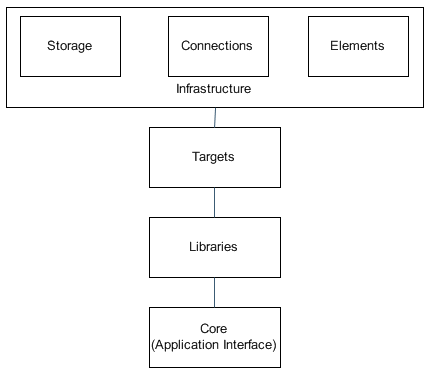
SRDF Reference Manual

In this Manual … we will describe each class in the Project … its functions, variables and example on it.

# The Design:



## Infrastructure:

This includes the essential elements of any development framework and it’s not related to security like: string, hash, list, serializer, database, registry manipulation, sockets and so on.

We decided to create this part rather than depending on any development framework to make this framework independent from any other development frameworks and to be portable on any development framework

## Targets:

This is the beginning of the SRDF. This part is simply the Target from your security tool. What do you want to secure or secure from. And it includes Files (PE Files and others), Processes and Packets.

## Libraries:

That’s the security tools that the SRDF support. And it’s divided into two namespaces: malware and network

Malware includes the assemblers and disassemblers, emulator, debugger, API Hooker, Yara Scanner (wildcard scanner) file recursive scanner and other tools

Network includes User-Mode capturing and Firewall

## Core (The Application Interface):

The Core includes the Logging system and the back-end Database.

And also, it’s the Application Interface. Like cConsoleApp … and you can inherit from it to create your own User-Interface.

We wish this part to be expanded to include more user interfaces and management systems

# Infrastructure:

## Elements:

### cHash:

**Discription:**

This class is created for saving an array contains 2 strings: key and value. The relation between the key and the value is one-to-many. So, each key could have more than one value

1. **GetNumberOfItems():** Get the number of items in the hash
2. **GetNumberOfItems(key):** Get the number of values for this key
3. **AddItem(…):** Add new item to the array (key and value)
4. **Operator[key]:** gets the 1st value for this key
5. **Operator[id]:** gets the value of the item number equal to id
6. **GetKey(id):** gets the key of the item number equal to id
7. **GetValue(key,id):** gets the value number (id) for this key
8. **RemoveItem(id):** remove an item with index equal to id
9. **RemoveItem(key,id):** remove item with this key and the value number "id"
10. **IsFound(key):** check if this key is found
11. **Serialize():** Serialize this hash into an XML string describe the content of this hash
12. **Deserialize(XML):** fill the hash with the serialized Hash is XML

**Example:**

cHash hash;

hash.AddItem("OllyDbg","Debugger for win32 apps");

hash.AddItem("IDA Pro","Hex Rays IDA Pro");

hash.AddItem("WireShark","Packet Capture Tool");

if (hash.IsFound("IDA Pro"))cout << hash["IDA Pro"] << "!!\n";

cout << (char\*)hash["OllyDbg"] << " " << hash["IDA Pro"] << " " << hash["WireShark"] << "\n";

cout << hash.Serialize() << "\n";

**The Output is:**

Debugger for win32 apps!!

Debugger for win32 apps Hex Rays IDA Pro Packet Capture Tool

<OllyDbg>Debugger for win32 apps</OllyDbg>

<IDA Pro>Hex Rays IDA Pro</IDA Pro>

<WireShark>Packet Capture Tool</WireShark>

true

### cList:

**Discription:**

This class is an array of buffer with fixed size. This buffer could be a struct in C.

1. **cList(size):** creates a list of buffer with size = size argument
2. **AddItem(buffer):** this function copies this buffer into the list.
3. **GetNumberOfItems():** get the number of items inside the list.
4. **GetItem(id):** get the item with index = id
5. **Operator[]:** get the item with index = id
6. **Serialize():** serialize the buffers into XML using Base64 Encoding. (Deserializing could has a bad effect on any pointer inside the buffers)
7. **Deserialize(XML):** deserialize XML into the List

**Example:**

cList\* CmdList = new cList(sizeof(CONSOLE\_COMMAND));

CONSOLE\_COMMAND cmd; //Any struct … but here’s an example

cmd.Name = new cString(Name);

cmd.Description = new cString(Description);

cmd.Format = new cString(Format);

cmd.nArgs = nArgs;

cmd.CommandFunc = CommandFunc;

CmdList->AddItem((char\*)&cmd);

CONSOLE\_COMMAND\* cmdstruct = (CONSOLE\_COMMAND\*)CmdList->GetItem(i);

### Serializer:

**Discription:**

It's an abstract class and it's used to serialize an object into XML:

1. **Serialize():** returns an XML String of the serialized object
2. **Deserialize(XML):** deserialize the XML String into an object
3. **SetSerialize(XMLHash):** this is a virtual function (should be overridden) and it's used to serialize the variables of the class inside the XMLHash and will be called from inside Serialize() function… and you can add your variables inside the hash using these functions:
   1. **AddText():** insert a string variable inside the hash
   2. **AddXML():** insert a serialized object inside the hash
   3. **AddBinary():** insert a binary buffer inside the hash (will be encoded with Base64 Encoding)
4. **GetSerialize(XMLHash):** it's the opposite of the setserialize() function and it gets the items (Text,XML, Binary) from inside the XMLHash

**Example:**

class XMLExample : public cSerializer

{

public:

int Number;

cString Text;

cHash Hash;

virtual void SetSerialize(cXMLHash& XMLParams);

virtual void GetSerialize(cXMLHash& XMLParams);

XMLExample();

~XMLExample(){};

//virtual void SetSerialize(cHash &XMLParams);

};

XMLExample::XMLExample()

{

Number = 5;

Text = "Could Serialize Text";

Hash.AddItem("OllyDbg","Debugger for win32 apps");

Hash.AddItem("IDA Pro","Hex Rays IDA Pro");

Hash.AddItem("WireShark","Packet Capture Tool");

}

void XMLExample::SetSerialize(cXMLHash& XMLParams)

{

XMLParams.AddText("Number",Number); //Will be converted automatically to string

XMLParams.AddText("Text",Text);

XMLParams.AddItem("HashObject",Hash.Serialize());

}

void XMLExample::GetSerialize(cXMLHash& XMLParams)

{

Number = atoi(XMLParams["Number"]);

Text = XMLParams["Text"];

Hash.Deserialize(XMLParams["HashObject"]);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

XMLExample\* X = new XMLExample();

cString XML = X->Serialize();

cout << XML << "\n";

X = new XMLExample();

X->Deserialize(XML);

cout << "X->Number = " << X->Number << "\n";

cout << "X->Text = " << X->Text << "\n";

cout << "X->Hash[\"OllyDbg\"] = " << X->Hash["OllyDbg"] << "\n";

return 0;

}

**Output:**

<Number>5</Number>

<Text>Could Serialize Text</Text>

<HashObject>

<OllyDbg>Debugger for win32 apps</OllyDbg>

<IDA Pro>Hex Rays IDA Pro</IDA Pro>

<WireShark>Packet Capture Tool</WireShark></HashObject>

Debugger for win32 apps

The New Value : Debugger for win32 apps

The New Value : Hex Rays IDA Pro

The New Value : Packet Capture Tool

X->Number = 5

X->Text = Could Serialize Text

X->Hash["OllyDbg"] = Debugger for win32 apps

### cMD5String:

**Discription:**

This class is inherited from cEnxcryptedString and it's used to encrypt a buffer/text into MD5 Hash

1. **cMD5String(string):** this function creates an MD5 String Object of the string (encrypting the string into MD5 and creates an object of this MD5String).
2. **cMD5String(buff,size):** encrypts a buffer and saves it into the object
3. **GetEncrypted():** returns the MD5 Hash.
4. **SetEncrypted(MD5):** Takes an already encrypted string and saves it into the object.
5. **Operator==:** this is used to compare the encrypted string (MD5 hash) with another encrypted string.
6. **Encrypt(buff,size):** this function encrypts a buffer and returns it … and it's a virtual function for the abstract class "cEncryptedString".

**Example:**

cMD5String MD5;

MD5.Encrypt("Hello World !!!", strlen("Hello World !!!"));

cout << (char\*)MD5 << "\n";

### cBase64String:

**Discription:**

This class is used to encode a buffer into Base64 string and it's inherited from "cEncodedString".

1. **cBase64String(string):** this function creates a Base64 String Object of the string (encoding the string into Base64 and creates an object of this Base64String).
2. **Base64String(buff,size):** encodes a buffer and saves it into the object
3. **GetEncoded():** returns the Base64 String.
4. **SetEncoded(Base64):** Takes an already encoded string and saves it into the object.
5. **Operator==:** this is used to compare the encoded string (Base64 string) with another encoded string.
6. **Encode(buff,size):** this function encodes a buffer and returns it … and it's a virtual function for the abstract class "cEncodedString".
7. **Decode(&size):** this function decodes the Base64 string into buffer again and returns the buffer and the size in the first argument

**Example:**

DWORD len;

cBase64String Base64;

Base64.Encode("Hello World !!!",strlen("Hello World !!!"));

cout << (char\*)Base64 << "\n";

cout << (char\*)Base64.Decode(len) << "\n";

### cStoredProcedure:

**Discription:**

it's a descriptive class … this class created to describe a function in a dll (its Author, Description .. and so on) and serialize it into database. And it could call to this function by LoadLibrary/GetProcAddress).

This class is used to save the storedprocedure into the database with information about it … and we will describe how to use it and how to write a dll stores its functions into the database.

1. **cStoredProcedure(…):** this creates the object of StoredProcedure with the information that you give
2. **Run(userdata):** this function load the dll and call to the API giving it the user-data parameter
3. **Serialize():** this function serializes the StoredProcedure to be stored in the Database
4. **Deserialize(XML):** this function deserializes the XML into Object

### cNativeCode:

**Discription:**

This class is created to represent a shellcode or a place-independent code. And it can be serialized into XML.

1. **cNativeCode(…):** this function creates a new object with the information that you give … and the buffer and the size of the shellcode.
2. **SetCode(buff,size):** sets the shellcode
3. **GetCode(&size):** gets the shellcode.
4. **Run(userdata):** runs the Shellcode.
5. **Serialize() and Deserialize(XML):** serializes and deserializes the shellcode from/into XML

## Storage:

### cLog:

**Discription:**

It's a small class and it's used to log an event inside a log file. It logs the name of the application + date&time + the string. It synchronizes the writing on this log using Critical Sections

1. **cLog(filename,logname):** this function opens the file to append or creates a new file.
2. **IsFound():** returns true if the file opened successfully
3. **WriteToLog(str):** write a string to the log (Logname+datetime+string).

### cRegistryKey:

**Discription:**

RegistryKey is a class created for working with registry keys, enumerating its values and subkeys.

1. **cRegistryKey(…):** open the registry key … if Create is true … so it will create the key if not found
2. **operator[entry]:** returns the entry inside the key with name entry.
3. **operator[id]:** returns the entry with id.
4. **GetNumberOfEntries():** Get the number of entries inside this key.
5. **GetNumberOfSubkeys():** Get the number of subkeys inside this key.
6. **IsFound():** returns true if the key is found or created successfully.
7. **cHash Subkeys:** this hash contains the Subkeys that's inside the key
8. **RefreshEntries():** reload again the entries inside the this key.

### cRegistryEntry:

**Discription:**

This class describes a new Value inside a specific key … returns its value and type … and modify it

1. **cRegistryEntry(…):** opens the value from the key
2. **IsFound():** return true if the value opened successfully
3. **GetEntryname():** returns the name of the entry
4. **Operator ==:** compare the value data to a string (used only with REG\_SZ values)
5. **Operator =:** Set value data to a string (used only with REG\_SZ values)
6. **SetValue(buff,len,type):** Set the value data with buffer and size … and with type of the value like (REG\_SZ …)
7. **GetValue(&Size):** returns the data of value (could be used with binary values)

### cSQLiteDatabase:

**Discription:**

This class is inherited from cDatabase Abstract class … and this class is used to work with SQLite Database and it's designed to save the XML Serialized objects. This database uses synchronization to synchronize the writing and reading from/to it.

It creates tables inside SQLite with only 2 fields … the Index Field (auto-numbering) and XML Field … the XML Field is a string field saves the XML String that represents a Serialized Object

1. **cSQLiteDatabase(…):** opens a database or create a new database
2. **GetItems(Table):** Gets the items inside the Table and return it in Hash
3. **GetItem(Table,id):** Gets a specific item with index = id … and return the XML Field.
4. **AddItem(…):** adds a new item in a Table inside the database.
5. **RemoveItem(Table,id):** removes the item with index = id
6. **RemoveItem(Table,XML):** removes the item with XML Field = XML.
7. **CreateTable(TableName):** creates a new Table inside the database.

**Example:**

cSQLiteDatabase sql;

if (sql.OpenDatabase("c:\\Amr2.db"))

{

//CreateTable returns false if already exists

cout << sql.CreateTable("Data") << "\n";

//returns true if successfully added

cout << sql.AddItem("Data","Ali") << "\n";

sql.RemoveItem("Data","Amr");

//return a hash of the data inside the database

cHash\* hash = sql.GetItems("Data");

cout << "Value 0 : " << hash->GetValue(0) << "\n";

}

sql.CloseDatabase();

### cHTTPSocket:

**Discription:**

This Class created for establishing connections to an HTTP Server and download files and pages from it

1. **cHTTPSocket(URL,Port):** this function connects to a server giving the Website name and Port
2. **SendRequest(Request):** this function sends a request like “index.php?x=xxxx”
3. **DownloadFile(Request,Filename):** similar to SendRequest but saves the output file in the Harddisk
4. **Close():** close the connection with the server

**User-Mode/Kernel-Mode Communication:**

**cDriver:**

**Discription:**

This Class created for loading and unloading a device driver and it manages the communication between it and the User-Mode application

1. **cDriver(ServiceName,Filename):** create a new object of Driver (doesn’t load the driver)
2. **LoadDriver():** Loads the Driver
3. **UnloadDriver():** unloads the driver
4. **UserComm[MAX\_DEVICES]:** it’s an Array of “cDevice\*” to communicate to each Device in the Driver separately

### cDevice:

**Discription:**

This class represents a DeviceObject in the Device driver with a symbolic link to communicate with. It could read and write to this DeviceObject or sending fast must reply messages.

1. **cDevice(DeviceObjectSymblicName):** it creates a communication channel between your application and a DeviceObject
2. **RegisterReadFunction(Func):** this function registers a read notification routine to receive messages from the Kernel-Mode and this function should be like this:

void ReadFunc(char msgcode,DWORD status,DWORD size,char\* data);

1. **Write(msgcode, status, data, size of data):** This Function writes to the DeviceObject a message with
   1. **Code:** The code of the message
   2. **Status:** if you need only to send a notification like: code: IO\_DELETE and status: FILE\_DELETED\_SUCCESSFULLY
   3. **Data:** that’s the data you want to send
   4. **Size:** the length of this data
2. **SendFastMsg:** this function sends a message and wait for its reply from the device driver

## Targets:

### cFile:

**Discription:**

This class represents the Memory Mapped Files for reading.

1. **cFile(Filename) or OpenFile(Filename):** Opens a memory mapped file for reading
2. **BaseAddress:** The Address of the file in memory (after being mapped)
3. **FileLength:** The Length of the File
4. **Attributes:** The Attributes of the File that was opened

### cPEFile:

**Discription:**

This Class created to describe a Win32 PE File, parse its header and represent it in a simple shape

1. **cPEFile(Filename):** opens a PE File and parses it
2. **OffsetToRVA(address):** converts an Address in the File to an RVA (Relative Virtual Address) in the memory after the PE File loaded
3. **RVAToOffset(RVA):** converts an RVA to an Address in the PE File
4. **PEHeader :** it’s a pointer to the PE Header inside the file
5. **DataDirectories:** this is a 16-bits value ... each bit represents a DataDirectory … like 1st bit represents the Export Table … if it sets, so there’s an Export Table … and so on.
6. **Section:** This is represents the array of sections inside the PE File and it’s a struct “SECTION\_STRUCT” and includes:
   1. SectionName
   2. VirtualAddress
   3. VirtualSize
   4. PointerToRawData
   5. SizeOfRawData
   6. Characterisics
   7. RealAddr: a pointer to the section in the opened file in memory
7. **ImportTable:** This field represents the Import Table and it contains:
   1. **nDlls:** The Number of DLLs imported by the file
   2. **DLL:** Array of struct “IMPORTTABLE\_DLL” and it represents each Imported DLL inside the File and it contains:
      1. **DLLName:** The DLL Name
      2. **nAPIs:** The Number of Imported APIs from this DLL
      3. **API:** it’s an Array of “IMPORTTABLE\_API” and it represents each API Imported from this DLL and it contains:
         1. **APIName:** The Name of the API
         2. **APIAddressPlace:** The place where the address will be set (used for searching of call DWORD PTR [APIAddressPlace] )

**Example:**

cPEFile\* PEFile = new cPEFile("C:\\01Msgbox.exe");

cout << "Imagebase : " << (int\*)PEFile->Imagebase << "\n";

cout << "DataDirectories : " << (int\*)PEFile->DataDirectories << "\n";

for (int i = 0; i<PEFile->nSections; i++)

{

cout << "Section " << i << " : " << PEFile->Section[i].SectionName << " at VirtualAddress = " << (int\*)PEFile->Section[i].VirtualAddress << "\n";

}

for (DWORD i = 0; i< PEFile->ImportTable.nDLLs;i++)

{

for (DWORD l = 0; l < PEFile->ImportTable.DLL[i].nAPIs; l++)

{

cout << PEFile->ImportTable.DLL[i].DLLName << " : " << PEFile->ImportTable.DLL[i].API[l].APIName << " : " << (int\*)PEFile->ImportTable.DLL[i].API[l].APIAddressPlace << "\n";

}

}

### cProcess:

**Discription:**

This Class describe a running process … it collects the information about its memory, the loaded dlls and the basic information about this Process

1. **cProcess(PID):** opens a process with specific PID and analyzes it
2. **Read(Address,Size):** copies a specific place in the memory of this process in your memory and return the pointer
3. **Allocate(PreferredAddress,Size):** Allocate a place in the memory of this process
4. **Write(StartAddressToWrite,YourBuffer,Size):** Writes in the process memory
5. **DLLInject(Filename):** This function injects a DLL inside the process using CreateRemoteThread and LoadLibrary
6. **CreateThread(Entrypoint,** **addressToParameter):** This function creates a remote thread inside the process
7. **IsFound():** returns true if the process is found

The Analysis Data are:

1. **ImageBase:** The Beginning of the Process Module in Memory
2. **SizeOfImage:** The Size of the Process Module in Memory
3. **ParentID:** The Id of The Process that creates this Process
4. **CommandLine:** The Commandline of this Process
5. **ProcessName, ProcessPath:** The Name and The Path of the process
6. **ModuleList:** it’s a cList and it represents the structure “MODULE\_INFO” which contains:
   1. **moduleName:** The Module or DLL Name
   2. **modulePath:** The Path to this module in the Harddisk
   3. **moduleImagebase:** The Imagebase of the Module
   4. **moduleSizeOfImage:** the size of this module in memory
7. **MemoryMap:** this is a cList of all memory regions inside the process with the protection of this map … and this cList represents a list of “MEMORY\_MAP” structure:
   1. **Address:** The Address of this region
   2. **Size:** the size of the reigion
   3. **Protection:** The protection of this region … read, write or/and executable

**Example:**

cProcess myProc(792);

cout<<"Process: "<< myProc->processName<<endl;

cout<<"Process Parent ID: "<< myProc->ParentID <<endl;

cout<< "Process Command Line: "<< myProc->CommandLine << endl;

cout<<"Process PEB:\t"<< myProc->ppeb<<endl;

cout<<"Process ImageBase:\t"<<hex<< myProc->ImageBase<<endl;

cout<<"Process SizeOfImageBase:\t"<<dec<< myProc->SizeOfImage<<" byte"<<endl;

cout<<"Process Modules\n";

for (int i=0 ; i<(int)( myProc->modulesList.GetNumberOfItems()) ;i++)

{

cout<<"Module "<<((MODULE\_INFO\*)myProc->modulesList.GetItem(i))->moduleName->GetChar() <<" ImageBase: "<<hex<<((MODULE\_INFO\*) myProc->modulesList.GetItem(i))->moduleImageBase<<endl;

}

cout << "Memory Map:\n";

for (int i=0 ; i<(int)( myProc->MemoryMap.GetNumberOfItems()) ;i++)

{

cout<<"Memory Address "<<((MEMORY\_MAP\*)myProc->MemoryMap.GetItem(i))->Address <<" Size: "<<hex<<((MEMORY\_MAP\*) myProc->MemoryMap.GetItem(i))->Size <<endl;

}

# Libraries:

### cProcessScanner:

**Discription:**

This is a scanner which returns all the running process in the memory by name and ProcessId

1. **cProcessScanner(…):** generate a list of running processes in ProcessList
2. **ProcessList:** it’s a hash with Key = processName and Value = id … as the relation is one-to-many … as it could be many processes with the same name

### cRecursiveScanner:

**Discription:**

This class scans a directory with all sub-directories and files … and gets a list of all Drives (Removal, Physical or CD-ROM).

1. **Scan(DirectoryName):** Begins the scanning inside this directory
2. **GetDrives():** This Function returns the Drives paths (ex: C:\, D:\ …) and returns the type of these drives.
3. **DirectoryCallback(…):** this function is a virtual function and it’s called when the scanner detects a directory … and it returns true if you want to scan this directory
4. **FileCallback(…):** it’s a virtual function and it’s called when the scanner detects a file inside a directory

**Example:**

class OurScanner : public cRecursiveScanner

{

public:

OurScanner(){};

~OurScanner();

virtual void FileCallback(cString Filename,cString FullName,int Level);

virtual bool DirectoryCallback(cString DirName,cString FullName,int Level);

};

bool OurScanner::DirectoryCallback(cString DirName,cString FullName,int Level)

{

//scan files and sub-directories inside this directory

return true;

}

void OurScanner::FileCallback(cString Filename,cString FullName,int Level)

{

cFile\* MD5File = new cFile(FullName);

cMD5String MD5;

cout << "Filename : " << (char\*)Filename << "\n";

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

OurScanner\* Scan = new OurScanner();

Scan->Scan("%systemroot%\\system32\\");

return 0;

}

## CPokasAsm:

**Discription:**

This Class is a wrapper class for Pokas x86 Emulator’s Assembler/ Disassembler Engine … this class disassemble and assemble instructions from opcodes to mnemonics or to a structure describe the instruction (for analysis) and it’s named “DISASM\_INSTRUCTION”

1. **CPokasAsm():** initializes the Pokas Emulator’s Disassembler Engine
2. **Assemble(InstructionString, &Length):** this function assembles an string to opodes (to bytes) and returns it … and returns the length in the 2nd parameter
3. **Disassemble(buffer, &InstructionLength):** this function disassembles an instruction from opcodes to mnemonics and returns the length of the instruction (the length of the opcode shape before disassemble … not the length of the returned string) in InstructionLength
4. **Disassemble(buffer, DISASM\_INSTRUCTION\* ins):** this function disassembles an instruction and returns a struct describes the instruction and it looks like:
   1. **hde:** it’s a struct created by Hacker Disassembler Engine and describes the opcode … The important Fields are:
      1. **len:** The length of the instruction
      2. **opcode:** the opcode byte … if the opcode is 2 bytes so see also opcode2
      3. **Flags:** This is the flags and it has some important flags like “F\_MODRM”and “F\_ERROR\_XXXX” (XXXX means anything here)
   2. **Entry:** unused
   3. **Opcode:** the opcode string … with class “string” not “cString”
   4. **Other:** used for mul to save the imm … other than that … it’s unused
   5. **Modrm:** it’s a structure describes what’s inside the RM (if there’s) like “[eax\*2 + ecx + 6]” for example … and it looks like:
      1. **Length:** the number of items inside … like “[eax+ 2000]” contains 2 items
      2. **Flags[3]:** this describes each item in the RM and its maximum is 3 … it’s flags is:
         1. **RM\_REG:** the item is a register like “[eax …”
         2. **RM\_MUL2:** this register is multiplied by 2
         3. **RM\_MUL4:** by 4
         4. **RM\_MUL8:** by 8
         5. **RM\_DISP:** it’s a displacement like “[0x401000 + …”
         6. **RM\_DISP8:** comes with RM\_DISP … and it means that the displacement is 8-bits
         7. **RM\_DISP16:** the displacement is 16 bits
         8. **RM\_DISP32:** the displacement is 32-bits
         9. **RM\_ADDR16:** this means that … the modrm is in 16-bits Addressing Mode
      3. **Items[3]:** this gives the value of the item in the modrm … like if the Item is a register … so it contains the number of this register (ex: ecx 🡪 item = 1)  
         and if the item is a displacement … so it contains the displacement value like “0x401000” and so on.
   6. **emu\_func:** unused
   7. **Flags:** this flags describes the instruction … some describes the instruction shape, some describes destination and some describes the source … let’s see
      1. **Instruction Shape:** there are some flags describe the instruction like:
         1. **NO\_SRCDEST:** this instruction doesn’t have source or destination like “nop”
         2. **SRC\_NOSRC:** this instruction has only destination like “push dest”
         3. **INS\_UNDEFINED:** this instruction is undefined in the disassembler … but you still can get the length of it from hde.len
         4. **OP\_FPU:** this instruction is an FPU instruction
         5. **FPU\_NULL:** means this instruction doesn’t have any destination or source
         6. **FPU\_DEST\_ONLY:** this means that this instruction has only a destination
         7. **FPU\_SRCDEST:** this means that this instruction has a source and destination
         8. **FPU\_BITS32:** the FPU instruction is in 32-bits
         9. **FPU\_BITS16:** means that the FPU Instruction is in 16-bits
         10. **FPU\_MODRM:** means that the instruction contains the ModRM byte
      2. **Destination Shape:**
         1. **DEST\_REG:** means that the destination is a register
         2. **DEST\_RM:** means that the destination is an RM like “dword ptr [xxxx]”
         3. **DEST\_IMM:** the destination is an immediate (only with enter instruction”
         4. **DEST\_BITS32:** the destination is 32-bits
         5. **DEST\_BITS16:** the destination is 16-bits
         6. **DEST\_BITS8:** the destination is 8-bits
         7. **FPU\_DEST\_ST:** means that the destination is “ST0” in FPU only instructions
         8. **FPU\_DEST\_STi:** means that the destination is “STx” like “ST1”
         9. **FPU\_DEST\_RM:** means that the destination is RM
      3. **Source Shape:** similar to destination … read the description in Destination flags above
         1. **SRC\_REG**
         2. **SRC\_RM**
         3. **SRC\_IMM**
         4. **SRC\_BITS32**
         5. **SRC\_BITS16**
         6. **SRC\_BITS8**
         7. **FPU\_SRC\_ST**
         8. **FPU\_SRC\_STi**
   8. **ndest:** this includes the value of the destination related to its type … if it’s a register … so it will contains the index of this register  
      if it’s an immediate … so it will have the immediate value  
      if it’s an RM … so it will be null
   9. **nsrc:** this includes the value of the source related to the type … see the ndest above

**Example:**

CPokasAsm\* Asm = new CPokasAsm();

DWORD InsLength;

char\* buff;

buff = Asm->Assemble("mov eax,dword ptr [ecx+ 00401000h]",InsLength);

cout << "The Length: " << InsLength << "\n";

cout << "Assembling mov eax,dword ptr [ecx+ 00401000h]\n\n";

for (DWORD i = 0;i < InsLength; i++)

{

cout << (int\*)buff[i] << " ";

}

cout << "\n\n";

cout << "Disassembling the same Instruction Again\n\n";

cout << Asm->Disassemble(buff,InsLength) << " ... and the instruction length : " << InsLength << "\n\n";

**Output:**

The Length: 6

Assembling mov eax,dword ptr [ecx+ 00401000h]

FFFFFF8B FFFFFF81 00000000 00000010 00000040 00000000

Disassembling the same Instruction Again

mov eax ,dword ptr [ecx + 401000h] ... and the instruction length : 6

## CPokasEmu:

**Discription:**

This Class is a wrapper class for Pokas x86 Emulator … it emulates a PE File … or a shellcode

1. **cPokasEmu(Filename, DLLPath):** this function emulates a PE File given its filename and The path to the system dlls like “C:\Windows\System32\”
2. **cPokasEmu(cPEFile,DLLPath):** this function is similar to the previous function but this takes a cPEFile object
3. **CPokasEmu(buff,size,ImageType, DLLPath):** this is used to emulate a PE File in memory (loaded or unloaded) … or emulating shellcode depends on ImageType
   1. **PROCESS\_LOADEDIMAGE:** for a loaded image in memory (using a PE Loader or a Process in memory)
   2. **PROCESS\_UNLOADEDIMAGE**: for normal unloaded image (Memory Mapped Files or something)
   3. **PROCESS\_SHELLCODE:** for shellcode
4. **Emulation Functions:**
   1. **Emulate():** runs the executable application in the virtual environment (emulate the application)
   2. **Step():** it goes one step in the application … emulate only one instruction
5. **Breakpoint Functions:**
   1. **SetBreakpoint(Breakpoint):** this function sets a breakpoint … here’s some examples:
      1. **“Eip==0x00401000”:** like Int3 Breakpoint or Hardware on Execution
      2. **“\_\_lastaccessed()==0x00401000”:** Memory on Access
      3. **“\_\_lastmodified()==0x00401000”:** Memory on Write
      4. **“\_\_isdirty(eip)”:** execution on modified data (used in encrypted viruses or polymorphic viruses … also used for packers)
      5. **“\_\_isdirty(eip) && eip>=0x401000 && eip<=0x405000”:** merging with ands and ORs
      6. **“\_\_isdirty(eip) && (\_\_read(eip) & 0xff) !=0xC3)”:** escaping from “ret” instruction … “read” function is used for reading from memory
      7. **“\_\_isapi()”:** breaks on calling to any API (stops on the 1st instruction of the API)
      8. **“\_\_isapiequal(‘Getprocaddress’)”:** breaks on calling to a specific API
   2. **SetBreakpoint(FuncName, FuncPointer):** this function sets a function breakpoint … you put a function and it will be called on every instruction emulated.
   3. **DisableBreakpoint(index):** this function disables a breakpoint you added before
6. **Memory Management Functions:**
   1. **GetNumberOfMemoryPages():** get all allocated memory regions inside (Pokas Emulator don’t handle a page == 0x1000 … but work with regions … like the kernel32.dll loaded module is a Page in the memory … or is one region)
   2. **GetMemoryPage(index):** Get a Memory Region with index (usually used with loops) and returns a struct “MEMORY\_STRUCT” describes the memory … the struct is described below
   3. **GetMemoryPageByVA(Addr):** this function gets the Memory Region by an address inside this region .. or return s null for nothing …the struct “MEMORY\_STRUCT” Fields are:
      1. **VirtualAddr:** this is the virtual address of this region (the beginning of this region)
      2. **RealAddr:** the equivalent real address
      3. **Size:** the size of this memory region
      4. **Flags:** describe the type and the protection of this memory region and it’s:
         1. **MEM\_READWRITE**
         2. **MEM\_READONLY**
         3. **MEM\_IMAGEBASE:** this is means that … this region is the module of the emulated PE File
         4. **MEM\_DLLBASE:** this means that … this region is a DLL module
         5. **MEM\_VIRTUALPROTECT:** this means that … this is not a new region … but it could be a part of another region but its protection is modified with VirtualProctect API
   4. **GetRealAddr(VA):** this function gets the real address of a virtual address inside the emulated application (the real address is the address in the real memory … not in the emulated memory)
   5. **GetNumberOfDirtyPages():** this function gets the number of dirty pages (regions) in memory (the modified regions in memory)
   6. **GetDirtyPage(index):** gets the modified memory region using the index … and returns a struct “DIRTYPAGES\_STRUCT” and it contains:
      1. **vAddr:** the virtual address of this region (in the emulated memory) …use GetRealAddress to read or write to it
      2. **Size:** the size of the modified region
      3. **Flags:** unused
   7. **ClearDirtyPages():** clear all dirty flags on all regions … begin monitoring the changes from the beginning (don’t clear the new written data … only clear the dirty flag)
7. **Registers and Information:**
   1. **GetReg(index):** Getting a register value using its index (only from eax to edi)
   2. **GetEip():** Getting the Eip
   3. **GetEFLAGS():** Gets the EFlags (OF, ZF …)
   4. **GetTIB():** gets the beginning of the Tib (FS:[0]) … as a virtual address not real address
   5. **GetImagebase():** gets the imagebase
   6. **SetReg(index,value):** modifies the value of a register
   7. **SetEip(value):** modifies the value of the Eip
   8. **SetEFLAGS(value):** modifies the value of the EFlags
   9. **GetDisassembly(VA, &outputString):** disassembles an instruction and returns its string output … using the virtual Address not the real address …and returns the length of the instruction (opcodes … not the length of the string)
8. **Dumping The File**
   1. **MakeDumpFile(OutputFile, ImportFixType):** This function dumps the process into a file again … unloads it and fix its import table again … and then saves it into OutputFile … the ImportFixType could be:
      1. **DUMP\_ZEROIMPORTTABLE:** this flags tells the dump function to Zero the Import Table DataDirectory as there’s no import table
      2. **DUMP\_UNLOADIMPORTTABLE:** this makes the dump function unloads the Import Table like it was loaded in the beginning
      3. **DUMP\_FIXIMPORTTABLE:** this makes the dump function to create a new Import Table with the Loaded Libraries using “LoadLibraryA” and the loaded APIs using “GetProcAddress” … it’s very efficient and works fine for most of packers
9. **APIs and DLLs:**  
   1. **DefineDLL(DLLName,DLLPath,VirtualAddress):** this functions loads a DLL given its name and its path (like DefineDLL( “kernel32.dll”, ”c:\windows\system32” …) and gives the preferred virtual address for it
   2. **DefineAPI(DLLBase,APIName,nArgs,APIFunc() ):** this functions defines a new API emulation function … given its DLL Imagebase, API Name, the number of Arguments for this function … and the Address of the emulation function for this API … and the Emulation function should be like that:  
      DWORD APIFunc(DWORD unused,DWORD ins,Arguments ...)

**Example:**

DWORD ptr;

CPokasEmu\* emu = new CPokasEmu("C:\\upx01.exe","C:\\WINDOWS\\SYSTEM32\\");

cout << "SRDFTest\n---------\n";

emu->SetBreakpoint("\_\_isdirty(eip)");

emu->SetBreakpoint("func01",(DWORD)&IsTimeEqual);

char\* buffer = (char\*)malloc(100);

MEMORY\_STRUCT\* mem = emu->GetMemoryPageByVA(emu->GetImagebase());

cout << (int\*)mem->VirtualAddr << " " << (int\*)mem->PhysicalAddr << "\n";

cout << "Start Emulation From : " <<(int\*)emu->GetEip() << "\n";

cout << "-------------------------------\n";

system("pause");

emu->Emulate();

cout << "Emulated Successfully\n\nThe Disassembled Code:\n";

memset(buffer,0,100);

ptr = emu->GetEip();

for (int i = 0; i < 30; i++)

{

memset(buffer,0,100);

cout << (int\*)ptr << " : ";

ptr += emu->GetDisassembly((char\*)ptr,buffer);

cout << buffer << "\n";

}

emu->MakeDumpFile("C:\\upx01\_unpacked.exe",DUMP\_FIXIMPORTTABLE);

system("pause");

delete emu;

## cDebugger:

**Discription:**

This class debugs an Application … monitors it and set breakpoints in it … let’s see:

1. **cDebugger(Filename, Commandline):** This function creates a new process and attach to it … it stops on the first breakpoint
2. **cDebugger(cProcess):** Attach to a running process
3. **Run():** execute the application to the next breakpoint … and returns one of these values:
   1. **Value > 0:** the int3 Breakpoint number
   2. **DBG\_STATUS\_ERROR:** There’s an Error … you can get much information from debug\_event
   3. **DBG\_STATUS\_STEP:** Single Step
   4. **DBG\_STATUS\_HARDWARE\_BP:** Hardware Breakpoint
4. **Step():** execute only one instruction using the dingle step flag
5. **Pause():** this function is used when you execute Run() function in a thread… and you need to stop it from another thread … so using another thread you can call to Pause()
6. **Resume():** this function resumes the running after calling to function Pause()
7. **Terminate():** this function terminates the process
8. **SetBreakpoint(Address):** Sets a breakpoint (int3 breakpoint) on a specific address … returns true if it successfully added the breakpoint
9. **RemoveBreakpoint(Address):** Remove an in3 breakpoint from an address
10. **SetHardwareBreakpoint(Address,Type):** this function sets a hardware breakpoint (maximum 4 breakpoints) … and the type is:
    1. **DBG\_BP\_TYPE\_CODE:** breakpoint on execution
    2. **DBG\_BP\_TYPE\_READWRITE:** breakpoint on read or write
    3. **DBG\_BP\_TYPE\_WRITE:** breakpoint on write
11. **RemoveHardwareBreakpoint(Address):** removes a hardware breakpoint
12. **Virtual DLLLoadedNotifyRoutine():** this function is a virtual function (to be overridden) and this function receives the notification of a loaded DLL.
13. **Virtual DLLUnloadedNotifyRoutine():** this function receives a notification of unloaded DLL … you can get more information from debug\_event
14. **Virtual ThreadCreatedNotifyRoutine():** this function receives a notification of a creation of a new Thread … you can get more information from debug\_event
15. **Virtual ThreadExitNotifyRoutine():** this function receives a notification of the exiting (or terminating) of a thread … you can get more information from debug\_event
16. **Virtual** **ProcessExitNotifyRoutine():** this function receives a notification of terminating the process (or using ExitProcess) … you can get more information from debug\_event
17. **DebuggeeProcess:** this is an object from cProcess class for the Debugee Process and you can refresh it to get the recent information about allocated memory and loaded DLLs and so on.
18. **DebuggeePE:** this is an object from cPEFile and it includes all information in the file headers and PE headers
19. **DebugStatus:** this is the Dr7 … and it’s used with Hardware Breakpoints
20. **ExceptionCode:** that’s the last exception code from the last error

**Example**

cDebugger\* Debugger = new cDebugger("C:\\upx01.exe");

Debugger->SetBreakpoint((DWORD)GetProcAddress(LoadLibrary("Kernel32.dll"),"GetProcAddress"));

cout << Debugger->Run() << "\n";

cout << "First GetProcAddress\n";

cout << "Debug Event: " << Debugger->ExceptionCode << "\n";

cout << "Eip: " << Debugger->Eip << "\n";

Debugger->Step();

cout << hex << Debugger->Eip << "\n";

Debugger->Step();

cout << hex << Debugger->Eip << "\n";

Debugger->Step();

cout << hex << Debugger->Eip << "\n";

Debugger->Step();

cout << hex << Debugger->Eip << "\n";

cout << Debugger->Run() << "\n";

cout << "Second GetProcAddress\n";

Debugger->RemoveBreakpoint((DWORD)GetProcAddress( LoadLibrary("Kernel32.dll"), "GetProcAddress"));

cout << "LastBp: " << Debugger->LastBreakpoint << "\n";

Debugger->SetHardwareBreakpoint(0x00401000, DBG\_BP\_TYPE\_CODE, DBG\_BP\_SIZE\_4);

cout << Debugger->Run() << "\n";

cout << "Eip: " << Debugger->Eip << "\n";

cout << Debugger->Run() << "\n";

# Core:

### cApp:

**Discription:**

This class is an abstract class. It’s created to be the interface and the management system of the SRDF. And it’s manages you LogFile, Back-end Database and Registry Entry for your application

1. **GetApplicationFilename():** Gets the application Filename
2. **GetApplicationPath():** Gets the path of the application file
3. **SetCustomSettings():** Set the main setting for the application
   1. **Flags:** It represents the setting of the Application
      1. **APP\_NOANOTHERINSTANCE:** Allows only one instance of the application to be running
      2. **APP\_ADDLOG:** Adds a LogFile for the Application
      3. **APP\_DEFINEDATABASE:** Adds a Back-end Database for your App
      4. **APP\_REGISTRYSETTINGS:** Adds a Registry Key for your Application for your settings
   2. **LogFilename:** Set the Log Filename to create a log file for the application
   3. **RegistryPath:** Set the registry path for the setting of the application
   4. **RegistryType:** HKEY\_CURRENT\_USER or HKEY\_LOCAL\_MACHINE
   5. **Options:** Set the command line options for your application like “ab:c”. which every character represents an option … and “:” represents that it will take a value like this:  
      yourapp.exe –a 🡪 that’s mean that it uses “a” option.  
      yourapp.exe –b <value> 🡪 as “b” option takes a value.
   6. **Request:** this is a cHash class contains all the options that the user entered which will be like this:  
      yourapp.exe –a 🡪 Request[“a”] == “Found”  
      yourapp.exe –b <value> 🡪 Request[“b”] == “<value>”  
      yourapp.exe –a <value> 🡪 Request[“a”] == “Found” / Request[“b”] == “” / Request[“default”] == “<value>”  
      yourapp.exe <value> 🡪 Request[“default”] == “<value>”
4. **Database:** your back-end database.
5. **Log:** your log.
6. **Settings:** your registry entry for settings.

**Example:**

cMyApp::cMyApp(cString AppName) : cApp(AppName)

{

}

cMyApp::~cMyApp()

{

((cApp\*)this)->~cApp();

}

void cMyApp::SetCustomSettings()

{

Flags |= (APP\_NOANOTHERINSTANCE | APP\_ADDLOG | APP\_REGISTRYSETTINGS | APP\_DEFINEDATABASE);

Options = "o:"; //myapp.exe -o <outputfile> <filename>

LogFilename = AppPath;

LogFilename += "\\LogFile.txt";

RegistryPath = "Software\\";

RegistryPath += AppName;

RegistryType = HKEY\_CURRENT\_USER;

}

int cMyApp::Run()

{

cout << "Running App\n";

return 0;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

cMyApp\* MyApp = new cMyApp("MyApp");

MyApp->SetCustomSettings();

MyApp->Initialize(argc,argv);

MyApp->Run();

return 0;

}

## cConsoleApp:

**Discription:**

this is an inherited class from cApp and it’s used to develop an interactive console application that contains commands inside and help menu … like msfconsole in Metasploit.

1. **Intro:** the application intro at the beginning of the application
2. **Prefix:** when the user requested to enter the input command … it makes like dos and put this mark “>” to tell the user to enter a command … you could add a prefix to it
3. **AddCommand(Name,Description, Format,nArgs,CmdFunc):** it’s used to add a command to the command list of the console application. It takes the name and the description (the description will appear in help menu) and the format (which will appear when the user enters the command missing arguments
   1. **CmdFunc:** this function s the callback routine for the command and it’s like that:  
      void CmdFunc(cConsoleApp\* App,int argc,char\* argv[]);  
      all argv[x] are strings
      1. **To Convert them into numbers:** you will use atoi(string) function
      2. **To Convert them from Hexadecimal String like “4DF8” to a number:** you will use sscanf like this:  
         DWORD Address = 0;

sscanf(argv[0], "%x", &Address);

1. **StartConsole():** you call to this function in Run() function to start the console application
2. **Exit():** it’s a virtual function (need to implement it yourself) to be called after the user input “quit” command.

**Example:**

cDebuggerApp::cDebuggerApp(cString AppName) : cConsoleApp(AppName)

{

}

cDebuggerApp::~cDebuggerApp()

{

((cApp\*)this)->~cApp();

}

void cDebuggerApp::SetCustomSettings()

{

Intro = "\

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\

\*\* Win32 Debugger \*\*\n\

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

AddCommand("step","one Step through code","step",0,&StepFunc);

AddCommand("run","Run the application until the first breakpoint","run",0,&RunFunc);

}

int cDebuggerApp::Run()

{

//Get the commandline argument .. default is the normal argument for no command

//not like Debug.exe -o:444 .. but like Debug.exe xxx.exe

Debugger = new cDebugger(Request.GetValue("default"));

Asm = new CPokasAsm();

if (Debugger->IsDebugging)

{

Debugger->Run();

Prefix = Debugger->DebuggeeProcess->processName;

if (Debugger->IsDebugging)StartConsole();

}

else

{

cout << Intro << "\n\n";

cout << "Error: File not Found";

}

return 0;

}

int cDebuggerApp::Exit()

{

cout << "Thanks For using our Debugger\n";

if (Debugger->IsDebugging)Debugger->Terminate();

delete Asm;

return 0;

}

**Output:**

D:\.... >DebuggerExample.exe C:\upx01.exe

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Win32 Debugger \*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

upx01.exe>help

The Commands List:

-------------------

help: Show Help

quit: Exit the Console

step: one Step through code

run: Run the application until the first breakpoint

upx01.exe>quit

Thanks For using our Debugger

# Kernel-Mode:

## Driver:

**Discription:**

This class is an abstract class and it is created for DeviceObject management and IRP Dispatching … it’s the main class for any driver you create using SRDF.

1. **DriverMain(pDriverObject,** **theRegistryPath):** this functionyou should implement it yourself and this function is the main function in the Driver
2. **DriverUnload():** this function also you should implement youself and it’s for unloading
3. **AddDevice(Device\* device):** this function adds an Object from Device Class to an array in the Driver Class to dispatch IRPs to it
4. **MultiDeviceIrpDispatcher**: This function is a private function and it dispatch the IRPs to the added Devices
5. **FileFilterNotificationDispatcher:** This function is a private function and it dispatch the FileFilterNotification to every FileFilterDevice Added to the Driver.

**Example:**

NTSTATUS Driver::DriverMain(IN PDRIVER\_OBJECT pDriverObject,IN PUNICODE\_STRING theRegistryPath)

{

DbgPrint("DriverEntry Called\n");

Amr=(SSDTDevice\*)misc::CreateClass(sizeof(SSDTDevice));

Amr->Initialize(this);

AddDevice(Amr);

return STATUS\_SUCCESS;

}

VOID Driver::DriverUnload()

{

DbgPrint("Device Detached");

Amr->Detach();

}

## Device:

**Discription:**

This class represents a Task or a Job to be done using the Device Driver. This class is the parent of every class that do a specific job … like SSDTDevice or FilterDevice or TdiSocket and so on.

This class manages the IRPs while being dispatched from the Driver and dispatches them to every Major Fuction you implment or ignore it. Also, this class includes a user-mode/kernel-mode communication protocol.

1. **CreateDevice(DeviceName, SymbolicName):** this function creates a Control DeviceObject to create a communication channel between this Device and the User-Mode
2. **Initialize(driver):** this is an intialization function and it takes the your main Driver as an input.
3. **MajorFunction[IRP\_MJ\_MAXIMUM\_FUNCTION]:** this is an array of functions you set anyone of them to handle the related IRP. You create a function like this:  
   int \_cdecl MJCreate(FileFilterDevice\* FFDevice,\_\_in PDEVICE\_OBJECT DeviceObject,\_\_in PIRP Irp)  
   and set it in the related MajorFunction like this:  
   SetValue(MyDevice->MajorFunction[IRP\_MJ\_CREATE],MJCreate);
4. **UserCommunication::** **Write(msgcode, status, data,size):** this function sends data to the User-Mode .. you set the MsgCode, status and the data for your message … and set the size of your data.
5. **UserCommunication::** **RegisterReadFunction(Func):** this function registers a function to be notified with any message received from the User-Mode and it should be like this:  
   void ReadFunc(char msgcode,DWORD status,DWORD size,char\* data);
6. **UserCommunication::** **RegisterFastMsgFunction(Func):** this function registers a function that receives a message from the User-Mode … the message is a must-reply data … so you must send data to the User-Mode as a reply for this message .. the function is like this:  
   void FastMsgFunc(char msgcode,DWORD status,DWORD size,char\* data,char\* output,DWORD outputsize,DWORD &return\_status,DWORD &return\_size);

**Example:**

MyDevice =(Device\*)misc::CreateClass(sizeof(Device));

MyDevice->Initialize(this);

AddDevice(MyDevice);

MyDevice->CreateDevice(L"\\Device\\dev",L"\\DosDevices\\dev");

## SSDTDevice:

**Discription:**

This class is created for hooking the SSDT Table and makes any function calls to your function at the beginning. This class is inherited from Device class

1. **GetRealAddress(FunctionName):** this function gets the address of any function from inside the SSDT Table
2. **AttachTo(FunctionName, newFunction):** This function modifies the SSDT Table and set you function pointer to be called rather than the real function. You can say it hooks the this function using the SSDT
3. **Detach():** This function unhooks the hooked function in the SSDT
4. **FuncIndex:** The Function Index inside the SSDT Table

**Example**

NTSTATUS newZwSetValueKey(IN HANDLE KeyHandle,IN PUNICODE\_STRING ValueName,IN ULONG TitleIndex OPTIONAL,IN ULONG Type,IN PVOID Data,IN ULONG DataSize)

{

DbgPrint("Yes %wZ\n",ValueName);

return (\*oldZwSetValueKey)(KeyHandle,ValueName,TitleIndex,Type,Data,DataSize);

}

NTSTATUS Driver::DriverMain(IN PDRIVER\_OBJECT pDriverObject,IN PUNICODE\_STRING theRegistryPath){

Amr=(SSDTDevice\*)misc::CreateClass(sizeof(SSDTDevice));

Amr->Initialize(this);

AddDevice(Amr);

int old = Amr->GetRealAddress(L"ZwSetValueKey");

DbgPrint("Real Address : 0x%x",old);

Amr->AttachTo(L"ZwSetValueKey",(DWORD)newZwSetValueKey);

SetValue(oldZwSetValueKey,old);

return STATUS\_SUCCESS;

}

VOID Driver::DriverUnload()

{

DbgPrint("Device Detached");

Amr->Detach();

}

## FilterDevice:

**Discription:**

This Class is inherited from Device Class and it’s created to be attached to a DeviceObjects chain and to hook the I/O Request Packets (IRPs).

1. **FilteredMajorFunction[IRP\_MJ\_MAXIMUM\_FUNCTION]:** this is an array of functions you set to hook on the DeviceObjects chain you hooked … you can set the PreModification and the PostModification:  
   struct FilteredMajorFunctionStruct

{

PDeviceMJ PreModification;

PDeviceMJ PostModification;

};

1. **AttachToDevice(DeviceName):** attach to a DeviceObjects chain given its name in Unicode
2. **AttachToDevice(DeviceObjec):** attach to a DeviceObjects chain given its DeviceObject
3. **DetachDevice( DeviceName):** detach from the DeviceObjects chain given its name in Unicode
4. **Pending(DeviceObject, Irp):** this function is called inside the PreModification Function to wait until the IRP be processed by all device objects under it and then retake the IRP again and modify the IRP (like post modification)

**Example:**

Filter->AttachToDevice(L"\\FileSystem\\ntfs",FILE\_DEVICE\_DISK\_FILE\_SYSTEM);

SetValue(Filter->FilteredMajorFunction[IRP\_MJ\_CREATE].PreModification,MJCreate);

…

int \_cdecl MJCreate(FileFilterDevice\* FFDevice,\_\_in PDEVICE\_OBJECT DeviceObject,\_\_in PIRP Irp)

{

PFILE\_OBJECT pFileObject = IoGetCurrentIrpStackLocation(Irp)->FileObject;

DbgPrint("File Created !!!");

DbgPrint("%wZ\n", &pFileObject->FileName);

return FILTER\_SKIP;

}

## FileFilterDevice:

**Discription:**

This class is inherited from FilterDevice and it’s a Filesystem Filter. This class attaches itself to the Deviceobject chain of the Filesystems and hooks them … it attaches itself to any newly attached filesystem (like Removable Disk) and hooks it.

1. **BeginHooking(BOOLEAN HookNewlyMountedVolumes):** this begins the hooking of the Filesystems … and could attach to the newly attached volumes.
2. **GetFileInformationOffsets(IRP):** This function is called when a Query Information is called … it gets the offset of every important field on the Filesystem Query structure … and returns them in a struct like this:  
   struct FileInformationOffsets

{

DWORD FileName;

DWORD ShortName;

DWORD ShortNameLength;

DWORD FileNameLength;

DWORD FileAttributes;

DWORD AllocationSize;

DWORD EndOfFile;

DWORD FileIndex;

DWORD CreationTime;

DWORD LastAccessTime;

DWORD LastWriteTime;

DWORD ChangeTime;

DWORD EaSize;

DWORD FileId;

DWORD OpFileReference;

DWORD OpObjectId;

DWORD OpBirthObjectId;

DWORD NextEntryOffset;

};

**Example:**

Filter->BeginHooking(true);

SetValue(Filter->FilteredMajorFunction[IRP\_MJ\_CREATE].PreModification,MJCreate);

## TdiTcpSocket:

**Discription:**

This is an TCP communication class and it’s based on Tdi. This class is inherited from Device Class.

1. **InitializeConnection(SocketType, port):** This function initialize the TDI and it takes the protocol that you need and the port for your socket … the type could be:
   1. **TCP\_SOCKET\_CLIENT:** creates a client socket
   2. **TCP\_SOCKET\_SERVER:** creates a server socket (listening socket)
2. **Connect(charIP1, charIP2, charIP3, charIP4, Port):** this function connects to a remote server given its IP and the destination port
3. **Listen():** this function listen to any connection and waits until a remote node connects to you.
4. **Disconnect():** disconnect your connection
5. **DeinitializeConnection():** deinitialize the TDI after the initialization … important to be called at the end of the usage of your socket or at the unload function.
6. **Send(Buffer, size, \*pDataSent):** this function sends data to the remote node and returns the number of bytes which actually sent.
7. **ReceiveEventFunc:** this variable you set with a pointer to a function which receives any data sent to your node … the function should be like this:  
   int ReceiveEvent(TdiTcpSocket\* Sock,char\* Buffer,DWORD Size);
8. **DisconnectEventFunc:** this variable you set with a pointer to a function which receives a notification that the remote node disconnected the connection and should be like that:  
   VOID DisconnectEvent(TdiTcpSocket\* Sock);

**Example:**

Socket = (TdiTcpSocket\*)misc::CreateClass(sizeof(TdiTcpSocket));

Socket->Initialize(this);

AddDevice(Socket);

Socket->ReceiveEventFunc = TdiReceiveEvent;

Socket->DisconnectEventFunc = TdiDisconnectEvent;

if (Socket->InitializeConnection(TCP\_SOCKET\_CLIENT,4003) != STATUS\_SUCCESS)

DbgPrint("Error Initializing Connection");

Socket->Connect(127,0,0,1,4400);

DWORD DataSent;

Socket->Send("From TCP Socket",strlen("From TCP Socket"),&DataSent);

LARGE\_INTEGER interval;

interval.QuadPart = 7 \* DELAY\_ONE\_SECOND;

KeDelayExecutionThread(KernelMode,FALSE,&interval);

Socket->Disconnect();

Socket->DeinitializeConnection();

## TdiFirewall:

**Discription:**

This class is used to intercept the inbound and the outbound connections from the Processes and could allow them to pass or deny the connection. This class is inherited from FilterDevice

1. **BeginHooking(bool OnlyThroughFirewall):** This function begins the connection monitoring … if OnlyThroughFirewall is set andit can’t find a connection object, it refuses it. It’s good if you don’t need to allow any abnormal connection and need to force all connection to be filtered
2. **CreateConnectionEvent:** this is a pointer to a function you set to intercept the creation of a server to listen or a client to be used to connect … and it looks like that:  
   int CreateConnectionEventFunc(TdiSniffer\* Sock,DWORD PID,USHORT Port,OUT PVOID &UserContext)

it gives you the Process Id and the Source Port.

1. **CloseConnectionEvent:** this is a pointer to a function you set to intercept the closing of a connection .. and you function should like that:  
   int CloseConnectionEventFunc(TdiSniffer\* Sock,DWORD PID,PVOID UserContext);
2. **ConnectEvent:** this intercepts a connect from a local node to a remote node  
   int ConnectEventFunc(TdiSniffer\* Sock,DWORD PID,DWORD ConnectionType,IPADDR\* IPAddress,DWORD Port,PVOID UserContext)  
     
   it gives you the Process Id, the connectionType (client or server), the IPAddress, the port and a usercontext you set
3. **SendEvent:** this monitors any send request from a local node  
   int SendEventFunc(TdiSniffer\* Sock,DWORD PID,char\* Buffer,DWORD\* Size,PVOID UserContext)
4. **ReceiveEvent:** this monitors any received data from the remote node … it’s still include bugs and don’t receive most of packets  
   int ReceiveEventFunc(TdiSniffer\* Sock,DWORD PID,char\* Buffer,DWORD\* Size,PVOID UserContext)

All of these functions can return “TDISNIFFER\_ALLOW” or “TDISNIFFER\_DENY” except send and receive requests.

**Example:**

int MyReceiveEvent(TdiTcpSocket\* Sock,int PID, char\* Buffer,DWORD\* Size,PVOID UserContext)

{

DbgPrint("From My Function !!!");

DbgPrint("Size : %x",\*Size);

return TDISNIFFER\_ALLOW;

}

Firewall=(TdiFirewall\*)misc::CreateClass(sizeof(TdiSniffer));

Firewall->Initialize(this);

AddDevice(Amr4);

Amr4->BeginHooking(true);

SetValue(Firewall->ReceiveEvent,MyReceiveEvent);

SetValue(Firewall->SendEvent,MyReceiveEvent);

# Misc:

## Registry Manager:

**Discription:**

This is a namespace includes 2 function for reading and writing from/to the registry. They work at anytime rather than the IRQL that you are running in.

1. **RegRead(KeyName, ValueName, &Length):** This function reads a value in the registry and return the value and the length.
2. **RegWrite(KeyName,ValueName,buffer, BufType, Length):** This function writes to the registry given the Registry Path and Value Name, The Type of the Registry Value (REG\_SZ or …), the buffer and the length of this buffer.

**Example:**

RegWrite(L"\\Registry\\Machine\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion",L"SRDF",(char\*)L"SRDF Kernel-Mode",REG\_SZ,strlen("SRDF Kernel-Mode")\*2);

DWORD size;

char\* buf = RegRead(L"\\Registry\\Machine\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion",L"ProgramFilesDir",size);

if(buf != 0)DbgPrint("Registry Read : %x",buf);

## File Manager:

This a namespace contains classes for managing reading and writing from/to Harddisk

### FileToWrite:

**Discription:**

This Class is inherited from an abstract class named File … this class writes to File any data rather than the IRQL that you are running in.

1. **open(Filename,bool Append):** This function opens a file to write … if it’s not found it will create it .. if not , it could rewrite on it or append to it.
2. **Close():** this closes the file.
3. **write(data, size):** this writes the data to the File.

**Example:**

FileToWrite\* s = (FileToWrite\*)misc::CreateClass(sizeof(FileToWrite));

s->open(L"\\DosDevices\\c:\\NewData.txt",false);

s->write("SRDF ... From KernelMode\n",strlen("SRDF ... From KernelMode\n"));

s->close();

### FileToRead:

**Discription:**

This Class is inherited from an abstract class named File … this class reads from a File any data rather than the IRQL that you are running in.

1. **open(Filename,):** This function opens a file to read
2. **Close():** this closes the file.
3. **read(&data, &size):** this reads the data from the File.

**Example:**

FileToRead\* readfile = (FileToRead\*)misc::CreateClass(sizeof(FileToRead));

NTSTATUS ntStatus = readfile->open(L"\\DosDevices\\c:\\KeyLog2dfdfdf.txt");

if (ntStatus != STATUS\_SUCCESS)DbgPrint("02.cpp : Failed To ReadFile");

readfile->close();