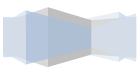
TitanEngine

SDK References



Contents

Introduction to TitanEngine

One of the greatest challenges of modern reverse engineering is taking apart and analyzing software protections. During the last decade a vast number of such shell modifiers have appeared. Software Protection as an industry has come a long way from simple encryption that protects executable and data parts to current highly sophisticated protections that are packed with tricks aiming at slow down in the reversing process. Number of such techniques increases every year. Hence we need to ask ourselves, can we keep up with the tools that we have?

Protections have evolved over the last few years, but so have the reverser's tools. Some of those tools are still in use today since they were written to solve a specific problem, or at least a part of it. Yet when it comes to writing unpackers this process hasn't evolved much. We are limited to writing our own code for every scenario in the field.

We have designed *TitanEngine* in such fashion that writing unpackers would mimic analyst's manual unpacking process. Basic set of libraries, which will later become the framework, had the functionality of the four most common tools used in the unpacking process: debugger, dumper, importer and realigner. With the guided execution and a set of callbacks these separate modules complement themselves in a manner compatible with the way any reverse engineer would use his tools of choice to unpack the file. This creates an execution timeline which parries the protection execution and gathers information from it while guided to the point from where the protection passes control to the original software code. When that point is reached file gets dumped to disk and fixed so it resembles the original to as great of a degree as possible. In this fashion problems of making static unpackers have been solved. Yet static unpacking is still important due to the fact that it will always be the most secure, and in some cases, fastest available method. That is why we will discuss both static and dynamic unpackers.. We will also see into methods of making generic code to support large number of formats without knowing the format specifics.

TitanEngine can be described as Swiss army knife for reversers. With its 250 functions, every reverser tool created to this date has been covered through its fabric. Best yet, *TitanEngine* can be automated. It is suitable for more than just file unpacking. *TitanEngine* can be used to make new tools that work with PE files. Support for both x86 and x64 systems make this framework the only framework supporting work with PE32+ files. As such, it can be used to create all known types of unpackers. Engine is open source making it open to modifications that will only ease its integration into existing solutions and would enable creation of new ones suiting different project needs.

TitanEngine SDK contains:

- Integrated x86/x64 debugger
- Integrated x86/x64 disassembler
- Integrated memory dumper
- Integrated import tracer & fixer
- Integrated relocation fixer
- Integrated file realigner
- Functions to work with TLS, Resources, Exports,…

Introduction to static unpackers

Most of basic unpackers are of the static variety. We take this observation very loosely as this depends on the complexity of the format being unpacked. In most cases writing such unpackers is easy because the format being unpacked is a simple one or more commonly referred to as a crypter.

This kind of PE file protectors (because packing is a very basic form of protection) have a simple layout that only encrypts the code and resources, and in some cases even takes the role of the import loader. Even if we encounter the most advanced representative of this shell protection type it won't differ much from its most basic protection model. Which is, no modification to the PE section layout other than adding a new section for the crypter code and encryption of the entire code and resource sections with possible import loader role for the crypter stub. Since these modifications don't impact the file in such way that major file reconstruction should be done writing static unpackers also has its general model. This is, get the needed data for decryption of the encrypter parts and reconstruction of the import table followed by removing the crypter section.

With the slight variations of the guidelines described above this could be considered as the basic crypter model. These variations could be variations in the position of the crypter code, way it handles imports and some protection shell specifics such as: protected entry point, import redirections or eliminations, code splices, code markers, etc.

However static unpackers can be used for a more difficult use cases which require the full file reconstruction in order to complete the unpacking process. In such cases static unpacking can be used and it's recommended only if the security is of the vital importance. These cases most commonly require the identification of the compression algorithm used and its adaptation to our own code. This code ripping must be done very carefully and it requires the full understanding of the algorithm which decompresses the code. There are a few standard compression algorithms in use by most PE shells so we can use this to create our own database of corresponding decompression algorithms to ease the unpacker writing process. No matter which path we choose we must always check whether or not the algorithm has changed since this is one way to tamper with the unpackers. Dynamic unpackers are resilient to such changes.

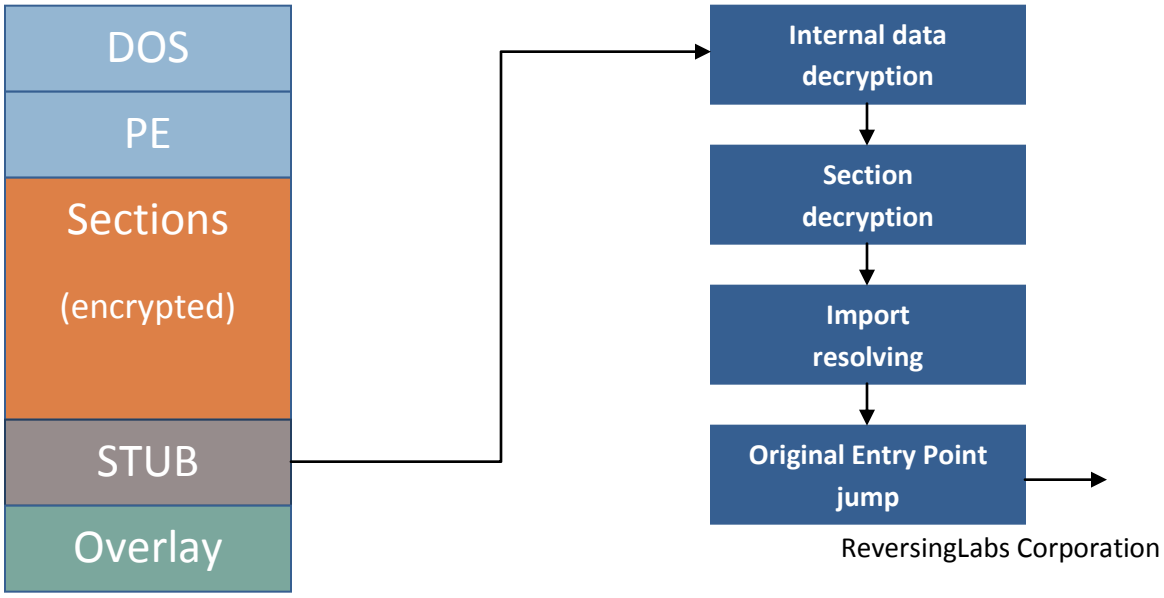


Figure (1) *Crypter file & execution layout*

Introduction to dynamic unpackers

Most common unpacker type is dynamic. This model is widely used because it is easy to implement and resilient to minor changes in packing shell decryption and decompression algorithms. But due to the fact that files do execute during unpacking process this method must be conducted with great care about system security. There is a risk of files being executed outside the unpacking process or even stuck in infinite loops. This can and must be avoided by the unpacker implementation. Most logical choice is creation of internal sandbox for the unpacking process itself.

Dynamic unpacking is used on specific shell modifier types. These types have a more complex layout and file content is commonly not only encrypted but compressed too. To avoid heavy coding to allow what is basically recompiling of the file we execute it to the original entry point which is the first instruction of the code before the file was protected. Even though all shells can be dynamically unpacked this kind of unpacking is only used on packers, protectors, bundlers and hybrids.

Basic layout of such shell modifiers includes compression of the file sections and vital data and optionally their protection by encryption. Stub is usually located in the last section and section layout before packing is either preserved or all sections are merged into one. First case usually implies that each section still contains its original data only compressed while the second one implies a physically empty section which only serves as virtual memory space reserve. In this second case compressed data is usually stored inside stub section and upon its decompression returned to original location.

When creating dynamic unpackers it is important to always keep control over executing sample. At no point this control must be left in a gray area in which we are uncertain what will occur. Furthermore unpacking must be conducted inside safe environment so software sandbox must be designed. This along with multiple checks before and during the unpacking process ensures that we retain maximum control during this risky process. This kind of unpackers has a standard model which will be described in *Dynamic unpacker layout*. That kind of unpacker is a basic unpacker type that can be created with *TitanEngine* whose largest number of functions is dedicated to writing.

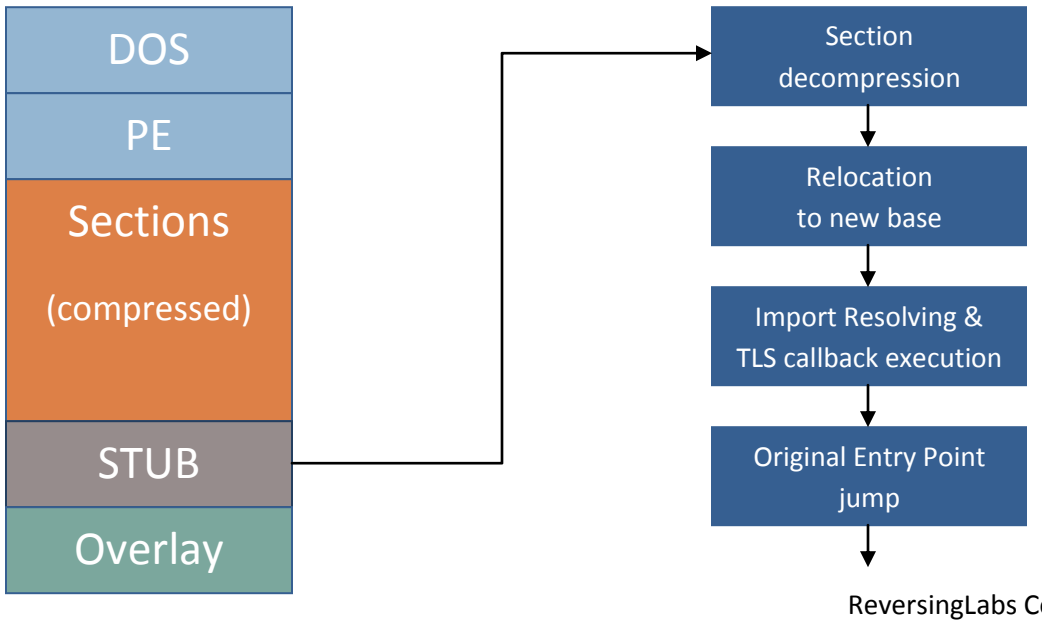


Figure (2) *Packer file & execution layout*

Introduction to generic unpackers

Most complex way of creating unpackers is creating generic unpackers. Totally opposite from the other two cases when creating generic unpackers you don't need to worry about extracting a good enough patter on code segment to create a good signature for your unpacker. Quite simply because these unpackers don't care about the shell specifics, they only care about their overall behavior which is common for shell modifiers of the same group. This means that there can never be a general generic unpacker but several wide range generic unpackers targeting specific behavior groups.

Here we will focus only on generic unpacking of packed executables and present a wide generic algorithm targeting these shell modifiers. Major challenge here is retaining as much control as possible without slowing down the unpacking process drastically. Slowdown occurs because we use memory breakpoints to monitor packed shell access to executable sections. If we reset the memory breakpoint each time the packer accesses the section we will have a major speed impact and if we don't we reset we risk not to catch the original entry point jump event and even let file execute. There are a few ways to do this but one is most common.

Generic unpackers commonly use WaitForDebugEvent timeouts to reset the breakpoints once one such breakpoint has been hit. Memory breakpoints can be hit for three reasons: when memory is read from, written to or executing. Since we only place memory breakpoints on places where we expect entry point to be we are only interested in execution case. To check whether or not that memory is executing we just check the EIP register to see if it matches our region. If it does that memory is executing. However we can set a breakpoint on the section which contains packer code. That is why we will track if the section has been written to prior its execution. If it has been written to it is highly possible that the entry point resides there and that we are at the right spot. We can track writing with memory breakpoints but one can also check that section hash just to make sure that the data has actually changed. Last check that should be performed before determining that we are sitting on a possible entry point is a check for known simple redirections that packers such as ASPack use to fool generic algorithms. Once we verify this last thing we can be fairly certain that we have found our entry point. To finish the unpacking we must dump and process imports which can be done with *TitanEngine*.

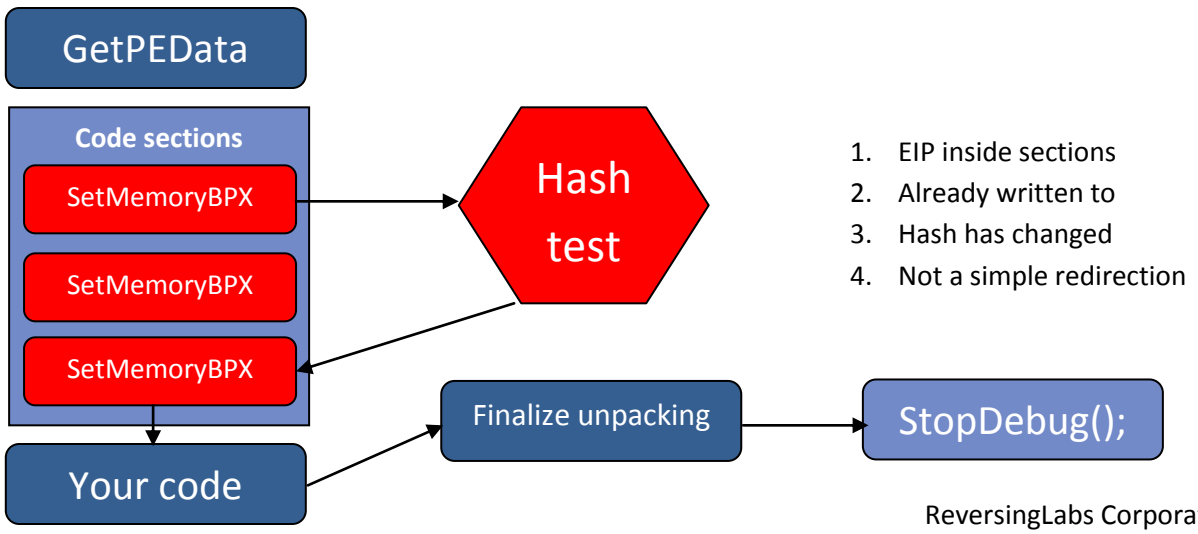


Figure (3) *Generic unpacker algorithm layout*

Dynamic unpacker layout

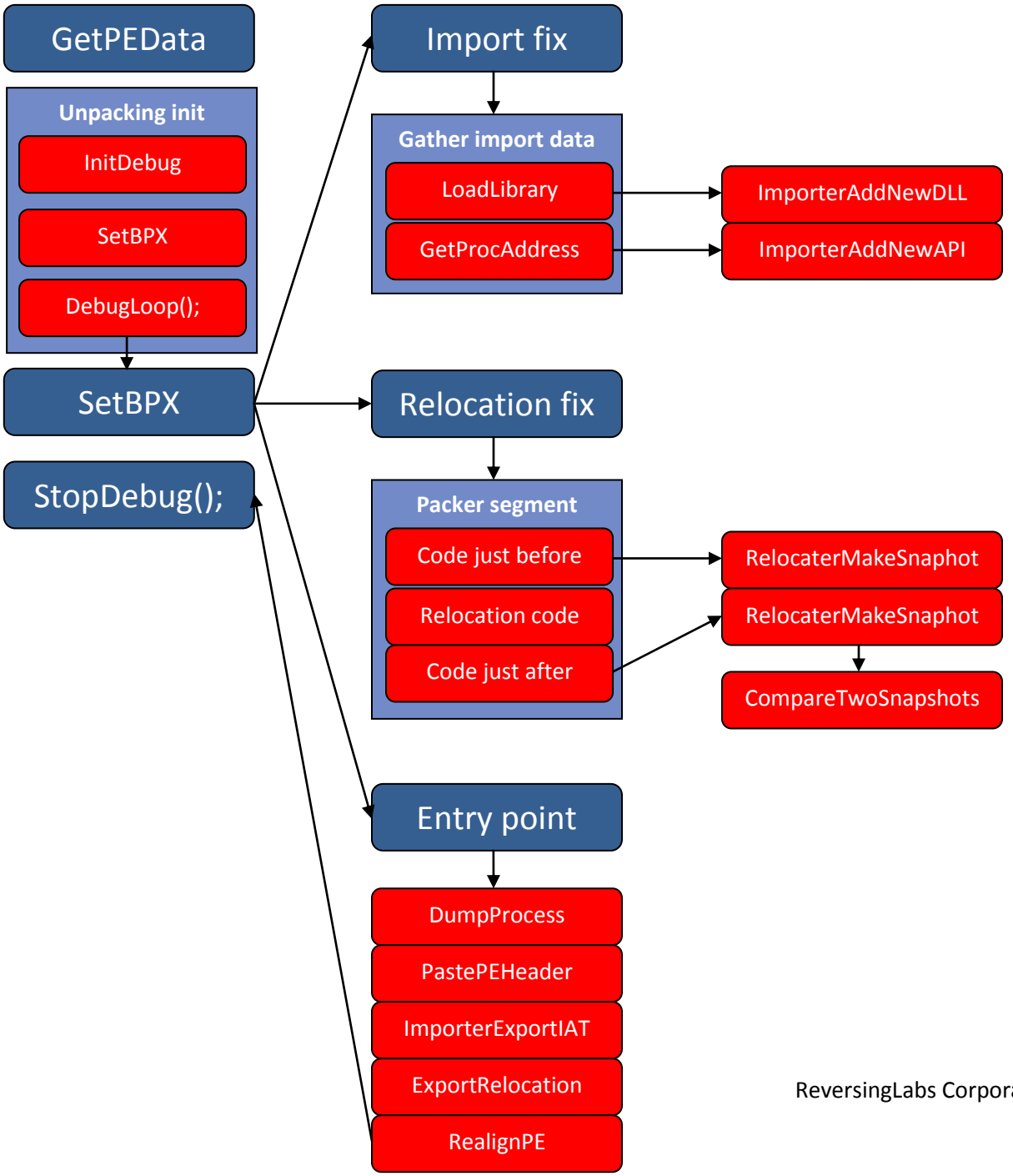
In order to use the *TitanEngine* SDK you must know the order in which APIs must be called. The order is pretty strict, and the layout of your unpacker will always be pretty much the same. Every unpacker starts with debugging and setting a breakpoint on your targets entry point. After this you must debug the program by running it until you get to the IAT filling code. This code uses LoadLibrary or GetModuleHandle API to load the dependent .dll files and GetProcAddress to find the locations of the necessary APIs. When this is done you need to break on original entry point, dump the file and paste imports to it.

To start the debugging you must first find the OEP address. To do this you can call GetPE32Data API and load the ImageBase and OriginalEntryPoint data. The sum of these two values is the address of the entry point. When this is done initialize the debugging by calling the InitDebug API. This API creates the debugged process but it does not start the actual debugging. In this suspended state call SetBpx to set the main breakpoint at OEP. This breakpoints callback will be called as soon as the debugged process finishes loading. To get the debugging process to this point you must call the DebugLoop API. After it is called the debugger takes over the debugging process. The only way to control the debugging process from this point is by callback procedures. Callbacks are set with all types of breakpoints. So if you set the breakpoint at OEP call the DebugLoop API first callback which will be called is the callback for original entry point breakpoint. Use that callback to set all the other breakpoints.

The best way to find where to set the breakpoint is by using the Find API. It will tell you the location on which your search pattern is found. If the packer for which you are writing an unpacker is single layered you can set all the breakpoints in the first or before the first callback, the one at original entry point. To get all the data needed to fix the IAT you need to set two or three breakpoints. First at LoadLibrary or GetModuleHandle API call, second at GetProcAddress call (use two if the packer calls GetProcAddress at two places for string API locating and ordinal API locating). But before you can actually call any of the importer functions in order to collect the IAT data you must first call ImporterInit API to initialize the importer.

After you do this and set the breakpoints you must code the breakpoint callbacks to get the IAT data. In LoadLibrary/GetModuleHandle callback you call ImporterAddNewDll API. One of its parameters is the string which holds the name of the .dll file which is loaded. This data location if located in a register or in a specific memory address. To get this data call the GetContextData API. If the data is a location on which the string is located and not an ordinal number you must call the ReadProcessMemory API to read the .dll name from the debugged processes. Then you can add the new .dll to importer engine. Note that once you add the .dll by calling the ImporterAddNewDll API all calls to ImporterAddNewAPI add APIs to last added .dll file. APIs are added by calling the ImporterAddNewAPI the same way you add a new .dll to importer engine. But unlike ImporterAddNewDll API here you must specify the location on which API pointer will be stored. This is a memory location on which pointer loaded with GetProcAddress API is written. After you collect all the data needed to fill in the IAT the unpacking is pretty much done.

Now you need to load the unpacked file OEP (this depends on the packer code) and dump the debugged process with DumpProcess. After this you can use the AddNewSection API to make space for the IAT which needs to be pasted to dump file. To know just how much space you need use the ImporterEstimatedSize API. Finally the IAT is exported to new section by calling the ImporterExportIAT API. Optionally you can realign the file and then you can stop the debugging process by calling the StopDebug API (if you don't stop the debugging target will keep running). This terminates the debugged process and then your program execution is resumed after the call to DebugLoop.



TitanEngine SDK References

Debugger module

Debugger module has functions designed for process debugging, disassembling, context accessing and memory manipulation.

Debugger module constants

Constants used by: **SetBPXEx** function, **SetHardwareBreakPoint** function, **DeleteHardwareBreakPoint** function, **GetContextDataEx** function, **GetContextData** function, **SetContextDataEx** function and **SetContextData** function

```
#define UE_EAX 1
#define UE_EBX 2
#define UE_ECX 3
#define UE_EDX 4
#define UE_EDI 5
#define UE_ESI 6
#define UE_EBP 7
#define UE_ESP 8
#define UE_EIP 9
#define UE_EFLAGS 10
#define UE_DR0 11
#define UE_DR1 12
#define UE_DR2 13
#define UE_DR3 14
#define UE_DR6 15
#define UE_DR7 16
#define UE_RAX 17
#define UE_RBX 18
#define UE_RCX 19
#define UE_RDX 20
#define UE_RDI 21
#define UE_RSI 22
#define UE_RBP 23
#define UE_RSP 24
#define UE_RIP 25
#define UE_RFLAGS 26
#define UE_R8 27
#define UE_R9 28
#define UE_R10 29
#define UE_R11 30
#define UE_R12 31
#define UE_R13 32
#define UE_R14 33
#define UE_R15 34
#define UE_CIP 35               // Generic, on x86 = EIP and on x64 = RIP
#define UE_CSP 36               // Generic, on x86 = ESP and on x64 = RSP
```


Constants used by: **SetBPXEx** function

```
#define UE_CMP_NOCONDITION 0
#define UE_CMP_EQUAL 1
#define UE_CMP_NOTEQUAL 2
#define UE_CMP_GREATER 3
#define UE_CMP_GREATEROREQUAL 4
#define UE_CMP_LOWER 5
#define UE_CMP_LOWEROREQUAL 6
#define UE_CMP_REG_EQUAL 7
#define UE_CMP_REG_NOTEQUAL 8
#define UE_CMP_REG_GREATER 9
#define UE_CMP_REG_GREATEROREQUAL 10
#define UE_CMP_REG_LOWER 11
#define UE_CMP_REG_LOWEROREQUAL 12
#define UE_CMP_ALWAYSFALSE 13
```

Constants used by: **SetBPX** function, **SetBPXEx** function, **SetMemoryBPX** function, **SetMemoryBPXEx** function and **SetHardwareBreakPoint** function

```
#define UE_BREAKPOINT 0
#define UE_SINGLESHOOT 1
#define UE_HARDWARE 2
#define UE_MEMORY 3
#define UE_MEMORY_READ 4
#define UE_MEMORY_WRITE 5

#define UE_HARDWARE_EXECUTE 4
#define UE_HARDWARE_WRITE 5
#define UE_HARDWARE_READWRITE 6
#define UE_HARDWARE_SIZE_1 7
#define UE_HARDWARE_SIZE_2 8
#define UE_HARDWARE_SIZE_4 9

#define UE_APISTART 0
#define UE_APIEND 1
```


Constants used by: **SetCustomHandler** function

```
#define UE_CH_BREAKPOINT 1
#define UE_CH_SINGLESTEP 2
#define UE_CH_ACCESSVIOLATION 3
#define UE_CH_ILLEGALINSTRUCTION 4
#define UE_CH_NONCONTINUABLEEXCEPTION 5
#define UE_CH_ARRAYBOUNDSEXCEPTION 6
#define UE_CH_FLOATDENORMALOPERAND 7
#define UE_CH_FLOATDEVIDEBYZERO 8
#define UE_CH_INTEGERDEVIDEBYZERO 9
#define UE_CH_INTEGEROVERFLOW 10
#define UE_CH_PRIVILEGEDINSTRUCTION 11
#define UE_CH_PAGEGUARD 12
#define UE_CH_EVERYTHINGELSE 13
#define UE_CH_CREATETHREAD 14
#define UE_CH_EXITTHREAD 15
#define UE_CH_CREATEPROCESS 16
#define UE_CH_EXITPROCESS 17
#define UE_CH_LOADDLL 18
#define UE_CH_UNLOADDLL 19
#define UE_CH_OUTPUTDEBUGSTRING 20
```

Constants used by: **RemoveAllBreakPoints** function

```
#define UE_OPTION_REMOVEALL 1
#define UE_OPTION_DISABLEALL 2
#define UE_OPTION_REMOVEALLDISABLED 3
#define UE_OPTION_REMOVEALLENABLED 4
```

Constants used by: **SetEngineVariable** function

```
#define UE_ENGINE_ALOW_MODULE_LOADING 1
#define UE_ENGINE_AUTOFIX_FORWARDERS 2
#define UE_ENGINE_PASS_ALL_EXCEPTIONS 3
#define UE_ENGINE_NO_CONSOLE_WINDOW 4
#define UE_ENGINE_BACKUP_FOR_CRITICAL_FUNCTIONS 5
```


StaticDisassembleEx function

The **StaticDisassembleEx** function is used to disassemble data from the context of the process using SDK. This is only used to disassemble instructions locally, meaning code inside your executable and its memory context.

Syntax

```
void* __stdcall StaticDisassembleEx(
            ULONG_PTR DisassmStart,
            LPVOID DisassmAddress
            );
```

Parameters

DisassmStart

[in] Used only to help with disassembling relative instructions such as jumps. This variable should be set to address which you are disassembling. If data was copied from the remote process make sure you use that address. Size of this variable varies, on x86 its 4 bytes and on x64 its 8 bytes. Therefore it can also be declared as void*.

DisassmAddress

[in] Pointer to memory which holds instruction to be disassembled. Only the first instruction in that block will be disassembled.

Return value

This function returns pointer to disassembled instruction string or *NULL* if disassemble fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

StaticDisassemble function

The **StaticDisassemble** function is used to disassemble data from the context of the process using SDK. This is only used to disassemble instructions locally, meaning code inside your executable and its memory context.

Syntax

```
void* __stdcall StaticDisassemble(
            LPVOID DisassmAddress
            );
```

Parameters

DisassmAddress

[in] Pointer to memory which holds instruction to be disassembled. Only the first instruction in that block will be disassembled. If you use this function to disassemble relative instructions such as jumps this function will assume that instructions originally reside at provided address and will disassemble them as such. In case you need to disassemble instructions that have been moved use *StaticDisassembleEx* function.

Return value

This function returns pointer to disassembled instruction string or *NULL* if disassemble fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

DisassembleEx function

The **DisassembleEx** function is used to disassemble data from the context of any running process provided that you have the access to this process.

Syntax

```
void* __stdcall DisassembleEx(
            HANDLE hProcess,
            LPVOID DisassmAddress,
            bool ReturnInstructionType
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be disassembled.

DisassmAddress

[in] Pointer to memory in remote process whose instruction will be disassembled. Only the first instruction in that block will be disassembled.

ReturnInstructionType

[in] Boolean switch which determines whether or not to return only instruction type. For example is the disassembled instruction is: MOV EAX,EBX and this switch is set to *TRUE* this function will only return MOV. If this switch is set to *FALSE* whole disassembled string will be returned.

Return value

This function returns pointer to disassembled instruction string or *NULL* if disassemble fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

Disassemble function

The **Disassemble** function is used to disassemble data from the context of currently debugged process. This function will fail if no process is being debugged or specified address doesn't exist inside debugged process.

Syntax

```
void* __stdcall Disassemble(
            LPVOID DisassmAddress,
            );
```

Parameters

DisassmAddress

[in] Pointer to memory in remote process whose instruction will be disassembled. Only the first instruction in that block will be disassembled.

Return value

This function returns pointer to disassembled instruction string or *NULL* if disassemble fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

StaticLengthDisassemble function

The **StaticLengthDisassemble** function is used to get length of disassembled instructions from the context of the process using SDK. This is only used to disassemble instructions locally, meaning code inside your executable and its memory context.

Syntax

```
void* __stdcall StaticLengthDisassemble(
            LPVOID DisassmAddress
            );
```

Parameters

DisassmAddress

[in] Pointer to memory which holds instruction whose size in bytes will be determined. Only the first instruction in that block will be disassembled.

Return value

This function returns the size of disassembled instruction string or *NULL* if disassemble fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

LengthDisassembleEx function

The **LengthDisassembleEx** function is used to get length of disassembled from the context of any running process provided that you have the access to this process.

Syntax

```
long __stdcall LengthDisassembleEx(
            HANDLE hProcess,
            LPVOID DisassmAddress
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be disassembled.

DisassmAddress

[in] Pointer to memory which holds instruction whose size in bytes will be determined. Only the first instruction in that block will be disassembled.

Return value

This function returns the size of disassembled instruction string or either *NULL* or minus one if disassembles fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

LengthDisassemble function

The **LengthDisassemble** function is used to get length of disassembled from the context currently debugged process. This function will fail if no process is being debugged or specified address doesn't exist inside debugged process.

Syntax

```
long __stdcall LengthDisassembleEx(
            LPVOID DisassmAddress
            );
```

Parameters

DisassmAddress

[in] Pointer to memory which holds instruction whose size in bytes will be determined. Only the first instruction in that block will be disassembled.

Return value

This function returns the size of disassembled instruction string or either *NULL* or minus one if disassembles fails.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

InitDebug function

The **InitDebug** function is used to initialize the debugging process. This function is always used first and it is creating a process which will be debugged.

Syntax

```
void* __stdcall InitDebug(
            char* szFileName,
            char* szCommandLine,
            char* szCurrentFolder
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be debugged.

szCommandLine

[in] Pointer to string which will be passed to created process and command line parameter.

szCurrentFolder

[in] Pointer to string which will be passed to *CreateProcess* API during the process creation.

Return value

This function returns the pointer to `PROCESS_INFORMATION` structure or *NULL* if it fails.

Remarks

None.

Example

None.

InitDebugEx function

The **InitDebugEx** function is used to initialize the debugging process. This function is always used first and it is creating a process which will be debugged.

Syntax

```
void* __stdcall InitDebug(
            char* szFileName,
            char* szCommandLine,
            char* szCurrentFolder,
            LPVOID EntryCallBack
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be debugged.

szCommandLine

[in] Pointer to string which will be passed to created process and command line parameter.

szCurrentFolder

[in] Pointer to string which will be passed to *CreateProcess* API during the process creation.

EntryCallBack

[in] Pointer to callback function which will be called when application reaches file entry point. This is equal to setting breakpoint to file entry point.

Return value

This function returns the pointer to `PROCESS_INFORMATION` structure or *NULL* if it fails.

Remarks

None.

Example

None.

InitDLLDebug function

The **InitDLLDebug** function is used to initialize the DLL debugging process. This function is always used first and it is creating a process which will be debugged and it is specialized only for debugging DLL files. All necessary DLL loaders needed to make DLL debugging work are automatically loaded.

Syntax

```
void* __stdcall InitDLLDebug(
            char* szFileName,
            bool ReserveModuleBase,
            char* szCommandLine,
            char* szCurrentFolder,
            LPVOID EntryCallBack
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file who will be debugged.

ReserveModuleBase

[in] Boolean variable which indicates whether or not to load debugged DLL on non default image base.

szCommandLine

[in] Pointer to string which will be passed to created process and command line parameter.

szCurrentFolder

[in] Pointer to string which will be passed to *CreateProcess* API during the process creation.

EntryCallBack

[in] Pointer to callback function which will be called when application reaches file entry point. This is equal to setting breakpoint to file entry point.

Return value

This function returns the pointer to `PROCESS_INFORMATION` structure or *NULL* if it fails.

Remarks

None.

AutoDebugEx function

The **AutoDebugEx** function is used to initialize the debugging process. This function is always used first and it is creating a process which will be debugged. It can be used to debug both executables and DLLs.

Syntax

```
void __stdcall AutoDebugEx(
            char* szFileName,
            bool ReserveModuleBase,
            char* szCommandLine,
            char* szCurrentFolder,
            DWORD TimeOut,
            LPVOID EntryCallBack
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be debugged.

ReserveModuleBase

[in] Boolean variable which indicates whether or not to load debugged DLL on non default image base.

szCommandLine

[in] Pointer to string which will be passed to created process and command line parameter.

szCurrentFolder

[in] Pointer to string which will be passed to *CreateProcess* API during the process creation.

TimeOut

[in] Value which will be passed to Windows *WaitForDebugEvent* API.

EntryCallBack

[in] Pointer to callback function which will be called when application reaches file entry point. This is equal to setting breakpoint to file entry point.

Remarks

Since debugging is performed in second thread all calls that you may address to *SendMessage* Windows API will cause the program to freeze. Keep this in mind while using this function.

AttachDebugger function

The **AttachDebugger** function is used to initialize the debugging process by attaching to already running process. Your program will not return from the call to *AttachDebugger* before the debugging is finished and debugged application has been terminated.

Syntax

```
void __stdcall AttachDebugger(
            DWORD ProcessId,
            bool KillOnExit,
            LPVOID DebugInfo,
            LPVOID CallBack
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

KillOnExit

[in] Boolean variable which indicates whether or not to terminate the process we attach to after the debugging has been completed.

DebugInfo

[out] Pointer to `PROCESS_INFORMATION` structure which will be filled when the process attaching completes.

CallBack

[in] Pointer to callback function which will be called when application reaches the first system breakpoint.

Return value

None.

Remarks

None.

DetachDebugger function

The **DetachDebugger** function is used to detach debugger from debugged process. After using this function process will be detached from the debugger and continue to run on its own. Because of this it is vital that all exceptions are processed before detaching. This can't be done from any *TitanEngine* callback and that is why this function should NOT be used. For detaching please use expert version of this function.

Syntax

```
bool __stdcall DetachDebugger(
            DWORD ProcessId
            );
```

Parameters

ProcessId

[in] Process ID of the debugged process which can be acquired with Windows API or internal structures.

Return value

Returns *TRUE* on successful detaching and *FALSE* on error.

Remarks

This function only works on Window 2003/XP and later.

Example

None.

DetachDebuggerEx function

The **DetachDebuggerEx** function is used to detach debugger from debugged process. After using this function process will be detached from the debugger and continue to run on its own. For detaching please always use this function.

Syntax

```
bool __stdcall DetachDebuggerEx(
            DWORD ProcessId
            );
```

Parameters

ProcessId

[in] Process ID of the debugged process which can be acquired with Windows API or internal structures.

Return value

Returns *TRUE* on successful detaching and *FALSE* on error.

Remarks

This function only works on Window 2003/XP and later.

Example

None.

GetProcessInformation function

The **GetProcessInformation** function is used to retrieve the pointer to initialization data for the debugged process.

Syntax

```
void* __stdcall GetProcessInformation();
```

Parameters

None.

Return value

Returns pointer to `PROCESS_INFORMATION` structure and therefore can be declared also as `LPPROCESS_INFORMATION`.

Remarks

Process must be active to retrieve this data.

Example

None.

GetStartupInformation function

The **GetStartupInformation** function is used to retrieve the pointer to initialization data for the debugged process.

Syntax

```
void* __stdcall GetStartupInformation();
```

Parameters

None.

Return value

Returns pointer to `STARTUPINFOA` structure and therefore can also be declared as `LPSTARTUPINFOA`.

Remarks

Process must be active to retrieve this data.

Example

None.

GetDebuggedDLLBaseAddress function

The **GetDebuggedDLLBaseAddress** function is used to retrieve base address on which the debugged DLL file is loaded. Due to loading on different base addresses because of that base being reserved for some other module or ASLR file is not always loaded on its default image base which is why when debugging DLL files this function is crucial.

Syntax

```
long long __stdcall GetDebuggedDLLBaseAddress();
```

Parameters

None.

Return value

This function returns pointer the loaded base on which debugged DLL file resides. Therefore it can also be declared as void*.

Remarks

Process must be active and debugging DLL to retrieve this data.

Example

None.

GetDebuggedFileBaseAddress function

The **GetDebuggedFileBaseAddress** function is used to retrieve base address on which the debugged file is loaded. Due to loading on different base addresses because of that base being reserved for some other module or ASLR file is not always loaded on its default image base which is why when debugging files this function is crucial. This function will retrieve the base address of the main module so in case that you are debugging DLL file this function will return base address of the DLL loader.

Syntax

```
long long __stdcall GetDebuggedDLLBaseAddress();
```

Parameters

None.

Return value

This function returns pointer the loaded base on which debugged file resides. Therefore it can also be declared as void*.

Remarks

Process must be active to retrieve this data.

Example

None.

GetExitCode function

The **GetExitCode** function is used to retrieve the process exit code.

Syntax

```
long __stdcall GetExitCode();
```

Parameters

None.

Return value

This function returns exit code provided by the debugged file on its termination.

Remarks

Process must be terminated to retrieve this data.

Example

None.

DebugLoop function

The **DebugLoop** function is used to start the debugging process. This function is always used after debugger initialization. Before running the debugged process make sure you have at least one breakpoint set otherwise the application will run. Your program will not return from the call to DebugLoop before the debugging is finished and debugged application has been terminated.

Syntax

```
void __stdcall DebugLoop();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

DebugLoopEx function

The **DebugLoopEx** function is used to start the debugging process. This function is always used after debugger initialization. Before running the debugged process make sure you have at least one breakpoint set otherwise the application will run. Your program will not return from the call to DebugLoopEx before the debugging is finished and debugged application has been terminated.

Syntax

```
void __stdcall DebugLoopEx(
            DWORD TimeOut
            );
```

Parameters

TimeOut

[in] Value which will be passed to Windows *WaitForDebugEvent* API.

Return value

None.

Remarks

None.

Example

None.

SetDebugLoopTimeOut function

The **SetDebugLoopTimeOut** function is used to set debug timeout at runtime. This value is passed to WaitForDebugEvent Windows API.

Syntax

```
void __stdcall SetDebugLoopTimeOut(
            DWORD TimeOut
            );
```

Parameters

TimeOut

[in] Value which will be passed to Windows *WaitForDebugEvent* API.

Return value

None.

Remarks

None.

Example

None.

SetNextDbgContinueStatus function

The **SetNextDbgContinueStatus** function is used to set the parameter passed to *ContinueDebugEvent* Windows API.

Syntax

```
void __stdcall SetNextDbgContinueStatus(
            DWORD SetDbgCode
            );
```

Parameters

SetDbgCode

[in] One of two values: `DBG_CONTINUE` or `DBG_EXCEPTION_NOT_HANDLED`.

Return value

None.

Remarks

Using this function can break the debugging process making the debugged target exit the debugging loop and terminating the application or showing an error message on Windows Vista and later. Use this function with caution.

Example

None.

StopDebug function

The **StopDebug** function is used to stop the debugging process. Debugged process is terminated just after the call to this function and program control is passed to first command right after the DebugLoop which started the debugging process.

Syntax

```
void __stdcall StopDebug();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ForceClose function

The **ForceClose** function is used to stop the debugging process. Debugged process is terminated just after the call to this function and program control is passed to first command right after the DebugLoop which started the debugging process. This is a more safe way to terminate debugging in case that your code has crashed with an exception.

Syntax

```
void __stdcall ForceClose();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

SetBPX function

The **SetBPX** function is used to set INT 3 breakpoints. Type of the INT3 breakpoint can be set with *SetBPXOptions* function.

Syntax

```
bool __stdcall SetBPX(
            ULONG_PTR bpxAddress,
            DWORD bpxType,
            LPVOID bpxCallBack
            );
```

Parameters

bpxAddress

[in] Address on which breakpoint will be set.

bpxType

[in] Type of the breakpoint to be set. Either `UE_BREAKPOINT` or `UE_SINGLESHOOT`.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

Boolean switch indicating whether or not the breakpoint was set.

Remarks

None.

Example

None.

SetBPXEx function

The **SetBPXEx** function is used to set INT 3 breakpoints. Type of the INT3 breakpoint can be set with *SetBPXOptions* function.

Syntax

```
bool __stdcall SetBPXEx(
            ULONG_PTR bpxAddress,
            DWORD bpxType,
            DWORD NumberOfExecution,
            DWORD CmpRegister,
            DWORD CmpCondition,
            ULONG_PTR CmpValue,
            LPVOID bpxCallBack,
            LPVOID bpxCompareCallBack,
            LPVOID bpxRemoveCallBack
            );
```

Parameters

bpxAddress

[in] Address on which breakpoint will be set.

bpxType

[in] Type of the breakpoint to be set. Either `UE_BREAKPOINT` or `UE_SINGLESHOOT`.

NumberOfExecutions

[in] Set maximum number of breakpoint executions. If this value is *NULL* then breakpoint has no execution limit.

CmpRegister

[in] Register which will be check if breakpoint execution condition is meet.

CmpCondition

[in] Type of comparing to be done with the register. See *Debugger module constants* for details.

CmpValue

[in] Value used for comparison. If condition has been meet breakpoint callback will be executed. See *Debugger module constants* for details.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

bpxCompareCallBack

[in] Reserved, always set to *NULL*.

bpxRemoveCallBack

[in] This callback will be called one breakpoint has been removed.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

Boolean switch indicating whether or not the breakpoint was set.

Remarks

All callbacks have the same declaration.

Example

None.

EnableBPX function

The **EnableBPX** function is used to enable currently disabled INT 3 breakpoints. This function can't be used to enable memory or hardware breakpoints, and should only be used after *DisableBPX*.

Syntax

```
bool __stdcall EnableBPX(
            ULONG_PTR bpxAddress
            );
```

Parameters

bpxAddress

[in] Address of already set breakpoint which will be enabled if currently disabled.

Return value

Boolean switch indicating whether or not the breakpoint was enabled.

Remarks

None.

Example

None.

DisableBPX function

The **DisableBPX** function is used to disable currently enabled or active INT 3 breakpoints. This function can't be used to disable memory or hardware breakpoints.

Syntax

```
bool __stdcall DisableBPX(
            ULONG_PTR bpxAddress
            );
```

Parameters

bpxAddress

[in] Address of already set breakpoint which will be disabled if currently enabled.

Return value

Boolean switch indicating whether or not the breakpoint was disabled.

Remarks

None.

Example

None.

IsBPXEnabled function

The **IsBPXEnabled** function is used whether or not the INT3 breakpoint is enabled. This function can't be used to check state of memory or hardware breakpoints.

Syntax

```
bool __stdcall DisableBPX(
            ULONG_PTR bpxAddress
            );
```

Parameters

bpxAddress

[in] Address of already set breakpoint which will be queried for active state.

Return value

Boolean switch indicating whether or not the breakpoint is enabled.

Remarks

None.

Example

None.

DeleteBPX function

The **DeleteBPX** function is used to remove set INT3 breakpoints. This function can't be used to remove memory or hardware breakpoints.

Syntax

```
bool __stdcall DeleteBPX(
            ULONG_PTR bpxAddress
            );
```

Parameters

bpxAddress

[in] Address of already set breakpoint which will be removed.

Return value

Boolean switch indicating whether or not the breakpoint is removed.

Remarks

If breakpoint has been set with *SetBPXEx* and it has a remove callback it will be called once this function is called.

Example

None.

SafeDeleteBPX function

The **SafeDeleteBPX** function is used to remove set INT3 breakpoints. This function can't be used to remove memory or hardware breakpoints. There is no need to use this function as it is now considered as an alias to *DeleteBPX* and preserved only for compatibility with earlier versions of *TitanEngine* SDK.

Syntax

```
bool __stdcall SafeDeleteBPX(
            ULONG_PTR bpxAddress
            );
```

Parameters

bpxAddress

[in] Address of already set breakpoint which will be removed.

Return value

Boolean switch indicating whether or not the breakpoint is removed.

Remarks

If breakpoint has been set with *SetBPXEx* and it has a remove callback it will be called once this function is called.

Example

None.

SetAPIBreakPoint function

The **SetAPIBreakPoint** function is used to set INT 3 breakpoints at exported functions of any loaded DLL file not just the system ones.

Syntax

```
bool __stdcall SetAPIBreakPoint(
            char* szDLLName,
            char* szAPIName,
            DWORD bpxType,
            DWORD bpxPlace,
            LPVOID bpxCallBack
            );
```

Parameters

szDLLName

[in] Pointer to string which is the name of the DLL which has the function on which breakpoint will be set, for example ***kernel32.dll***.

szAPIName

[in] Pointer to string which is the name of the API on which breakpoint will be set, for example ***VirtualAlloc***.

bpxType

[in] Type of the breakpoint to be set. Either `UE_BREAKPOINT` or `UE_SINGLESHOOT`.

bpxPlace

[in] Place where the breakpoint will be set. Use `UE_APISTART` or `UE_APIEND`. Here start is the first instruction of that API and end the last which is always *RET* and its variations.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

Boolean switch indicating whether or not the breakpoint was set.

Remarks

None.

DeleteAPIBreakPoint function

The **DeleteAPIBreakPoint** function is used to remove set INT3 breakpoints from functions inside DLL files.

Syntax

```
bool __stdcall DeleteAPIBreakPoint(
            char* szDLLName,
            char* szAPIName,
            DWORD bpxPlace
            );
```

Parameters

szDLLName

[in] Pointer to string which is the name of the DLL which has the function on which breakpoint was set, for example ***kernel32.dll***.

szAPIName

[in] Pointer to string which is the name of the API on which breakpoint was set, for example ***VirtualAlloc***.

bpxPlace

[in] Place where the breakpoint was set. Use `UE_APISTART` or `UE_APIEND`. Here start is the first instruction of that API and end the last which is always *RET* and its variations.

Return value

Boolean switch indicating whether or not the breakpoint is removed.

Remarks

None.

Example

None.

SafeDeleteAPIBreakPoint function

The **SafeDeleteAPIBreakPoint** function is used to remove set INT3 breakpoints from functions inside DLL files. There is no need to use this function as it is now considered as an alias to *DeleteAPIBreakPoint* and preserved only for compatibility with earlier versions of *TitanEngine* SDK.

Syntax

```
bool __stdcall SafeDeleteAPIBreakPoint(
            char* szDLLName,
            char* szAPIName,
            DWORD bpxPlace
            );
```

Parameters

szDLLName

[in] Pointer to string which is the name of the DLL which has the function on which breakpoint was set, for example ***kernel32.dll***.

szAPIName

[in] Pointer to string which is the name of the API on which breakpoint was set, for example ***VirtualAlloc***.

bpxPlace

[in] Place where the breakpoint was set. Use `UE_APISTART` or `UE_APIEND`. Here start is the first instruction of that API and end the last which is always *RET* and its variations.

Return value

Boolean switch indicating whether or not the breakpoint is removed.

Remarks

None.

Example

None.

SetMemoryBPX function

The **SetMemoryBPX** function is used to set memory breakpoints. These breakpoints are set by *PAGE_GUARD* to targeted memory region. Even if your breakpoint size is lesser then one page whole page will be affected by the *PAGE_GUARD* and therefore it is possible that that page will be hit and that your breakpoint will not be handled. Keep this in mind when setting this kind of breakpoints and always set your breakpoints to the whole pages.

Syntax

```
bool __stdcall SetMemoryBPX(
            ULONG_PTR MemoryStart,
            DWORD SizeOfMemory,
            LPVOID bpxCallBack
            );
```

Parameters

MemoryStart

[in] Address on which breakpoint will be set. Ideally this is equal to page start.

SizeOfMemory

[in] Size of the memory to be affected by *PAGE_GUARD*.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

Boolean switch indicating whether or not the breakpoint was set.

Remarks

None.

Example

None.

SetMemoryBPXEx function

The **SetMemoryBPXEx** function is used to set memory breakpoints. These breakpoints are set by *PAGE_GUARD* to targeted memory region. Even if your breakpoint size is lesser then one page whole page will be affected by the *PAGE_GUARD* and therefore it is possible that that page will be hit and that your breakpoint will not be handled. Keep this in mind when setting this kind of breakpoints and always set your breakpoints to the whole pages.

Syntax

```
bool __stdcall SetMemoryBPXEx(
            ULONG_PTR MemoryStart,
            DWORD SizeOfMemory,
            DWORD BreakPointType,
            bool RestoreOnHit,
            LPVOID bpxCallBack
            );
```

Parameters

MemoryStart

[in] Address on which breakpoint will be set. Ideally this is equal to page start.

SizeOfMemory

[in] Size of the memory to be affected by *PAGE_GUARD*.

BreakPointType

[in] Defines type of memory breakpoint. Depending on the usage this can be either `UE_MEMORY` or `UE_MEMORY_READ` or `UE_MEMORY_WRITE`. Here the first type defines any access to that memory.

RestoreOnHit

[in] Indicates whether or not to restore the breakpoint once it is executed. Due to the nature of memory breakpoints they are always one shoot unless this option is used.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

Boolean switch indicating whether or not the breakpoint was set.

RemoveMemoryBPX function

The **RemoveMemoryBPX** function is used to remove set memory breakpoints. This function only removes *PAGE_GUARD* from page protection flags.

Syntax

```
bool __stdcall RemoveMemoryBPX(
            ULONG_PTR MemoryStart,
            DWORD SizeOfMemory
            );
```

Parameters

MemoryStart

[in] Address on which breakpoint was set. Ideally this is equal to page start.

SizeOfMemory

[in] Size of the memory to be affected by *PAGE_GUARD*.

Return value

Boolean switch indicating whether or not the breakpoint is removed.

Remarks

None.

Example

None.

SetHardwareBreakPoint function

The **SetHardwareBreakPoint** function is used to set hardware breakpoints. These breakpoints can only be set on CPUs that support them.

Syntax

```
bool __stdcall SetHardwareBreakPoint(
            ULONG_PTR bpxAddress,
            DWORD IndexOfRegister,
            DWORD bpxType,
            DWORD bpxSize,
            LPVOID bpxCallBack
            );
```

Parameters

bpxAddress

[in] Address on which breakpoint will be set.

IndexOfRegister

[in] Register which will be used to set the breakpoint DR0 – DR3. If no register is specified first available and free register will be used.

bpxType

[in] Type of the breakpoint to be set. `UE_HARDWARE_EXECUTE`, `UE_HARDWARE_WRITE` or `UE_HARDWARE_READWRITE`. First type sets breakpoint on that memory execution while other two set breakpoint on memory access.

bpxSize

[in] Size of the breakpoint to be set. `UE_HARDWARE_SIZE_1`, `UE_HARDWARE_SIZE_2` or `UE_HARDWARE_SIZE_4` indicating the size in bytes affected by the breakpoint.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

Boolean switch indicating whether or not the breakpoint was set.

Remarks

None.

DeleteHardwareBreakPoint function

The **DeleteHardwareBreakPoint** function is used to remove set hardware breakpoints.

Syntax

```
bool __stdcall DeleteHardwareBreakPoint(
            DWORD IndexOfRegister
            );
```

Parameters

IndexOfRegister

[in] Register with which breakpoint was set, from DR0 to DR3.

Return value

Boolean switch indicating whether or not the breakpoint is removed.

Remarks

None.

Example

None.

RemoveAllBreakPoints function

The **RemoveAllBreakPoints** function is used to remove all breakpoints matching the selected option.

Syntax

```
bool __stdcall CurrentExceptionNumber(
            DWORD RemoveOption
            );
```

Parameters

RemoveOption

[in] One of the following:

- `UE_OPTION_REMOVEALL`, removes all breakpoints.
- `UE_OPTION_DISABLEALL`, disables all breakpoints excluding hardware ones.
- `UE_OPTION_REMOVEALLDISABLED`, removes all disabled INT3 breakpoints.
- `UE_OPTION_REMOVEALLENABLED`, removes all active INT3 breakpoints.

Return value

Boolean switch indicating whether or not the breakpoints were removed.

Remarks

None.

Example

None.

ClearExceptionNumber function

The **ClearExceptionNumber** function is used to reset the number of exceptions that have been registered and processed until that point in execution. Counter is reset to zero.

Syntax

```
long __stdcall ClearExceptionNumber();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

GetDebugData function

The **GetDebugData** function is used to retrieve the pointer to internal `DEBUG_EVENT` structure.

Syntax

```
void* __stdcall GetDebugData();
```

Parameters

None.

Return value

Pointer to `DEBUG_EVENT` structure. Therefore this functions return can also be declared as `LPDEBUG_EVENT`.

Remarks

Process must be active for this function to work.

Example

None.

GetTerminationData function

The **GetTerminationData** function is used to retrieve the pointer to internal `DEBUG_EVENT` structure. Similar to *GetDebugData* but used only when the process has terminated its execution.

Syntax

```
void* __stdcall GetTerminationData();
```

Parameters

None.

Return value

Pointer to `DEBUG_EVENT` structure. Therefore this functions return can also be declared as `LPDEBUG_EVENT`.

Remarks

Function can only be used for terminated processes.

Example

None.

GetContextDataEx function

The **GetContextDataEx** function is used to retrieve data from context of any debugged process thread. Handles of all active threads can be got by using engine thread handling functions.

Syntax

```
long long __stdcall GetContextDataEx(
            HANDLE hActiveThread,
            DWORD IndexOfRegister
            );
```

Parameters

hActiveThread

[in] Handle of the open thread from which context will be read.

IndexOfRegister

[in] Indicator on which register will be retrieved from the context. See *Debugger module constants* for details.

Return value

This function returns the requested data.

Remarks

None.

Example

None.

GetContextData function

The **GetContextData** function is used to retrieve data from context of currently active debugged process thread. Here currently active refers to the thread that debugger is currently processing, more specifically one that has generated the last exception or debugging timeout. For more specific thread handling use expert version of this function.

Syntax

```
long long __stdcall GetContextData(
            DWORD IndexOfRegister
            );
```

Parameters

IndexOfRegister

[in] Indicator on which register will be retrieved from the context. See *Debugger module constants* for details.

Return value

This function returns the requested data.

Remarks

None.

Example

None.

SetContextDataEx function

The **SetContextDataEx** function is used to set data to context of any debugged process thread. This function modifies context of the thread so be careful using it since corruption of certain register can lead to application crash.

Syntax

```
bool __stdcall SetContextDataEx(
            HANDLE hActiveThread,
            DWORD IndexOfRegister,
            ULONG_PTR NewRegisterValue
            );
```

Parameters

hActiveThread

[in] Handle of the open thread from which context will be set.

IndexOfRegister

[in] Indicator on which context register will be modified. See *Debugger module constants* for details.

NewRegisterValue

[in] This variables value will be used to set selected register state.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

SetContextData function

The **SetContextData** function is used to set data to context of currently active debugged process thread. Here currently active refers to the thread that debugger is currently processing, more specifically one that has generated the last exception or debugging timeout. For more specific thread handling use expert version of this function. This function modifies context of the thread so be careful using it since corruption of certain register can lead to application crash.

Syntax

```
bool __stdcall SetContextData(
            DWORD IndexOfRegister,
            ULONG_PTR NewRegisterValue
            );
```

Parameters

IndexOfRegister

[in] Indicator on which context register will be modified. See *Debugger module constants* for details.

NewRegisterValue

[in] This variables value will be used to set selected register state.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

StepInto function

The **StepInto** function is used to trace code by single stepping into calls. This tracing function sets trap flag and calls your callback once that trap flag has been activated. This assures execution of just one instruction and can also be used to trace through the execution of every instruction.

Syntax

```
void __stdcall StepInto(
            LPVOID StepCallBack
            );
```

Parameters

StepCallBack

[in] Address of a callback which will be called once trap flag has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

None.

Remarks

None.

Example

None.

StepOver function

The **StepOver** function is used to trace code by single stepping over calls. This tracing function sets INT3 breakpoint after the call which is used to call your callback. There is no guarantee that code execution will return from that call and that your callback will ever be called. Set breakpoint is a single shoot one and therefore it will be removed once your callback has finished.

Syntax

```
void __stdcall StepOver(
            LPVOID StepCallBack
            );
```

Parameters

StepCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

None.

Remarks

None.

Example

None.

SingleStep function

The **SingleStep** function is used to trace code by single stepping over instructions. This tracing function sets trap flag and calls your callback once that trap flag has been activated. This assures execution of just one instruction and can also be used to trace through the execution of every instruction.

Syntax

```
void __stdcall SingleStep(
            DWORD StepCount,
            LPVOID StepCallBack
            );
```

Parameters

StepCount

[in] Number of instructions to execute with tracing. Your callback will be called each time an instruction executes.

StepCallBack

[in] Address of a callback which will be called once for each instruction that executes.

CallBack definition

```
typedef void(__stdcall *cbBreakPoint)(void);
```

Return value

None.

Remarks

None.

Example

None.

FindEx function

The **FindEx** function is used to search the process memory for binary patterns.

Syntax

```
long long __stdcall FindEx(
            HANDLE hProcess,
            LPVOID MemoryStart,
            DWORD MemorySize,
            LPVOID SearchPattern,
            DWORD PatternSize,
            LPBYTE WildCard
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be searched.

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the search.

MemorySize

[in] Size of the memory which will be searched for byte pattern.

SearchPattern

[in] Pointer to sequence of bytes which represent the search pattern.

PatternSize

[in] Size of the search pattern in bytes.

WildCard

[in] Pointer to wild card byte which will be ignored during search. This wild card is equal to search asterisk "?" and those bytes inside the search pattern will always be considered as found. Usually this byte is *NULL*.

Return value

Function returns pointer to first byte of the found pattern inside the remote process. Therefore it can also be declared as void*, or *NULL* if byte pattern is not found.

Remarks

None.

Find function

The **Find** function is used to search the process memory for binary patterns. This function always searches the memory of the currently debugger process.

Syntax

```
long long __stdcall Find(
            LPVOID MemoryStart,
            DWORD MemorySize,
            LPVOID SearchPattern,
            DWORD PatternSize,
            LPBYTE WildCard
            );
```

Parameters

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the search.

MemorySize

[in] Size of the memory which will be searched for byte pattern.

SearchPattern

[in] Pointer to sequence of bytes which represent the search pattern.

PatternSize

[in] Size of the search pattern in bytes.

WildCard

[in] Pointer to wild card byte which will be ignored during search. This wild card is equal to search asterisk "?" and those bytes inside the search pattern will always be considered as found. Usually this byte is *NULL*.

Return value

Function returns pointer to first byte of the found pattern inside the remote process. Therefore it can also be declared as void*, or *NULL* if byte pattern is not found.

Remarks

None.

FillEx function

The **FillEx** function is used to fill the part of the memory with a single byte by its repetition. Most commonly this function is used to *NOP* the parts of the code or to zero out memory regions.

Syntax

```
bool __stdcall FillEx(
            HANDLE hProcess,
            LPVOID MemoryStart,
            DWORD MemorySize,
            PBYTE FillByte
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be patched.

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the filling.

MemorySize

[in] Size of the memory which will be filled with selected byte.

FillByte

[in] Pointer to byte which will be used for memory filling.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

Fill function

The **Fill** function is used to fill the part of the memory with a single byte by its repetition. Most commonly this function is used to *NOP* the parts of the code or to zero out memory regions. Process which will be patched with this function is always the currently debugged process.

Syntax

```
bool __stdcall Fill(
            LPVOID MemoryStart,
            DWORD MemorySize,
            PBYTE FillByte
            );
```

Parameters

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the filling.

MemorySize

[in] Size of the memory which will be filled with selected byte.

FillByte

[in] Pointer to byte which will be used for memory filling.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

PatchEx function

The **PatchEx** function is used to fill the part of the memory with a selected byte pattern. This byte pattern is copied to selected memory and the bytes non-affected by the patch are optionally NOPed or left unmodified.

Syntax

```
bool __stdcall PatchEx(
            HANDLE hProcess,
            LPVOID MemoryStart,
            DWORD MemorySize,
            LPVOID ReplacePattern,
            DWORD ReplaceSize,
            bool AppendNOP,
            bool PrependNOP
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be patched.

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the patch.

MemorySize

[in] Size of the memory which will be patched with selected byte pattern.

ReplacePattern

[in] Pointer to sequence of bytes which will be written to targeted memory.

ReplaceSize

[in] Size of the replace pattern.

AppendNOP

[in] If the patch size is lesser then targeted memory size patching NOPs can be appended to patch bytes to make that memory execution safe.

PrependNOP

[in] If the patch size is lesser then targeted memory size patching NOPs can be prepend to patch bytes to make that memory execution safe.

Return value

Boolean switch indicating whether or not the function was successful.

Patch function

The **Patch** function is used to fill the part of the memory with a selected byte pattern. This byte pattern is copied to selected memory and the bytes non-affected by the patch are optionally NOPed or left unmodified. Process which will be patched with this function is always the currently debugged process.

Syntax

```
bool __stdcall Patch(
            LPVOID MemoryStart,
            DWORD MemorySize,
            LPVOID ReplacePattern,
            DWORD ReplaceSize,
            bool AppendNOP,
            bool PrependNOP
            );
```

Parameters

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the patch.

MemorySize

[in] Size of the memory which will be patched with selected byte pattern.

ReplacePattern

[in] Pointer to sequence of bytes which will be written to targeted memory.

ReplaceSize

[in] Size of the replace pattern.

AppendNOP

[in] If the patch size is lesser then targeted memory size patching NOPs can be appended to patch bytes to make that memory execution safe.

PrependNOP

[in] If the patch size is lesser then targeted memory size patching NOPs can be prepend to patch bytes to make that memory execution safe.

Return value

Boolean switch indicating whether or not the function was successful.

ReplaceEx function

The **ReplaceEx** function is used to fill the part of the memory with a selected byte pattern. This byte pattern is copied to selected memory and the bytes non-affected by the patch are optionally NOPed or left unmodified.

Syntax

```
bool __stdcall ReplaceEx(
            HANDLE hProcess,
            LPVOID MemoryStart,
            DWORD MemorySize,
            LPVOID SearchPattern,
            DWORD PatternSize,
            DWORD NumberOfRepetitions,
            LPVOID ReplacePattern,
            DWORD ReplaceSize,
            PBYTE WildCard
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be patched.

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the search.

MemorySize

[in] Size of the memory which will be searched for the byte pattern.

SearchPattern

[in] Pointer to sequence of bytes which represent the search pattern.

NumberOfRepetitions

[in] Maximum number of patterns which will be replaced.

PatternSize

[in] Size of the search pattern in bytes.

ReplacePattern

[in] Pointer to sequence of bytes which will be written to targeted memory.

ReplaceSize

[in] Size of the replace pattern.

WildCard

[in] Pointer to wild card byte which will be ignored during search and replace. This wild card is equal to search asterisk "?" and those bytes inside the search pattern will always be considered as found. Usually this byte is *NULL*.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

This function finds the search patterns in selected part of the process memory and replaces them with the replace pattern. Here both search and replace patterns can have wild card bytes which will be ignored during search and replace.

Example

None.

Replace function

The **Replace** function is used to fill the part of the memory with a selected byte pattern. This byte pattern is copied to selected memory and the bytes non-affected by the patch are optionally NOPed or left unmodified. Process which will be patched with this function is always the currently debugged process.

Syntax

```
bool __stdcall Replace(
            LPVOID MemoryStart,
            DWORD MemorySize,
            LPVOID SearchPattern,
            DWORD PatternSize,
            DWORD NumberOfRepetitions,
            LPVOID ReplacePattern,
            DWORD ReplaceSize,
            PBYTE WildCard
            );
```

Parameters

MemoryStart

[in] Pointer to memory in remote process which is used as a start point for the search.

MemorySize

[in] Size of the memory which will be searched for the byte pattern.

SearchPattern

[in] Pointer to sequence of bytes which represent the search pattern.

NumberOfRepetitions

[in] Maximum number of patterns which will be replaced.

PatternSize

[in] Size of the search pattern in bytes.

ReplacePattern

[in] Pointer to sequence of bytes which will be written to targeted memory.

ReplaceSize

[in] Size of the replace pattern.

WildCard

[in] Pointer to wild card byte which will be ignored during search and replace. This wild card is equal to search asterisk “?” and those bytes inside the search pattern will always be considered as found. Usually this byte is *NULL*.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

This function finds the search patterns in selected part of the process memory and replaces them with the replace pattern. Here both search and replace patterns can have wild card bytes which will be ignored during search and replace.

Example

None.

GetRemoteString function

The **GetRemoteString** function is used to retrieve the string from the remote process. This function can read both ASCII and UNICODE strings.

Syntax

```
bool __stdcall GetRemoteString(
            HANDLE hProcess,
            LPVOID StringAddress,
            LPVOID StringStorage,
            int MaximumStringSize
            );
```

Parameters

hProcess

[in] Handle of the process from which the string will be read.

StringAddress

[in] Pointer to string in remote process which will be copied to selected memory.

StringStorage

[out] Pointer to memory location inside your code which will receive the remote string content.

MaximumStringSize

[in] Size of the local memory buffer reserved for reading the remote string.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

Always copies the maximum available bytes but keeping in mind the bounds imposed by the maximum size of the string.

Example

None.

GetFunctionParameter function

The **GetFunctionParameter** function is used to retrieve the input parameter from standard function types. To be able to do this program execution must be paused at one of two places in the function, either at the first instruction of the call or last instruction inside the call.

Syntax

```
long long __stdcall GetFunctionParameter(
            HANDLE hProcess,
            DWORD FunctionType,
            DWORD ParameterNumber,
            DWORD ParameterType
            );
```

Parameters

hProcess

[in] Handle of the process from which the function parameter will be read. Debugged process handle should always be used.

FunctionType

[in] Can be one of the following:

- `UE_FUNCTION_STDCALL`, EIP/RIP at first instruction inside the call.
- `UE_FUNCTION_CCALL`, EIP/RIP at first instruction inside the call.
- `UE_FUNCTION_FASTCALL`, EIP/RIP at first instruction inside the call.
- `UE_FUNCTION_STDCALL_RET`, EIP/RIP at last instruction of the call (RET).
- `UE_FUNCTION_CCALL_RET`, EIP/RIP at last instruction of the call (RET).
- `UE_FUNCTION_FASTCALL_RET`, EIP/RIP at last instruction of the call (RET).
- `UE_FUNCTION_STDCALL_CALL`, reserved for future use.
- `UE_FUNCTION_FASTCALL_CALL`, reserved for future use.

ParameterNumber

[in] Number if the input parameter whose value will be returned.

ParameterType

[in] Can be one of the following: `UE_PARAMETER_BYTE`, `UE_PARAMETER_WORD`, `UE_PARAMETER_DWORD`, `UE_PARAMETER_QWORD`, `UE_PARAMETER_PTR_BYTE`, `UE_PARAMETER_PTR_WORD`, `UE_PARAMETER_PTR_DWORD`, `UE_PARAMETER_PTR_QWORD`, `UE_PARAMETER_STRING` or `UE_PARAMETER_UNICODE`.

Return value

Returns the requested value or a pointer to string depending on the parameter type.

Remarks

Maximum length of the read string is 512 characters.
Stack is acquired for the currently paused thread inside the debugged process.
If parameter type is *PTR* then this function will return the data the pointer is pointing to. So if you use *PARAMETER_PTR_DWORD* it will return the *DWORD* on which pointer from that parameter number is pointing at.

Example:

```
/*7630B86F*/  MOV EDI,EDI                          ;<- EIP at VirtualAlloc
/*7630B871*/  PUSH EBP
/*7630B872*/  MOV EBP,ESP
/*7630B874*/  PUSH DWORD PTR SS:[EBP+14]
/*7630B877*/  PUSH DWORD PTR SS:[EBP+10]
/*7630B87A*/  PUSH DWORD PTR SS:[EBP+C]
/*7630B87D*/  PUSH DWORD PTR SS:[EBP+8]
/*7630B880*/  PUSH -1
/*7630B882*/  CALL kernel32.VirtualAllocEx
/*7630B887*/  POP EBP
/*7630B888*/  RET 10
```

Stack:

```
0012FA6C   004015A0  /CALL to VirtualAlloc from 0040159B
0012FA70   00000000  |Address = NULL
0012FA74   00100000  |Size = 100000 (1048576.)
0012FA78   00002000  |AllocationType = MEM_RESERVE
0012FA7C   00000001  \Protect = PAGE_NOACCESS
```

If we need call *GetFunctionParameter* to return second input parameter of VirtualAlloc function we will need to call it like this:

GetFunctionParameter(hProcess, UE_FUNCTION_STDCALL , 2, UE_PARAMETER_DWORD);

and it will, in this case, return 0x00100000. If the EIP was at the RET instruction this function would be called like this:

GetFunctionParameter(hProcess, UE_FUNCTION_STDCALL_RET , 2, UE_PARAMETER_DWORD);

and it would return the same value.

GetJumpDestinationEx function

The **GetJumpDestinationEx** function is used to calculate jump destination for the selected jump or call instruction type.

Syntax

```
long long __stdcall GetJumpDestinationEx(
            HANDLE hProcess,
            ULONG_PTR InstructionAddress,
            bool JustJumps
            );
```

Parameters

hProcess

[in] Handle of the process in which the jump or call resides.

InstructionAddress

[in] Address on which the jump or call is located.

JustJumps

[in] Boolean switch which indicates whether or not to get jump destinations for calls or just jumps.

Return value

Returns the address targeted by jump/call or *NULL* if the instruction on selected address isn't jump or call.

Remarks

None.

Example

None.

GetJumpDestination function

The **GetJumpDestination** function is used to calculate jump destination for the selected jump or call instruction type.

Syntax

```
long long __stdcall GetJumpDestination(
            HANDLE hProcess,
            ULONG_PTR InstructionAddress
            );
```

Parameters

hProcess

[in] Handle of the process in which the jump or call resides.

InstructionAddress

[in] Address on which the jump or call is located.

Return value

Returns the address targeted by jump/call or *NULL* if the instruction on selected address isn't jump or call.

Remarks

Function calls *GetJumpDestinationEx* with *JustJumps* parameter set to *FALSE*.

Example

None.

IsJumpGoingToExecuteEx function

The **IsJumpGoingToExecuteEx** function is used to check if the targeted jump is going to execute or not.

Syntax

```
bool __stdcall IsJumpGoingToExecuteEx(
            HANDLE hProcess,
            HANDLE hThread,
            ULONG_PTR InstructionAddress,
            ULONG_PTR RegFlags
            );
```

Parameters

hProcess

[in] Handle of the process in which the jump resides.

hThread

[in] Handle of the thread from which EFLAGS/RFLAGS will be read.

InstructionAddress

[in] Address on which the jump is located. Optional parameter, if it is not specified instruction at EIP/RIP will be targeted.

RegFlags

[in] Used to override current EFLAGS/RFLAGS. Used only if EIP/RIP isn't at targeted instruction. Optional parameter, if not specified EFLAGS/RFLAGS will be read from the specified thread.

Return value

Returns *TRUE* if jump would execute if execution continues or *FALSE* if not.

Remarks

None.

Example

None.

IsJumpGoingToExecute function

The **IsJumpGoingToExecute** function is used to check if the targeted jump is going to execute or not.

Syntax

```
bool __stdcall IsJumpGoingToExecute();
```

Parameters

None.

Return value

Returns *TRUE* if jump would execute if execution continues or *FALSE* if not.

Remarks

Function assumes currently debugged process and currently active thread executing jump at current EIP/RIP.

Example

None.

SetCustomHandler function

The **SetCustomHandler** function is used to set the way debugger handles specific exception codes. You can set handling of most common generated error via built-in definitions or you can handle all exceptions and filter only ones of interest to you.

Syntax

```
void __stdcall SetCustomHandler(
            DWORD ExceptionId,
            LPVOID CallBack
            );
```

Parameters

ExceptionId

[in] Exception identifier, exact code and alias can be found at *Debugger module constants*.

CallBack

[in] Pointer to callback function which will be called when application encounters that specific exception.

CallBack definition

```
typedef void(__stdcall *cbCustomHandler)(void* ExceptionData);
```

Return value

None.

Remarks

See below for *ExceptionData* callback details.

Example

None.

SetCustomHandler CallBack details

`UE_CH_BREAKPOINT`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_SINGLESTEP`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_ACCESSVIOLATION`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_ILLEGALINSTRUCTION`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_NONCONTINUABLEEXCEPTION`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_ARRAYBOUNDSEXCEPTION`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_FLOATDENORMALOPERAND`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_FLOATDEVIDEBYZERO`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_INTEGERDEVIDEBYZERO`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_INTEGEROVERFLOW`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_PRIVILEGEDINSTRUCTION`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_PAGEGUARD`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_EVERYTHINGELSE`

ExceptionData points to: `&DBGEvent.u.Exception.ExceptionRecord`

`UE_CH_CREATETHREAD`

ExceptionData points to: `&DBGEvent.u.CreateThread`

`UE_CH_EXITTHREAD`

ExceptionData points to: `&DBGEvent.u.ExitThread`

`UE_CH_CREATEPROCESS`

ExceptionData points to: `&DBGEvent.u.CreateProcessInfo`

`UE_CH_EXITPROCESS`

ExceptionData points to: `&DBGEvent.u.ExitProcess`

`UE_CH_LOADDLL`

ExceptionData points to: `&DBGEvent.u.LoadDll`

`UE_CH_UNLOADDLL`

ExceptionData points to: `&DBGEvent.u.UnloadDll`

`UE_CH_OUTPUTDEBUGSTRING`

ExceptionData points to: `&DBGEvent.u.DebugString`

HideDebugger function

The **HideDebugger** function is used to hide the debugger from being detected by various detection tricks.

Syntax

```
bool __stdcall HideDebugger(
            HANDLE hThread,
            HANDLE hProcess,
            DWORD PatchAPILevel
            );
```

Parameters

hProcess

[in] Handle of the debugged process.

hThread

[in] Handle of the thread whose context will be modified.

PatchAPILevel

[in] Reserved for future use.

Return value

Returns *TRUE* if the debugger is now hidden or *FALSE* if there were errors.

Remarks

This function works only for x86 operating systems at the moment.

Example

None.

SetEngineVariable function

The **SetEngineVariable** function is used to set various global settings for the *TitanEngine* SDK.

Syntax

```
void __stdcall SetEngineVariable(
            DWORD VariableId,
            bool VariableSet
            );
```

Parameters

VariableId

[in] Can be one of the following:

- UE_ENGINE_ALOW_MODULE_LOADING
- UE_ENGINE_AUTOFIX_FORWARDERS
- UE_ENGINE_PASS_ALL_EXCEPTIONS
- UE_ENGINE_NO_CONSOLE_WINDOW
- UE_ENGINE_BACKUP_FOR_CRITICAL_FUNCTIONS

VariableSet

[in] Boolean value which will be set to the selected option.

Return value

None.

Remarks

None.

Example

None.

Threader module

Threader module has functions designed for working with threads. It covers aspects of thread identification, thread manipulation and remote thread injection.

Threader module structures

Structures used by: **ThreaderGetThreadInfo** function and **ThreaderEnumThreadInfo** function

```
typedef struct{
      HANDLE hThread;
      DWORD dwThreadId;
      void* ThreadStartAddress;
      void* ThreadLocalBase;
}THREAD_ITEM_DATA, *PTHREAD_ITEM_DATA;
```


ThreaderGetThreadInfo function

The **ThreaderGetThreadInfo** function retrieves the information about the existing threads inside the debugged process. Data is collected each time new thread is created or any of the existing ones terminates.

Syntax

```
void* __stdcall ThreaderGetThreadInfo(
            HANDLE hThread,
            DWORD ThreadId
            );
```

Parameters

hThread

[in] Handle of the thread whose info will be returned.

ThreadIs

[in] ID of thread whose info will be returned.

Return value

This function returns pointer to `THREAD_ITEM_DATA` structure or *NULL* if thread is no longer active or not found.

Remarks

Only one of two input parameters is needed.

Example

None.

ThreaderGetThreadData function

The **ThreaderGetThreadData** function retrieves the pointer to array of `THREAD_ITEM_DATA` entries who hold the information about the existing threads. Last item of the array has *hThread* item in the structure set to *NULL*. Number of items in the array is the number of existing threads inside the debugged process. Size of this array isn't stored anywhere and must be determined on the fly.

Syntax

```
void* __stdcall ThreaderGetThreadData();
```

Parameters

None.

Return value

This function returns pointer to `THREAD_ITEM_DATA` structure array.

Remarks

None.

Example

None.

ThreaderEnumThreadInfo function

The **ThreaderEnumThreadInfo** function enumerates data about existing threads inside the debugged process. Data is collected each time new thread is created or any of the existing ones terminates.

Syntax

```
void __stdcall ThreaderEnumThreadInfo(
            void* EnumCallBack
            );
```

Parameters

EnumCallBack

[in] Pointer to callback function which will process all thread data one by one.

CallBack definition

```
typedef void(__stdcall *fEnumCallBack)(LPVOID fThreadDetail);

// fThreadDetail is a pointer to THREAD_ITEM_DATA structure
```

Return value

None.

Remarks

None.

Example

None.

ThreaderPauseThread function

The **ThreaderPauseThread** function changes the state of any active thread from active to suspend.

Syntax

```
bool __stdcall ThreaderPauseThread(
            HANDLE hThread
            );
```

Parameters

hThread

[in] Handle of the thread which will be paused.

Return value

This function returns *TRUE* if thread gets paused or *FALSE* if its execution can't be currently paused.

Remarks

None.

Example

None.

ThreaderResumeThread function

The **ThreaderResumeThread** function resumes execution of any currently paused thread inside the debugged process.

Syntax

```
bool __stdcall ThreaderResumeThread(
            HANDLE hThread
            );
```

Parameters

hThread

[in] Handle of the thread whose execution will be resumed.

Return value

This function returns *TRUE* if thread gets resumed or *FALSE* if its execution can't be currently resumed.

Remarks

None.

Example

None.

ThreaderTerminateThread function

The **ThreaderTerminateThread** function tries to terminate existing thread inside the debugged process.

Syntax

```
bool __stdcall ThreaderTerminateThread(
            HANDLE hThread,
            DWORD ThreadExitCode
            );
```

Parameters

hThread

[in] Handle of the thread which will be terminated.

ThreadExitCode

[in] Exit code which will be passed to *TerminateThread* API used to terminate the thread.

Return value

This function returns *TRUE* if thread gets terminated or *FALSE* if thread cannot be terminated.

Remarks

None.

Example

None.

ThreaderPauseAllThreads function

The **ThreaderPauseAllThreads** function is used to pause all running threads inside the debugged process optionally leaving the main thread running.

Syntax

```
bool __stdcall ThreaderPauseAllThreads(
            bool LeaveMainRunning
            );
```

Parameters

LeaveMainRunning

[in] Boolean switch indicating whether or not to leave the main thread running or not.

Return value

This function returns *TRUE* if all the threads get paused or *FALSE* if there are no threads running.

Remarks

None.

Example

None.

ThreaderResumeAllThreads function

The **ThreaderResumeAllThreads** function is used to resume all paused threads from the debugged process optionally leaving the main thread paused.

Syntax

```
bool __stdcall ThreaderResumeAllThreads(
            bool LeaveMainPaused
            );
```

Parameters

LeaveMainPaused

[in] Boolean switch indicating whether or not to leave the main thread paused or not.

Return value

This function returns *TRUE* if all the threads get resumed or *FALSE* if there are no threads running.

Remarks

None.

Example

None.

ThreaderPauseProcess function

The **ThreaderPauseProcess** function is used to pause all active threads inside the debugged process making that process suspended.

Syntax

```
bool __stdcall ThreaderPauseProcess();
```

Parameters

None.

Return value

This function returns *TRUE* if all threads get paused or *FALSE* if there are some threads still running.

Remarks

None.

Example

None.

ThreaderResumeProcess function

The **ThreaderResumeProcess** function is used to resume all paused threads inside the debugged process.

Syntax

```
bool __stdcall ThreaderResumeProcess();
```

Parameters

None.

Return value

This function returns *TRUE* if all threads get resumed or *FALSE* if there are no threads inside the debugged process.

Remarks

None.

Example

None.

ThreaderIsThreadStillRunning function

The **ThreaderIsThreadStillRunning** function is used to check if the selected thread still exists regardless of its state inside the debugged process.

Syntax

```
bool __stdcall ThreaderIsThreadStillRunning(
            HANDLE hThread
            );
```

Parameters

hThread

[in] Handle of the thread whose existence will be checked.

Return value

This function returns *TRUE* if the thread exists and FALSE if it has terminated.

Remarks

None.

Example

None.

ThreaderIsThreadActive function

The **ThreaderIsThreadActive** function is used to check if the selected thread is active and running inside the debugged process.

Syntax

```
bool __stdcall ThreaderIsThreadActive(
            HANDLE hThread
            );
```

Parameters

hThread

[in] Handle of the thread whose execution state will be queried.

Return value

This function returns *TRUE* if the thread is running and FALSE if it has terminated or it is suspended.

Remarks

None.

Example

None.

ThreaderIsAnyThreadActive function

The **ThreaderIsAnyThreadActive** function is used to check if any of the existing threads is active and running.

Syntax

```
bool __stdcall ThreaderIsAnyThreadActive();
```

Parameters

None.

Return value

This function returns *TRUE* if any of the threads is running and FALSE if all threads are suspended.

Remarks

None.

Example

None.

ThreaderIsExceptionInMainThread function

The **ThreaderIsExceptionInMainThread** function is used to if the last exception occurred inside the main debugged process thread.

Syntax

```
bool __stdcall ThreaderIsExceptionInMainThread();
```

Parameters

None.

Return value

This function returns *TRUE* if the last exception occurred inside the main thread and *FALSE* if it occurred in one of the other running threads.

Remarks

None.

Example

None.

ThreaderGetOpenHandleForThread function

The **ThreaderGetOpenHandleForThread** function is used resolve the existing open handle for ID to the same selected thread.

Syntax

```
long long __stdcall ThreaderGetOpenHandleForThread(
            DWORD ThreadId
            );
```

Parameters

ThreadId

[in] ID of the active thread returned from thread data enumeration or *Windows* API.

Return value

This function returns handle to the selected thread or NULL if the selected thread don't exist anymore.

Remarks

None.

Example

None.

ThreaderSetCallBackForNextExitThreadEvent function

The **ThreaderSetCallBackForNextExitThreadEvent** function is used set a custom callback which will be called next time one of the active threads terminates.

Syntax

```
void __stdcall ThreaderSetCallBackForNextExitThreadEvent(
            LPVOID exitThreadCallBack
            );
```

Parameters

exitThreadCallBack

[in] Pointer to callback function which will be called when the next active thread terminates.

CallBack definition

```
typedef void(__stdcall *fCustomHandler)(void* SpecialDBG);

// Here SpecialDBG is defined as a pointer to &DBGEvent.u.ExitThread
```

Return value

None.

Remarks

None.

Example

None.

ThreaderCreateRemoteThreadEx function

The **ThreaderCreateRemoteThreadEx** function is used to create a new thread inside the targeted process.

Syntax

```
long long __stdcall ThreaderCreateRemoteThreadEx(
            HANDLE hProcess,
            ULONG_PTR ThreadStartAddress,
            bool AutoCloseTheHandle,
            LPVOID ThreadPassParameter,
            LPDWORD ThreadId
            );
```

Parameters

hProcess

[in] Handle of the process in which the new thread will be created.

ThreadStartAddress

[in] Start address for the new thread located in the remote process.

AutoCloseTheHandle

[in] Boolean switch indicating whether or not to close the handle to remote thread automatically.

ThreadPassParameter

[in] Parameter which will be passed to newly created thread.

ThreadId

[in] Pointer to DWORD which will receive the ID for the newly created thread.

Return value

This function returns handle for the new thread or *NULL* if the thread wasn't created or *AutoCloseTheHandle* was set to *TRUE*.

Remarks

None.

Example

None.

ThreaderCreateRemoteThread function

The **ThreaderCreateRemoteThread** function is used to create a new thread inside the currently debugged process.

Syntax

```
long long __stdcall ThreaderCreateRemoteThreadEx(
            ULONG_PTR ThreadStartAddress,
            bool AutoCloseTheHandle,
            LPVOID ThreadPassParameter,
            LPDWORD ThreadId
            );
```

Parameters

ThreadStartAddress

[in] Start address for the new thread located in the remote process.

AutoCloseTheHandle

[in] Boolean switch indicating whether or not to close the handle to remote thread automatically.

ThreadPassParameter

[in] Parameter which will be passed to newly created thread.

ThreadId

[in] Pointer to DWORD which will receive the ID for the newly created thread.

Return value

This function returns handle for the new thread or *NULL* if the thread wasn't created or *AutoCloseTheHandle* was set to *TRUE*.

Remarks

None.

Example

None.

ThreaderInjectAndExecuteCodeEx function

The **ThreaderInjectAndExecuteCodeEx** function is used to create a new thread inside the targeted process and auto execute the injected code.

Syntax

```
bool __stdcall ThreaderInjectAndExecuteCodeEx(
            HANDLE hProcess,
            LPVOID InjectCode,
            DWORD StartDelta,
            DWORD InjectSize
            );
```

Parameters

hProcess

[in] Handle of the process in which the new thread will be created.

InjectedCode

[in] Pointer to data which will be injected in the remote process.

StartDelta

[in] Start address of the new thread will be increased by this value. Use this option if you need to execute the code from any other point other then the first byte of the *InjectedCode* memory.

InjectedSize

[in] Size of the memory which will be injected inside the remote process.

Return value

This function returns *TRUE* if the thread has been created and *FALSE* if there were problems.

Remarks

Before creating new thread data is allocated in targeted process and written there.

Example

None.

ThreaderInjectAndExecuteCode function

The **ThreaderInjectAndExecuteCode** function is used to create a new thread inside the debugged process and auto execute the injected code.

Syntax

```
bool __stdcall ThreaderInjectAndExecuteCode(
            LPVOID InjectCode,
            DWORD StartDelta,
            DWORD InjectSize
            );
```

Parameters

InjectedCode

[in] Pointer to data which will be injected in the remote process.

StartDelta

[in] Start address of the new thread will be increased by this value. Use this option if you need to execute the code from any other point other then the first byte of the *InjectedCode* memory.

InjectedSize

[in] Size of the memory which will be injected inside the remote process.

Return value

This function returns *TRUE* if the thread has been created and *FALSE* if there were problems.

Remarks

Before creating new thread data is allocated in targeted process and written there.

Example

None.

ThreaderExecuteOnlyInjectedThreads function

The **ThreaderExecuteOnlyInjectedThreads** function is used to pause all active non injected threads inside the debugged process making that process suspended. All threads that get injected after using this function will be executed normally. Once all injected threads finish their execution process execution must be resumed with *ThreaderResumeProcess*.

Syntax

```
bool __stdcall ThreaderExecuteOnlyInjectedThreads();
```

Parameters

None.

Return value

This function returns *TRUE* if all non injected threads get paused or *FALSE* if there are some threads still running.

Remarks

None.

Example

None.

TLS module

TLS module has functions designed for working with thread local storage both on disk and in memory.

TLSBreakOnCallBack function

The **TLSBreakOnCallBack** function is used to set a breakpoint on all TLS callbacks inside the PE header.

Syntax

```
bool __stdcall TLSBreakOnCallBack(
            LPVOID ArrayOfCallBacks,
            DWORD NumberOfCallBacks,
            LPVOID bpxCallBack
            );
```

Parameters

ArrayOfCallBacks

[in] Pointer to array of callbacks on which the breakpoints will be set.

NumberOfCallBacks

[in] Number of callbacks in the provided array.

bpxCallBack

[in] Address of a callback which will be called once each TLS breakpoint has been hit.

Return value

This function returns *TRUE* if the breakpoint has been set and *FALSE* if breakpoint cannot be set.

Remarks

None.

Example

None.

TLSBreakOnCallBackEx function

The **TLSBreakOnCallBackEx** function is used to set a breakpoint on all TLS callbacks inside the PE header.

Syntax

```
bool __stdcall TLSBreakOnCallBackEx(
            char* szFileName,
            LPVOID bpxCallBack
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be debugged.

bpxCallBack

[in] Address of a callback which will be called once each TLS breakpoint has been hit.

Return value

This function returns *TRUE* if the breakpoint has been set and *FALSE* if breakpoint cannot be set.

Remarks

None.

Example

None.

TLSGrabCallBackData function

The **TLSGrabCallBackData** function is used to set a breakpoint on all TLS callbacks inside the PE header.

Syntax

```
bool __stdcall TLSGrabCallBackData(
            char* szFileName,
            LPVOID ArrayOfCallBacks,
            LPDWORD NumberOfCallBacks
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose TLS callback data will be read and copied to selected array.

ArrayOfCallBacks

[out] Pointer to array which will receive the callback addresses.

NumberOfCallBacks

[out] Number of callbacks in the TLS callback array.

Return value

This function returns *TRUE* if the breakpoint has been set and *FALSE* if breakpoint cannot be set.

Remarks

None.

Example

None.

TLSRemoveCallback function

The **TLSRemoveCallback** function is used to remove TLS callbacks from the PE header of the selected file.

Syntax

```
bool __stdcall TLSRemoveCallback(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose TLS callback table will be removed.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

TLSRemoveTable function

The **TLSRemoveTable** function is used to remove TLS table from the PE header of the selected file.

Syntax

```
bool __stdcall TLSRemoveTable(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose TLS table will be removed.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

TLSBackupData function

The **TLSBackupData** function is used to make an internal backup of the TLS table so that it can be restored at runtime if it gets corrupted.

Syntax

```
bool __stdcall TLSBackupData(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose TLS table will be backed up.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

TLSRestoreData function

The **TLSRestoreData** function is used to restore data from internal backup of the TLS table directly to running process memory. In case of TLS table corruption this function can be used to restore the previously backed up data.

Syntax

```
bool __stdcall TLSRestoreData();
```

Parameters

None.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

TLSBuildNewTable function

The **TLSBuildNewTable** function is used to build and store completely new TLS table inside the selected PE file. This option can be used to create a new TLS table in case of dealing with protections that use TLS elimination protection technique.

Syntax

```
bool __stdcall TLSBuildNewTable(
            ULONG_PTR FileMapVA,
            ULONG_PTR StorePlace,
            ULONG_PTR StorePlaceRVA,
            LPVOID ArrayOfCallBacks,
            DWORD NumberOfCallBacks
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content which must be mapped in read/write mode. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping.

StorePlace

[in] Physical address inside PE file on which the new TLS table will be written. Usually this is a new section but can also be the part of the file which is unused but still in read/write mode.

StorePlaceRVA

[in] Relative virtual address inside PE file on which the new TLS table will be written. This input is just conversion from physical to relative virtual offset.

ArrayOfCallBacks

[in] Pointer to array of custom TLS callback.

NumberOfCallBacks

[in] Number of callbacks in the provided array.

Return value

Boolean switch indicating whether or not the function was successful.

Example

None.

TLSBuildNewTableEx function

The **TLSBuildNewTableEx** function is used to build and store completely new TLS table inside the selected PE file. This option can be used to create a new TLS table in case of dealing with protections that use TLS elimination protection technique.

Syntax

```
bool __stdcall TLSBuildNewTableEx(
            char* szFileName,
            char* szSectionName,
            LPVOID ArrayOfCallBacks,
            DWORD NumberOfCallBacks
            );
```

Parameters

szFileName

[in] Pointer to string which is a full path to the file to which new TLS table will be written.

szSectionName

[in] Name of the new PE section in which the new TLS table will be written in. This section will be added to the file. Length of this string is capped at 8 characters.

ArrayOfCallBacks

[in] Pointer to array of custom TLS callback.

NumberOfCallBacks

[in] Number of callbacks in the provided array.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

Librarian module

Librarian module has functions designed for manipulation of loaded libraries, setting breakpoints for specific library loading events and retrieving information about loaded libraries.

Librarian module constants

Constants used by: **LibrarianSetBreakPoint** function and **LibrarianRemoveBreakPoint** function

```
#define UE_ON_LIB_LOAD 1
#define UE_ON_LIB_UNLOAD 2
#define UE_ON_LIB_ALL 3
```

Librarian module structures

Structure used by: **LibrarianGetLibraryInfo** function, **LibrarianGetLibraryInfoEx** function and **LibrarianEnumLibraryInfo** function

```
typedef struct{
      HANDLE hFile;
      void* BaseOfDll;
      HANDLE hFileMapping;
      void* hFileMappingView;
      char szLibraryPath[MAX_PATH];
      char szLibraryName[MAX_PATH];
}LIBRARY_ITEM_DATA, *PLIBRARY_ITEM_DATA;
```


LibrarianSetBreakPoint function

The **LibrarianSetBreakPoint** function is used to set a breakpoint on specific library events such as library loading or unloading.

Syntax

```
bool __stdcall LibrarianSetBreakPoint(
            char* szLibraryName,
            DWORD bpxType,
            bool SingleShoot,
            LPVOID bpxCallBack
            );
```

Parameters

szLibraryName

[in] Name of the library which will trigger the breakpoint for specified event. For example **kernel32.dll**

bpxType

[in] Specifies the event on which the breakpoint will be triggered. Can be one of the following: `UE_ON_LIB_LOAD`, `UE_ON_LIB_UNLOAD` or `UE_ON_LIB_ALL`.

SingleShoot

[in] Specifies if the breakpoint will be executed only once or each time.

bpxCallBack

[in] Address of a callback which will be called once breakpoint has been hit.

CallBack definition

```
typedef void(__stdcall *fCustomBreakPoint)(void* SpecialDBG);

// SpecialDBG is a pointer to &DBGEvent.u.LoadDll
```

Return value

This function returns *TRUE* if the breakpoint has been set and *FALSE* if breakpoint cannot be set.

Remarks

Maximum number of breakpoints is defined with `MAX_LIBRARY_BPX`.

LibrarianRemoveBreakPoint function

The **LibrarianRemoveBreakPoint** function is used to remove a breakpoint set on specific library events such as library loading or unloading.

Syntax

```
bool __stdcall LibrarianRemoveBreakPoint(
            char* szLibraryName,
            DWORD bpxType
            );
```

Parameters

szLibraryName

[in] Name of the library which was used as a breakpoint trigger. For example **kernel32.dll**

bpxType

[in] Specifies the event on which the breakpoint was set. Can be one of the following: `UE_ON_LIB_LOAD`, `UE_ON_LIB_UNLOAD` or `UE_ON_LIB_ALL`.

Return value

This function returns *TRUE* if the breakpoint has been removed and *FALSE* if breakpoint cannot be removed which should never happen.

Remarks

Maximum number of breakpoints is defined with `MAX_LIBRARY_BPX`.

Example

None.

LibrarianGetLibraryInfo function

The **LibrarianGetLibraryInfo** function is used to retrieve additional data about the modules loaded by the debugged process.

Syntax

```
void* __stdcall LibrarianGetLibraryInfo(
            char* szLibraryName
            );
```

Parameters

szLibraryName

[in] Name of the library loaded inside the debugged process. For example **kernel32.dll**

Return value

This function returns the pointer to `LIBRARY_ITEM_DATA` structure or *NULL* if selected DLL cannot be found.

Example

None.

LibrarianGetLibraryInfoEx function

The **LibrarianGetLibraryInfoEx** function is used to retrieve additional data about the modules loaded by the debugged process.

Syntax

```
void* __stdcall LibrarianGetLibraryInfoEx(
            void* BaseOfDll
            );
```

Parameters

BaseOfDll

[in] Base address on which the selected module is loaded in remote process.

Return value

This function returns the pointer to `LIBRARY_ITEM_DATA` structure or *NULL* if selected DLL cannot be found.

Example

None.

LibrarianEnumLibraryInfo function

The **LibrarianEnumLibraryInfo** function is used to enumerate data for all DLL files loaded by the debugged process. This list only contains data about currently loaded modules. All unloaded modules are automatically removed from the list.

Syntax

```
void __stdcall LibrarianEnumLibraryInfo(
            void* EnumCallBack
            );
```

Parameters

EnumCallBack

[in] Address of a callback function which will be used to process loaded library data.

CallBack definition

```
typedef void(__stdcall *fEnumCallBack)(LPVOID fLibraryDetail);

// Here fLibraryDetail is a pointer to LIBRARY_ITEM_DATA structure
```

Return value

None.

Remarks

None.

Example

None.

OEP Finder module

OEP Finder module has functions designed for generic entry point finding.

FindOEPInit function

The **FindOEPInit** function is used to initialize the OEP tracing process. It is not necessary to call it since it will be auto called by the engine itself.

Syntax

```
void __stdcall FindOEPInit();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

FindOEPGenerically function

The **FindOEPGenerically** function is used to generically find the packed file original entry point. There are some limitations to what formats are supported. This function only supports packers which use *LoadLibrary* in order to load more than just *kernel32.dll*. **WARNING**: File might get executed so use this function with caution!

Syntax

```
void __stdcall FindOEPGenerically(
            char* szFileName,
            LPVOID TraceInitCallBack,
            LPVOID CallBack
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be debugged in order to locate its entry point.

TraceInitCallBack

[in] Callback which will be called once packed file hits its packed entry point.

CallBack

[in] Callback which will be called once packed file hits its original entry point.

Return value

None.

Remarks

All callbacks used here are normal breakpoint callbacks. See *SetBPX function* for details.

Example

See RL!dePacke2 source code.

Process module

Process module has additional functions designed for process enumeration and execution basic operations inside the context of the remote process.

GetActiveProcessId function

The **GetActiveProcessId** function is used to find an active process with by its image name.

Syntax

```
long __stdcall GetActiveProcessId(
            char* szImageName
            );
```

Parameters

szImageName

[in] Running process name image. For example **explorer.exe**

Return value

This function returns process ID if the process is running and found and NULL if the image with the specified name isn't currently running.

Remarks

In case of multiple process images with the same name this function always returns the ID of the first found.

Example

None.

EnumProcessesWithLibrary function

The **EnumProcessesWithLibrary** function is used to enumerate all processes which loaded the specified DLL image.

Syntax

```
void __stdcall EnumProcessesWithLibrary(
            char* szLibraryName,
            void* EnumFunction
            );
```

Parameters

szLibraryName

[in] Name of the library loaded inside the debugged process. For example **kernel32.dll**

EnumFunction

[in] Address of a callback function which will be used to process found data.

CallBack definition

```
typedef void(__stdcall *fEnumFunction)(DWORD ProcessId,
                                       HMODULE ModuleBaseAddress);
```

Return value

None.

Remarks

None.

Example

None.

RemoteLoadLibrary function

The **RemoteLoadLibrary** function is used to make the remote process load the selected DLL file. This function injects a remote thread in the selected process which calls *LoadLibraryA* in order to load the DLL file from the disk.

Syntax

```
bool __stdcall RemoteLoadLibrary(
            HANDLE hProcess,
            char* szLibraryFile,
            bool WaitForThreadExit
            );
```

Parameters

hProcess

[in] Handle of the process in which the new thread that loads the DLL will be created.

szLibraryName

[in] Name of the library to be loaded inside remote process. For example **advapi32.dll**

WaitForThreadExit

[in] Boolean switch indicating whether or not to wait for remote thread to terminate before returning from this function call.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

RemoteFreeLibrary function

The **RemoteFreeLibrary** function is used to make the remote process unload the selected DLL file. This function injects a remote thread in the selected process which calls *FreeLibrary* in order to unload the selected DLL file.

Syntax

```
bool __stdcall RemoteFreeLibrary(
            HANDLE hProcess,
            HMODULE hModule,
            char* szLibraryFile,
            bool WaitForThreadExit
            );
```

Parameters

hProcess

[in] Handle of the process in which the new thread that unloads the DLL will be created.

hModule

[in] Base address on which the DLL file is loaded in remote process.

szLibraryName

[in] Name of the library which will be unloaded from the remote process. For example **advapi32.dll**

WaitForThreadExit

[in] Boolean switch indicating whether or not to wait for remote thread to terminate before returning from this function call.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

It is sufficient to supply only one argument which identifies the module, either name or base address.

Example

None.

RemoteFreeLibrary function

The **RemoteExitProcess** function is used to make the remote process terminate its execution. This function injects a remote thread in the selected process which calls *ExitProcess* in order to terminate it.

Syntax

```
bool __stdcall RemoteExitProcess(
            HANDLE hProcess,
            DWORD ExitCode
            );
```

Parameters

hProcess

[in] Handle of the process in which the new thread that terminates it will be created.

ExitCode

[in] Exit code that will be passed to *ExitProcess* API. Can be NULL.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

TranslateNativeName function

The **TranslateNativeName** function is used to resolve full path to file from its native name. Paths which contain physical devices in their file references are resolved with this function.

Syntax

```
void* __stdcall TranslateNativeName(
            char* szNativeName
            );
```

Parameters

hProcess

[in] Handle of the process in which the new thread that terminates it will be created.

Return value

Function returns a pointer to decoded file name, or NULL if the supplied string can't be decoded.

Remarks

String with the translated native name is stored inside the engine which makes this function multi thread unsafe.

Example

None.

Dumper module

Dumper module has functions designed for dumping process, region and module memory dumping. This module also contains functions to aid in work with PE header specifics and file overlay.

Dumper module constants

Constants used by: **GetPE32DataFromMappedFile** function, **GetPE32Data** function, **SetPE32DataForMappedFile** function, and **SetPE32Data** function**.**

```
#define UE_PE_OFFSET 0
#define UE_IMAGEBASE 1
#define UE_OEP 2
#define UE_SIZEOFIMAGE 3
#define UE_SIZEOFHEADERS 4
#define UE_SIZEOFOPTIONALHEADER 5
#define UE_SECTIONALIGNMENT 6
#define UE_IMPORTTABLEADDRESS 7
#define UE_IMPORTTABLESIZE 8
#define UE_RESOURCETABLEADDRESS 9
#define UE_RESOURCETABLESIZE 10
#define UE_EXPORTTABLEADDRESS 11
#define UE_EXPORTTABLESIZE 12
#define UE_TLSTABLEADDRESS 13
#define UE_TLSTABLESIZE 14
#define UE_RELOCATIONTABLEADDRESS 15
#define UE_RELOCATIONTABLESIZE 16
#define UE_TIMEDATESTAMP 17
#define UE_SECTIONNUMBER 18
#define UE_CHECKSUM 19
#define UE_SUBSYSTEM 20
#define UE_CHARACTERISTICS 21
#define UE_NUMBEROFRVAANDSIZES 22
#define UE_SECTIONNAME 23
#define UE_SECTIONVIRTUALOFFSET 24
#define UE_SECTIONVIRTUALSIZE 25
#define UE_SECTIONRAWOFFSET 26
#define UE_SECTIONRAWSIZE 27
#define UE_SECTIONFLAGS 28
```

Constants used by: **GetPE32SectionNumberFromVA** function

```
#define UE_VANOTFOUND -2
```


Dumper module structures

Structures used by: **GetPE32DataFromMappedFileEx** function, **GetPE32DataEx** function, **SetPE32DataForMappedFileEx** function and **SetPE32DataEx** function.

```
typedef struct{
      DWORD PE32Offset;
      DWORD ImageBase;
      DWORD OriginalEntryPoint;
      DWORD NtSizeOfImage;
      DWORD NtSizeOfHeaders;
      WORD SizeOfOptionalHeaders;
      DWORD FileAlignment;
      DWORD SectionAligment;
      DWORD ImportTableAddress;
      DWORD ImportTableSize;
      DWORD ResourceTableAddress;
      DWORD ResourceTableSize;
      DWORD ExportTableAddress;
      DWORD ExportTableSize;
      DWORD TLSTableAddress;
      DWORD TLSTableSize;
      DWORD RelocationTableAddress;
      DWORD RelocationTableSize;
      DWORD TimeDateStamp;
      WORD SectionNumber;
      DWORD CheckSum;
      WORD SubSystem;
      WORD Characteristics;
      DWORD NumberOfRvaAndSizes;
}PE32Struct, *PPE32Struct;
```


Structures used by: **GetPE32DataFromMappedFileEx** function, **GetPE32DataEx function**, **SetPE32DataForMappedFileEx** function and **SetPE32DataEx** function.

```
typedef struct{
      DWORD PE64Offset;
      DWORD64 ImageBase;
      DWORD OriginalEntryPoint;
      DWORD NtSizeOfImage;
      DWORD NtSizeOfHeaders;
      WORD SizeOfOptionalHeaders;
      DWORD FileAlignment;
      DWORD SectionAligment;
      DWORD ImportTableAddress;
      DWORD ImportTableSize;
      DWORD ResourceTableAddress;
      DWORD ResourceTableSize;
      DWORD ExportTableAddress;
      DWORD ExportTableSize;
      DWORD TLSTableAddress;
      DWORD TLSTableSize;
      DWORD RelocationTableAddress;
      DWORD RelocationTableSize;
      DWORD TimeDateStamp;
      WORD SectionNumber;
      DWORD CheckSum;
      WORD SubSystem;
      WORD Characteristics;
      DWORD NumberOfRvaAndSizes;
}PE64Struct, *PPE64Struct;
```


DumpProcess function

The **DumpProcess** function creates a file on disk which is a memory dump of a running process. This image is not a valid PE file but a state of memory at the time of using this function.

Syntax

```
bool __stdcall DumpProcess(
            HANDLE hProcess,
            LPVOID ImageBase,
            char* szDumpFileName,
            ULONG_PTR EntryPoint
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be dumped to disk.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

EntryPoint

[in] Virtual address which will be set to the new file. Size of this variable varies, on x86 its 4 bytes and on x64 its 8 bytes. Therefore it can also be declared as void*.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpProcessEx function

The **DumpProcessEx** function creates a file on disk which is a memory dump of a running process. This image is not a valid PE file but a state of memory at the time of using this function.

Syntax

```
bool __stdcall DumpProcessEx(
            DWORD ProcessId,
            LPVOID ImageBase,
            char* szDumpFileName,
            ULONG_PTR EntryPoint
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

EntryPoint

[in] Virtual address which will be set to the new file. Size of this variable varies, on x86 its 4 bytes and on x64 its 8 bytes. Therefore it can also be declared as void*.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpMemory function

The **DumpMemory** function creates a file on disk which is a memory dump of selected memory part of the running process.

Syntax

```
bool __stdcall DumpMemory(
            HANDLE hProcess,
            LPVOID MemoryStart,
            ULONG_PTR MemorySize,
            char* szDumpFileName
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be dumped to disk.

MemoryStart

[in] Start of a memory range to be dumped to disk. This start can be different from the start of a page.

MemorySize

[in] This variable is a size of the memory which will be copied to disk. Size of this variable varies, on x86 its 4 bytes and on x64 its 8 bytes. Therefore it can also be declared as SIZE_T.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpMemoryEx function

The **DumpMemoryEx** function creates a file on disk which is a memory dump of selected memory part of the running process.

Syntax

```
bool __stdcall DumpMemoryEx(
            DWORD ProcessId,
            LPVOID MemoryStart,
            ULONG_PTR MemorySize,
            char* szDumpFileName
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

MemoryStart

[in] Start of a memory range to be dumped to disk. This start can be different from the start of a page.

MemorySize

[in] This variable is a size of the memory which will be copied to disk. Size of this variable varies, on x86 its 4 bytes and on x64 its 8 bytes. Therefore it can also be declared as SIZE_T.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpRegions function

The **DumpRegions** function creates series of files on disk which are a memory dump of all used memory regions inside the running process. Optionally this function can dump only regions above the image base of the file.

Syntax

```
bool __stdcall DumpRegions(
            HANDLE hProcess,
            char* szDumpFolder,
            bool DumpAboveImageBaseOnly
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be dumped to disk.

szDumpFolder

[in] Pointer to a null terminated string which is a full path to folder in which the memory content will be dumped. Every region will be dumped in separate file.

DumpAboveImageBaseOnly

[in] This variable is a switch which tells the engine which regions to dump. If its value is *FALSE* it will dump all regions and if it is *TRUE* it will dump only regions above image base of the targeted PE file.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpRegionsEx function

The **DumpRegionsEx** function creates series of files on disk which are a memory dump of all used memory regions inside the running process. Optionally this function can dump only regions above the image base of the file.

Syntax

```
bool __stdcall DumpRegions(
            DWORD ProcessId,
            char* szDumpFolder,
            bool DumpAboveImageBaseOnly
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

szDumpFolder

[in] Pointer to a null terminated string which is a full path to folder in which the memory content will be dumped. Every region will be dumped in separate file.

DumpAboveImageBaseOnly

[in] This variable is a switch which tells the engine which regions to dump. If its value is *FALSE* it will dump all regions and if it is *TRUE* it will dump only regions above image base of the targeted PE file.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpModule function

The **DumpModule** function creates a file on disk which is a memory dump of one module of a running process. This image is not a valid PE file but a state of memory at the time of using this function.

Syntax

```
bool __stdcall DumpModule(
            HANDLE hProcess,
            LPVOID ModuleBase,
            char* szDumpFileName
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be dumped to disk.

ModuleBase

[in] Loaded modules base address of a running process to be dumped.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

DumpModuleEx function

The **DumpModuleEx** function creates a file on disk which is a memory dump of one module of a running process. This image is not a valid PE file but a state of memory at the time of using this function.

Syntax

```
bool __stdcall DumpModuleEx(
            DWORD ProcessId,
            LPVOID ModuleBase,
            char* szDumpFileName
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

ModuleBase

[in] Loaded modules base address of a running process to be dumped.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

Return value

This function returns *TRUE* on successful dump and *FALSE* if the memory dump fails.

Remarks

None.

Example

None.

PastePEHeader function

The **PastePEHeader** function loads the PE header from the file on disk and writes it to running process memory. This can be used to fix damages to PE header during process runtime. Such damages only occur as a result of memory protection algorithms used by some protection solutions.

Syntax

```
bool __stdcall PastePEHeader(
            HANDLE hProcess,
            LPVOID ImageBase,
            char* szDebuggedFileName
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be dumped to disk.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

szDebuggedFileName

[in] Pointer to a null terminated string which is a full path to file from which PE header will be read.

Return value

This function returns *TRUE* on successful fix and *FALSE* if the header paste fails.

Remarks

None.

Example

None.

ExtractSection function

The **ExtractSection** function copies the physical content of selected file section to new file on disk.

Syntax

```
bool __stdcall ExtractSection(
            char* szFileName,
            char* szDumpFileName,
            DWORD SectionNumber
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose section will be extracted.

szDumpFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content.

SectionNumber

[in] Number of the section which will be extracted to disk. Section numbers go from *zero* to section count minus one.

Return value

This function returns *TRUE* on successful extract and *FALSE* if the extraction fails.

Remarks

None.

Example

None.

ResortFileSections function

The **ResortFileSections** function sorts the file physical sections so they are in the order of ascending physical offset. This can be useful if there is a need to add data to the last section or expand it but it isn't physically last.

Syntax

```
bool __stdcall ResortFileSections(
            char* szFileName,
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose section will be resorted.

Return value

This function returns *TRUE* on successful resort and *FALSE* if the sorting fails.

Remarks

File doesn't change its size but it will change its hash because sections will be physically moved to new positions.

Example

None.

FindOverlay function

The **FindOverlay** function finds the extra appended data at the end of the PE file. This data can be file certificate or other more important data for further file analysis.

Syntax

```
bool __stdcall FindOverlay(
            char* szFileName,
            LPDWORD OverlayStart,
            LPDWORD OverlaySize
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be queried for overlay presence.

OverlayStart

[out] Pointer to a DWORD which will receive the file pointer to overlay data. This pointer can be used with Windows API to position the file pointer to data which represents the overlay.

OverlaySize

[out] Pointer to a DWORD which will receive the size of the found overlay.

Return value

This function returns *TRUE* on found overlay and *FALSE* if the overlay is not found.

Remarks

None.

Example

None.

ExtractOverlay function

The **ExtractOverlay** function finds the extra appended data at the end of the PE file and copies that memory content to new file. This data can be file certificate or other more important data for further file analysis.

Syntax

```
bool __stdcall ExtractOverlay(
            char* szFileName,
            char* szExtactedFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be queried for overlay presence and whose overlay will be extracted.

szExtractedFileName

[in] Pointer to a null terminated string which is a full path to file which will contain the overlay content from the input file.

Return value

This function returns *TRUE* on found and extracted overlay and *FALSE* if the overlay is not found or overlay export fails.

Remarks

Output file is always overwritten.

Example

None.

AddOverlay function

The **AddOverlay** function appends data to the end of PE files. This data can be file certificate or any other type of data.

Syntax

```
bool __stdcall AddOverlay(
            char* szFileName,
            char* szOverlayFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be queried for overlay presence and whose overlay will be extracted.

szOverlayFileName

[in] Pointer to a null terminated string which is a full path to file which will be appended to PE file.

Return value

This function returns *TRUE* on data successfully appended and *FALSE* if one of the files is not found.

Remarks

This function can also be used to merge two non PE files.

Example

None.

CopyOverlay function

The **CopyOverlay** function copies overlay from one PE file to another. If target file already has overlay data new data will be copied just after existing.

Syntax

```
bool __stdcall CopyOverlay(
            char* szInFileName,
            char* szOutFileName
            );
```

Parameters

szInFileName

[in] Pointer to a null terminated string which is a full path to file which will be queried for overlay presence and whose overlay will be copied.

szOutFileName

[in] Pointer to a null terminated string which is a full path to file to which new overlay data will be appended.

Return value

This function returns *TRUE* on data successfully appended and *FALSE* if one of the files is not found or not a PE file.

Remarks

None.

Example

None.

RemoveOverlay function

The **RemoveOverlay** function removes overlay data from PE files.

Syntax

```
bool __stdcall RemoveOverlay(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be stripped of its overlay data.

Return value

This function returns *TRUE* if overlay is removed and *FALSE* if overlay or file isn't found.

Remarks

None.

Example

None.

MakeAllSectionsRWE function

The **MakeAllSectionsRWE** function set characteristics of all PE file sections to read/write/executable.

Syntax

```
bool __stdcall MakeAllSectionsRWE(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose PE sections characteristics will be set to read/write/executable.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

None.

Example

None.

AddNewSectionEx function

The **AddNewSectionEx** function adds new PE section to a file. Newly created section is physical and its content is filled with zeroes if no content to be copied to new section is specified. This reserved space can be later used to store data.

Syntax

```
long __stdcall AddNewSectionEx(
            char* szFileName,
            char* szSectionName,
            DWORD SectionSize,
            DWORD SectionAttributes,
            LPVOID SectionContent,
            DWORD ContentSize
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file to which new section will be added.

szSectionName

[in] Pointer to a null terminated string which will be the new section name. This string can only be 8 characters long.

SectionSize

[in] Size of the new section, both virtual and physical. Virtual size will be rounded up to next modus of SectionAlignment, and physical size will be rounded up to next modus of FileAlignment.

SectionAttributes

[in] Section attributes as defined by PECOFF 8. If this value is NULL attributes will be set to default read/write/execute `0xE0000020` value.

SectionContent

[in] Pointer to memory whose content will be copied to newly created section.

ContentSize

[in] Size of the memory content which will be copied to new section.

Return value

This function returns relative virtual offset of newly created section or NULL if adding new section fails.

AddNewSection function

The **AddNewSection** function adds new PE section to a file. Newly created section is physical and its content is filled with zeroes with no content copied to new section. This reserved space can be later used to store data.

Syntax

```
long __stdcall AddNewSection(
            char* szFileName,
            char* szSectionName,
            DWORD SectionSize
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file to which new section will be added.

szSectionName

[in] Pointer to a null terminated string which will be the new section name. This string can only be 8 characters long.

SectionSize

[in] Size of the new section, both virtual and physical. Virtual size will be rounded up to next modus of SectionAlignment, and physical size will be rounded up to next modus of FileAlignment.

Return value

This function returns relative virtual offset of newly created section or NULL if adding new section fails.

Remarks

None.

Example

None.

ResizeLastSection function

The **ResizeLastSection** function increases the size of the last PE file section. Section is increased both physically and virtually. Optionally new section size is aligned to FileAlignment.

Syntax

```
bool __stdcall ResizeLastSection(
            char* szFileName,
            DWORD NumberOfExpandBytes,
            bool AlignResizeData
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose last PE section will be resized.

NumberOfExpandBytes

[in] Last section will be increased by this variable value.

AlignResizeData

[in] If set to *TRUE* last section size will be aligned to FileAlignment.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

File is backup before modification and restored if the file cannot be resized.

Example

None.

SetSharedOverlay function

The **SetSharedOverlay** function is used only to store string pointer provided to it. This function is a residue from the old SDK and there is no actual reason to use it now.

Syntax

```
void __stdcall SetSharedOverlay(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file. This string pointer will be stored in case other modules need to retrieve it but have no direct access to the variable. String itself won't be moved or modified so it must remain at that location for all time it is needed.

Return value

This function has no return value.

Remarks

None.

Example

None.

GetSharedOverlay function

The **GetSharedOverlay** function is used only to retrieve store string pointer provided by *SetSharedOverlay* function. This function is a residue from the old SDK and there is no actual reason to use it now.

Syntax

```
char* __stdcall GetSharedOverlay();
```

Parameters

None.

Return value

This function returns the previously stored pointer.

Remarks

None.

Example

None.

DeleteLastSection function

The **DeleteLastSection** function is used to physically delete last PE section.

Syntax

```
bool __stdcall DeleteLastSection(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose last PE section will be deleted.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

File is backup before modification and restored if the file cannot be resized.

Example

None.

DeleteLastSectionEx function

The **DeleteLastSectionEx** function is used to physically delete selected number PE sections from the end of the file.

Syntax

```
bool __stdcall DeleteLastSectionEx(
            char* szFileName,
            DWORD NumberOfSections
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose last PE section(s) will be deleted.

NumberOfSections

[in] Number of the sections which will be removed from the end of the PE file.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

File is backup before modification and restored if the file cannot be resized.

Example

None.

GetPE32DataFromMappedFile function

The **GetPE32DataFromMappedFile** function is used to retrieve data from the PE header for both x86 and x64 files.

Syntax

```
long long __stdcall GetPE32DataFromMappedFile(
            ULONG_PTR FileMapVA,
            DWORD WhichSection,
            DWORD WhichData
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

WhichSection

[in] Number of the PE section from which data will be read. Here the first PE section is referred to as zero section, so the section numbers go from zero to section count minus one.

WhichData

[in] Indicator on which PE header info this function will return. List of constants which are used by this function is located at the beginning of this section under *Dumper module constants*.

Return value

This function returns requested value. Return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system.

Remarks

File must be mapped before using this function.

GetPE32DataFromMappedFileEx function

The **GetPE32DataFromMappedFileEx** function is used to retrieve data from the PE header for both x86 and x64 files.

Syntax

```
bool __stdcall GetPE32DataFromMappedFileEx(
            ULONG_PTR FileMapVA,
            LPVOID DataStorage
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64. But since this is a pointer void* can also be used.

DataStorage

[in] Pointer to a structure which will receive all PE header data. This structure is different for x86 and x64 and its definition is located at the beginning of this section under *Dumper module structures*.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

File must be mapped before using this function.

Example

None.

GetPE32Data function

The **GetPE32Data** function is used to retrieve data from the PE header for both x86 and x64 files.

Syntax

```
long long __stdcall GetPE32Data(
            char* szFileName,
            DWORD WhichSection,
            DWORD WhichData
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file from which PE header data will be read.

WhichSection

[in] Number of the PE section from which data will be read. Here the first PE section is referred to as zero section, so the section numbers go from zero to section count minus one.

WhichData

[in] Indicator on which PE header info this function will return. List of constants which are used by this function is located at the beginning of this section under *Dumper module constants*.

Return value

This function returns requested value. Return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system.

Remarks

None.

Example

None.

GetPE32DataEx function

The **GetPE32DataEx** function is used to retrieve data from the PE header for both x86 and x64 files.

Syntax

```
bool __stdcall GetPE32DataEx(
            char* szFileName,
            LPVOID DataStorage
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file from which PE header data will be read.

DataStorage

[in] Pointer to a structure which will receive all PE header data. This structure is different for x86 and x64 and its definition is located at the beginning of this section under *Dumper module structures*.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

None.

Example

None.

SetPE32DataForMappedFile function

The **SetPE32DataFromMappedFile** function is used to set data to PE header for both x86 and x64 files.

Syntax

```
bool __stdcall SetPE32DataForMappedFile(
            ULONG_PTR FileMapVA,
            DWORD WhichSection,
            DWORD WhichData,
            ULONG_PTR NewDataValue
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

WhichSection

[in] Number of the PE section from which data will be read. Here the first PE section is referred to as zero section, so the section numbers go from zero to section count minus one.

WhichData

[in] Indicator on which PE header info this function will return. List of constants which are used by this function is located at the beginning of this section under *Dumper module constants*.

NewDataValue

[in] Value which will be set for the selected PE header field.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

File must be mapped before using this function.

SetPE32DataForMappedFileEx function

The **SetPE32DataForMappedFileEx** function is used to set data to the PE header for both x86 and x64 files.

Syntax

```
bool __stdcall SetPE32DataForMappedFileEx(
            ULONG_PTR FileMapVA,
            LPVOID DataStorage
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64. But since this is a pointer void* can also be used.

DataStorage

[in] Pointer to a structure which will be used to reset all PE header data. Ideally this structure is first filled by using *GetPE32DataFromMappedFileEx* function. This structure is different for x86 and x64 and its definition is located at the beginning of this section under *Dumper module structures*.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

File must be mapped before using this function.

Example

None.

SetPE32Data function

The **SetPE32Data** function is used to set data to the PE header for both x86 and x64 files.

Syntax

```
bool __stdcall SetPE32Data(
            char* szFileName,
            DWORD WhichSection,
            DWORD WhichData
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose PE header data will be modified.

WhichSection

[in] Number of the PE section from which data will be read. Here the first PE section is referred to as zero section, so the section numbers go from zero to section count minus one.

WhichData

[in] Indicator on which PE header info this function will return. List of constants which are used by this function is located at the beginning of this section under *Dumper module constants*.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

None.

Example

None.

SetPE32DataEx function

The **SetPE32DataEx** function is used to set data to the PE header for both x86 and x64 files.

Syntax

```
bool __stdcall SetPE32DataEx(
            char* szFileName,
            LPVOID DataStorage
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose PE header data will be modified.

DataStorage

[in] Pointer to a structure which will be used to reset all PE header data. Ideally this structure is first filled by using *GetPE32DataFromMappedFileEx* function. This structure is different for x86 and x64 and its definition is located at the beginning of this section under *Dumper module structures*.

Return value

This function returns *TRUE* on success and *FALSE* if file doesn't exist or PE header is broken.

Remarks

None.

Example

None.

GetPE32SectionNumberFromVA function

The **GetPE32SectionNumberFromVA** function is used to determine in which PE section selected virtual address resides in.

Syntax

```
long __stdcall GetPE32SectionNumberFromVA(
            ULONG_PTR FileMapVA,
            ULONG_PTR AddressToConvert
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64. But since this is a pointer void* can also be used.

AddressToConvert

[in] Virtual address which will be located inside mapped file sections.

Return value

This function returns number of section in which virtual address resides or UE_VANOTFOUND. Here the first PE section is referred to as zero section, so the section numbers go from zero to section count minus one.

Remarks

File must be mapped before using this function.

Example

None.

ConvertVAtoFileOffset function

The **ConvertVAtoFileOffset** function is used to convert virtual addresses to their physical counterpart.

Syntax

```
long long __stdcall ConvertVAtoFileOffset(
            ULONG_PTR FileMapVA,
            ULONG_PTR AddressToConvert,
            bool ReturnType
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system.

AddressToConvert

[in] Virtual address which will be converted to physical address.

ReturnType

[in] Boolean variable which indicates whether or not to add *FileMapVA* to function return.

Return value

This function returns the converted physical address. Return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system. If *ReturnType* is *FALSE* this value will never be larger than DWORD. If this function returns *NULL* conversion has failed.

Remarks

File must be mapped before using this function.

Example

None.

ConvertVAtoFileOffsetEx function

The **ConvertVAtoFileOffsetEx** function is used to convert virtual or relative virtual addresses to their physical counterpart. Using this function is considered safer then *ConvertVAtoFileOffset* because safety checks that make sure that PE file is valid and memory is there and accessible.

Syntax

```
long long __stdcall ConvertVAtoFileOffsetEx(
            ULONG_PTR FileMapVA,
            DWORD FileSize,
            ULONG_PTR ImageBase,
            ULONG_PTR AddressToConvert,
            bool AddressIsRVA,
            bool ReturnType
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system.

FileSize

[in] Size of the mapped file.

ImageBase

[in] ImageBase of the mapped file read directly from its PE header.

AddressToConvert

[in] Virtual address which will be converted to physical address.

AddressIsRVA

[in] Boolean variable which indicates whether or not to input address is relative or virtual which is the default expected input.

ReturnType

[in] Boolean variable which indicates whether or not to add *FileMapVA* to function return.

Return value

This function returns the converted physical address. Return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system. If *ReturnType* is *FALSE* this value will never be larger than DWORD. If this function returns *NULL* conversion has failed.

ConvertFileOffsetToVA function

The **ConvertFileOffsetToVA** function is used to convert physical addresses to their virtual counterpart.

Syntax

```
long long __stdcall ConvertFileOffsetToVA(
            ULONG_PTR FileMapVA,
            ULONG_PTR AddressToConvert,
            bool ReturnType
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system.

AddressToConvert

[in] Physical address which will be converted to virtual address. It must reside in address space allocated with *FileMapVA*.

ReturnType

[in] Boolean variable which indicates whether or not to add *ImageBase* to function return.

Return value

This function returns the converted virtual address. Return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system. If *ReturnType* is *FALSE* this value will never be larger than DWORD. If this function returns *NULL* conversion has failed.

Remarks

File must be mapped before using this function.

Example

None.

ConvertFileOffsetToVAEx function

The **ConvertFileOffsetToVAEx** function is used to convert physical addresses to their virtual counterpart. Using this function is considered safer then *ConvertFileOffsetToVA* because safety checks that make sure that PE file is valid and memory is there and accessible.

Syntax

```
long long __stdcall ConvertFileOffsetToVAEx(
            ULONG_PTR FileMapVA,
            DWORD FileSize,
            ULONG_PTR ImageBase,
            ULONG_PTR AddressToConvert,
            bool ReturnType
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system.

FileSize

[in] Size of the mapped file.

ImageBase

[in] ImageBase of the mapped file read directly from its PE header.

AddressToConvert

[in] Physical address which will be converted to virtual address. It must reside in address space allocated with *FileMapVA*.

ReturnType

[in] Boolean variable which indicates whether or not to add *ImageBase* to function return.

Return value

This function returns the converted virtual address. Return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system. If *ReturnType* is *FALSE* this value will never be larger than DWORD. If this function returns *NULL* conversion has failed.

Remarks

File must be mapped before using this function.

Importer module

Importer module has functions designed for import manipulation, forward handling and automatic import locating and fixing.

Importer module structures

Structure used by: **ImporterEnumAddedData** function

```
typedef struct{
      bool NewDll;
      int NumberOfImports;
      ULONG_PTR ImageBase;
      ULONG_PTR BaseImportThunk;
      ULONG_PTR ImportThunk;
      char* APIName;
      char* DLLName;
}ImportEnumData, *PImportEnumData;
```


ImporterInit function

The **ImporterInit** function initializes the importer module and it must be used before using any of the functions used in the process of manual import fixing.

Syntax

```
void __stdcall ImporterInit(
            DWORD MemorySize,
            ULONG_PTR ImageBase
            );
```

Parameters

MemorySize

[in] Default memory size allocated for each of the new DLLs files you add. This size must be large enough to hold all data needed by the engine. Usually there is no need to reserve more than 40kb of memory.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

Return value

None.

Remarks

None.

Example

None.

ImporterSetImageBase function

The **ImporterSetImageBase** function is used to update information passed to engine on importer initialization.

Syntax

```
void __stdcall ImporterSetImageBase(
            ULONG_PTR ImageBase
            );
```

Parameters

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

Return value

None.

Remarks

None.

Example

None.

ImporterAddNewDll function

The **ImporterAddNewDll** function adds new DLLs to the new import tree. This function creates a new DLL entry making all following *ImporterAddNewAPI* function calls adding APIs to current DLL. If APIs don't belong to current DLL add a new DLL entry first. PECOFF specifications imply that trunks are in a plus four (or eight on x64) sequence. Importer takes care of this automatically and adds a new DLL entry equal to the last entered DLL if this sequence is broken.

Syntax

```
void __stdcall ImporterAddNewDll(
            char* szDLLName,
            ULONG_PTR FirstThunk
            );
```

Parameters

szDLLName

[in] Pointer to string which is a name of DLL to be added to new import tree. For example: **kernel32.dll** or **SomeFolder\mydll.dll** in case of relative path loading.

FirstThunk

[in] Optional parameter which is an address inside the PE files memory that holds the pointer to API belonging to that DLL. This is a first pointer in the sequence. Parameter is optional and can be set to *NULL* in which case first next call to *ImporterAddNewAPI* will set the first trunk to data provided to it. In case trunk is outside the PE files memory you must use a special approach described in examples.

Return value

None.

Remarks

None.

Example

None.

ImporterAddNewAPI function

The **ImporterAddNewAPI** function adds new API to the current import tree. This function creates a new API entry under currently selected DLL added by *ImporterAddNewDLL*. If APIs don't belong to current DLL add a new DLL entry first. PECOFF specifications imply that trunks are in a plus four (or eight on x64) sequence. Importer takes care of this automatically and adds a new DLL entry equal to the last entered DLL if this sequence is broken.

Syntax

```
void __stdcall ImporterAddNewAPI(
            char* szAPIName,
            ULONG_PTR ThunkValue
            );
```

Parameters

szAPIName

[in] Pointer to string which is a name of API to be added to new import tree but belonging to current DLL. For example: **VirtualProtect** or **VirtualAlloc**, which are added to currently added DLL which is in this case **kernel32.dll**

ThunkValue

[in] Mandatory parameter which is an address inside the PE files memory that holds the pointer to API belonging to that DLL. In case trunk is outside the PE files memory you must use a special approach described in examples.

Return value

None.

Remarks

None.

Example

None.

ImporterGetAddedDllCount function

The **ImporterGetAddedDllCount** function gets the current number of added DLLs inside the import tree.

Syntax

```
long __stdcall ImporterGetAddedDllCount();
```

Parameters

None.

Return value

Returns the number of added DLLs inside the import tree.

Remarks

None.

Example

None.

ImporterGetAddedAPICount function

The **ImporterGetAddedAPICount** function gets the current number of added APIs inside the import tree.

Syntax

```
long __stdcall ImporterGetAddedAPICount();
```

Parameters

None.

Return value

Returns the number of added APIs inside the import tree.

Remarks

None.

Example

None.

ImporterEnumAddedData function

The **ImporterEnumAddedData** function enumerates all added import tree data and calls the designated callback for each added item with the details about it.

Syntax

```
void __stdcall ImporterEnumAddedData(
            LPVOID EnumCallBack
            );
```

Parameters

EnumCallBack

[in] Address of a callback function which will be used to process added import data.

CallBack definition

```
typedef void(__stdcall *fEnumCallBack)(LPVOID ptrImportEnumData);

typedef struct{
      bool NewDll;                  // Indicator on if the dll has changed
      int NumberOfImports;
      ULONG_PTR ImageBase;
      ULONG_PTR BaseImportThunk;    // Original first trunk
      ULONG_PTR ImportThunk;        // Current import trunk
      char* APIName;
      char* DLLName;
}ImportEnumData, *PImportEnumData;
```

Return value

None.

Remarks

Strings with the API and the DLL name is stored inside the engine which makes this function multi thread unsafe.

ImporterEstimatedSize function

The **ImporterEstimatedSize** function estimates the size of memory needed to write the import data. This value can be used to determine the size of the new section in which the import data will be written.

Syntax

```
long __stdcall ImporterEstimatedSize();
```

Parameters

None.

Return value

Returns the size needed to write the import data.

Remarks

None.

Example

None.

ImporterCleanup function

The **ImporterCleanup** function clears all added DLLs and APIs from the import tree. This resets the inputted data to original state. Before using the functions to add the data to import tree you must initialize the importer again.

Syntax

```
void __stdcall ImporterCleanup();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ImporterExportIATEx function

The **ImporterExportIATEx** function exports the added import data to existing PE file creating the valid import table for the selected PE file. After this function has executed import data will be cleared by using *ImporterCleanup* function.

Syntax

```
bool __stdcall ImporterExportIATEx(
            char* szExportFileName,
            char* szSectionName
            );
```

Parameters

szExportFileName

[in] Pointer to string which is a full path to the file to which new import table will be written. This file is usually created by using *DumpProcess* function.

szSectionName

[in] Name of the PE section in which the new import table content will be written in. This section will be added to the file. Length of this string is capped at 8 characters.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ImporterExportIAT function

The **ImporterExportIAT** function exports the added import data to existing PE file creating the valid import table for the selected PE file. After this function has executed import data will be cleared by using *ImporterCleanup* function.

Syntax

```
bool __stdcall ImporterExportIAT(
            ULONG_PTR StorePlace,
            ULONG_PTR FileMapVA
            );
```

Parameters

StorePlace

[in] Physical address inside PE file on which the new import table will be written. Usually this is a new section but can also be the part of the file which is unused but still in read/write mode.

FileMapVA

[in] Pointer to the mapped file content which must be mapped in read/write mode. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ImporterGetAPIName function

The **ImporterGetAPIName** function returns the name of the API whose address is supplied to the function. This address is usually the pointer to API located inside the import table of the file whose IAT you are fixing.

Syntax

```
void* __stdcall ImporterGetAPIName(
            ULONG_PTR APIAddress
            );
```

Parameters

APIAddress

[in] Address on which possible API is located. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest.

Return value

Pointer to string which is the name of the API located at supplied address. If there is no DLL export function at that address this string will be empty but the pointer to it will still be returned.

Remarks

String with the API name is stored inside the engine which makes this function multi thread unsafe. Process which is searched for the API is always the currently debugged process.

Example

ImporterGetAPIName(GetProcAddress(GetModuleHandleA(“kernel32.dll”), “VirtualAlloc”));

This will return a pointer to “VirtualAlloc” string, without the quotes. Example can fail in ASLR environment since API address must reside inside the debugged process, therefore if used like this local API address inside debugger must be relocated to remote one inside the debugge.

ImporterGetAPINameEx function

The **ImporterGetAPINameEx** function returns the name of the API whose address is supplied to the function. This address is usually the pointer to API located inside the import table of the file whose IAT you are fixing. Expert version of this function only searches for API in the provided module list. This list is compiled of DLL module base addresses from the debugged process.

Syntax

```
void* __stdcall ImporterGetAPINameEx(
            ULONG_PTR APIAddress,
            ULONG_PTR DLLBasesList
            );
```

Parameters

APIAddress

[in] Address on which possible API is located. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest.

DLLBasesList

[in] Pointer to array of module base addresses inside the remote process. This list is either manually compiled or generated with *EnumProcessModules* Windows API.

Return value

Pointer to string which is the name of the API located at supplied address. If there is no DLL export function at that address this string will be empty but the pointer to it will still be returned.

Remarks

String with the API name is stored inside the engine which makes this function multi thread unsafe. Process which is searched for the API is always the currently debugged process.

Example

None.

ImporterGetRemoteAPIAddress function

The **ImporterGetRemoteAPIAddress** function is used to realign the local API address to remote one inside the debugged process. This function is usefully in cases when local and remote DLL are not loaded on the same base address or in case of ASLR. Keep in mind that your process might not have loaded all the remote DLL files so that this function cannot be used in case that module in which the API resides isn't loaded.

Syntax

```
long long __stdcall ImporterGetRemoteAPIAddress(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process whose modules will be searched for the supplied API.

APIAddress

[in] Address on which API is located. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest.

Return value

Realigned API address matching the API address inside the debugged process.

Remarks

None.

Example

None.

ImporterGetRemoteAPIAddressEx function

The **ImporterGetRemoteAPIAddressEx** function is used to retrieve the remote API address from any module that debugged process has loaded. There is no need to have the remote DLL loaded locally in order to use this function.

Syntax

```
long long __stdcall ImporterGetRemoteAPIAddressEx(
            char* szDLLName,
            char* szAPIName
            );
```

Parameters

szDLLName

[in] Name of the remote DLL file which contains the needed API.

szAPIName

[in] Name of the API inside the remote DLL whose address inside the remote process will be returned.

Return value

Remote API address matching the API address inside the debugged process.

Remarks

None.

Example

None.

ImporterGetLocalAPIAddress function

The **ImporterGetLocalAPIAddress** function is used relocate the remote API address to local one. This is used when the remote module is loaded inside the debugger and you need to know the local address of that remote API, which is usefully in cases when local and remote DLL are not loaded on the same base address or in case of ASLR.

Syntax

```
long long __stdcall ImporterGetLocalAPIAddress(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process whose modules will be searched for the supplied API.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

Return value

Local API address for the remotely found API or *NULL* if that API can't be found in your process.

Remarks

None.

Example

None.

ImporterGetDLLNameFromDebugee function

The **ImporterGetDLLNameFromDebugee** function is used to get the name of the remote DLL which has the selected API. Address of the API is the remote one so this function can be used to query if the remote address is an API pointer.

Syntax

```
void* __stdcall ImporterGetDLLNameFromDebugee(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process whose modules will be searched for the supplied API.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

Return value

Pointer to string which is the name of the DLL that holds API located at supplied address. If there is no DLL export function at that address this string will be empty but the pointer to it will still be returned.

Remarks

String with the DLL name is stored inside the engine which makes this function multi thread unsafe.

Example

None.

ImporterGetAPINameFromDebugee function

The **ImporterGetAPINameFromDebugee** function is used to resolve the API name for the remote process API pointer. Address of the API is the remote one so this function can be used to query if the remote address is an API pointer.

Syntax

```
void* __stdcall ImporterGetAPINameFromDebugee(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process whose modules will be searched for the supplied API.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

Return value

Pointer to string which is the name of the API that is located at supplied address. If there is no DLL export function at that address this string will be empty but the pointer to it will still be returned.

Remarks

String with the API name is stored inside the engine which makes this function multi thread unsafe.

Example

None.

ImporterGetDLLIndexEx function

The **ImporterGetDLLIndexEx** function is used to identify in which DLL of the module list selected API is located. Process which is searched for the API is always the currently debugged process.

Syntax

```
long __stdcall ImporterGetDLLIndexEx(
            ULONG_PTR APIAddress,
            ULONG_PTR DLLBasesList
            );
```

Parameters

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

DLLBasesList

[in] Pointer to array of module base addresses inside the remote process. This list is either manually compiled or generated with *EnumProcessModules* Windows API.

Return value

Function returns the index of the DLL in the list to which selected API belongs to or *NULL* if API isn't valid or found in the provided DLL list.

Remarks

List contains items with indexes going from zero to list count but first (zero) item is ignored because it is usually the base address of the debugged executable module. If API is found in the provided list return can only be greater or equal to one.

Example

HMODULE mList[3] = {0x00400000, 0x7E000000, 0x7F000000};
ImporterGetDLLIndexEx(0x7E555105, & mList);

Function call would return one because the API on 0x7E555105 belongs to second module.

ImporterGetDLLIndex function

The **ImporterGetDLLIndex** function is used to identify in which DLL of the module list selected API is located.

Syntax

```
long __stdcall ImporterGetDLLIndex(
            HANDLE hProcess,
            ULONG_PTR APIAddress,
            ULONG_PTR DLLBasesList
            );
```

Parameters

hProcess

[in] Handle of the process whose modules will be searched for the supplied API.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

DLLBasesList

[in] Pointer to array of module base addresses inside the remote process. This list is either manually compiled or generated with *EnumProcessModules* Windows API.

Return value

Function returns the index of the DLL in the list to which selected API belongs to or *NULL* if API isn't valid or found in the provided DLL list.

Remarks

List contains items with indexes going from zero to list count but first (zero) item is ignored because it is usually the base address of the debugged executable module. If API is found in the provided list return can only be greater or equal to one.

Example

None.

ImporterGetRemoteDLLBase function

The **ImporterGetRemoteDLLBase** function is used to get the remote DLL base for a locally loaded DLL file. In this case debugger and the debugged process load the same module but due to Windows nature those two can be loaded on two different base addresses so we use this function to resolve the module base for the remote process. It is commonly used to get remote module bases for system DLL files in ASLR environment.

Syntax

```
long long __stdcall ImporterGetRemoteDLLBase(
            HANDLE hProcess,
            HMODULE LocalModuleBase
            );
```

Parameters

hProcess

[in] Handle of the process which will be queried for local module presence.

LocalModuleBase

[in] Handle of the local DLL file which will be searched for in the remote process.

Return value

Function returns the remote DLL base for the locally loaded module or *NULL* if module isn't found.

Remarks

None.

Example

None.

ImporterGetRemoteDLLBaseEx function

The **ImporterGetRemoteDLLBaseEx** function is used to get the remote DLL base for a specified file. This function does not require that the remote module is loaded locally.

Syntax

```
long long __stdcall ImporterGetRemoteDLLBaseEx(
            HANDLE hProcess,
            char* szModuleName
            );
```

Parameters

hProcess

[in] Handle of the process which will be queried for local module presence.

szModuleName

[in] Name of the module inside the remote process whose loaded base address will be returned.

Return value

Function returns the remote DLL base for the specified module or *NULL* if module isn't found.

Remarks

None.

Example

None.

ImporterIsForwardedAPI function

The **ImporterIsForwardedAPI** function is used to check if the supplied API address is forwarded from another top level dynamic link library to lower system one. Usually forwarders can be found in **kernel32.dll** which are forwarded to **ntdll.dll**. These APIs are automatically resolved to correct APIs by the engine itself.

Syntax

```
bool __stdcall ImporterIsForwardedAPI(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process which will be inspected for API forwarding.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

Return value

Function returns *TRUE* if API is forwarded and *FALSE* if it isn't.

Remarks

None.

Example

hModule = GetModuleHandleA("ntdll.dll");
ImporterIsForwardedAPI(hProcess, GetProcAddress(hModule, "RtlAllocateHeap"));

Function would return *TRUE* because this API is a forward for **kernel32.HeapAlloc**

ImporterGetForwardedAPIName function

The **ImporterGetForwardedAPIName** function is used to retrieve the name of the forwarded API.

Syntax

```
void* __stdcall ImporterGetForwardedAPIName(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process which will be inspected for API forwarding.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

Return value

Pointer to string which is the name of the API forwarded for supplied address. If DLL export function at that address isn't a forwarder this string will be empty but the pointer to it will still be returned.

Remarks

None.

Example

hModule = GetModuleHandleA("ntdll.dll");
ImporterGetForwardedAPIName(hProcess, GetProcAddress(hModule, "RtlAllocateHeap"));

Function would return *HeapAlloc* because this API is a forward for **kernel32.HeapAlloc**

ImporterGetForwardedDLLName function

The **ImporterGetForwardedDLLName** function is used to retrieve the name of DLL which holds the forwarded API.

Syntax

```
void* __stdcall ImporterGetForwardedDLLName(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process which will be inspected for API forwarding.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

Return value

Pointer to string which is the name of the DLL which holds that API forwarded for supplied address. If DLL export function at that address isn't a forwarder this string will be empty but the pointer to it will still be returned.

Remarks

None.

Example

hModule = GetModuleHandleA("ntdll.dll");
ImporterGetForwardedDLLName(hProcess, GetProcAddress(hModule, "RtlAllocateHeap"));

Return would be *kernel32.dll* because this API is in that DLL as a forward for **kernel32.HeapAlloc**

ImporterGetForwardedDLLIndex function

The **ImporterGetForwardedDLLIndex** function is used to retrieve the index of the DLL in the module list which holds the forwarded API.

Syntax

```
long __stdcall ImporterGetForwardedDLLIndex(
            HANDLE hProcess,
            ULONG_PTR APIAddress,
            ULONG_PTR DLLBasesList
            );
```

Parameters

hProcess

[in] Handle of the process which will be inspected for API forwarding.

APIAddress

[in] Address on which API is located in the remote process. This address is equal to address returned by *GetProcAddress* Windows API for DLL and function of interest if called in debugged process.

DLLBasesList

[in] Pointer to array of module base addresses inside the remote process. This list is either manually compiled or generated with *EnumProcessModules* Windows API.

Return value

Function returns the index of the DLL in the list to which resolved API forwarder belongs to or *NULL* if API isn't valid or found in the provided DLL list.

Remarks

List contains items with indexes going from zero to list count but first (zero) item is ignored because it is usually the base address of the debugged executable module. If API is found in the provided list return can only be greater or equal to one.

Example

None.

ImporterFindAPIWriteLocation function

The **ImporterFindAPIWriteLocation** function is used to search through the list of added APIs by *ImporterAddNewAPI* in order to locate the trunk location for already added API.

Syntax

```
long long __stdcall ImporterFindAPIWriteLocation(
            char* szAPIName
            );
```

Parameters

szAPIName

[in] Pointer to string which is the name of the API on which has been added to import tree, for example ***VirtualAlloc***.

Return value

Function returns the address on which the import trunk for selected API is written or *NULL* is that API wasn't added to import tree.

Remarks

None.

Example

None.

ImporterFindAPIByWriteLocation function

The **ImporterFindAPIByWriteLocation** function is used to search through the list of added APIs by *ImporterAddNewAPI* in order to locate the name of the added API by using its trunk location.

Syntax

```
long long __stdcall ImporterFindAPIByWriteLocation(
            ULONG_PTR APIWriteLocation
            );
```

Parameters

APIWriteLocation

[in] Trunk location on which pointer to selected API will be written. This was the trunk value passed to *ImporterAddNewAPI* or *ImporterAddNewDLL* at the time of adding that API.

Return value

Function returns pointer to string which is the name of the API for supplied trunk address or *NULL* if API isn't found.

Remarks

None.

Example

None.

ImporterFindDLLByWriteLocation function

The **ImporterFindDLLByWriteLocation** function is used to search through the list of added APIs by *ImporterAddNewAPI* in order to locate the name of the DLL to which the added API belongs to.

Syntax

```
long long __stdcall ImporterFindDLLByWriteLocation(
            ULONG_PTR APIWriteLocation
            );
```

Parameters

APIWriteLocation

[in] Trunk location on which pointer to selected API will be written. This was the trunk value passed to *ImporterAddNewAPI* or *ImporterAddNewDLL* at the time of adding that API.

Return value

Function returns pointer to string which is the name of the DLL which holds the API for supplied trunk address or *NULL* if API isn't found.

Remarks

None.

Example

None.

ImporterGetNearestAPIAddress function

The **ImporterGetNearestAPIAddress** function is used to estimate the correct API by closeness to provided API. This is useful if by tracing you get to the address which is inside the API itself but it is unknown how many instructions the API has before the one you are on.

Syntax

```
long long __stdcall ImporterGetNearestAPIAddress(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process which will be inspected for correct API estimation.

APIAddress

[in] Address near the possible API start in the remote process.

Return value

Function returns the address of the nearest possible API or *NULL* if no close API can be found.

Remarks

None.

Example

None.

ImporterGetNearestAPIName function

The **ImporterGetNearestAPIName** function is used to estimate the correct API by closeness to provided API. This is useful if by tracing you get to the address which is inside the API itself but it is unknown how many instructions the API has before the one you are on.

Syntax

```
void* __stdcall ImporterGetNearestAPIName(
            HANDLE hProcess,
            ULONG_PTR APIAddress
            );
```

Parameters

hProcess

[in] Handle of the process which will be inspected for correct API estimation.

APIAddress

[in] Address near the possible API start in the remote process.

Return value

Pointer to string which is the name of the nearest API to supplied address. If there is no API close to that address this string will be empty but the pointer to it will still be returned.

Remarks

String with the API name is stored inside the engine which makes this function multi thread unsafe.

Example

None.

ImporterMoveIAT function

The **ImporterMoveIAT** function is used to set on a switch which will make importer export the import table in a specific fashion. This function ensures that strings are written after the import tree which is important in the case of fixing import eliminations when we don't know where APIs will be written. If that is the case all APIs will need to be added to the tree with relative addresses starting from NULL and incrementing by four (or eight for x64) or double that value if we need to write a pointer for new DLL. This data will later be relocated to match the new section in which it will be written. As by default strings are written to the section first function *ImporterMoveIAT* must be called to move those strings behind the pointers which will be written at the beginning of that section. This functionality has been added to automatic import fixing functions, so example of using this kind of model can be seen in the *TitanEngine* source code.

Syntax

```
void __stdcall ImporterMoveIAT();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ImporterRelocateWriteLocation function

The **ImporterRelocateWriteLocation** function is used only in specific situation describe with function *ImporterMoveIAT* to relocate all import data by the same value. This value can be the offset of the newly added section or a code cave inside the file.

Syntax

```
bool __stdcall ImporterRelocateWriteLocation(
            ULONG_PTR AddValue
            );
```

Parameters

AddValue

[in] Value which will be added to every import tree entry added with engine functions *ImporterAddNewAPI* and/or *ImporterAddNewDLL*.

Return value

Function returns *TRUE* on success or *FALSE* if function fails to relocate added data.

Remarks

None.

Example

None.

ImporterSetUnknownDelta function

The **ImporterSetUnknownDelta** function is used only in specific situation describe with function *ImporterMoveIAT* to relocate all import data by the same value. This value can be the offset of the newly added section or a code cave inside the file.

Syntax

```
void __stdcall ImporterSetUnknownDelta(
            ULONG_PTR DeltaAddress
            );
```

Parameters

DeltaAddress

[in] Value which will be used as a temporary import tree entry offset added with engine functions *ImporterAddNewAPI* and/or *ImporterAddNewDLL*.

Return value

None.

Remarks

None.

Example

None.

ImporterGetCurrentDelta function

The **ImporterGetCurrentDelta** function is used only in specific situation describe with function *ImporterMoveIAT* to relocate all import data by the same value. This function returns the current delta which will be used for writing new virtual trunk.

Syntax

```
long long __stdcall ImporterGetCurrentDelta();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ImporterLoadImportTable function

The **ImporterLoadImportTable** function is used to load import table from any PE file. Loaded table will be converted to internal engine import tree making it available for further modifications before being exported to the same or any other PE file.

Syntax

```
bool __stdcall ImporterLoadImportTable(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose import table will be loaded by the engine.

Return value

Function returns *TRUE* on success or *FALSE* if function fails to load data.

Remarks

None.

Example

None.

ImporterCopyOriginalIAT function

The **ImporterCopyOriginalIAT** function is used to copy IAT from one file to another. This function assumes that the IAT will be on the same virtual location in both files so it is only used in cases when you dynamically unpack crypters and that format doesn't handle imports by itself. Instead it leaves the import table handling to Windows loader just like if the file wasn't packed.

Syntax

```
bool __stdcall ImporterCopyOriginalIAT(
            char* szOriginalFile,
            char* szDumpFile
            );
```

Parameters

szOriginalFile

[in] Pointer to a null terminated string which is a full path to file from which import table will be copied.

szDumpFile

[in] Pointer to a null terminated string which is a full path to file to which import table will be copied to.

Return value

Function returns *TRUE* on success or *FALSE* if function fails to copy the import table.

Remarks

None.

Example

None.

ImporterMoveOriginalIAT function

The **ImporterMoveOriginalIAT** function is used to move IAT from one file to another. This function doesn't actually modify the original file but it loads the import table and exports it to selected dump file.

Syntax

```
bool __stdcall ImporterCopyOriginalIAT(
            char* szOriginalFile,
            char* szDumpFile,
            char* szSectionName
            );
```

Parameters

szOriginalFile

[in] Pointer to a null terminated string which is a full path to file from which import table will be copied.

szDumpFile

[in] Pointer to a null terminated string which is a full path to file to which import table will be copied to.

szSectionName

[in] Pointer to a null terminated string which will be the new section name which will hold the import table. This string can only be 8 characters long.

Return value

Function returns *TRUE* on success or *FALSE* if function fails to move the import table.

Remarks

None.

Example

None.

ImporterAutoSearchIAT function

The **ImporterAutoSearchIAT** function is used to automatically locate the possible import table location inside the packed file memory. Returns from this function can be used to automatically fix the import table for the selected file.

Syntax

```
void __stdcall ImporterAutoSearchIAT(
            HANDLE hProcess,
            char* szFileName,
            ULONG_PTR ImageBase,
            ULONG_PTR SearchStart,
            DWORD SearchSize,
            LPVOID pIATStart,
            LPVOID pIATSize
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be searched for import table.

szFileName

[in] Pointer to a null terminated string which is a full path to file whose content will be searched for import table. This file is a memory dump from the running process whose handle you have provided.

ImageBase

[in] Default image base of the targeted PE file dump on the disk.

SearchStart

[in] Virtual address inside the file which is used as a start marker for the search. It is safe to use virtual offset of the first section as a start position as only the code should be searched.

SearchSize

[in] Size of the memory to be searched for import pointers. It is safe to use NtSizeOfImage to search the whole file memory.

pIATStart

[out] Pointer to *ULONG_PTR* variable which will receive the virtual address on which the import table has been found.

pIATSize

[out] Pointer to *ULONG_PTR* variable which will receive the size of the found import table.

ImporterAutoSearchIATEx function

The **ImporterAutoSearchIATEx** function is used to automatically locate the possible import table location inside the packed file memory. This function will automatically dump the targeted process before searching for the import table. Returns from this function can be used to automatically fix the import table for the selected file.

Syntax

```
void __stdcall ImporterAutoSearchIATEx(
            HANDLE hProcess,
            ULONG_PTR ImageBase,
            ULONG_PTR SearchStart,
            DWORD SearchSize,
            LPVOID pIATStart,
            LPVOID pIATSize
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be searched for import table.

ImageBase

[in] Default image base of the targeted PE file dump on the disk.

SearchStart

[in] Virtual address inside the file which is used as a start marker for the search. It is safe to use virtual offset of the first section as a start position as only the code should be searched.

SearchSize

[in] Size of the memory to be searched for import pointers. It is safe to use NtSizeOfImage to search the whole file memory.

pIATStart

[out] Pointer to *ULONG_PTR* variable which will receive the virtual address on which the import table has been found.

pIATSize

[out] Pointer to *ULONG_PTR* variable which will receive the size of the found import table.

ImporterAutoFixIATEx function

The **ImporterAutoFixIATEx** function is used to automatically fix the import table for the running process. This function can fix all known redirections and import eliminations with the optional callback to manually fix unknown import pointers.

Syntax

```
long __stdcall ImporterAutoFixIATEx(
            HANDLE hProcess,
            char* szDumpedFile,
            char* szSectionName,
            bool DumpRunningProcess,
            bool RealignFile,
            ULONG_PTR EntryPointAddress,
            ULONG_PTR ImageBase,
            ULONG_PTR SearchStart,
            DWORD SearchSize,
            DWORD SearchStep,
            bool TryAutoFix,
            bool FixEliminations,
            LPVOID UnknownPointerFixCallback
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be searched for import table.

szDumpedFile

[in] Pointer to a null terminated string which is a full path to file which will contain the memory content if *DumpRunningProcess* is *TRUE* or already contains the dump memory content if *DumpRunningProcess* is *FALSE*.

DumpRunningProcess

[in] Boolean switch which indicates if file was dumped or not.

RealignFile

[in] Boolean switch which indicates if file need realigning after fixing imports or not.

EntryPointAddress

[in] Virtual address which will be set to the new file. Size of this variable varies, on x86 its 4 bytes and on x64 its 8 bytes. Therefore it can also be declared as void*.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

SearchStart

[in] Virtual address inside the file which is used as a start marker for the search. This is the return value of *ImporterAutoSearchIAT* function.

SearchSize

[in] Size of the memory to be searched for import pointers. This is the return value of *ImporterAutoSearchIAT* function.

SearchStep

[in] Search step is a value which will be used to iterate the search position. Default value is four (or eight on x64) and it will be used is you don't specify the search step and use *NULL*.

TryAutoFix

[in] Boolean switch which indicates whether or not to trace possible import pointers in order to fix the import table. This can be always set to *TRUE* but can be disabled in case you are sure that the target doesn't use import redirection.

FixEliminations

[in] Boolean switch which indicates whether or not to fix possible import eliminations in order to fix the import table. This can be always set to *TRUE* but can be disabled in case you are sure that the target doesn't use import elimination.

UnknownPointerFixCallback

[in] Pointer to callback which will be called for every possible but unknown import redirection or elimination. Use this callback to correct the import table fixing in case that import protection is not yet recognized by *TitanEngine*.

CallBack definition

```
typedef void*(__stdcall *fFixerCallback)(LPVOID fIATPointer);

// Returns API address in remote process or NULL
```

Return value

One of the following:

- **NULL** - Critical error! *just to be safe, but it should never happen
- **0x400** - Success
- **0x401** - Error, process terminated
- **0x404** - Error, memory could not be read
- **0x405** - Error, no API found
- **0x406** - Success, but realign failed

ImporterAutoFixIAT function

The **ImporterAutoFixIAT** function is used to automatically fix the import table for the running process. This function can fix all known redirections and import eliminations with the optional callback to manually fix unknown import pointers.

Syntax

```
long __stdcall ImporterAutoFixIAT(
            HANDLE hProcess,
            char* szDumpedFile,
            ULONG_PTR ImageBase,
            ULONG_PTR SearchStart,
            DWORD SearchSize,
            DWORD SearchStep
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be searched for import table.

szDumpedFile

[in] Pointer to a null terminated string which is a full path to file which already contains the dump memory content.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

SearchStart

[in] Virtual address inside the file which is used as a start marker for the search. This is the return value of *ImporterAutoSearchIAT* function.

SearchSize

[in] Size of the memory to be searched for import pointers. This is the return value of *ImporterAutoSearchIAT* function.

SearchStep

[in] Search step is a value which will be used to iterate the search position. Default value is four (or eight on x64) and it will be used is you don't specify the search step and use *NULL*.

Return value

See *ImporterAutoFixIATEx* function for return details.

Tracer module

Tracer module has functions designed for detecting and fixing import redirections. It has integrated import tracers and can also use ImpRec modules to fix known redirections.

TracerInit function

The **TracerInit** function is not used in *TitanEngine* and was retained only for minimal compatibility issued between versions 1.5 and latest.

Syntax

```
void __stdcall TracerInit();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

TracerLevel1 function

The **TracerLevel1** function is used to trace the provided address via code disassembling in order to try to find the correct API hiding behind selected import redirection. This function uses common code tracing in order to try to identify the API which is being redirected.

Syntax

```
long long __stdcall TracerLevel1(
            HANDLE hProcess,
            ULONG_PTR AddressToTrace
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be traced.

AddressToTrace

[in] Pointer to memory which holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

Return value

If return value is *NULL* the trace has failed, and if return value is grater then 0x1000 then the return is an API address inside the remote process. For cases where return is greater than *NULL* and lower then 0x1000 the return is the number of valid instructions detected while tracing and that info is used for *HashTracerLevel1* API.

Remarks

None.

Example

None.

HashTracerLevel1 function

The **HashTracerLevel1** function is used to trace the provided address via code hashing in order to try to find the correct API hiding behind selected import redirection. This function uses advanced code tracing in order to try to identify the API which is being redirected.

Syntax

```
long long __stdcall HashTracerLevel1(
            HANDLE hProcess,
            ULONG_PTR AddressToTrace,
            DWORD InputNumberOfInstructions
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be traced.

AddressToTrace

[in] Pointer to memory which holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

InputNumberOfInstructions

[in] Number of valid instructions detected while tracing with TraceLevel1.

Return value

If return value is *NULL* or minus one then trace has failed, and if return value is greater than *NULL* then the return is an API address inside the remote process.

Remarks

None.

Example

None.

TracerDetectRedirection function

The **TracerDetectRedirection** function is used to check if the memory at the selected address is equal to one of the known import redirection patterns.

Syntax

```
long __stdcall TracerDetectRedirection(
            HANDLE hProcess,
            ULONG_PTR AddressToTrace
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be checked for known import redirections.

AddressToTrace

[in] Pointer to memory which holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

Return value

If return value is *NULL* then no known redirection was detected, and if return value is greater than *NULL* then the return is an ID of the known redirection inside the internal *TitanEngine* import redirection database.

Remarks

None.

Example

None.

TracerFixKnownRedirection function

The **TracerFixKnownRedirection** function is used to fix known import redirection based on the known import redirection ID inside the *TitanEngine* database.

Syntax

```
long long __stdcall TracerFixKnownRedirection(
            HANDLE hProcess,
            ULONG_PTR AddressToTrace,
            DWORD RedirectionId
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be checked for known import redirections.

AddressToTrace

[in] Pointer to memory which holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

RedirectionId

[in] ID of the detected import redirection inside the *TitanEngine* database.

Return value

If return value is *NULL* the trace has failed, and if return value is grater then *NULL* then the return is an API address inside the remote process.

Remarks

None.

Example

None.

TracerFixRedirectionViaImpRecPlugin function

The **TracerFixRedirectionViaImpRecPlugin** function is used to fix import redirections with ImpRec modules. To be able to use this option you must install all needed ImpRec plugins by placing them in **.\import\ImpRec** folder.

Syntax

```
long __stdcall TracerFixRedirectionViaImpRecPlugin(
            HANDLE hProcess,
            char* szPluginName,
            ULONG_PTR AddressToTrace
            );
```

Parameters

hProcess

[in] Handle of the process whose import redirection will be fixed.

szPluginName

[in] Name of the ImpRec module inside the **.\import\ImpRec** folder. For example **aspr1.dll**

AddressToTrace

[in] Pointer to memory which holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

Return value

If return value is *NULL* the trace has failed, and if return value is grater then *NULL* then the return is an API address inside the remote process.

Remarks

This function only works on x86 systems since ImpRec and its plugins are designed that way.

Example

None.

Realigner module

Realigner module has functions designed for PE file validation, fixing and realigning.

Realigner module structures and constants

Constants used by: **IsPE32FileValidEx** function

```
#define UE_DEPTH_SURFACE 0
#define UE_DEPTH_DEEP 1
```

Structures used by: **IsPE32FileValidEx** function

```
typedef struct{
      BYTE OveralEvaluation;
      bool EvaluationTerminatedByException;
      bool FileIs64Bit;
      bool FileIsDLL;
      bool FileIsConsole;
      bool MissingDependencies;
      bool MissingDeclaredAPIs;
      BYTE SignatureMZ;
      BYTE SignaturePE;
      BYTE EntryPoint;
      BYTE ImageBase;
      BYTE SizeOfImage;
      BYTE FileAlignment;
      BYTE SectionAlignment;
      BYTE ExportTable;
      BYTE RelocationTable;
      BYTE ImportTable;
      BYTE ImportTableSection;
      BYTE ImportTableData;
      BYTE IATTable;
      BYTE TLSTable;
      BYTE LoadConfigTable;
      BYTE BoundImportTable;
      BYTE COMHeaderTable;
      BYTE ResourceTable;
      BYTE ResourceData;
      BYTE SectionTable;
}FILE_STATUS_INFO, *PFILE_STATUS_INFO;
```

Constants used by: **IsPE32FileValidEx** function and **FixBrokenPE32FileEx** function

```
#define UE_FIELD_OK 0
#define UE_FIELD_BROKEN_NON_FIXABLE 1
#define UE_FIELD_BROKEN_NON_CRITICAL 2
#define UE_FIELD_BROKEN_FIXABLE_FOR_STATIC_USE 3
#define UE_FIELD_BROKEN_BUT_CAN_BE_EMULATED 4
#define UE_FILED_FIXABLE_NON_CRITICAL 5
```


```
#define UE_FILED_FIXABLE_CRITICAL 6
#define UE_FIELD_NOT_PRESET 7
#define UE_FIELD_NOT_PRESET_WARNING 8
#define UE_RESULT_FILE_OK 10
#define UE_RESULT_FILE_INVALID_BUT_FIXABLE 11
#define UE_RESULT_FILE_INVALID_AND_NON_FIXABLE 12
#define UE_RESULT_FILE_INVALID_FORMAT 13
```

Structures used by: **FixBrokenPE32FileEx** function

```
typedef struct{
      BYTE OveralEvaluation;
      bool FixingTerminatedByException;
      bool FileFixPerformed;
      bool StrippedRelocation;
      bool DontFixRelocations;
      DWORD OriginalRelocationTableAddress;
      DWORD OriginalRelocationTableSize;
      bool StrippedExports;
      bool DontFixExports;
      DWORD OriginalExportTableAddress;
      DWORD OriginalExportTableSize;
      bool StrippedResources;
      bool DontFixResources;
      DWORD OriginalResourceTableAddress;
      DWORD OriginalResourceTableSize;
      bool StrippedTLS;
      bool DontFixTLS;
      DWORD OriginalTLSTableAddress;
      DWORD OriginalTLSTableSize;
      bool StrippedLoadConfig;
      bool DontFixLoadConfig;
      DWORD OriginalLoadConfigTableAddress;
      DWORD OriginalLoadConfigTableSize;
      bool StrippedBoundImports;
      bool DontFixBoundImports;
      DWORD OriginalBoundImportTableAddress;
      DWORD OriginalBoundImportTableSize;
      bool StrippedIAT;
      bool DontFixIAT;
      DWORD OriginalImportAddressTableAddress;
      DWORD OriginalImportAddressTableSize;
      bool StrippedCOM;
      bool DontFixCOM;
      DWORD OriginalCOMTableAddress;
      DWORD OriginalCOMTableSize;
}FILE_FIX_INFO, *PFILE_FIX_INFO;
```


RealignPE function

The **RealignPE** function is used to realign the PE file sections so that virtual and physical data are aligned to PECOFF specifications and extra data removed in order to minimize the file size. After file unpacking it is recommended to use this function to make the file a valid PE image.

Syntax

```
long __stdcall RealignPE(
            ULONG_PTR FileMapVA,
            DWORD FileSize,
            DWORD RealingMode
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

FileSize

[in] Size of the mapped file.

RealignMode

[in] Reserved for future use, set to *NULL* always.

Return value

This function returns the new realigned file size which can be used to trim the file during the unmapping process, or *NULL* if it fails.

Remarks

None.

Example

None.

RealignPEEx function

The **RealignPEEx** function is used to realign the PE file sections so that virtual and physical data are aligned to PECOFF specifications and extra data removed in order to minimize the file size. After file unpacking it is recommended to use this function to make the file a valid PE image.

Syntax

```
long __stdcall RealignPEEx(
            char* szFileName,
            DWORD RealingFileSize,
            DWORD ForcedFileAlignment
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be realigned.

RealignedFileSize

[in] Force the engine to make the file requested size. This option should only be used to increase the file size and not the other way around.

ForcedFileAlignment

[in] Specify *FileAlignment* manually, if no value is specified default 0x200 will be used.

Return value

This function returns the new realigned file size or *NULL* if it fails.

Remarks

None.

Example

None.

FixHeaderCheckSum function

The **FixHeaderCheckSum** function is used to recalculate the PE header field checksum which refers to checksum of the whole header. After new value is calculated selected file will be updated.

Syntax

```
bool __stdcall FixHeaderCheckSum(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose checksum will be updated.

Return value

This function returns *TRUE* on successful checksum update and *FALSE* if the function fails.

Remarks

None.

Example

None.

WipeSection function

The **WipeSection** function is used to remove the section from the file but preserve that virtual space by expanding nearby section.

Syntax

```
bool __stdcall WipeSection(
            char* szFileName,
            int WipeSectionNumber,
            bool RemovePhysically
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose section will be removed.

WipeSectionNumber

[in] Removes the selected PE section. Section numbers go from NULL to *SectionNumber* minus one.

RemovePhysically

[in] If set to *TRUE* file size will decrease because the section content will be removed from the file and in case this switch is set to FALSE content will remain in the file and only the PE header content will be changed.

Return value

This function returns *TRUE* on successful section wipe and *FALSE* if the function fails.

Remarks

None.

Example

None.

IsFileDLL function

The **IsFileDLL** function is used to determine if the selected file is a DLL by inspecting its PE header flags.

Syntax

```
bool __stdcall IsFileDLL(
            char* szFileName,
            ULONG_PTR FileMapVA
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose PE header will be checked to see if it is a DLL file.

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

Return value

This function returns *TRUE* if the selected file is a DLL and *FALSE* if it isn't.

Remarks

You can specify both input parameters but only one is required.

Example

None.

IsPE32FileValidEx function

The **IsPE32FileValidEx** function is used to determine if the selected file is a valid PE file and provide as much additional information about the file and detected errors.

Syntax

```
bool __stdcall IsPE32FileValidEx(
            char* szFileName,
            DWORD CheckDepth,
            LPVOID FileStatusInfo
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose PE header will be validated.

CheckDepth

[in] Indicates how detail the checks will be. Can be either `UE_DEPTH_SURFACE` or `UE_DEPTH_DEEP`.

FileStatusInfo

[out] Pointer to a `FILE_STATUS_INFO` structure which will be filled by this function.

Return value

This function returns *TRUE* if the function completes without critical errors in verification process which can occur in broken files and *FALSE* if there are errors during validation.

Remarks

OveralEvaluation member of the structure tells the state of the file.

Example

None.

FixBrokenPE32FileEx function

The **FixBrokenPE32FileEx** function is used to try to fix PE file errors and provide as much additional information about the file, both detected errors and fixed errors data.

Syntax

```
bool __stdcall FixBrokenPE32FileEx(
            char* szFileName,
            LPVOID FileStatusInfo,
            LPVOID FileFixInfo
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose PE header will be validated.

FileStatusInfo

[in] Pointer to a `FILE_STATUS_INFO` structure which was filled as a result of calling *IsPE32FileValidEx* function, or *NULL* which would make the engine call file validation function before fixing the file automatically.

FileFixInfo

[in & out] Pointer to a `FILE_FIX_INFO` structure which will be filed during file fixing but it also serves as an input parameter which tells the engine which tables not to fix or strip.

Return value

This function returns *TRUE* if the function completes without critical errors in file repairing process which can occur in broken files and *FALSE* if there are errors during fixing.

Remarks

OveralEvaluation member of the structure tells the state of the file after fixing.

Example

None.

Relocater module

Relocater module has functions designed for relocation manipulation, file memory snapshot making and relocation table removing. This module is used to fix relocation table when unpacking DLL files.

RelocaterInit function

The **RelocaterInit** function initializes the relocater module and it must be used before using any of the functions used in the process of manual relocation fixing.

Syntax

```
void __stdcall RelocaterInit(
            DWORD MemorySize,
            ULONG_PTR OldImageBase,
            ULONG_PTR NewImageBase
            );
```

Parameters

MemorySize

[in] Default memory size allocated for all relocations you add. This size must be large enough to hold all data needed by the engine. Usually there is no need to reserve more than 100kb of memory.

OldImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

NewImageBase

[in] Base address on which the targeted DLL file whose relocation table you are fixing is loaded.

Return value

None.

Remarks

None.

Example

None.

RelocaterCleanup function

The **RelocaterCleanup** function is used to free allocated memory by the engine for relocation fixing.

Syntax

```
void __stdcall RelocaterCleanup();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

RelocaterAddNewRelocation function

The **RelocaterAddNewRelocation** function is used to add an address from the remote process to list of addresses which need relocating if the file is allocated on base address other then default one. Just like when adding import via importer here you must add relocations one page at the time. Engine itself will take care of page switching but once the page is switched you can't go back to adding data to any of the previous ones.

Syntax

```
void __stdcall RelocaterAddNewRelocation(
            HANDLE hProcess,
            ULONG_PTR RelocateAddress,
            DWORD RelocateState
            );
```

Parameters

hProcess

[in] Handle of the process which has loaded the targeted DLL whose relocation table you are fixing.

RelocateAddress

[in] Address inside the remote process belonging to targeted DLL which need relocation in case of file being loaded on base address other than default.

RelocateState

[in] Reserved for future usage, for now always set to *NULL*.

Return value

None.

Remarks

None.

Example

None.

RelocaterEstimatedSize function

The **RelocaterEstimatedSize** function is used to estimate the needed space to write the relocation table to the file. This value can be used to determine the size of the new section in which the relocation data will be written.

Syntax

```
long __stdcall RelocaterEstimatedSize();
```

Parameters

None.

Return value

Returns the size needed to write the relocation data.

Remarks

None.

Example

None.

RelocaterExportRelocationEx function

The **RelocaterExportRelocationEx** function exports the added relocation data to existing PE file creating the valid relocation table for the selected PE file. After this function has executed relocation data will be cleared by using *RelocaterCleanup* function.

Syntax

```
bool __stdcall RelocaterExportRelocationEx(
            char* szFileName,
            char* szSectionName
            );
```

Parameters

szFileName

[in] Pointer to string which is a full path to the file to which new relocation table will be written. This file is usually created by using *DumpProcess* function.

szSectionName

[in] Name of the PE section in which the new relocation table content will be written in. This section will be added to the file. Length of this string is capped at 8 characters.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

RelocaterExportRelocation function

The **RelocaterExportRelocation** function exports the added relocation data to existing PE file creating the valid relocation table for the selected PE file. After this function has executed relocation data will be cleared by using *RelocaterCleanup* function.

Syntax

```
bool __stdcall RelocaterExportRelocation(
            ULONG_PTR StorePlace,
            DWORD StorePlaceRVA,
            ULONG_PTR FileMapVA
            );
```

Parameters

StorePlace

[in] Physical address inside PE file on which the new relocation table will be written. Usually this is a new section but can also be the part of the file which is unused but still in read/write mode.

StorePlaceRVA

[in] Relative virtual address inside PE file on which the new relocation table will be written. This input is just conversion from physical to relative virtual offset.

FileMapVA

[in] Pointer to the mapped file content which must be mapped in read/write mode. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

RelocaterGrabRelocationTableEx function

The **RelocaterGrabRelocationTableEx** function is used to copy PECOFF valid relocation table from targeted process to engine relocation data storage. This function can automatically determine the size the relocation table but the size parameter must be close to or higher than the actual relocation table size so that the targeted memory slice contains the end of the relocation table. If this function succeeds relocation table is ready to be exported.

Syntax

```
bool __stdcall RelocaterGrabRelocationTableEx(
            HANDLE hProcess,
            ULONG_PTR MemoryStart,
            ULONG_PTR MemorySize,
            DWORD NtSizeOfImage
            );
```

Parameters

hProcess

[in] Handle of the process from which the relocation table will be copied.

MemoryStart

[in] Pointer to memory in remote process which is the start of a valid relocation table.

MemorySize

[in] Size of the memory inside which the relocation table resides.

NtSizeOfImage

[in] PE header variable read from the file on the disk.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

RelocaterGrabRelocationTable function

The **RelocaterGrabRelocationTable** function is used to copy PECOFF valid relocation table from targeted process to engine relocation data storage. If this function succeeds relocation table is ready to be exported.

Syntax

```
bool __stdcall RelocaterGrabRelocationTable(
            HANDLE hProcess,
            ULONG_PTR MemoryStart,
            ULONG_PTR MemorySize
            );
```

Parameters

hProcess

[in] Handle of the process from which the relocation table will be copied.

MemoryStart

[in] Pointer to memory in remote process which is the start of a valid relocation table.

MemorySize

[in] Exact size of the relocation table inside the remote process.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

RelocaterMakeSnapshot function

The **RelocaterMakeSnapshot** function is used to copy the selected memory part to file on the disk. This memory part should contain all data that can and will be relocated by the packer itself and most commonly it covers all the executable code inside the PE files memory. By creating two snapshots, one just before the code which relocates the file inside the packer and one right after its execution you create two memory state images which can be used to create a valid relocation table by their comparing.

Syntax

```
bool __stdcall RelocaterMakeSnapshot(
            HANDLE hProcess,
            char* szSaveFileName,
            LPVOID MemoryStart,
            ULONG_PTR MemorySize
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be snapshot.

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be created and filled with the selected memory content.

MemoryStart

[in] Pointer to memory in remote process which is the start of a snapshot image. Start of the snapshot must remain the same for both snapshots.

MemorySize

[in] Size of the snapshot which must remain the same for both snapshots.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

RelocaterCompareTwoSnapshots function

The **RelocaterCompareTwoSnapshots** function is used to create a valid relocation table by comparing two created memory snapshots. One snapshot taken just before the relocation code execution inside the packer and another right after it. If this function succeeds relocation table is ready to be exported.

Syntax

```
bool __stdcall RelocaterCompareTwoSnapshots(
            HANDLE hProcess,
            ULONG_PTR LoadedImageBase,
            ULONG_PTR NtSizeOfImage,
            char* szDumpFile1,
            char* szDumpFile2,
            ULONG_PTR MemStart
            );
```

Parameters

hProcess

[in] Handle of the process whose memory will be snapshot.

LoadedImageBase

[in] Base address on which the targeted file is loaded.

NtSizeOfImage

[in] PE header variable read from the file on the disk for the targeted file.

MemoryStart

[in] Pointer to memory in remote process which was used as a start for a snapshot image.

szDumpFile1

[in] Pointer to a null terminated string which is a full path to file which was created as a memory snapshot number one.

szDumpFile2

[in] Pointer to a null terminated string which is a full path to file which was created as a memory snapshot number two.

Return value

Boolean switch indicating whether or not the function was successful.

RelocaterWipeRelocationTable function

The **RelocaterWipeRelocationTable** function is used to remove the relocation table from any PE file. However it is only recommended that you remove the relocation table from the executable files in order to reduce their size.

Syntax

```
bool __stdcall RelocaterWipeRelocationTable(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose relocation table will be removed.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

Exporter module

Exporter module has functions designed for export manipulation and new export table building.

ExporterInit function

The **ExporterInit** function initializes the importer module and it must be used before using any of the functions used in the process of manual import fixing.

Syntax

```
void __stdcall ExporterInit(
            DWORD MemorySize,
            ULONG_PTR ImageBase,
            DWORD ExportOrdinalBase,
            char* szExportModuleName
            );
```

Parameters

MemorySize

[in] Default memory size allocated for the entire export table data. This size must be large enough to hold all data needed by the engine. Usually there is no need to reserve more than 20kb of memory.

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

ExportOrdinalBase

[in] Sets the default ordinal base for the new export table. Default value is *NULL*.

szExportModuleName

[in] String which will be set as a default library name in the export table. For example: **mylib.dll**

Return value

None.

Remarks

None.

Example

None.

ExporterSetImageBase function

The **ExporterSetImageBase** function is used to update information passed to engine on exporter initialization.

Syntax

```
void __stdcall ExporterSetImageBase(
            ULONG_PTR ImageBase
            );
```

Parameters

ImageBase

[in] Default image base of the targeted PE file. This value should be read from the file on disk.

Return value

None.

Remarks

None.

Example

None.

ExporterCleanup function

The **ExporterCleanup** function clears all added exports. This resets the inputted data to original state. Before using the functions to add the export data you must initialize the importer again.

Syntax

```
void __stdcall ExporterCleanup();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ExporterAddNewExport function

The **ExporterAddNewExport** function is used to add new function to the export table.

Syntax

```
void __stdcall ExporterAddNewExport(
            char* szExportName,
            DWORD ExportRelativeAddress
            );
```

Parameters

szExportName

[in] Name of the functions which will be used to locate this function with APIs such as *GetProcAddress*.

ExportRelativeAddress

[in] Relative virtual address for the exported function. Location in the PE file on which the function you want to export is located on.

Return value

None.

Remarks

None.

Example

None.

ExporterAddNewOrdinalExport function

The **ExporterAddNewOrdinalExport** function is used to add new function to the export table.

Syntax

```
void __stdcall ExporterAddNewOrdinalExport(
            DWORD OrdinalNumber,
            DWORD ExportRelativeAddress
            );
```

Parameters

OrdinalNumber

[in] Function will be exported as an ordinal with the selected ordinal number.

ExportRelativeAddress

[in] Relative virtual address for the exported function. Location in the PE file on which the function you want to export is located on.

Return value

None.

Remarks

None.

Example

None.

ExporterGetAddedExportCount function

The **ExporterGetAddedExportCount** function gets the current number of added functions inside the export table.

Syntax

```
long __stdcall ExporterGetAddedExportCount();
```

Parameters

None.

Return value

Returns the number of added items inside the export table.

Remarks

None.

Example

None.

ExporterEstimatedSize function

The **ExporterEstimatedSize** function estimates the size of memory needed to write the export data. This value can be used to determine the size of the new section in which the export table will be written.

Syntax

```
long __stdcall ExporterEstimatedSize();
```

Parameters

None.

Return value

Returns the size needed to write the export data.

Remarks

None.

Example

None.

ExporterBuildExportTableEx function

The **ExporterBuildExportTableEx** function exports the added export data to existing PE file creating the valid export table for the selected PE file. After this function has executed export data will be cleared by using *ExporterCleanup* function.

Syntax

```
bool __stdcall ExporterBuildExportTableEx(
            char* szExportFileName,
            char* szSectionName
            );
```

Parameters

szExportFileName

[in] Pointer to string which is a full path to the file to which new export table will be written.

szSectionName

[in] Name of the PE section in which the new import table content will be written in. This section will be added to the file. Length of this string is capped at 8 characters.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ExporterBuildExportTable function

The **ExporterBuildExportTable** function exports the added export data to existing PE file creating the valid export table for the selected PE file. After this function has executed export data will be cleared by using *ExporterCleanup* function.

Syntax

```
bool __stdcall ImporterExportIAT(
            ULONG_PTR StorePlace,
            ULONG_PTR FileMapVA
            );
```

Parameters

StorePlace

[in] Physical address inside PE file on which the new export table will be written. Usually this is a new section but can also be the part of the file which is unused but still in read/write mode.

FileMapVA

[in] Pointer to the mapped file content which must be mapped in read/write mode. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ExporterLoadExportTable function

The **ExporterLoadExportTable** function is used to load export table from any PE file. Loaded table will be converted to internal engine export tree making it available for further modifications before being exported to the same or any other PE file.

Syntax

```
bool __stdcall ExporterLoadExportTable(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose export table will be loaded by the engine.

Return value

Function returns *TRUE* on success or *FALSE* if function fails to load data.

Remarks

None.

Example

None.

Resourcer module

Resourcer module has functions designed to load and access PE files in order to work with them and functions to extract those resources out.

ResourcerLoadFileForResourceUse function

The **ResourcerLoadFileForResourceUse** function is used to simulate the PE file loading in order to make all the virtual addresses equal to physical ones in the newly loaded file. This is different than file mapping because Windows PE loader behavior concerning PE file loading and storing sections in memory is simulated. No other function of the Windows PE loader other than that is emulated.

Syntax

```
long long __stdcall ResourcerLoadFileForResourceUse(
            char* szFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose loading will be a simulated PE loading so that all the virtual addresses match the newly loaded physical ones. This is a done by simulating the Windows PE loader.

Return value

Return is the base address of the newly loaded file or *NULL* is the file isn't a PE file or it couldn't be loaded.

Remarks

None.

Example

None.

ResourcerFreeLoadedFile function

The **ResourcerFreeLoadedFile** function is used to unload the previously loaded file by using *ResourcerLoadFileForResourceUse* function. Due to the nature of *TitanEngine* PE file loader simulator using this function is equal to using *VirtualFree* on selected memory range.

Syntax

```
bool __stdcall ResourcerFreeLoadedFile(
            LPVOID LoadedFileBase
            );
```

Parameters

LoadedFileBase

[in] Base address on which the file is loaded. This is usually a return value from *ResourcerLoadFileForResourceUse* function.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ResourcerExtractResourceFromFileEx function

The **ResourcerExtractResourceFromFileEx** function is used to extract resources from the loaded file to disk. This function goes through the resource tree of the loaded file and extracts the selected resource.

Syntax

```
bool __stdcall ResourcerExtractResourceFromFileEx(
            ULONG_PTR FileMapVA,
            char* szResourceType,
            char* szResourceName,
            char* szExtractedFileName
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping. This input variable is defined as ULONG_PTR which defines its size on x86 and x64 operating system. On x86 systems this variable is 4 bytes long and equal to DWORD, and on x64 platform this variable is 8 bytes long and equal to DWORD64, but since this is a pointer void* can also be used.

ResourceType

[in] Pointer to string which is a resource type identifier. If the resource type is an integer you must convert it to string. For this conversion you can use `MAKEINTRESOURCEA` macro available in **WinUser.h**

ResourceName

[in] Pointer to string which is a resource name identifier. If the resource type is an integer you must convert it to string. For this conversion you can use `MAKEINTRESOURCEA` macro available in **WinUser.h**

szExtractFileName

[in] Pointer to a null terminated string which is a full path to file which will be created and filled with selected resource memory content.

Return value

Boolean switch indicating whether or not the function was successful.

ResourcerExtractResourceFromFile function

The **ResourcerExtractResourceFromFile** function is used to extract resources from the loaded file to disk. This function goes through the resource tree of the loaded file and extracts the selected resource.

Syntax

```
bool __stdcall ResourcerExtractResourceFromFile(
            char* szFileName,
            char* szResourceType,
            char* szResourceName,
            char* szExtractedFileName
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file whose resource will be extracted.

ResourceType

[in] Pointer to string which is a resource type identifier. If the resource type is an integer you must convert it to string. For this conversion you can use `MAKEINTRESOURCEA` macro available in **WinUser.h**

ResourceName

[in] Pointer to string which is a resource name identifier. If the resource type is an integer you must convert it to string. For this conversion you can use `MAKEINTRESOURCEA` macro available in **WinUser.h**

szExtractFileName

[in] Pointer to a null terminated string which is a full path to file which will be created and filled with selected resource memory content.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

Static module

Static module has functions designed to load and access PE files in order to work with them or use predefined decryption behaviors in order to decrypt common crypters.

Static module constants

Constants used by: **StaticFileLoad** function

```
#define UE_ACCESS_READ 0
#define UE_ACCESS_WRITE 1
#define UE_ACCESS_ALL 2
```

Constants used by: **StaticMemoryDecrypt** function, **StaticMemoryDecryptEx** function and StaticSectionDecrypt function

```
#define UE_STATIC_DECRYPTOR_XOR 1
#define UE_STATIC_DECRYPTOR_SUB 2
#define UE_STATIC_DECRYPTOR_ADD 3

#define UE_STATIC_KEY_SIZE_1 1
#define UE_STATIC_KEY_SIZE_2 2
#define UE_STATIC_KEY_SIZE_4 4
#define UE_STATIC_KEY_SIZE_8 8
```


StaticFileLoad function

The **StaticFileLoad** function is used to either map the selected file or simulate its loading. Depending on the type of static unpacker being developed one of the two file memory content access is needed. File content can be changed in both cases with no difference to loading type you are using.

Syntax

```
bool __stdcall StaticFileLoad(
            char* szFileName,
            DWORD DesiredAccess,
            bool SimulateLoad,
            LPHANDLE FileHandle,
            LPDWORD LoadedSize,
            LPHANDLE FileMap,
            LPDWORD FileMapVA
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will be mapped or loaded.

DesiredAccess

[in] Determines the type of memory access you will have in case you are mapping the file. Can be one of the following: `UE_ACCESS_READ`, `UE_ACCESS_WRITE` and `UE_ACCESS_ALL`.

SimulateLoad

[in] Boolean switch indicating whether or not to simulate the Windows PE loader.

FileHandle

[out] Pointer to HANDLE variable which will receive the open file handle.

LoadedSize

[out] Pointer to DWORD variable which will receive the size of the mapped file.

FileMap

[out] Pointer to HANDLE variable which will receive the file mapping handle.

FileMapVA

[out] Pointer to HANDLE variable which will receive the file mapping handle.

Return value

Boolean switch indicating whether or not the function was successful.

StaticFileUnload function

The **StaticFileUnload** function is used to either map the selected file or simulate its loading. Depending on the type of static unpacker being developed one of the two file memory content access is needed. File content can be changed in both cases with no difference to loading type you are using.

Syntax

```
bool __stdcall StaticFileUnload(
            char* szFileName,
            bool CommitChanges,
            HANDLE FileHandle,
            DWORD LoadedSize,
            HANDLE FileMap,
            DWORD FileMapVA
            );
```

Parameters

szFileName

[in] Pointer to a null terminated string which is a full path to file which will receive the changed file content. In case of simulated load this parameter is mandatory if you need to save the changes to the file.

CommitChanges

[in] Boolean switch indicating whether or not to commit done changes to files loaded by Windows PE loader simulation.

FileHandle

[in] Handle of the mapped file returned by *StaticFileLoad*.

LoadedSize

[in] Size of the mapped file returned by *StaticFileLoad*. You can change the file size and by doing that either trim or grow its size.

FileMap

[in] File mapping handle returned by *StaticFileLoad*.

FileMapVA

[in] Base address on which the file is loaded returned by *StaticFileLoad*.

Return value

Boolean switch indicating whether or not the function was successful.

StaticMemoryDecrypt function

The **StaticMemoryDecrypt** function is used to decrypt the selected memory range with provided decryption key and decryption method.

Syntax

```
void __stdcall StaticMemoryDecrypt(
            LPVOID MemoryStart,
            DWORD MemorySize,
            DWORD DecryptionType,
            DWORD DecryptionKeySize,
            ULONG_PTR DecryptionKey
            );
```

Parameters

MemoryStart

[in] Pointer to memory in remote process which is a start of the memory block selected for decrypting.

MemorySize

[in] Size of the memory which will be decrypted with provided decryption key and selected decryption method.

DecryptionType

[in] Determines which decryption engine should use. One of the following decryption algorithms: `UE_STATIC_DECRYPTOR_XOR`, `UE_STATIC_DECRYPTOR_SUB` or `UE_STATIC_DECRYPTOR_ADD`.

DecryptionKeySize

[in] Determines the size of the decryption key to be used. Can only be one of the following key sizes: `UE_STATIC_KEY_SIZE_1`, `UE_STATIC_KEY_SIZE_2`, `UE_STATIC_KEY_SIZE_4` or `UE_STATIC_KEY_SIZE_8`.

DecryptionKey

[in] Key which will used for memory decryption.

Return value

None.

Example

```
StaticMemoryDecrypt(0x00401000, 0x1000, UE_STATIC_DECRYPTOR_XOR, UE_STATIC_KEY_SIZE_1, 0x90);
```


StaticMemoryDecryptEx function

The **StaticMemoryDecryptEx** function is used to decrypt the selected memory range with a custom decryption algorithm. Selected callback will be called for each member of the encrypted block.

Syntax

```
void __stdcall StaticMemoryDecryptEx(
            LPVOID MemoryStart,
            DWORD MemorySize,
            DWORD DecryptionKeySize,
            void* DecryptionCallBack
            );
```

Parameters

MemoryStart

[in] Pointer to memory in remote process which is a start of the memory block selected for decrypting.

MemorySize

[in] Size of the memory which will be decrypted with the custom decryption callback.

DecryptionKeySize

[in] Determines the size of the decryption key to be used. Can be custom or one of the following key sizes: `UE_STATIC_KEY_SIZE_1`, `UE_STATIC_KEY_SIZE_2`, `UE_STATIC_KEY_SIZE_4` or `UE_STATIC_KEY_SIZE_8`. In case of a custom size make sure the *MemorySize* % *DecryptionKeySize* is NULL. If modus isn't *NULL* last few bytes of the memory content will not be decrypted.

DecryptionCallBack

[in] Callback which will decrypt the targeted memory slice. It is called for each member of the encrypted block with the block pointer increasing by the size of the decryption key.

CallBack definition

```
typedef bool(__stdcall *fStaticCallBack)(void* sMemoryStart, int sKeySize);
```

Return value

None.

StaticSectionDecrypt function

The **StaticSectionDecrypt** function is used to decrypt the selected memory range with a custom decryption algorithm. Selected callback will be called for each member of the encrypted block.

Syntax

```
void __stdcall StaticSectionDecrypt(
            ULONG_PTR FileMapVA,
            DWORD SectionNumber,
            bool SimulateLoad,
            DWORD DecryptionType,
            DWORD DecryptionKeySize,
            ULONG_PTR DecryptionKey
            );
```

Parameters

FileMapVA

[in] Pointer to the mapped file content. This pointer is set by using either *StaticFileLoad* function or Windows API for file mapping.

SectionNumber

[in] Number of the section which will be decrypted. Section numbers go from *zero* to section count minus one.

SimulatedLoad

[in] Boolean switch indicating whether or not the file was loaded by simulating Windows PE loader. If *FALSE* engine assumes that the file was mapped.

DecryptionType

[in] Determines which decryption engine should use. One of the following decryption algorithms: `UE_STATIC_DECRYPTOR_XOR`, `UE_STATIC_DECRYPTOR_SUB` or `UE_STATIC_DECRYPTOR_ADD`.

DecryptionKeySize

[in] Determines the size of the decryption key to be used. Can only be one of the following key sizes: `UE_STATIC_KEY_SIZE_1`, `UE_STATIC_KEY_SIZE_2`, `UE_STATIC_KEY_SIZE_4` or `UE_STATIC_KEY_SIZE_8`.

DecryptionKey

[in] Key which will used for memory decryption.

Return value

None.

Handler module

Handler module has functions designed to work with open handles and mutexes. Additionally it can find processes which use designated mutex or close all lock handles to selected files.

Handler module structures

Structures used by: **HandlerEnumerateOpenHandles** function and **HandlerEnumerateLockHandles** function

```
typedef struct{
      ULONG ProcessId;
      HANDLE hHandle;
}HandlerArray, *PHandlerArray;
```


HandlerGetActiveHandleCount function

The **HandlerGetActiveHandleCount** function is used to get the number of open handles inside the selected process.

Syntax

```
long __stdcall HandlerGetActiveHandleCount(
            DWORD ProcessId
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

HandlerIsHandleOpen function

The **HandlerIsHandleOpen** function is used to check if the remote handle is still open.

Syntax

```
bool __stdcall HandlerIsHandleOpen(
            DWORD ProcessId,
            HANDLE hHandle
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

hHandle

[in] Handle inside the remote process whose state will be queried.

Return value

Boolean switch indicating whether or not the handle is still open.

Remarks

None.

Example

None.

HandlerGetHandleName function

The **HandlerGetHandleName** function is used to retrieve the name of remotely open handle.

Syntax

```
void* __stdcall HandlerGetHandleName(
            HANDLE hProcess,
            DWORD ProcessId,
            HANDLE hHandle,
            bool TranslateName
            );
```

Parameters

hProcess

[in] Handle of the process whose handle info is needed.

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

hHandle

[in] Handle inside the remote process whose name will be queried.

TranslateName

[in] Boolean switch indicating whether or not to translate the name of the handle to non native name. Names or paths which contain physical devices in their file references are resolved with this function.

Return value

Function returns a pointer to handle name, or NULL if the supplied string can't be retrieved.

Remarks

String with the translated native name is stored inside the engine which makes this function multi thread unsafe.

Example

None.

HandlerGetHandleDetails function

The **HandlerGetHandleDetails** function is used to retrieve additional information about the remotely open handle.

Syntax

```
long long __stdcall HandlerGetHandleDetails(
            HANDLE hProcess,
            DWORD ProcessId,
            HANDLE hHandle,
            DWORD InformationReturn
            );
```

Parameters

hProcess

[in] Handle of the process whose handle info is needed.

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

hHandle

[in] Handle inside the remote process whose name will be queried.

InformationReturn

[in] Defines the type of the return information for the selected remote handle. It can be one of the following:

- `UE_OPTION_HANDLER_RETURN_HANDLECOUNT`
- `UE_OPTION_HANDLER_RETURN_ACCESS`
- `UE_OPTION_HANDLER_RETURN_FLAGS`
- `UE_OPTION_HANDLER_RETURN_TYPENAME`

Return value

Function can return a DWORD value of the selected handle property or a pointer to handle name depending on the *InformationReturn* value.

Remarks

String with the translated native name is stored inside the engine which makes this function multi thread unsafe.

HandlerEnumerateOpenHandles function

The **HandlerEnumerateOpenHandles** function is used to get the information about all open handles for the selected process inside one array.

Syntax

```
long __stdcall HandlerEnumerateOpenHandles(
            DWORD ProcessId,
            LPVOID HandleBuffer,
            DWORD MaxHandleCount
            );
```

Parameters

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

HandleDataBuffer

[out] Pointer to array which will receive the lock handle information. Array is defined as stated in the *Handler module structures*.

MaxHandleCount

[in] Defines the maximum possible entries in the array.

Return value

Returns the number of open handles found.

Remarks

None.

Example

None.

HandlerIsFileLocked function

The **HandlerIsFileLocked** function is used to check if the selected file or folder is locked by open handles from any of the running processes.

Syntax

```
bool __stdcall HandlerIsFileLocked(
            char* szFileOrFolderName,
            bool NameIsFolder,
            bool NameIsTranslated
            );
```

Parameters

szFileOrFolderName

[in] Pointer to a null terminated string which is a full path to file or folder which will be checked for locking handles.

NameIsFolder

[in] Boolean switch indicating if the provided string is a folder.

NameIsTranslated

[in] Boolean switch indicating if the string has already been translated.

Return value

Boolean switch indicating whether or not the file or folder is locked.

Remarks

None.

Example

None.

HandlerCloseAllLockHandles function

The **HandlerCloseAllLockHandles** function is used to check if the selected file or folder is locked by open handles from any of the running processes and if it is to try to close all of them regardless of the process locking the file or folder. Use this function with caution because it can lead to application crash if those applications rely on those handles.

Syntax

```
bool __stdcall HandlerCloseAllLockHandles(
            char* szFileOrFolderName,
            bool NameIsFolder,
            bool NameIsTranslated
            );
```

Parameters

szFileOrFolderName

[in] Pointer to a null terminated string which is a full path to file or folder which will be checked for locking handles and whose lock handles will be closed.

NameIsFolder

[in] Boolean switch indicating if the provided string is a folder.

NameIsTranslated

[in] Boolean switch indicating if the string has already been translated.

Return value

Boolean switch indicating whether or not the file or folder is still locked.

Remarks

None.

Example

None.

HandlerEnumerateLockHandles function

The **HandlerEnumerateLockHandles** function is used to get the information about all file or folder locking handles inside one array.

Syntax

```
long __stdcall HandlerEnumerateLockHandles(
            char* szFileOrFolderName,
            bool NameIsFolder,
            bool NameIsTranslated,
            LPVOID HandleDataBuffer,
            DWORD MaxHandleCount
            );
```

Parameters

szFileOrFolderName

[in] Pointer to a null terminated string which is a full path to file or folder which will be checked for locking handles.

NameIsFolder

[in] Boolean switch indicating if the provided string is a folder.

NameIsTranslated

[in] Boolean switch indicating if the string has already been translated.

HandleDataBuffer

[out] Pointer to array which will receive the lock handle information. Array is defined as stated in the *Handler module structures*.

MaxHandleCount

[in] Defines the maximum possible entries in the array.

Return value

Returns the number of lock handles found.

Remarks

None.

Example

None.

HandlerCloseRemoteHandle function

The **HandlerCloseRemoteHandle** function is used to close handles in remote processes. Use this function with caution because it can lead to application crash if those applications rely on those handles.

Syntax

```
bool __stdcall HandlerCloseRemoteHandle(
            HANDLE hProcess,
            HANDLE hHandle
            );
```

Parameters

hProcess

[in] Handle of the process whose handle will be closed.

hHandle

[in] Handle inside the remote process which will be closed.

Return value

Boolean switch indicating whether or not the handle has closed.

Remarks

None.

Example

None.

HandlerEnumerateOpenMutexes function

The **HandlerEnumerateOpenMutexes** function is used to get the information about all open mutex handles for selected process inside one array.

Syntax

```
long __stdcall HandlerEnumerateOpenMutexes(
            HANDLE hProcess,
            DWORD ProcessId,
            LPVOID HandleBuffer,
            DWORD MaxHandleCount
            );
```

Parameters

hProcess

[in] Handle of the process whose mutexes will be enumerated.

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

HandleDataBuffer

[out] Pointer to array which will receive the open mutex handle information.

MaxHandleCount

[in] Defines the maximum possible entries in the array.

Return value

Returns the number of open mutex handles found.

Remarks

Array is defined as array of HANDLEs.

Example

None.

HandlerGetOpenMutexHandle function

The **HandlerGetOpenMutexHandle** function is used to get the handle for the remotely opened mutex.

Syntax

```
long long __stdcall HandlerGetOpenMutexHandle(
            HANDLE hProcess,
            DWORD ProcessId,
            char* szMutexString
            );
```

Parameters

hProcess

[in] Handle of the process whose mutexes will be enumerated.

ProcessId

[in] Process ID of the running process which can be acquired with Windows API.

szMutexString

[in] Pointer to string which is the mutex whose handle will be returned.

Return value

Returns the handle inside the remote process for the selected mutex or *NULL* if mutex isn't found.

Remarks

None.

Example

None.

HandlerGetProcessIdWhichCreatedMutex function

The **HandlerGetProcessIdWhichCreatedMutex** function is used to get the process ID which has opened the selected mutex.

Syntax

```
long __stdcall HandlerGetProcessIdWhichCreatedMutex(
            char* szMutexString
            );
```

Parameters

szMutexString

[in] Pointer to string which is the mutex whose presence will be queried in all of the running processes.

Return value

Returns the ID of the process which has opened the selected mutex.

Remarks

None.

Example

None.

License

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

a) under this License, provided that you make a good faith effort to
ensure that, in the event an Application does not supply the
function or data, the facility still operates, and performs
whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of
this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from
a header file that is part of the Library. You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the
Library is used in it and that the Library and its use are
covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license
document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.