F
(https://
T
(https://
R
(https://

Dive into exceptions: caution, this may be hard

Written by hackmag
(https://web.archive.org/web/20190712173953/https://hackmag.com/author/hackmagnet/)

__try

Suppose that you are facing a practical task that requires a full implementation of exception handling in a code embedded in someone else's process, or you are creating your next PE packer/cryptor to ensure the functionality of exceptions in an unpacked image. In any case, it all comes down to the fact that the code using the exceptions is executed outside the image projected by the system boot loader, which will be the main cause of your problems.

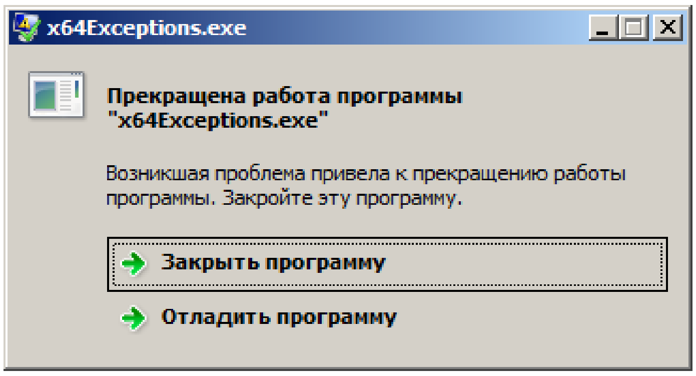
To demonstrate the problem, consider a simple example of code that copies its own image to a new area within the current AP process:

```
void exceptions_test()
{
    __try {
        int *i = 0;
        *i = 0;
    } __except (EXCEPTION_EXECUTE_HANDLER) {
        /* You may not be able to get here */
        MessageBoxA(0, "Exception intercepted", "", 0);
    }
}
void main()
{
    /* Test exceptions */
    exceptions_test ();

    / * Copy the entire current image to a new area */
    PVOID ImageBase = GetModuleHandle(NULL);
    DWORD SizeOfImage = RtlImageNtHeader(ImageBase)-&gt;OptionalHeader.Size
    PVOID NewImage = VirtualAlloc(NULL, SizeOfImage, MEM_COMMIT, PAGE_EXECU
    memcpy(NewImage, ImageBase, SizeOfImage);
    /* Editing relocks */
    ULONG_PTR Delta = (ULONG_PTR) NewImage - ImageBase;
    RelocateImage(NewImage, Delta);

    /* Call 'exceptions_test' to image copies */
    void (*new_exceptions_test)() = (void (*)()) ((ULONG_PTR) &amp;exceptic
    new_exceptions_test();
}
```

In the ‘exceptions_test’ procedure, the attempt to access a null pointer is wrapped in ‘try-except’, an MSVC extension, instead of exception filter, there is a stub that returns ‘EXCEPTION_EXECUTE_HANDLER’, which should immediately lead to the execution of code in ‘except’ block. In the first call, ‘exceptions_test’ works as expected: the exception is intercepted, and the system displays a message box. But after copying the code to a new place and calling up a copy of ‘exceptions_test’, the exception handling stops, and the application simply “crashes” with an unhandled exception message, so characteristic for this version of OS. The specific reason of such behavior will depend on the platform used for the test and, to identify it, we need to understand the exception dispatching mechanism.


x64Exceptions.exe

Прекращена работа программы "x64Exceptions.exe"

Возникшая проблема привела к прекращению работы программы. Закройте эту программу.

(https://web.archive.org/web/20190712173953/https://hackmag.com/wp-content/uploads/2015/04/01_unhandled.png)

Exception Dispatching

Regardless of the platform and the exception type, the dispatching for user mode always starts from ‘KiUserExceptionDispatcher’ in ‘ntdll’ module, the control of which is handed over from ‘KiDispatchException’, a core dispatcher (if the exception was called from user mode and has not been handled by debugger). In this example, the control is handed over for both cases of exception (during the execution of ‘exceptions_test’ and its copy in the new address), you can verify this by setting the breakpoint on ‘ntdll!KiUserExceptionDispatcher’. The code of ‘KiUserExceptionDispatcher’ is very sin and similar to the following:

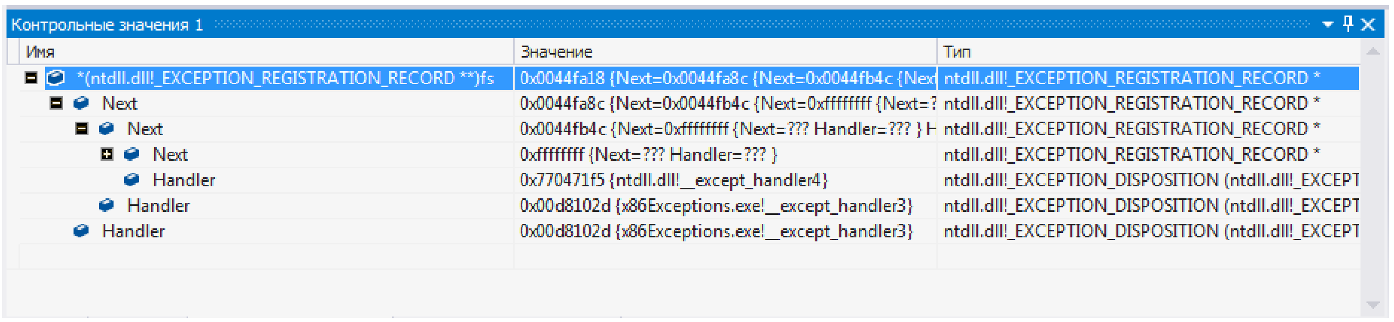
```
    VOID NTAPI KiUserExceptionDispatcher (EXCEPTION_RECORD *ExceptionRecord, CO
    {
        NTSTATUS Status;
        if (RtlDispatchException(ExceptionRecord, Context)) {
            /* Exception is handled, you can continue the execution */
            Status = NtContinue(Context, FALSE);
        }
        else {
            /* We are again raising an exception but, this time, without trying
            Status = NtRaiseException(ExceptionRecord, Context, FALSE);
        }
        ...
        RtlRaiseException(&amp;NestedException);
    }

```

where ‘EXCEPTION_RECORD’ is the structure with information about the exception, and ‘CONTEXT’ is structure of the thread context state at the time of exception. Both structures are documented in MSDN, but you are probably already familiar with them. Pointers to these data are sent to ‘ntdll!RtlDispatchException’, where the real dispatching actually takes place. In 32-bit and 64-bit systems, the mechanics of exception handling are different.

x86

The main mechanism for x86-platform is Structured Exception Handling (SEH) based on a singly linked list of exception handlers located in the stack and always available from ‘NT_TIB.ExceptionList’. The basics of this mechanism have been described many times in various papers (see Useful Resources box). So I will not repeat all this but only focus your attention to those points that intersect with our task.



Имя	Значение	Тип
*(ntdll.dll!_EXCEPTION_REGISTRATION_RECORD **)fs	0x0044fa18 {Next=0x0044fa8c {Next=0x0044fb4c {Next	ntdll.dll!_EXCEPTION_REGISTRATION_RECORD *
Next	0x0044fa8c {Next=0x0044fb4c {Next=0xffffffff {Next=?	ntdll.dll!_EXCEPTION_REGISTRATION_RECORD *
Next	0x0044fb4c {Next=0xffffffff {Next=??? Handler=??? } H	ntdll.dll!_EXCEPTION_REGISTRATION_RECORD *
Next	0xffffffff {Next=??? Handler=??? }	ntdll.dll!_EXCEPTION_REGISTRATION_RECORD *
Handler	0x770471f5 {ntdll.dll!__except_handler4}	ntdll.dll!_EXCEPTION_DISPOSITION (ntdll.dll!_EXCEPT
Handler	0x00d8102d {x86Exceptions.exe!__except_handler3}	ntdll.dll!_EXCEPTION_DISPOSITION (ntdll.dll!_EXCEPT
Handler	0x00d8102d {x86Exceptions.exe!__except_handler3}	ntdll.dll!_EXCEPTION_DISPOSITION (ntdll.dll!_EXCEPT

(https://web.archive.org/web/20190712173953/https://hackmag.com/wp-content/uploads/2015/04/02_seh_chain.png)

SEH chain dump

The fact is that, in SEH, all items of the handler list must be in the stack, which means that they are potentially exposed to overwriting in case of buffer overflow in the stack. This was successfully used by the creators of exploits — the pointer to the handler was overwritten by an address required to execute the shell code, and this was also accompanied by overwriting the pointer to the next list item, which violated the integrity of the handler chain. To increase the resilience to attacks against the programs t use SEH, Microsoft developed such mechanisms as SafeSEH (a table with addresses of “safe” hand located in ‘IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG’ of PE file), SEHOP (a simple integrity check for the frame chain) and integrated the checks under DEP system policy to be conducted in the proce of exception dispatching.

(https://

A simplified pseudocode for basic dispatching procedure ‘RtlDispatchException’ for x86-version of ‘ntdll.dll’ in Windows 8.1 can be represented (with certain assumptions) as follows:

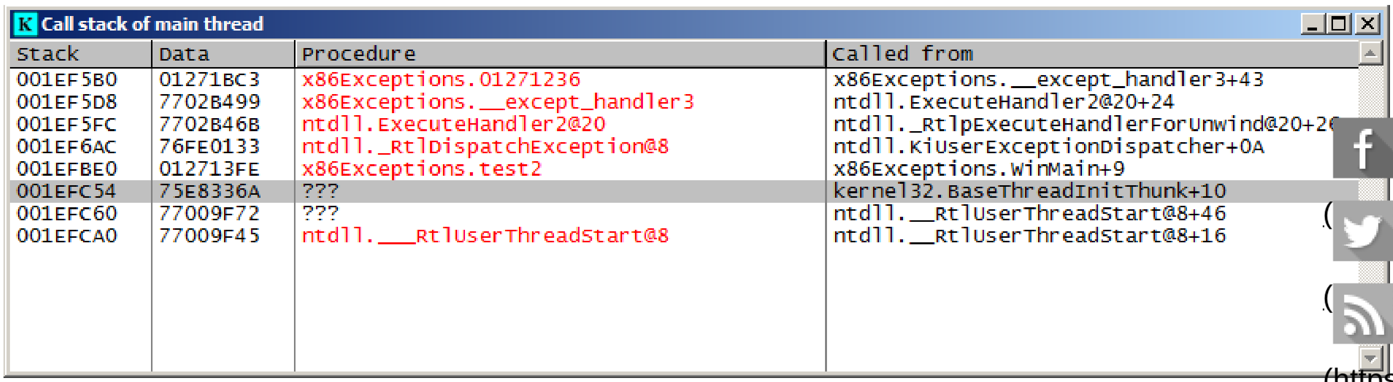
```
    void RtlDispatchException(...) // NT 6.3.9600
    {
        /* Call the chain 'Vectored Exception Handlers' */
        if (RtlpCallVectoredHandlers(exception, 1)) return 1;
        ExceptionRegistration = RtlpGetRegistrationHead();
        /* ECV (SEHOP) */
        if (!DisableExceptionChainValidation &amp;&amp;
            !RtlpIsValidExceptionChain(ExceptionRegistration, ...)) {
                if (_RtlpProcessECVPolicy != 2)
                    goto final;
                else
                    RtlReportException();
        }
        /* Go through the handler chain until you find the right handler */
        while (ExceptionRegistration != EXCEPTION_CHAIN_END) {
            /* Check the stack limits */
            if (!STACK_LIMITS(ExceptionRegistration)) {
                ExceptionRecord-&gt;ExceptionFlags |= EXCEPTION_STACK_INVALID;
                goto final;
            }
            /* Validate the handler */
            if (!RtlIsValidHandler(ExceptionRegistration, ProcessFlags)) goto f
            /* Hand over control to handler */
            RtlpExecuteHandlerForException(..., ExceptionRegistration-&gt;Handl
            ...
            ExceptionRegistration = ExceptionRegistration-&gt;Next;
        }
        ...
        final:
        /* Call the chain 'Vectored Continue Handlers' */
        RtlpCallVectoredHandlers(exception, 1);
    }

```

The presented pseudocode allows you to conclude that a successful transfer of control to SEH handler during the dispatching of an exception requires meeting the following conditions:

1. The chain of SEH frames must be correct (end with the handler ‘ntdll!FinalExceptionHandler’). The test is performed with SEHOP enabled for the process.

2. SEH frame must be located in the stack.
3. SEH frame must contain a pointer to a “valid” handler.



```
Call stack of main thread
Stack     Data      Procedure                         Called from
001EF5B0  01271BC3  x86Exceptions.01271236            x86Exceptions.__except_handler3+43
001EF5D8  7702B499  x86Exceptions.__except_handler3   ntdll.ExecuteHandler2@20+24
001EF5FC  7702B46B  ntdll.ExecuteHandler2@20          ntdll._RtlpExecuteHandlerForUnwind@20+2
001EF6AC  76FE0133  ntdll._RtlDispatchException@8     ntdll.KiUserExceptionDispatcher+0A
001EFBE0  012713FE  x86Exceptions.test2               x86Exceptions.WinMain+9
001EFC54  75E8336A  ???                               kernel32.BaseThreadInitThunk+10
001EFC60  77009F72  ???                               ntdll.__RtlUserThreadStart@8+46
001EFCA0  77009F45  ntdll.___RtlUserThreadStart@8     ntdll.__RtlUserThreadStart@8+16
```

(https://web.archive.org/web/20190712173953/https://hackmag.com/wp-content/uploads/2015/04/03_callstack_x86.png)

The call stack for exception filter

While everything is absolutely clear for the first two points and no further steps are needed to implement them, the procedure for checking the handler for its “validity” requires a more detailed explanation. The handler is checked by using ‘ntdll! RtlIsValidHandler’, the pseudocode of which for Vista SP1 version was for the first time presented to the public back in 2008 at the Black Hat conference in the United States. Even though it contained some inaccuracies, this did not prevent it from roaming through ‘copy-paste’ from one resource to another for several years. Since then, the code of this function did not change significantly, and the analysis of its version for Windows 8.1 allowed to generate the following pseudocode:

```
BOOL RtlIsValidHandler(Handler) // NT 6.3.9600
{
    if (/* Handler within the image */) {
        if (DllCharacteristics&amp;IMAGE_DLLCHARACTERISTICS_NO_SEH)
            goto InvalidHandler;
        if (/* The image is .Net assembly, 'ILonly' flag is enabled */)
            goto InvalidHandler;
        if (/* Found 'SafeSEH' table */) {
            if (/* The image is registered in 'LdrpInvertedFunctionTable
                if (/* Handler found in 'SafeSEH' table */)
                    return TRUE;
                else
                    goto InvalidHandler;
            }
        return TRUE;
    } else {
        if (/* 'ExecuteDispatchEnable' and 'ImageDispatchEnable' flags are
            return TRUE;
        if (/* Handler is in non-executable area of the memory */) {
            if (ExecuteDispatchEnable) return TRUE;
        }
        else if (ImageDispatchEnable) return TRUE;
    }
    InvalidHandler:
        RtlInvalidHandlerDetected(...);
        return FALSE;
}

```

The above pseudocode has a modified procedure for checking the conditions (in the original version, some conditions are checked twice, some are checked in nested functions). After analyzing the pseudocode, you can see that, for a successful validation, it is necessary to fulfill one set of conditions where the handler belongs:

- Image with no 'SafeSEH', no 'NO_SEH' flag, no 'ILonly' flag;
- Image with SafeSEH, no 'NO_SEH' flag, no 'ILonly' flag, the image must be registered in 'LdrpInvertedFunctionTable' (not required if the exception occurred at the time of process initialization);
- Non-executable area of memory, 'ExecuteDispatchEnable (ExecuteOptions)' flag must be enabled (will work only with disabled 'No Execute' for the process);
- Executable area of the memory, 'ImageDispatchEnable' flag must be enabled.

In this case, the area of memory is considered as an image, if the region attributes have a 'MEM_IMAGE' flag enabled for it (you can get the attributes by using 'NtQueryVirtualMemory'), and the content corresponds to PE structure. You can get the process flags by using 'NtQueryInformationProces' from 'KPROCESS.KEXECUTE_OPTIONS'. Based on the obtained information, we can suggest at least three ways to implement the support for exceptions in dynamically loaded code on x86 platform:

1. Enabling/substituting 'ImageDispatchEnable' flag for the process.
2. Substituting the memory region type for 'MEM_IMAGE' (for PE image with no SafeSEH).
3. Implementing your own exception dispatcher to bypass all checks.

Now, let’s examine in detail each of these options. Separately, it is worth mentioning the support for SafeSEH that you may need when writing, for example, a usual legal PE packer or protector. To implement it, you will need to think about adding manually a mapped image record (with a pointer to SafeSEH) to global table ‘ntdll!LdrpInvertedFunctionTable’. In this case, the functions that work directly with this table are not exported by ‘ntdll.dll’ and it doesn’t make much sense to search for them manually, because, in the older OS, they still require a pointer to the table. After somehow finding the pointer, y should also think about blocking the access to the table in order to safely make changes. An alternative option may be to unpack the file into a section of the unpacker and transfer ‘SafeSEH’ table from the unpacked file into the main image. Unfortunately, a detailed discussion of these and other techniques is beyond the scope of this article, as here we review the options that do not imply the support for Safe (by the way, you can always simply set the values in this table to zero).

Substituting ‘ExecuteOptions’ of the Process

ExecuteOptions (‘KEXECUTE_OPTIONS’) is a part of kernel structure ‘KPROCESS’, which contains the DEP settings for the process. The structure has the following form:

```
    typedef struct _KEXECUTE_OPTIONS {
        UCHAR ExecuteDisable : 1;
        UCHAR ExecuteEnable : 1;
        UCHAR DisableThunkEmulation : 1;
        UCHAR Permanent : 1;
        UCHAR ExecuteDispatchEnable : 1;
        UCHAR ImageDispatchEnable : 1;
        UCHAR Spare : 2;
    } KEXECUTE_OPTIONS, PKEXECUTE_OPTIONS;

```

At the user level, you can obtain the values of these settings (flags) by using ‘NtQueryInformationProcess’ with the information class parameter equal to 0x22 (ProcessExecuteFlags). The flags are set similarly by using ‘NtSetInformationProcess’. Starting with Vista SP1, the processes with enabled DEP, by default, have enabled ‘Permanent’ flag that bans any change to the settings after initializing the process. This is confirmed by the fragment of ‘KeSetExecuteOptions’ procedure called in kernel mode from ‘NtSetInformationProcess’:

```
@PermanentCheck:        ; KeSetExecuteOptions +2Fh
mov     al, [edi+6Ch]   ; current KEXECUTE_OPTIONS
mov     byte ptr [ebp+arg_0+3], al
test    al, 8           ; test Permanent
jnz     short @Fail     ; returns 0C0000022h (STATUS_ACCESS_DENIED)

```

Therefore, while in the user mode, you cannot modify ‘ExecuteOptions’ with enabled DEP. But you still have an option just to “cheat” ‘RtlIsValidHandler’ by setting a hook on ‘NtQueryInformationProcess’, where the flags will be substituted by those that you need. Setting up such interception would make workable the exceptions in the code outside the modules loaded by the system. Here’s an example of interceptor code:

```

CPP
NTSTATUS __stdcall xNtQueryInformationProcess(HANDLE ProcessHandle, INT ProcessI
{
NTSTATUS Status = org_NtQueryInformationProcess(ProcessHandle, ProcessInformatio
```

```
    if (!Status &amp;&amp; ProcessInformationClass == 0x22) /* ProcessExecuteFla
        *(PDWORD)ProcessInformation |= 0x20; /* ImageDispatchEnable */
    return Status;
}

```

## Substituting Memory Attributes

An alternative to substituting the flags of the process is substituting the att

``` cpp
    NTSTATUS NTAPI xNtQueryVirtualMemory(HANDLE ProcessHandle, PVOID BaseAddres
    {
        NTSTATUS Status = org_NtQueryVirtualMemory(ProcessHandle, BaseAddress,
        if (!Status &amp;&amp; !MemoryInformationClass) /* MemoryBasicInformati
        {
            if((UINT_PTR)MemInformation-&gt;AllocationBase == g_ImageBase) MemI
        }
        return Status;
    }

```

This method for exceptions is suitable when injecting the entire PE image or for manually mapped images. Moreover, this option is somewhat more preferable than the previous one, because it does not diminish the security of the process by partially disabling DEP (you do not need more malware, right?). As a bonus, this method allows you to pass an internal check of the handler in modern versions of CRT when using 'try-except' and 'try-finally' structures (these structures can be used without CRT, for more details, see the relevant box). In CRT, the check is performed by ' __ValidateEH3RN' called from '_except_handler3', it assumes that the type 'MEM_IMAGE' is set for the region and correct PE structure.

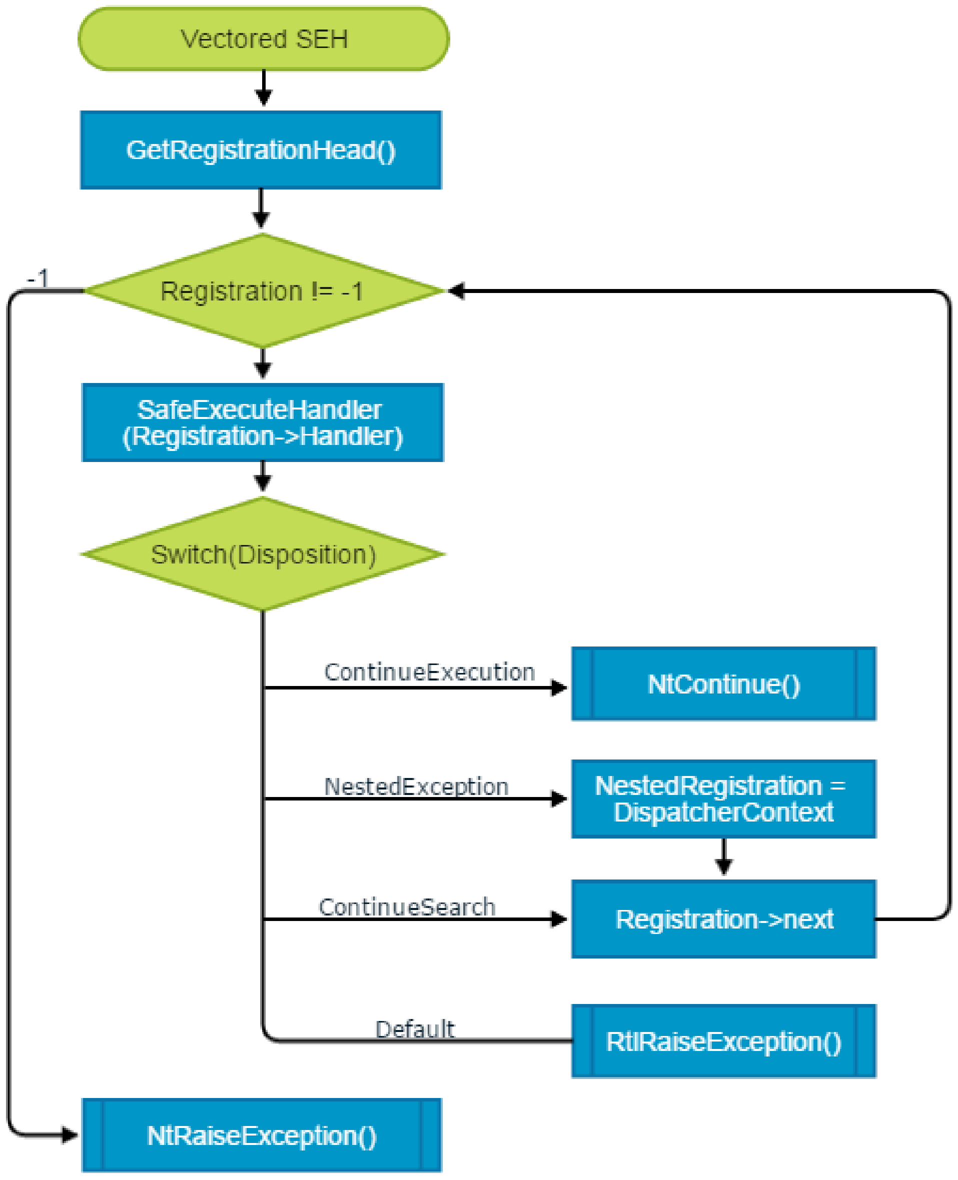
Using Your Own Exception Dispatcher

If the options that involve setting up a hook are not suitable for any reason or you simply do not like them, you can go even further and fully replace SEH dispatching by your own code after implementing all required SEH dispatcher logic inside the vector handler. The above pseudocode of 'RtlDispatchException' shows that VEH is called before the system starts to process SEH chain. Nothing prevents you from seizing control over the exception by using the vector handler and then deciding on your own what to do with it and what handlers to call. You can set VEH handler just with one line of code:

```
    AddVectoredExceptionHandler(0, (PVECTORED_EXCEPTION_HANDLER) &amp;VectoredSE

```

where ‘VectoredSEH’ is a handler which, in fact, is SEH dispatcher. The complete chain of calls for this handler will look like this: `KiUserExceptionDispatcher` -> `RtlDispatchException` -> `RtlpCallVectoredHandlers` -> `VectoredSEH`. In this case, you may not return the control of the calling function, and to call on your own ‘NtContinue’ or ‘NtRaiseException’ depending on the success of dispatching. For complete source code of implementing SEH through VEH, see the materials accompanying this article or go to [GitHub] (https://github.com/Teq2/SEH-Over-VEH). The implementation code is fully workable, and the dispatching logic corresponds to the system logic.



(https://web.archive.org/web/20190712173953/https://hackmag.com/wp-content/uploads/2015/04/05_vectored_seh.png)

SEH dispatcher inside the vector handler

x64 and IA64

64-bit versions of Windows for x64 and Itanium-based platforms use a completely different method to handle exceptions than in x86 versions. This method is based on tables that contain all information necessary for exception dispatching, including the offsets for the beginning and end of a code block, for which the exception is handled. Therefore, the code compiled for these platforms has no operations to set and remove the handler for each ‘try-except’ block. The static table of exceptions is in the ‘Exception Directory’ of PE file and represents an array of elements from structures of ‘RUNTIME_FUNCTION’ that look as follows:

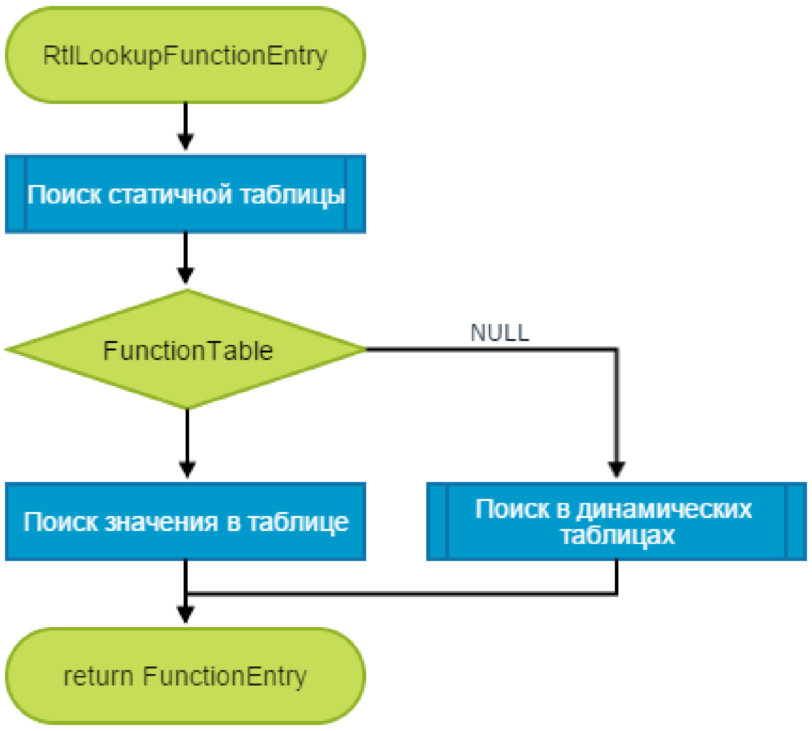
```
typedef struct _RUNTIME_FUNCTION {
        ULONG BeginAddress;
        ULONG EndAddress;
        ULONG UnwindData;
 } RUNTIME_FUNCTION, *PRUNTIME_FUNCTION;

```

It is nice to note that, at the system level, there is support provided for dynamic code exceptions. If the code is in an area of memory which is not an image or this image has no exception table generated by the compiler, the information for exception handling is taken from dynamic tables of exceptions (DynamicFunctionTable). A pointer to the list is stored in ‘ tdll!RtlpDynamicFunctionTable’ and several functions for working with this list are exported from ‘ntdll.dll’. A cursory review of the listings for these functions allowed to obtain the following structure of list items in ‘DynamicFunctionTable’:

```
    struct _DynamicFunctionTable {
        /* +0h */
        PVOID   Next;
        PVOID   Prev;           // First item points to itself
        /* +10h */
        PRUNTIME_FUNCTION Table;// Pointer to the table, for a callback, the fi
        PVOID   TimeCookie;     // ZwQuerySystemTime
        /* +20h */
        PVOID   RegionStart;    // Offset relative to BaseAddress
        DWORD   RegionLength;   // Area covered by the table (callback)
        /* +30h */
        DWORD64 BaseAddress;
        PGET_RUNTIME_FUNCTION_CALLBACK Callback;
        /* +40h */
        PVOID   Context;        // User argument for callback
        DWORD64 CallbackDll;    // Points to '+58h', if 'DLL' is specified
        /* +50h */
        DWORD   Type;           // 1 - table, 2 - callback
        DWORD   EntryCount;
        WCHAR   DllName[1];
    };

```



F
(
T
(
R
(https://

(https://web.archive.org/web/20190712173953/https://hackmag.com/wp-content/uploads/2015/04/06_lookup.png)

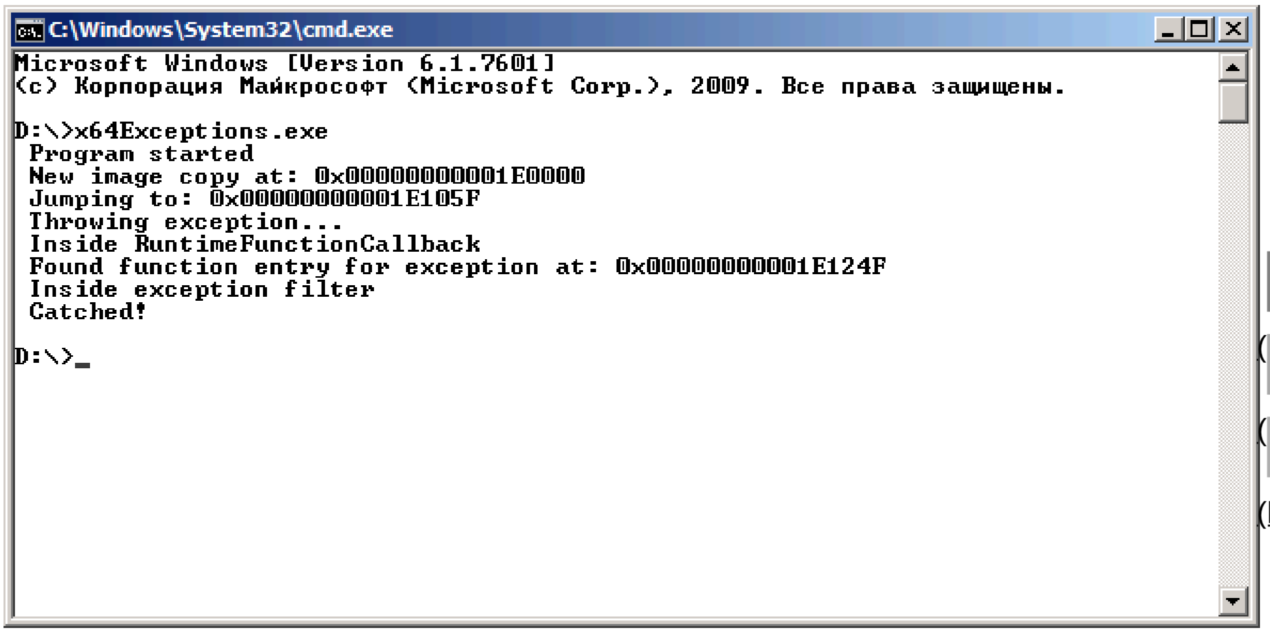
RUNTIME_FUNCTION’ lookup algorithm

The items are added by ‘RtlAddFunctionTable’ and ‘RtlInstallFunctionTableCallback’, and deleted by ‘RtlDeleteFunctionTable’. All these functions are well documented in MSDN and are very easy to use. Here’s an example of adding a dynamic table for the image that has been just displayed manually:

```
ULONG Size, Length;
/* This provides a table generated by compiler for displayed image */
PRUNTIME_FUNCTION Table = (PRUNTIME_FUNCTION) RtlImageDirectoryEntryToData(NewIm
Length = Size/sizeof(PRUNTIME_FUNCTION);
/* Add the image table to the list of 'DynamicFunctionTable' */
RtlAddFunctionTable(Table, Length, (UINT_PTR)NewImage);

```

That’s all, there are no hooks or your own exception dispatchers, and there is no need to bypass the system checks. I would only like to note that ‘DynamicFunctionTable’ is global for the process. So, if the code for which an entry has been added did its job and has to be removed, it is better also to remove the corresponding entry from the table. Instead of adding a table, you can set a callback for specific range of addresses in AP that will receive control each time when the ‘RUNTIME_FUNCTION’ entry is needed for the code from that area. For a version with callback, see the source codes accompanying this article.



```
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

D:\>x64Exceptions.exe
 Program started
 New image copy at: 0x00000000001E0000
 Jumping to: 0x00000000001E105F
 Throwing exception...
 Inside RuntimeFunctionCallback
 Found function entry for exception at: 0x00000000001E124F
 Inside exception filter
 Catched!

D:\>_
```

(https://web.archive.org/web/20190712173953/https://hackmag.com/wp-content/uploads/2015/04/07_handled.png)

Exception is handled

__finally

The low-level programming under Windows that involves the use of native API does not impose exceptions as a method for handling errors, and the developers of "special software" often either simply neglect them or limit themselves just to setting an unhandled exception filter or to simply using VEH. Nevertheless, the exceptions remain a powerful tool that enables you to extract more gain from a more complex architecture of your program. The methods described in this article will allow you to use exceptions even in the most unusual circumstances.

2 Responses to "Dive into exceptions: caution, this may be hard"

1. ***Cmv*** March 1, 2018 (https://web.archive.org/web/20190712173953/https://hackmag.com/uncategorized/exceptions-for-hardcore-users/#comment-8122)

   The logic of your VEHtoSEH is good but it won't work because SAFESEH will not add a new exception handler at runtime.

   Your SafeExecuteHandler will fail on line mov dword ptr fs:[0], esp.

   Infact, even the compiler will warn you at build time. Didn't you see the warnings when compiling the code?

Reply (https://web.archive.org/web/20190712173953/https://hackmag.com/uncategorized/exceptions-for-hardcore-users/?replytocom=8122#respond)

- ***Vx00001*** April 8, 2019 (https://web.archive.org/web/20190712173953/https://hackmag.com/uncategorized/exception for-hardcore-users/#comment-36356)

  Using RtlAddFunctionTable makes possible to handle exceptions based on __try/__except but what about CRT exceptions, is there any solution for try/catch blocks?

  Reply (https://web.archive.org/web/20190712173953/https://hackmag.com/uncategorized/exceptions-for-hardcore-users/?replytocom=36356#respond)

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

Comment

XHTML: You can use these tags: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Submit Comment

Search