limbioliong

limbioliong.wordpress.com/2022/01/09/understanding-windows-structured-exception-handling-part-1

January 9, 2022

//

you're reading...

C++, SEHException

Understanding Windows Structured Exception Handling Part 1 – The Basics

Posted by Lim Bio Liong · January 9, 2022 · 1 Comment
Filed Under ABI, Buffer Overflow, EXCEPTION_ROUTINE, RaiseException(), SEH, Thread Information Block, TIB, UnhandledExceptionFilter, _CONTEXT, _EXCEPTION_REGISTRATION_RECORD

Introduction

1. Just what is meant by an *exception* ? The following definition from MSDN explains it well :

"An exception is an event that occurs during the execution of a program, and requires the execution of code outside the normal flow of control. There are two kinds of exceptions: hardware exceptions and software exceptions. Hardware exceptions are initiated by the CPU. They can result from the execution of certain instruction sequences, such as division by zero or an attempt to access an invalid memory address. Software exceptions are initiated explicitly by applications or the operating system. For example, the system can detect when an invalid parameter value is specified."

2. Because it requires code "outside the normal flow of control" to be executed, the *event* which is associated with the exception cannot be considered something normal. It is usually connected with some erroneous situation.

3. Exception Handling is the means by which this execution of code "outside the normal flow of control" can be carried out.

4. However, there seems to be a whole host of potentially bewildering information on Exception Handling, e.g. C++ Exception Handling, Structured Exception Handling (SEH), Visual C++ language extensions for its implementation of SEH, non-compatibility between C/C++ Exception Handling and SEH, non-support for SEH in some compilers.

5. In this series of articles, I aim to clarify as much as possible these issues. My focus is on SEH and I aim to provide some useful code to demonstrate how it works internally.

6. I assume that the Reader is sufficiently familiar with SEH. If not, I recommend starting with the MSDN documentation.

7. Although not intended for debugging, SEH nevertheless offers great value in debugging as we shall see.

8. I have two highly recommended references which are :

These valuable references greatly helped my personal research. The article by Matt Pietrek, although over 20 years old, still contains useful basic information on SEH and its internals. However, some of the provided source codes will not work correctly without some tweaking. This is because the code has been designed for Windows NT 4.0 and times have changed since then.

9. The full source codes of this article can be found in GitHub.

What is Structured Exception Handling ?

1. SEH can be described as a generalized error handling mechanism supported by the Windows OS. It is an Operating System feature and not tied to any programming language. It forms part of the Windows Application Binary Interface (ABI) so it's a contract between an application and the Windows OS.

2. Is it C++ Exception Handling by another name ? No, it is not. C++ Exception Handling is an ISO standard language construct which ensures C++ code portability across all C++ compilers.

3. C++ Exception Handling is not compatible with SEH. However, the Visual C++ compiler uses SEH to implement its C++ Exception Handling code generation.

4. Can SEH be used in other languages like C#. To answer this, remember that SEH is an OS feature and is not specifically tied to a language or a compiler. While the C# language does not have any language constructs for SEH, the Visual C# Compiler likely also uses SEH for its underlying implementation of the C# Exception Handling.

5. The Visual C++ Compiler has provided language extensions in the form of __try/__except/__finally blocks for direct in-built C++ language support for SEH. However, note that these are Microsoft C++ language extensions and may not be supported in other C++ compilers.

6. Hence use of __try/__except/__finally blocks may render your C++ code non-portable.

7. In subsequent parts of this series of articles, I hope to provide more clarifications on SEH.

The Basics

1. Let's study some C/C++ code that uses SEH.

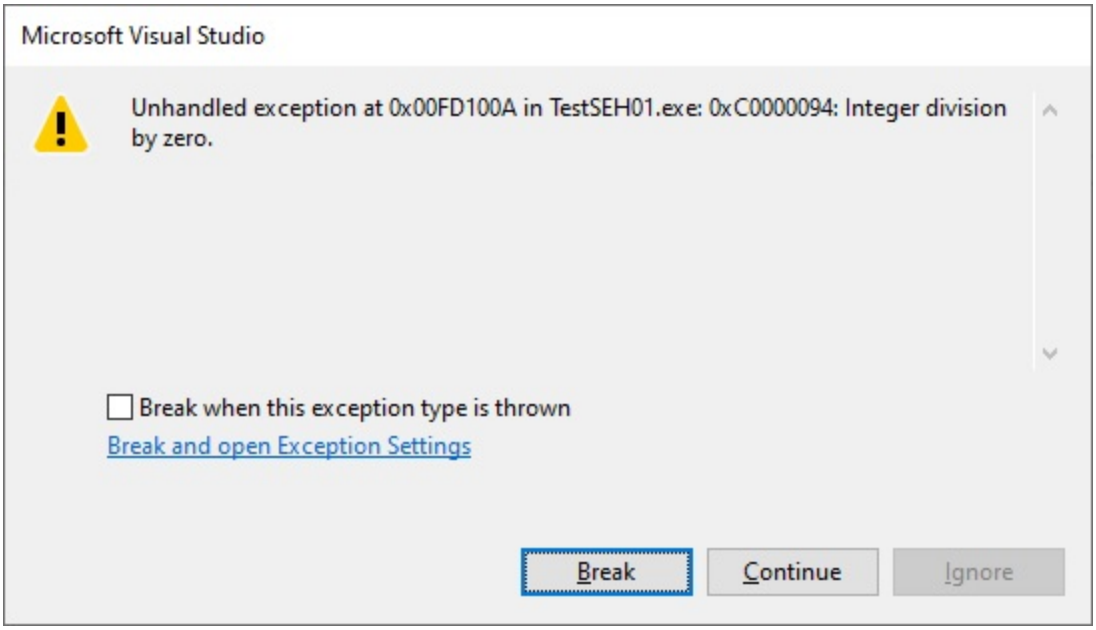
2. Let's say we have the following C/C++ code :

```
int g_iDividend = 1000;
int g_iDivisor = 0;

int TestDivisionByZero()
{
    int iValue = g_iDividend / g_iDivisor;

    return iValue;
}
```

3. When executed, the division will give rise to a "Divide by Zero" exception which will cause the following message box to appear :



4. Let's say we want to handle the exception and wish to recover from it : whenever we have a Division by Zero Exception, we want to change the divisor to value 1.

5. Here, we have defined a new function TestDivisionByZeroTryExcept() which performs the same division operation as TestDivisionByZero() but with error recovery in the form of __try/__except blocks :

```
int TestDivisionByZeroFilter()
{
    g_iDivisor = 1;

    return EXCEPTION_CONTINUE_EXECUTION;
}

int TestDivisionByZeroTryExcept()
{
    int iValue = 0;

    __try
    {
        iValue = g_iDividend / g_iDivisor;
    }
    __except(TestDivisionByZeroFilter())
    {
    }

    return iValue;
}
```

- The division code is embedded inside a __try block which makes it part of a guarded section of code.
- When the division code is executed, a Division by Zero Exception will be thrown as expected.
- Now, since the division code is guarded, the Exception will be caught by the __except block.
- The __except block requires something known as an Exception Filter.
- In our sample code, the Exception Filter comes in the form of the TestDivisionByZeroFilter() function.
- The job of an Exception Filter is to instruct the SEH engine on the action to perform following the Exception.
- It is to return one of the following values :
    - EXCEPTION_EXECUTE_HANDLER
    - EXCEPTION_CONTINUE_SEARCH
    - EXCEPTION_CONTINUE_EXECUTION
- A return value of EXCEPTION_EXECUTE_HANDLER signals to SEH to execute the code in the __except block.
- A return value of EXCEPTION_CONTINUE_SEARCH signals to SEH to continue to search for a handler elsewhere.
- A return value of EXCEPTION_CONTINUE_EXECUTION signals to SEH to re-execute the same code that resulted in the exception (in our case, this is the division code).
- TestDivisionByZeroFilter() returns EXCEPTION_CONTINUE_EXECUTION. But before that, it performs the recovery process by changing the divisor g_iDivisor to 1.
- Hence, when the division code is re-run, it succeeds and iValue is set to value 1000.

6. Now observe the following code :

```
int g_iDividend = 1000;
int g_iDivisor = 0;

EXCEPTION_DISPOSITION NTAPI _Function_class_(EXCEPTION_ROUTINE)
MyDivisionByZero01ExceptionRoutine
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    DISPLAY_EXCEPTION_INFO(pExceptionRecord)

    g_iDivisor = 1;

    return ExceptionContinueExecution;
}

int TestDivisionByZero01SEH()
{
    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();

    EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.Handler = (PEXCEPTION_ROUTINE)(&MyDivisionByZero01ExceptionRoutine);
    Registration.Next = TIB->ExceptionList;
    TIB->ExceptionList = &Registration;

    int iValue = g_iDividend / g_iDivisor;

    TIB->ExceptionList = TIB->ExceptionList->Next;

    return iValue;
}
```

7. In the above code, we have defined a new function TestDivisionByZero01SEH() which is functionally equivalent to TestDivisionByZeroTryExcept() but expressed in a form that is rather unfamiliar to most C++ developers.

8. TestDivisionByZero01SEH() contains low-level code that will put in place a SEH Exception Handling mechanism just before a set of guarded code (the division operation) is run. The Exception Handler is MyDivisionByZero01ExceptionRoutine().

9. After the SEH mechanism has been put in place, the division code executes and the Division by Zero Exception will happen. Control is then passed onto MyDivisionByZero01ExceptionRoutine().

10. MyDivisionByZero01ExceptionRoutine() performs the error recovery by setting g_iDivisor to 1 and then returning ExceptionContinueExecution. This signals to the SEH system to re-do the division code in TestDivisionByZero01SEH() albeit with the divisor now modified to value 1. The division goes through successfully this time.

11. MyDivisionByZero01ExceptionRoutine() is thus similar to the TestDivisionByZeroFilter() Exception Filter function that we met earlier. As mentioned earlier, the TestDivisionByZero01SEH() is functionally equivalent to TestDivisionByZeroTryExcept().

12. To fully understand what went on, we must note that SEH works on a per-thread basis. In other words, each thread contains its own Structured Exception Handling Mechanism. In order to understand this better, we need to learn something known as a Thread Information Block (TIB).

13. Every thread running in the OS is associated with a TIB. The TIB is a data structure in which information on a thread is stored. This includes information on the structured exception handlers for the current thread. Note that I mentioned "structured exception *handlers*" (plural) as there are actually more than one exception handlers defined per thread albeit only one of which will be used when an exception occurs. We shall see this later.

The Thread Information Block (TIB)

1. The TIB is a complex structure and we shall not be discussing it in detail in this article. We shall instead focus on the part of the TIB which are related to SEH.

2. The following structure definition of the TIB is taken from winnt.h :

```
typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
#if defined(_MSC_EXTENSIONS)
    union {
        PVOID FiberData;
        DWORD Version;
    };
#else
    PVOID FiberData;
#endif
    PVOID ArbitraryUserPointer;
    struct _NT_TIB *Self;
} NT_TIB;
typedef NT_TIB *PNT_TIB;
```

3. As clearly displayed above, the first member of the TIB is a pointer to a _EXCEPTION_REGISTRATION_RECORD structure. This is the Exception Handler List of the current thread :

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD;
```

4. Each _EXCEPTION_REGISTRATION_RECORD structure holds a pointer to the next structure. Hence a _EXCEPTION_REGISTRATION_RECORD structure is a linked list of Exception Handler routines. Recall I mentioned in point 13 of section “The Basics” that there is more than one handler when an exception occurs.

5. The Exception Handler is a user-defined function with the following signature :

```
EXCEPTION_DISPOSITION
NTAPI
EXCEPTION_ROUTINE (
    _Inout_ struct _EXCEPTION_RECORD *ExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT *ContextRecord,
    _In_ PVOID DispatcherContext
    );
```

6. When an exception occurs, the OS looks at the TIB of the faulting thread and retrieves a pointer to an _EXCEPTION_REGISTRATION_RECORD structure contained in the TIB. The OS then navigates through the linked list of EXCEPTION_ROUTINE routines and determines if any of these routines will handle the exception.

7. Each handler indicates its interest in handling the exception via its return value which is of type EXCEPTION_DISPOSITION :

```
// Exception disposition return values
typedef enum _EXCEPTION_DISPOSITION
{
    ExceptionContinueExecution,
    ExceptionContinueSearch,
    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

8. For the purposes of the basic demo in this blog, we will only consider the return values ExceptionContinueSearch and ExceptionContinueExecution. The other two are used in more advanced situations.

9. When a handler returns ExceptionContinueSearch, it means that the handler is not interested in doing any recovery for the current exception. The OS will therefore move on to the next _EXCEPTION_REGISTRATION_RECORD and execute its handler.

10. If all handlers will not manage the current exception, the OS will call the current Unhandled Exception Filter that has been put in place. Developers may call the UnhandledExceptionFilter() API install a custom filter. However, note that the Unhandled Exception Filter() is not part of the Linked List of Exception Handlers mentioned previously. That is, there is no _EXCEPTION_REGISTRATION_RECORD structure that will point to this function.

11. The default Unhandled Exception Filter is __scrt_unhandled_exception_filter() and is put in place as the default handler by the C/C++ Runtime Library code at the start of the application. See the __scrt_set_unhandled_exception_filter() function in the ..\crt\src\vcruntime\utility_desktop.cpp source code file (for Visual Studio 2019).

Back to the Basic Example

1. We were examining the TestDivisionByZero01SEH() function. The function aims to demonstrate recovery from a Division by Zero exception :

```
EXCEPTION_DISPOSITION NTAPI _Function_class_(EXCEPTION_ROUTINE)
MyDivisionByZero01ExceptionRoutine
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    DISPLAY_EXCEPTION_INFO(pExceptionRecord)

    g_iDivisor = 1;

    return ExceptionContinueExecution;
}

int TestDivisionByZero01SEH()
{
    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();

    EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.Handler = (PEXCEPTION_ROUTINE)(&MyDivisionByZero01ExceptionRoutine);
    Registration.Next = TIB->ExceptionList;
    TIB->ExceptionList = &Registration;

    int iValue = g_iDividend / g_iDivisor;

    TIB->ExceptionList = TIB->ExceptionList->Next;

    return iValue;
}
```

2. We begin by calling the NtCurrentTeb() function which will return a pointer to a _TEB structure. This _TEB structure is synonymous with a Thread Information Block structure. Hence we can simply cast the return pointer to a pointer to a NT_TIB.

3. We then register MyDivisionByZero01ExceptionRoutine() as a SEH exception handler. This is done by defining a EXCEPTION_REGISTRATION_RECORD structure (Registration) and then setting the MyDivisionByZero01ExceptionRoutine() function as its EXCEPTION_ROUTINE Handler. Registration's Next is set to the original TIB ExceptionList. Thereafter, we set the TIB ExceptionList to point to Registration.

4. Note that this does not mean that MyDivisionByZero01ExceptionRoutine() has higher priority over the other handlers that are below it in the linked list of handlers. It only has higher priority over other lower handlers for the types of exceptions that it will handle.

5. We next examine the MyDivisionByZero01ExceptionRoutine() function. This function is the handler that will get called first in line when an exception occurs. We have put this handler in place in anticipation of a division by zero exception. Hence when

TestDivisionByZero01SEH() is run again, this handler will be called when g_iDividend is divided by g_iDivisor (being of value zero).

6. What does MyDivisionByZero01ExceptionRoutine() do ? First it displays some useful information about the current exception via the DISPLAY_EXCEPTION_INFO() macro. Then it simply changes the value of g_iDivisor to 1 and then returns ExceptionContinueExecution.

7. The return value of ExceptionContinueExecution will cause control to return to the exact location where the exception occurred, i.e. at the following line in TestDivisionByZero01SEH() :

```
int iValue = g_iDividend / g_iDivisor;
```

This time, g_iDivisor has been changed to value 1 and so the division goes through successfully and iValue is set to value 1000.

8. But that's not all, note the line that follows :

```
TIB->ExceptionList = TIB->ExceptionList->Next;
```

This line removes MyDivisionByZero01ExceptionRoutine() as a SEH handler. This is a necessary step to perform because our EXCEPTION_REGISTRATION_RECORD structure (Registration) is defined on the stack of the TestDivisionByZero01SEH() function and so when TestDivisionByZero01SEH() exits, it will no longer be available in memory which will leave the TIB's ExceptionList pointing to a non-existent structure.

9. Well, why don't we simply create a EXCEPTION_REGISTRATION_RECORD globally then ? It will not work. Windows will not call the custom exception handler. I have included an example code to demonstrate this issue : TestSEHGlobalExRegRec().

10. This function takes in a pointer to a EXCEPTION_REGISTRATION_RECORD as parameter. if EXCEPTION_REGISTRATION_RECORD is defined globally, the exception handler MyTestExceptionRoutine() will not be called when an exception is raised.

11. We have thus shown how SEH handlers can be implemented and inserted into the flow of a function. The basic steps are really quite simple.

12. In the next 2 sections, I will provide more advanced examples demonstrating creative things that a SEH handler can perform.

Accessing the Arguments of a Faulting Function.

1. In the previous example code, we have seen how a division by zero exception can be corrected by the exception handler.

2. This is easily accomplished due to the fact that g_iDivisor is a global variable. What if the divisor was an argument passed into the function ?

3. This can still be addressed through the use of the _CONTEXT parameter of the Exception Handler function.

4. This is demonstrated by the TestDivisionByZero02SEH() and MyDivisionByZero02ExceptionRoutine() functions which are listed below :

```
EXCEPTION_DISPOSITION NTAPI _Function_class_(EXCEPTION_ROUTINE)
MyDivisionByZero02ExceptionRoutine
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    DISPLAY_EXCEPTION_INFO(pExceptionRecord)

    int* pDividendParam = (int*)((pContextRecord->Ebp) + 8);
    int* pDivisorParam = (int*)((pContextRecord->Ebp) + 12);

    printf("Dividend : [%d]. Divisor : [%d]\r\n",
        *pDividendParam,
        *pDivisorParam);

    *pDivisorParam = 1;

    return ExceptionContinueExecution;
}

int TestDivisionByZero02SEH(int iDividend, int iDivisor)
{
    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();

    EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.Handler = (PEXCEPTION_ROUTINE)(&MyDivisionByZero02ExceptionRoutine);
    Registration.Next = TIB->ExceptionList;
    TIB->ExceptionList = &Registration;

    int iValue = iDividend / iDivisor;

    TIB->ExceptionList = TIB->ExceptionList->Next;

    return iValue;
}
```

5. The third parameter to MyDivisionByZero02ExceptionRoutine() is a pointer to a _CONTEXT structure. This structure contains processor specific data including register values which are relevant at the time of occurrence of the exception.

6. Using this _CONTEXT structure, we can access the value of the EBP register at the time of the exception and from the EBP access the pointers to the arguments of TestDivisionByZero02SEH() iDividend and iDivisor.

7. The EBP is commonly known as the "Base Pointer" of a stack frame. See this article and many others on the web for more information on stack frames and base pointers.

8. We access the arguments of a function by adding offsets to the EBP register. For x86 machines, the first argument to a function is always 8 bytes higher in memory from memory location where EBP points to. See the following diagram :

16(%ebp)	- third function parameter
12(%ebp)	- second function parameter
8(%ebp)	- first function parameter
4(%ebp)	- old %EIP (the function's "return address")
0(%ebp)	- old %EBP (previous function's base pointer)
-4(%ebp)	- first local variable
-8(%ebp)	- second local variable
-12(%ebp)	- third local variable

9. The second parameter is at location 12 bytes higher and so on with each subsequent parameter at 4 bytes boundaries.

10. Hence the following code which is used to access the memory locations of the 2 parameters :

```
int* pDividendParam = (int*)((pContextRecord->Ebp) + 8);
int* pDivisorParam = (int*)((pContextRecord->Ebp) + 12);
```

11. pDivisorParam is a pointer to the iDivisor parameter. Modifying it is thus possible :

```
*pDivisorParam = 1;
```

12. MyDivisionByZero02ExceptionRoutine() then returns ExceptionContinueExecution which allows the division in TestDivisionByZero02SEH() to continue successfully with iDivisor modified to value 1.

13. This demonstrates an effective use of the _CONTEXT structure in a SEH Handler.

Analyzing a Buffer Overflow Situation.

1. Buffer overflows are nasty situations which arise when a local variable gets assigned a value beyond its memory size.

2. The following code snippet illustrates this :

```
char Buffer[8] = { 0 };

strcpy(Buffer, "AAAAAAAAAAAAAAAA");
```

3. In the above code, Buffer is a character buffer of size 8 bytes. The strcpy() call assigns 16 'A' characters to Buffer which causes a buffer overflow. The 'A' characters which could not fit into Buffer will overwrite the memory higher in location above Buffer.

4. This situation is particularly disastrous when it happens to a character buffer declared on a function stack.

5. Referring to the EBP offset diagram shown in the last section "Accessing the Arguments of a Faulting Function", we can see that the return address of a function lies 4 bytes higher in memory to where EBP points to.

6. If the buffer overflow overwrites the return address, a crash will occur when the function ends. This is inevitable as the return address is now overwritten with invalid values (e.g. "AAAA").

7. Normally, a buffer overflow situation is irrecoverable. However, mechanisms have been devised to trap such situations via the use of "safe" functions for string management e.g. strcpy_s()_, strcat_s(), sprintf_s(), etc.

8. In the sample code of this section, I aim to demonstrate a simple way to trap a buffer overflow situation and reporting this via a SEH Exception.

9. This is demonstrated by TestBufferOverflowSEH() and MyBufferOverflowExceptionRoutine() :

```
EXCEPTION_DISPOSITION NTAPI _Function_class_(EXCEPTION_ROUTINE)
MyBufferOverflowExceptionRoutine
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    DISPLAY_EXCEPTION_INFO(pExceptionRecord)

    void* pBufferAddress = (void*)(pExceptionRecord->ExceptionInformation[0]);
    size_t  stSize = pExceptionRecord->ExceptionInformation[1];

    printf("Overflowed Buffer : [0x%p]. Size : [%d]\r\n",
        pBufferAddress,
        stSize);

    char* pBuffer = new char[stSize + 1];
    memset(pBuffer, 0, stSize + 1);
    memcpy(pBuffer, pBufferAddress, stSize);

    printf("Overflowed Buffer Contents : [%s]\r\n", pBuffer);

    free(pBuffer);
    pBuffer = NULL;

    printf("iStackGuard Value : [0x%08X].\r\n",
        pExceptionRecord->ExceptionInformation[2]);

    int* pReturnAddress = (int*)(pExceptionRecord->ExceptionInformation[3]);

    printf("Return Address    : [0x%08X].\r\n", *pReturnAddress);

    return ExceptionContinueSearch;
}

void TestBufferOverflowSEH(const char* pString)
{
    int  iStackGuard = 0;
    char szCopy[8];

    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();
    EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.Handler = (PEXCEPTION_ROUTINE)(&MyBufferOverflowExceptionRoutine);
    Registration.Next = TIB->ExceptionList;
    TIB->ExceptionList = &Registration;

    strcpy(szCopy, pString);

    if (iStackGuard != 0)
    {
```

```
        ULONG_PTR args[4] = { 0 };
        args[0] = (ULONG_PTR)(szCopy);
        args[1] = 8;
        args[2] = iStackGuard;
        args[3] = (ULONG_PTR)(&iStackGuard + 8);

        RaiseException(EXCEPTION_BUFFER_OVERFLOW, EXCEPTION_NONCONTINUABLE, 4, args);
    }

    TIB->ExceptionList = TIB->ExceptionList->Next;
}
```

10. In TestBufferOverflowSEH(), we define a local string buffer szCopy of size 8 bytes. Just above szCopy is a local integer iStackGuard which is used to detect whether szCopy has buffer overflowed.

11. The principle is simple : because iStackGuard is defined before szCopy, it will be located on the stack next to szCopy and higher in memory. If szCopy buffer overflows, iStackGuard will change in value (from 0 to some unpredictable value).

12. After strcpy() is called, we check the value of iStackGuard. If it is not 0, a buffer overflow has occurred. We will call RaiseException() with a custom Exception Code (EXCEPTION_BUFFER_OVERFLOW defined as 0xE0000001) and information on the buffer overflow situation. The extra information are : a pointer to the szCopy buffer, its size, the value of iStackGuard and the current value of the return address (which may now have been corrupted).

13. The information is passed via the 3rd and 4th parameters of the RaiseException() API. These parameters will be passed to the Exception Handler through the ExceptionInformation array which is contained in the _EXCEPTION_RECORD parameter.

14. When MyBufferOverflowExceptionRoutine() is invoked by the SEH engine, these information is displayed on the console. The following is the output using out sample code :

```
An excepton occured at address : [0x764BB522]. Exception Code : [0xE0000001]. Exception Flags : [0x00000001]
 EXCEPTION_NONCONTINUABLE
Overflowed Buffer : [0x009BF924]. Size : [8]
Overflowed Buffer Contents : [AAAAAAAA]
iStackGuard Value : [0x41414141].
Return Address    : [0x41414141].
```

15. As can be observed, the value of iStackGuard has been changed to 0x41414141 (0x41 being the ASCII value of the character ‘A’) and the Return Address has also been overwritten to 0x41414141.

16. This sample code demonstrates both a Custom Exception Code which can be defined and raised together with useful information.

Summary

1. This is the first part of a series of articles which studies the internals of Structured Exception Handling.

2. In this article, we touched on how a Custom SEH Handler can be setup and used effectively.

3. I hope the samples provided here have inspired the Reader with creative ideas.

4. In the next part, we will dive deeper into SEH and make more observations of its internals.

Notes on Usage of the Sample Code

1. The sample source code file has been compiled on Visual Studio 2019.

2. I assume that the Reader will be using a version of Visual Studio.

3. To ensure simplicity and to focus on SEH, I have set the following compiler settings for the TestSEH01 project :

3.1 The project is a **Win32** console program.

3.2 Optimization is turned off. This will ensure that function proglogs and epilogs (i.e. the use of the ebp register as a base pointer) are put in place by the compiler. This is especially important for the TestDivisionByZero02SEH() and MyDivisionByZero02ExceptionRoutine() sample code.

3.3 SAFESEH Setting. The following has been set :

> Linker | Advanced Properties : set Image Has Safe Exception Handlers to : No (/SAFESEH:NO)

It is important to turn off the SAFESEH setting otherwise the compiled code may not run correctly.

3.4 _CRT_SECURE_NO_WARNINGS Setting.

> C/C++ | Preprocessor Properties | Preprocessor Definitions settings : add _CRT_SECURE_NO_WARNINGS.

This is necessary because the sample code uses strcpy() (a non-safe character buffer management function) to demonstrate Buffer Overflow.

4. The full source codes for this article can be found in GitHub.

References

Structured Exception Handling

What Is an Exception?

Fun with low level SEH

A Crash Course on the Depths of Win32™ Structured Exception Handling by Matt Pietrek

CppCon 2018: James McNellis “Unwinding the Stack: Exploring How C++ Exceptions Work on Windows”

Exception Handler not called in C

About Lim Bio Liong

I've been in software development for nearly 20 years specializing in C , COM and C#. It's truly an exicting time we live in, with so much resources at our disposal to gain and share knowledge. I hope my blog will serve a small part in this global knowledge sharing network. For many years now I've been deeply involved with C development work. However since circa 2010, my current work has required me to use more and more on C# with a particular focus on COM interop. I've also written several articles for CodeProject. However, in recent years I've concentrated my time more on helping others in the MSDN forums. Please feel free to leave a comment whenever you have any constructive criticism over any of my blog posts.
View all posts by Lim Bio Liong »