563        }
1564    }
1565
1566
```

```
1567      VirtualFree(buffer, 0, MEM_RELEASE);
1568      printf("\n\nExiting.\n");
1569      system("pause");
1570      return 0;
1571
1572  }
1573
```