API（应用程序接口）：

操作系统是用户与计算机硬件系统之间的接口，用户通过操作系统的帮助，可以快速、有效和安全、可靠地操纵计算机系统中的各类资源。为使用户能更方便地使用操作系统，OS又向用户提供了如下两类接口：

1. 用户接口：操作系统专门为用户提供了“用户与操作系统的接口”，通常称为用户接口。该用户接口支持用户与OS之间进行交互，即由用户向OS请求提供特定的服务，而系统则把服务的结果返回给用户。
2. 程序接口：操作系统向编程人员提供了“程序与操作系统的接口”，通常称为用户程序接口，即API。该接口是为程序员在编程时使用的，系统和应用程序通过这个接口，可在执行中访问系统中的资源各取得OS服务，它也是程序能取得操作系统服务的唯一途径。

Windows下API Hook技术：

1. 前言：

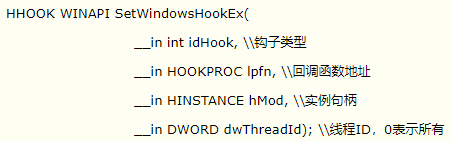
在Windows中，每个进程都有自己的私有地址空间。当我们用指针来引用内存的时候，指针的值表示的是进程自己的自制空间的一个内存地址。进程不能创建一个指针来引用属于其他进程的内存。

独立的地址空间对开发人员和用户来说都是非常有利的。对开发人员来说，系统更有可能捕获错误的内存读、写。对用户来说，操作系统变得更加健壮。当然这样的健壮性也是要付出代价的，因为它使我们很难编写能够与其他进程通信的应用程序或对其他进程进行操控的应用程序。

在《Windows核心编程》第二十二章《DLL注入和API拦截》中讲解了很多机制，它们可以将一个DLL注入到另一个进程地址的空间中。一旦DLL代码进入另一个地址空间，那么我们就可以在那个进程中为所欲为了。

1. 使用Windows挂钩来注入DLL
   1. 简单Windows消息hook
      1. SetWindowsHookEx

安装钩子的函数声明如下：



让我们通过一个例子来学习这个API。进程A（一个游戏改键工具，需要截获键盘消息）安装了WH\_KEYBOARD\_LL挂钩，如下所示：HOOK hHook = SetWindowsHookEx(WH\_KEYBOARD\_LL,LowLevelKeyBoardProc,hInstDll,0);第一个参数表示挂钩类型，是一个低级键盘钩子。第二个参数是一个函数地址（在我们的地址空间内），在窗口即将有键盘消息的时候，系统应该调用这个函数。第三个参数hInstDll标识一个DLL，这个DLL中包含了LowLevelKeyBoardProc函数。在Windows中，hInstDll的值就是DLL被映射到的虚拟内存地址。最后一个参数表示要给哪个子线程安装钩子，0表示给所有GUI线程安装。

现在让我们来看看接下来会发生什么：

1. 进程B中的一个线程准备向一个窗口发送一个键盘消息。
2. 系统检查该线程是否已经安装了WH\_KEYBOARD\_LL钩子
3. 系统检查LowLevelKeyBoardProc所在的dll是否已经映射到进程B的地址空间中

（在WIN32环境中，DLL函数中的代码所创建的任何对象（包括变量）都归调用它的线程或进程所有。当进程在载入DLL时，操作系统自动把DLL地址映射到该进程的私有空间，也就是进程的虚拟地址空间，而且也复制该DLL的全局数据的一份拷贝到该进程空间。也就是说每个进程所拥有的的相同的DLL全局数据，它们的名称相同，但其值却并不一定相同，而且是互不干涉的。）

1. 如果DLL尚未被映射，那么系统将会强制将该DLL映射到进程B的地址空间中，并将进程B中该DLL的锁计数器递增。
2. 由于DLL的hInstDll是在进程B中映射的，因此系统会对他进行检查，看他与该DLL在进程A中的位置是否相同。如果hInstDll相同，那么这两个进程地址空间中，LowLevelKeyBoardProc函数位于相同的位置，在这种情况下，系统可以直接在进程A的地址空间中调用LowLevelKeyBoardProc。如果hInstDll不同，那么系统必须确定LowLevelKeyBoardProc函数在进程B地址空间中的虚拟地址（仍会调用进程A中的LowLevelKeyboardProc），这个地址通过下面的公式得出 LowLevelKeyboardProc B = hInstDll B + （LowLevelKeyboardProc A  - hInstDll A）;通过把LowLevelKeyboardProc A减去hInstDll A，我们得到LowLevelKeyboardProc的偏移量，以字节为单位，再把这个偏移量与hInstDll B相加就得到LowLevelKeyboardProc在B地址空间中位置。
3. 系统在进程B中递增该DLL的锁计数器
4. 系统在B的地址空间中调用LowLevelKeyboardProc函数
5. 当LowLevelKeyboardProc返回的时候，系统减去DLL在进程B中的锁计数器

注意，当系统把挂钩过滤函数所在的DLL注入或映射到地址空间时，会映射整个DLL，而不仅仅时挂钩过滤函数。这意味着，DLL内所有函数存在于进程B中，能够为进程B中所有线程所调用。

1. API Hook

原理：api hook并不是什么特别不同的hook，它也需要通过基本的hook提高自己的权限，跨越不同进程间访问的限制，达到修改api函数地址的目的。对于自身进程空间下使用到的api函数地址的修改，是不需要用到api hook就可以实现。

我们知道，系统函数都是以dll封装起来的，应用程序应用到系统函数时，应首先把该dll加载到当前的进程空间中，调用的系统函数的入口地址，可以通过GetProcAddress函数进行获取。当系统函数进行调用的时候，首先把所必要的信息保存下来（包括参数和返回地址，等一些别的信息），然后就跳转到函数的入口地址，继续执行。其实函数地址，就是系统函数“可执行代码”的开始地址。那么怎么才能让函数首先执行到我们的函数呢？实际上就是把开始的那段可执行代码替换成我们自己定制的一小段可执行代码，这样系统函数调用时，不就可以按照我们的意图乖乖行事了吗？简单的说，就可以修改系统函数入口的地方，让他调转到我们的函数的入口点就行了。采用汇编代码就能简单实现Jmp XXXX，其中XXXX就是要跳转的相对地址。而Jmp后面要求的是相对偏移，也就是我们的函数入口地址到系统函数入口地址之间的差异，再减去我们这条指令的大小。

用公式表达如下：



为了保持源程序的健壮性，我们的函数里做完必要的处理后，要回调原来的系统函数，然后返回。否则会发生自己调用自己的死循环。

那么说下程序执行过程：

1. 我们的dll“注射”入被hook的进程（这一步只需要安装一个钩子就能实现）
2. 保存系统函数入口处代码
3. 替换掉进程中的系统函数入口并指向我们的函数
4. 当系统函数被调用，立即跳转到我们的函数
5. 我们函数处理
6. 调用原来的系统函数
7. 再修改系统函数入口指向我们的函数（为了下次hook）->返回

来看我们Hook自定义的add函数：

首先我们创建一个AddFunc的dll工程，这个dll只有一个导出函数：

Int WINAPI add(inta, intb);

这个add函数就是我们稍后需要拦截的函数。有了dll后我们就能直接新建一个工程调用add函数