Praise be to Allah

Astor: A Practical Parallel Antivirus Engine

Initial Report

2/2/2014

Ahmad Siavashi
E-Mail: a.siavosh@yahoo.com
Department of Computer Science and Engineering
School of Engineering, Shiraz University, Shiraz, Iran

Submitted to—
Prof. Farshad Khunjush
Department of Computer Science and Engineering
School of Engineering, Shiraz University, Shiraz, Iran

Statement of Problem

While scanning files, H.D.D is system's bottleneck. Since it provides a small amount of data, there will be no need for parallelization, i.e**. it is not worthwhile to implement a concurrent engine**.

Solution

Making it worthwhile by trying to improve the read speed of the drive.

Technical Approach

There is a technology used in hard disk drives, named NCQ (Native Command Queuing). If we send read requests for the H.D.D controller, e.g. Read requests for files A, B, C and D then the device starts reading the file which is nearest to the Head, not the file which has been sent first.

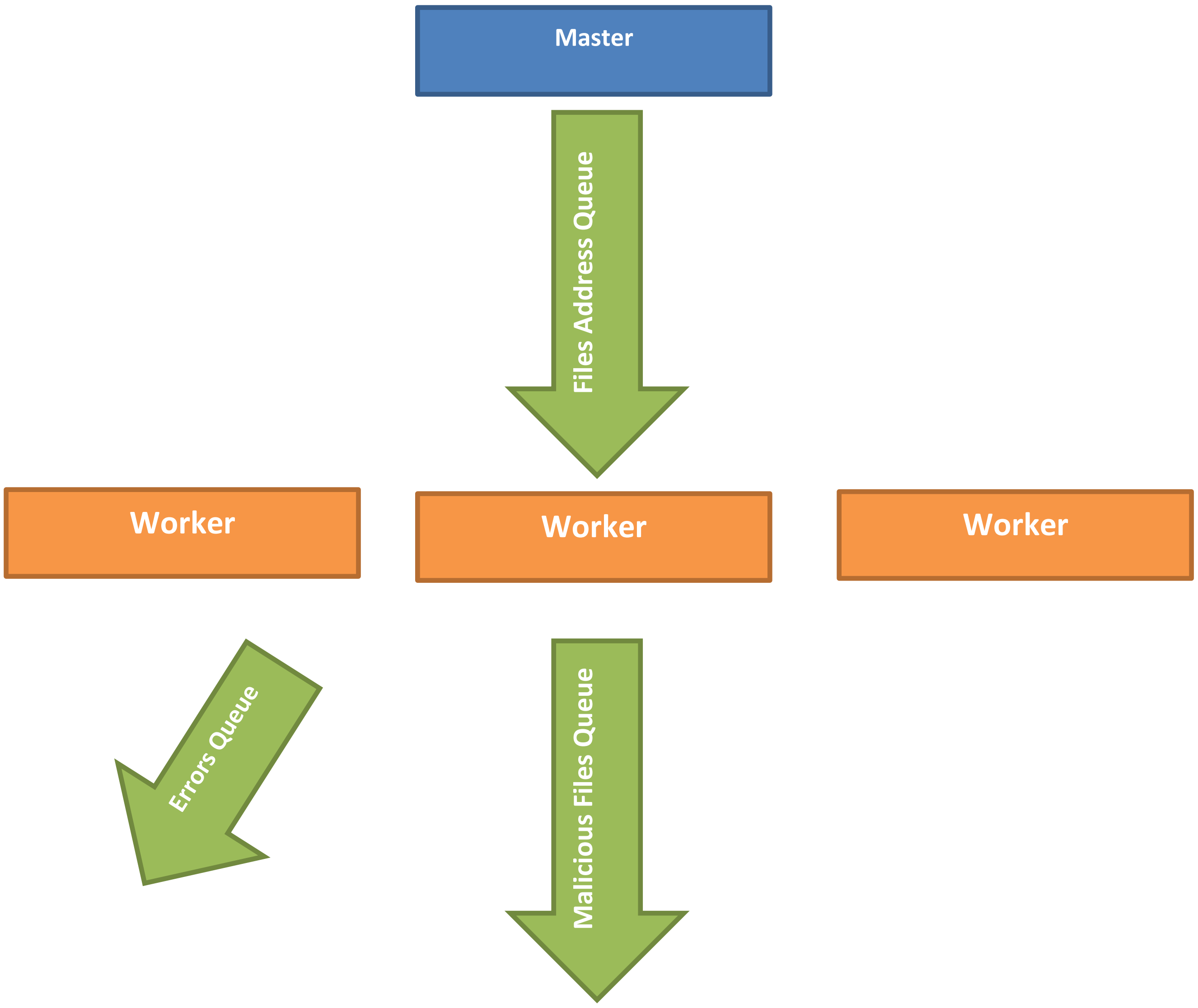
Implementation

I implemented a signature based AV engine (VeronicaAV) to test the idea. There are 5 revisions of the engine in Appendix A but I will explain the architecture of the latest version.

- We have <u>a master</u> thread and <u>10 worker threads.</u>
- Master: Collects all the file addresses in the given directory and inserts them in a global queue.
- Workers: Each worker Takes one file, scans it then goes for the next file address in the global queue.
- Workers insert the malicious files into a separated queue. They also insert the occurred errors into the global error queue.
- The master thread listens to the malicious files queue as well as error queue.

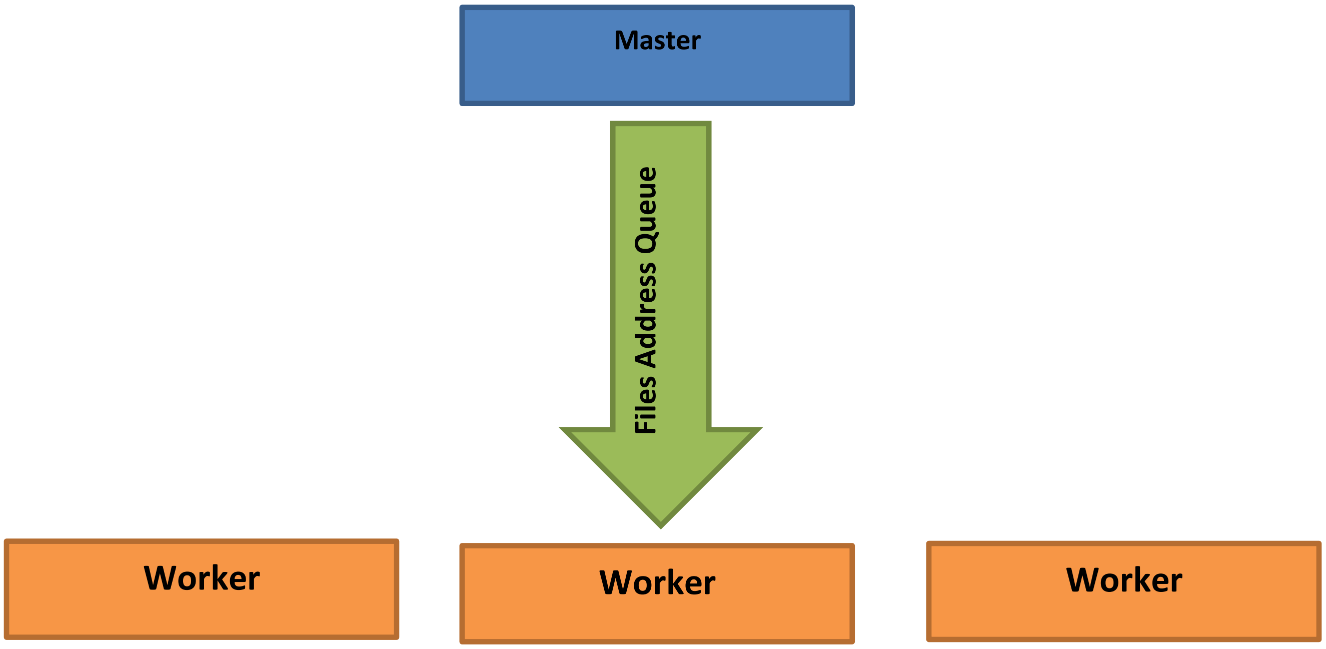
(Actually the true concurrency of my system is 4 threads simultaneously but as I was working with I/O I could achieve a better result with more threads.)

This way we have several requests sent to the H.D.D approximately at the same time.



The Final Architecture

I am still studying concurrency in C++ programming language, thus to test the idea I implemented a very simple and naïve implementation which looks like this:



Current Implementation

And here are the results. I have tested the implementation on 2 different systems and I have rebooted the system to take every sample.

System 1

Directory: C:\Python27 (Python 2.7.3)

Size: 43.7794 MB

#Files: 3158 Files

	Elapsed Time (Second)	Average Speed (MB/Second)
WinC++.cpp (Simple Sequential Implementation Using Win32 APIs)	47.1820	0.9279
WinC++_Queue_MultiThreaded.cpp (Using a global queue and 4 threads)	26.6940	1.6400
WinC++_Queue_MultiThreaded.cpp (Using a global queue and 10 threads)	24.3020	1.8015

Speed-up: 1.94X

System 2

Directory: C:\Python27 (Python 2.7.3)

+ Plugins & Extensions

Size: 69.50 MB

#Files: 7492 Files

	Elapsed Time (Second)	Average Speed (MB/Second)
WinC++.cpp (Simple Sequential Implementation Using Win32 APIs)	199.5140	0.3483
WinC++_Queue_MultiThreaded.cpp (Using a global queue and 10 threads)	54.4840	1.2756

Speed-up: 3.66X

Implementation Pitfalls:

- Although I've improved the read speed, but the implementation is very basic, I did not even have a thread-safe queue for C++.
- There are a lot of prints on the screen done by worker threads, which may decrease the overall performance of the program.
- The master thread collects all the addresses before workers begin their work; it's not a good idea.
- Etc.

Big Question

- **Why this method is not used in ClamAV?**

  I do not have the answer yet.

Next Step

I have to implement something similar inside ClamAV for further studies.

Appendix A

WinC++.cpp

Sequential Version

```
#include <Windows.h>
#include <ctime>
#include <iostream>
#include <string>
#include <new>
#include <sstream>

using namespace std;

enum class ReturnCode { VAV_FALUIRE, VAV_SUCCESS };
enum class LogCode {
PRINT_FINAL_RESULTS,READ_FILE_ERR,DIR_NOT_FOUND,MEM_ALOC_FILE_ERR,OPEN_SIGN_FILE_ERR,MEM_
ALOC_SIGN_FILE_ERR,OPEN_FILE_ERR,AFFECTED_FILE,CLEAN_FILE };

struct File {
	BYTE *pFile;
	LARGE_INTEGER FileSize;
	string Target;
	HANDLE hFileStream;
	ReturnCode Release();
};

ReturnCode File::Release(){
	delete[] pFile;
	CloseHandle(hFileStream);
	return ReturnCode::VAV_SUCCESS;
}

struct SignatureDatabase {
	BYTE *pDatabase;
	DWORD DatabaseSize;
	string Target;
	HANDLE hFile;
	ReturnCode Release();
};

ReturnCode SignatureDatabase::Release(){
	delete[] pDatabase;
	CloseHandle(hFile);
	return ReturnCode::VAV_SUCCESS;
}

struct ScanResult {
	SIZE_T Detected;
	SIZE_T TotalScanned;
	SIZE_T Errors;
	FLOAT Clock;
	ULONG ScanSize;
	FLOAT ElapsedTime() { return (clock() - Clock)/CLOCKS_PER_SEC; };
	FLOAT AverageSizePerTime(){ return ScanSizeInMB() / ElapsedTime(); };
	FLOAT ScanSizeInMB(){return ScanSize / (1024.0*1024); };
	VOID Reset() { Clock = clock() ; ScanSize = 0; Detected = 0 ;  TotalScanned = 0 ;
Errors = 0; };
	ScanResult() : Clock(clock()) , ScanSize(0) , Detected(0) , TotalScanned(0) ,
Errors(0){};
};

/******************** Global Variables ********************/
ScanResult                   GlobalScanResult;
SignatureDatabase    GlobalSignatureDatabase;
/*********************************************************/

VOID Log(LogCode Code, const string& Argument = ""){
	switch(Code){
	case LogCode::PRINT_FINAL_RESULTS:
		printf(" -----------------------------------------------------------------
\n");
		printf("|              # Scanned  Files   : %-10d  File(s)
|\n",GlobalScanResult.TotalScanned);
		printf("|              # Detected Files   : %-10d  File(s)
|\n",GlobalScanResult.Detected);
		printf("|              # Occured Errors   : %-10d  File(s)
|\n",GlobalScanResult.Errors);
		printf("|              # Elapsed Time     : %-10.4f  Second(s)
|\n",GlobalScanResult.ElapsedTime());
		printf("|              # Scan Size        : %-10.4f  MB
|\n",GlobalScanResult.ScanSizeInMB());
		printf("|              # Size/Time        : %-10.4f  MB/S
|\n",GlobalScanResult.AverageSizePerTime());
		printf(" -----------------------------------------------------------------
\n");
		break;
	case LogCode::READ_FILE_ERR:
		cout << "[-] Error While Reading File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
		break;
	case LogCode::DIR_NOT_FOUND:
		cout << "[-] Directory Not Found : \'" << Argument << "'\n";
		break;
	case LogCode::MEM_ALOC_FILE_ERR:
		cout << "[-] Error While Allocating Memory For File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
		break;
	case LogCode::OPEN_SIGN_FILE_ERR:
		cout << "[-] Error While Opening The Signature File.\n";
		break;
```

```
        case LogCode::MEM_ALOC_SIGN_FILE_ERR:
                cout << "[-] Error While Allocating Memory For The Signature File.\n";
                break;
        case LogCode::OPEN_FILE_ERR:
                cout << "[-] Error While Opening File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
                break;
        case LogCode::AFFECTED_FILE:
                cout << "[-] AFFECTED  : " << strrchr(Argument.c_str(),'\\') + 1 << "\n";
                break;
        case LogCode::CLEAN_FILE:
                cout << "[-] CLEAN     : " << strrchr(Argument.c_str(),'\\') + 1 << "\n";
                break;
        }
}

//      Converting a One Byte Hex String To Its Equivalent Integer Value.
INT OneByteAsciiHexToInt(CHAR *String){
        INT i=0,Value=0;
        while(String[i] != '\0'){
                if(String[i] >= 'A' && String[i] <= 'F'){
                        Value += (String[i]-'A' + 10)*(pow((DOUBLE)16,(INT)1-i));
                }else if(String[i] >= 'a' && String[i] <= 'f'){
                        Value += (String[i]-'a' + 10)*(pow((DOUBLE)16,(INT)1-i));
                }else{
                        Value += (String[i]-'0')*(pow((DOUBLE)16,(INT)1-i));
                }
                i++;
        }
        return Value;
}

VOID ConsoleInitializer(VOID){
        SetConsoleTitle(L"Veronica Antivirus");
        cout << " ------------------------------------------------------------- \n";
        cout << "|                      Veronica Antivirus                     |\n";
        cout << "|                  Written By Ahmad Siavashi                  |\n";
        cout << "|                 Email : a.siavosh@yahoo.com                 |\n";
        cout << "|                      Shiraz  University                     |\n";
        cout << "|                         Spring 2012                         |\n";
        cout << " ------------------------------------------------------------- \n";
}

//      String Matching Algorithm.
bool DetectSignatureInFile(File * pFile){
        for(int i=0;i<pFile->FileSize.QuadPart;i++){
                int j,k;
                if(pFile->pFile[i] == GlobalSignatureDatabase.pDatabase[0]){
                        for(j=1,k=i+1;j < GlobalSignatureDatabase.DatabaseSize && k < pFile-
>FileSize.QuadPart;j++,k++){
                                if(GlobalSignatureDatabase.pDatabase[j] != pFile->pFile[k])
                                        break;
                        }
                        if(j == GlobalSignatureDatabase.DatabaseSize){
                                return true;
                        }
                }
        }
        return false;
}

//      Obtaining The Signature File.
ReturnCode LoadSignature(string& SignatureFileName){
        INT i=0,j=0,c='\0',k=0;
        CHAR aChar[3];
        CHAR *pReadBuffer;
        DWORD dwBytesRead;
        LARGE_INTEGER FileSize;
        GlobalSignatureDatabase.Target = string(SignatureFileName);
        if((GlobalSignatureDatabase.hFile =
CreateFileA(SignatureFileName.c_str(),GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FIL
E_ATTRIBUTE_NORMAL | FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH ,NULL)) ==
INVALID_HANDLE_VALUE){
                Log(LogCode::OPEN_SIGN_FILE_ERR);
                system("PAUSE");
                exit(EXIT_FAILURE);
        }
        GetFileSizeEx(GlobalSignatureDatabase.hFile,&FileSize);
        GlobalSignatureDatabase.DatabaseSize = (FileSize.QuadPart+1)/3;
        if((GlobalSignatureDatabase.pDatabase = new (nothrow)
BYTE[int(ceil((float)FileSize.QuadPart / 512)*512)])==nullptr){
                Log(LogCode::MEM_ALOC_SIGN_FILE_ERR);
                system("PAUSE");
                exit(EXIT_FAILURE);
        }
        pReadBuffer = new (nothrow) CHAR[FileSize.QuadPart];
        ReadFile(GlobalSignatureDatabase.hFile,pReadBuffer,ceil((float)FileSize.QuadPart /
512)*512, &dwBytesRead,nullptr);
        while((c=pReadBuffer[k]) != '\n' && c!=EOF && k++ <= dwBytesRead){
                if(c != ' '){
                        aChar[j++] = c;
                        if(j==2){
                                aChar[j] = '\0';
                                GlobalSignatureDatabase.pDatabase[i] =
OneByteAsciiHexToInt(aChar);
                                j = 0;
                                i++;
                        }
                }
        }
        delete[] pReadBuffer;
        return ReturnCode::VAV_SUCCESS;
}

File * LoadFile(CONST CHAR *FileName){
        File * pFile = new File;
        pFile->Target = string(FileName);
        if((pFile->hFileStream = CreateFileA(pFile-
>Target.c_str(),GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH,NULL))==INVALID_HANDLE_VALUE){
                Log(LogCode::OPEN_FILE_ERR,pFile->Target);
                delete pFile;
                return nullptr;
        }

        DWORD dwBytesRead;
```

```
	GetFileSizeEx(pFile->hFileStream,&pFile->FileSize);
	if((pFile->pFile = new BYTE[int(ceil((float)pFile->FileSize.QuadPart / 512)*512)])
== NULL){
		Log(LogCode::MEM_ALOC_FILE_ERR,pFile->Target);
		delete pFile;
		return nullptr;
	}

	if(ReadFile(pFile->hFileStream,pFile->pFile,ceil((float)pFile->FileSize.QuadPart /
512)*512, &dwBytesRead,NULL) == 0){
		Log(LogCode::READ_FILE_ERR,pFile->Target);
		free(pFile->pFile);
		delete pFile;
		return nullptr;
	}
	return pFile;
}

bool ScanDirectoryFiles(CHAR * Dir){
	HANDLE				hFindFile;
	File	*			pFile = nullptr;
	WIN32_FIND_DATAA	Win32FindData;
	CHAR				Directory[MAX_PATH];

	sprintf(Directory,"%s\\*.*",Dir);
	if((hFindFile=FindFirstFileA(Directory,&Win32FindData))==INVALID_HANDLE_VALUE){
		Log(LogCode::DIR_NOT_FOUND,Dir);
		return false;
	}

	do{
		if(strcmp(Win32FindData.cFileName,".") != 0 &&
strcmp(Win32FindData.cFileName,"..") != 0){
			sprintf(Directory,"%s\\%s",Dir,Win32FindData.cFileName);
			if(Win32FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
				ScanDirectoryFiles(Directory);
			}else{
				if((pFile = LoadFile(Directory)) != nullptr){
					if(DetectSignatureInFile(pFile)){
						GlobalScanResult.Detected++;
						Log(LogCode::AFFECTED_FILE,Directory);
					}else{
						Log(LogCode::CLEAN_FILE,Directory);
					}
					GlobalScanResult.TotalScanned++;
					GlobalScanResult.ScanSize += pFile->FileSize.QuadPart;
					pFile->Release();
					delete pFile;

					CHAR ConsoleTitle[MAX_PATH];
					sprintf(ConsoleTitle,"VAV - %d File(s) Scanned : %d
File(s) Detected - %d Error(s)
Occurred\n",GlobalScanResult.TotalScanned,GlobalScanResult.Detected,GlobalScanResult.Erro
rs);
					SetConsoleTitleA(ConsoleTitle);
				}else{
					GlobalScanResult.Errors++;
```

```
                            }
                    }
            }
    }while(FindNextFileA(hFindFile,&Win32FindData));
    FindClose(hFindFile);
    return TRUE;
}

INT main(){
    ConsoleInitializer();
    LoadSignature(string("signature.txt"));
    stringstream Args(" ");
    string Cmd,temp,Line,Directory("C:\\");
    cout << "For more information, type 'help'." << endl;
    do {
            cout << ">> ";
            getline(cin,Line);
            Args.clear();
            Args << Line;
            Cmd.erase();
            Args >> Cmd;

            if(Cmd == "exit"){
                    break;
            }else if(Cmd == "help"){
                    cout << "exit" << endl;
                    cout << "scan <dir>" << endl;
            }else if(Cmd == "scan"){
                    Args >> temp;
                    if(temp != "-"){
                            Directory = temp;
                    }
                    GlobalScanResult.Reset();
                    ScanDirectoryFiles((CHAR *)Directory.c_str());
                    Log(LogCode::PRINT_FINAL_RESULTS);
            }
    }while(true);
    GlobalSignatureDatabase.Release();
    return EXIT_SUCCESS;
}
```

WinC++_Queue_MultiThreaded.cpp

Final Version (10 Threads)

```
#include <Windows.h>
#include <ctime>
#include <iostream>
#include <string>
#include <new>
#include <sstream>
```

```
#include <thread>
#include <mutex>
#include <queue>

using namespace std;

enum class ReturnCode { VAV_FALUIRE, VAV_SUCCESS };
enum class LogCode {
PRINT_FINAL_RESULTS,READ_FILE_ERR,DIR_NOT_FOUND,MEM_ALOC_FILE_ERR,OPEN_SIGN_FILE_ERR,MEM_
ALOC_SIGN_FILE_ERR,OPEN_FILE_ERR,AFFECTED_FILE,CLEAN_FILE };

struct File {
	BYTE *pFile;
	LARGE_INTEGER FileSize;
	string Target;
	HANDLE hFileStream;
	ReturnCode Release();
};

ReturnCode File::Release(){
	delete[] pFile;
	CloseHandle(hFileStream);
	return ReturnCode::VAV_SUCCESS;
}

struct SignatureDatabase {
	BYTE *pDatabase;
	DWORD DatabaseSize;
	string Target;
	HANDLE hFile;
	ReturnCode Release();
};

ReturnCode SignatureDatabase::Release(){
	delete[] pDatabase;
	CloseHandle(hFile);
	return ReturnCode::VAV_SUCCESS;
}

struct ScanResult {
	SIZE_T Detected;
	SIZE_T TotalScanned;
	SIZE_T Errors;
	FLOAT Clock;
	ULONG ScanSize;
	FLOAT ElapsedTime() { return (clock() - Clock)/CLOCKS_PER_SEC; };
	FLOAT AverageSizePerTime(){ return ScanSizeInMB() / ElapsedTime(); };
	FLOAT ScanSizeInMB(){return ScanSize / (1024.0*1024); };
	VOID Reset() { Clock = clock() ; ScanSize = 0; Detected = 0 ;  TotalScanned = 0 ;
Errors = 0; };
	ScanResult() : Clock(clock()) , ScanSize(0) , Detected(0) , TotalScanned(0) ,
Errors(0){};
};

/******************** Global Variables *********************/
ScanResult                 GlobalScanResult;
SignatureDatabase    GlobalSignatureDatabase;
queue<string>        DirectoryQueue;
```

```
mutex                       QueueMutex;
/************************************************************/

VOID Log(LogCode Code, const string& Argument = ""){
       switch(Code){
       case LogCode::PRINT_FINAL_RESULTS:
              printf(" -----------------------------------------------------------------
\n");
              printf("|              # Scanned  Files   : %-10d  File(s)
|\n",GlobalScanResult.TotalScanned);
              printf("|              # Detected Files   : %-10d  File(s)
|\n",GlobalScanResult.Detected);
              printf("|              # Occured Errors   : %-10d  File(s)
|\n",GlobalScanResult.Errors);
              printf("|              # Elapsed Time     : %-10.4f  Second(s)
|\n",GlobalScanResult.ElapsedTime());
              printf("|              # Scan Size        : %-10.4f  MB
|\n",GlobalScanResult.ScanSizeInMB());
              printf("|              # Size/Time        : %-10.4f  MB/S
|\n",GlobalScanResult.AverageSizePerTime());
              printf(" -----------------------------------------------------------------
\n");
              break;
       case LogCode::READ_FILE_ERR:
              cout << "[-] Error While Reading File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
              break;
       case LogCode::DIR_NOT_FOUND:
              cout << "[-] Directory Not Found : \'" << Argument << "'\n";
              break;
       case LogCode::MEM_ALOC_FILE_ERR:
              cout << "[-] Error While Allocating Memory For File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
              break;
       case LogCode::OPEN_SIGN_FILE_ERR:
              cout << "[-] Error While Opening The Signature File.\n";
              break;
       case LogCode::MEM_ALOC_SIGN_FILE_ERR:
              cout << "[-] Error While Allocating Memory For The Signature File.\n";
              break;
       case LogCode::OPEN_FILE_ERR:
              cout << "[-] Error While Opening File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
              break;
       case LogCode::AFFECTED_FILE:
              cout << "[-] AFFECTED  : " << strrchr(Argument.c_str(),'\\') + 1 << "\n";
              break;
       case LogCode::CLEAN_FILE:
              cout << "[-] CLEAN     : " << strrchr(Argument.c_str(),'\\') + 1 << "\n";
              break;
       }
}

//     Converting a One Byte Hex String To Its Equivalent Integer Value.
INT OneByteAsciiHexToInt(CHAR *String){
       INT i=0,Value=0;
       while(String[i] != '\0'){
              if(String[i] >= 'A' && String[i] <= 'F'){
```

```
                        Value += (String[i]-'A' + 10)*(pow((DOUBLE)16,(INT)1-i));
                }else if(String[i] >= 'a' && String[i] <= 'f'){
                        Value += (String[i]-'a' + 10)*(pow((DOUBLE)16,(INT)1-i));
                }else{
                        Value += (String[i]-'0')*(pow((DOUBLE)16,(INT)1-i));
                }
                i++;
        }
        return Value;
}

VOID ConsoleInitializer(VOID){
        SetConsoleTitle(L"Veronica Antivirus");
        cout << " ------------------------------------------------------------ \n";
        cout << "|                      Veronica Antivirus                      |\n";
        cout << "|                  Written By Ahmad Siavashi                   |\n";
        cout << "|                 Email : a.siavosh@yahoo.com                  |\n";
        cout << "|                      Shiraz  University                      |\n";
        cout << "|                         Spring 2012                          |\n";
        cout << " ------------------------------------------------------------ \n";
}

//      String Matching Algorithm.
bool DetectSignatureInFile(File * pFile){
        for(int i=0;i<pFile->FileSize.QuadPart;i++){
                int j,k;
                if(pFile->pFile[i] == GlobalSignatureDatabase.pDatabase[0]){
                        for(j=1,k=i+1;j < GlobalSignatureDatabase.DatabaseSize && k < pFile-
>FileSize.QuadPart;j++,k++){
                                if(GlobalSignatureDatabase.pDatabase[j] != pFile->pFile[k])
                                        break;
                        }
                        if(j == GlobalSignatureDatabase.DatabaseSize){
                                return true;
                        }
                }
        }
        return false;
}

//      Obtaining The Signature File.
ReturnCode LoadSignature(string& SignatureFileName){
        INT i=0,j=0,c='\0',k=0;
        CHAR aChar[3];
        CHAR *pReadBuffer;
        DWORD dwBytesRead;
        LARGE_INTEGER FileSize;
        GlobalSignatureDatabase.Target = string(SignatureFileName);
        if((GlobalSignatureDatabase.hFile =
CreateFileA(SignatureFileName.c_str(),GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FIL
E_ATTRIBUTE_NORMAL | FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH ,NULL)) ==
INVALID_HANDLE_VALUE){
                Log(LogCode::OPEN_SIGN_FILE_ERR);
                system("PAUSE");
                exit(EXIT_FAILURE);
        }
        GetFileSizeEx(GlobalSignatureDatabase.hFile,&FileSize);
        GlobalSignatureDatabase.DatabaseSize = (FileSize.QuadPart+1)/3;
```

```
	if((GlobalSignatureDatabase.pDatabase = new (nothrow)
BYTE[int(ceil((float)FileSize.QuadPart / 512)*512)])==nullptr){
		Log(LogCode::MEM_ALOC_SIGN_FILE_ERR);
		system("PAUSE");
		exit(EXIT_FAILURE);
	}
	pReadBuffer = new (nothrow) CHAR[FileSize.QuadPart];
	ReadFile(GlobalSignatureDatabase.hFile,pReadBuffer,ceil((float)FileSize.QuadPart /
512)*512, &dwBytesRead,nullptr);
	while((c=pReadBuffer[k]) != '\n' && c!=EOF && k++ <= dwBytesRead){
		if(c != ' '){
			aChar[j++] = c;
			if(j==2){
				aChar[j] = '\0';
				GlobalSignatureDatabase.pDatabase[i] =
OneByteAsciiHexToInt(aChar);
				j = 0;
				i++;
			}
		}
	}
	delete[] pReadBuffer;
	return ReturnCode::VAV_SUCCESS;
}

File * LoadFile(CONST CHAR *FileName){
	File * pFile = new File;
	pFile->Target = string(FileName);
	if((pFile->hFileStream = CreateFileA(pFile-
>Target.c_str(),GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH,NULL))==INVALID_HANDLE_VALUE){
		Log(LogCode::OPEN_FILE_ERR,pFile->Target);
		delete pFile;
		return nullptr;
	}

	DWORD dwBytesRead;

	GetFileSizeEx(pFile->hFileStream,&pFile->FileSize);
	if((pFile->pFile = new BYTE[int(ceil((float)pFile->FileSize.QuadPart / 512)*512)])
== NULL){
		Log(LogCode::MEM_ALOC_FILE_ERR,pFile->Target);
		delete pFile;
		return nullptr;
	}

	if(ReadFile(pFile->hFileStream,pFile->pFile,ceil((float)pFile->FileSize.QuadPart /
512)*512, &dwBytesRead,NULL) == 0){
		Log(LogCode::READ_FILE_ERR,pFile->Target);
		free(pFile->pFile);
		delete pFile;
		return nullptr;
	}
	return pFile;
}

bool QueueDirectoryFiles(CHAR * Dir){
	HANDLE				hFindFile;
```

```
        WIN32_FIND_DATAA     Win32FindData;
        CHAR                        Directory[MAX_PATH];

        sprintf(Directory,"%s\\*.*",Dir);
        if((hFindFile=FindFirstFileA(Directory,&Win32FindData))==INVALID_HANDLE_VALUE){
                Log(LogCode::DIR_NOT_FOUND,Dir);
                return false;
        }

        do{
                if(strcmp(Win32FindData.cFileName,".") != 0 &&
strcmp(Win32FindData.cFileName,"..") != 0){
                        sprintf(Directory,"%s\\%s",Dir,Win32FindData.cFileName);
                        if(Win32FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
                                QueueDirectoryFiles(Directory);
                        }else{
                                DirectoryQueue.push(string(Directory));
                        }
                }
        }while(FindNextFileA(hFindFile,&Win32FindData));
        FindClose(hFindFile);
        return TRUE;
}

VOID ScanFile(){
        while(true){
                QueueMutex.lock();
                if(DirectoryQueue.empty()) break;
                string FileName = DirectoryQueue.front();
                DirectoryQueue.pop();
                QueueMutex.unlock();
                File * pFile = nullptr;
                if((pFile = LoadFile((const char *)FileName.c_str())) != nullptr){
                        if(DetectSignatureInFile(pFile)){
                                GlobalScanResult.Detected++;
                                Log(LogCode::AFFECTED_FILE,FileName);
                        }else{
                                Log(LogCode::CLEAN_FILE,FileName);
                        }
                        GlobalScanResult.TotalScanned++;
                        GlobalScanResult.ScanSize += pFile->FileSize.QuadPart;
                        pFile->Release();
                        delete pFile;

                        CHAR ConsoleTitle[MAX_PATH];
                        sprintf(ConsoleTitle,"VAV - %d File(s) Scanned : %d File(s) Detected
- %d Error(s)
Occurred\n",GlobalScanResult.TotalScanned,GlobalScanResult.Detected,GlobalScanResult.Erro
rs);
                        SetConsoleTitleA(ConsoleTitle);
                        /**/
                }else{
                        GlobalScanResult.Errors++;
                }
                /**/
        }
        if(!QueueMutex.try_lock())
                QueueMutex.unlock();
```

```
}

VOID Scan(string Directory){
	GlobalScanResult.Reset();
	QueueDirectoryFiles((CHAR *) Directory.c_str());
	vector<thread> threads(10);
	threads[0] = thread(ScanFile);
	threads[1] = thread(ScanFile);
	threads[2] = thread(ScanFile);
	threads[3] = thread(ScanFile);
	threads[4] = thread(ScanFile);
	threads[5] = thread(ScanFile);
	threads[6] = thread(ScanFile);
	threads[7] = thread(ScanFile);
	threads[8] = thread(ScanFile);
	threads[9] = thread(ScanFile);
	for_each(threads.begin(),threads.end(),std::mem_fn(&thread::join));
}

INT main(){
	ConsoleInitializer();
	LoadSignature(string("signature.txt"));
	stringstream Args(" ");
	string Cmd,temp,Line,Directory("C:\\");
	cout << "For more information, type 'help'." << endl;
	do {
		cout << ">> ";
		getline(cin,Line);
		Args.clear();
		Args << Line;
		Cmd.erase();
		Args >> Cmd;

		if(Cmd == "exit"){
			break;
		}else if(Cmd == "help"){
			cout << "exit" << endl;
			cout << "scan <dir>" << endl;
		}else if(Cmd == "scan"){
			Args >> temp;
			if(temp != "-"){
				Directory = temp;
			}
			Scan(Directory);
			Log(LogCode::PRINT_FINAL_RESULTS);
		}
	}while(true);
	GlobalSignatureDatabase.Release();
	return EXIT_SUCCESS;
}
```

------------------------------ Older Implementations ----------------------------

WinC++_Queue.cpp

Sequential + Gathering Addresses at first

```
#include <Windows.h>
#include <ctime>
#include <iostream>
#include <string>
#include <new>
#include <sstream>
#include <queue>

using namespace std;

enum class ReturnCode { VAV_FALUIRE, VAV_SUCCESS };
enum class LogCode {
PRINT_FINAL_RESULTS,READ_FILE_ERR,DIR_NOT_FOUND,MEM_ALOC_FILE_ERR,OPEN_SIGN_FILE_ERR,MEM_
ALOC_SIGN_FILE_ERR,OPEN_FILE_ERR,AFFECTED_FILE,CLEAN_FILE };

struct File {
	BYTE *pFile;
	LARGE_INTEGER FileSize;
	string Target;
	HANDLE hFileStream;
	ReturnCode Release();
};

ReturnCode File::Release(){
	delete[] pFile;
	CloseHandle(hFileStream);
	return ReturnCode::VAV_SUCCESS;
}

struct SignatureDatabase {
	BYTE *pDatabase;
	DWORD DatabaseSize;
	string Target;
	HANDLE hFile;
	ReturnCode Release();
};

ReturnCode SignatureDatabase::Release(){
	delete[] pDatabase;
	CloseHandle(hFile);
	return ReturnCode::VAV_SUCCESS;
}

struct ScanResult {
	SIZE_T Detected;
	SIZE_T TotalScanned;
	SIZE_T Errors;
	FLOAT Clock;
	ULONG ScanSize;
	FLOAT ElapsedTime() { return (clock() - Clock)/CLOCKS_PER_SEC; };
	FLOAT AverageSizePerTime(){ return ScanSizeInMB() / ElapsedTime(); };
	FLOAT ScanSizeInMB(){return ScanSize / (1024.0*1024); };
```

```
       VOID Reset() { Clock = clock() ; ScanSize = 0; Detected = 0 ;  TotalScanned = 0 ;
Errors = 0; };
       ScanResult() : Clock(clock()) , ScanSize(0) , Detected(0) , TotalScanned(0) ,
Errors(0){};
};

/******************** Global Variables ********************/
ScanResult                 GlobalScanResult;
SignatureDatabase    GlobalSignatureDatabase;
queue<string> DirectoryQueue;
/**********************************************************/

VOID Log(LogCode Code, const string& Argument = ""){
       switch(Code){
       case LogCode::PRINT_FINAL_RESULTS:
              printf(" ----------------------------------------------------------------
\n");
              printf("|              # Scanned  Files   : %-10d  File(s)
|\n",GlobalScanResult.TotalScanned);
              printf("|              # Detected Files   : %-10d  File(s)
|\n",GlobalScanResult.Detected);
              printf("|              # Occured Errors   : %-10d  File(s)
|\n",GlobalScanResult.Errors);
              printf("|              # Elapsed Time     : %-10.4f  Second(s)
|\n",GlobalScanResult.ElapsedTime());
              printf("|              # Scan Size        : %-10.4f  MB
|\n",GlobalScanResult.ScanSizeInMB());
              printf("|              # Size/Time        : %-10.4f  MB/S
|\n",GlobalScanResult.AverageSizePerTime());
              printf(" ----------------------------------------------------------------
\n");
              break;
       case LogCode::READ_FILE_ERR:
              cout << "[-] Error While Reading File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
              break;
       case LogCode::DIR_NOT_FOUND:
              cout << "[-] Directory Not Found : \'" << Argument << "'\n";
              break;
       case LogCode::MEM_ALOC_FILE_ERR:
              cout << "[-] Error While Allocating Memory For File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
              break;
       case LogCode::OPEN_SIGN_FILE_ERR:
              cout << "[-] Error While Opening The Signature File.\n";
              break;
       case LogCode::MEM_ALOC_SIGN_FILE_ERR:
              cout << "[-] Error While Allocating Memory For The Signature File.\n";
              break;
       case LogCode::OPEN_FILE_ERR:
              cout << "[-] Error While Opening File : \'" <<
strrchr(Argument.c_str(),'\\') + 1 << "'\n";
              break;
       case LogCode::AFFECTED_FILE:
              cout << "[-] AFFECTED  : " << strrchr(Argument.c_str(),'\\') + 1 << "\n";
              break;
       case LogCode::CLEAN_FILE:
              cout << "[-] CLEAN     : " << strrchr(Argument.c_str(),'\\') + 1 << "\n";
```

```
			break;
		}
}

//	Converting a One Byte Hex String To Its Equivalent Integer Value.
INT OneByteAsciiHexToInt(CHAR *String){
	INT i=0,Value=0;
	while(String[i] != '\0'){
		if(String[i] >= 'A' && String[i] <= 'F'){
			Value += (String[i]-'A' + 10)*(pow((DOUBLE)16,(INT)1-i));
		}else if(String[i] >= 'a' && String[i] <= 'f'){
			Value += (String[i]-'a' + 10)*(pow((DOUBLE)16,(INT)1-i));
		}else{
			Value += (String[i]-'0')*(pow((DOUBLE)16,(INT)1-i));
		}
		i++;
	}
	return Value;
}

VOID ConsoleInitializer(VOID){
	SetConsoleTitle(L"Veronica Antivirus");
	cout << " ------------------------------------------------------------- \n";
	cout << "|                     Veronica Antivirus                      |\n";
	cout << "|                 Written By Ahmad Siavashi                   |\n";
	cout << "|                Email : a.siavosh@yahoo.com                  |\n";
	cout << "|                     Shiraz  University                      |\n";
	cout << "|                        Spring 2012                          |\n";
	cout << " ------------------------------------------------------------- \n";
}

//	String Matching Algorithm.
bool DetectSignatureInFile(File * pFile){
	for(int i=0;i<pFile->FileSize.QuadPart;i++){
		int j,k;
		if(pFile->pFile[i] == GlobalSignatureDatabase.pDatabase[0]){
			for(j=1,k=i+1;j < GlobalSignatureDatabase.DatabaseSize && k < pFile-
>FileSize.QuadPart;j++,k++){
				if(GlobalSignatureDatabase.pDatabase[j] != pFile->pFile[k])
					break;
			}
			if(j == GlobalSignatureDatabase.DatabaseSize){
				return true;
			}
		}
	}
	return false;
}

//	Obtaining The Signature File.
ReturnCode LoadSignature(string& SignatureFileName){
	INT i=0,j=0,c='\0',k=0;
	CHAR aChar[3];
	CHAR *pReadBuffer;
	DWORD dwBytesRead;
	LARGE_INTEGER FileSize;
	GlobalSignatureDatabase.Target = string(SignatureFileName);
	if((GlobalSignatureDatabase.hFile = 
CreateFileA(SignatureFileName.c_str(),GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FIL
E_ATTRIBUTE_NORMAL | FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH ,NULL)) == 
INVALID_HANDLE_VALUE){
		Log(LogCode::OPEN_SIGN_FILE_ERR);
		system("PAUSE");
		exit(EXIT_FAILURE);
	}
	GetFileSizeEx(GlobalSignatureDatabase.hFile,&FileSize);
	GlobalSignatureDatabase.DatabaseSize = (FileSize.QuadPart+1)/3;
	if((GlobalSignatureDatabase.pDatabase = new (nothrow) 
BYTE[int(ceil((float)FileSize.QuadPart / 512)*512)])==nullptr){
		Log(LogCode::MEM_ALOC_SIGN_FILE_ERR);
		system("PAUSE");
		exit(EXIT_FAILURE);
	}
	pReadBuffer = new (nothrow) CHAR[FileSize.QuadPart];
	ReadFile(GlobalSignatureDatabase.hFile,pReadBuffer,ceil((float)FileSize.QuadPart / 
512)*512, &dwBytesRead,nullptr);
	while((c=pReadBuffer[k]) != '\n' && c!=EOF && k++ <= dwBytesRead){
		if(c != ' '){
			aChar[j++] = c;
			if(j==2){
				aChar[j] = '\0';
				GlobalSignatureDatabase.pDatabase[i] = 
OneByteAsciiHexToInt(aChar);
				j = 0;
				i++;
			}
		}
	}
	delete[] pReadBuffer;
	return ReturnCode::VAV_SUCCESS;
}

File * LoadFile(CONST CHAR *FileName){
	File * pFile = new File;
	pFile->Target = string(FileName);
	if((pFile->hFileStream = CreateFileA(pFile-
>Target.c_str(),GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL | 
FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH,NULL))==INVALID_HANDLE_VALUE){
		Log(LogCode::OPEN_FILE_ERR,pFile->Target);
		delete pFile;
		return nullptr;
	}

	DWORD dwBytesRead;

	GetFileSizeEx(pFile->hFileStream,&pFile->FileSize);
	if((pFile->pFile = new BYTE[int(ceil((float)pFile->FileSize.QuadPart / 512)*512)]) 
== NULL){
		Log(LogCode::MEM_ALOC_FILE_ERR,pFile->Target);
		delete pFile;
		return nullptr;
	}

	if(ReadFile(pFile->hFileStream,pFile->pFile,ceil((float)pFile->FileSize.QuadPart / 
512)*512, &dwBytesRead,NULL) == 0){
```

```
            Log(LogCode::READ_FILE_ERR,pFile->Target);
            free(pFile->pFile);
            delete pFile;
            return nullptr;
        }
        return pFile;
}

bool QueueDirectoryFiles(CHAR * Dir){
        HANDLE                          hFindFile;
        WIN32_FIND_DATAA        Win32FindData;
        CHAR                            Directory[MAX_PATH];

        sprintf(Directory,"%s\\*.*",Dir);
        if((hFindFile=FindFirstFileA(Directory,&Win32FindData))==INVALID_HANDLE_VALUE){
                Log(LogCode::DIR_NOT_FOUND,Dir);
                return false;
        }

        do{
                if(strcmp(Win32FindData.cFileName,".") != 0 &&
strcmp(Win32FindData.cFileName,"..") != 0){
                        sprintf(Directory,"%s\\%s",Dir,Win32FindData.cFileName);
                        if(Win32FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
                                QueueDirectoryFiles(Directory);
                        }else{
                                DirectoryQueue.push(string(Directory));
                        }
                }
        }while(FindNextFileA(hFindFile,&Win32FindData));
        FindClose(hFindFile);
        return TRUE;
}

VOID ScanFile(string FileName){
        File * pFile = nullptr;
        if((pFile = LoadFile((const char *)FileName.c_str())) != nullptr){
                if(DetectSignatureInFile(pFile)){
                        GlobalScanResult.Detected++;
                        Log(LogCode::AFFECTED_FILE,FileName);
                }else{
                        Log(LogCode::CLEAN_FILE,FileName);
                }
                GlobalScanResult.TotalScanned++;
                GlobalScanResult.ScanSize += pFile->FileSize.QuadPart;
                pFile->Release();
                delete pFile;

                CHAR ConsoleTitle[MAX_PATH];
                sprintf(ConsoleTitle,"VAV - %d File(s) Scanned : %d File(s) Detected - %d
Error(s)
Occurred\n",GlobalScanResult.TotalScanned,GlobalScanResult.Detected,GlobalScanResult.Erro
rs);
                SetConsoleTitleA(ConsoleTitle);
                /**/
        }else{
                GlobalScanResult.Errors++;
        }
        /**/

}

VOID Scan(string Directory){
        GlobalScanResult.Reset();
        QueueDirectoryFiles((char *)Directory.c_str());
        while(!DirectoryQueue.empty()){
                ScanFile(DirectoryQueue.front());
                DirectoryQueue.pop();
        }
}

INT main(){
        ConsoleInitializer();
        LoadSignature(string("signature.txt"));
        stringstream Args(" ");
        string Cmd,temp,Line,Directory("C:\\");
        cout << "For more information, type 'help'." << endl;
        do {
                cout << ">> ";
                getline(cin,Line);
                Args.clear();
                Args << Line;
                Cmd.erase();
                Args >> Cmd;

                if(Cmd == "exit"){
                        break;
                }else if(Cmd == "help"){
                        cout << "exit" << endl;
                        cout << "scan <dir>" << endl;
                }else if(Cmd == "scan"){
                        Args >> temp;
                        if(temp != "-"){
                                Directory = temp;
                        }
                        Scan(Directory);
                        Log(LogCode::PRINT_FINAL_RESULTS);
                }
        }while(true);
        GlobalSignatureDatabase.Release();
        return EXIT_SUCCESS;
}
```

WinC.c

Sequential implemented in pure C but using Win32 APIs

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <math.h>
#include <Windows.h>
#include <time.h>

#define WRITE_LOG_FILE      FALSE
#define FLUSH_LOG_FILE      FALSE
#define NUM_COUNTERS 1

VOID Log(CHAR * Message, CONST CHAR * Argument);


typedef enum ErrorCode { VAV_FALUIRE, VAV_SUCCESS } RET_CODE;
typedef enum LogCode { INIT_LOG_FILE, END_LOG_FILE,
PRINT_FINAL_RESULTS,READ_FILE_ERR,DIR_NOT_FOUND,MEM_ALOC_FILE_ERR,OPEN_SIGN_FILE_ERR,MEM_
ALOC_SIGN_FILE_ERR,OPEN_FILE_ERR,AFFECTED_FILE,CLEAN_FILE } LOG_CODE;


typedef struct File {
       BYTE *pFile;
       LARGE_INTEGER FileSize;
       CHAR Target[MAX_PATH];
       HANDLE hFileStream;
} *VAV_File;

typedef struct SignatureDatabase {
       BYTE *pDatabase;
       DWORD DatabaseSize;
       CHAR Target[MAX_PATH];
       HANDLE hFile;
} *VAV_DB;

typedef struct ScanResult {
       SIZE_T Detected;
       SIZE_T TotalScanned;
       SIZE_T Errors;
} VAV_Result;

typedef struct Counter{
       clock_t       Clock;
       ULONG  ScanSize;
} VAV_Counter;

// Global Variables

FILE   *pLogFile;
VAV_Result    ScanResult;
VAV_Counter Counter[NUM_COUNTERS];

VOID ResetResults(VOID){
       ScanResult.Detected = 0;
       ScanResult.Errors = 0;
       ScanResult.TotalScanned = 0;
}

VOID AddSize(INT CounterId,ULONG Size){
       Counter[CounterId].ScanSize += Size;
}
```

```
DOUBLE ElapsedTime(INT CounterId){
	return ((DOUBLE)Counter[CounterId].Clock)/CLOCKS_PER_SEC;
}

DOUBLE TotalSizeInMB(INT CounterId){
	return ((DOUBLE)Counter[CounterId].ScanSize)/(1024*1024);
}

DOUBLE AverageSizePerTime(INT CounterId){
	return TotalSizeInMB(CounterId) / ElapsedTime(CounterId);
}

VOID SetStopWatch(INT CounterId){
	Counter[CounterId].Clock = clock();
}

VOID HoldStopWatch(INT CounterId){
	Counter[CounterId].Clock = clock() - Counter[CounterId].Clock;
}

VOID ResetStopWatch(INT CounterId){
	Counter[CounterId].Clock = 0;
}

VOID ResetSizeCounter(INT CounterId){
	Counter[CounterId].ScanSize = 0;
}

VOID ResetCounter(INT CounterId){
	Counter[CounterId].Clock = 0;
	Counter[CounterId].ScanSize = 0;
}

//	Converting a One Byte Hex String To Its Equivalent Integer Value.
INT OneByteAsciiHexToInt(CHAR *String){
	INT i=0,Value=0;
	while(String[i] != '\0'){
		if(String[i] >= 'A' && String[i] <= 'F'){
			Value += (String[i]-'A' + 10)*(pow((DOUBLE)16,(INT)1-i));
		}else if(String[i] >= 'a' && String[i] <= 'f'){
			Value += (String[i]-'a' + 10)*(pow((DOUBLE)16,(INT)1-i));
		}else{
			Value += (String[i]-'0')*(pow((DOUBLE)16,(INT)1-i));
		}
		i++;
	}
	return Value;
}

//	String Matching Algorithm.
BOOL DetectSignatureInFile(VAV_File File,VAV_DB Signature){
	UINT i=0,j=0,k=0;
	for(i=0;i<File->FileSize.QuadPart;i++){
		if(File->pFile[i] == Signature->pDatabase[0]){
			for(j=1,k=i+1;j < Signature->DatabaseSize && k < File-
>FileSize.QuadPart;j++,k++){
				if(Signature->pDatabase[j] != File->pFile[k])
```

```
                                        break;
                                }
                                if(j == Signature->DatabaseSize){
                                        return TRUE;
                                }
                        }
                }
                return FALSE;
}

//      Obtaining The Signature File.
RET_CODE LoadSignature(CONST CHAR * SignatureFileName, VAV_DB *pSignature){
        INT i=0,j=0,c='\0',k=0;
        CHAR aChar[3];
        CHAR *pReadBuffer;
        DWORD dwBytesRead;
        LARGE_INTEGER FileSize;
        *pSignature = (VAV_DB) malloc(sizeof(struct SignatureDatabase));
        strcpy((*pSignature)->Target,SignatureFileName);
        if(((*pSignature)->hFile =
CreateFileA(SignatureFileName,GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIB
UTE_NORMAL | FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH ,NULL)) ==
INVALID_HANDLE_VALUE){
                Log(OPEN_SIGN_FILE_ERR,NULL);
                system("PAUSE");
                exit(EXIT_FAILURE);
        }
        GetFileSizeEx((*pSignature)->hFile,&FileSize);
        (*pSignature)->DatabaseSize = (FileSize.QuadPart+1)/3;
        if(((*pSignature)->pDatabase = (BYTE *)
malloc(sizeof(BYTE)*(ceil((float)FileSize.QuadPart / 512)*512)))==NULL){
                Log(MEM_ALOC_SIGN_FILE_ERR,NULL);
                system("PAUSE");
                exit(EXIT_FAILURE);
        }
        pReadBuffer = (CHAR *) malloc(sizeof(CHAR)*FileSize.QuadPart);
        ReadFile((*pSignature)->hFile,pReadBuffer,ceil((float)FileSize.QuadPart /
512)*512, &dwBytesRead,NULL);
        while((c=pReadBuffer[k]) != '\n' && c!=EOF && k++ <= dwBytesRead){
                if(c != ' '){
                        aChar[j++] = c;
                        if(j==2){
                                aChar[j] = '\0';
                                (*pSignature)->pDatabase[i] = OneByteAsciiHexToInt(aChar);
                                j = 0;
                                i++;
                        }
                }
        }
        free(pReadBuffer);
        return VAV_SUCCESS;
}

RET_CODE GetFile(CONST CHAR *FileName, VAV_File *pFile){
        *pFile = (VAV_File) malloc(sizeof(struct File));
        if(((*pFile)->hFileStream =
CreateFileA(FileName,GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMA
L | FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH,NULL))==INVALID_HANDLE_VALUE){
```

```
              Log(OPEN_FILE_ERR,FileName);
              free(*pFile);
              *pFile = NULL;
              return VAV_FALUIRE;
       }
       strcpy((*pFile)->Target,FileName);
       return VAV_SUCCESS;
}


DWORD LoadFile(VAV_File File){
       DWORD dwBytesRead;
       GetFileSizeEx(File->hFileStream,&File->FileSize);
       if((File->pFile = (BYTE *) malloc(sizeof(BYTE) * (ceil((float)File-
>FileSize.QuadPart / 512)*512))) == NULL){
              Log(MEM_ALOC_FILE_ERR,File->Target);
              return VAV_FALUIRE;
       }

       if(ReadFile(File->hFileStream,File->pFile,ceil((float)File->FileSize.QuadPart /
512)*512, &dwBytesRead,NULL) == 0){
              Log(READ_FILE_ERR,File->Target);
              free(File->pFile);
              return VAV_FALUIRE;
       }
       return VAV_SUCCESS;
}

RET_CODE ReleaseFile(VAV_File *pFile){
       free((*pFile)->pFile);
       CloseHandle((*pFile)->hFileStream);
       free((*pFile));
       *pFile = NULL;
       return VAV_SUCCESS;
}

RET_CODE ReleaseDatabase(VAV_DB *pDatabase){
       free((*pDatabase)->pDatabase);
       CloseHandle((*pDatabase)->hFile);
       free((*pDatabase));
       *pDatabase = NULL;
       return VAV_SUCCESS;
}

BOOL ScanDirectoryFiles(CONST CHAR *Dir,VAV_DB Signature,INT CounterId){
       HANDLE                     hFindFile;
       WIN32_FIND_DATAA    Win32FindData;
       CHAR                       Directory[MAX_PATH];
       VAV_File                   File = NULL;
       CHAR                       ConsoleTitle[MAX_PATH];
       sprintf(Directory,"%s\\*.*",Dir);
       if((hFindFile=FindFirstFileA(Directory,&Win32FindData))==INVALID_HANDLE_VALUE){
              Log(DIR_NOT_FOUND,Dir);
              return FALSE;
       }
       do{
              if(strcmp(Win32FindData.cFileName,".") != 0 &&
strcmp(Win32FindData.cFileName,"..") != 0){
```

```
                        sprintf(Directory,"%s\\%s",Dir,Win32FindData.cFileName);
                        if(Win32FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
                                ScanDirectoryFiles(Directory,Signature,CounterId);
                        }else{
                                if(GetFile(Directory,&File) != VAV_FALUIRE)
                                        if(LoadFile(File) == VAV_FALUIRE)
                                                ReleaseFile(&File);
                                if(File != NULL){
                                        if(DetectSignatureInFile(File,Signature)==TRUE){
                                                ScanResult.Detected++;
                                                Log(AFFECTED_FILE,Directory);
                                        }else{
                                                Log(CLEAN_FILE,Directory);
                                        }
                                        ScanResult.TotalScanned++;
                                        AddSize(CounterId,File->FileSize.QuadPart);
                                        ReleaseFile(&File);
                                        sprintf(ConsoleTitle,"VAV - %d File(s) Scanned : %d
File(s) Detected - %d Error(s)
Occurred\n",ScanResult.TotalScanned,ScanResult.Detected,ScanResult.Errors);
                                        SetConsoleTitleA(ConsoleTitle);
                                }else{
                                        ScanResult.Errors++;
                                }
                        }
                }
        }while(FindNextFileA(hFindFile,&Win32FindData));
        FindClose(hFindFile);
        return TRUE;
}

INT GetLine(CHAR Line[],INT MaxLen){
        UINT i = 0;
        INT     c = '\0';
        while((c=getchar())!='\n' && i < MaxLen)
                Line[i++] = c;
        Line[i] = '\0';
        return i;
}

VOID ConsoleInitializer(VOID){
        SetConsoleTitle(L"Veronica Antivirus");
        printf(" ------------------------------------------------------------- \n");
        printf("|                       Veronica Antivirus                     |\n");
        printf("|                   Written By Ahmad Siavashi                  |\n");
        printf("|                  Email : a.siavosh@yahoo.com                 |\n");
        printf("|                       Shiraz  University                     |\n");
        printf("|                          Spring 2012                         |\n");
        printf(" ------------------------------------------------------------- \n");
}

VOID Log(LOG_CODE Code, CONST CHAR * Argument){
        switch(Code){
        case INIT_LOG_FILE:
                if(WRITE_LOG_FILE)pLogFile = fopen("log.txt","w");
                break;
        case PRINT_FINAL_RESULTS:
                printf(" -----------------------------------------------------------------
\n");
                printf("|               # Scanned  Files   : %-10d File(s)
|\n",ScanResult.TotalScanned);
                printf("|               # Detected Files   : %-10d File(s)
|\n",ScanResult.Detected);
                printf("|               # Occured Errors   : %-10d File(s)
|\n",ScanResult.Errors);
                printf("|               # Elapsed Time(%d)  : %-10lf Second(s)
|\n",(INT)Argument,ElapsedTime((INT)Argument));
                printf("|               # Scan Size(%d)     : %-10lf MB
|\n",(INT)Argument,TotalSizeInMB((INT)Argument));
                printf("|               # Size/Time(%d)     : %-10lf MB/S
|\n",(INT)Argument,AverageSizePerTime((INT)Argument));
                printf(" -----------------------------------------------------------------
\n");
                if(WRITE_LOG_FILE){
                        fprintf(pLogFile,"[-] %d File(s) Scanned : %d File(s) Detected, %d
Error(s) Occurred.\n",ScanResult.TotalScanned,ScanResult.Detected,ScanResult.Errors);
                        fprintf(pLogFile,"[-] Elapsed Time : %lf Second(s)
\n",ElapsedTime((INT)Argument));
                }
                break;
        case READ_FILE_ERR:
                printf("[-] Error While Reading File : \'%s\'\n",strrchr(Argument,'\\')+1);
                if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] Error While Reading File :
\'%s\'\n",Argument);
                break;
        case DIR_NOT_FOUND:
                printf("[-] Directory Not Found : \'%s\'\n",Argument);
                break;
        case MEM_ALOC_FILE_ERR:
                printf("[-] Error While Allocating Memory For File :
\'%s\'\n",strrchr(Argument,'\\')+1);
                if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] Error While Allocating Memory For
File : \'%s\'\n", Argument);
                break;
        case OPEN_SIGN_FILE_ERR:
                printf("[-] Error While Opening The Signature File.\n");
                break;
        case MEM_ALOC_SIGN_FILE_ERR:
                printf("[-] Error While Allocating Memory For The Signature File.\n");
                break;
        case OPEN_FILE_ERR:
                printf("[-] Error While Opening File : \'%s\'\n",strrchr(Argument,'\\')+1);
                if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] Error While Opening File :
\'%s\'\n",Argument);
                break;
        case AFFECTED_FILE:
                printf("[-] AFFECTED  : %s\n",strrchr(Argument,'\\')+1);
                if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] AFFECTED  : %s\n",Argument);
                break;
        case CLEAN_FILE:
                printf("[-] CLEAN     : %s\n",strrchr(Argument,'\\')+1);
                if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] CLEAN     : %s\n",Argument);
                break;
        case END_LOG_FILE:
                fflush(pLogFile);
```

```c
			fclose(pLogFile);
			break;
		}
		if(FLUSH_LOG_FILE) fflush(pLogFile);
}

INT main(){
		VAV_DB Signature = NULL;
		CHAR   Path[MAX_PATH] = "";
		ConsoleInitializer();
GET_DIR:
		printf(">>> Directory Address : ");
		if(!GetLine(Path,MAX_PATH))
			goto GET_DIR;
		LoadSignature("signature.txt",&Signature);
		Log(INIT_LOG_FILE,NULL);
START_SCAN:
		ResetResults();
		ResetCounter(0);
		SetStopWatch(0);
		ScanDirectoryFiles(Path,Signature,0);
		HoldStopWatch(0);
		Log(PRINT_FINAL_RESULTS,0);
AGAIN:
		{
			CHAR TemporaryPath[MAX_PATH];
			printf(">>> Directory Address [ '-' For Previous Directory ] : ");
			if(!GetLine(TemporaryPath,MAX_PATH))
				goto AGAIN;
			if(!strcmp(TemporaryPath,"-"))
				goto START_SCAN;
			strcpy(Path,TemporaryPath);
			goto START_SCAN;
		}
END:
		ReleaseDatabase(&Signature);
		Log(END_LOG_FILE,NULL);
		return EXIT_SUCCESS;
}
```

PlainC.c

Sequential Implementation in Pure C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <Windows.h>
#include <time.h>
```

```
#define WRITE_LOG_FILE      TRUE
#define FLUSH_LOG_FILE      TRUE
#define NUM_COUNTERS 1

VOID Log(CHAR * Message, CONST CHAR * Argument);


typedef enum ErrorCode { VAV_SUCCESS, VAV_FALUIRE } RET_CODE;
typedef enum LogCode { INIT_LOG_FILE, END_LOG_FILE,
PRINT_FINAL_RESULTS,READ_FILE_ERR,DIR_NOT_FOUND,MEM_ALOC_FILE_ERR,OPEN_SIGN_FILE_ERR,MEM_
ALOC_SIGN_FILE_ERR,OPEN_FILE_ERR,AFFECTED_FILE,CLEAN_FILE } LOG_CODE;


typedef struct File {
       BYTE *pFile;
       SIZE_T FileSize;
       CHAR Target[MAX_PATH];
       FILE *pFileStream;
} *VAV_File;

typedef struct SignatureDatabase {
       BYTE *pDatabase;
       SIZE_T DatabaseSize;
       CHAR Target[MAX_PATH];
       FILE *pFileStream;
} *VAV_DB;

typedef struct ScanResult {
       SIZE_T Detected;
       SIZE_T TotalScanned;
       SIZE_T Errors;
} VAV_Result;

typedef struct Counter{
       clock_t       Clock;
       ULONG  ScanSize;
} VAV_Counter;

// Global Variables

FILE   *pLogFile;
VAV_Result    ScanResult;
VAV_Counter Counter[NUM_COUNTERS];

VOID ResetResults(VOID){
       ScanResult.Detected = 0;
       ScanResult.Errors = 0;
       ScanResult.TotalScanned = 0;
}

VOID AddSize(INT CounterId,ULONG Size){
       Counter[CounterId].ScanSize += Size;
}

DOUBLE ElapsedTime(INT CounterId){
       return ((DOUBLE)Counter[CounterId].Clock)/CLOCKS_PER_SEC;
}
```

```
DOUBLE TotalSizeInMB(INT CounterId){
	return ((DOUBLE)Counter[CounterId].ScanSize)/(1024*1024);
}

DOUBLE AverageSizePerTime(INT CounterId){
	return TotalSizeInMB(CounterId) / ElapsedTime(CounterId);
}

VOID SetStopWatch(INT CounterId){
	Counter[CounterId].Clock = clock();
}

VOID HoldStopWatch(INT CounterId){
	Counter[CounterId].Clock = clock() - Counter[CounterId].Clock;
}

VOID ResetStopWatch(INT CounterId){
	Counter[CounterId].Clock = 0;
}

VOID ResetSizeCounter(INT CounterId){
	Counter[CounterId].ScanSize = 0;
}

VOID ResetCounter(INT CounterId){
	Counter[CounterId].Clock = 0;
	Counter[CounterId].ScanSize = 0;
}

//	Converting a One Byte Hex String To Its Equivalent Integer Value.
INT OneByteAsciiHexToInt(CHAR *String){
	INT i=0,Value=0;
	while(String[i] != '\0'){
		if(String[i] >= 'A' && String[i] <= 'F'){
			Value += (String[i]-'A' + 10)*(pow((DOUBLE)16,(INT)1-i));
		}else if(String[i] >= 'a' && String[i] <= 'f'){
			Value += (String[i]-'a' + 10)*(pow((DOUBLE)16,(INT)1-i));
		}else{
			Value += (String[i]-'0')*(pow((DOUBLE)16,(INT)1-i));
		}
		i++;
	}
	return Value;
}

//	String Matching Algorithm.
BOOL DetectSignatureInFile(VAV_File File,VAV_DB Signature){
	UINT i=0,j=0,k=0;
	for(i=0;i<File->FileSize;i++){
		if(File->pFile[i] == Signature->pDatabase[0]){
			for(j=1,k=i+1;j < Signature->DatabaseSize && k < File-
>FileSize;j++,k++){
				if(Signature->pDatabase[j] != File->pFile[k])
					break;
			}
			if(j == Signature->DatabaseSize){
				return TRUE;
			}
			}
		}
		return FALSE;
}

//	Obtaining The Signature File.
RET_CODE LoadSignature(CONST CHAR * SignatureFileName, VAV_DB *pSignature){
		INT i=0,j=0,c='\0';
		CHAR aChar[3];
		*pSignature = (VAV_DB) malloc(sizeof(struct SignatureDatabase));
		strcpy((*pSignature)->Target,SignatureFileName);
		if(((*pSignature)->pFileStream = fopen(SignatureFileName,"r"))==NULL){
			Log(OPEN_SIGN_FILE_ERR,NULL);
			system("PAUSE");
			exit(EXIT_FAILURE);
		}
		fseek((*pSignature)->pFileStream,0,SEEK_END);
		(*pSignature)->DatabaseSize = (ftell((*pSignature)->pFileStream)+1)/3;
		rewind((*pSignature)->pFileStream);
		if(((*pSignature)->pDatabase = (BYTE *) malloc(sizeof(BYTE)*(*pSignature)-
>DatabaseSize))==NULL){
			Log(MEM_ALOC_SIGN_FILE_ERR,NULL);
			system("PAUSE");
			exit(EXIT_FAILURE);
		}
		rewind((*pSignature)->pFileStream);
		while((c=fgetc((*pSignature)->pFileStream)) != '\n' && c!=EOF){
			if(c != ' '){
				aChar[j++] = c;
				if(j==2){
					aChar[j] = '\0';
					(*pSignature)->pDatabase[i] = OneByteAsciiHexToInt(aChar);
					j = 0;
					i++;
				}
			}
		}
		return VAV_SUCCESS;
}

RET_CODE GetFile(CONST CHAR *FileName, VAV_File *pFile){
		*pFile = (VAV_File) malloc(sizeof(struct File));
		if(((*pFile)->pFileStream = fopen(FileName,"rb"))==NULL){
			Log(OPEN_FILE_ERR,FileName);
			free(*pFile);
			*pFile = NULL;
			return VAV_FALUIRE;
		}
		strcpy((*pFile)->Target,FileName);
		return VAV_SUCCESS;
}


RET_CODE LoadFile(VAV_File File){
		fseek(File->pFileStream, 0, SEEK_END);
		File->FileSize = ftell(File->pFileStream);
		rewind(File->pFileStream);
```

```
        if((File->pFile = (BYTE *) malloc(sizeof(BYTE) * File->FileSize )) == NULL){
                Log(MEM_ALOC_FILE_ERR,File->Target);
                free(File->pFile);
                return VAV_FALUIRE;
        }

        if(fread(File->pFile,File->FileSize,1,File->pFileStream) == 0){
                Log(READ_FILE_ERR,File->Target);

                return VAV_FALUIRE;
        }
        return VAV_SUCCESS;
}

RET_CODE ReleaseFile(VAV_File *pFile){
        free((*pFile)->pFile);
        fclose((*pFile)->pFileStream);
        free((*pFile));
        *pFile = NULL;
        return VAV_SUCCESS;
}

RET_CODE ReleaseDatabase(VAV_DB *pDatabase){
        free((*pDatabase)->pDatabase);
        fclose((*pDatabase)->pFileStream);
        free((*pDatabase));
        *pDatabase = NULL;
        return VAV_SUCCESS;
}

BOOL ScanDirectoryFiles(CONST CHAR *Dir,VAV_DB Signature,INT CounterId){
        HANDLE                          hFindFile;
        WIN32_FIND_DATAA        Win32FindData;
        CHAR                            Directory[MAX_PATH];
        VAV_File                        File = NULL;
        CHAR                            ConsoleTitle[MAX_PATH];
        sprintf(Directory,"%s\\*.*",Dir);
        if((hFindFile=FindFirstFileA(Directory,&Win32FindData))==INVALID_HANDLE_VALUE){
                Log(DIR_NOT_FOUND,Dir);
                return FALSE;
        }
        do{
                if(strcmp(Win32FindData.cFileName,".") != 0 &&
strcmp(Win32FindData.cFileName,"..") != 0){
                        sprintf(Directory,"%s\\%s",Dir,Win32FindData.cFileName);
                        if(Win32FindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
                                ScanDirectoryFiles(Directory,Signature,CounterId);
                        }else{
                                if(GetFile(Directory,&File) == VAV_SUCCESS)
                                        if(LoadFile(File)!=VAV_SUCCESS)
                                                ReleaseFile(&File);
                                if(File != NULL){
                                        if(DetectSignatureInFile(File,Signature)==TRUE){
                                                ScanResult.Detected++;
                                                Log(AFFECTED_FILE,Directory);
                                        }else{
                                                Log(CLEAN_FILE,Directory);
                                        }
```

```
                                    ScanResult.TotalScanned++;
                                    AddSize(CounterId,File->FileSize);
                                    ReleaseFile(&File);
                                    sprintf(ConsoleTitle,"VAV - %d File(s) Scanned : %d
File(s) Detected - %d Error(s)
Occurred\n",ScanResult.TotalScanned,ScanResult.Detected,ScanResult.Errors);
                                    SetConsoleTitleA(ConsoleTitle);
                             }else{
                                    ScanResult.Errors++;
                             }
                      }
               }
       }while(FindNextFileA(hFindFile,&Win32FindData));
       FindClose(hFindFile);
       return TRUE;
}

INT GetLine(CHAR Line[],INT MaxLen){
       UINT i = 0;
       INT    c = '\0';
       while((c=getchar())!='\n' && i < MaxLen)
              Line[i++] = c;
       Line[i] = '\0';
       return i;
}

VOID ConsoleInitializer(VOID){
       SetConsoleTitle(L"Veronica Antivirus");
       printf(" -------------------------------------------------------------- \n");
       printf("|                      Veronica Antivirus                      |\n");
       printf("|                  Written By Ahmad Siavashi                   |\n");
       printf("|                 Email : a.siavosh@yahoo.com                  |\n");
       printf("|                      Shiraz  University                      |\n");
       printf("|                         Spring 2012                          |\n");
       printf(" -------------------------------------------------------------- \n");
}

VOID Log(LOG_CODE Code, CONST CHAR * Argument){
       switch(Code){
       case INIT_LOG_FILE:
              if(WRITE_LOG_FILE)pLogFile = fopen("log.txt","w");
              break;
       case PRINT_FINAL_RESULTS:
              printf(" -----------------------------------------------------------------
\n");
              printf("|              # Scanned  Files   : %-10d File(s)
|\n",ScanResult.TotalScanned);
              printf("|              # Detected Files   : %-10d File(s)
|\n",ScanResult.Detected);
              printf("|              # Occured Errors   : %-10d File(s)
|\n",ScanResult.Errors);
              printf("|              # Elapsed Time(%d)  : %-10lf Second(s)
|\n",(INT)Argument,ElapsedTime((INT)Argument));
              printf("|              # Scan Size(%d)     : %-10lf MB
|\n",(INT)Argument,TotalSizeInMB((INT)Argument));
              printf("|              # Size/Time(%d)     : %-10lf MB/S
|\n",(INT)Argument,AverageSizePerTime((INT)Argument));
```

```
              printf(" ---------------------------------------------------------------
\n");
              if(WRITE_LOG_FILE){
                     fprintf(pLogFile,"[-] %d File(s) Scanned : %d File(s) Detected, %d
Error(s) Occurred.\n",ScanResult.TotalScanned,ScanResult.Detected,ScanResult.Errors);
                     fprintf(pLogFile,"[-] Elapsed Time : %lf Second(s)
\n",ElapsedTime((INT)Argument));
              }
              break;
       case READ_FILE_ERR:
              printf("[-] Error While Reading File : \'%s\'\n",strrchr(Argument,'\\')+1);
              if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] Error While Reading File :
\'%s\'\n",Argument);
              break;
       case DIR_NOT_FOUND:
              printf("[-] Directory Not Found : \'%s\'\n",Argument);
              break;
       case MEM_ALOC_FILE_ERR:
              printf("[-] Error While Allocating Memory For File :
\'%s\'\n",strrchr(Argument,'\\')+1);
              if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] Error While Allocating Memory For
File : \'%s\'\n", Argument);
              break;
       case OPEN_SIGN_FILE_ERR:
              printf("[-] Error While Opening The Signature File.\n");
              break;
       case MEM_ALOC_SIGN_FILE_ERR:
              printf("[-] Error While Allocating Memory For The Signature File.\n");
              break;
       case OPEN_FILE_ERR:
              printf("[-] Error While Opening File : \'%s\'\n",strrchr(Argument,'\\')+1);
              if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] Error While Opening File :
\'%s\'\n",Argument);
              break;
       case AFFECTED_FILE:
              printf("[-] AFFECTED  : %s\n",strrchr(Argument,'\\')+1);
              if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] AFFECTED  : %s\n",Argument);
              break;
       case CLEAN_FILE:
              printf("[-] CLEAN     : %s\n",strrchr(Argument,'\\')+1);
              if(WRITE_LOG_FILE) fprintf(pLogFile,"[-] CLEAN     : %s\n",Argument);
              break;
       case END_LOG_FILE:
              fflush(pLogFile);
              fclose(pLogFile);
              break;
       }
       if(FLUSH_LOG_FILE) fflush(pLogFile);
}

INT main(){
       VAV_DB Signature = NULL;
       CHAR   Path[MAX_PATH] = "";
       ConsoleInitializer();
GET_DIR:
       printf(">>> Directory Address : ");
       if(!GetLine(Path,MAX_PATH))
              goto GET_DIR;
```

```
	LoadSignature("signature.txt",&Signature);
	Log(INIT_LOG_FILE,NULL);
START_SCAN:
	ResetResults();
	ResetCounter(0);
	SetStopWatch(0);
	ScanDirectoryFiles(Path,Signature,0);
	HoldStopWatch(0);
	Log(PRINT_FINAL_RESULTS,0);
AGAIN:
	{
		CHAR TemporaryPath[MAX_PATH];
		printf(">>> Directory Address [ '-' For Previous Directory ] : ");
		if(!GetLine(TemporaryPath,MAX_PATH))
			goto AGAIN;
		if(!strcmp(TemporaryPath,"-"))
			goto START_SCAN;
		strcpy(Path,TemporaryPath);
		goto START_SCAN;
	}
END:
	ReleaseDatabase(&Signature);
	Log(END_LOG_FILE,NULL);
	return EXIT_SUCCESS;
}
```