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 400 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*

ReversingLabs Corporation

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*

ReversingLabs Corporation

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*.

1. EIP inside sections
2. Already written to
3. Hash has changed
4. Not a simple redirection

ReversingLabs Corporation

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 fairly 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 target's 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 the Original Entry Point (OEP), dump the file and paste imports to it.

To start 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 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 breakpoint's 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 at 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 the 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 the *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 the *LoadLibrary/GetModuleHandle* callbacks, 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 in 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 the importer engine. But unlike *ImporterAddNewDll* API, you must specify the location on which the API pointer will be stored. This is a memory location to which the 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's 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 the dump file. To know just how much space you need, use the *ImporterEstimatedSize* API. Finally the IAT is exported to a new section by calling the *ImporterExportIAT* API. Optionally you can realign the file and then 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 your program's execution resumes after the call to *DebugLoop*.

```
GetPEData

Unpacking init
  InitDebug
  SetBPX
  DebugLoop();

SetBPX

StopDebug();

Import fix
  Gather import data
    LoadLibrary        → ImporterAddNewDLL
    GetProcAddress     → ImporterAddNewAPI

Relocation fix
  Packer segment
    Code just before   → RelocaterMakeSnaphot
    Relocation code
    Code just after    → RelocaterMakeSnaphot
                           → CompareTwoSnapshots

Entry point
  DumpProcess
  PastePEHeader
  ImporterExportIAT
  ExportRelocation
  RealignPE
```

ReversingLabs Corporation

TitanEngine SDK References

ReversingLabs Corporation

Unicode support

Unicode support has been added to *TitanEngine* with version 2.0.2. However Unicode functions are not documented in this document because changes between function versions that use ASCII or UNICODE strings as input/output parameters are minor. Unicode functions are defined in the SDK and can be used normally. Such functions can be easily recognized by the appendix "W" which they have. For specific function definitions please refer to the SDK header files.

Python support

Python support has been added to *TitanEngine* with version 2.0.3. To use the python SDK include teSDK.py with your project. SDK wrapper around the engine uses the default python libraries which ship with Python 2.6.5.

LUA support

LUA support has been added to *TitanEngine* with version 2.0.3. To use the LUA SDK include SDK.lua with your project. SDK wrapper around the engine uses Alien LUA module version 0.51. To be able to use the LUA SDK alien module that comes with LUA 5.1.4 must be updated. We suggest that you use LuaRocks for a simple alien update procedure.

Debugger module

Debugger module's functions are used for process debugging, disassembling, accessing context and manipulating memory.

Debugger module constants

Constants used by: **SetBPXEx** function, **SetHardwareBreakPoint** function, **SetHardwareBreakPointEx** 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_BREAKPOINT_TYPE_INT3 0x10000000
#define UE_BREAKPOINT_TYPE_LONG_INT3 0x20000000
#define UE_BREAKPOINT_TYPE_UD2 0x30000000

#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
#define UE_ENGINE_RESET_CUSTOM_HANDLER 7
#define UE_ENGINE_CALL_PLUGIN_DEBUG_CALLBACK 8
```

Constants used by: **HideDebugger** function

```
#define UE_HIDE_BASIC 1
```

Constants used by: **SetBPXOptions** function

```
#define UE_BREAKPOINT_INT3 1
#define UE_BREAKPOINT_LONG_INT3 2
#define UE_BREAKPOINT_UD2 3
```

StaticDisassembleEx function

The **StaticDisassembleEx** function is used to disassemble data from the context of the process using the 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 the address you are disassembling. If data was copied from a 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 the address where the instruction to be disassembled is located. Only the first instruction in that block will be disassembled.

Return value

If disassembly succeeds, *StaticDisassembleEx* returns a pointer to the disassembled instruction string, otherwise, it returns *NULL*.

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 the 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 the address where the instruction to be disassembled is located. Only the first instruction in that block will be disassembled. If you use *StaticDisassemble* to disassemble relative instructions, such as jumps, it will assume that instructions resides at DiassmAddress and will disassemble them as such. If you need to disassemble instructions that have been moved, use *StaticDisassembleEx*.

Return value

>   If disassembly succeeds, *StaticDisassemble* returns a pointer to the disassembled instruction string, otherwise, it returns *NULL*.

Remarks

>   diStorm64 is used for instruction disassembling.

Example

>   None.

DisassembleEx function

The function **DisassembleEx** disassembles data from the context of any running process, as long as you have access to the 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 address of the remote process containing the instruction to disassemble. Only the first instruction in that block will be disassembled.

ReturnInstructionType
> [in] Boolean switch specifying whether to return the disassembled string or the instruction type of that string. For example is the disassembled instruction is: MOV EAX,EBX and this switch is set to *TRUE DisassembleEx* returns only the string MOV. If this switch is set to *FALSE, DisassembleEx* returns the entire disassembled string.

Return value

If *DisassembleEx* succeeds, it returns a pointer to either the disassembled instruction string or the string's instruction type. If it fails, it returns *NULL*.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

Disassemble function

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

Syntax

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

Parameters

> *DisassmAddress*
>> [in] Pointer to address of the remote process containing the instruction to disassemble. Only the first instruction in that block will be disassembled.

Return value

If it succeeds, *Disassemble* returns a pointer to disassembled instruction string, otherwise it returns *NULL*.

Remarks

diStorm64 is used for instruction disassembling.

Example

None.

StaticLengthDisassemble function

The function **StaticLengthDisassemble** gets the length, in bytes, of a disassembled instruction from the context of a process, using the 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

> *StaticLengthDisassemble* returns the size of disassembled instruction string or *NULL* if disassemble fails.

Remarks

> diStorm64 is used for instruction disassembling.

Example

> None.

LengthDisassembleEx function

The function **LengthDisassembleEx** gets the length, in bytes, of a disassembled instruction from the context of any running process, as long as you have the access to the 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 the address of the instruction whose size you want to know. Only the first instruction in that block will be disassembled.

Return value

> If it succeeds, *LengthDisassembleEx* returns the size of disassembled instruction string. If it fails, it returns either *NULL* or minus one.

Remarks

> diStorm64 is used for instruction disassembling.

Example

> None.

LengthDisassemble function

The function **LengthDisassemble** gets the length, in bytes, of the disassembled instruction from the context of the process currently being debugged. This function fails if no process is being debugged or if the specified address doesn't exist inside the debugged process.

Syntax

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

Parameters

> *DisassmAddress*
>> [in] Pointer to the address of the instruction whose size you want to know. Only the first instruction in that block will be disassembled.

Return value

> If it succeeds, **LengthDisassemble** returns the size of the disassembled instruction string. If it fails, it returns either *NULL* or minus one.

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 the full path of the file to debug.
>
> *szCommandLine*
>> [in] Pointer to a command line parameter string that is passed to the created process.
>
> *szCurrentFolder*
>> [in] Pointer to string which will be passed to *CreateProcess* API during the process creation.

Return value

> *InitDebug* 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. It creates a process which will be debugged.

Syntax

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

Parameters

> *szFileName*
>> [in] Pointer to the full path of the file to debug.
>
> *szCommandLine*
>> [in] Pointer to a command line parameter string that is passed to the created process.
>
> *szCurrentFolder*
>> [in] Pointer to string which will be passed to *CreateProcess* API during the process creation.
>
> *EntryCallBack*
>> [in] Pointer to the callback function to call when the application reaches the file entry point. This is equal to setting breakpoint at the file entry point.

Return value

> If it succeeds, *InitDebugEx* returns a pointer to a PROCESS_INFORMATION structure, otherwise it returns *NULL*.

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. This function automatically loads any DLL loaders required to make DLL debugging work.

Syntax

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

Parameters

szFileName

[in] Pointer to the full path of the file to debug.

ReserveModuleBase

[in] Boolean variable that specifies whether or not to load a debugged DLL on a non-default image base.

szCommandLine

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

szCurrentFolder

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

EntryCallBack

[in] Pointer to a callback function to call when the application reaches the file entry point. This is equal to setting a breakpoint at the file entry point.

Return value

If it succeeds, *InitDLLDebug* returns a pointer to the `PROCESS_INFORMATION` structure, otherwise it returns *NULL*.

AutoDebugEx function

AutoDebugEx initializes the debugging process for both executables and DLLs. This function is always used first. It creates a process which will be debugged

Syntax

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

Parameters

> *szFileName*
>> [in] Pointer to the full path of the file to debug.
>
> *ReserveModuleBase*
>> [in] Boolean variable specifying whether or not to load debugged DLL on non default image base.
>
> *szCommandLine*
>> [in] Pointer to string which will be passed to created process as a 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] [in] Pointer to a callback function to call when the application reaches the file entry point. This is equal to setting a breakpoint at the file entry point.

Remarks

> Since debugging is performed in second thread, any calls to the *SendMessage Windows* API will cause the program to freeze. Keep this in mind while using this function.

SetErrorModel function

The **SetErrorModel** function is used to change the way that Windows loader behaves errors found while loading PE files. If you chose not to display error messages no warnings about broken or invalid files you try to debug will be shown to the user.

Syntax

```
void __stdcall SetErrorModel(
          bool DisplayErrorMessages
          );
```

Parameters

> *DisplayErrorMessages*
>> [in] Boolean switch which tells the *TitanEngine* should Windows loader errors be displayed or not.

Return value

> None.

Remarks

> None.

IsFileBeingDebugged function

The **IsFileBeingDebugged** function is used to determine if a file is currently being debugged or not.

Syntax

```
bool __stdcall IsFileBeingDebugged();
```

Parameters

>   None.

Return value

>   Boolean switch indicating whether or not a file is being debugged.

Remarks

>   None.

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 specifying whether or not to terminate the process to which the debugger was attached when debugging is complete.
>
> *DebugInfo*
>> [out] Pointer to the `PROCESS_INFORMATION` structure to fill when the process is attached.
>
> *CallBack*
>> [in] Pointer to the callback function to call when the first system breakpoint is reached.

Return value

> None.

Remarks

> None.

DetachDebugger function

Please use *DetachDebuggerEx* instead. The **DetachDebugger** function detaches the debugger from a debugged process. DetachDebugger detaches the debugger from a running process, allowing the process to continue running. All exceptions must be processed before detaching. Since exception processing can't be done from any *TitanEngine* callback, this function should NOT be used.

Syntax

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

Parameters

> *ProcessId*
>> [in] Process ID of the process from which you want to detach. The ID can be acquired from the *Windows* API or internal structures.

Return value

> Returns *TRUE* when detachment is complete, and *FALSE* if there is an error.

Remarks

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

Example

> None.

DetachDebuggerEx function

The **DetachDebuggerEx** function detaches the debugger from a debugged process, leaving the process running. *DetachDebuggerEx* ensures that all exceptions are processed before detaching.

Syntax

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

Parameters

> *ProcessId*
>> [in] Process ID of the process from which you want to detach. The ID can be acquired from the *Windows* API or internal structures.

Return value

> Returns *TRUE* when detachment is complete, and *FALSE* if there is an error.

Remarks

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

Example

> None.

GetProcessInformation function

The **GetProcessInformation** function retrieves a pointer to the PROCESS_INFORMATION structure that contains the initialization data for the debugged process.

Syntax

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

Parameters

None.

Return value

Returns a pointer to the `PROCESS_INFORMATION` structure, and therefore can be declared as `LPPROCESS_INFORMATION`.

Remarks

This data can only be retrieved from an active process.

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 a STARTUPINFOA structure, and can also be declared as LPSTARTUPINFOA.

Remarks

>   This data can only be retrieved from an active process.

Example

>   None.

GetDebuggedDLLBaseAddress function

GetDebuggedDLLBaseAddress retrieves the base address on which the debugged DLL file was loaded. This function is needed for DLL debugging, because a DLL may not be loaded on its default image base, for example, if the default base address is reserved for some other module or ASLR file.

Syntax

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

Parameters

None.

Return value

This function returns a pointer to the base address on which debugged DLL file was loaded. It can also be declared as void*.

Remarks

This data can only be retrieved from an active process whose DLL is being debugged.

Example

None.

ReversingLabs Corporation

GetDebuggedFileBaseAddress function

GetDebuggedFileBaseAddress retrieves the base address on which the debugged file is loaded. It retrieves the base address on which the debugged file was loaded. This function is needed for file debugging, because a file may not be loaded on its default image base, for example, if the default base address is reserved for some other module or ASLR file. Note, this function retrieves the base address of the main module, so if you are debugging a DLL file, it will return the base address of the DLL loader.

Syntax

```
long long __stdcall GetDebuggedFileBaseAddress();
```

Parameters

None.

Return value

This function returns a pointer the base address at which debugged file resides. It can also be declared as void*.

Remarks

Process must be active to retrieve this data.

Example

None.

ReversingLabs Corporation

GetExitCode function

GetExitCode retrieves a process exit code.

Syntax

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

Parameters

None.

Return value

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

Remarks

The process must be terminated to retrieve this data.

Example

None.

DebugLoop function

DebugLoop starts the debugging process. Always call this function or *DebugLoopEx* after initializing the debugger with one of the initialization functions. Set at least one breakpoint before running the debugged or the application whose process is being debugged will run. *DebugLoop* will not return until the debugger is finished and the 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 makes 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 to pass to the *Windows WaitForDebugEvent* API.

Return value

>   None.

Remarks

>   None.

Example

>   None.

SetDebugLoopTimeOut function

SetDebugLoopTimeOut sets a debug timeout at runtime. This value is passed to the *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

SetNextDbgContinueStatus sets the parameter passed to the *ContinueDebugEvent Windows* API.

Syntax

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

Parameters

> *SetDbgCode*
> > [in] Specify one of two values: DBG_CONTINUE or DBG_EXCEPTION_NOT_HANDLED.

Return value

> None.

Remarks

> USE WITH CAUTION: This function can break the debugging process. It may cause the debugged target to exit the debugging loop, and terminate the application, or it may show an error message on Windows Vista and later.

Example

> None.

StopDebug function

StopDebug stops the debugging process. The Debugged process is terminated just after the call to this function, and program control is passed to the first command following the *DebugLoop* function that started the debugging process.

Syntax

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

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ForceClose function

ForceClose stops the debugging process. The debugged process is terminated just after the call to this function and program control is passed to the first command following the *DebugLoop* function that started the debugging process. If your code has crashed with an exception, *ForceClose* is a safer way to terminate debugging than *StopDebug*.

Syntax

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

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

SetBPXOptions function

SetBPXOptions sets the global breakpoint type. Set type is global making all breakpoints set after calling this function to be set as the selected type.

Syntax

```
void __stdcall SetBPXOptions(
            long DefaultBreakPointType
            );
```

Parameters

> *DefaultBreakPointType*
> > [in] Can be one of the following:

- UE_BREAKPOINT_INT3 (0xCC)
- UE_BREAKPOINT_LONG_INT3 (0xCD03)
- UE_BREAKPOINT_UD2 (0x0F0B)

Return value

> None.

Remarks

> UE_BREAKPOINT_UD2 refers to two byte INT3 defined as 0xCD03.

Example

> None.

ReversingLabs Corporation

SetBPX function

SetBPX sets INT 3 breakpoints. Specify the type of INT3 breakpoint with the *SetBPXOptions* function.

Syntax

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

Parameters

> *bpxAddress*
>> [in] Address on which to set the breakpoint.
>
> *bpxType*
>> [in] Type of breakpoint to set, either `UE_BREAKPOINT` or `UE_SINGLESHOOT`. By adding one of the following values you can select the breakpoint type:
>>
>> - `UE_BREAKPOINT_TYPE_INT3 (0xCC)`
>> - `UE_BREAKPOINT_TYPE_LONG_INT3 (0xCD03)`
>> - `UE_BREAKPOINT_TYPE_UD2 (0x0F0B)`
>
> *bpxCallBack*
>> [in] Address of a callback to call when the breakpoint is hit.

CallBack definition

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

Return value

> Boolean switch indicating whether or not the breakpoint was set.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

SetBPXEx function

SetBPXEx sets INT3 breakpoints. Set the type of INT3 breakpoint with the *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 at which a breakpoint will be set.
>
> *bpxType*
>> [in] Type of the breakpoint to set. Either UE_BREAKPOINT or UE_SINGLESHOOT. By adding one of the following values you can select the breakpoint type:
>>
>> - UE_BREAKPOINT_TYPE_INT3 (0xCC)
>> - UE_BREAKPOINT_TYPE_LONG_INT3 (0xCD03)
>> - UE_BREAKPOINT_TYPE_UD2 (0x0F0B)
>
> *NumberOfExecutions*
>> [in] Set maximum number of breakpoint executions. Specify *NULL* for no execution limit.
>
> *CmpRegister*
>> [in] Register to check to determine whether the breakpoint execution condition is met.
>
> *CmpCondition*
>> [in] Type of comparison to run with the register. See *Debugger module constants* for details.
>
> *CmpValue*
>> [in] Value to use in the comparison. If condition has been met, the breakpoint callback will be executed. See *Debugger module constants* for details.

bpxCallBack

      [in] Address of a callback to call once breakpoint has been hit.

bpxCompareCallBack

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

bpxRemoveCallBack

      [in] The callback to call when the breakpoint is 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

EnableBPX enables a currently-disabled INT 3 breakpoint. 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 an existing breakpoint which you would like to re-enable after it has been disabled with *DisableBPX*.

Return value

> Boolean switch indicating whether or not the breakpoint was enabled.

Remarks

> None.

Example

> None.

DisableBPX function

DisableBPX disables a currently enabled or active INT 3 breakpoint. This function can't be used to disable memory or hardware breakpoints.

Syntax

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

Parameters

> *bpxAddress*
>
>> [in] Address of the existing INT 3 breakpoint to disable.

Return value

> Boolean switch indicating whether or not the breakpoint was disabled.

Remarks

> None.

Example

> None.

IsBPXEnabled function

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

Syntax

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

Parameters

bpxAddress

[in] Address of the existing INT3 breakpoint whose state you want to know.

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 the INT3 breakpoint to remove.

Return value

> Boolean switch indicating whether or not the breakpoint was removed.

Remarks

> If the breakpoint was set with *SetBPXEx,* and a remove callback was specified via *SetBPXEx*, the callback will be called once this function is called.

Example

> None.

SafeDeleteBPX function

This function has been deprecated. It has been preserved only for compatibility with earlier versions of *TitanEngine* SDK. **SafeDeleteBPX** is used to remove an existing INT3 breakpoint. This function can't be used to remove memory or hardware breakpoints.

Syntax

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

Parameters

> *bpxAddress*
> > [in] Address of the existing INT3 breakpoint to remove.

Return value

> Boolean switch indicating whether or not the breakpoint has been removed.

Remarks

> If the breakpoint was set with *SetBPXEx,* and a remove callback was specified via *SetBPXEx*, the callback will be called once this function is called.

Example

> None.

SetAPIBreakPoint function

The **SetAPIBreakPoint** sets an INT 3 breakpoint at an exported function in 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. Specify the name of the DLL containing the function on which to set the breakpoint, for example *kernel32.dll*.
>
>   *szAPIName*
>
>>   [in] Pointer to string. Specify the name of the API on which to set the breakpoint, for example *VirtualAlloc*.
>
>   *bpxType*
>
>>   [in] Type of breakpoint to set. Either UE_BREAKPOINT or UE_SINGLESHOOT.
>
>   *bpxPlace*
>
>>   [in] Where to set the breakpoint. Use UE_APISTART to specify the first instruction of the API or UE_APIEND to specify the last instruction, which is always RET or one of its variations.
>
>   *bpxCallBack*
>
>>   [in] Address of a callback to call once the breakpoint is hit.

CallBack definition

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

Return value

>   Boolean switch indicating whether or not the breakpoint was set.

Remarks

>   None.

ReversingLabs Corporation

DeleteAPIBreakPoint function

DeleteAPIBreakPoint removes an existing INT3 breakpoint from a function inside a DLL file.

Syntax

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

Parameters

> *szDLLName*
>> [in] Pointer to string. Specify the name of the DLL containing the function on which to set the breakpoint, for example ***kernel32.dll***.
>
> *szAPIName*
>> [in] Pointer to string. Specify the name of the API on which the breakpoint was set, for example ***VirtualAlloc***.
>
> *bpxPlace*
>> [in] Where on the API the breakpoint was set. Valid values: `UE_APISTART` if the breakpoint was set on the first instruction in the API, or `UE_APIEND` if the breakpoint was set on the last instruction in the API, which is always *RET* or a variation.

Return value

> Boolean switch indicating whether or not the breakpoint has been removed.

Remarks

> None.

Example

> None.

SafeDeleteAPIBreakPoint function

The **SafeDeleteAPIBreakPoint** function removes INT3 breakpoints from functions inside DLL files. This function has been preserved only for backward compatibility with earlier versions of *TitanEngine* SDK.

Syntax

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

Parameters

> *szDLLName*
>
>> [in] Pointer to string specifying the name of the DLL containing the function on which the breakpoint was set, for example **kernel32.dll**.
>
> *szAPIName*
>
>> [in] Pointer to string specitying the name of the API on which the breakpoint was set, for example **VirtualAlloc**.
>
> *bpxPlace*
>
>> [in] Location in the API at which the breakpoint was set. Use UE_APISTART if the breakpoint was set at the first instruction in the API or UE_APIEND if the breakpoint was set at the last instructionin the API, which is always *RET* or a variation.

Return value

> Boolean switch indicating whether or not the breakpoint has been removed.

Remarks

> This function has been replaced by *DeleteAPIBreakpoint*.

Example

> None.

SetMemoryBPX function

SetMemoryBPX sets memory breakpoints. These breakpoints are set by *PAGE_GUARD* to a targeted memory region. If your breakpoint size is smaller than one page, the whole page will be affected by the *PAGE_GUARD* and therefore it is possible that the page will be hit, but your breakpoint will not be handled. Keep this in mind when setting this kind of breakpoint, and always set your breakpoint to a whole page.

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 the callback to call once breakpoint is hit.

CallBack definition

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

Return value

> Boolean switch indicating whether or not the breakpoint was set.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

SetMemoryBPXEx function

Like *SetMemoryBPX*, the function **SetMemoryBPXEx** sets memory breakpoints, but *SetMemoryBPXEx* allows you to specify which type of memory access will trigger the break, and whether or not to restore the breakpoint once it has been hit.

These breakpoints are set by *PAGE_GUARD* to targeted memory region. If your breakpoint size is smaller than one page, the whole page will be affected by the *PAGE_GUARD* and therefore it is possible that the page will be hit, but your breakpoint will not be handled. Keep this in mind when setting this kind of breakpoint, and always set your breakpoint to a whole page.

Syntax

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

Parameters

MemoryStart

[in] Address on which to set the breakpoint. Ideally this is the start of a page.

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, UE_MEMORY_READ, or UE_MEMORY_WRITE. Where UE_MEMORY breaks on any type of access, UE_MEMORY_READ breaks only on read access, and UE_MEMORY_WRITE breaks only on write access.

RestoreOnHit

[in] Indicates whether or not to restore the breakpoint once it is executed. By default, a memory breakpoint is triggered only once, unless you use this option to restore it.

bpxCallBack

[in] Address of a callback to call when the breakpoint is hit.

CallBack definition

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

Return value

Boolean switch indicating whether or not the breakpoint was set.

ReversingLabs Corporation

RemoveMemoryBPX function

RemoveMemoryBPX removes previously-set memory breakpoints by removing *PAGE_GUARD* from page protection flags.

Syntax

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

Parameters

> *MemoryStart*
>> [in] Address on which breakpoint was set.
>
> *SizeOfMemory*
>> [in] Size of the memory that was specified when the breakpoint was set.

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 to set the breakpoint.
>
> *IndexOfRegister*
>
>> [in] Register that holds the bpxAddress. Specify a register in the range DR0 to DR3. If no register is specified, the first available free register will be used.
>
> *bpxType*
>
>> [in] Type of the breakpoint to 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 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 to call when the breakpoint is hit.

CallBack definition

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

Return value

> Boolean switch indicating whether or not the breakpoint was set.

Remarks

> None.

ReversingLabs Corporation

SetHardwareBreakPointEx function

The **SetHardwareBreakPointEx** function is used to set hardware breakpoints. These breakpoints can only be set on CPUs that support them. Function has the same callback definition as non expert version.

Syntax

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

Parameters

> *hActiveThread*
>> [in] Handle of the open thread from which context will be set.
>
> *bpxAddress*
>> [in] Address on which to set the breakpoint.
>
> *IndexOfRegister*
>> [in] Register that holds the bpxAddress. Specify a register in the range DR0 to DR3. If no register is specified, the first available free register will be used.
>
> *bpxType*
>> [in] Type of the breakpoint to 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 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.
>
> *IndexOfSelectedRegister*
>> [out] Pointer to DWORD variable which will receive the index of the register used to set the hardware breakpoint.

Return value

> Boolean switch indicating whether or not the breakpoint was set.

ReversingLabs Corporation

DeleteHardwareBreakPoint function

DeleteHardwareBreakPoint removes a previously-set hardware breakpoint.

Syntax

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

Parameters

IndexOfRegister
        [in] The register containing the breakpoint's address pointer (DR0 to DR3).

Return value

    Boolean switch indicating whether or not the breakpoint has been removed.

Remarks

    None.

Example

    None.

GetUnusedHardwareBreakPointRegister function

GetUnusedHardwareBreakPointRegister get the currently free DRx register.

Syntax

```
bool __stdcall GetUnusedHardwareBreakPointRegister(
            LPDWORD RegisterIndex
            );
```

Parameters

> *IndexOfRegister*
> > [out] Pointer to the variable which will receive the free DRx register (DR0 to DR3).

Return value

> Boolean switch indicating whether or not the any of the registers is free.

Remarks

> None.

Example

> None.

RemoveAllBreakPoints function

RemoveAllBreakPoints removes/disables all breakpoints of the specified type(s).

Syntax

```
bool __stdcall RemoveAllBreakPoints(
            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.

CurrentExceptionNumber function

CurrentExceptionNumber retrieves the number of exceptions that were registered and processed prior to this function being called.

Syntax

```
long __stdcall CurrentExceptionNumber();
```

Parameters

None.

Return value

Function returns the number of registered exceptions.

Remarks

None.

Example

None.

ClearExceptionNumber function

ClearExceptionNumber resets to zero the number of exceptions that were registered and processed prior to this function being called.

Syntax

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

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

GetDebugData function

GetDebugData retrieves a pointer to an internal `DEBUG_EVENT` structure.

Syntax

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

Parameters

None.

Return value

Pointer to `DEBUG_EVENT` structure. Can be declared as `LPDEBUG_EVENT` instead of void*.

Remarks

Process must be active for this function to work.

Example

None.

GetTerminationData function

GetTerminationData retrieves a pointer to an internal `DEBUG_EVENT` structure. *GetTerminationData* is similar to *GetDebugData* but is used only when the process has terminated its execution.

Syntax

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

Parameters

None.

Return value

Pointer to a `DEBUG_EVENT` structure. Can be declared as `LPDEBUG_EVENT` instead of void*.

Remarks

Function can only be used for terminated processes.

Example

None.

GetContextDataEx function

GetContextDataEx retrieves data from the context of any debugged process thread. You can use engine thread handling functions to get the handles of all active threads.

Syntax

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

Parameters

> *hActiveThread*
>> [in] Handle of the open thread from which to read the context.
>
> *IndexOfRegister*
>> [in] Indicator on which register will be read from the context of the selected thread. See *Debugger module constants* for details.

Return value

> This function returns the requested data.

Remarks

> None.

Example

> None.

GetContextData function

GetContextData retrieves context data from the thread that generated the last exception or timeout in the active process that the debugger is currently processing. Use *GetContextDataEx* if you want to get the context for a specific thread.

Syntax

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

Parameters

IndexOfRegister

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

Return value

This function returns the requested data.

Remarks

None.

Example

None.

ReversingLabs Corporation

SetContextDataEx function

SetContextDataEx sets the context data for the process thread that is being debugged.

Syntax

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

Parameters

> *hActiveThread*
>> [in] Handle of the open thread whose context will be set.
>
> *IndexOfRegister*
>> [in] Specifies the context register to modify. See *Debugger module constants* for details.
>
> *NewRegisterValue*
>> [in] The new context value to place in the register.

Return value

> Boolean switch indicating whether or not the value in the register was updated.

Remarks

> **CAUTION**: This function modifies context of the thread, so be careful using it, since corrupting certain registers can cause the application to crash.

Example

> None.

SetContextData function

SetContextData sets the context data for the active process thread that generated the most recent exception or debug timeout.

Syntax

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

Parameters

IndexOfRegister
> [in] Specifies the context register to modify. See *Debugger module constants* for details.

NewRegisterValue
> [in] The new context value to use.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

CAUTION: This function modifies context of the thread, so be careful using it, since corrupting certain registers can cause the application to crash.

Example

None.

GetContextFPUDataEx function

GetContextFPUDataEx retrieves FPU data from the context of any debugged process thread. You can use engine thread handling functions to get the handles of all active threads.

Syntax

```
bool __stdcall GetContextFPUDataEx(
        HANDLE hActiveThread,
        void* FPUSaveArea
        );
```

Parameters

> *hActiveThread*
>> [in] Handle of the open thread from which to read the context.
>
> *FPUSaveArea*
>> [out] Pointer to `FLOATING_SAVE_AREA` structure defined in WinNT.h for 32 bit systems
>>> or to `XMM_SAVE_AREA32` for 64 bit systems.

Return value

> This function returns *TRUE* on successful data retrieval or *FALSE* if data isn't available.

Remarks

> None.

Example

> None.

SetContextFPUDataEx function

SetContextFPUDataEx sets FPU data in the context of any debugged process thread. You can use engine thread handling functions to get the handles of all active threads.

Syntax

```
bool __stdcall SetContextFPUDataEx(
          HANDLE hActiveThread,
          void* FPUSaveArea
          );
```

Parameters

>   *hActiveThread*
>>   [in] Handle of the open thread from which to read the context.
>   *FPUSaveArea*
>>   [in] Pointer to FLOATING_SAVE_AREA structure defined in WinNT.h for 32 bit systems
>>   or to XMM_SAVE_AREA32 for 64 bit systems.

Return value

>   This function returns *TRUE* on successful data update or *FALSE* if update failed.

Remarks

>   None.

Example

>   None.

StepInto function

StepInto traces code by single-stepping into calls. This tracing function sets a trap flag and calls your callback once that trap flag has been activated. This allows you to execute one instruction at a time, making it possible to trace through the execution of each instruction.

Syntax

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

Parameters

StepCallBack
> [in] Address of the callback to call when trap flag has been hit.

CallBack definition

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

Return value

> None.

Remarks

> None.

Example

> None.

StepOver function

The **StepOver** traces code by single-stepping over calls. This tracing function sets an 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 thus no guarantee that your callback will ever be called. The breakpoint is run only once, and is removed once your callback has finished.

Syntax

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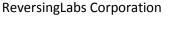
Parameters

> *StepCallBack*
>> [in] Address of the callback to call when the breakpoint has been hit.

CallBack definition

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

Return value

> None.

Remarks

> None.

Example

> None.

SingleStep function

The **SingleStep** traces code by single-stepping through a series of instructions. This tracing function sets a trap flag and calls your callback once that trap flag has been activated. This allows you to execute one instruction at a time, making it possible to trace through the execution of each instruction.

Syntax

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

Parameters

StepCount

[in] Number of instructions to trace. Your callback will be called each time an instruction executes, up to the number of instructions specified.

StepCallBack

[in] Address of the callback to call when each instruction executes.

CallBack definition

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

Return value

None.

Remarks

None.

Example

None.

FindEx function

FindEx searches 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 the memory location to use as the start point for the search.
>
> *MemorySize*
>> [in] Size of the memory region to search for the byte pattern.
>
> *SearchPattern*
>> [in] Pointer to the sequence of bytes to find.
>
> *PatternSize*
>> [in] Size of the search pattern, in bytes.
>
> *WildCard*
>> [in] Pointer to a 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 the first byte of the found pattern inside the remote process. It can also be declared as void*, or *NULL* if byte pattern is not found.

Remarks

> None.

ReversingLabs Corporation

Find function

Find searches 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 a 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.

MatchPatternEx function

MatchPatternEx tries to match the selected pattern to specified memory address.

Syntax

```
bool __stdcall MatchPatternEx(
        HANDLE hProcess,
        void* MemoryToCheck,
        int SizeOfMemoryToCheck,
        void* PatternToMatch,
        int SizeOfPatternToMatch,
        PBYTE WildCard
        );
```

Parameters

> *hProcess*
>
>> [in] Handle of the process whose memory will be searched.
>
> *MemoryToCheck*
>
>> [in] Pointer to the memory location to use as the start point for the pattern match.
>
> *SizeOfMemoryToCheck*
>
>> [in] Size of the memory region to check the pattern match.
>
> *PattternToMatch*
>
>> [in] Pointer to the sequence of bytes to match.
>
> *SizeOfPatternToMatch*
>
>> [in] Size of the match pattern, in bytes.
>
> *WildCard*
>
>> [in] Pointer to a 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 *TRUE* if the provided pattern matches the memory content or *FALSE* if it doesn't.

Remarks

None.

ReversingLabs Corporation

MatchPattern function

MatchPattern tries to match the selected pattern to specified memory address.

Syntax

```
bool __stdcall MatchPattern(
        void* MemoryToCheck,
        int SizeOfMemoryToCheck,
        void* PatternToMatch,
        int SizeOfPatternToMatch,
        PBYTE WildCard
        );
```

Parameters

> *MemoryToCheck*
>> [in] Pointer to the memory location to use as the start point for the pattern match.
>
> *SizeOfMemoryToCheck*
>> [in] Size of the memory region to check the pattern match.
>
> *PattternToMatch*
>> [in] Pointer to the sequence of bytes to match.
>
> *SizeOfPatternToMatch*
>> [in] Size of the match pattern, in bytes.
>
> *WildCard*
>> [in] Pointer to a 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 *TRUE* if the provided pattern matches the memory content or *FALSE* if it doesn't.

Remarks

None.

FillEx function

FillEx fills the specified process memory location with the specified byte. If the location size is larger than a byte, the byte is repeated until the location is full. Most commonly this is done to *NOP* parts of the code, or to zero out a memory region.

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

Fill fills the currently debugged process memory location with the specified byte. If the location size is larger than a byte, the byte is repeated until the location is full. Most commonly this is done to *NOP* parts of the code, or to zero out a memory region.

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.

ReversingLabs Corporation

PatchEx function

PatchEx fills a process memory location with the specified multi-byte pattern. If the region being filled is larger than the pattern, the remaining bytes may optionally be 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

Patch fills a memory location with the specified multi-byte pattern. If the region being filled is larger than the pattern, the remaining bytes may optionally be NOPed or left unmodified. This function always patches the process currently being debugged.

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

ReplaceEx does a search and replace on a specific byte pattern in a process memory location. When the byte pattern is found, it is replaced with the byte pattern you specify. Number of matches which will be replaced can be specified. Any remaining bytes may optionally be 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 a 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

Both search and replace patterns can have wild card bytes to ignore during search and replace.

Example

None.

Replace function

Replace does a search and replace on a specific byte pattern in a memory location. When the byte pattern is found, it is replaced with the byte pattern you specify. This function always replaces the byte pattern in the process currently being debugged.

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 a 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

Both search and replace patterns can have wild card bytes which will be ignored during search and replace.

Example

None.

GetRemoteString function

GetRemoteString retrieves a string from a 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.

ReversingLabs Corporation

GetFunctionParameter function

GetFunctionParameter retrieves the value of an input parameter from a standard function types. *GetFunctionParameter* can only work if the function's execution has been paused at either the first instruction of the call or the 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 either the requested value or a pointer to string, depending on the parameter type. If the parameter type is *PTR* then this function will return the data to which the pointer points. So if you use *PARAMETER_PTR_DWORD* it will return the *DWORD* to which that parameter points.

Remarks

Maximum length of the read string is 512 characters.
The Stack is acquired for the currently paused thread inside the debugged process.

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
```

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

GetFunctionParameter(hProcess, UE_FUNCTION_STDCALL , 2, UE_PARAMETER_DWORD);

Which returns 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);

Which would return the same value.

ReversingLabs Corporation

GetJumpDestinationEx function

GetJumpDestinationEx determines where the specified jump or call instruction will land.

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 of the jump or call whose destination you want to find.
>
> *JustJumps*
>> [in] Boolean switch that indicates whether or to get destinations for calls or only jumps.

Return value

> Returns the address targeted by jump/call or *NULL* if the instruction at the specified address isn't a jump or call.

Remarks

> None.

Example

> None.

GetJumpDestination function

GetJumpDestination determines where the specified jump or call instruction will land.

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

IsJumpGoingToExecuteEx determines whether or not the targeted jump is going to execute. *IsJumpGoingToExecuteEx* allows you to specify which process and which thread to check.

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.

ReversingLabs Corporation

IsJumpGoingToExecute function

IsJumpGoingToExecute check if the targeted jump is going to execute or not for the currently active thread in the process currently being debugged.

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

SetCustomHandler allows you to specify a custom exception handler for the specified exception. You can handle the most commonly generated errors via the built-in definitions, or you can handle all exceptions and filter only the 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

HideDebugger hides the debugger from a variety of detection tricks, reducing the chances that it will be detected.

Syntax

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

Parameters

> *hProcess*
>
>> [in] Handle of the debugged process.
>
> *PatchAPILevel*
>
>> [in] Patches following APIs when set to UE_HIDE_BASIC:
>>
>> - CheckRemoteDebuggerPresent
>> - GetTickCount

Return value

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

Remarks

> None.

Example

> None.

UnHideDebugger function

UnHideDebugger reverts hiding the debugger from a variety of detection tricks.

Syntax

```
bool __stdcall UnHideDebugger(
          HANDLE hProcess,
          DWORD PatchAPILevel
          );
```

Parameters

> *hProcess*
>> [in] Handle of the debugged process.
>
> *PatchAPILevel*
>> [in] Patches following APIs when set to UE_HIDE_BASIC:
>
>> - CheckRemoteDebuggerPresent
>> - GetTickCount

Return value

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

Remarks

> None.

Example

> None.

GetPEBLocation function

GetPEBLocation gets the PEB location inside the remote process.

Syntax

```
void* __stdcall GetPEBLocation(
          HANDLE hProcess
          );
```

Parameters

> *hProcess*
>> [in] Handle for process whose PEB location you are interested in.

Return value

> Returns a pointer to PEB inside remote process.

Remarks

> None.

Example

> None.

SetEngineVariable function

The **SetEngineVariable** sets 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
> > - UE_ENGINE_RESET_CUSTOM_HANDLER
> > - UE_ENGINE_CALL_PLUGIN_DEBUG_CALLBACK
>
> *VariableSet*
>
> > [in] Boolean value which will be set to the selected option.

Return value

> None.

Remarks

> None.

Example

> None.

Threader module

Functions in the threader module are designed for working with threads. Threader module functions cover thread identification; thread manipulation and remote thread injection.

Threader module structures

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

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

ThreaderGetThreadInfo function

ThreaderGetThreadInfo retrieves information about the threads inside the debugged process. Data is collected each time a new thread is created, or any of the existing ones terminates. You may specify either the thread handle or thread ID, but do not need to specify both.

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 a `THREAD_ITEM_DATA` structure or *NULL* if the thread is no longer active or not found.

Remarks

>   Only one of the two input parameters is needed.

Example

>   None.

ThreaderGetThreadData function

ThreaderGetThreadData retrieves a pointer to an array of `THREAD_ITEM_DATA` entries containing information about the existing threads. The *hThread* item in the last structure in the array is set to *NULL*. The number of items in the array is the number of existing threads inside the debugged process. The 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

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

Syntax

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

Parameters

EnumCallBack

[in] Pointer to the callback function that will process the thread item data for each thread.

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

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

Syntax

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

Parameters

> *hThread*
> > [in] Handle of the thread to pause.

Return value

> This function returns *TRUE* if thread is paused or *FALSE* if its execution can't be paused at this time.

Remarks

> None.

Example

> None.

ThreaderResumeThread function

ThreaderResumeThread 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 resumes or *FALSE* if its execution can't resume at this time.

Remarks

> None.

Example

> None.

ThreaderTerminateThread function

ThreaderTerminateThread tries to terminate an existing thread inside the debugged process.

Syntax

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

Parameters

>   *hThread*
>> [in] Handle of the thread to terminate.
>   *ThreadExitCode*
>> [in] Exit code to pass to the *TerminateThread Windows* API that terminates the thread.

Return value

>   This function returns *TRUE* if the thread is terminated or *FALSE* if thread cannot be terminated.

Remarks

>   None.

Example

>   None.

ThreaderPauseAllThreads function

ThreaderPauseAllThreads pauses 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.

Return value

This function returns *TRUE* if the specified threads are paused or *FALSE* if not.

Remarks

None.

Example

None.

ThreaderResumeAllThreads function

ThreaderResumeAllThreads resumes execution of all paused threads in 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.

Return value

> This function returns *TRUE* if all the existing threads are resumed or *FALSE* if not.

Remarks

> None.

Example

> None.

ThreaderPauseProcess function

ThreaderPauseProcess pauses all active threads inside the debugged process, suspending that process.

Syntax

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

Parameters

None.

Return value

This function returns *TRUE* if all threads are paused successfully or *FALSE* otherwise.

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

ThreaderIsThreadStillRunning checks whether the selected thread still exists, regardless of its state, in 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

ThreaderIsThreadActive checks whether 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

ThreaderIsAnyThreadActive checks whether any thread in the debugged process 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

ThreaderIsExceptionInMainThread determines whether 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, *FALSE* if it occurred in another running thread.

Remarks

None.

Example

None.

ThreaderGetOpenHandleForThread function

ThreaderGetOpenHandleForThread is used resolve the existing open handle for thread with the specified ID.

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 the handle of the specified thread or NULL if the thread doesn't exist anymore.

Remarks

> None.

Example

> None.

ThreaderSetCallBackForNextExitThreadEvent function

ThreaderSetCallBackForNextExitThreadEvent specifies a custom callback to call the next time an active thread terminates.

Syntax

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

Parameters

> *exitThreadCallBack*
>> [in] Pointer to callback function to call 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

ThreaderCreateRemoteThreadEx creates 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 to create the thread.
>
>*ThreadStartAddress*
>>[in] Start address at which to create the thread in the remote process.
>
>*AutoCloseTheHandle*
>>[in] Boolean switch indicating whether or not to close the handle to this new remote thread automatically.
>
>*ThreadPassParameter*
>>[in] Parameter which to pass to newly created thread.
>
>*ThreadId*
>>[in] Pointer to DWORD to hold 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.

ReversingLabs Corporation

ThreaderCreateRemoteThread function

ThreaderCreateRemoteThread creates a new thread inside the process currently being debugged.

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 to this new remote thread automatically.
>
> *ThreadPassParameter*
>> [in] Parameter which to pass to newly created thread.
>
> *ThreadId*
>> [in] Pointer to DWORD to hold 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.

ReversingLabs Corporation

ThreaderInjectAndExecuteCodeEx function

ThreaderInjectAndExecuteCodeEx creates a new thread inside the targeted process and auto-executes 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 the data to inject in the remote process.
>
> *StartDelta*
>> [in] Used when you need to to execute the code from any other point other then the first byte of the *InjectedCode* memory, the start address of the new thread will be increased by this value.
>
> *InjectedSize*
>> [in] Size of the memory the new thread will occupy in 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

ThreaderInjectAndExecuteCode creates a new thread inside the currently debugged process and auto-executes the injected code.

Syntax

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

Parameters

> *InjectedCode*
>> [in] Pointer to the data to inject into 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** 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.

ThreaderImportRunningThreadData function

ThreaderImportRunningThreadData collects data about running threads for the specified process. This function can be used to get data about remote process threads and manipulate them. However this function overwrites internal data and should only be used if no program is being currently debugged.

Syntax

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

Parameters

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

Return value

> This function returns *TRUE* if the threads have been imported and *FALSE* if there were problems.

Remarks

> This action removes ALL data about the threads and should only be used if no program is debugged.

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** sets 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 the callback to call when 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** sets a breakpoint on all TLS callbacks inside the PE header.

Syntax

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

Parameters

>   *szFileName*
>>   [in] Pointer to the full path of the file to debug.
>   *bpxCallBack*
>>   [in] Address of the callback to call when 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 retrieve the TLS callbacks from the PE header.

Syntax

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

Parameters

>   *szFileName*
>>   [in] Pointer to a null terminated string which is the full path to the file whose TLS callback data will be read and copied to the specified 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** remove TLS callbacks from the PE header of the selected file.

Syntax

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

Parameters

szFileName

[in] Pointer to the full path of the 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** removes TLS table from the PE header of the selected file.

Syntax

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

Parameters

szFileName

[in] Pointer to the full path of the 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** 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 the full path of the 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** 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** 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** 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 specifying the full path to the file in which to write the new TLS table.
>
> *szSectionName*
>> [in] The new TLS table will be written to a new PE section in the file specified by *szFileName*. This variable specifies the name to use for the new section. The section name may be up to 8 characters long.
>
> *ArrayOfCallBacks*
>> [in] Pointer to array of custom TLS callback functions.
>
> *NumberOfCallBacks*
>> [in] Number of callbacks in the specified array.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

Librarian module

The Librarian module contains functions designed to manipulate 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

LibrarianSetBreakPoint sets a breakpoint on a specific type of library event, 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 to watch for specified events. For example **kernel32.dll**
>
> *bpxType*
>> [in] Specifies the type of 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 whether the breakpoint will be executed only once, or each time the specified event occurs.
>
> *bpxCallBack*
>> [in] Address of the callback to call when the breakpoint is triggered.

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 is 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

LibrarianGetLibraryInfo retrieves library item data for 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 a pointer to a LIBRARY_ITEM_DATA structure or *NULL* if selected DLL cannot be found.

Example

>   None.

LibrarianGetLibraryInfoEx function

LibrarianGetLibraryInfoEx retrieves additional library item data for the modules loaded by the debugged process.

Syntax

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

Parameters

> *BaseOfDll*
>> [in] Base address at 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

LibrarianEnumLibraryInfo enumerates data for all DLL files loaded by the debugged process. This list contains data about only currently loaded modules. Unloaded modules are automatically removed from the list.

Syntax

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

Parameters

> *EnumCallBack*
> > [in] Address of the callback function to use for processing 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.

ReversingLabs Corporation

Hooks module

The Hooks module has functions designed for in process function hooking. For hooking to be possible entire engine or just *TitaniumHooks* must be loaded in the context of the process in which the hooks are being applied.

Hooks module constants

Constants used by: **HooksInsertNewRedirection** function

```
#define TEE_HOOK_NRM_JUMP 1
#define TEE_HOOK_NRM_CALL 3

#define TEE_HOOK_IAT 5
#define TEE_MAXIMUM_HOOK_RELOCS 7
```

Hooks module structures

Structure used by: **HooksGetHookEntryDetails** function

```
typedef struct HOOK_ENTRY{
        bool IATHook;
        BYTE HookType;
        DWORD HookSize;
        void* HookAddress;
        void* RedirectionAddress;
        BYTE HookBytes[TEE_MAXIMUM_HOOK_SIZE];
        BYTE OriginalBytes[TEE_MAXIMUM_HOOK_SIZE];
        void* IATHookModuleBase;
        DWORD IATHookNameHash;
        bool HookIsEnabled;
        bool HookIsRemote;
        void* PatchedEntry;
        DWORD RelocationInfo[TEE_MAXIMUM_HOOK_RELOCS];
        int RelocationCount;
}HOOK_ENTRY, *PHOOK_ENTRY;
```

HooksSafeTransition function

HooksSafeTransition puts the running process in suspended state leaving only the thread that inserts new hooks running. Once all the hooks are inserted paused process can be resumed by calling the same function again. **WARNING**: Using this function to resume paused threads will cause all threads to be resumed not only the ones paused by the previous use of the same function.

Syntax

```
bool __stdcall HooksSafeTransition(
        LPVOID HookAddress,
        bool TransitionStart
        );
```

Parameters

>   *HookAddress*
>>   [in] Single address which will be hooked inside this safe transition block. If there is more than one hook to insert use the expert version of *HookSafeTransition* function.
>   *TransitionStart*
>>   [in] Since the same function can be used to both pause and resume the process execution this boolean switch indicates which of the two needs to be performed.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

HooksSafeTransitionEx function

HooksSafeTransitionEx puts the running process in suspended state leaving only the thread that inserts new hooks running. Once all the hooks are inserted paused process can be resumed by calling the same function again. **WARNING**: Using this function to resume paused threads will cause all threads to be resumed not only the ones paused by the previous use of the same function.

Syntax

```
bool __stdcall HooksSafeTransitionEx(
        LPVOID HookAddressArray,
        int NumberOfHooks,
        bool TransitionStart
        );
```

Parameters

> *HookAddressArray*
>> [in] Array of addresses which will be hooked inside this safe transition block.
>
> *NumberOfHooks*
>> [in] Number of items in the provided array.
>
> *TransitionStart*
>> [in] Since the same function can be used to both pause and resume the process execution this boolean switch indicates which of the two needs to be performed.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

HooksDisableRedirection function

HooksDisableRedirection disables the selected hook. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksDisableRedirection(
          LPVOID HookAddress,
          bool DisableAll
          );
```

Parameters

> *HookAddress*
>> [in] Hooked address whose hook will be disabled.
>
> *DisableAll*
>> [in] Boolean switch indicating whether or not to disable all installed hooks.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

HooksDisableRedirectionsForModule function

HooksDisableRedirectionsForModule disables all installed hooks inside the selected module. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksDisableRedirectionsForModule(
          HMODULE ModuleBase
          );
```

Parameters

> *ModuleBase*
> > [in] Base address of the loaded library whose hooks need disabling.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> Here module refers to loaded library which is hooked.

Example

> None.

HooksDisableIATRedirection function

HooksDisableIATRedirection disables all installed hooks inside the selected module's import address table. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksDisableIATRedirection(
          char* szModuleName,
          char* szHookFunction,
          bool DisableAll
          );
```

Parameters

>   *szModuleName*
>> [in] Name of the loaded module, for example: **kernel32.dll**
>   *szHookFunction*
>> [in] Name of the hooked function, for example: **VirtualProtect**
>   *DisableAll*
>> [in] Boolean switch indicating whether or not to disable all installed import address hooks for the selected module.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

ReversingLabs Corporation

HooksEnableRedirection function

HooksEnableRedirection enables the selected hook. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksEnableRedirection(
          LPVOID HookAddress,
          bool EnableAll
          );
```

Parameters

>   *HookAddress*
>> [in] Hooked address whose hook will be disabled.
>   *EnableAll*
>> [in] Boolean switch indicating whether or not to enable all disabled hooks.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

HooksEnableRedirectionsForModule function

HooksEnableRedirectionsForModule enables all disabled hooks inside the selected module. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksEnableRedirectionsForModule(
            HMODULE ModuleBase
            );
```

Parameters

> *ModuleBase*
> > [in] Base address of the loaded library whose hooks need enabling.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> Here module refers to loaded library which is hooked.

Example

> None.

HooksEnableIATRedirection function

HooksEnableIATRedirection enables all disabled hooks inside the selected module's import address table. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksEnableIATRedirection(
        char* szModuleName,
        char* szHookFunction,
        bool EnableAll
        );
```

Parameters

> *szModuleName*
>> [in] Name of the loaded module, for example: **kernel32.dll**
>
> *szHookFunction*
>> [in] Name of the hooked function, for example: **VirtualProtect**
>
> *EnableAll*
>> [in] Boolean switch indicating whether or not to enable all disabled import address hooks for the selected module.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

HooksRemoveRedirection function

HooksRemoveRedirection removes the selected hook. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksRemoveRedirection(
          LPVOID HookAddress,
          bool RemoveAll
          );
```

Parameters

> *HookAddress*
>> [in] Hooked address whose hook will be disabled.
>
> *RemoveAll*
>> [in] Boolean switch indicating whether or not to remove all installed hooks.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

HooksRemoveRedirectionsForModule function

HooksRemoveRedirectionsForModule removes all installed hooks inside the selected module. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksRemoveRedirectionsForModule(
          HMODULE ModuleBase
          );
```

Parameters

>   *ModuleBase*
>   [in] Base address of the loaded library whose hooks will be removed.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   Here module refers to loaded library which is hooked.

Example

>   None.

HooksRemoveIATRedirection function

HooksEnableIATRedirection removes all installed hooks inside the selected module's import address table. Original bytes are restored in the process and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode.

Syntax

```
bool __stdcall HooksRemoveIATRedirection(
          char* szModuleName,
          char* szHookFunction,
          bool RemoveAll
          );
```

Parameters

>   *szModuleName*
>>   [in] Name of the loaded module, for example: **kernel32.dll**
>   *szHookFunction*
>>   [in] Name of the hooked function, for example: **VirtualProtect**
>   *RemoveAll*
>>   [in] Boolean switch indicating whether or not to remove all import address hooks for the selected module.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

ReversingLabs Corporation

HooksInsertNewRedirection function

HooksInsertNewRedirection installs a new hook on the selected address. Memory is changed in the process of installing hooks and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode. You can only have one hook on the selected address and therefore trying to hook the same address twice will result into this function returning false indicating that the hook wasn't installed.

Syntax

```
bool __stdcall HooksInsertNewRedirection(
        LPVOID HookAddress,
        LPVOID RedirectTo,
        int HookType
        ;
```

Parameters

> *HookAddress*
>> [in] Address which will be redirected by a hook.
>
> *RedirectTo*
>> [in] Installed hook will redirect code execution to this address.
>
> *HookType*
>> [in] Indicates which type of hook to use. Can be one of the following:
>>> `TEE_HOOK_NRM_JUMP` or `TEE_HOOK_NRM_CALL.`

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

HooksInsertNewIATRedirectionEx function

HooksInsertNewIATRedirectionEx installs a new import address hook. Memory is changed in the process of installing hooks and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode. You can only have one hook on the selected address and therefore trying to hook the same address twice will result into this function returning false indicating that the hook wasn't installed.

Syntax

```
bool __stdcall HooksInsertNewIATRedirectionEx(
          ULONG_PTR FileMapVA,
          ULONG_PTR LoadedModuleBase,
          char* szHookFunction,
          LPVOID RedirectTo
          );
```

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.
>
> *LoadedModuleBase*
>> [in] Base address on which the module whose IAT is being patched is loaded on.
>
> *szHookFunction*
>> [in] Name of the hooked function, for example: **VirtualProtect**
>
> *RedirectTo*
>> [in] Installed hook will redirect code execution to this address.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

HooksInsertNewIATRedirection function

HooksInsertNewIATRedirection installs a new import address hook. Memory is changed in the process of installing hooks and therefore using this function in multithreaded environment is recommended only after transitioning to safe patching mode. You can only have one hook on the selected address and therefore trying to hook the same address twice will result into this function returning false indicating that the hook wasn't installed.

Syntax

```
bool __stdcall HooksInsertNewIATRedirection(
            char* szModuleName,
            char* szHookFunction,
            LPVOID RedirectTo
            );
```

Parameters

> *szModuleName*
>> [in] Name of the loaded module, for example: **kernel32.dll**
>
> *szHookFunction*
>> [in] Name of the hooked function, for example: **VirtualProtect**
>
> *RedirectTo*
>> [in] Installed hook will redirect code execution to this address.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

HooksGetTrampolineAddress function

HooksGetTrampolineAddress retrieves the trampoline address which can be called if the hooked function needs to execute the original function and not just replace its functionality. This function is used for both API and IAT hooks.

Syntax

```
void* __stdcall HooksGetTrampolineAddress(
        LPVOID HookAddress
        );
```

Parameters

HookAddress

[in] Address which was hooked for which the function will return the appropriate trampoline.

Return value

Function returns trampoline address or *NULL* is there is no hook at the selected address.

Remarks

None.

Example

None.

HooksGetHookEntryDetails function

HooksGetHookEntryDetails retrieves the information about installed hooks. This function is used for all hooks. Data returned can be manipulated to affect the hook state.

Syntax

```
void* __stdcall HooksGetHookEntryDetails(
        LPVOID HookAddress
        );
```

Parameters

HookAddress

[in] Address which was hooked for which the function will return the hook info.

Return value

Function returns the hook info or *NULL* is there is no hook at the selected address.

Remarks

None.

Example

None.

HooksScanModuleMemory function

HooksScanModuleMemory function scans the selected module memory for installed hooks which are registered to internal hooking system. All found hooks are registered regardless of which component of the program installed them. Therefore this can be used to uninstall existing hooks not inserted by the Hooks module.

Syntax

```
void __stdcall HooksScanModuleMemory(
        HMODULE ModuleBase,
        LPVOID CallBack
        );
```

Parameters

> *ModuleBase*
>> [in] Base address of the loaded library whose memory will be scanned for hooks.
>
> *CallBack*
>> [in] Pointer to the callback function to call for each found hook.

CallBack definition

```
typedef bool(__stdcall *fEnumCallBack)(PHOOK_ENTRY HookDetails, \
void* ptrOriginalInstructions, PLIBRARY_ITEM_DATA ModuleInformation, \
DWORD SizeOfImage);
```

Return value

> None.

Remarks

> None.

Example

> None.

HooksScanEntireProcessMemory function

HooksScanEntireProcessMemory function scans the entire process memory for installed hooks which are registered to internal hooking system. All found hooks are registered regardless of which component of the program installed them. Therefore this can be used to uninstall existing hooks not inserted by the Hooks module.

Syntax

```
void __stdcall HooksScanEntireProcessMemory(
        LPVOID CallBack
        );
```

Parameters

> *CallBack*
>> [in] Pointer to the callback function to call for each found hook.

CallBack definition

```
typedef bool(__stdcall *fEnumCallBack)(PHOOK_ENTRY HookDetails, \
void* ptrOriginalInstructions, PLIBRARY_ITEM_DATA ModuleInformation, \
DWORD SizeOfImage);
```

Return value

> None.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

HooksScanEntireProcessMemoryEx function

HooksScanEntireProcessMemoryEx function scans the entire process memory for installed hooks which are registered to internal hooking system. All found hooks are registered regardless of which component of the program installed them. Therefore this can be used to uninstall existing hooks not inserted by the Hooks module.

Syntax

```
void __stdcall HooksScanEntireProcessMemoryEx();
```

Parameters

> None.

Return value

> None.

Remarks

> Found hooks are registered without any notification.

Example

> None.

ReversingLabs Corporation

OEP Finder module

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

FindOEPInit function

FindOEPInit initializes the OEP tracing process. It is not necessary to call it directly, since it will be automatically called by the engine itself.

Syntax

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

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

FindOEPGenerically function

FindOEPGenerically finds the packed file's 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**: It is possible for the file to execute when this function is called, so use this function with caution!

Syntax

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

Parameters

>   *szFileName*
>> [in] Pointer to the full path of the file whose entry point you want to find.
>   *TraceInitCallBack*
>> [in] Callback to call once the packed file hits its packed entry point.
>   *CallBack*
>> [in] Callback to call once the 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.

ReversingLabs Corporation

Process module

The Process module includes functions that enumerate processes and execute basic operations inside the context of a remote process.

GetActiveProcessId function

GetActiveProcessId finds an active process using its image name.

Syntax

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

Parameters

szImageName
> [in] The image name of the running process. For example **explorer.exe**

Return value

This function returns process ID if the process is running and found, or 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 one found.

Example

None.

EnumProcessesWithLibrary function

EnumProcessesWithLibrary enumerates all processes that have loaded the specified DLL image.

Syntax

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

Parameters

szLibraryName

[in] Name of the library in which you are interested. For example **kernel32.dll**

EnumFunction

[in] Address of a callback function that will process the data.

CallBack definition

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

Return value

None.

Remarks

None.

Example

None.

RemoteLoadLibrary function

RemoteLoadLibrary makes a 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 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 load inside remote process. For example **advapi32.dll**
>
> *WaitForThreadExit*
>> [in] Boolean switch indicating whether or not to wait for the 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.

ReversingLabs Corporation

RemoteFreeLibrary function

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

Syntax

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

Parameters

> *hProcess*
>> [in] Handle of the process in which to create the new thread that will unload the DLL.
>
> *hModule*
>> [in] Base address at which the DLL file is loaded in remote process.
>
> *szLibraryName*
>> [in] Name of the library to unload from the remote process. For example **advapi32.dll**
>
> *WaitForThreadExit*
>> [in] Boolean switch indicating whether or not to wait for the remote thread to terminate before returning from this function call.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> You only need to specify the name or the base address of the module, not both.

Example

> None.

RemoteExitProcess function

RemoteExitProcess injects a remote thread in the selected process to terminate the process by calling *ExitProcess*.

Syntax

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

Parameters

> *hProcess*
>> [in] Handle of the process to terminate.
>
> *ExitCode*
>> [in] Exit code that will be passed to the *ExitProcess* API. Can be NULL.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

TranslateNativeName function

TranslateNativeName resolves the full path to a file using its native name. *TranslateNativeName* is used to resolve paths that contain physical devices in their file references.

Syntax

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

Parameters

>   *szNativeName*
>   > [in] The native name of the file whose path you want to resolve.

Return value

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

Remarks

>   **CAUTION:** A 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 the 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

DumpProcess creates, for the currently running process, a memory dump in a file on disk. This image is not a valid PE file, but a state of memory at the time the function is called.

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 active file image from which to dump the memory. This value should be read from the file on disk.
>
> *szDumpFileName*
>> [in] Pointer to the full path of the file in which to write the memory content.
>
> *EntryPoint*
>> [in] Virtual address which will be set to the new file's entry point. The 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.

ReversingLabs Corporation

DumpProcessEx function

DumpProcessEx, for the specified running process, a memory dump in a file on disk. This image is not a valid PE file, but a state of memory at the time the function is called.

Syntax

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

Parameters

> *ProcessId*
>> [in] Process ID of the process, which can be acquired with the *Windows* API.
>
> *ImageBase*
>> [in] Default image base of the active file image from which to dump the memory. This value should be read from the file on disk.
>
> *szDumpFileName*
>> [in] Pointer to the full path of the file in which to write the memory content.
>
> *EntryPoint*
>> [in] Virtual address which will be set to the new file's entry point. The 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

DumpMemory creates memory dump to a file on disk which for a specified part of memory from 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 the memory range to dump. This does not need to be the start of a page.
> *MemorySize*
>> [in] The size of memory to copy 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 the full path of the file in which to write 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

DumpMemoryEx dumps to a file on disk the specified part of memory from the specified running process.

Syntax

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

Parameters

> *ProcessId*
>> [in] Process ID of the process whose memory you want to dump, which can be acquired with the *Windows* API.
>
> *MemoryStart*
>> [in] Start of the memory range dump to disk. This start does not have to be the start of a page.
>
> *MemorySize*
>> [in] Specifies the size of the memory to copy to disk. The 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 the full path of the file in which to write 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

DumpRegions creates a memory dump for all used memory regions in the specified running process. The dump is written to a series of files in the specified folder on disk. Optionally this function can dump only those regions located above the image base of the folder.

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 the full path of the folder in which to write the memory content. Each region will be dumped into a separate file within this folder.
>
> *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

DumpRegionsEx creates a memory dump for all used memory regions in the specified running process. The dump is written to a series of files in the specified folder on disk. Optionally this function can dump only those regions located above the image base of the folder.

Syntax

```
bool __stdcall DumpRegionsEx(
        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 the full path of the folder in which to write the memory content. Each region will be dumped into a separate file within this folder.
>
>   *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

DumpModule dumps to a file on disk the memory of one module within the specified running process. This image is not a valid PE file, but the state of memory at the time this function is called.

Syntax

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

Parameters

> *hProcess*
>> [in] Handle of the running process in which the module is running.
>
> *ModuleBase*
>> [in] The base address of the loaded module in the specified process.
>
> *szDumpFileName*
>> [in] Pointer to the full path of the file in which to write the module's memory content.

Return value

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

Remarks

> None.

Example

> None.

DumpModuleEx function

DumpModuleEx dumps to a file on disk the memory of one module within the specified running process. This image is not a valid PE file, but the state of memory at the time this function is called.

Syntax

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

Parameters

> *ProcessId*
>> [in] Process ID of the running process in which the module is loaded - can be acquired with the *Windows* API.
>
> *ModuleBase*
>> [in] The base address of the loaded module in the specified process.
>
> *szDumpFileName*
>> [in] Pointer to the full path of the file in which to write the module's memory content.

Return value

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

Remarks

> None.

Example

> None.

PastePEHeader function

PastePEHeader loads the PE header from a file on disk and writes it to running process memory. This can be used to fix damage to PE header during process runtime. Such damage only occurs 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 repaired.
>
> *ImageBase*
>> [in] Default image base of the targeted PE file. This value should be read from the file on disk.
>
> *szDebuggedFileName*
>> [in] Pointer to the full path of the file from which the PE header will be read.

Return value

> This function returns *TRUE* if the PE header is fixed and *FALSE* otherwise.

Remarks

> None.

Example

> None.

ExtractSection function

ExtractSection copies the physical content of the specified section in a file to a new file on disk.

Syntax

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

Parameters

>   *szFileName*
>> [in] Pointer to the full path of the file from which the section will be extracted.
>   *szDumpFileName*
>> [in] Pointer to the full path of the file to which the section will be written.
>   *SectionNumber*
>> [in] Number of the section to extract. Section numbers range from *zero* to *section count minus one*.

Return value

>   This function returns *TRUE* if the extraction succeeds, and *FALSE* if the extraction fails.

Remarks

>   None.

Example

>   None.

ResortFileSections function

ResortFileSections sorts a file's physical sections, putting them in the order of ascending physical offset. This can be useful if there you need to add data to, or expand; the last logical section of the file, but it isn't physically located in the last physical section of the file.

Syntax

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

Parameters

> *szFileName*
> > [in] Pointer to the full path of the file to re-sort.

Return value

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

Remarks

> The file's size doesn't change, but its hash does, because sections will be physically moved to new positions.

Example

> None.

FindOverlay function

FindOverlay finds overlay data (extra data appended to the end of a PE file). This data can be the file certificate or other important data that is useful for further file analysis.

Syntax

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

Parameters

> *szFileName*
>> [in] Pointer to the full path of the file from which to extract any overlay data.
> *OverlayStart*
>> [out] Pointer to a DWORD which will hold a file pointer that points to the overlay data. This file pointer can be used with the *Windows* API to access the overlay.
> *OverlaySize*
>> [out] Pointer to a DWORD to hold the size of the overlay.

Return value

> This function returns *TRUE* if an overlay is found, *FALSE* if the overlay is not found.

Remarks

> None.

Example

> None.

ExtractOverlay function

ExtractOverlay finds overlay data (extra data appended to the end of a PE file) and copies it to new file. This data can be the file certificate or other important data that is useful 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 the full path of the file to hold the overlay content extracted from the input file.

Return value

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

Remarks

> The output file is always overwritten.

Example

> None.

AddOverlay function

AddOverlay appends extracted overlay data to the end of PE files. This data can be the file certificate or other important data that is useful for further file analysis.

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 the full path of the PE file to which the overlay content will be appended (null terminated string).

Return value

> This function returns *TRUE* if data is 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

CopyOverlay copies overlay data from one PE file to another. If target file already has overlay data, the new data will be appended just after the existing data.

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* if the data is successfully appended and *FALSE* if one of the files is not found or not a PE file.

Remarks

None.

Example

None.

RemoveOverlay function

RemoveOverlay removes overlay data from PE files.

Syntax

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

Parameters

> *szFileName*
>
> > [in] Pointer to the full path of the file which will be stripped of its overlay data.

Return value

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

Remarks

> None.

Example

> None.

MakeAllSectionsRWE function

MakeAllSectionsRWE sets the characteristics of all sections in a PE file to read/write/executable.

Syntax

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

Parameters

szFileName

[in] Pointer to the full path of the file whose PE sections will be set to read/write/executable.

Return value

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

Remarks

None.

Example

None.

AddNewSectionEx function

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

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 be up to 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 the newly created section. If null, the new section will be filled with zeroes.
>
> *ContentSize*
>> [in] Size of the memory whose content which will be copied to the new section.

Return value

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

AddNewSection function

AddNewSection adds a new physical PE section, filled with zeroes, to a file, creating space in the file that can be used to store data later.

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 to use as the new section name. This string can be up to 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 the newly created section, or NULL if adding the new section fails.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

ResizeLastSection function

ResizeLastSection increases the size of the last PE file section in a file. The section is increased both physically and virtually. Optionally, the new section's size can be 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] Set to *TRUE to align the* last section's size to FileAlignment.

Return value

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

Remarks

> The file is backed up before modification, and restored if the file cannot be resized.

Example

> None.

SetSharedOverlay function

SetSharedOverlay is used only to store string pointer provided to it. This function is from the old SDK and is retained only for backward compatibility.

Syntax

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

Parameters

szFileName

[in] Pointer to the full path of a file. This string pointer will be stored in case other modules need to retrieve it but have no direct access to the variable. The 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

GetSharedOverlay is retrieves a store string pointer provided by *SetSharedOverlay* function. This function is from the old SDK and is retained only for backward compatibility

Syntax

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

Parameters

None.

Return value

This function returns the previously stored pointer.

Remarks

None.

Example

None.

DeleteLastSection function

DeleteLastSection physically deletes the last PE section from the specified file.

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 the file doesn't exist or the PE header is broken.

Remarks

The file is backed up before modification and restored if the file cannot be resized.

Example

None.

ReversingLabs Corporation

DeleteLastSectionEx function

DeleteLastSectionEx physically deletes the selected number PE sections from the end of the specified 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 sections to remove from the end of the PE file.

Return value

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

Remarks

> The file is backed up before modification and restored if the file cannot be resized.

Example

> None.

GetPE32DataFromMappedFile function

GetPE32DataFromMappedFile retrieves 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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 first PE section from which to read the data. The first PE section is section *zero*, so the section numbers range from *zero* to *section count minus one*.
>
>   *WhichData*
>
>>   [in] Specifies which PE header info this function will return. The list of constants used by this function is located at the beginning of this section under *Dumper module constants*.

Return value

>   This function returns the requested PE data. The return variable should be defined as ULONG_PTR which defines its size on x86 and x64 operating system.

Remarks

>   The 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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 to hold all PE header data. This structure is different for x86 and x64 systems. 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 the file doesn't exist or the PE header is broken.

Remarks

>   The file must be mapped before using this function.

Example

>   None.

GetPE32Data function

GetPE32Data retrieves 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. The first PE section is section *zero*, so the section numbers range from *zero* to *section count minus one*.
>
> *WhichData*
>> [in] Specifies which PE header info this function will return. The list of constants used by this function is located at the beginning of this section under *Dumper module constants*.

Return value

> This function returns the PE header data. The 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 to hold the PE header data. This structure is different for x86 and x64. 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 the file doesn't exist or the 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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. The first PE section is section *zero*, so the section numbers range from *zero* to *section count minus one*.

WhichData

[in] Indicator on which PE header info this function will return. The list of constants 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 the file doesn't exist or the PE header is broken.

Remarks

The file must be mapped before using this function.

SetPE32DataForMappedFileEx function

SetPE32DataForMappedFileEx stores the data from a PE header into a data structure, for both x86 and x64 files.

Syntax

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

Parameters

FileMapVA

[in] Pointer to the mapped file content you want to store. This pointer is set by using either *StaticFileLoad* function or *Windows* API for file mapping. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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 that will hold the PE header data. Ideally this structure is first filled by using *GetPE32DataFromMappedFileEx*. This structure is different for x86 and x64, 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 the file doesn't exist or the PE header is broken.

Remarks

The file must be mapped before using this function.

Example

None.

ReversingLabs Corporation

SetPE32Data function

SetPE32Data sets 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. The first PE section is section *zero*, so the section numbers range from *zero* to *section count minus one*.
>
> *WhichData*
>> [in] Indicator on which PE header info this function will return. Specifies which PE header info this function will return. The list of constants used by this function is located at the beginning of this section *Dumper module constants*.

Return value

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

Remarks

> None.

Example

> None.

SetPE32DataEx function

SetPE32DataEx sets 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 from which the PE header data is reset. 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 the file doesn't exist or the PE header is broken.

Remarks

None.

Example

None.

GetPE32SectionNumberFromVA function

GetPE32SectionNumberFromVA determines in which PE section the selected virtual address resides.

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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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. The first PE section is section zero, so the section numbers range from zero to section count minus one.

Remarks

The file must be mapped before using this function.

Example

None.

ConvertVAtoFileOffset function

ConvertVAtoFileOffset converts virtual addresses to its 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems.
>
> *AddressToConvert*
>> [in] Virtual address to convert to a physical address.
>
> *ReturnType*
>> [in] Boolean variable which indicates whether or not to add the *FileMapVA* to the function's return value.

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

The file must be mapped before using this function.

Example

None.

ConvertVAtoFileOffsetEx function

ConvertVAtoFileOffsetEx converts virtual or relative virtual addresses to its physical counterpart. Using this function is considered safer then *ConvertVAtoFileOffset* because it uses safety checks to ensure that the 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems.

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 to converted to its physical address.

AddressIsRVA

[in] Boolean variable which indicates whether the input address is relative or virtual. Virtual 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.

ReversingLabs Corporation

ConvertFileOffsetToVA function

ConvertFileOffsetToVA converts a physical address to its 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems.
>
> *AddressToConvert*
>> [in] The physical address to convert to a virtual address. It must reside in address space allocated with *FileMapVA*.
>
> *ReturnType*
>> [in] Boolean variable which indicates whether or not to add *ImageBase* to the function's 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

The file must be mapped before using this function.

Example

None.

ConvertFileOffsetToVAEx function

ConvertFileOffsetToVAEx converts physical addresses to its virtual counterpart. Using this function is considered safer then *ConvertFileOffsetToVA* because it does safety checks to ensure that the 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems.

    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 the *ReturnType* is *FALSE* this value will never be larger than DWORD. If this function returns *NULL* conversion has failed.

Remarks

The file must be mapped before using this function.

Importer module

The 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

ImporterInit initializes the importer module. It must be used before using any of the functions that do 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

ImporterSetImageBase updates information passed to the engine when the importer is initialized.

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

ImporterAddNewDll adds new DLLs to the new import tree. This function creates a new DLL entry making all subsequent *ImporterAddNewAPI* function calls add APIs to the current DLL. If you want to add APIs that don't belong to the current DLL, add a new DLL entry first. PECOFF specifications imply that trunks are in a plus four (or eight on x64) sequence. The 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 the name of the DLL to add to the new import tree. For example: **kernel32.dll** or **SomeFolder\mydll.dll** in case of relative path loading.

> *FirstThunk*
>> [in] Optional: address inside the PE file's memory that holds the pointer to an API belonging to that DLL. This is the first pointer in the sequence. If set to *NULL* the next call to *ImporterAddNewAPI* will set the first trunk to the data provided to it. If the trunk is outside the PE file's memory you must use a special approach described in examples.

Return value

> None.

Remarks

> None.

Example

> None.

ImporterAddNewAPI function

ImporterAddNewAPI adds a new API to the current import tree. This function creates a new API entry under currently selected DLL added by *ImporterAddNewDLL*. If the APIs don't belong to the current DLL, add a new DLL entry first. PECOFF specifications imply that trunks are in a plus four (or eight on x64) sequence. The 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 the name of the API to be added to the new import tree, but belonging to current DLL. For example: **VirtualProtect** or **VirtualAlloc**, which are added to the current DLL, which is in this case **kernel32.dll**
>
>   *ThunkValue*
>
>>   [in] Mandatory parameter, specifying the address inside the PE file's memory that holds the pointer to the API belonging to that DLL. In case the trunk is outside the PE file's memory you must use a special approach described in examples.

Return value

>   None.

Remarks

>   None.

Example

>   None.

ReversingLabs Corporation

ImporterAddNewOrdinalAPI function

ImporterAddNewOrdinalAPI adds a new ordinal API to the current import tree. This function creates a new API entry under currently selected DLL added by *ImporterAddNewOrdinalAPI*. If the APIs don't belong to the current DLL, add a new DLL entry first. PECOFF specifications imply that trunks are in a plus four (or eight on x64) sequence. The 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 ImporterAddNewOrdinalAPI(
            ULONG_PTR OrdinalNumber,
            ULONG_PTR ThunkValue
            );
```

Parameters

> *OrdinalNumber*
>> [in] API ordinal number to be added. Can have a `IMAGE_ORDINAL_FLAG` mask.
>
> *ThunkValue*
>> [in] Mandatory parameter, specifying the address inside the PE file's memory that holds the pointer to the API belonging to that DLL. In case the trunk is outside the PE file's memory you must use a special approach described in examples.

Return value

> None.

Remarks

> None.

Example

> None.

ImporterGetLastAddedDLLName function

ImporterGetLastAddedDLLName retrieves the name of the last added DLL in the current importer tree.

Syntax

```
void* __stdcall ImporterGetLastAddedDLLName();
```

Parameters

> None.

Return value

> Pointer to string holding the last added DLL name.

Remarks

> **CAUTION:** A string with the DLL name is stored inside the engine which makes this function multi thread unsafe.

Example

> None.

ImporterGetAddedDllCount function

ImporterGetAddedDllCount 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

ImporterGetAddedAPICount 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

ImporterEnumAddedData enumerates all added import tree data and calls the designated callback for each one, with the details about it.

Syntax

```
void __stdcall ImporterEnumAddedData(
            LPVOID EnumCallBack
            );
```

Parameters

> *EnumCallBack*
>> [in] Address of a callback function that will process the 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

ImporterEstimatedSize 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

ImporterCleanup clears all added DLLs and APIs from the import tree. This resets the inputted data to its 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

ImporterExportIATEx exports the added import data to existing PE file creating the valid import table for the selected PE file. After this function has executed, the import data will be cleared by using the *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*.

   szSectionName

   [in] Name of the PE section in which the new import table content will be written. 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

ImporterExportIAT exports the added import data to the existing PE file creating the valid import table for the selected PE file. After this function has executed, the 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 it can also be an unused part of the file that is 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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.

ReversingLabs Corporation

ImporterGetDLLName function

ImporterGetDLLName returns the name of the DLL in which address supplied to the function resides in. This address is usually the pointer to API located inside the import table of the file whose IAT you are fixing.

Syntax

```
void* __stdcall ImporterGetDLLName(
            ULONG_PTR APIAddress
            );
```

Parameters

>   *APIAddress*
>
>   >   [in] Address on which possible API is located. This address is equal to the address returned by the *GetProcAddress Windows* API for DLL and the function of interest.

Return value

>   Pointer to string, which is the name of the DLL which holds the API located at specified address or *NULL* if no DLL is found.

Remarks

>   String with the DLL name is stored inside the engine which makes this function multi thread unsafe. Process which is searched for the DLL is always the currently debugged process.

Example

>   ImporterGetDLLName(GetProcAddress(GetModuleHandleA("kernel32.dll"), "VirtualAlloc"));

>   This will return a pointer to "kernel32.dll" 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.

ImporterGetAPIName function

ImporterGetAPIName 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 the address returned by the *GetProcAddress Windows* API for DLL and the function of interest.

Return value

>   Pointer to string, which is the name of the API located at specified address or *NULL* if no API is found.

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

ImporterGetAPINameEx 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 or *NULL* if no API is found.

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.

ImporterGetAPIOrdinalNumber function

ImporterGetAPIOrdinalNumber returns the ordinal number 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

```
long long __stdcall ImporterGetAPIOrdinalNumber(
            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

Ordinal number of the API located at supplied address or *minus one* if no API is found.

Remarks

None.

Example

None.

ImporterGetRemoteAPIAddress function

ImporterGetRemoteAPIAddress realigns 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

ImporterGetRemoteAPIAddressEx retrieves 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

ImporterGetLocalAPIAddress 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

ImporterGetDLLNameFromDebugee gets 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 or *NULL* if no API is found.

Remarks

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

Example

> None.

ImporterGetAPINameFromDebugee function

ImporterGetAPINameFromDebugee resolves 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 or *NULL* if no API is found.

Remarks

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

Example

None.

ImporterGetAPIOrdinalNumberFromDebugee function

ImporterGetAPIOrdinalNumberFromDebugee resolves the API ordinal number 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 ImporterGetAPIOrdinalNumberFromDebugee(
        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

>   Ordinal number of the API that is located at supplied address or *minus one* if no API is found.

Remarks

>   None.

Example

>   None.

ImporterGetDLLIndexEx function

ImporterGetDLLIndexEx 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

ImporterGetDLLIndex 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

ImporterGetRemoteDLLBase gets 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

ImporterGetRemoteDLLBaseEx gets 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

ImporterIsForwardedAPI checks 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

ImporterGetForwardedAPIName retrieves 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 or *NULL* if no API is found.

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**

ImporterGetForwardedAPIOrdinalNumber function

ImporterGetForwardedAPIOrdinalNumber retrieves the ordinal number of the forwarded API.

Syntax

```
long long __stdcall ImporterGetForwardedAPIOrdinalNumber(
            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

> Ordinal number name of the API forwarded for supplied address or *minus one* if no API is found.

Remarks

> None.

Example

> None.

ImporterGetForwardedDLLName function

ImporterGetForwardedDLLName retrieves 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 or *NULL* if no DLL is found.

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

ImporterGetForwardedDLLIndex retrieves 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.

ReversingLabs Corporation

ImporterFindAPIWriteLocation function

ImporterFindAPIWriteLocation searches 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.

ReversingLabs Corporation

ImporterFindOrdinalAPIWriteLocation function

ImporterFindOrdinalAPIWriteLocation searches 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 ImporterFindOrdinalAPIWriteLocation(
          ULONG_PTR OrdinalNumber
          );
```

Parameters

> *OrdinalNumber*
>> [in] Ordinal number for which the write address will be located. This was the ordinal number passed to *ImporterAddNewAPI* or *ImporterAddNewDLL* at the time of adding that API.

Return value

> Function returns the address on which the selected ordinal API is written or *NULL* is that ordinal number wasn't added to import tree.

Remarks

> Function assumes that ordinal numbers are unique.

Example

> None.

ImporterFindAPIByWriteLocation function

ImporterFindAPIByWriteLocation searches 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

ImporterFindDLLByWriteLocation searches 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

ImporterGetNearestAPIAddress estimates 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

ImporterGetNearestAPIName gets the name of the API closest to the specified API. This is useful if, by tracing, you get to the address of the target API from is inside the current API, but you don't know how many instructions the target API executes prior to the start of the one you are in.

Syntax

```
void* __stdcall ImporterGetNearestAPIName(
        HANDLE hProcess,
        ULONG_PTR APIAddress
        );
```

Parameters

> *hProcess*
>> [in] Handle of the process in which the target API is located.
>
> *APIAddress*
>> [in] Address near the possible start of the target API the specified process.

Return value

> Pointer to string which is the name of the nearest API to the specified address or *NULL* if no API is found.

Remarks

> **CAUTION:** The string containing the API name is stored inside the engine, which makes this function multi thread unsafe.

Example

> None.

ImporterMoveIAT function

ImporterMoveIAT turns on a switch to make the importer export the import table in a way that ensures strings are written after the import tree, which is important when fixing import eliminations if we don't know where APIs will be written. If that is the case, all APIs 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. Since default strings are written to the section first, *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 the automatic import fixing functions, so examples 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

ImporterRelocateWriteLocation relocates all import data by the same value. It is used only in the situation describes for the *ImporterMoveIAT* function. 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] Offset to add 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 to use as a temporary import tree entry offset, when using the 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.

Syntax

```
long long __stdcall ImporterGetCurrentDelta();
```

Parameters

None.

Return value

This function returns the current delta which will be used for writing new virtual trunk.

Remarks

None.

Example

None.

ImporterLoadImportTable function

ImporterLoadImportTable loads an import table from any PE file. The loaded table will be converted to an internal engine import tree, making it available for further modifications before it's 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

ImporterCopyOriginalIAT copies IAT from one file to another. This function assumes that the IAT will be in the same virtual location in both files, so it is only used in cases when you dynamically unpack crypters and where the format doesn't handle imports by itself. Instead it leaves the import table handling to *Windows* loader, as 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.

ReversingLabs Corporation

ImporterMoveOriginalIAT function

ImporterMoveOriginalIAT moves IAT from one file to another. This function doesn't actually modify the original file, but loads the import table and exports it to selected dump file.

Syntax

```
bool __stdcall ImporterMoveOriginalIAT(
          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

ImporterAutoSearchIAT automatically locates a 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

ImporterAutoSearchIATEx automatically locates 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

ImporterAutoFixIATEx automatically fixes 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 that indicates whether the file was dumped or not.
>
> *RealignFile*
>
>> [in] Boolean switch that indicates whether or not the file needs realigning after fixing imports.
>
> *EntryPointAddress*
>
>> [in] [in] Virtual address which will be set to the new file. Size of this variable varies, on x86 its 4 bytes and on x64 is 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 - 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*.

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 that 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 that 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 the callback to call for every possible but unknown import redirection or elimination. Use this callback to correct import table fixing when a particular 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

ImporterAutoFixIAT automatically fixes the import table for the running process. This function can fix all known redirections and import eliminations, using an 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. It will be 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 used to iterate the search position. If you specify NULL, the default is used: four on x86, or eight on x64.

Return value

See *ImporterAutoFixIATEx* function for return details.

Tracer module

The 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

TracerInit is not used in *TitanEngine* and is retained only for backward compatibility issued with version 1.5.

Syntax

```
void __stdcall TracerInit();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

TracerLevel1 function

TracerLevel1 traces the provided address via code disassembling, in order to try to find the API hiding behind an import redirection. This function uses common code tracing to try to identify the API that 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 the memory that holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

Return value

   If the return value is *NULL* the trace has failed, and if the return value is greater than 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, which can then be used by the *HashTracerLevel1* API.

Remarks

   None.

Example

   None.

ReversingLabs Corporation

HashTracerLevel1 function

HashTracerLevel1 traces 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 that 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 the memory that 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 the return value is *NULL* or minus one, the trace has failed; if the return value is greater than *NULL* then the return is an API address inside the remote process.

Remarks

>   None.

Example

>   None.

TracerDetectRedirection function

TracerDetectRedirection checks whether 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 the memory that holds the redirection code. This memory is commonly
> > > outside the PE file memory and is allocated by the packer/protector code.

Return value

> If the 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

TracerFixKnownRedirection fixes 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 the memory that 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 the return value is *NULL,* the trace has failed; if the return value is greater than *NULL* then the return is an API address inside the remote process.

Remarks

> None.

Example

> None.

TracerFixRedirectionViaImpRecPlugin function

TracerFixRedirectionViaImpRecPlugin fixes import redirections with ImpRec modules. To use this option you must install all needed ImpRec plugins by placing them in the **.\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 the memory that holds the redirection code. This memory is commonly outside the PE file memory and is allocated by the packer/protector code.

Return value

> If the return value is *NULL* the trace has failed and if the return value is greater than *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

RealignPE realigns 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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.

ReversingLabs Corporation

RealignPEEx function

RealignPEEx realigns 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 the specified size. This option should only be used to **increase** the file size.

   ForcedFileAlignment

   [in] Specify *FileAlignment* manually. If no value is specified, the default of 0x200 will be used.

Return value

   This function returns the new realigned file size or *NULL* if it fails.

Remarks

   None.

Example

   None.

ReversingLabs Corporation

FixHeaderCheckSum function

FixHeaderCheckSum recalculates 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

WipeSection removes the section from the file but preserves 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

IsFileDLL determines 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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

IsPE32FileValidEx determines 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. Can be one of the following:
>   `UE_RESULT_FILE_INVALID_FORMAT`, `UE_RESULT_FILE_INVALID_BUT_FIXABLE`,
>   `UE_RESULT_FILE_INVALID_AND_NON_FIXABLE` or `UE_RESULT_FILE_OK`.

Example

>   None.

FixBrokenPE32FileEx function

FixBrokenPE32FileEx tries 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 that was filled by the *IsPE32FileValidEx* function, or *NULL,* which would make the engine call a file validation function before fixing the file automatically.

FileFixInfo

[in & out] Pointer to a FILE_FIX_INFO structure to fill during file fixing. 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 the file repair process which can occur in broken files. It returns *FALSE* if there are errors during fixing.

Remarks

The *OveralEvaluation* member of the structure tells the state of the file after fixing.

Example

None.

ReversingLabs Corporation

Relocater module

The relocater module has functions designed for relocation manipulation, making file memory snapshots, and removing relocation tables. This module is used to fix a relocation table when unpacking DLL files.

RelocaterInit function

RelocaterInit initializes the relocater module. It must be used before using any of the relocator functions to do 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 at which the targeted DLL file, whose relocation table you are fixing, is loaded.

Return value

> None.

Remarks

> None.

Example

> None.

RelocaterCleanup function

RelocaterCleanup frees memory allocated by the engine during relocation fixing.

Syntax

```
void __stdcall RelocaterCleanup();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

RelocaterAddNewRelocation function

RelocaterAddNewRelocation adds an address from the remote process to the list of addresses tnat need relocating, if the file is allocated at a base address other then default one. Just like when adding import via the importer, you must add relocations one page at a time. The 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 pages.

Syntax

```
void __stdcall RelocaterAddNewRelocation(
          HANDLE hProcess,
          ULONG_PTR RelocateAddress,
          DWORD RelocateState
          );
```

Parameters

> *hProcess*
>> [in] Handle of the process that has loaded the targeted DLL, whose relocation table you are fixing.
>
> *RelocateAddress*
>> [in] Address inside the remote process, belonging to targeted DLL, which needs relocation if the file is being loaded at a 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

RelocaterEstimatedSize estimates the space needed 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

RelocaterExportRelocationEx exports the added relocation data to an existing PE file, creating a valid relocation table for the selected PE file. After this function has executed, relocation data will be cleared by using the *RelocaterCleanup* function.

Syntax

```
bool __stdcall RelocaterExportRelocationEx(
            char* szFileName,
            char* szSectionName
            );
```

Parameters

>   *szFileName*
>
>>   [in] Pointer to string which is the full path to the file in which the new relocation table will be written. This file is usually created by the *DumpProcess* function.
>
>   *szSectionName*
>
>>   [in] Name (up to 8 characters long) of the PE section in which the new relocation table content will be written. This section will be added to the file.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

RelocaterExportRelocation function

RelocaterExportRelocation exports the added relocation data to an existing PE file, creating a 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 at which the new relocation table will be written. Usually this is a new section but, it can also be part of the file which is unused, but still in read/write mode.
>
> *StorePlaceRVA*
>
> > [in] Relative virtual address inside the PE file at which the new relocation table will be written. This input is just a 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 either the *StaticFileLoad* function or *Windows* API for file mapping. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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.

ReversingLabs Corporation

RelocaterGrabRelocationTableEx function

RelocaterGrabRelocationTableEx copies a PECOFF valid relocation table from the targeted process to engine relocation data storage. This function can automatically determine the size of 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, the 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 the remote process: 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.

ReversingLabs Corporation

RelocaterGrabRelocationTable function

RelocaterGrabRelocationTable copies a PECOFF valid relocation table from the targeted process to engine relocation data storage. If this function succeeds the 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 the remote process: 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

RelocaterMakeSnapshot copies the selected memory segment to a file on the disk. This memory segment should contain all data that can and will be relocated by the packer itself. Most commonly, it covers all the executable code inside the PE files memory. By creating two snapshots, one just before the code that 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 comparing the two.

Syntax

```
bool __stdcall RelocaterMakeSnapshot(
        HANDLE hProcess,
        char* szSaveFileName,
        LPVOID MemoryStart,
        ULONG_PTR MemorySize
        );
```

Parameters

> *hProcess*
>> [in] Handle of the process whose memory will be snapshotted.
>
> *szFileName*
>> [in] Pointer to the full path of the file to create and fill with the snapshot.
>
> *MemoryStart*
>> [in] Pointer to memory in the remote process: the start of the snapshot image. The start of the snapshot must be the same for both snapshots.
>
> *MemorySize*
>> [in] Size of the snapshot, which must be the same for both snapshots.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

RelocaterCompareTwoSnapshots function

RelocaterCompareTwoSnapshots creates a valid relocation table by comparing two memory snapshots: one taken just before the relocation code inside the packer was executed and another taken right afterward. If this function succeeds, the 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 at 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: used as the start for a snapshot image.
>
> *szDumpFile1*
>> [in] Pointer to the full path of the file which was created for the first memory snapshot.
>
> *szDumpFile2*
>> [in] Pointer to the full path of the file which for the second memory snapshot.

Return value

> Boolean switch indicating whether or not the function was successful.

RelocaterWipeRelocationTable function

RelocaterWipeRelocationTable removes the relocation table from any PE file. However it is only recommended that you remove the relocation table from executable files if you need 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.

RelocaterRelocateMemoryBlock function

RelocaterRelocateMemoryBlock uses mapped file relocation table to relocate selected memory segment to new loaded base. Data about which parts of the memory need relocating is read directly from the relocation table and therefore presence of such table is a necessity.

Syntax

```
bool __stdcall RelocaterRelocateMemoryBlock(
          ULONG_PTR FileMapVA,
          ULONG_PTR MemoryLocation,
          void* RelocateMemory,
          DWORD RelocateMemorySize,
          ULONG_PTR CurrentLoadedBase,
          ULONG_PTR RelocateBase
          );
```

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.
>
> *MemoryLocation*
>> [in] Virtual address of the memory segment to be relocated.
>
> *RelocateMemory*
>> [in] Pointer to memory segment which will be relocated.
>
> *RelocateMemorySize*
>> [in] Size of the memory segment to relocate.
>
> *CurrentLoadedBase*
>> [in] Current base on which the memory segment to be relocated is loaded.
>
> *RelocateBase*
>> [in] New base to which the memory segment will be relocated.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

ReversingLabs Corporation

Exporter module

The exporter module has functions designed for export manipulation and building new export tables.

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 to use as a default library name in the export table. For example: **mylib.dll**

Return value

> None.

Remarks

> None.

Example

> None.

ExporterSetImageBase function

ExporterSetImageBase updates information passed to the engine when exporter is initialized.

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

ExporterCleanup clears all added exports. This resets the inputted data to its original state. Before using the functions to add the export data, you must initialize the exporter again.

Syntax

```
void __stdcall ExporterCleanup();
```

Parameters

None.

Return value

None.

Remarks

None.

Example

None.

ExporterAddNewExport function

ExporterAddNewExport adds new function to the export table.

Syntax

```
void __stdcall ExporterAddNewExport(
          char* szExportName,
          DWORD ExportRelativeAddress
          );
```

Parameters

szExportName

[in] Name of the function that 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 at which the function you want to export is located.

Return value

None.

Remarks

None.

Example

None.

ExporterAddNewOrdinalExport function

ExporterAddNewOrdinalExport adds a 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 specified ordinal number.
>
> *ExportRelativeAddress*
>> [in] Relative virtual address for the exported function. The location in the PE file at which the function you want to export is located.

Return value

> None.

Remarks

> None.

Example

> None.

ExporterGetAddedExportCount function

ExporterGetAddedExportCount gets the current number of added functions in the export table.

Syntax

```
long __stdcall ExporterGetAddedExportCount();
```

Parameters

None.

Return value

Returns the number of added items in the export table.

Remarks

None.

Example

None.

ExporterEstimatedSize function

ExporterEstimatedSize estimates the size of memory needed to write the export data. This value can be used to determine the size of the new section to 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

ExporterBuildExportTableEx exports the added export data to an existing PE file, creating a valid export table for the selected PE file. After this function executes, 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 in which new export table will be written.

szSectionName

[in] Name (up to 8 characters long) of the PE section in which the new import table content will be written. This section will be added to the file.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ExporterBuildExportTable function

ExporterBuildExportTable exports the added export data to existing PE file, creating a valid export table for the selected PE file. After this function executes, the export data will be cleared by using *ExporterCleanup* function.

Syntax

```
bool __stdcall ExporterBuildExportTable(
        ULONG_PTR StorePlace,
        ULONG_PTR FileMapVA
        );
```

Parameters

StorePlace

[in] Physical address inside PE file to which the new export table will be written. Usually this is a new section, but it can also be a 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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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

ExporterLoadExportTable loads an export table from any PE file. The loaded table will be converted to an internal engine export tree, making it available for further modifications before it's 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

The 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

ResourcerLoadFileForResourceUse simulates 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 it simulates only *Windows* PE loader behavior concerning PE file loading and storing sections in memory. 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* if the file isn't a PE file or it couldn't be loaded.

Remarks

> None.

Example

> None.

ResourcerFreeLoadedFile function

ResourcerFreeLoadedFile unloads a file previously loaded by the *ResourcerLoadFileForResourceUse* function. Due to the nature of the *TitanEngine* PE file loader simulator, using this function is equal to using *VirtualFree* on the selected memory range.

Syntax

```
bool __stdcall ResourcerFreeLoadedFile(
            LPVOID LoadedFileBase
            );
```

Parameters

LoadedFileBase

[in] Base address at which the file is loaded. This is usually a return value from the *ResourcerLoadFileForResourceUse* function.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ResourcerExtractResourceFromFileEx function

ResourcerExtractResourceFromFileEx goes through the resource tree of the loaded file and extracts the specified resource to disk.

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. It is a ULONG_PTR which defines its size on the x86 and x64 operating systems. 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 the 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 the full path of the file to create and fill with the specified resource memory content.

Return value

Boolean switch indicating whether or not the function was successful.

ResourcerExtractResourceFromFile function

ResourcerExtractResourceFromFile goes through the resource tree of the loaded file and extracts the specified resource to disk.

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.

ResourcerFindResource function

ResourcerFindResource goes through the resource tree of the selected file and finds the specified resource.

Syntax

```
bool __stdcall ResourcerFindResource(
            char* szFileName,
            char* szResourceType,
            DWORD ResourceType,
            char* szResourceName,
            DWORD ResourceName,
            DWORD ResourceLanguage,
            PULONG_PTR pResourceData,
            LPDWORD pResourceSize
            );
```

Parameters

szFileName

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

szResourceType

[in] Pointer to string which is a resource type identifier. If the resource type is an integer use the integer input variable to locate such resource.

ResourceType

[in] Pointer to string which is a resource type identifier. If the resource type is an integer use this variable to pass the value and set its string version to *NULL*.

szResourceName

[in] Pointer to string which is a resource name identifier. If the resource type is an integer use the integer input variable to locate such resource.

ResourceName

[in] Pointer to string which is a resource name identifier. If the resource type is an integer use this variable to pass the value and set its string version to *NULL*.

ResourceLanguage

[in] Integer value that indicates the language of the resource to use.

pResourceData

[out] Pointer to a *ULONG_PTR* variable which will receive the offset on which the resource is located on.

pResourceSize

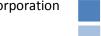[in] Pointer to a *DWORD* variable which will receive the size of located resource.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ResourcerFindResourceEx function

ResourcerFindResourceEx goes through the resource tree of the selected file and finds the specified resource.

Syntax

```
bool __stdcall ResourcerFindResourceEx(
          ULONG_PTR FileMapVA,
          DWORD FileSize,
          char* szResourceType,
          DWORD ResourceType,
          char* szResourceName,
          DWORD ResourceName,
          DWORD ResourceLanguage,
          PULONG_PTR pResourceData,
          LPDWORD pResourceSize
          );
```

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.
>
> *FileSize*
>> [in] Size of the mapped file.
>
> *szResourceType*
>> [in] Pointer to string which is a resource type identifier. If the resource type is an integer use the integer input variable to locate such resource.
>
> *ResourceType*
>> [in] Pointer to string which is a resource type identifier. If the resource type is an integer use this variable to pass the value and set its string version to *NULL*.
>
> *szResourceName*
>> [in] Pointer to string which is a resource name identifier. If the resource type is an integer use the integer input variable to locate such resource.
>
> *ResourceName*
>> [in] Pointer to string which is a resource name identifier. If the resource type is an integer use this variable to pass the value and set its string version to *NULL*.
>
> *ResourceLanguage*
>> [in] Integer value that indicates the language of the resource to use.

pResourceData

[out] Pointer to a *ULONG_PTR* variable which will receive the offset on which the resource is located on.

pResourceSize

[in] Pointer to a *DWORD* variable which will receive the size of located resource.

Return value

Boolean switch indicating whether or not the function was successful.

Remarks

None.

Example

None.

ResourcerEnumerateResource function

ResourcerEnumerateResource goes through the resource tree of the loaded file and enumerates all found resources.

Syntax

```
void __stdcall ResourcerEnumerateResource(
        char* szFileName,
        void* CallBack
        );
```

Parameters

> *szFileName*
>> [in] Pointer to a null terminated string which is a full path to file whose resource will be enumerated.
>
> *CallBack*
>> [in] Address of a callback to call for each found resource.

CallBack definition

```
typedef bool(__stdcall *fResourceEnumerator)(wchar_t* szResourceType, \
    DWORD ResourceType, wchar_t* szResourceName, DWORD ResourceName, \
    DWORD ResourceLanguage, DWORD ResourceData, DWORD ResourceSize);
```

Return value

> None.

Remarks

> None.

Example

> None.

ResourcerEnumerateResourceEx function

ResourcerEnumerateResourceEx goes through the resource tree of the loaded file and enumerates all found resources.

Syntax

```
void __stdcall ResourcerEnumerateResourceEx(
        ULONG_PTR FileMapVA,
        DWORD FileSize,
        void* CallBack
        );
```

Parameters

> *szFileName*
>> [in] Pointer to a null terminated string which is a full path to file whose resource will be enumerated.
>
> *CallBack*
>> [in] Address of a callback to call for each found resource.

CallBack definition

```
typedef bool(__stdcall *fResourceEnumerator)(wchar_t* szResourceType, \
    DWORD ResourceType, wchar_t* szResourceName, DWORD ResourceName, \
    DWORD ResourceLanguage, DWORD ResourceData, DWORD ResourceSize);
```

Return value

> None.

Remarks

> None.

Example

> None.

Static module

The 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,
                       StaticSectionDecrypt function and **StaticMemoryDecryptSpecial** 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

#define UE_STATIC_DECRYPTOR_FOREWARD 1
#define UE_STATIC_DECRYPTOR_BACKWARD 2
```

Constants used by: **StaticMemoryDecompress** function

```
#define UE_STATIC_APLIB 1
#define UE_STATIC_APLIB_DEPACK 2
#define UE_STATIC_LZMA 3
```

Constants used by: **StaticHashMemory** function and **StaticHashFile** function

```
#define UE_STATIC_HASH_MD5 1
#define UE_STATIC_HASH_SHA1 2
#define UE_STATIC_HASH_CRC32 3
```

StaticFileOpen function

StaticFileOpen opens a handle to a selected file. Using this function produces a normal Windows file handle which can be used for other operations.

Syntax

```
bool __stdcall StaticFileOpen(
            char* szFileName,
            DWORD DesiredAccess,
            LPHANDLE FileHandle,
            LPDWORD FileSizeLow,
            LPDWORD FileSizeHigh
            );
```

Parameters

> *szFileName*
>
>> [in] Pointer to a null terminated string which is a full path to file which will be opened.
>
> *DesiredAccess*
>
>> [in] Determines the type of file access you will have. These values are defined in Windows as `GENERIC_READ` and `GENERIC_WRITE`.
>
> *FileHandle*
>
>> [out] Pointer to HANDLE variable which will receive the open file handle.
>
> *FileSizeLow*
>
>> [out] Pointer to DWORD variable which will receive the file size.
>
> *FileSizeHigh*
>
>> [out] Pointer to DWORD variable which will receive the high file size. This parameter can be *NULL* if you don't intend to open files larger than 4 Gb.

Return value

> Boolean switch indicating whether or not the function was successful.

Example

> None.

StaticFileGetContent function

StaticFileGetContent retrieves data from opened file at the specified location from the start of the file.

Syntax

```
bool __stdcall StaticFileGetContent(
            HANDLE FileHandle,
            DWORD FilePositionLow,
            LPDWORD FilePositionHigh,
            void* Buffer,
            DWORD Size
            );
```

Parameters

FileHandle

[in] Opened file HANDLE. File must be opened with read access.

FilePositionLow

[in] Low offset from which data retrieval will start. For files fewer than 4 Gb in size this is the only positioning parameter you need.

FilePositionHigh

[in] High offset from which data retrieval will start. For file greater than 4 Gb in size this parameter is needed only if the data needs to be read from addresses greater than 0xFFFFFFFF.

Buffer

[out] Pointer to buffer which will receive the retrieved data. Buffer must be large enough to hold all requested data.

Size

[in] Size of the data to retrieve from the opened file.

Return value

Boolean switch indicating whether or not the function was successful.

Example

None.

StaticFileClose function

StaticFileClose closes an open file handle.

Syntax

```
void __stdcall StaticFileClose(
            HANDLE FileHandle
            );
```

Parameters

>   *FileHandle*
>
>>   [in] Opened file HANDLE which will be closed by the function.

Return value

>   None.

Example

>   None.

StaticFileLoad function

StaticFileLoad either maps the selected file, or simulates its loading. Depending on the type of static unpacker being developed, you need to specify the type of file memory access. File content can be changed with either type, without affecting the loading type you are using.

Syntax

```
bool __stdcall StaticFileLoad(
          char* szFileName,
          DWORD DesiredAccess,
          bool SimulateLoad,
          LPHANDLE FileHandle,
          LPDWORD LoadedSize,
          LPHANDLE FileMap,
          PULONG_PTR 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 if you are mapping the file. It 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 ULONG_PTR variable which will receive the mapped file base address.

Return value

Boolean switch indicating whether or not the function was successful.

StaticFileUnload function

StaticFileUnload either maps the selected file or simulates its loading. Depending on the type of static unpacker being developed, you need to specify the type of file memory access. File content can be changed with either type, without affecting the loading type you are using.

Syntax

```
bool __stdcall StaticFileUnload(
          char* szFileName,
          bool CommitChanges,
          HANDLE FileHandle,
          DWORD LoadedSize,
          HANDLE FileMap,
          ULONG_PTR 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 increase its size.
>
> *FileMap*
>> [in] File mapping handle returned by *StaticFileLoad*.
>
> *FileMapVA*
>> [in] Base address at which the file is loaded returned by *StaticFileLoad*.

Return value

Boolean switch indicating whether or not the function was successful.

StaticMemoryDecrypt function

StaticMemoryDecrypt decrypts the selected memory range, using the specified 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 the beginning of the memory block you want to decrypt in the remote process.
>
> *MemorySize*
>> [in] Size of the memory to decrypt.
>
> *DecryptionType*
>> [in] Specifies which decryption engine to use. Decryption algorithm options: `UE_STATIC_DECRYPTOR_XOR`, `UE_STATIC_DECRYPTOR_SUB` or `UE_STATIC_DECRYPTOR_ADD`.
>
> *DecryptionKeySize*
>> [in] Specifies the size of the decryption key that will 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] Specifies the decryption key to use.

Return value

> None.

Example

> StaticMemoryDecrypt(0x00401000, 0x1000, UE_STATIC_DECRYPTOR_XOR, UE_STATIC_KEY_SIZE_1, 0x90);

StaticMemoryDecryptEx function

StaticMemoryDecryptEx decrypts the selected memory range with a custom decryption algorithm. The specified 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 the beginning of the memory block you want to decrypt in the remote process.
>
>   *MemorySize*
>
>>   [in] Size of the memory to decrypt.
>
>   *DecryptionKeySize*
>
>>   [in] Specifies the size of the decryption key that will 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`. If you use a custom size, make sure the *MemorySize % DecryptionKeySize* is NULL. If modus isn't *NULL*, the last few bytes of the memory content will not be decrypted.
>
>   *DecryptionCallBack*
>
>>   [in] Callback that 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

StaticSectionDecrypt decrypts the selected memory range with a custom decryption algorithm. The specified 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 either the *StaticFileLoad* function or *Windows* API for file mapping.
>
> *SectionNumber*
>> [in] Number of the section to decrypte. Section numbers range 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* the engine assumes that the file was mapped.
>
> *DecryptionType*
>> [in] Specifies which decryption engine to use. Decryption algorithm options: `UE_STATIC_DECRYPTOR_XOR`, `UE_STATIC_DECRYPTOR_SUB` or `UE_STATIC_DECRYPTOR_ADD`.
>
> *DecryptionKeySize*
>> [in] Determines the size of the decryption key that will 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] The decryption key to use.

Return value

> None.

StaticMemoryDecryptSpecial function

StaticMemoryDecryptSpecial decrypts the selected memory range with a custom decryption algorithm. The specified callback will be called for each member of the encrypted block.

Syntax

```
void __stdcall StaticMemoryDecryptSpecial(
        LPVOID MemoryStart,
        DWORD MemorySize,
        DWORD DecryptionKeySize,
        DWORD SpecDecryptionType,
        void* DecryptionCallBack
        );
```

Parameters

> *MemoryStart*
>> [in] Pointer to the beginning of the memory block you want to decrypt in the remote process.
>
> *MemorySize*
>> [in] Size of the memory to decrypt.
>
> *DecryptionKeySize*
>> [in] Specifies the size of the decryption key that will 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`. If you use a custom size, make sure the *MemorySize % DecryptionKeySize* is NULL. If modus isn't *NULL*, the last few bytes of the memory content will not be decrypted.
>
> *SpecDecryptionType*
>> [in] Type of decryption to perform, can be: `UE_STATIC_DECRYPTOR_FOREWARD` or `UE_STATIC_DECRYPTOR_BACKWARD`. This sets decryption direction.
>
> *DecryptionCallBack*
>> [in] Callback that 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.

StaticMemoryDecompress function

StaticMemoryDecompress decompresses the selected memory block to designated destination. Source memory block can be compressed with the following methods: aplib and lzma.

Syntax

```
void __stdcall StaticMemoryDecompress(
          void* Source,
          DWORD SourceSize,
          void* Destination,
          DWORD DestinationSize,
          int Algorithm
          );
```

Parameters

> *Source*
>
> > [in] Pointer to the beginning of the memory block you want to decompress.
>
> *SourceSize*
>
> > [in] Size of the memory to decompress.
>
> *Destination*
>
> > [in] Pointer to memory buffer to hold decompressed content.
>
> *DestinationSize*
>
> > [in] Size of the destination buffer.
>
> *Algorithm*
>
> > [in] Specifies algorithm to use for decompression, can be one of the following:
> > `UE_STATIC_APLIB, UE_STATIC_APLIB_DEPACK` or `UE_STATIC_LZMA.`

Return value

> Boolean switch indicating whether or not the function was successful.

Example

> None.

StaticRawMemoryCopy function

StaticRawMemoryCopy copies data from mapped file directly to file on the disk. This function is used as a workaround the memory usage problem when reading data from mapped files.

Syntax

```
bool __stdcall StaticRawMemoryCopy(
        HANDLE hFile,
        ULONG_PTR FileMapVA,
        ULONG_PTR VitualAddressToCopy,
        DWORD Size,
        bool AddressIsRVA,
        char* szDumpFileName
        );
```

Parameters

>
> *hFile*
>
> > [in] Handle of the opened file.
>
> *FileMapVA*
>
> > [in] Pointer to the mapped file content. This pointer is set by either the *StaticFileLoad* function or *Windows* API for file mapping.
>
> *VitualAddressToCopy*
>
> > [in] Virtual address of data inside the mapped file which will be copied to new file.
>
> *Size*
>
> > [in] Number of bytes to copy.
>
> *AddressIsRVA*
>
> > [in] Boolean switch indicating whether or not the *VirtualAddressToCopy* is a relative address.
>
> *szDumpFileName*
>
> > [in] Pointer to the full path of the file in which to write the memory content.

Return value

>
> Boolean switch indicating whether or not the function was successful.

Example

>
> None.

StaticRawMemoryCopyEx function

StaticRawMemoryCopyEx copies data from opened file directly to another file on the disk.

Syntax

```
bool __stdcall StaticRawMemoryCopyEx(
            HANDLE hFile,
            DWORD RawAddressToCopy,
            DWORD Size,
            char* szDumpFileName
            );
```

Parameters

>   *hFile*
>
>> [in] Handle of the opened file.
>
>   *RawAddressToCopy*
>
>> [in] Offset from which the file copy will be performed.
>
>   *Size*
>
>> [in] Number of bytes to copy.
>
>   *szDumpFileName*
>
>> [in] Pointer to the full path of the file in which to write the memory content.

Return value

>   Boolean switch indicating whether or not the function was successful.

Example

>   None.

StaticRawMemoryCopyEx64 function

StaticRawMemoryCopyEx64 copies data from opened file directly to another file on the disk. This function is used for file greater than 4 Gb in size.

Syntax

```
bool __stdcall StaticRawMemoryCopyEx64(
          HANDLE hFile,
          DWORD64 RawAddressToCopy,
          DWORD64 Size,
          char* szDumpFileName
          );
```

Parameters

> *hFile*
>
>> [in] Handle of the opened file.
>
> *RawAddressToCopy*
>
>> [in] Offset from which the file copy will be performed.
>
> *Size*
>
>> [in] Number of bytes to copy.
>
> *szDumpFileName*
>
>> [in] Pointer to the full path of the file in which to write the memory content.

Return value

> Boolean switch indicating whether or not the function was successful.

Example

> None.

StaticHashMemory function

StaticHashMemory hashes the selected memory part with a selected hashing algorithm.

Syntax

```
bool __stdcall StaticHashMemory(
            void* MemoryToHash,
            DWORD SizeOfMemory,
            void* HashDigest,
            bool OutputString,
            int Algorithm
            );
```

Parameters

> *MemoryToHash*
>> [in] Pointer to memory to be hashed.
>
> *SizeOfMemory*
>> [in] Size of the memory to be hashed.
>
> *HashDigest*
>> [out] Pointer to a buffer which will receive the hash. If *OutputString* is set to *TRUE* output will be a string otherwise it will be an array containing the hash value.
>
> *OutputString*
>> [in] This input variable determines the output type. If it is set to *TRUE* the output will be a string otherwise it will be an array containing the hash value.
>
> *Algorithm*
>> [in] Determines which algorithm to use for hashing, can be one of the following: `UE_STATIC_HASH_MD5, UE_STATIC_HASH_SHA1` or `UE_STATIC_HASH_CRC32.`

Return value

> Boolean switch indicating whether or not the function was successful.

Example

> None.

StaticHashFile function

StaticHashFile hashes the selected file with a selected hashing algorithm.

Syntax

```
bool __stdcall StaticHashFile(
        char* szFileName,
        char* HashDigest,
        bool OutputString,
        int Algorithm
        );
```

Parameters

> *szFileName*
>> [in] Path to file which will be hashed.
>
> *HashDigest*
>> [out] Pointer to a buffer which will receive the hash. If *OutputString* is set to *TRUE* output will be a string otherwise it will be an array containing the hash value.
>
> *OutputString*
>> [in] This input variable determines the output type. If it is set to *TRUE* the output will be a string otherwise it will be an array containing the hash value.
>
> *Algorithm*
>> [in] Determines which algorithm to use for hashing, can be one of the following: `UE_STATIC_HASH_MD5, UE_STATIC_HASH_SHA1` or `UE_STATIC_HASH_CRC32.`

Return value

> Boolean switch indicating whether or not the function was successful.

Example

> None.

Handler module

The handler module has functions designed to work with open handles and mutexes. Additionally, it can find processes which use the designated mutex, or close all lock handles to selected files.

Handler module constants

Constants used by: **HandlerGetHandleDetails** function

```
#define UE_OPTION_HANDLER_RETURN_HANDLECOUNT 1
#define UE_OPTION_HANDLER_RETURN_ACCESS 2
#define UE_OPTION_HANDLER_RETURN_FLAGS 3
#define UE_OPTION_HANDLER_RETURN_TYPENAME 4
#define UE_OPTION_HANDLER_RETURN_TYPENAME_UNICODE 5
```

Handler module structures

Structures used by: **HandlerEnumerateOpenHandles** function and **HandlerEnumerateLockHandles** function

```
typedef struct{
        ULONG ProcessId;
        HANDLE hHandle;
}HandlerArray, *PHandlerArray;
```

HandlerGetActiveHandleCount function

HandlerGetActiveHandleCount gets 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 the *Windows* API.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

ReversingLabs Corporation

HandlerIsHandleOpen function

HandlerIsHandleOpen checks whether 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 the *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

HandlerGetHandleName retrieves the name of an open handle in a remote process.

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 the *Windows* API.
>
> *hHandle*
>> [in] Handle inside the remote process whose name you want to find.
>
> *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 the handle name, or NULL if the supplied string can't be retrieved.

Remarks

> **CAUTION:** The string with the translated native name is stored inside the engine, which makes this function multi thread unsafe.

Example

> None.

HandlerGetHandleDetails function

HandlerGetHandleDetails retrieves additional information about an open handle in a remote process.

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 the *Windows* API.
>
> *hHandle*
>> [in] Handle inside the remote process whose name you want.
>
> *InformationReturn*
>> [in] Defines the type of the information to return about the specified 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
>> - UE_OPTION_HANDLER_RETURN_TYPENAME_UNICODE

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

CAUTION: The string with the translated native name is stored inside the engine, which makes this function multi thread unsafe.

ReversingLabs Corporation

HandlerEnumerateOpenHandles function

HandlerEnumerateOpenHandles gets 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 the *Windows* API.
>
> *HandleDataBuffer*
> > [out] Pointer to the array which will receive the lock handle information. The array is
> > > defined as stated in the
> >
> > *Handler module* .
>
> *MaxHandleCount*
> > [in] Specifies the maximum possible entries in the array.

Return value

> Returns the number of open handles found.

Remarks

> None.

Example

> None.

HandlerIsFileLocked function

HandlerIsFileLocked checks whether 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 whether the provided string is a folder.
>
> *NameIsTranslated*
>> [in] Boolean switch indicating whether 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

HandlerCloseAllLockHandles checks whether the selected file or folder is locked by open handles from any of the running processes, and if so, whether it should to try to close all of them, regardless of the process locking the file or folder. Use this function with caution because it can cause any applications relying on those handles to crash.

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 whether the provided string is a folder.
>   *NameIsTranslated*
>>   [in] Boolean switch indicating whether 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

HandlerEnumerateLockHandles puts the information about all file or folder locking handles into one array.

Syntax

```
long __stdcall HandlerEnumerateLockHandles(
        char* szFileOrFolderName,
        bool NameIsFolder,
        bool NameIsTranslated,
        LPVOID HandleDataBuffer,
        DWORD MaxHandleCount
        );
```

Parameters

> *NameIsFolder*
>> [in] Boolean switch indicating whether the provided string is a folder.
>
> *NameIsTranslated*
>> [in] Boolean switch indicating whether the string has already been translated.
>
> *HandleDataBuffer*
>> [out] Pointer to the array that will receive the lock handle information. The array is
>>> defined as stated in the
>>
>> *Handler module* .
>
> *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

HandlerCloseRemoteHandle closes handles in a remote process. Use this function with caution because it can cause any applications relying on those handles to crash.

Syntax

```
bool __stdcall HandlerCloseRemoteHandle(
         HANDLE hProcess,
         HANDLE hHandle
         );
```

Parameters

> *hProcess*
>> [in] Handle of the process whose handle will be closed.
>
> *hHandle*
>> [in] Handle to close inside the remote process.

Return value

> Boolean switch indicating whether or not the handle has closed.

Remarks

> None.

Example

> None.

HandlerEnumerateOpenMutexes function

HandlerEnumerateOpenMutexes puts the information about all open mutex handles for the specified process into an 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 the *Windows* API.

HandleDataBuffer

[out] Pointer to the array that 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 an array of HANDLEs.

Example

None.

HandlerGetOpenMutexHandle function

HandlerGetOpenMutexHandle gets 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 the *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

HandlerGetProcessIdWhichCreatedMutex gets 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.

Extension module

The extension module has functions designed to work with plugins created for the TitanEngine. Functions inside this module provide interface to manipulating loaded plugins.

Guide to writing extensions for TitanEngine

TitanEngine extensions are created as normal dynamic link libraries placed in the selected folder (either **.\plugins\x86** or **.\plugins\x64**) for the engine to load. Following export functions are used by the engine:

- **TitanResetPlugin** – Function which is called every time the debugging starts within the DebuLoop function.
- **TitanReleasePlugin** – Function which is called when the plugin gets unloaded or the engine shuts down.
- **TitanRegisterPlugin** – Function which is called when the plugin gets loaded by the engine. Plugin should register itself by using a unique name (up to 64 characters long) and optionally fill the version information.
- **TitanDebuggingCallBack** – Function which is called for every debug even registered by the engine. This function has a CallReason parameter which can be one of the following: `UE_PLUGIN_CALL_REASON_PREDEBUG`, `UE_PLUGIN_CALL_REASON_EXCEPTION` or `UE_PLUGIN_CALL_REASON_POSTDEBUG`.

Extension structure and function definitions

```
typedef struct{
    char PluginName[64];
    DWORD PluginMajorVersion;
    DWORD PluginMinorVersion;
    HMODULE PluginBaseAddress;
    void* TitanDebuggingCallBack;
    void* TitanRegisterPlugin;
    void* TitanReleasePlugin;
    void* TitanResetPlugin;
    bool PluginDisabled;
}PluginInformation, *PPluginInformation;

__declspec(dllexport) void __stdcall TitanResetPlugin();

__declspec(dllexport) void __stdcall TitanReleasePlugin();

__declspec(dllexport) bool __stdcall TitanRegisterPlugin(char* szPluginName,
LPDWORD titanPluginMajorVersion, LPDWORD titanPluginMinorVersion);

__declspec(dllexport) void __stdcall TitanDebuggingCallBack(LPDEBUG_EVENT
debugEvent, int CallReason);
```

ExtensionManagerIsPluginLoaded function

ExtensionManagerIsPluginLoaded checks if the selected plugin is loaded.

Syntax

```
bool __stdcall ExtensionManagerIsPluginLoaded(
            char* szPluginName
            );
```

Parameters

szPluginName
> [in] Unique identifier plugin uses to register itself.

Return value

> Boolean switch indicating whether or not the plugin is still loaded.

Remarks

> None.

Example

> None.

ExtensionManagerIsPluginEnabled function

ExtensionManagerIsPluginEnabled checks if the selected plugin is enabled.

Syntax

```
bool __stdcall ExtensionManagerIsPluginEnabled(
            char* szPluginName
            );
```

Parameters

> *szPluginName*
>> [in] Unique identifier plugin uses to register itself.

Return value

> Boolean switch indicating whether or not the plugin is enabled.

Remarks

> None.

Example

> None.

ExtensionManagerDisablePlugin function

ExtensionManagerDisablePlugin temporarily disables the use of the selected plugin.

Syntax

```
bool __stdcall ExtensionManagerDisablePlugin(
            char* szPluginName
            );
```

Parameters

>   *szPluginName*
>   [in] Unique identifier plugin uses to register itself.

Return value

>   Boolean switch indicating whether or not the plugin was disabled.

Remarks

>   None.

Example

>   None.

ExtensionManagerDisableAllPlugins function

ExtensionManagerDisableAllPlugins temporarily disables the use of all loaded plugins.

Syntax

```
bool __stdcall ExtensionManagerDisableAllPlugins();
```

Parameters

None.

Return value

Boolean switch indicating whether or not the plugins were disabled.

Remarks

None.

Example

None.

ExtensionManagerEnablePlugin function

ExtensionManagerEnablePlugin enables the use of the selected previously disabled plugin.

Syntax

```
bool __stdcall ExtensionManagerEnablePlugin(
            char* szPluginName
            );
```

Parameters

> *szPluginName*
> > [in] Unique identifier plugin uses to register itself.

Return value

> Boolean switch indicating whether or not the plugin was enabled.
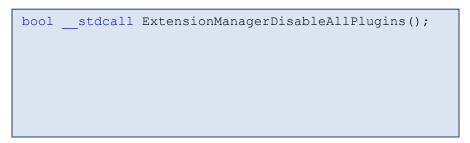
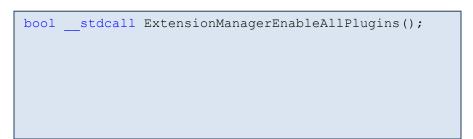**Remarks**

> None.

Example

> None.

ExtensionManagerEnableAllPlugins function

ExtensionManagerEnableAllPlugins enables the use of all previously disabled plugins.

Syntax

```
bool __stdcall ExtensionManagerEnableAllPlugins();
```

Parameters

None.

Return value

Boolean switch indicating whether or not the plugins were enabled.

Remarks

None.

Example

None.

ExtensionManagerUnloadPlugin function

ExtensionManagerUnloadPlugin unloads the selected plugin from the current session. It will still be loaded the next time the engine starts. Only way to actually remove the plugin completely is to delete it from the plugins folder.

Syntax

```
bool __stdcall ExtensionManagerUnloadPlugin(
            char* szPluginName
            );
```

Parameters

> *szPluginName*
> > [in] Unique identifier plugin uses to register itself.

Return value

> Boolean switch indicating whether or not the plugin was unloaded.

Remarks

> None.

Example

> None.

ExtensionManagerUnloadAllPlugins function

ExtensionManagerUnloadAllPlugins unloads the selected plugin from the current session. All plugins will still be loaded the next time the engine starts. Only way to actually remove any of the plugins completely is to delete them from the plugins folder.

Syntax

```
bool __stdcall ExtensionManagerUnloadAllPlugins();
```

Parameters

None.

Return value

Boolean switch indicating whether or not the plugins were unloaded.

Remarks

None.

Example

None.

ExtensionManagerGetPluginInfo function

ExtensionManagerGetPluginInfo retrieves the information about the selected plugin. Data returned can be modified in order to manipulate the plugin behavior.

Syntax

```
void* __stdcall ExtensionManagerGetPluginInfo(
            char* szPluginName
            );
```

Parameters

>   *szPluginName*
>
>>   [in] Unique identifier plugin uses to register itself.

Return value

>   Pointer to *PluginInformation* structure or *NULL* if the selected plugin isn't currently loaded.

Remarks

>   None.

Example

>   None.

Engine module

The engine module isn't a separate module or a functional part; instead it is a top level engine functionality which utilizes multiple engine functions to perform a certain task.

EngineCreateMissingDependencies function

EngineCreateMissingDependencies goes thought the import table of the selected file and creates all dynamic link libraries needed by the file which are not present on the system.

Syntax

```
bool __stdcall EngineCreateMissingDependencies(
          char* szFileName,
          char* szOutputFolder,
          bool LogCreatedFiles
          );
```

Parameters

>   *szFileName*
>
>>   [in] Pointer to the full path of the file to inspect for missing dependencies.
>
>   *szOutputFolder*
>
>>   [in] Pointer to the full path of the folder in which the files will be created. Most commonly this is the folder in which the file which will be unpacked is.
>
>   *LogCreatedFile*
>
>>   [in] Indicates whether or not to internally log all created files. Using this option later enables simple deletion of created files.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

EngineFakeMissingDependencies function

EngineFakeMissingDependencies performs low level API hooking in user mode to ensure that dynamically loaded libraries are always virtually present. This ensures that all libraries which are loaded by Windows API are seemingly present. This function should be called once the debugee execution hit the packed entry point in cases when packing shell dynamically loads libraries.

Syntax

```
bool __stdcall EngineCreateMissingDependencies(
        HANDLE hProcess
        );
```

Parameters

> *hProcess*
>
> > [in] Handle of the process whose dynamically loaded modules will be simulated.

Return value

> Boolean switch indicating whether or not the function was successful.

Remarks

> None.

Example

> None.

EngineDeleteCreatedDependencies function

EngineDeleteCreatedDependencies deletes logged created missing dependencies. This function performs disk cleanup at the end of unpacking process. If files can't be deleted at that particular time they will be moved to system's temporary folder for user deletion.

Syntax

```
bool __stdcall EngineDeleteCreatedDependencies();
```

Parameters

>   None.

Return value

>   Boolean switch indicating whether or not the function was successful.

Remarks

>   None.

Example

>   None.

EngineCreateUnpackerWindow function

EngineCreateUnpackerWindow creates a default and easily customizable graphical user interface for your unpacker project. Program doesn't return from this call until the window has been closed.

Syntax

```
bool __stdcall EngineCreateUnpackerWindow(
          char* WindowUnpackerTitle,
          char* WindowUnpackerLongTitle,
          char* WindowUnpackerName,
          char* WindowUnpackerAuthor,
          void* StartUnpackingCallBack
          );
```

Parameters

>
> *WindowUnpackerTitle*
>> [in] Custom graphical user interface window title string.
>
> *WindowUnpackerLongTitle*
>> [in] Custom graphical user interface long unpacker name string.
>
> *WindowUnpackerName*
>> [in] Custom graphical user interface unpacker name string.
>
> *WindowUnpackerAuthor*
>> [in] Custom graphical user interface unpacker author name string.
>
> *StartUnpackingCallBack*
>> [in] Callback which will be called once the user presses unpack button.

CallBack definition

```
typedef void(__stdcall *fStartUnpacking)(
               char* szInputFile, bool RealignFile, bool CopyOverlay);
```

Return value

> Boolean switch indicating whether or not the function was successful.

Example

EngineCreateUnpackerWindow("UPX 1.x - 3.x", "Unpacker for UPX packed files", "RL!deUPX 1.x - 3.x", "ReversingLabs Corporation", &InitializeUnpacking);

EngineAddUnpackerWindowLogMessage function

EngineAddUnpackerWindowLogMessage can be used to show unpacking log messages while the unpacker uses built-in graphical user interface.

Syntax

```
void __stdcall EngineAddUnpackerWindowLogMessage(
            char* szLogMessage
            );
```

Parameters

szLogMessage

[in] Message which will be shown inside the unpacker log window.

Return value

None.

Remarks

None.

Example

None.

Engine unpacker simplification module

The engine module isn't a separate module or a functional part; instead it is a top level engine functionality which utilizes multiple engine functions to perform a certain task. Engine unpacker simplification functions enable easy unpacker coding for most common portable executable packers.

Engine simplification module constants

Constants used by: **EngineUnpackerSetBreakCondition** function

```
#define UE_UNPACKER_CONDITION_SEARCH_FROM_EP 1

#define UE_UNPACKER_CONDITION_LOADLIBRARY 1
#define UE_UNPACKER_CONDITION_GETPROCADDRESS 2
#define UE_UNPACKER_CONDITION_ENTRYPOINTBREAK 3
#define UE_UNPACKER_CONDITION_RELOCSNAPSHOT1 4
#define UE_UNPACKER_CONDITION_RELOCSNAPSHOT2 5
```

EngineUnpackerInitialize function

EngineUnpackerInitialize initializes simplified unpacking routines. Initialization creates an unpacking process for any provided input portable executable be it EXE or DLL.

Syntax

```
void __stdcall EngineUnpackerInitialize(
         char* szFileName,
         char* szUnpackedFileName,
         bool DoLogData,
         bool DoRealignFile,
         bool DoMoveOverlay,
         void* EntryCallBack
         );
```

Parameters

> *szFileName*
>> [in] Pointer to the full path of the file to debug.
>
> *szUnpackedFileName*
>> [in] Pointer to the full path of the file which will be created as the end result of the unpacking process.
>
> *DoLogData*
>> [in] Enables or disable automatic data logging.
>
> *DoRealignFile*
>> [in] Enables or disables file realigning at the end of the unpacking process.
>
> *DoMoveOverlay*
>> [in] Enables or disables overlay moving from original to the unpacked file at the end of the unpacking process.
>
> *EntryCallBack*
>> [in] Callback which will be called once the file reaches its packed entry point.

Return value

> Boolean switch indicating whether or not the function was successful.

Example

> None.

EngineUnpackerSetBreakCondition function

EngineUnpackerSetBreakCondition function searches the debugee memory and sets breakpoints for selected patterns.

Syntax

```
bool __stdcall EngineUnpackerSetBreakCondition(
            void* SearchStart,
            DWORD SearchSize,
            void* SearchPattern,
            DWORD PatternSize,
            DWORD PatternDelta,
            ULONG_PTR BreakType,
            bool SingleBreak,
            DWORD Parameter1,
            DWORD Parameter2
            );
```

Parameters

> *SearchStart*
>> [in] Indicates the point in memory from which search will be performed.
> *SearchSize*
>> [in] Size of the memory to search.
> *SearchPattern*
>> [in] Pointer to pattern to be searched.
> *PatternDelta*
>> [in] On successfully found pattern its offset is affected by delta value. This helps to set the breakpoint on any byte relative to the found pattern. This variable is a signed integer which means that delta can be negative value as well.
> *BreakType*
>> [in] Callback address which will be called once the breakpoint has been hit. This is a default breakpoint callback type. Additionally following values are considered valid:
>>
>> - `UE_UNPACKER_CONDITION_LOADLIBRARY`
>> - `UE_UNPACKER_CONDITION_GETPROCADDRESS`
>> - `UE_UNPACKER_CONDITION_ENTRYPOINTBREAK`
>> - `UE_UNPACKER_CONDITION_RELOCSNAPSHOT1`
>> - `UE_UNPACKER_CONDITION_RELOCSNAPSHOT2`

SingleBreak

[in] Boolean switch indicating whether or not the breakpoint will be used only once.

Parametar1

[in] If you use predefined values as your BreakType Parameter1 indicates the following data will be used by the TitanEngine when breakpoints are processed:

- LoadLibrary
  - o Parameter1 is the register (UE_EAX ... UE_EDI) which points to the remote DLL name string.
- GetProcAddress
  - o Parameter1 is the register (UE_EAX ... UE_EDI) which points to the remote API name string or contains API ordinal number. Difference between the two is automatically detected.
- EntryPointBreak
  - o Parameter1 is unused.
- RelocateSnapshot1
  - o Relative virtual address from which the memory snapshot will start.
- RelocateSnapshot2
  - o Relative virtual address from which the memory snapshot will start.

Parametar2

[in] If you use predefined values as your BreakType Parameter2 indicates the following data will be used by the TitanEngine when breakpoints are processed:

- LoadLibrary
  - o Parameter2 is the register (UE_EAX ... UE_EDI) which holds the first import trunk address. This is optional and can be set to NULL.
- GetProcAddress
  - o Parameter2 is the register (UE_EAX ... UE_EDI) which holds the API write location.
- EntryPointBreak
  - o Parameter2 is unused.
- RelocateSnapshot1
  - o Size of the memory to snapshot.
- RelocateSnapshot2
  - o Size of the memory to snapshot.

ReversingLabs Corporation

Return value

Boolean switch indicating whether or not the function was successful in finding the selected pattern and placing the breakpoint there.

Remarks

For simplification to work correctly you must define LoadLibrary, GetProcAddress and entry point breakpoints as a minimum of functionality.

If relocation snapshot two isn't defined second snapshot is automatically performed once the entry point breakpoint is hit.

If you don't set the original entry point and use the default entry point callback EIP/RIP address will be used to set that data.

Example

None.

EngineUnpackerSetEntryPointAddress function

EngineUnpackerSetEntryPointAddress sets the original entry point address. This function is used if breaking at the original entry point isn't possible and the address of the original entry point must be read by the program.

Syntax

```
void __stdcall EngineUnpackerSetEntryPointAddress(
          ULONG_PTR UnpackedEntryPointAddress
          );
```

Parameters

> *UnpackedEntryPointAddress*
> > [in] Sets the original entry point before the unpacking finalization.

Return value

> None.

Example

> None.

EngineUnpackerFinalizeUnpacking function

EngineUnpackerFinalizeUnpacking finalizes the unpacking process performing the memory dumping and image fixing procedures. This function should only be called if EngineUnpackerSetEntryPointAddress calling was necessary.

Syntax

```
void __stdcall EngineUnpackerFinalizeUnpacking();
```

Parameters

None.

Return value

None.

Example

None.

TitanEngine 2.0.1 - history

- **SDK**
  - Extended SDK headers to support Delphi
  - Changes to SDK.h to support non MSVC compilers
  - Changed the file/folder layout and fixed relative paths in RC file
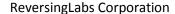 - Removed type-o mistakes and bad function definitions
- **Bug fixes**
  - Fixed some mistakes in SDK.h
  - Added missing entries in TitanEngine.def
  - FindEx searched only 0x1000 bytes
  - Fixed a minor bug inside injectTerminator
  - EngineExtractForwarderData crash on invalid input
  - RelocaterGrabRelocationTableEx doesn't grab whole table
  - RelocaterCompareTwoSnapshots crash when first DWORD of the snapshot is relocated
  - RelocaterCompareTwoSnapshots doesn't correctly compare two snapshots for some packers
  - ImporterAutoFixIATEx very rare crash handled by handler but it makes import table invalid
  - ImporterAutoFixIATEx incorrect import fixing if import is only exported by ordinal
  - ImporterAutoFixIATEx incorrect stepping if step is greater than one
  - Fixed StaticLoadFile & StaticUnloadFile to be compatible with x64
  - Fixed a problem with imports and WinSxS folder
  - Fixed a crash with Librarian on Windows XP x64
  - Fixed problem with NtdllDefWindowProc on Vista x64
  - Fixed problem with deleting temp files: DLLLoader.exe & *.module base reserve file
  - Fixed problem with DumpProcess on x64 systems
  - Fixed problem with DumpProcess and empty last PE sections
  - Fixed problem with DumpProcess and files with non default SectionAlignment
  - Fixed problem with DumpProcess and dumping PAGE_GUARD protected memory
  - Fixed UPX unpacker sample not working on files packed with --lzma option
  - Fixed problem with Exporter module and building new export table under x64 systems
  - Fixed problem with Importer module and Windows 7 kernelbase.dll MiniWin
  - Fixed problem with RealignPE/RealignPEEx and files with non default SectionAlignment
- **Additions**
  - Fill & FillEx now have a default fill byte value of 0x90 if no fill byte is supplied
  - SetHardwareBreakPointEx function for setting breakpoints in custom threads
  - Global variable UE_ENGINE_RESET_CUSTOM_HANDLER set to TRUE resetting custom handler on debug init
  - Improved speed of API & DLL data resolving
  - Added function: ImporterAddNewOrdinalAPI
  - Added function: ImporterGetAPIOrdinalNumber
  - Added function: ImporterGetAPIOrdinalNumberFromDebugee
  - Added function: ImporterGetForwardedAPIOrdinalNumber
  - Added function: ImporterGetLastAddedDLLName
  - Added function: ImporterGetDLLName
  - Added function: GetUnusedHardwareBreakPointRegister
  - Changed function: HideDebugger, less parameters and x64 compliant
  - Added function: UnHideDebugger
  - Added function: GetPEBLocation
- **Samples**
  - Unpacking ASPack 2.12, features usage of: RelocaterGrabRelocationTableEx and GetRemoteString
  - Unpacking FSG 2.0, features usage of: overlay detection
  - Unpacking PeCompact 2.0 - 3.x, features usage of: ImporterAutoFixIATEx with custom callback for fixing redirection
  - Unpacking DEF 1.0, features usage of: static unpacker functions
  - Unpacking LameCrypt 1.0, features usage of: static unpacker functions

TitanEngine 2.0.2 - history

- **SDK/PDK**
  - o Added functions to support UNICODE
  - o Extended SDK headers to support MASM32
  - o Changes to SDK.h to support non MSVC compilers, Set default structure align to: 1 byte
  - o Removed type-o mistakes and bad function definitions
- **Bug fixes**
  - o Fixed some mistakes in SDK.h
  - o Fixed a bug in SetPE32DataForMappedFile
  - o Fixed a bug with import processing and Windows7 x64
  - o Fixed a bug in PastePEHeader which made it not work on x64
  - o Fixed a bug in PastePEHeader which prevented header paste when header doesn't have write attribute
  - o Fixed a rare disassemble crash which happened due to distorm not having enough space
  - o Fixed problem with GetPE32DataFromMappedFileEx and x64
  - o Fixed a rare problem with FindEx and some memory ranges
  - o Fixed a bug inside LibrarianGetLibraryInfoEx
- **Additions**
  - o Added functions
    - ▪ RelocaterRelocateMemoryBlock, ThreaderImportRunningThreadData, EngineFakeMissingDependencies
    - ▪ EngineDeleteCreatedDependencies, EngineCreateMissingDependencies, EngineCreateMissingDependenciesW
    - ▪ ExtensionManagerIsPluginLoaded, ExtensionManagerIsPluginEnabled, ExtensionManagerDisablePlugin
    - ▪ ExtensionManagerDisableAllPlugins, ExtensionManagerEnablePlugin, ExtensionManagerEnableAllPlugins
    - ▪ ExtensionManagerUnloadPlugin, ExtensionManagerUnloadAllPlugins, ExtensionManagerGetPluginInfo
    - ▪ HooksSafeTransition, HooksSafeTransitionEx, HooksIsAddressRedirected, HooksGetTrampolineAddress
    - ▪ HooksGetHookEntryDetails, HooksInsertNewRedirection, HooksInsertNewIATRedirection
    - ▪ HooksInsertNewIATRedirectionEx, HooksRemoveRedirection, HooksRemoveRedirectionsForModule
    - ▪ HooksRemoveIATRedirection, HooksDisableRedirection, HooksDisableRedirectionsForModule
    - ▪ HooksDisableIATRedirection, HooksEnableRedirection, HooksEnableRedirectionsForModule
    - ▪ HooksEnableIATRedirection, HooksScanModuleMemory, HooksScanEntireProcessMemory
    - ▪ HooksScanEntireProcessMemoryEx, StaticRawMemoryCopy, StaticMemoryDecompress,
    - ▪ StaticMemoryDecryptSpecial, ResourcerFindResource, ResourcerEnumerateResource,
    - ▪ StaticHashMemory, StaticHashFile, IsFileBeingDebugged
  - o Added TitaniumHooks as a module and a separate project
- **Samples**
  - o Samples of using Hooks module: HooksDemo and LoaderSpy
  - o Plugin samples
    - ▪ TitanEngine: ASMExtractor, DataExtractor, lynxImpRec & Nexus
    - ▪ OllyDBG: TitaniumHandles & ImportStudio
    - ▪ PeID: TitaniumOverlay
  - o Tool samples
    - ▪ LameCrypter
  - o Unpacker samples
    - ▪ C
      - • MarioPack, CryptoCrackPE, ExeFog, MEW, PackMan, nPack, yC
    - ▪ Delphi
      - • MEW 5, PeX
    - ▪ MASM
      - • UPX

TitanEngine 2.0.3 - history

- **SDK/PDK**
  - Extended SDK headers to support Python, LUA and C++
  - Removed type-o mistakes and bad function definitions
  - Documented missing functions
- **Bug fixes**
  - Fixed a bug inside Find/Ex
  - Fixed a bug fix inside EngineSimulateNtLoaderW
  - Enabled code execution inside mapped images loaded with all access
  - Fixed a bug in SetPE32Data/W which prevented data update due to incorrect file access
  - Fixed a bug inside ConvertVAtoFileOffset/Ex which incorrectly converted addresses in some cases
  - Multiple changes inside IsPE32FileValidEx/W making it compatible with Microsoft compilers
  - Multiple changes inside FixBrokenPE32FileEx/W making it compatible with Microsoft compilers
  - Fixed a bug inside debugging logic that causes the debugger to handle hardware breakpoints set by the debugee
  - Fixed a bug inside StepOver making it unsafe to execute with self modifying code with multi byte breakpoints turned on
  - Fixed a bug inside StepInto making it execute one time more than specified by the program
  - Added an option to select the type of breakpoint set with SetBPX and SetBPXEx
  - Implemented UD2 breakpoints correctly inside debugging logic
  - Fixed a bug inside ImporterAddNewAPI with ordinal logic
  - Improved the accuracy of ImporterEstimatedSize function
  - Improved ImporterFindAPIWriteLocation to take ordinals into the account
  - Fixed a bug inside ImporterRelocateWriteLocation and cases of single imported DLL file
  - Fixed a bug inside ImporterLoadImportTableW with certain compilers
  - Improved ImporterEnumAdded data to take ordinals into the account
  - Fixed a bug in HooksInsertNewIATRedirectionEx with certain compilers
  - Fixed a bug inside RemoteLoadLibraryW making it fail with DEP turned on
  - Fixed a bug inside RemoteFreeLibrary making it fail with DEP turned on
  - Fixed a bug inside RemoteExitProcess making it fail with DEP turned on
  - Fixed a bug inside StaticRawMemoryCopyW preventing it from copying all data
  - Fixed a bug inside DetachDebuggerEx that crashed the debugee upon detaching
- **Additions**
  - Added functionality
    - Added GUI interface for unpackers
    - Added unpacker simplification functions
  - Added functions
    - GetContextFPUDataEx, SetContextFPUDataEx, MatchPatternEx, MatchPattern
    - SetErrorModel, ImporterFindOrdinalAPIWriteLocation, ImporterIsForwardedAPI
    - StaticFileOpen, StaticFileGetContent, StaticRawMemoryCopyEx64, StaticFileClose
    - EngineCreateUnpackerWindow, EngineAddUnpackerWindowLogMessage
- **Samples**
  - Tool samples
    - LameCrypt, PEValidate
  - Unpacker samples
    - C
      - tELock, AHPack, AlexProtector, UPX (simplified) and FSG (simplified), DEB
    - Python
      - FSG, UPX, LameCrypt
    - LUA
      - UPX

ReversingLabs Corporation

License

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

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.