STE Platform & SHADE Plugin API by SHADE SANDBOX LLC

Table of contents

Overview

STE Platform (Security Toolkit Engine) platform by SHADE Sandbox LLC is aimed to simplify development of security-related software such as virtual environments (featherweight sandboxes) ,system monitors , antiviral software or file management software. The platform consists of a kernel mode driver , a proxy dll and a user mode service. The service is able to load plugins which are to implement desired business logic. STE is functionally divided into following components:

File System Monitor

This component intercepts the following file system operations : CREATE/OPEN of a file , LOADing and UNLOADing of executable modules , DELETion of files and directories, CLOSing of files, and enumerating of directories contents. FSSDK user is able to monitor , allow or deny all of mentioned operations. It is also possible to redirect OPEN/CREATE request to any subdirectory or registry branch. Redirection is, however, quite complex operation, and includes not only redirection of an operation but some additional actions which are required to implement a sandbox functionality that was kept in mind during the platform development. To support most functionality of a sandbox, a developer only has to create a group of rules for the platform in their sandbox platform. Cybergenic Shade is a great example of a sandbox that sits back on this platform. In order to protect the product against detection by malware or a malicious employee (in case, an enterprise product is to be built on the back of this platform), it is also possible to hide any number of files)/directories in the file system.

Registry Monitor

This component intercepts almost all registry related operations such as opening , creating and deleting a key or value. It is also possible to redirect registry operations to a given subkey, thus, implementing a «registry part» of a sandbox in a similar to a file system monitor way.

Process Monitor

This component monitors creation and termination of processes in the system. It is possible to monitor loading and unloading of processes, however, there is no way to prevent a process from loading using this component. User should use File System Monitor for preventing modules from being loaded.

GUI Interface

This component is to be linked to plugins and provides an unified interface for communication with GUI engine, which is supposed to support STE GUI communication protocol. The GUI engine is required to support the interface to be compatible with plugins. It is possible to have several different GUI engines for the final product . The platform communicates with GUI using COMLAYER, an interface based on named pipes or sockets. The interaction is based on sending/receiving messages of CData structure. The only requirement the platform requires GUI engine to meet is to use COMLAYER interface for

communication and, yet, the GUI engine should provide interface.lib (or similar file) that plugins can link to on the server side. The details of communication are up to particular implementation of GUI engine. If you don't need the whole platform functionality, but only want to use sandboxing features, please refer to SHADE API.

Structures

This section describes structures used by Platform.

EVENT_NOTIFICATION

EVENT_NOTIFICATION structure documentation is preliminary and may change in the future ! Please look at attached header file for accurate structure definition and see comments on its fields in that file.

This structure is used to inform user about all events in the file system , registry or process manager and is defined as follows:

```
class EVENT_NOTIFICATION
{
public:
        unsigned long bNonPagedPoolAllocated; // user-mode code must ignore this flag
        unsigned long iComponent;
        unsigned long Operation; // OPERATION : CREATE , ERASE , FIND_FILE
        unsigned long suboperation; // WRITE, RO
        unsigned long ValType; // type of value for registry or TRUE if isDirectory

        ACCESS_MASK OriginalDesiredAccess;
        unsigned long isNameValid; // TRUE if wsFileName contains name. FALSE if file opened
by ID.
 union
{
    HANDLE pid; // PROCESS ID
    HANDLE parentPid; // parent process ID on process create notification
    HANDLE originatingPid; // PID that has initiated thread creation
    unsigned __int64 x64allignment0; //x64 compatibility issue
  };
        unsigned long  datalen;  // size of data in Data
        unsigned long actuallen; // actual length of registry data (can be greater than data
        unsigned long usMandatoryObjectNameSizeInBytes; // includes zero terminator
  union
  {
    WCHAR* wsFileName; // File or directory  name operation is performed on
    WCHAR* wsRegKeyName; // RegKey name operation is performed on
    WCHAR* wsOldName; // for rename op
    WCHAR* szChildName; // child process name
   };
  unsigned long usOptionalObjectNameSizeInBytes; // includes zero terminator
  union
  {
    WCHAR* wsFoundFileName; // currently found file name for QUERY_DIRECTORY requests
    WCHAR* wsValKeyName; // value of registry key inside the key
    WCHAR* wsNewName;
  };
```

```
    HANDLE childPid; // child process Id
    union
    {
                unsigned char  Data[MAX_DATA];
                unsigned long PostStatus;
                BOOLEAN bReplaceIfExists; // for rename
                IMAGE_LOAD_DATA imageLoadData; // for LOAD_IMAGE notification only
    };
    GUID user_mode_rule;
};
```

iComponent

Defines FSSDK component that has issued an event:

COM_FILE – Originating component is a File System.

COM_REGISTRY – Originating component is a Registry.

COM_PROCESS_NOTIFY – Originating component is Process Manager.

Operation

Defines an operation that is currently being performed by a process. The operation is component specific and can take the follwing values:

For File System component:

CREATE - The file is being created or opened.
CLOSE - The file is being closed, i.e. last opened handle is closed.
ERASE - The file is being deleted.
FIND_FILE – The directory is being enumerated.
QUERY_DIRECTORY – Same as previous, but this is a post-operation event.
RENAME – The file is being renamed.
LOAD - An executable module is being loaded, such as process or DLL module.
UNLOAD – An executable module is being unloaded.

For Registry component:

ROP_NtDeleteKey – The key is being deleted.
ROP_NtDeleteValueKey – The key value is being deleted.
ROP_NtCreateKeyEx – The key is being created or opened.
ROP_NtOpenKeyEx – The key is being opened.
ROP_NtRenameKey – The key is being renamed.
ROP_NtSetValueKey – The key value is being set.

ROP_NtCloseHandle – The key handle is being closed.
ROP_NtGeneric – Other registry operations[i].

For Process Manager component:
LOAD - A process is being loaded, such as process or DLL module.
UNLOAD – A process is being unloaded.

Suboperation

Defines suboperation that is currently being performed by a process and is component specific:

For File System component:

RO - defined for CREATE request and specifies that file is being opened for read only access.

WRITE - defined for CREATE/ERASE request and specifies that file is being opened for write access.

For Registry component:

REG_PRE – the operation is pre-operation.

REG_POST - the operation is post-operation.

For Process manager component this field is undefined.

ValType

Type of registry value such as REG_SZ. See Windows Driver Kit for complete list of possible values.

OriginalDesiredAccess

Desired access for a file operation. See ACCESS_MASK description for details.

isNameValid

Defines if a File System Component operation is being performed on a file that was opened by name.

pid

Process ID that performs this operation.

parentPid

Parent process of one that performs this operation. Defined only for Process Mangager component.

originatingPid

Currently unused.

x64allignment0

Currently unused.

wsFileName

File name in native format that the current operation is being performed on. For example, it contains file name being created for CREATE operation.

wsRegKeyName

Registry key name in native format the the current operation is being performed on.

wsOldName

Used for rename operation. An old name in native format of an object being renamed. Currently defined only for Registry component.

szChildName

Defined for process manger. The name of a child process for LOAD operation. It is not recommended to rely on this value – use process manager facilities to get process name by PID instead.

wsFoundFileName

Defined for FIND_FILE/QUERY_DIRECTORY operation. Contains file name in native format that is currently being found during enumeration process. For example, if a directory is being enumerated, this field will consequently contain every file name in that directory.

wsValKeyName

The name of a value for a given key name for registry operations.

wsNewName

New name of a registry key for rename operation. In native format.

childPid

Child process ID for an operations of Process Manager.

usMandatoryObjectNameSizeInBytes

Size of mandatory name of the object (wszFilename or wszRegKeyName).

usOptionalObjectNameSizeInBytes

Size of optional name of the object (wszFoundFileName).

IREPLY

This structure is used to reply events in the file system , registry or process manager and is defined as follows:

```
class IReply
{
public:
        virtual BOOL& Allow() = 0;
        virtual BOOL& Redirect() = 0;
        virtual ULONG& Flags() = 0;
        virtual ULONG& ErrorCode() = 0;
        virtual void SetFileName(const wchar_t* wszNewName, ULONG sizeInBytes) = 0;
        virtual const wchar_t* GetFileName()  const = 0;
        virtual const ULONG GetFileNameSize()  const = 0 ;
};
```

Allow()

If set to TRUE, current operation will be allowed. This is default value for this field. If set to FALSE, current operation will be denied. If an enumeration operation is denied, found file will be hidden.

Redirect()

if set to TRUE, a file operation will be redirected to sandbox. bAllow member must be set to TRUE in this case. The default value is FALSE.

wsFileName

Fully qualified file name in native format where the file operation is being redirected. Defined for CREATE operation only.

SetFileName()

Sets new value for wsFileName, sizeInBytes should be set to size of wszFileName.

GetFileName()

Gets current value for wsFileName.

GetFileNameSize()

Gets size of filename.

IRule

File System service allows to set up rules that describe default behavior for an operation if defined conditions are met. This is most usefull for defining sandbox (described later), when a user can define sandbox location and conditions (for example, process name to put into sandbox) when sand box is activated, rather than reacting for every particular event to implemet a sandbox by itself. The rule is defned by this structure (partially shown).

```
class IRule
{
public:
        virtual DWORD& RuleFlags() = 0;
        virtual BOOL& PermanentRule() = 0;
        virtual BOOL& InheritableRule() = 0;
        virtual BOOL& RemoveOnProcessDeath() = 0;
        virtual BOOL& Notify() = 0;
        virtual DWORD& Tag() = 0;
        virtual DWORD& ProcessId() = 0;
        virtual GUID& PluginMark() = 0;
        virtual GUID GetUniqueGuid() = 0;
        virtual DWORD& RuleAction() = 0;
        virtual unsigned long& Component() = 0;
        virtual unsigned long& operation() = 0;
        virtual unsigned long& subOperation() = 0;
        virtual void SetObjectName(int type, const wchar_t* wszObjectName, ULONG ulSize)=0;
        virtual const ULONG GetObjectNameSize(int type) const = 0 ;
        virtual const wchar_t* const GetObjectName(int type) const = 0 ;
        virtual bool IsEqual(const IRule* r2) = 0;
        virtual void Initialize(const IRule* r) = 0;
        virtual ~IRule() {}
}
```

PermanentRule()

If set to TRUE, the rule becomes permanent, i.e is being swapped to disk and therefore survives reboots and service unloadings. However, it is not recommended to use permanent rules since they become active as soon as service is loaded. And if some rule requires additional processings from your plugin side, it cannot function properly if your plugin is not loaded for some reason.

InheritableRule()

If set to TRUE, the rule is being automatically replicated and applied to child processes, spawned by a process(es) that meets this rule conditions (i.e this rule is applied to this process). Replicated rule (clone) is always non-permanent and is

scheduled for deletion as soon as child process terminates. If child process happens to spawn another process, replication procedure will be repeated recursively for that particular child.

RemoveOnProcessDeath()

If set to TRUE, the rule will be deleted as soon as process, described in the rule terminates.

Notify()

If set to true, all plugins will be notified about events that matches this rule. This is default value.

Tag()

An unique value that allows your plugin to recognize its rules. This value is not preserved for permanent rules.

ProcessId()

Process ID that this rule is applied to. Set this field to 0 if this rule must be applied to all processes in the system.

PluginMark()

An unique GUID which identifies rule as being managed by given plugin. Mark is defined by plugin. Plugins must assign the same GUID for all rules they create.

GetUniqueGuid()

An unique GUID which identifies rule.

SetObjectName()

Sets object Name of given type of given size (ulSize).

Types are

CRULE_OBJ_MANDATORY - mandatory object name – a file name, or a registry key or value name.

CRULE_OBJ_OPTIONAL - optional name. Used only in some requests, for example, for FIND_FILE it contains found file inside a directory. If it is a enumeration of %Dir%, which contains %file% , then mandatory name is %Dir% and optional name is %file% for each found file, one at a time.

CRULE_OBJ_SANDBOXROOT - Sandbox is a place on a file system or inside the registry, where operations are being redirected to. Set root folder or key for a

sandbox in this field. This field must be filled with name of a folder or key in native format.

CRULE_OBJ_IMAGE - Process name (short, for example, FAR.EXE) to which this rule must be applied. This field is an alternative to pid. Never use both pid and this field. This may lead to an unpredictable results.

wszObjectName – object name in native format.

ulSize – size of wszObjectName in bytes.

GetObjectNameSize()

Gets object name size for object of given type.

GetObjectName()

Gets object name of given type.

RuleAction()

An action that must be performed for an operation that matches this rule. Can take follwing values:

allow – an operation will be allowed.

deny – an operation will be denied.

emulate – an operation will be emulated , i.e. redirected to sandbox.

Component()

Gets/Sets Component which can be COM_FILE, COM_REGISTRY, COM_PROCESS_NOTIFY for FS/Registry/Process events (see iComponent).

Operation()

An operation such as CREATE , ERASE, FIND_FILE (See Operation).

SubOperation()

See SubOperation.

IProcessData

This interface is primarly used by file system service Process Manager. It is very unlikely that you will need this in your plugin. Flags are used internally, please do not rely on them and do not modify them. Use this interface to get process ID (Pid) and/or executable file name.

```
class IProcessData
{
public:
	virtual const wchar_t* GetFullName() const =0;
	virtual void SetFullName(const wchar_t* wszName) = 0;
	virtual DWORD GetPid() = 0;
	virtual DWORD GetParentPid() = 0;
	virtual DWORD GetFlags() = 0;
	virtual DWORD SetFlags(DWORD dwFlags) = 0;
};
```

FSProxy

FSProxy is dynamic-linked library that provides convenient way to communicate with kernel mode filtering driver. Win32 process may subscribe for notifications about file system and registry events using this library. The reminder of this section describes functions and interfaces exported by this library. Please note that only one process may connect to this library at any given time. Therefore you should use this library only if your version of FSSDK doesn't contain file system service – a special user mode Win32 service which allows third-party functionality to be loaded as plugins. File System Service is also described in this document later.

FSProxy exported functions

FSPROXY_API ISetup* Init()

This function returns interface for loading kernel mode driver and attaching to the driver.

ISetup

ISetup interface is defined as follows:

```
class ISetup
{
public:
	virtual int   setSimplifiedCallbacks(ISimplifiedCallbacks* pCallbacks) =0;
	virtual int   DoFiltering(bool bStartFiltering, int iThreadsCount) = 0;
	virtual bool  FilteringStatus() = 0;
	virtual ULONG ThreadsNumber() = 0;
};
```

setSimplifiedCallbacks()

This function attaches simplified version of above interface to the caller.

DoFiltering()

This function initiates filtering process. Set bStartFiltering to TRUE to start filtering process , or false to stop. Second parameter specifies number of threads used to process requests from the driver. Set this value to zero to let platform define optimal value of threads.

FilteringStatus()

Returns filtering status. TRUE – if filtering is started, FALSE – otherwise.

ThreadsNumber()

Returns number of threads used by the filtering engine to intercept events from file system and registry.

In order to use this interface , please follow the following scenario:

First, create a descendant class that implements ISimplifiedCallbacks interface and create an instance of this class. Then call SetSimplifiedCallbacks.Call DoFiltering() function in order to start filtering process.

ISimplifiedCallbacks

ISimplifiedCallbacks interface is defined as follows:

```
class ISimplifiedCallbacks
{
public:
	virtual int GenericCallback(EVENT_NOTIFICATION* pNotification, IREPLY* pReply);
};
```

GenericCallback()

This function is called when any event in the system has happened and there is a rule, that requires notification for user mode plugins.

File System Service

Overview

This section describes File System Windows Service that is shipped in some versions of the Kit. The service is inteded to be used by your solution via plugins – a dynamic loaded libraries, that implement specific C++ interfaces and receive notifications from the service when one or another event occurs in the operating system.

Commication Layer

The service is shipped together with communication layer – a dynamic linked library that is aimed to provide interface for communications between plugins and service itself and user mode component of your solution, that typically implement GUI facilities and other user-mode logic. The remainder of the section describes it's structures and interfaces.

CData structure

Components, that use communication layer for communicating passes their messages to each other through unified CData structure. This structure contains the message itself and peer address (if it is a TCP IP connection) or a name in case of named pipes. IP addresses are formed as "IP:port" string, where IP is an ip address in form XX.XX.XX.XX and port – is decimal port number. Even such format might also be used in case of named pipes, since it will be automatically translated into suitable for pipe representation. Note also ,that in case of using pipes – you must not provide full pipe name such as "\\.\\pipe\\Mypipe ", but only "Mypipe" part. The structure is defined as follows:

```
struct CData
{
  char szServer[40];
  union
  {
   DWORD dwCommand;
   DWORD dwErrorCode;
  };
   union
   {
     CMultiConnectSession mc_sess;
    CPluginData plugin_info;
```

```
    unsigned char UserMessage[USER_MSG_SIZE];
    wchar_t ComputerName[ (USER_MSG_SIZE / sizeof(wchar_t)) / 2];

  };
  union
  {
    DWORD dwPluginIndex;
   DWORD dwPluginCount;

 };
};
```

szServer

Name of the server in form ip:port, for example “192.168.1.1:1088” or name of named pipe in shortened form, for example “mypype”. If server name is given in form of IP and is used in pipe session, it will be automatically converted to suitable name transparently to the user. This field is meaningfull only in WaitForConnection() and EstablishConnection() calls and is ignored by other fuctions.

dwCommand

A command code that is recoginzed by your plugin.

dwErrorCode

An error code, recoginzed by your plugin. Used in answers from plugins.

CPluginData

Contains information about the plugin – it’s displayable name, unique GUID , flags and the language which, if supported, displayable name should be written in.

CMultiConnectSession

Used only for multithreaded versions of session and contains maximum allowed number of simultaneous connections , thread function pointer and an event handle which should be set to signaled to abort all waiting threads for that particular session.

UserMessage

Your messsage. Format is completely custom.

Exported Functions

The communication layers exports these functions:

COMLAYER_API ISession* CreateSession(DWORD type)

Creates a sessison of given type.

Type can have the following values:

TCP_IP_SESSION – creates a sockets based session.

PIPE_SESSION – creates a pipes based session.

TCP_IP_MULTITHEREADED_SESSION – creates a sockets based session which is able to accept several simultaneous connections.

PIPE_MULTITHREADED – creates a pipes based session which is able to accept several simultaneous connections.

COMLAYER_API VOID DeleteSession(ISession* session)

Deletes a sessison.

Exported Interface

ISession

ISession interface is defined as follows:

```
class ISession
{
public:
DWORD virtual WaitForConnection(const CData& data, bool bRestore) = 0;
DWORD virtual EstablishConnection(const CData& data, bool bRestore) = 0;
DWORD virtual RecvMessage( CData& data) = 0;
DWORD virtual SendMessage( CData& data) = 0;
DWORD virtual CloseConnection() = 0;
virtual ~ISession() {};
};
```

WaitForConnection()

This function waits for incoming connection from client. Parameters are:

- A data, filled with address of waiting host and port or with a name of pipe. An address of waiting host may be also set to 127.0.0.1 for convinience.

- Restore flag : if set to TRUE, connection will be automaically restored in case of loss.

Returns ID of new connection.

EstablishConnection()

Establishes connection from a client side. The parameters are the same, as for WaitForConnection, and server data must be filled up with a server real IP address and port or name of a pipe.

Returns an ID of established connection.

RecvMessage()

Receives message from the host. Return ID of connection. The message is received in data.

SendMessage()

Sends message to the host. Returns ID of connection.

CloseConnection()

Closes currently active connection. Returns ID of closed connection.

Plugins System

Overview

This sections describes plugins architecture – a dynamically loaded modules, that are supposed to implement business logic of applications, based on the Filewall service.

Registering plugins

Registering your plugin is relatively simple. This is done by editing config.xml file , located in Config subdirectory of installation directory.The format of the file is as follows:

<INPROC_AGENT>filename1.dll</INPROC_AGENT>

<INPROC_AGENT>filename2.dll</INPROC_AGENT>

….

<INPROC_AGENT>filenameN.dll</INPROC_AGENT>

Where filename(x).dll – is the name of plugin.

Plugins are located in the Plugins subfolder under Filewall folder.

Structures used by plugins

IPluginCallBacks

A plugin receives notifications from the service via IPluginCallback interface which is defined as follows:

```
class IPluginCallbacks
{
public:
        virtual bool __stdcall Initialize(IProcessManager* pProcessManager, IRuleManager*
        pRuleManager, IFileSystemService* pFileSystemService) = 0;
        virtual bool __stdcall onNotification(const EVENT_NOTIFICATION* pNotification,IReply*
        pReply) = 0;
        virtual void __stdcall Finalize() = 0;
        virtual void __stdcall GetPluginInfo(GUID* pPlugin_id, DWORD* pdwFlags,  wchar_t*
        szPrintableName, DWORD dwLangId ) = 0;
};
```

Initialize()

This function is called when plugin is being loaded.
The function should return true in case of successful initialization or false otherwise.

onNotification()

This function is called when there is an event on the file system or registry has occurred and there is a rule, marked by Notify() flag exists for this event.

Finalize()

This function is called when plugin is being unloaded.

GetPluginInfo()

This function may be called by plugin's client to obtatin data about the plugin. An unique plugin_id shiuld be returned, flags , printable name and language id. Flags can have the following values:

PLUGIN_WATCHER = 0;
PLUGIN_IMPLEMENTS_SANDBOX_REGISTRY = 1;
PLUGIN_IMPLEMENTS_SANDBOX_FILESYSTEM = 2;
PLUGIN_DENIES_REGISTRY = 4;
PLUGIN_DENIES_FILESYSTEM = 8;

IProcessManager

This interface is used to manage processes by plugins.

```
class IProcessManager
{
public:
   int virtual __stdcall GetProcessCount() = 0;
   TProcessData  virtual __stdcall GetProcessData(int index) = 0;
   void virtual __stdcall AddProcess(const TProcessData& data)= 0;
   void virtual __stdcall RemoveProcess(int index)=0;
   TProcessData virtual __stdcall GetProcessByPid(DWORD pid, DWORD parentPid = 0) = 0;
};
```

GetProcessCount()

This function returns number of processes currently active.

GetProcessData()

This function returns information about process.

TProcessData

This structure is defined as follows:

```
struct TProcessData
{
private:
	WCHAR* wszFullName;
public:
	TProcessData();
	~TProcessData();
	TProcessData(const wchar_t* FullName,DWORD pid , DWORD parentPid, DWORD
dwFlags);
	TProcessData(const TProcessData& r);
	const wchar_t* GetFullName() const;
	void SetFullName(const wchar_t* wszName);
	TProcessData& operator=(const TProcessData& r);
	bool operator==(const TProcessData& r);
	bool operator!=(const TProcessData& r);
	DWORD pid;
	DWORD parentPid;
	DWORD dwFlags;
};
```

GetProcessByPid()

Returns process data by process ID. Never specify parentPid.

AddProcess()

Reserved for use by platform developers.

RemoveProcess()

Reserved for use by platform developers.

IRuleManager

This interface is used to manage rules by plugins.

```
class IRuleManager
{
public:
	virtual IRule* CreateRule() = 0;
	virtual void DeleteRuleObject(IRule* r) = 0;
	virtual bool __stdcall LoadRules(const char* pRuleText = NULL) = 0;
	virtual bool __stdcall SerializeRules() = 0;
	virtual size_t __stdcall GetRuleSnapShot(void** ppSnapshot) = 0;
	virtual IRule* __stdcall GetRule(void* hSnapShot, int index) = 0;
	virtual void __stdcall FreeSnapshot(void* pSnapshot)   = 0;
```

```
        virtual BOOL __stdcall AddRule(const IRule* rule) = 0;
        virtual bool __stdcall DeleteRule(const IRule* rule) = 0;
        virtual bool __stdcall FindRuleByTag(DWORD dwTag, DWORD count, IRule* rule) = 0;
        virtual bool __stdcall FindRuleByGUID(GUID guid, IRule* rule) = 0;
        virtual bool __stdcall FindEmulationRuleByPid(DWORD pid) = 0;
        virtual bool __stdcall FindEmulationRuleByName(wchar_t* wszName) = 0;

        virtual DWORD __stdcall TagRule(IRule* r) = 0;
        virtual VOID __stdcall DropRules() =0 ;
        virtual bool CommitDelayedRulesForProcess(const wchar_t* wszProcessName, DWORD
pid) = 0;
        virtual bool __stdcall CleanupMinirulesForPid(const HANDLE hPid ) =0;
        virtual bool __stdcall RegisterNotifications(IRuleManagerCallbacks*
notifications_receiver) = 0;
        virtual bool __stdcall UnRegisterNotifications(IRuleManagerCallbacks*
notifications_receiver) = 0;
        virtual  GUID* __stdcall GetRulesForPid(DWORD pid, size_t& count) = 0;
        virtual void __stdcall DeleteReturnedGuids(GUID* p) = 0;
};
```

LoadRules()

Loads rules. Pass NULL as a parameter.

SerializeRules()

Serializes rules to disk.

GetRuleSnapshot()

Returns rule count and sets ppSnapshot to a value of type void*. Use this value to work with current rules state (snapshot).

GetRule()

Returns rule of given index for given snapshot.

FreeSnapshot()

Frees memory, occupied by given snapshot.

RegisterNotification()

Registers notification receiver. It will be called each time given rule is applied.

UnRegisterNotification()

Unregisters notification receiver.

AddRule()

Adds new rule, specified as parameter.

DeleteRule()

Removes rule, specified as parameter.

OnActivation()

Called each time when rule with given unique guid (first parameter) is applied to process with given pid. Return value is currently ignored. The rules are applied each time given process starts up.

OnDeactivation()

Same as previous function, but this one is called on rule deactivation – when given process is being terminated.

IFileSystemService

This interface is used to call directly to the service from within plugins.

```
class IFileSystemInterface
{
public:
        virtual bool __stdcall CallService(const CData& calldata, CData& reply) = 0;
        virtual HMODULE __stdcall GetPluginByGUID(const GUID* pGuid) = 0;
        virtual void __stdcall UnloadPlugin(HMODULE h) = 0;
        virtual bool IsBESTProcess(HANDLE pid) = 0;
        virtual GUID RegisterAPIInterceptor(IAPIInterceptor* pApiInterceptor) = 0;
        virtual void UnregisterAPIInterceptor(const GUID guid) = 0;
        virtual bool __stdcall CallService(const CData& calldata, CData& reply) = 0;
}
```

CallService()

Sends specific command to the service and receives a reply. Reserved for use by service developers.

Currently this interface is reserved for use by service developers. Third-party plugins should not use this method.

GetPluginByGUID()

Returns handle of loaded plugin module, specified by it's guid. This method is intended for use by plugins which rely on other installed plugins and their interfaces. By obtaining handle of a required plugin, dependant one may call to it via it's exported functions.

UnloadPlugin()

Unloads specified plugin (by handle).

IsBESTProcess()

Determines if a given process is a part of platform. (is Belongs to the Engine of Security Toolkit) Returns true if given process belongs to platform or false – otherwise.

RegisterAPIInterceptor()

Registers it's caller as an API interceptor and returns given GUID. This guid should be used when caller unregisters itself later on.

UnregisterAPIInterceptor()

Unregisters previously registered interceptor.

IApiIntercepor

This interface is used by plugins that want to intercept APIs called by particular processes or otherwise modify behavior of them.

```
class IAPIInterceptor
{
public:
        virtual bool __stdcall ProcessStartup(const wchar_t* wszModuleName, DWORD pid,
wchar_t* wszInjectedDLL) = 0;
};
```

ProcessStartup()

This method is called by platform each time new process is loaded. First parameter gets module name which is being loaded. This is usually an EXE module of process which is starting up as a first parameter, process id of process which is starting up as a second parameter and a module name (DLL) which should be loaded into process. The module should be located in System32 or WOW64System32 system folder and must its name must end on '32' or '64' for 32 and 64 bit versions of module respectively. For instance names could be 'mymodule32.dll' for 32 bit module and 'mymodule64.dll' for 64 bit module. You may specify either 32 or 64 bit version of module, such as 'mymodule32.dll'.Correct version of moduel to load will be determined automatically based on bitness of process to load module into. You may specify several modules to load separated with semicolon. Such as 'moduleA32.dll;moduleb32.dll;modulec64.dll'.

This method should return true in case of success (so that given injectee module will be loaded) or false otherwise.

Plugin exported entry point

Each plugin must declare exported function named RegisterAgent or _RegisterAgent@4 but not both which prototype is as follows:

```
typedef int ( __stdcall *CRegisterAgent)(void** pSimplifiedCallbacks);
```

This function must return non-zero value to indicate success. pSimplifiedCallbacks parameter is a pointer to pointer to IPluginCallbacks. Given pointer to must be assigned with valid instance of class that implements IPluginCallbacks interface within plugin.

SHADE Plugin API Overview

For those developers who need just high level programmatic control of Sandboxing functionality, SHADE plugin (SHADE.DLL) provides an API for this. First thing you should do is to call from your plugin GetPluginByGUID() method ofI IFileSystemService:

```
virtual HMODULE __stdcall GetPluginByGUID(const GUID*
pGuid);
```

Provide the following GUID as a parameter:

```
static const GUID sandbox_guid =
{ 0xb75dcd72, 0x75aa, 0x4173, { 0x9b, 0xe7, 0xd4, 0xf, 0x72,
0x2a, 0xc8, 0x2d } };
```

After you get the HMODULE, a handle to the DLL, you can control creation of sandboxes via provided exported functions and interfaces. The rest of this chapter describes those functions and interfaces. We will start from description of interfaces and after that, we will describe exported function which return those interfaces. Interfaces are a pure C++ interfaces.

IVirtualMachine

A sandbox is represented by IVirtualMachine interface which is declared as follows:

```
class IVirtualMachine
{
public:
    virtual bool Initialize(void) = 0;
```

```
    virtual DWORD AddExecutable(const wchar_t* wszFileName,
const wchar_t* args, DWORD dwFlags) = 0;
    virtual DWORD AddPid(DWORD pid, DWORD dwFlags) = 0;
    virtual DWORD RemoveExecutable(const wchar_t*
wszFileName) = 0;
    virtual bool IsFileSandboxed(const wchar_t* wszFileName)
= 0;
    virtual bool IsProcessSandboxed(DWORD pid) = 0;
    virtual GUID GetGUID() = 0;
    virtual BOOL GetCleanFlag()= 0;
    virtual BOOL SetCleanFlag(BOOL) = 0;
    virtual bool Serialize() = 0;
    virtual DWORD GetSandboxedFilesList(wchar_t** pList) =
0;
    virtual void FreeList(wchar_t* pList) = 0;
    virtual bool Load() = 0;
    virtual const wchar_t* GetFSRoot() = 0;
    virtual void AssignView(IViewCallbacks* view) = 0;
    virtual bool IsCleaningUp() = 0;
    virtual bool SetName(const wchar_t* wszName) = 0;
    virtual const wchar_t* GetName() = 0;
    virtual bool PrepareForDeletion() = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

IVirtualMachine methods

Initialize

Call this method after your created new virtual machine. It performs necessary initialization for virtual machine. Only after this initialization is done, the virtual machine is ready to use.

AddExecutable

This method adds executable file to the virtual machine (sandbox). When any file with the name provided is executed , the process is automatically placed into sandbox. Provide full path to filename in wszFileName parameter. Pass zeroes as all other parameters – they are reserved for internal SHADE use. Return values are defined as follows:

```
#define SANDBOX_OK 0
#define SANDBOX_DEMO_LIMITIAION 1
#define SANDBOX_WRONG_FILE 2
#define SANDBOX_ALREADY_EXISTS 3
#define SANDBOX_ERROR 255
```

AddPid

This method adds a process with given pid to sandbox. The process must be a just-started process and put into suspended state. Use Windows API CreateProcess() to create a process in suspended state and use returned pid as a parameter to this function. In case of success, “unfreeze” the process and it will run sandboxed. The dwFlags parameter must be zero. Return value is TRUE in case of success.

RemoveExecutable

This method removes given process by filename from sandbox. Return values are same as for AddExecutable() method.

IsFileSandboxed

This method returns if a process with given filename will run sandboxed.

IsProcessSandboxed

Checks if given process (by pid) is sandboxed or not.

GetGUID

Returns GUID of this virtual machine (sandbox).

GetCleanFlag

Returns if given virtual machine is scheduled for cleaning

SetCleanFlag

Pass TRUE to schedule virtual machine cleanup. SHADE will try to clean up sandbox immediately or after reboot if not possible.

Serialize

This method flushes to disk and registry sandbox state (settings)

GetSandboxedFilesList

```
Pass a pointer to wchar_t* to get an array of null-
terminated strings – filenames of sandboxed processes.
```

FreeList

Frees memory occupied by sandboxed file list, created by GetSandboxedFilesList() method.

Load

Loads sandbox settings from disk.

GetFSRoot

Returns root to sandboxed files on the disk, looks like C:\Shade\{ … } , where {..} is sandbox GUID

AssignView

This method allows you to receive notifications for certain events, like adding a file to sandbox, or removing it. You can build your GUI, responsive for given events and displaying the state of sandbox accordingly.

IsCleaningUp

Retruns true if sandbox is being cleaned up.

SetName

This method assigns a name to sandbox to be displayed by GUI. Retruns true in case of success.

GetName

Returns sandbox name.

PrepareForDeletion

When you want to completely delete sandbox, call this function before doing this. It will try to cleanup the sandbox. Returns true in case of success.

AddRef

Increases reference count to this sandbox.

Release

Decreases reference count to this sandbox.

IVMManager

This interface controls creating and managing virtual machines.

```
class IVMManager
{
public:
     virtual GUID CreateVM(GUID guid) = 0;
     virtual void DeleteVM(GUID vmGuid) = 0;
     virtual IVirtualMachine* GetVM(GUID vmGUID) =0;
     virtual GUID* GetVMS() = 0;
     virtual void DropGUIDS(GUID* pArray) =0;
     virtual size_t VMCount() = 0;
     virtual bool SelectVM(GUID guid) = 0;
     virtual GUID GetSelected() = 0;
     virtual void ProcessOrphanagedFiles() = 0;
     virtual IVirtualMachine* GetVMByName(const wchar_t*
wszName) = 0;
     virtual bool IsUniqueName(const wchar_t* wszName) = 0;
};
```

IVirtualMachine methods

CreateVM

Creates new virtual machine. Pass GUID_NULL to automatically generate GUID for new virtual machine. Returns GUID of newly created machine.

DeleteVM

Deletes VM with given GUID.

GetVM

Returns pointer to IVirtualMachine interface for VM with given GUID. This function increases reference count to IVirtualMachine pointed object. Use Release() method when you no longer need this machine to prevent memory leak.

GetVMS

Returns ponter to array of GUIDS of existing virtual machines. The array is terminated with GUID_NULL element.

VMCount

Returns number of virtual machines.

SelectVM

Selects given VM (by guid). This means that selected VM is now default. If user uses SHADE GUI, this machine is displayed when they open GUI. Returns true in case of success.

GetSelectedVM

Returns GUID of currently selected VM or GUID_NULL if none is selected.

ProcessOrphanagedFiles

For internal use. When sandbox is being cleaned up, at first, the whole folder is renamed and deleted later, the deletion could happen even after reboot. This function forces deletion of orphanaged sandboxes.

GetVMByName

Returns virtual machine by name.

IsUniqueName

Checks if given name is unique. If true, it is safe to create a VM with given name – no duplicates are guaranteed.

IViewCallbacks

You can create a class that implements this interface to receive notifications which are usefull for a custom-made UI.

```
class IViewCallbacks
{
public:
     virtual bool SBUIAddFileToUI( const wchar_t*
wszFileName, const wchar_t* cmd_args, DWORD tag, int nWindow
)=0;
     virtual bool SBUI_RemoveFileFromUI( const wchar_t*
wszFileName, int nWindow, DWORD pid /*= 0*/ )=0;
     virtual void ClearView() = 0;
};
```

IViewCallabacks methods

SBUIAddFileToUI

Called when a user adds new application to sandbox. wszFileName contains name of file. Other parameters are reserved for internal use and are not documented here. Return true.

SBUI_RemoveFileFromUI

Called when a user removes a file given in wszFileName parameter from UI. Other parameters are reserved. Return true.

ClearView

Called when the plugin requires to clean the view of UI. SHADE UI removes all icons from the view.

Exported fuctions

SHADE.DLL provides several exported functions to work with interfaces described above.

```
STDAPI_(IVMManager*) GetVMManager()
```

This function returns pointer to Virtual Machine Manager.

```
STDAPI_(int) LicenseActivate(const char* serial)
```

Activates SHADE plugin with given activation key, returns 0 in case of failure and a value > 0 in case of success.

```
STDAPI_(int) LicenseIsTrial()
```

Checks if given license is trial. Returns 0 if license is NOT trial. Returns -1 in case of failure.

```
STDAPI_(int) LicenseIsActivated()
```

Returns -1 in case of failure, 0 – if not activated, a value > 0 if activated.

```
STDAPI_(int) LicenseGetRunningMode()
```

Returns running mode (0 = home , 1 = corporate) , -1 in case of error.