Windows x86 SEH 机制暨MoeCTF2023 Unwind 解题报告

前言

去年 MoeCTF 2022 的时候云姐姐出了个除零异常的题，当时在机房几个人七嘴八舌怼汇编还能勉强做出来....

今年 MoeCTF 2023 的时候云姐姐又双出了这么一个题，我做的时候就比较坐牢了..... 想了想还是从原理出发完全地了解一下这个知识点，再做做这么个题。

~~高难警告~~

SEH 学习

认识 SEH

> 使用 SEH，你可以确保在执行意外终止时，可以正确地释放资源（如内存块和文件）。 ——《Structured Exception Handling》

所谓 SEH，全称即 "Structure Exception Handler" —— 结构化异常处理器。

需要明确：SEH 是针对于**异常**的一种**处理机制**。首先，异常分为两方面来讲：硬件异常和软件异常。

- 硬件异常：这里的“硬件”的定义十分狭隘 —— 仅限于 CPU 异常，例如除零异常，它就是 CPU 在执行除零操作时会自动触发的异常处理机制；
- 软件异常：由程序模拟的异常，软件异常可以既可以由操作系统触发，也可以由程序员随意触发。

SEH 并非专为 C/C++ 设计，它是 windows 给的一套通用性的解决方案，因此尽管我们可以在代码中随时使用它们，但是应当优先使用 c++ ISO 标准化的异常处理（try - catch）~~（也就是说不要把这东西往工程代码里写啊喂）~~

SEH 也有两种处理机制：

- 异常处理程序 __except 块：它基于 `filter-expression` 值响应或消除异常；
- 终止处理程序 __finally 块：无论异常是否终止都要继续调用之。

Windows x86 提供的异常处理机制其实只是一个简单的框架，在此基础上有各编译器提供的增强版异常处理机制。故我们将 windows SEH 机制分为系统实现的原始版本、编译器实现的增强版本 两方面来讲。

约定：简单地将 SEH 理解为：系统提供的异常处理机制，包括编译器对其进行增强的部分。

系统实现版本

系统实现的原生 SEH 只是一个简单的框架，它有一个前提是“**允许任何人触发异常，允许任何人处理异常**”。它的运作机制如下：

- 登记：系统把触发异常时的栈帧信息保存在一个链表里，并且将这个链表保存在线程的数据结构里。也就是说，异常涉及的行为是线程相关的。
- 抛出：线程抛出异常；
- 响应：系统找到抛出异常的线程的异常处理链表；
- 分发、处理异常（异常处理函数必须由用户具体实现）。

最后一步是最核心的系统管理工作，后续会重点讲解。

源代码

先给出使用的数据结构与宏，后续看到不记得的数据结构与宏可以随时回滚鼠标翻阅：

```
#define EXCEPTION_CHAIN_END ((struct _EXCEPTION_REGISTRATION_RECORD *
POINTER_32)-1)
// 标记链表结束的常量


typedef enum _EXCEPTION_DISPOSITION {
        ExceptionContinueExecution,
        ExceptionContinueSearch,
        ExceptionNestedException,
        ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
 // 声明一个关于异常处理的枚举类型

typedef struct _EXCEPTION_RECORD {
        DWORD ExceptionCode; // 赋予异常代码，后文会列举较为常见的一些
        DWORD ExceptionFlags;
        struct _EXCEPTION_RECORD *ExceptionRecord;
        PVOID ExceptionAddress; // 异常发生的地址
```

```
        DWORD NumberParameters;
        ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
 // 这个结构定义在 WINNT.H，结构中的其余项暂不关注

typedef EXCEPTION_RECORD *PEXCEPTION_RECORD;


typedef
EXCEPTION_DISPOSITION
(*PEXCEPTION_ROUTINE) (
        IN struct _EXCEPTION_RECORD *ExceptionRecord,
        IN PVOID EstablisherFrame,
        IN OUT struct _CONTEXT *ContextRecord,
        IN OUT PVOID DispatcherContext
    );


typedef struct _EXCEPTION_REGISTRATION_RECORD {
        struct _EXCEPTION_REGISTRATION_RECORD *Next; // 链表指针
        PEXCEPTION_ROUTINE Handler; // 指向异常处理函数
} EXCEPTION_REGISTRATION_RECORD;
// 异常登记信息链表，链表中的最后一个结点会将 Next 置为 EXCEPTION_CHAIN_END，表示链表
到此结束。
// 线程信息块的第一个DWORD（在基于Intel CPU的机器上是FS:[0]）指向这个链表的头部。


typedef EXCEPTION_REGISTRATION_RECORD *PEXCEPTION_REGISTRATION_RECORD;


typedef struct _CONTEXT {
            DWORD ContextFlags;
            DWORD Dr0;
            DWORD Dr1;
            DWORD Dr2;
            DWORD Dr3;
            DWORD Dr6;
            DWORD Dr7;
            FLOATING_SAVE_AREA FloatSave;
            DWORD SegGs;
            DWORD SegFs;
            DWORD SegEs;
            DWORD SegDs;
            DWORD Edi;
            DWORD Esi;
            DWORD Ebx;
            DWORD Edx;
            DWORD Ecx;
            DWORD Eax;
            DWORD Ebp;
            DWORD Eip;
            DWORD SegCs;
            DWORD EFlags;
            DWORD Esp;
            DWORD SegSs;
 } CONTEXT;
```

使用的关键函数：

```
EXCEPTION_DISPOSITION
__cdecl _except_handler( struct _EXCEPTION_RECORD *ExceptionRecord,
                         void * EstablisherFrame,
                         struct _CONTEXT *ContextRecord,
                         void * DispatcherContext );
```

依照源代码，我们可以更详细地理解系统最后一步的管理工作：

- 系统找到当前异常线程的链表；
- 从链表中的第一个结点开始遍历，找到一个 `EXCEPTION_REGISTRATION_RECORD` 就调用它的 `Handler`；
- 把该异常（由第一个类型为 `EXCEPTION_RECORD` 的参数表示）传递给该 `Handler`，`Handler` 处理并返回一个类型为 `EXCEPTION_DISPOSITION` 的枚举值；
  - 没有处理异常，继续在链表中搜索（`EXCEPTION_CONTINUE_SEARCH`）；
  - 识别异常，但已将其消除（`EXCEPTION_CONTINUE_EXECUTION`）；
  - 在处理异常过程中，调用表达式函数时再次触发异常（`ExceptionNestedException`）；
  - 在展开过程中调用处理异常函数时再次触发异常（`ExceptionCollidedUnwind`）；
- 系统根据不同的返回值来继续遍历异常链表或者回到触发点继续执行。

更进一步

注意：以下的异常处理流程均为内核模式。

~~按理来说用户模式大差不差，但我没看（bushi~~

在 x86 架构的操作系统中，`IDT`（Interrupt Descriptor Table）是一种用于管理中断和异常处理的数据结构。IDT 中的 `KiTrap??` 项实际上是操作系统内核中的中断或异常处理程序的入口点。`KiTrap` 是一种命名约定，后面的 `??` 表示具体的中断或异常号。

硬件异常

我们来看看更具体的内核异常流程。

首先，CPU 执行的指令触发了异常，CPU 改执行 `IDT` 中 `KiTrap??`，`KiTrap??` 会调用 `KiDispatchException`，该函数源代码如下：

```
VOID KiDispatchException (
        IN PEXCEPTION_RECORD ExceptionRecord,
        IN PKEXCEPTION_FRAME ExceptionFrame,
        IN PKTRAP_FRAME TrapFrame,
        IN KPROCESSOR_MODE PreviousMode,
        IN BOOLEAN FirstChance
    );
// 该函数的主要功能就是分派异常。
```

1. 在当前栈中分配一个 `CONTEXT`，调用 `KeContextFromKframes` 初始化它；
2. 检查 `ExceptionRecord->ExceptionCode`；
3. 检查 `PreviousMode`：
    - 如果为 `KernalMode`：
        - 如果 `FirstChance` 为 `TRUE`，那么将该异常传达给内核调试器，如果内核调试器没有处理，那么调用 `RtlDispatchException` 进行处理。
        - 如果 `FirstChance` 为 `FALSE`，那么再次将该异常传达给内核调试器，如果内核调试器没有处理，那么 `BUGCHECK`（程序崩溃）。
    - 如果为 `UserMode`：
        - 如果 `FirstChance` 为 `TRUE`，那么将该异常传达给内核调试器，如果内核调试器没有处理，那么将异常传达给应用层调试器。如果仍然没有处理，那么将 `KTRAP_FRAME` 和 `EXCEPTION_RECORD` 拷贝到 `UserMode` 的栈中，并设置 `KTRAP_FRAME::Eip` 设置为 `ntdll!KiUserExceptionDispatcher`，返回（将该异常交由应用层异常处理程序进行处理）。
        - 如果 `FirstChance` 为 `FALSE`，那么再次将异常传达给应用层调试器，如果仍然没有处理，那么调用 `ZwTerminateProcess` 结束进程，并 `BUGCHECK`。

开应用层异常不说，我们来看 `PreviousMode` 是 `KernelMode` 的情况，其重点是调用 `RtlDispatchException` 的操作。在这里可以看到函数源代码，篇幅所限，**非必要我不会贴上完整的源代码**。

首先是这个函数的简单介绍，通过读注释了解到：这个函数的功能是通过向后搜索 `EXCEPTION_REGISTRATION_RECORD` 来把异常分发给以调用 `EXCEPTION_REGISTRATION_RECORD` 为基础的处理函数，这个搜索开始于上下文初始时指定的 `EXCEPTION_REGISTRATION_RECORD` ，结束于异常被处理、对栈帧的搜索非法或者已经搜索完毕。

定义：

```
BOOLEAN RtlDispatchException (
        IN PEXCEPTION_RECORD ExceptionRecord,
        IN PCONTEXT ContextRecord
    )
```

这个函数的第一个参数是提供一个指向异常记录信息的指针，第二个参数是提供一个指向堆栈信息的指针（相关的数据结构在上文已经给过）；返回值是一个 `bool`，如果异常被处理返回 `True`，否则返回 `False`。

有一个关键函数是 `RtlpExecuteHandlerForException`，可以在 34 行看到它的定义：

```
EXCEPTION_DISPOSITION
RtlpExecuteHandlerForException (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN PVOID EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PVOID DispatcherContext,
    IN PEXCEPTION_ROUTINE ExceptionRoutine
    );
```

根据 `RtlDispatchException` 具体实现可以获知以下关键点：

- 函数一开始会先进行栈空间检测，以免遍历超出栈空间；
- 依次遍历 `EXCEPTION_REGISTRATION_RECORD` 链表，通过 `RtlpExecuteHandlerForException` 调用异常处理函数，再根据后者的返回值做出不同的处理：
  - `ExceptionContinueExecution`：对于 `ExceptionRecord->ExceptionFlags` 为 `EXCEPTION_NONCONTINUABLE` 的异常，调用一个 `RtlRaiseException`； 否则结束遍历，直接返回 True；
  - `ExceptionContinueSearch`：对于 `ExceptionRecord->ExceptionFlags` 为 `EXCEPTION_STACK_INVALID` 的函数，结束遍历，返回 `False`；否则继续遍历；
  - `ExceptionNestedException`：出现了嵌套异常，从指定的新异常继续遍历。
  - 其他的返回值：先调用 `RtlRaiseException` 起一个新异常，然后继续遍历；

总结一下 CPU 触发异常的处理与调用流程：

`CPU test the Exception -> KiTrap?? -> KiDispatchException -> RtlDispatchException -> RtlpExecuteHandlerForException.`

软件异常

软件异常跟硬件异常的处理流程非常接近，只有触发点的不同，调用流程是：

`RtlRaiseException -> RtlDispatchException -> RtlpExecuteHandlerForException`

后面两个被调用的函数咱已经聊过了，主要来看看 `RtlRaiseException`。

我们可以在这里的 43 行看到原型如下：

```
VOID
RtlRaiseException (
    IN PEXCEPTION_RECORD ExceptionRecord
    )
```

同样的，我们先根据注释对这个函数做一个初步了解：这个函数通过构建一个堆栈信息记录（`CONTEXT`）和调用异常分发程序（`RtlDispatcherException` ）来抛出一个软件异常。如果 `RtlDispatcherException` 找到了一个处理程序来处理异常，则调用该处理程序并恢复原程序执行，否则调用 `ZwRaiseException` 系统服务来提供默认处理。注意：i386 下此例程无法捕捉 `non-fp` 异常。

根据具体实现可得知：

- 先调用 `_RtlCaptureContext` 获得调用者的 `CONTEXT`（即包括各种寄存器），注意，在此调用之前，状态标志的状态没有受到干扰；
- 获取 `PExceptionRecord`，并且将异常地址设置为这个函数的返回地址；
- 设置 `CONTEXT` 标志，表明将捕获所有的上下文信息；
- 直接运行一次 `RtlDispatchException`。
    - 如果异常处理，则调用 `_ZwContinue`（它一般不会返回），故原程序继续执行；
    - 如果异常未被处理，则进入异常处理分支，函数再次调用 `_ZwRaiseException`，**尝试引发一个“二次”异常（second chance exception）**，这次传入的异常的 `FirstChance` 被置为 `FALSE`（忘记 `FirstChance` 的可以回滚至硬件异常部分源代码）。
- 如果函数参数列表自身有问题，才有可能跑到这里：它会调用 `_RtlRaiseStatus` 来引发一个非可继续异常，其中参数是前面异常处理的返回值，表示异常无法继续

处理，程序将 BUGCHECK。

简单总结

到这里，系统实现的 SEH 版本基本上已经讲解完毕。整个异常处理过程可以大概总结为：遍历异常链表，挨个调用异常注册信息的处理函数，如果其中有某个处理函数处理了该异常（返回值为 `ExceptionContinueExecution`），那么就从异常触发点（如果是断点异常，则要回退一个字节的指令（`int 3` 指令本身））重新执行。否则不管是整个链表中没有找到合适的处理函数（返回值为 `ExceptionContinueSearch` ），或者遍历过程中出现问题（返回值为 `ExceptionNestedException`），系统都会简单粗暴的 BUGCHECK。

编译器增强版本

增强版本有很多个，不同的编译器提供的的 SEH 增强版本或多或少都有不同之处。但是，他们一般都是基于 windows 系统提供的原始版本进行完善的。一个典型的增强版就是微软的编译器（后面简称为 MSC）里提供的 `__try`、`__finally`、`__except`。约定接下来使用这个增强版作为目标进行分析。

初步认识

在 Win32 SDK 文档中，有所谓的“基于帧”的异常处理程序模型：

```
// . . .
    __try {
        // guarded code
    }
    __except ( /* filter expression */ ) {
        // termination code
    }
    // . . .
```

称 `__try` 子句后的复合语句为受保护节，称 `except` 表达式为筛选表达式，它的值决定了异常的处理方式，称 `__except` 子句后的复合语句为异常处理程序。执行过程为

1. 执行受保护节
2. 如果在受保护节执行过程中未发生异常，则继续执行 **`__except`** 子句之后的语句（注意：不会跳过异常处理程序）。
3. 如果在受保护节的执行过程中或受保护节调用的任何例程中发生异常，则会计算 **`__except`** 表达式。

- 无法识别异常并将控制权转交给其他处理程序（EXCEPTION_CONTINUE_SEARCH）（0）
- 识别异常，已将其消除（EXCEPTION_CONTINUE_EXECUTION）（-1）
- 识别异常，将进行处理（EXCEPTION_EXECUTE_HANDLER）（1）

简单地说，在一个函数中，一个 __try 块中的所有代码就通过创建在这个函数的堆栈帧上的一个 EXCEPTION_REGISTRATION 结构来保护。在函数的入口处，这个新的 EXCEPTION_REGISTRATION 结构被放在异常处理程序链表的头部。在 __try 块结束后，相应的 EXCEPTION_REGISTRATION 结构从这个链表的头部被移除。正如系统原始版本中提过的，异常处理程序链表的头部被保存在 FS:[0] 处。因此，如果你在调试器中单步跟踪时看到类似下面的指令时

MOV DWORD PTR FS:[00000000],ESP 或者 MOV DWORD PTR FS:[00000000],ECX

那么就可以确定这段代码正在进入或退出一个 __try / __except 块。

系统实现的 EXCEPTION_REGISTRATION 结构中的 Handler 函数相当于这里的 filter expression，这个过滤器表达式代码决定了后面的大括号中的代码是否执行。

由于过滤器表达式代码是程序员自己写的，当然可以自由决定在代码中的某个地方是否处理某个特定的异常。它可以简单的只是一句 EXCEPTION_EXECUTE_HANDLER，也可以是先 sleep(114514)，然后再返回一个值来告诉操作系统下一步做什么...（重点在于过滤器表达式返回值必须是我前面讲的有效的宏定义

以上听起来很简单，然而事实上，过滤器表达式代码并不是被操作系统直接调用的。实际上各个 EXCEPTION_REGISTRATION 结构的 handler 域都指向了同一个函数。这个函数在 Visual C++ Runtime 中，它被称为 __except_handler4；此外，并不是每次进入或退出一个 __try 块时就创建或撤销一个 EXCEPTION_REGISTRATION 结构。相反，在使用 SEH 的任何函数中只创建一个 EXCEPTION_REGISTRATION 结构。同样，你可以在一个函数中嵌套使用 __try 块，但 Visual C++ 仍旧只是创建一个 EXCEPTION_REGISTRATION 结构。

显然，如果不扩增一些新的数据结构或者是使用一些 trick，以上功能是不可能实现的，这便是接下来要讲的东西。

深入了解

VC 的标准异常帧

Visual C++的 SEH实现并没有使用原始的 EXCEPTION_REGISTRATION 结构——它对原有的数据结构进行了修改（其余结构沿用），它是允许单个函数 __except_handler4 处理所有异常并将执行流程传递到相应的过滤器表达式和 __except 块的关键。

```
#define TRYLEVEL_NONE               -2
#define TRYLEVEL_INVALID            -1

struct SCOPETABLE_ENTRY
{
    unsigned int EnclosingLevel; // 构成链表结构
    unsigned int FilterFunc; // 过滤器表达式代码的地址
    unsigned int HandlerFunc; // 相应的__except块的地址
    DWORD GSCookieOffset;
    DWORD GSCookieXOROffset;
    DWORD EHCookieOffset;
    DWORD EHCookieXOROffset;
};

struct EH4_EXCEPTION_REGISTRATION {
        PEXCEPTION_POINTERS xpointers; // 指向 EXCEPTION_POINTERS 结构（一个标准的 Win32 结构）的指针
                                       // 即调用 GetExceptionInformation 这个API 时返回的指针
        struct _EXCEPTION_REGISTRATION *prev; // 将原有的 next 修改为了 prev
        // 保留了 handler
        void (*handler) (PEXCEPTION_RECORD,
                         PEXCEPTION_REGISTRATION,
                         PCONTEXT,
                         PEXCEPTION_RECORD);
        // 新增加了 3 个域
        struct SCOPETABLE_ENTRY *scopetable; // 作用表域
        int trylevel; // 作为 scopetable_entry 的索引
        int _ebp; // EXCEPTION_REGISTRATION 结构创建之前 EBP 的值
};
```

注：本结构体只用于理解原始版和增强版的区别，实际代码中并没有这种形式的定义

_ebp 域成为扩展的 EXCEPTION_REGISTRATION 结构的一部分并非偶然。它是通过 PUSH EBP 这条指令被包含进这个结构中的，而大多数函数开头都是这条指令（通常编译器并不为使用 FPO 优化的函数生成标准的堆栈帧，这样其第一条指令可能不是 PUSH EBP 。但是如果使用了 SEH 的话，那么无论你是否使用了 FPO 优化，编译器一定生成标准的堆栈帧）。这条指令可以使 EXCEPTION_REGISTRATION 结构中所有其它的域都可以用一个相对于栈帧指针（EBP）的负偏移来访问，这样效率显然高得多。

根据以上源代码，可以基本模拟出一个 VSC 标准异常堆栈帧的内存布局：

```
[EBP-00] _ebp
[EBP-04] trylevel
[EBP-08] scopetable数组指针
[EBP-0C] handler函数地址
[EBP-10] 指向前一个EXCEPTION_REGISTRATION结构
[EBP-14] GetExceptionInformation
[EBP-18] 栈帧中的标准ESP
```

根据以上信息，我们以一个简单的伪代码分析一开始说的“嵌套的 `__try/__except` 块或同一个函数里多个并列 `__try/__except` 块都共用同一个 `EXCEPTION_REGISTRATION` 是怎么实现的。

```
VOID SimpleSEH() {
    __try {
        // A
    } __except(ExceptionFilter_0(...)) {
        ExceptCodeBlock_0;
    }
    __try {
        // C
        __try {
            // B
        } __except(ExceptionFilter_1(...)) {
            ExceptCodeBlock_1;
            }
    } __except(ExceptionFilter_2(...)) {
        ExceptCodeBlock_2;
    }
}
```

编译时，编译器会为 SimpleSEH 分配一个 `EXCEPTION_REGISTRATION` 和一个拥有3个成员的 `scopetable` 数组，并将 `EXCEPTION_REGISTRATION::scopetable` 指向该数组（*请留意：`EXCEPTION_REGISTRATION::scopetable` 只是一个指针，不是数组*）。然后按照 `__try` 关键字出现的顺序，将对应的 `__except/__finally` 都存入该数组。假设我们已经了解了 `__except_handler4` 的具体实现（事实上我会在后文给出），并且已知这个函数会依次调用 `scopetable` 中的 `HandlerFunc`。

编译后得到的完整 `scopetable` 数组如下：

```
    scopetable[0].previousTryLevel = TRYLEVEL_NONE;
    scopetable[0].lpfnFilter = ExceptionFilter_0;
    scopetable[0].lpfnHandler = ExceptCodeBlock_0;

    scopetable[1].previousTryLevel = TRYLEVEL_NONE;
    scopetable[1].lpfnFilter = ExceptionFilter_1;
    scopetable[1].lpfnHandler = ExceptCodeBlock_1;
```

```
    scopetable[2].previousTryLevel = 1;
    scopetable[2].lpfnFilter = ExceptionFilter_2;
    scopetable[2].lpfnHandler = ExceptCodeBlock_2;
```

这个 `scopetable` 数组显然是一个链表。我们来模拟执行一下：

- 假如 B 部分触发异常：首先遍历到 `scopetable[2]`，处理完后，找到 `scopetable[2].EnclosingTryLevel`，发现其值为1，那么遍历到 `scopetable[1]`，处理完后，找到 `scopetable[1].EnclosingTryLevel`，发现其值为 `TRYLEVEL_NONE`，于是停止遍历。
- 假如 A 部分触发异常：问题来了，这次的异常是在第一个 `__try` / `__except` 中触发的，轮不到 `scopetable[2]` 来处理，怎么办？

不知道读者是否还记得前面给出的 `EH4_EXCEPTION_REGISTRATION::trylevel` 域，它的作用是标识从哪个数组单元开始遍历。与 `SCOPETABLE_ENTRY::EnclosingTryLevel` 不同，`EXCEPTION_REGISTRATION::trylevel` 是动态变化的，也就是说，这个值在 SimpleSEH 执行过程中是会经常改变的。比如：

- 执行到 A 时，该值被修改为 0；
- 执行到 C 时，该值被修改为 1；
- 执行到 B 时，该值被修改为 2。

这样，当异常触发时候，MSC 就能正确的遍历 `scopetable` 了。

到这里，我们已经熟悉了 MSC 增强版 SEH 的概要流程了，接下来将结合真实汇编代码继续分析。

汇编代码很简单，注释里也有详细的分析过程了，我不再多啰嗦。只提两点：

1. `EXCEPTION_REGISTRATION::scopetable` 指针被用 `__security_cookie` 进行了异或加密。
2. `EXCEPTION_REGISTRATION::scopetable` 并不直接指向 `scopetable_entry` 数组，在第一个 `scopetable_entry` 之前有 16 个字节的坑。后续分析中会看到，它的主要作用是帮助验证 `scopetable` 是否被破坏。

下面给出了 `__except_handler4` 的具体实现，MSC 并没有完全的公开这一部分的源代码，目前手上只有反编译 + 反汇编版本 将就着看吧：

```
#define EXCEPTION_NONCONTINUABLE 0x1    // Noncontinuable exception
#define EXCEPTION_UNWINDING 0x2         // Unwind is in progress
```

```
#define EXCEPTION_EXIT_UNWIND 0x4       // Exit unwind is in progress
#define EXCEPTION_STACK_INVALID 0x8     // Stack out of limits or unaligned
#define EXCEPTION_NESTED_CALL 0x10      // Nested exception handler call
#define EXCEPTION_TARGET_UNWIND 0x20    // Target unwind in progress
#define EXCEPTION_COLLIDED_UNWIND 0x40  // Collided exception handler call
#define EXCEPTION_UNWIND (EXCEPTION_UNWINDING | EXCEPTION_EXIT_UNWIND | \

EXCEPTION_TARGET_UNWIND | EXCEPTION_COLLIDED_UNWIND)
                                                    // 也就是 0x66

#define EH_EXCEPTION_NUMBER ('msc' | 0xE0000000)

EXCEPTION_DISPOSITION
__cdecl _except_handler4(_EXCEPTION_RECORD* ExceptionRecord,
    EXCEPTION_REGISTRATION_RECORD* EstablisherFrame,
    _CONTEXT* ContextRecord,
    PVOID DispatcherContext)
{
    EH4_EXCEPTION_REGISTRATION_RECORD* EH4 = CONTAINING_RECORD(
        EstablisherFrame, EH4_EXCEPTION_REGISTRATION_RECORD, SubRecord);

    void* EH4_END = EH4 + 1;

    EH4_SCOPETABLE* ScopeTable = (EH4_SCOPETABLE*)(__security_cookie ^
(DWORD)EH4->EncodedScopeTable);

    // Stack integrity checks:
    if (ScopeTable->GSCookieOffset != -2)
        __security_check_cookie(
            *(DWORD*)(ScopeTable->GSCookieXOROffset + (DWORD)EH4_END) ^ *
(DWORD*)(ScopeTable->GSCookieOffset + (DWORD)EH4_END));

    __security_check_cookie(
        *(DWORD*)(ScopeTable->EHCookieXOROffset + (DWORD)EH4_END) ^ *
(DWORD*)(ScopeTable->EHCookieOffset + (DWORD)EH4_END));

    EXCEPTION_DISPOSITION HandlerResult = ExceptionContinueSearch;

    if (ExceptionRecord->ExceptionFlags & 0x66) { // EXCEPTION_UNWIND 局部展
开
        if (EH4->TryLevel == -2)
            return ExceptionContinueSearch;
        _EH4_LocalUnwind(
            (DWORD)EstablisherFrame, -2, (DWORD)EH4_END,
(DWORD)&__security_cookie);
    } else {
        EXCEPTION_POINTERS ExceptionPointers = {};
        ExceptionPointers.ExceptionRecord = ExceptionRecord;
        ExceptionPointers.ContextRecord = ContextRecord;
        EH4->ExceptionPointers = &ExceptionPointers;

        bool v13 = false;

        DWORD LastTryLevel = EH4->TryLevel;

        if (LastTryLevel == -2)
            return ExceptionContinueSearch;
```

```
        do {
            void* FilterFunc = ScopeTable-
>ScopeRecord[LastTryLevel].FilterFunc;
            int EnclosingLevel = ScopeTable-
>ScopeRecord[LastTryLevel].EnclosingLevel;
            EH4_SCOPETABLE_RECORD* pScopeRecord = &ScopeTable-
>ScopeRecord[LastTryLevel];
            if (FilterFunc) {
                int FilterResult = _EH4_CallFilterFunc(FilterFunc, EH4_END);
                v13 = true;

                if (FilterResult < 0) {
                    HandlerResult = ExceptionContinueExecution;
                    goto LABEL_23;
                }
                if (FilterResult > 0) {
                    if (ExceptionRecord->ExceptionCode ==
EH_EXCEPTION_NUMBER && _pDestructExceptionObject &&
_IsNonwritableInCurrentImage((char*)&_pDestructExceptionObject)) {
                        _pDestructExceptionObject(ExceptionRecord, 1);
                    }
                    _EH4_GlobalUnwind2(EstablisherFrame, ExceptionRecord);
// 全局展开

                    if (EH4->TryLevel != LastTryLevel) {
                        _EH4_LocalUnwind(EH4_END, &__security_cookie);
                    }
                    EH4->TryLevel = EnclosingLevel;

                    // Stack integrity checks:
                    if (ScopeTable->GSCookieOffset != -2)
                        __security_check_cookie(
                            *(DWORD*)(ScopeTable->GSCookieXOROffset +
(DWORD)EH4_END) ^ *(DWORD*)(ScopeTable->GSCookieOffset + (DWORD)EH4_END));

                    __security_check_cookie(
                        *(DWORD*)(ScopeTable->EHCookieXOROffset +
(DWORD)EH4_END) ^ *(DWORD*)(ScopeTable->EHCookieOffset + (DWORD)EH4_END));

                    _EH4_TransferToHandler(pScopeRecord->HandlerFunc,
EH4_END);
                     // 这段函数不会返回，即这段代码中没有 ret 指令。执行完整个
HandlerFunc 后，会接着执行其后的指令，并不会返回 _except_handler4。

                    __debugbreak();
                    _crt_debugger_hook();
                }
            } else {
                v13 = true;
            }
            LastTryLevel = EnclosingLevel;
        } while (LastTryLevel != -2);
        if (!v13)
            return HandlerResult;
    }
LABEL_23:
    // Stack integrity checks:
```

```
    if (ScopeTable->GSCookieOffset != -2)
        __security_check_cookie(
            *(DWORD*)(ScopeTable->GSCookieXOROffset + (DWORD)EH4_END) ^ *
(DWORD*)(ScopeTable->GSCookieOffset + (DWORD)EH4_END));

    __security_check_cookie(
        *(DWORD*)(ScopeTable->EHCookieXOROffset + (DWORD)EH4_END) ^ *
(DWORD*)(ScopeTable->EHCookieOffset + (DWORD)EH4_END));

    return HandlerResult;
}
```

关于 `_EH4_CallFilterFunc` 和 `_EH4_TransferToHandler`，手上只有反汇编代码：

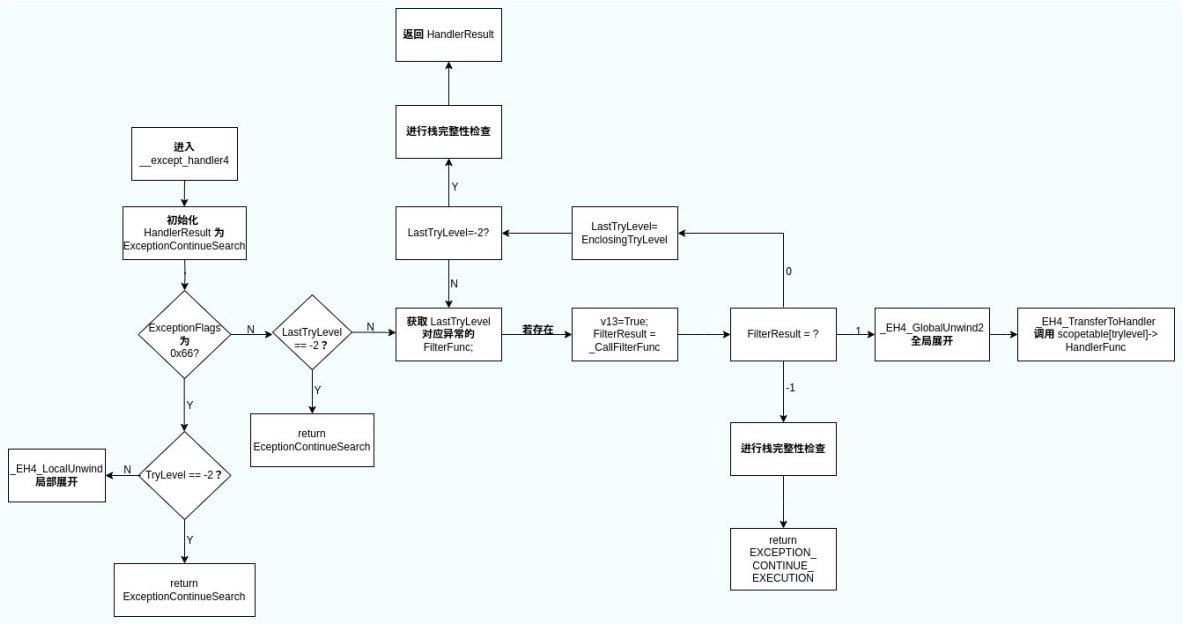
```
; _EH4_CallFilterFunc
push    ebp
push    esi
push    edi
push    ebx
mov     ebp,edx
xor     eax,eax
xor     ebx,ebx
xor     edx,edx
xor     esi,esi
xor     edi,edi
call    ecx ; FilterFunc();
pop     ebx
pop     edi
pop     esi
pop     ebp
ret

; _EH4_TransferToHandler
mov     ebp,edx
mov     esi,ecx ; esi = HandlerFunc
mov     eax,ecx
xor     eax,eax
xor     ebx,ebx
xor     ecx,ecx
xor     edx,edx
xor     edi,edi
jmp     esi     ; jmp HandlerFunc
```

根据源代码，可以总结出以下要点：

- VC的 `__except_handler4` 函数与系统实现版本里的 `_except_handler` 十分相似，带有同样的四个参数；
- 之后的流程见下面的示意图：

{{< rawhtml >}}



{{<

/rawhtml >}}

{{< rawhtml >}}
{{< /rawhtml >}}

展开

为了说明这个概念，需要先回顾下异常发生后的处理流程。

我们假设一系列使用 SEH 的函数调用流程：`func1` -> `func2` -> `func3`，假定在 `func3` 执行的过程中触发了异常，由 `func1` 完成处理。

回想一下我们在学习系统实现版本的 SEH 时总结的分发软件异常流程：`RtlRaiseException` -> `RtlDispatchException` -> `RtlpExecuteHandlerForException`，由于 `R tlDispatchException` 会遍历异常链表，对每个 `EXCEPTION_REGISTRATION` 都调用 `RtlpExecuteHandlerForException`，而 `RtlpExecuteHandlerForException` 会调用 `EXCEPTION_REGISTRATION::handler`，后者在 MSC 中被实现为 `_exception_handler4`。又，如上分析，该函数内部遍历 `EH4_EXCEPTION_REGISTRATION::scopetable`，如果遇到有 `scopetable_entry::FilterFunc` 返回 `EXCEPTION_EXECUTE_HANDLER`，那么 `scopetable_entry::HandlerFunc` 就会被调用，来处理该异常。

因为 `HandlerFunc` 不会返回到 `_except_handler4`（`_EH4_TransferToHandler` 没有 `ret/retn` 指令），于是执行完 `HandlerFunc` 后，就会从 `HandlerFunc` 之后的代码继续执行下去。也就是说，假设 `func3` 中触发了一个异常，该异常被 `func1` 中的 `__except` 处理块处理了，那 `__except` 处理块执行完毕后，就从其后的指令继续执行下去，即异常处理完毕后，接着执行的就是 `func1` 的代码，不会再回到 `func2` 或

者 func3。问题来了，func2 和 func3 中占用的资源怎么办？这些资源（比如申请的内存）是不会自动释放的，那岂不是会有资源泄漏问题？

从这里，我们可以引申出“展开”的概念：“展开”，说白了就是“清理”。（注：这里的清理主要包含动态分配的资源的清理，栈空间是由 func1 的 mov esp,ebp 这类操作顺手清理的）

保持思考，这个展开工作由谁来完成呢？

显然，由 func1 来完成肯定不合适，毕竟 func2 和 func3 有没有申请资源、申请了哪些资源，func1 无从得知。于是这个展开工作只能交给 func2 和 func3 自己来完成。

展开分为两种：全局展开、局部展开。

全局展开是指针对异常链表中的某一段，局部展开针对指定的 EXCEPTION_REGISTRATION。仍然用上面的例子，局部展开是指具体某一函数内部的清理（例如对 func3 内部分配的动态资源）；而全局展开是指，从异常触发点（func3）到异常处理点（func1）之间所有函数（包含异常触发点 func3）的局部清理的总和。

全局展开

有一个尴尬的事实是没有人能找到用于全局展开的 _EH4_GlobalUnwind2 的源代码（官方也没有给出），下面是利用 ida 反编译得到的结果：

```
; __fastcall _EH4_GlobalUnwind2(EstablisherFrame, ExceptionRecord)
@_EH4_GlobalUnwind2@4 proc near
push    ebp
mov     ebp, esp
push    ebx
push    esi
push    edi
push    0
push    0
push    offset ReturnPoint
push    ecx ; pExceptionRegistration
call    _RtlUnwind@16   ; RtlUnwind(x,x,x,x)

ReturnPoint:
pop     edi
pop     esi
pop     ebx
pop     ebp
retn
```

```
@_EH4_GlobalUnwind@4 endp
; RtlUnwind 为导入函数
```

根据 _EH4_GlobalUnwind2 的汇编代码可以得知其调用了 RtlUnwind 函数，这个函数在 wrk 有源码实现，MSC 实现版本与之相差不多，这里我们就以其为例：

```
VOID
RtlUnwind (
    IN PVOID TargetFrame OPTIONAL, // 要展开的目标帧
    IN PVOID TargetIp OPTIONAL, // 展开的继续地址
    IN PEXCEPTION_RECORD ExceptionRecord OPTIONAL, // 异常记录
    IN PVOID ReturnValue // 整数返回寄存器的返回值
    )
```

代码比较长，不过并不复杂：

- 如果没有指定要展开的目标帧的话，程序会自己构建一个，同时在异常标志中同时设置 EXCEPTION_EXIT_UNWIND 和 EXCEPTION_UNWINDING 标志
- 如果指定了目标帧，在异常标志中设置 EXCEPTION_UNWINDING 标志；
- 扫描异常注册链表：
  - 如果是展开的目标：调用 _ZwContinue 进行系统服务继续执行；
  - 如果不是展开的目标：正常情况下先调用 RtlpExecuteHandlerForUnwind，它会调用当前 EH4_EXCEPTION_REGISTRATION->Handler，根据处理结果：
    - ExceptionContinueSearch：获取下一个帧地址并搜索
    - ExceptionCollidedUnwind：调用 handler 时再次触发异常，并继续搜索。
    - EXCEPTION_NONCONTINUABLE：其他处理值均无效；
  - 非正常情况：
    - 如果堆栈没有对齐或正在处理的 EXCEPTION_REGISTERATION 超出了堆栈限制：调用 RtlRaiseException 起一个异常 STATUS_BAD_STACK，且 EXCEPTION_NONCONTINUABLE；
    - DPC 堆栈上，切换堆栈限制到 DPC 堆栈并重新启动循环
- 调用 RtlpUnlinkHandler 从链表中删除已被调用的 Handler。

局部展开

另一个尴尬的事实是 _EH4_LocalUnwind 的源代码也没有，只能看汇编了（

```
_EH4_LocalUnwind(
        (DWORD)EstablisherFrame,
        EstablisherFrame->TravelLevel,
        &EstablisherFrame->_ebp,
        __security_cookie);

push    ebp
mov     ebp,dword ptr [esp+8] ; ebp = pExceptionRegistartion->_ebp
push    edx ; ulUntilTryLevel
push    ecx ; pExceptionRegistartion
push    dword ptr [esp+14h] ; push __security_cookie
call    __local_unwind4
add     esp,0Ch
pop     ebp
ret     8
```

注意到其调用了 `__local_unwind4`，这个倒是有反汇编...个 p 啊：

```
__declspec(naked) VOID __cdecl
__local_unwind4(DWORD* CookiePointer, PVOID EstablishFrame , DWORD
TargetEnclosingLevel)
{
    __asm {
        push ebx
        push esi
        push edi
        mov  edx, [esp+0x10]        // edx = CookiePointer
        mov  eax, [esp+0x14]        // eax = EstablishFrame
        mov  ecx, [esp+0x18]        // ecx = EnclosingLevel

        // establish new EstablishFrame for the protected seh node.
        push ebp                     // context frame
        push edx                     // CookiePointer
        push eax                     // PrevEstablishFrame
        push ecx                     // EnclosingLevel
        push ecx                     // Cookie
        push _unwind_handler4        // handler
        push fs:[0]                  // next

        mov  eax, __security_cookie  // edit the Cookie above to real value
        xor  eax, esp
        mov  [esp+0x08], eax

        // complete establish the seh.
        mov  fs:[0], esp

_lu_top:
        mov  eax, [esp+0x30]        // EstablishFrame
        mov  ebx, [eax+8]           // EstablishFrame.EH4_SCOPETABLE
        mov  ecx, [esp+0x2c]        // CookiePointer
        xor  ebx, [ecx]             // ebx = decoded EH4_SCOPETABLE
        mov  esi, [eax+0x0c]        // eax = EstablishFrame.EnclosingLevel
        cmp  esi, END_POS
```

```
        jz   _lu_done               // meet the last level. exit.

        mov  edx, [esp+0x34]        // edx = EnclosingLevel
        cmp  edx, END_POS
        jz   __update_establishframe

        cmp  esi, edx               // EstablishFrame.EnclosingLevel <=
EnclosingLevel
        jbe  _lu_done

__update_establishframe:
        // Update the EstablishFrame.EnlosingLevel to EnclosingLevel
        lea  esi, [esi+esi*2]
        lea  ebx, [ebx+esi*4+0x10]      // ebx =
&EH4_SCOPETABLE.ScopeRecord[esi]
        mov  ecx, [ebx]                 // ecx =
EH4_SCOPETABLE.ScopeRecord[esi].EnclosingLevel
        mov  [eax+0x0c], ecx            // EstablishFrame.EnclosingLevel =
EH4_SCOPETABLE.ScopeRecord[esi].EnclosingLevel
        cmp  [ebx+4], 0                 //
EH4_SCOPETABLE.ScopeRecord[esi].FilterFunc != NULL ?
        jnz  _lu_top

//        push 0x101
        mov  eax, [ebx+8]               // eax =
EH4_SCOPETABLE.ScopeRecord[esi].HandlerFunc
//       call __NLG_Notify

        mov  ecx, 1
        mov  eax, [ebx+8]
//        call __NLG_Call
        call eax

        jmp  _lu_top

_lu_done:
        pop  ebx
        mov  fs:[0], ebx
        add  esp, 0x18
        pop  edi
        pop  esi
        pop  ebx
        retn
    }
}
```

这东西的逻辑倒是不复杂，关键点在于它会调用当前节点的 `HandlerFunc`。

总结一下，在异常处理过程中：

- 每个异常至少会遍历异常链表两次：
  - 一次是在 `RtlDispatchException` 中，遍历的目的是找到愿意处理该异常的 `_EXCEPTION_REGISTRATION_RECORD`，遍历的同时会调用

`RtlpExecuteHandlerForException` 运行异常处理程序；
    - 第二次是在展开过程中，遍历的目的是为了对每个遍历到的 `EXCEPTION_REGISTRATION_RECORD` 进行局部展开，遍历的同时会调用 `RtlpExecuteHandlerForUnwind` 运行异常处理程序（局部展开的话调用 `__local_unwind4` 起 `HandlerFunc`）
- 每个异常的 `scopetable` 也会被遍历至少两次：
    - 一次是 `__except_handler4` 中，遍历目的也是找到愿意处理该异常的 `scopetable_entry`，遍历的同时调用 `scopetable` 中的 HandlerFunc；
    - 第二次是 `__local_unwind4` 函数内，遍历的目的是找到所有指定范围内的能用的 `FilterFunc`，遍历的同时调用 `scopetable` 中的 HandlerFunc。

到这里，windows 的 SEH 机制就基本讲完了。

相关题目

~~有谁还记得这篇博客本来是 wp 的（（（~~

我们现在以 MoeCTF 2023 unwind 为例来应用一下上面的知识。

先看看 `main_0` 函数：

```
int __cdecl main_0(int argc, const char **argv, const char **envp)
{
    char v4; // [esp+0h] [ebp-100h]

    __CheckForDebuggerJustMyCode(&unk_41C063);
    sub_4113E3();
    puts("Welcome to moectf2023!!! Now you find YunZh1Jun's revenge!!!");
    puts("Do you know TEA(an encryption algorithm)? Do you know unwind in
SEH? ");
    puts("I believe you can understand them! So let me check your flag~");
    sub_4110CD("Input:", v4);
    sub_4113CA("%64s", (char)&byte_41A578);
    MEMORY[0] = 0;
    sub_4110FF();
    puts("Right flag! Have fun in moectf2023~");
    return 0;
}
```

可以看到 `MEMORY[0] = 0;` 被标着显眼的大红色，这其实是一个很强的提示：即程序中触发了非法内存访问的异常（`MEMORY[0]` 一定是一个非法的地址）。

来看汇编，在开头可以看到 `unwind`、`__except_handler4` 等 ida 加的注释，实际上，整个 main_0 函数 其实就是一个巨大的 __except_handler4 函数。

在输入函数正下方是第一个 try-except：

```
.text:00415908 ;   __try { // __except at loc_415928
.text:00415908 mov     [ebp+ms_exc.registration.TryLevel], 0 ; 注册的第一个
try 块，其 scopetable->trylevel 标记为 0
.text:0041590F mov     large dword ptr ds:0, 0
.text:0041590F ;   } // starts at 415908
.text:00415919 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
.text:00415920 jmp     short loc_41597A
.text:00415920
.text:00415922 ; ---------------------------------------------------------
---------------
.text:00415922
.text:00415922 loc_415922:                             ; DATA XREF:
.rdata:stru_4192E8↓o
.text:00415922 ;   __except filter // owned by 415908
.text:00415922 mov     eax, 1
.text:00415927 retn
.text:00415927
.text:00415928 ; ---------------------------------------------------------
---------------
.text:00415928
.text:00415928 loc_415928:                             ; DATA XREF:
.rdata:stru_4192E8↓o
.text:00415928 ;   __except(loc_415922) // owned by 415908
.text:00415928 mov     esp, [ebp+ms_exc.old_esp]
.text:0041592B push    offset aDx3906                  ; "DX3906"
.text:00415930 push    offset flag
.text:00415935 call    sub_41136B
.text:00415935
.text:0041593A add     esp, 8
.text:0041593D push    offset aDoctor3                 ; "doctor3"
.text:00415942 push    offset unk_41A580
.text:00415947 call    sub_41136B
.text:00415947
.text:0041594C add     esp, 8
.text:0041594F push    offset aFux1aoyun               ; "FUX1AOYUN"
.text:00415954 push    offset unk_41A588
.text:00415959 call    sub_41136B
.text:00415959
.text:0041595E add     esp, 8
.text:00415961 push    offset aR3verier                ; "R3verier"
.text:00415966 push    offset unk_41A590
.text:0041596B call    sub_41136B
.text:0041596B
.text:00415970 add     esp, 8
.text:00415973 mov     [ebp+ms_exc.registration.TryLevel], -2
```

既然有了前面知识的铺垫，那么这一段汇编代码理解起来应该没有什么难度：注意到 `.text:0041590F` 处会触发了一个读写非法地址异常，往下紧接着就是 __except filter，它直接返回了 1，也就是 EXCEPTION_EXECUTE_HANDLER，根据前面的知识，__except_handler4 在获取 FilterFunc 后会根据调用 filter 并根据 FilterResult 决定下一步的操作，这里为 1，故会进行全局展开，也就是接着调用 RtlUnwind，RtlUnwind 又会起一个 EH4_EXCEPTION_REGISTRATION->Handler，也就是我们这里的 __except 块。

注意到这里的 except 块调用了 4 次 sub_41136B，这个函数是一个正常的 TEA 加密，每次 TEA 会加密 8*8 位二进制，后面会讲。

我们接着来看，在一个异常被处理后，程序继续执行：

```
.text:0041597A                                  loc_41597A:
; CODE XREF: _main_0+D0↑j
.text:0041597A                                  ;   __try { // __except at
loc_415995
.text:0041597A C7 45 FC 01 00 00 00             mov
[ebp+ms_exc.registration.TryLevel], 1 ; 注册第二个异常快
.text:00415981 E8 79 B7 FF FF                   call    sub_4110FF
.text:00415981                                  ;   } // starts at 41597A
.text:00415981
.text:00415986 C7 45 FC FE FF FF FF             mov
[ebp+ms_exc.registration.TryLevel], -2
.text:0041598D EB 72                            jmp     short loc_415A01
.text:0041598D
.text:0041598F                                  ; ---------------------------
---------------------------------------------
.text:0041598F
.text:0041598F                                  loc_41598F:
; DATA XREF: .rdata:stru_4192E8↓o
.text:0041598F                                  ;   __except filter // owned by
41597A
.text:0041598F B8 01 00 00 00                   mov     eax, 1
.text:00415994 C3                               retn
.text:00415994
.text:00415995                                  ; ---------------------------
---------------------------------------------
.text:00415995
.text:00415995                                  loc_415995:
; DATA XREF: .rdata:stru_4192E8↓o
.text:00415995                                  ;   __except(loc_41598F) //
owned by 41597A
.text:00415995 8B 65 E8                         mov     esp,
[ebp+ms_exc.old_esp]
.text:00415998 C7 45 E0 00 00 00 00             mov     [ebp+var_20], 0
.text:0041599F EB 09                            jmp     short loc_4159AA
```

注意到 `.text:00415981` 调用了一个 `sub_4110FF`，双击进去看：

```
.text:00411820                          sub_411820 proc near
; CODE XREF: sub_4110FF↑j
.text:00411820
.text:00411820                          var_E4= dword ptr -0E4h
.text:00411820                          var_10= byte ptr -10h
.text:00411820                          var_C= dword ptr -0Ch
.text:00411820                          var_4= dword ptr -4
.text:00411820
.text:00411820 55                       push    ebp
.text:00411821 8B EC                    mov     ebp, esp
.text:00411823 81 EC D0 00 00 00        sub     esp, 0D0h
.text:00411829 53                       push    ebx
.text:0041182A 56                       push    esi
.text:0041182B 57                       push    edi
.text:0041182C 8D 7D F0                 lea     edi, [ebp+var_10]
.text:0041182F B9 04 00 00 00           mov     ecx, 4
.text:00411834 B8 CC CC CC CC           mov     eax, 0CCCCCCCCh
.text:00411839 F3 AB                    rep stosd
.text:0041183B A1 08 A1 41 00           mov     eax, ___security_cookie
.text:00411840 33 C5                    xor     eax, ebp
.text:00411842 89 45 FC                 mov     [ebp+var_4], eax
.text:00411845 C7 45 F4 FD 12 41 00     mov     [ebp+var_C], offset
loc_4112FD ; trick
.text:0041184C FF 75 F4                 push    [ebp+var_C]
.text:0041184F 64 FF 35 00 00 00 00     push    large dword ptr fs:0
.text:00411856 64 89 25 00 00 00 00     mov     large fs:0, esp
.text:0041185D CC                       int     3
; Trap to Debugger
.text:0041185E 8B 04 24                 mov     eax, [esp+0E4h+var_E4]
.text:00411861 64 A3 00 00 00 00        mov     large fs:0, eax
.text:00411867 83 C4 08                 add     esp, 8
.text:0041186A 5F                       pop     edi
.text:0041186B 5E                       pop     esi
.text:0041186C 5B                       pop     ebx
.text:0041186D 8B 4D FC                 mov     ecx, [ebp+var_4]
.text:00411870 33 CD                    xor     ecx, ebp
; StackCookie
.text:00411872 E8 D3 F8 FF FF           call
j_@__security_check_cookie@4    ; __security_check_cookie(x)
.text:00411872
.text:00411877 81 C4 D0 00 00 00        add     esp, 0D0h
.text:0041187D 3B EC                    cmp     ebp, esp
.text:0041187F E8 C0 F9 FF FF           call    j___RTC_CheckEsp
.text:0041187F
.text:00411884 8B E5                    mov     esp, ebp
.text:00411886 5D                       pop     ebp
.text:00411887 C3                       retn
```

注意到 `.text:0041185D` 的 `int 3` 触发了调试异常，第一次做到这里的时候，我们可能会立马返回去看 `loc_415995` 处的 `except` 块，毕竟它的 filterfunc 也是直接返回了 EXCEPTION_EXECUTE_HANDLER，_except_handler4 肯定会调用这里。

停下来思考。

还记得在前言中提到的 trick 吗？这道题目最大的坑就是在进行栈完整性检查后的一段汇编中：

```
.text:00411845 C7 45 F4 FD 12 41 00          mov     [ebp+var_C], offset
loc_4112FD ; trick
.text:0041184C FF 75 F4                      push    [ebp+var_C]
.text:0041184F 64 FF 35 00 00 00 00          push    large dword ptr fs:0
.text:00411856 64 89 25 00 00 00 00          mov     large fs:0, esp
```

在这里，其再次注册了一个异常处理块，也就是说，出题人在这里插入了一个异常链表的节点，这个节点里的 loc_4112FD 函数第一次被 RtlDispatchException 调用，紧接着在栈展开时又被调用一次，但是这个 新的 EXECEPTION_REGISTRATION 结构并不处理异常，`loc_415995` 处的 `except` 块，这个块的工作正是逐个检查输入值是否正确。

然后我们来看那个正常的 TEA：

```
int __cdecl sub_415700(unsigned int *v, _DWORD *k)
{
    int i; // [esp+D0h] [ebp-68h]
    unsigned int v1; // [esp+118h] [ebp-20h]
    unsigned int v0; // [esp+124h] [ebp-14h]
    int sum; // [esp+130h] [ebp-8h]

    __CheckForDebuggerJustMyCode(&unk_41C063);
    sum = 0;
    v0 = *v;
    v1 = v[1];
    for ( i = 0; i < 32; ++i )
    {
        sum -= 1640531527;
        v0 += (k[1] + (v1 >> 5)) ^ (sum + v1) ^ (*k + 16 * v1);
        v1 += (k[3] + (v0 >> 5)) ^ (sum + v0) ^ (k[2] + 16 * v0);
    }
    *v = v0;
    v[1] = v1;
    return ++cnt;
}
```

我简单改了一下变量名称，这个函数也就是 TEA 的加密函数，通过动调可以获得 sum 值，于是可以简单写一个解密脚本：

```
void decrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0 = v[0], v1 = v[1], sum = 0xC6EF3720, i;  /* set up */
    uint32_t delta = 0x61C88647;                       /* a key schedule
constant */
    uint32_t k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];   /* cache key */
```

```
    for (i = 0; i < 32; i++) {                          /* basic cycle start
*/
        v1 -= ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5) + k3);
        v0 -= ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5) + k1);
        sum += delta;
    }                                                   /* end cycle */
    v[0] = v0; v[1] = v1;
}
```

加密流程大概是：将 flag 转为 16 进制，然后每次取 16 位进行加密，密钥在 {"DX3906", "doctor3", "FUX1AOYUN", "R3verier"} 四个里轮换（密钥也是转为 16 进制，不足 32 位补 0，注意下字节序），这个流程进行四次可以将 flag 前一半加密；然后再对后一半以此流程加密，重复两次。

最终用来对比的密文在：

```
.text:004159A1                                      loc_4159A1:
; CODE XREF: _main_0:loc_4159F8↓j
.text:004159A1 8B 45 E0                             mov     eax, [ebp+var_20]
.text:004159A4 83 C0 01                             add     eax, 1
.text:004159A7 89 45 E0                             mov     [ebp+var_20], eax
.text:004159AA                                      loc_4159AA:
; CODE XREF: _main_0+14F↑j
.text:004159AA 83 7D E0 40                          cmp     [ebp+var_20], 40h ; '@'
.text:004159AE 7D 4A                                jge     short loc_4159FA
.text:004159AE
.text:004159B0 8B 45 E0                             mov     eax, [ebp+var_20]
.text:004159B3 0F B6 88 00 A0 41 00                 movzx   ecx, byte_41A000[eax]
.text:004159BA 8B 55 E0                             mov     edx, [ebp+var_20]
.text:004159BD 0F B6 82 78 A5 41 00                 movzx   eax, flag[edx]
.text:004159C4 3B C8                                cmp     ecx, eax
.text:004159C6 74 30                                jz      short loc_4159F8
.text:004159C6
.text:004159C8 8B F4                                mov     esi, esp
.text:004159CA 68 38 7C 41 00                       push    offset aOoooooooooopsT
; "Ooooooooooops!Try again!"
.text:004159CF FF 15 74 B1 41 00                    call    ds:puts

.text:004159F8                                      loc_4159F8:
; CODE XREF: _main_0+176↑j
.text:004159F8 EB A7                                jmp     short loc_4159A1
```

这一部分汇编比较简单（我删掉了一部分没用的），实际上实现了一个逐位比较密文的循环，故正确的 flag 对应的密文可以在 byte_41A000 里找到，双击进去就有，正好 64 个字符（128 位 16 进制）。

解密就是先将密文一分为二，前一半解密一次，后一半解密两次（注意字节序），拼接一下输出为字符串就得到 flag 了。

参考

（排名不分先后）

1. 15pb调试器学习总结_except_handler4以及栈展开分析 - By 流韵山庄
2. 深入解析结构化异常处理(SEH) - by Matt Pietrek
3. SEH分析笔记（x86篇） - By boxcounter
4. Windows异常处理机制简介
5. SEH进阶（2）
6. 部分反编译代码来源 - By cradiator
7. 逆向分析MSVCR90D.dll!_EH4_LocalUnwind函数 - By yuzl32
8. Windows-SEH学习笔记 - By 云之君
9. windows 异常处理
10. Windows-Research-Kernel-WRK - By HighSchoolSoftwareClub
11. ManualMapped_SEH_32bit - By Speedi13
12. chatgpt.