这一章开始描述的是SafeSEH相关的内容，什么事SafeSEH呢？

在以前的调试中，我们曾经直接覆盖异常处理指针来执行shellcode，那种简单粗暴的办法相当的好用，先溢出覆盖异常处理指针，然后构造个异常或者溢出后就会触发异常，触发异常后会取出异常处理指针，这时的异常处理指针已经被我们覆盖掉了

一般是这样构造的：S.E.H指针覆盖成pop pop retn，为什么呢？因为在执行异常处理的时候会先将nextSEH指针的地址压栈，然后再将两个现场的参数压栈，所以我们先弹掉两个现场参数，然后就可以retn到nextSEH指针的地址了，这时候nextSEH的指针被我们覆盖成jmp 6，一般来说后面直接跟上shellcode就行，这样就可以直接跳到shellcode的起始地址

从上面的描述可以看出来，我们只要覆盖S.E.H就可以很方便的执行shellcode，微软为了防止这类的攻击，在Win XP sp2开始，加入了对异常处理的保护机制，接下来主要是理论知识，还有一些补充的知识

SafeSEH整个工作流程

1. 检查异常处理链是否位于当前程序的栈中，如果不在当前栈中，程序将终止异常处理函数的调用，因为直接覆盖就会把异常处理指针覆盖，会导致异常处理指针不在栈中
2. 检查异常处理函数指针是否指向当前程序的栈中，如果指向当前程序的栈中，程序将终止异常处理函数的调用，这个明显是针对传统直接覆盖S.E.H的攻击手法
3. 在前面的两项检查都通过后，程序会调用一个全新的函数RtlIsValidHandler()，来对异常处理函数的有效性进行验证

RtlIsValidHandler()函数会判断异常处理函数是不是在加载模块的内存空间，如果是则进行如下校验

1. 判断程序是否设置IMAGE\_DLLCHARACTERISTICS\_NO\_SHE标识，如果设置了这个标识，这个程序内的异常会被忽略，返回校验失败，也就是不执行异常
2. 检测程序是否包含安全S.E.H表，如果程序包含安全S.E.H表，则将当前异常处理函数地址与该表进行匹配，匹配成功返回校验成功
3. 判断程序是否设置了ILonly标识，如果设置了这个标识，则直接返回校验失败
4. 判断异常处理函数是否处于不可执行页上，当异常处理函数位于不可执行页上，校验函数将检测DEP是否开启，如果系统未开启DEP则返回校验成功

如果RtlIsValidHandler()函数不在加载模块的内存空间，校验函数将直接进行DEP检测

1. 判断异常处理函数是否处于不可执行页上，当异常处理函数位于不可执行页上，校验函数将检测DEP是否开启，如果系统未开启DEP则返回校验成功
2. 判断系统是否允许跳转到加载模块的内存空间外执行，如果允许则返回校验成功

从上面的分析来看，SafeSEH对S.E.H的校验机制已经非常完善了，以前那种直接覆盖异常处理指针的手法看来是行不通了

但是我们来仔细看看都有哪些情况是允许异常处理函数执行的

1. 异常处理函数位于加载模块内存范围外，DEP关闭
2. 异常处理函数位于加载模块内存范围外，相应模块未启用SafeSEH，同时相应模块不是纯ILonly
3. 异常处理函数位于加载模块内存范围内，相应模块启用SafeSEH，异常处理函数包含在安全S.E.H表中

来分析三种情况的可能性

1. 在排除DEP的干扰后，我们找一个加载模块外的跳板，这样就可以跳到shellcode，可行性比较高
2. 找到未启用SafeSEH模块中的指令作为跳板就可以跳到shellcode，可行性也是不错的
3. 两种思路，一个是清空安全S.E.H表，第二个是将我们的伪造异常处理函数指针注册到安全S.E.H表中，然而这个表在内存中是加密存放的，直接突破它几乎不太可能

看了上面的分析，有没有一种不安的感觉，仿佛想要突破SafeSEH看来很棘手，不着急，接下来介绍一个大招

当我们伪造的异常处理函数指针指向了堆区，那么就算安全校验发现不安全，依旧会执行我们伪造的异常处理函数，所以有个堆就好办了。