云南大学软件学院期末课程报告

Final Course Report
School of Software, Yunnan University

学　　期: 2019 学年秋季学期

课程名称: 软件安全技术

实践题目: 软件安全保护技术实践

摘要

摘要：本次大作业主要完成了通过 4 个花指令保护、1 个 SMC 保护、6 个反动态调试功能以及基于一机一码的注册码对程序进行了保护。4 个花指令将很多无用的汇编代码插入到程序的关键位置，保护关键 API 函数与关键算法，用来防止对程序的静态分析。SMC 保护的实现原理是通过异或对 MD5 加密的一个关键函数进行保护，通过 SMC pack 将 SMC 区段进行加密，在调用函数时通过 UnPack 函数进行解密，调用完成之后再次使用 UnPack 函数进行加密。6 个反动态调试功能分别使用了父进程检测、调试器进程检测、BeingDebugged 字段检测、时序检测、检测 STARTUPINFO 字段以及硬件断点检测的方式对程序进行反动态调试功能，当程序检测到正在被调试时，就会关闭自身或者崩溃。最后，通过一机一码的注册码保护对软件进行保护。一机一码使用的是 CPU 序列号，注册码生成算法使用的是 MD5 算法、异或、移位等操作对用户输入的用户名进行计算，将最后生成的注册码与用户输入的注册码进行比较。

Absrtact: In this task, the program is protected by four flower instructions protection, one SMC protection, six anti dynamic debugging functions and one machine one code registration code. Four flower instructions insert a lot of useless assembly code into the key position of the program, protect the key API functions and key algorithms to prevent the static analysis of the program. The implementation principle of SMC protection is to protect a key function of MD5 encryption by XOR, encrypt SMC section by SMC pack, decrypt by unpack function when calling function, and use unpack function again after calling. The six anti dynamic debugging functions respectively use the methods of parent process detection, debugger process detection, beingdebugged field detection, timing detection, startup info field detection and hardware breakpoint detection to perform anti dynamic debugging on the program. When the program detects that it is being debugged, it will shut itself down or crash. Finally, the software is protected by the registration code protection of one machine one code. One machine one code uses the CPU serial number, and the registration code generation algorithm uses MD5 algorithm, exclusive or, shift and other operations to calculate the user name entered by the user, and compares the final generated registration code with the registration code entered by the user.

关键字：花指令 SMC 保护 反动态调试 一机一码保护

Key words: Flower instruction, SMC protection, anti dynamic debugging, one machine one code protection

一.软件保护的原理

1.1 软件保护概论

软件保护技术从广义的角度来说，可以包括计算机软件和系统的安全。目前大多数关于计算机的安全研究，主要是研究如何防止合法用户和其数据被恶意客户端程序所攻击，以及如何设计和管理计算机系统来实现一个严密的安全系统。典型的方法是用户限制客户端程序的行为。例如用户可以使用字节码校验来保证不被信任的客户端程序的类型安全，不被信任的代码将被禁止执行一些特定操作，例如可以限制在本地文件系统写文件操作。类似的技术还有软件故障隔离（Software Fault Isolation），它可以监视客户端程序，确保其不能够在它赋予范围之外进行写操作。

软件保护技术从狭义的角度来说，即如何防止合法软件被盗版，主要包括基于硬件的保护方式和基于软件的保护方式。关于软件保护技术的研究，实际上是一项综合的技术。软件保护产品和其他安全产品相比具有一定的特殊性，它所涉及的内容非常的广泛，从上层的应用软件、操作系统、驱动、硬件、网络等等，是一个综合的技术范畴，不能够单一的由某个方面来以偏概全的断定其安全与否。

软件保护又可分为硬件保护和软件保护。

1.1.1 基于硬件的保护方式

基于硬件的软件保护策略可以包含多种功能，主要有认证过程、数据加密、访问控制、唯一的系列号、密钥产生、可靠的数据传输和硬件识别。这些手段主要的目的是防止硬件被复制，有一些产品也支持许可证策略。基于硬件的保护可以具有很好的安全性。主要包括以下几种典型方式：

（1）加密狗、加密锁(Special-purpose Dongles)

加密狗是一种智能性加密产品，又被称为加密锁。它是一个可以安装在并口、串口或USB 接口上的硬件电路。在安全性上和基于软件的保护方式相比，具有更高的安全性，但是对于使用被保护软件的最终用户而言，就不得不被迫接收在自己的机器上安装相应的保护硬件和驱动程序，易用性上存在一定问题。同时和基于软件的保护方式相比，价格也比较高。

（2）光盘、软盘保护

被保护软件的部分密钥可以放在可移动的软盘或光盘当中，只有当软盘或光盘存在的时候，被保护软件才可以运行，游戏软件经常采用此种方式。

其基本原理是，例如 Macrovision SafeDisk 工具，它是在光盘的光轨上隐藏一个密钥，而一般的光盘刻录机无法复制此密钥，通过此方法达到不可以复制光盘的目的，软盘使用的技术类似。

存在问题：如果一旦原盘被划坏或者毁坏，用户就无法继续使用软件，同时这种保护方式可以被黑客很容易的分析或跟踪找到判断代码处，通过修改可执行文件，跳过此段代码，达到破解的目的。而且有的加密光盘可用工作在原始模式（RAW MODE）的光盘拷贝程序来原样拷贝，比如用 Padus 公司的 DiscJuggler 和 Elaborate Bytes 公司的 CloneCD 等拷贝工具，所以此种保护技术的安全性并不是很高，但是由于其具有价格优势，目前还是有一些软件开发商使用此种技术来保护自己的软件。

1.1.2 基于软件的保护方式

基于软件的保护技术和基于硬件的保护技术相比，在价格上具有明显的优势，但是在安全性上和硬件相比还是相差很大，一般正式的商业软件都使用基于硬件的保护方式。基于软件的软件保护方式一般分为：注册码、许可证文件、许可证服务器、应用服务器模式、软件老化等。

（1）注册码 (License Key)

软件开发商对一个唯一串（可能是软件最终用户的相关信息，例如：主机号、网卡号、硬盘序列号、计算机名称等），使用对称或非对称算法以及签名算法等方法产生注册码。然后需要用户进行输入（可以在软件安装过程或单独的注册过程）。当输入注册码后，被保护软件运行时进行解密，并和存储在软件中的原始串进行比较。存在问题：密钥隐藏在程序代码中，比较容易泄漏，同时黑客可以使用逆向工程，分析或跟踪找到判断代码处，通过暴力破解的方法进行破解。

（2）许可证文件(License File)

和使用注册码类似,但是许可证文件可以包含更多的信息,通常是针对用户的一些信息。文件中可以包含试用期时间,以及允许软件使用特定功能的一些信息。被保护软件在运行时,将每次检查许可证文件是否存在。典型的方法是使用非对称算法的私钥对许可证文件进行签名，而公钥嵌在软件代码中。存在问题：可以通过修改系统时钟来延长使用试用期许可证，当许可证到期时，还可以重新安装操作系统，继续使用。同时黑客可以使用逆向工程，分析或跟踪找到判断代码处，通过暴力破解的方法进行破解。

（3）许可证服务器(License Server)

主要适用于网络环境中，可以为多套被保护软件提供服务，例如一个网络许可证，可以限制并发最大用户数为 10。当客户端被保护程序运行时，将占用一个用户数，退出时将释放出用户数，如果超过最大用户数，服务器将禁止多余的被保护程序运行。存在问题：一般必须面向企业级用户，黑客可以使用逆向工程，分析或跟踪找到判断代码处，通过暴力破解的方法进行破解。

（4）应用服务器模式(Application Server Model)

所有程序代码存储在受信任的服务器端，最终用户不需要安装代码。典型应用为最终用户不需要安装软件，只需要使用浏览器访问服务器来使用被保护软件。一般游戏软件都是采用这种方式进行保护的。目前这种保护方式朝着两个方向进行发展，一个是瘦客户端程序，另一个是胖客户端程序。存在问题：此种程序受到服务器性能和网络带宽，以及扩展性，成本等因素的影响。

（5）软件老化(Software Aging)

这是一种极端的软件保护方式，依赖于软件的定期升级更新，每次更新都将使老版本的软件功能不能继续使用，例如不兼容的文件格式。盗版者必须给他的用户经常升级。存在问题：经常升级造成很大的不便，如果可以自动化的进行此项工作，可以节省一部分精力。如果最终用户需要共享数据，将依赖于每个人都有最新版本的软件。这种保护方式并不适用于所有领域，例如：Microsoft Word 可能工作的很好，但是如果是单用户的游戏程序将不适合。

1.2 大作业实现原理

本次大作业运用的主要是基于软件的保护，通过花指令、添加反调试功能以及一机一码的注册码验证方式来保护软件。

1.2.1 花指令保护

实际上，把花指令按照“乱指令”来理解可能更贴切一些，汇编语言其实就是机器指令的符号化，从某种程度上看，它只是更容易理解一点的机器指令而已。每一条汇编语句，在汇编时，都会根据 CPU 特定的指令符号表将汇编指令翻译成二进制代码。应用中，通常通过 VC 的 IDE 或其它如 OD 等反汇编、反编译软件也可以将一个二进制程序反汇编成汇编代码。

机器的一般格式为：指令+数据。而反汇编的大致过程是：首先会确定指令开始的首地址，然后根据这个指令字判断是哪个汇编语句，然后再将后面的数据反汇编出来。由此，我

们可以看到,在反汇编过程中存在漏洞:如果有人故意将错误的机器指令放在了错误的位置，那反汇编时，就有可能连同后面的数据一起错误地反汇编出来，这样，我们看到的就可能是一个错误的反汇编代码，这就是“花指令”。简而言之，花指令是利用了反汇编时单纯根据机器指令字来决定反汇编结果的漏洞。

花指令与免杀的关系加花就时单纯根据机器指令字来决定反汇编结果的漏洞。在程序汇编中，加了一些无用的废话，用来扰乱杀软对特征码的扫描对比，来达到免杀的目的。是一种逃避方式，主要用于表面免杀。加花指令的确没有改变特征码的位置，但是改变了程序执行顺序，有的也能改变文件结构。使杀毒软件扫描的时候跳到花指令处，即判断没有病毒。通常情况下，在文件免杀的时候，加花指令是最简单、有效的方法，而且一般能通杀很多杀毒软件，所以一般文件免杀通用此法。

花指令执行顺序花指令一般添加到程序的头部。执行顺序：花指令入口-->执行花指令-->程序原入口-->执行原程序。花指令的好坏直接决定程序是否可以躲避杀毒软件的查杀，花指令和加壳的本质差不多，都是为了保护程序而做，所以我们做免杀的时候，可以多结合壳和花指令各自的优点对程序进行处理，达到更好的免杀效果。

1.2.2 SMC 保护

自修改代码（Self-modifying Code，SMC）是一类代码的统称，该类代码中加入了自修改保护机制，在执行过程中通过修改自身代码，使实际运行的代码和运行之前的静态二进制代码不同，以隐藏程序的关键代码。该技术需要直接对内存中的机器码进行操作，具有一定的技术难度。

自修改代码技术开始用以减少程序的内存占用和提高指令的执行效率,到后来演变成用于软件保护。该技术的基本思路是:在代码中嵌入自修改保护机制以隐藏关键代码,增加攻击者对代码理解难度，在代码运行之前，伪装的目标指令 r,运行代码时，替换成指令 q,最终执行的指令 q。目前，主流的 SMC 技术主要有 3 类。

（1）Y.Kanzaki 的 SMC 软件保护技术

Y. Kanzaki 等人在 2006 年提出基于指令伪装的自修改保护方法，并分析了该软件保护方法在抵抗静态分析和动态分析方面的性能。指令伪装的过程如下:首先对软件通过反汇编得到汇编代码，然后在汇编代码中选取一条或者一段要加密的目标指令 O，接下来用 O’替换原先的指令 O,再确定加、解密代码 E 和 D,通过计算程序的执行流程确定 E 和 D 的插入位置并进行插入,最后重新生成新的二进制代码。其中 O’为随机生成的伪装指令，可以确保目标指令被各种不同的指令所覆盖。该方法也存在以下不足:①依赖于反汇编工具，如果反汇编结果出错，极可能会导致软件无法使用:②需要对软件的执行流程有清晰的了解，如果程序规模大,就会变成一个复杂的过程;③主要通过 MOV 指令对内存进行修改,大量 MOV 指令的出现容易引起攻击者的注意。

（2）Jan Cappaert 的基于 SMC 的自加密技术

Jan Cappaert, Nessim Kisserli 等人在 2006 年提出了--种基于 SMC 的自加密技术，该方法的核心是通过 SMC 技术实现代码的加密。首先列出程序中函数相互调用图,对函数 A 进行加密，加密的密钥 K 则来自于代码 B 的散列值，称为 A 依赖于 B;然后对除入口函数外的所有函数加密，当调用 A 时，先计算 A 所依赖函数的散列值获得密钥归再调用解密函数 E 解密 A,同时，再在函数返回之前重新加密。该方法能够有效地抗逆向分析和防止软件的非法篡改,但同时也存在一些不足:①需要对程序的结构有很好的了解;②程序的性能受到各函数间相互依赖关系的影响，特别是在递归调用，循环调用中将会对程序的性能造成很大的影响。

（3）其他 SMC 技术

中国科技大学的王祥根,司端锋等人在2009年提出的SMC技术通过将关键代码转换为数

据存储在原程序中，以隐藏关键代码;受保护的可执行文件在执行过程中，修改进程中存储有隐藏代码的虚拟内存页面属性为可执行,实现数据到可执行代码的转换，此方法简单，易实现，可以有效提高 SMC 的抗逆向能力。同时也存在程序执行时攻击者可以通过跟踪发现隐藏的关键代码、抗非法篡改能力差等不足。电子科技大学的任云韬提出基于二进制多台变形的恶意代码反检测方法，多态恶意代码能够对每一个变种的解密代码进行变化，主要优点是随机性和变化性，但此方法只对解密模块进行多态处理，易被分析者定位并进行有效分析。

基于 SMC 的软件保护机制，能够对软件实施有效地保护，主要有两个特点：

（1）机密性:程序在静态时，关键代码是以密文形式存储;而当程序载入内存运行时，受保护的代码块分布在程序的各个部分，只在调用前解密，调用返回后到下一次调用前重新加密，所以任何时刻大部分受保护代码都是以密文形式驻于内存，具有良好的机密性。

(2)防篡改:第一层的白修改程序 Cipher 在恢复 Decrypt 函数时,是以程序其余代码的校验值作为密钥。攻击者如果修改 Cipher,将会恢复出错误的代码;如果修改程序的其余代码，则会计算出错误的校验值，必然会恢复出错误的指令或者数据,程序的执行将会出错甚至崩溃。所以该保护机制具备一定的防篡改能力。

1.2.3 反调试功能

反调试技术，恶意代码用它识别是否被调试，或者让调试器失效。恶意代码编写者意识到分析人员经常使用调试器来观察恶意代码的操作,因此他们使用反调试技术尽可能地延长恶意代码的分析时间。为了阻止调试器的分析，当恶意代码意识到自己被调试时，它们可能改变正常的执行路径或者修改自身程序让自己崩溃，从而增加调试时间和复杂度。很多种反调试技术可以达到反调试效果。

反调试技术大致分为四大类，探测 Windows 调试器、识别调试器的行为、干扰调试器的功能以及调试器的漏洞。

（1） 探测 Windows 调试器

很多代码会使用探测调试器调试它的痕迹，其中包括 Windows API、手动检测调试器人工痕迹的内存结构，查询调试器遗留在系统中的痕迹等。

使用 Windows API 函数探测调试器是否存在是最简单的反调试技术。Windows 操作系统中提供了这样一些 API，应用程序可以通过调用这些 API 来探测自己是否正在被调试。

1.IsDebuggerPresent：探测调试器是否存在的最简单的 API 函数是 IsDebuggerPresent。它查询进程环境块中的 IsDebugged 标志。

2.CheckRemoteDebuggerPresent：检测本地机器中的一个进程是否运行在调试器中，而不是检测远程主机的调试器。同时，它也检查 PEB 结构中的 IsDebugged 属性。它不仅可以探测进程自身是否被调试，同时可以探测系统其他进程是否被调试。

3.NtQueryInformationProcess：这个函数是 Ntdll.dll 中的一个 API，用来提取一个给定进程的信息。将该函数的第二个参数设置为 0x07，它将会告诉你这个句柄标识的进程是否正在被调试。

4.OutputDebugString：这个函数的作用是在调试器中显示一个字符串，同时也可以探测调试器的存在。

使用 Windows API 是探测调试器存在的最简单方法，但很多时候可以手动检查。手动检测中，PEB 结构中的一些标志暴露了调试器存在的信息。比如 BeingDebugged 属性，ProcessHeap 属性和 NtGlobalFlag 属性等。

同时，使用调试工具分析代码的时候，这些工具通常会在系统中驻留一些痕迹，搜索调试器引用的注册表项，就可以判断程序是否正在被调试。

（2） 识别调试器的行为

在逆向工程中，为了帮助代码分析人员进行分析，调试器通常会有设置断点的功能，或

者单步执行的功能。然而，当调试器执行这些操作时，它们会修改进程中的代码，因此使用INT 扫描、完整性校验以及时钟检测等方式识别调试器的行为就可以判断程序是否正在被调试。

1.INT 扫描：调试器设置断点的机制是软件中断命令 INT 3，临时替换运行程序中的一条指令,然后当程序运行到这条指令时,调用调试异常处理例程。INT 3 指令的机器码是 0xCC，因此无论何时，使用调试器设置断点时，它都会插入一个 0xCC 来修改代码。因此通过在程序代码中查找机器码 0xCC，就可以证明调试器的存在。

2.执行代码校验和检查：程序可以计算代码段的校验并实现与扫描中断相同的目的。与扫描 0xCC 不同，这种检查仅执行恶意代码中机器码的 CRC 或者 MD5 校验和检查。

3.时钟检测：程序被调试时，进程的运行速度会大幅度降低，因此使用时钟检测是程序探测调试器存在的有效方法。有两种使用时钟检测来探测调试器存在的方法，一是记录执行一段操作前后的时间戳，然后比较这两个时间戳，如果存在之后则可以认为存在调试器；二是记录触发一个异常前后的时间戳，如果不调试进程，可以很快地处理完异常，因为调试器处理异常的速度非常慢。常用的时钟检测方法是利用 rdtsc 指令，它返回至系统重新启动以来的时钟数，并将其作为一个 64 位的值存入 EDX:EAX 中。恶意代码运行两次 rdtsc 指令，然后比较两次读取之间的差值。另一种常用的方法是使用 Windows API 函数中的 QueryPerformanceCounter 函数和 GetTickCount 函数来执行一个反调试的时钟检测。

（3） 干扰调试器的功能

程序还可以利用本地存储回调、异常、插入中断等技术来干扰调试器的功能，这些技术当且仅当程序处于调试器控制之下时才会视图扰乱程序的运行。

（4） 调试器的漏洞

同所有软件一样，调试器也存在漏洞。有时程序的编写者为了防止被调试，会攻击调试器的漏洞。OllyDbg 会严格遵守微软对于 PE 文件头部的规定，因此当修改二进制文件的 PE 头后,OllyDbg 调试器加载修改后的二进制文件时,就会导致 OllyDbg 的崩溃,返回错误“Bad or Unknown 32-bit Executable File”，而程序在调试器外运行时会一切正常。

1.2.4 一机一码注册码保护

使用注册码方式软件保护的目的是向合法用户提供完整的功能,所以软件保护必然要包括验证用户合法性的环节，而这一环节通常采用注册码验证的方式实现，步骤如下：

① 用户向软件作者提交用户码 U，申请注册；

② 软件作者计算出注册码 R=f(U)，将结果返回合法用户；

③ 用户在软件注册页面输入 U 和 R；

④ 软件通过验证 F(U,R)的值是否合法来判定用户的合法性。

所谓计算机软件注册码是指为了不受限制地实现计算机软件的功能,而在软件安装或使用的过程中，按照指定的要求所输入的、由字母、数字或其它符号所组成的序列，因此，注册码有时又可称为序列号，只是在特定的条件下，两者会有所区别。之所以要对计算机软件设置注册码，开发者的初衷在于防止用户使用盗版软件，至今仍有部分软件注册码在发挥着这样的功能,最为典型的就是安装型注册码,即在软件安装过程中按要求必须输入的注册码，如果没有正确输入注册码，则软件根本不能安装到计算机中去。但是现在已经有了愈来愈多的软件注册码并非是对软件安装的限制，而是对软件其它方面的限制，比如，如果不输入正确的注册码，虽然可以安装并使用，但不能实现软件的全部功能等。之所以要这样设计，主要是因为有很多新开发出来的软件，人们对其功能并不了解，通过这种让用户先尝一点“甜头”的做法，使公众对该软件的基本功能有所认识，并有可能刺激其消费欲，如果用户要想不受限制地实现软件的全部功能，则要花钱向软件开发商购买注册码。因而可以这么说，现在很多软件设计者为了实现其经济利益，其向社会公众出售的不再是其所设计的软件本身，

而是该软件的注册码。软件开发者往往并不限制对软件本身的随意复制、传播和使用，相反，他们还会充分利用络这种便利的传播媒体，来扩大对自己软件的宣传。对他们来说，自己所开发的软件传播的范围越广越好，使用的人越多越好，而这些行为并不必要经过开发者的授权同意，因为他们要通过人们购买软件的注册码来获得收益，而不是通过出售软件本身来获得收益。而这一点正是其与一般的商品买卖的不同之处，因为注册码只是一串没有包含任何价值的符号，但由于其对软件的使用者来说具有无比重大的意义，因此在这里却成了买卖的对象。尽管我们可以从深远一点的意义上说，购买者所购买的还是软件开发者的智力成果，即软件本身，但却无可否认这种买卖的直接标的就是注册码，而非计算机软件。因此，正是从这个意义上说，计算机软件注册码很多情况下已经不是软件的组成部分，计算机软件注册码的侵权也不同于计算机软件侵权。

二.程序实现

2.1 花指令实现

2.1.1 设计方案

在我们定义了四套花指令宏嵌入程序内，在程序的关键处调用该花指令，即可实现花指令保护。

2.1.2 源码分析

第一套花指令宏如下所示。

1._FLOWER_XX00：

```
#define _FLOWER_XX00 _asm{\
__asm jz  $+0xe  \                //Jz 和 jnz 执行一条，跳到当前地址+0xe 或+8 的位置
__asm jnz $+8 \
__asm _emit 0xe8 \                 //插入 e8，混淆机器指令
__asm _emit -0xe8 \
__asm push ebp \
__asm mov ebp,esp \
__asm push -1 \                   //向堆栈中放入数据进行混淆
__asm push 666666 \
__asm push 888888 \
__asm mov eax,dword ptr fs:[0] \    //fs:[0]数据放入 EAX，将程序伪装为 vc++5.0
__asm nop \
__asm mov dword ptr fs:[0],esp \    // ESP 数据放入 fs:[0]
__asm nop \
__asm mov dword ptr fs:[0],eax \    // fs:[0]数据还原
__asm pop eax \                   //以下的 pop 指令用于控制堆栈平衡
__asm pop eax \
__asm nop \
__asm pop eax \
__asm pop eax \
__asm nop \
__asm mov ebp,eax \              //还原 ebp
}
```

第二套花指令宏如下所示。

2._FLOWER_XX01：

```
#define _FLOWER_XX01 __asm{\
__asm call l1 \                    //执行 l1 处指令
__asm l1: \                        //l1
__asm pop eax \                   //出栈，值放入 EAX
__asm call f1 \                    //执行 f1 处指令
```

```
__asm _EMIT 0xEA \              //插入 EA，混淆机器指令
__asm jmp l2 \                  //跳转至 l2 处
__asm f1 : \                     //f1
__asm pop ebx \                 //出栈，值放入 EBX
__asm inc ebx \                  //EBX 值加 1
__asm push ebx \                //入栈
__asm mov eax, 0x11111111 \     //改变 EAX 值为 0x11111111
__asm ret \                     //返回
__asm l2 : \                     //l2
__asm call f2 \                  //执行 f2 处指令
__asm mov ebx, 0x33333333 \    //改变 EBX 值为 0x33333333
__asm jmp e \                   //跳转至 e 处
__asm f2 : \                     //f2
__asm mov ebx, 0x11111111 \    //改变 EBX 值为 0x11111111
__asm pop ebx \                 //出栈，值放入 EBX
__asm mov ebx, offset e \        //改变 EBX 值为 e 的首地址
__asm push ebx \                //入栈
__asm ret \                     //返回
__asm e : \                     //e
__asm mov ebx, 0x22222222 \   //改变 EBX 值为 0x22222222
}
```

第三套花指令宏如下所示。

3._FLOWER_XXO2：

```
#define _FLOWER_XXO2 __asm{\
__asm nop \
__asm push eax \          //插入具有含义的汇编指令，混淆函数执行流程
__asm inc eax \
__asm push edx \
__asm nop \
__asm pop eax \
__asm dec eax \
__asm pop edx \
__asm nop \
__asm jz $+0x15 \           //实现跳转
__asm jnz $+0xf \
__asm _emit 0xe8 \          //添加虚假的机器指令
__asm _emit 0x57 \
__asm _emit 0x74 \
__asm _emit 0x58 \
__asm _emit 0x22 \
__asm _emit 0x90 \
__asm _emit 0x90 \
```

```
__asm _emit 0x60 \
__asm _emit 0x77 \
}
```

第四套花指令宏如下所示：

4._FLOWER_XX03：

```
#define _FLOWER_XX03 __asm{\
__asm jz  $+0xe  \        //Jz 和 jnz 执行一条，跳到当前地址+0xe 或+8 的位置
__asm jnz $+8 \
__asm _emit 0xe8 \         //插入 e8，混淆机器指令
__asm _emit -0xe8 \
__asm push ebp \          //插入具有一定含义的汇编指令，混淆函数执行流程
__asm mov ebp,esp \
__asm inc ecx \
__asm push edx \
__asm pop edx \
__asm dec ecx \
__asm pop ebp \
__asm inc ecx \
}
```

然后在程序中的关键位置调用这四套花指令宏，如图 2.1、图 2.2、图 2.3 所示。

```
else
{
    _FLOWER_XX00;
    if(GenRegCode(cCode, cName ,len))    //此处调用序列号计算的子程序
    {   lstrcpy(szBuffer,szSucc);
        _FLOWER_XX03;
        EnableWindow(GetDlgItem(hDlg,IDC_TXT0),FALSE);
        EnableWindow(GetDlgItem(hDlg,IDC_TXT1),FALSE);
        _FLOWER_XX02;
    }

    else
        lstrcpy(szBuffer,szFail);
SetFocus (GetDlgItem(hDlg,IDC_TXT1));
}
_FLOWER_XX03;
MessageBeep (MB_OK);
_FLOWER_XX02;
DialogBox (hInst, MAKEINTRESOURCE (IDD_CHECK), hDlg, CheckDlgProc ) ;
```

图 2.1 调用花指令

```
TCHAR cName[MAXINPUTLEN]={0};
TCHAR cCode[100]={0};
int len;
TCHAR szEnchar[] = TEXT ("你输入字符要大于四个！") ;
_FLOWER_XX00;
TCHAR szSucc[] = TEXT ("注册成功！") ;
_FLOWER_XX02;
TCHAR szFail[] = TEXT ("序列号错误！") ;
_FLOWER_XX03;
```

图 2.2 调用花指令

```
_FLOWER_XX00;
for (i = 0, j = 0; j < 10; i++, j++)
{
    _FLOWER_XX02;
    if (i > len - 1)
    {
        _FLOWER_XX00;
        i = 0;
        _FLOWER_XX00;
    }
    _FLOWER_XX03;
    code += ((BYTE)name[i]) ^ Table[j];
}
code = 2017 * code;
x=MD5((char)code);
code = x * code;
code = code + 58;
code = code + 69;
code = code + 83;
code = code + 107;

code = code * code;

    _FLOWER_XX02;
    wsprintf(name,TEXT("%ld"),code);
    _FLOWER_XX01;
    if(lstrcmp(rCode, name)==0)
        return TRUE;
```

图 2.3 调用花指令

至此，调用花指令宏完毕，可以实现使用花指令保护程序。

2.2 SMC 实现

2.2.1 设计方案

SMC 保护的实现原理是通过异或算法对 MD5 加密代码的一个关键函数进行保护，通过 SMC pack 将 SMC 区段进行加密，在调用函数时通过 UnPack 函数进行解密，调用完成之后再次使用 UnPack 函数进行加密。

2.2.2 源码分析

SMC 保护的函数如下所示。该函数为 MD5 算法内的一个移位函数 move。

```
/*左移函数*/
#pragma code_seg(".SMC")
void move(int step, int* temp1, int* temp2)
{
    int i;
    for (i = 0; i < 32 - step; i++)
        temp2[i] = temp1[i + step];
    for (i = 0; i < step; i++)
        temp2[32 - step + i] = temp1[i];
}

#pragma code_seg()
#pragma comment(linker, "/SECTION:.SMC,ERW")
```

SMC 加密代码具体实现如下所示。其中包括 xorPlus 函数、SMC 函数和 UnPack 函数，通过这三个函数对 move 函数进行加密。

```
void xorPlus(char* soure, int dLen, char* Key, int Klen)
{
    if (!Klen)
        return;
    for (int i = 0; i < dLen;)
    {
        for (int j = 0; (j < Klen) && (i < dLen); j++, i++)
        {
            soure[i] = soure[i] ^ Key[j];
            soure[i] = ~soure[i];
        }
    }//
}
void SMC(char* pBuf, char* key)
{
    // SMC 加密 XX 区段
    const char* szSecName = ".SMC";
    short nSec;
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS pNtHeader;
```

```
        PIMAGE_SECTION_HEADER pSec;
        pDosHeader = (PIMAGE_DOS_HEADER)pBuf;
        pNtHeader = (PIMAGE_NT_HEADERS)& pBuf[pDosHeader->e_lfanew];
        nSec = pNtHeader->FileHeader.NumberOfSections;
        pSec   =   (PIMAGE_SECTION_HEADER)&   pBuf[sizeof(IMAGE_NT_HEADERS)   +
pDosHeader->e_lfanew];
        for (int i = 0; i < nSec; i++)
        {
            if (strcmp((char*)& pSec->Name, szSecName) == 0)
            {
                int pack_size;
                char* packStart;
                pack_size = pSec->SizeOfRawData;
                packStart = &pBuf[pSec->VirtualAddress];

    //VirtualProtect(packStart,pack_size,PAGE_EXECUTE_READWRITE,&old);
                xorPlus(packStart, pack_size, key, strlen(key));
                //MessageBox(_T("SMC 解密成功。"));
                //MessageBox(NULL, TEXT("解密成功"), TEXT("1"), MB_OK);
                return;
            }
            pSec++;
        }
    }

    void UnPack()
    {
        char* Key = "12345";
        char* hMod;
        hMod = (char*)GetModuleHandle(0);
        SMC(hMod, Key);
    }
```

2.3 反调试实现

2.3.1 设计方案

一共使用了 6 种反调试方法，在源程序中的对应函数分别为 CheckDebug1-6，下面详细介绍各函数原理与 OD 运行的检测部分。

2.3.2 源码分析

1.枚举进程列表

查看任务管理器内是否存在调试器进程（如 OllyDBG.EXE,x64_dbg.exe 等）。利用 kernel32!ReadProcessMemory()函数读取进程内存，然后寻找调试器相关的字符串（如”OLLYDBG”）以防止逆向分析人员修改调试器的可执行文件名。

```
    BOOL CheckDebug1()//调试器进程检测
```

```
    {
        DWORD ID;
        DWORD ret = 0;
        PROCESSENTRY32 pe32;
        pe32.dwSize = sizeof(pe32);
        HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
        if (hProcessSnap == INVALID_HANDLE_VALUE)
        {
            return FALSE;
        }
        BOOL bMore = Process32First(hProcessSnap, &pe32);
        while (bMore)
        {
            if     (stricmp(pe32.szExeFile,     "OllyDBG.EXE")     ==     0     ||
stricmp(pe32.szExeFile, "吾爱破解[LCG].exe") == 0 || stricmp(pe32.szExeFile,
"x64_dbg.exe")  ==  0  ||  stricmp(pe32.szExeFile,  "windbg.exe")  ==  0  ||
stricmp(pe32.szExeFile, "ImmunityDebugger.exe") == 0)
            {
               MessageBox(NULL, TEXT("发现OD--反调试1"), TEXT("Debug Warning!"),
MB_OK);
                system("Pause");
                exit(-1);
            }
            bMore = Process32Next(hProcessSnap, &pe32);
        }
        CloseHandle(hProcessSnap);
        return FALSE;
    }
```

在使用带插件的 OD 进行调试时，立即被检测出来，弹出对话框后退出程序，实际保护中可以不弹框，或者隐藏字符串，为方便演示和检测反调试效果故弹出对话框，如图 2.4 所示。

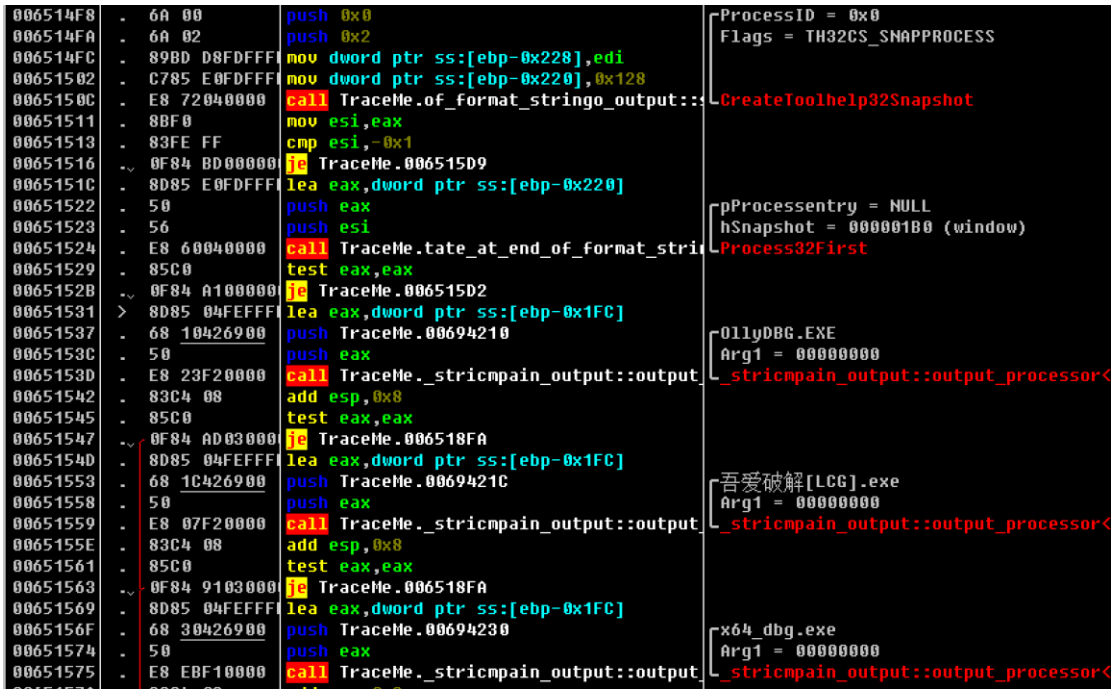

图 2.4 枚举进程列表

2.父进程检测

实现原理：通常在双击执行程序的情况下，进程的父进程是 explorer.exe。因此，如果进程的父进程不是 explorer.exe，则程序可能正在被调试。通过枚举窗口名称发现调试器，从而提前中止程序，并弹出对话框，如图 2.5 所示。

```
BOOL CheckDebug2()//父进程检测
{
    if (FindWindowA("OllyDbg", NULL) != NULL || FindWindowA("WinDbgFrameClass", NULL) != NULL || FindWindowA("吾爱破解[LCG]", NULL) != NULL)
    {
        MessageBox(NULL, TEXT("发现 OD--反调试 2"), TEXT("Debug Warning!"), MB_OK);
        system("Pause");
        exit(-1);
    }
    else
    {
        return FALSE;
    }
}
```

可检测出原版 OD，在插件保护下的 OD 不能被发现。

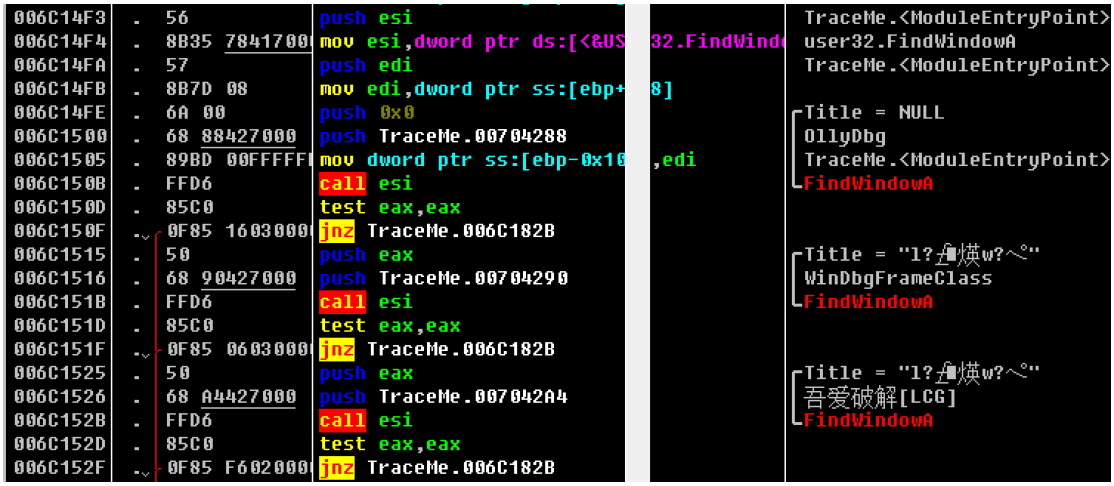

图 2.5 父进程检测

3.检测 BeingDebugged 字段

实现原理：调用 isDebuggerPresent、CheckRemoteDebuggerPresent 两个 API 函数检测 BeingDebugged 字段，实现反调试。kernel32!IsDebuggerPresent() API 检测进程环境块(PEB)中的 BeingDebugged 标志检查这个标志以确定进程是否正在被用户模式的调试器调试，如图 2.6 所示。

```
BOOL CheckDebug3()//BeingDebugged 字段检测
{
    BOOL isDebuggerPresent = false;
    if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPresent))
    {
        if (isDebuggerPresent)
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 3"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
    }
}
```

可检测出原版 OD，在插件保护下的 OD 不能被发现。

图 2.6 检测 BeingDebugged

4.检查时序 GetTickCount

实现原理：当进程被调试时，调试器事件处理代码、步过指令等将占用 CPU 循环 。如果相邻指令之间所花费的时间如果大大超出常规，就意味着进程很可能是在被调试，如图 2.7 所示。

```
BOOL CheckDebug4()//利用时间判断
{
    DWORD time1 = GetTickCount();
    __asm
    {
        mov     ecx, 10
        mov     edx, 6
        mov     ecx, 10
    }
    DWORD time2 = GetTickCount();
    if (time2 - time1 > 0xA)
    {
        MessageBox(NULL, TEXT("发现 OD--反调试 4"), TEXT("Debug Warning!"),
MB_OK);
        system("Pause");
        exit(-1);
    }
    else
    {
        return FALSE;
    }
}
```

弱保护，能通过带插件的 OD，但在不同电脑上不一定能发现 OD，在某些有时间处理的程序中不易被调试出来。

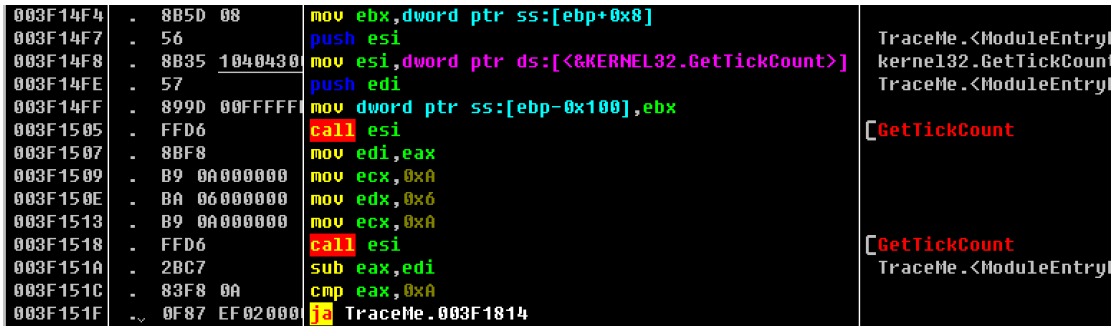

图 2.7 检查时序

5.检测 STARTUPINFO 字段

实现原理：Windows 操作系统中的 explorer.exe 创建进程的时候会把 STARTUPINFO 结构中的值设为 0,而非 explorer.exe 创建进程的时候会忽略这个结构中的值，也就是结构中的值不为 0，所以可以利用这个来判断 OD 是否在调试程序，如图 2.8 所示。

```
BOOL CheckDebug5()//系统打开程序下列值为 0，OD 打开则不为 0
{
    STARTUPINFO si;
    GetStartupInfo(&si);
    if (si.dwX != 0 || si.dwY != 0 || si.dwFillAttribute != 0 || si.dwXSize !=
0 || si.dwYSize != 0 || si.dwXCountChars != 0 || si.dwYCountChars != 0)
```

```
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 5"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
        else
        {
            return FALSE;
        }
    }
```

可检测出原版 OD，在插件保护下的 OD 不能被发现。

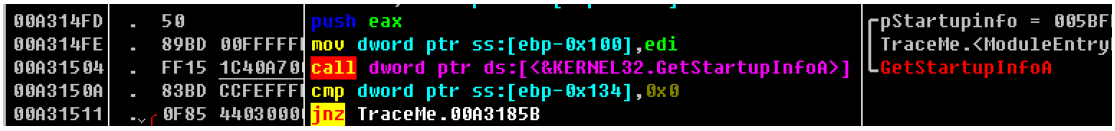

图 2.8 检测 STARTUPINFO

6.检测硬件断点

实现原理：硬件断点是通过设置名为 Dr0 到 Dr7 的调试寄存器来实现的。Dr0-Dr3 包含至多 4 个断点的地址 ，Dr6 是个标志，它指示哪个断点被触发了，Dr7 包含了控制 4 个硬件断点诸如启用/禁用或者中断于读/写的标志。由于调试寄存器无法在 Ring3 下访问 ，硬件断点的检测需要执行一小段代码。可以利用含有调试寄存器值的 CONTEXT 结构 ，该结构可以通过传递给异常处理例程的 ContextRecord 参数来访问，如图 2.9 所示。

```
    BOOL CheckDebug6()//硬件断点检测
    {
        CONTEXT context;
        HANDLE hThread = GetCurrentThread();
        context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
        GetThreadContext(hThread, &context);
        if (context.Dr0 != 0 || context.Dr1 != 0 || context.Dr2 != 0 || context.Dr3 !=
0)
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 6"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
        return FALSE;
    }
```

能通过有插件 OD 的反调试，下硬件断点即被检测。

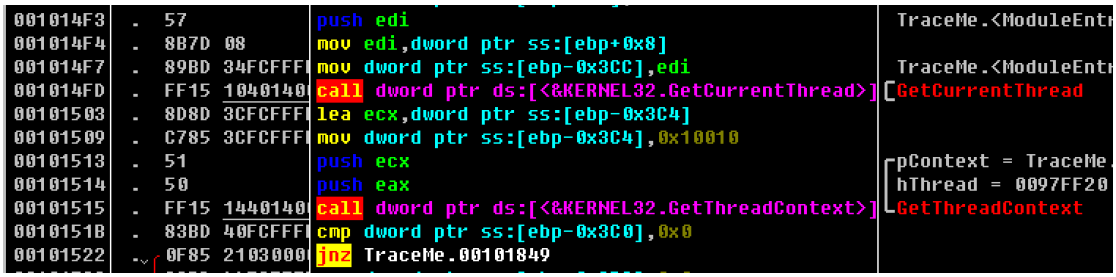

图 2.9 检测硬件断点

2.4 注册码实现

2.4.1 设计方案

因为需要实现一机一码的注册码，因此需要获取用户电脑的唯一序号，这里采用获取用户电脑的 CPU 序列号，然后将用户输入的用户名与本机的 CPU 序列号进行运算，得出特定的注册码。计算序列号的具体算法为：首先使用 CPU 序列号与用户输入的用户名进行异或运算，然后将异或运算后的结果使用 MD5 进行加密

2.4.2 源码分析

注册码算法具体代码 GenRegCode 函数如下：

```
BOOL GenRegCode( TCHAR  *rCode, TCHAR  *name ,int len)
{
int i,j;
long code=0;
char Table[10];
long x=0;
::GetVolumeInformation("C:\\", namebuf, namesize, &serialnumber, &maxlen,
&fileflag, sysnamebuf, sysnamesize);
itoa(serialnumber, Table, 10);
for (i = 0, j = 0; j < 10; i++, j++)
{
    if (i > len - 1)
    {
        i = 0;
    }
    code += ((BYTE)name[i]) ^ Table[j];
}
code = 2017 * code;
x=MD5((char)code);
code = x * code;
code = code + 58;
code = code + 69;
code = code + 83;
code = code + 107;

code = code * code;
```

```
        wsprintf(name,TEXT("%ld"),code);
        if(lstrcmp(rCode, name)==0)
            return TRUE;
        else
        {
                return FALSE;
        }
    }
```

其中，GenRegCode 函数需要接收用户输入的用户名和注册码，然后通过 GetVolumeInformation 函数获取 CPU 的序列号，然后将 CPU 以十进制的形式保存在 Table 中。将用户输入的用户名与 CPU 序列号进行异或运算，运算的结果进行 MD5 加密，因为 MD5 加密后为固定的 32 位，对于用户输入用户名来说太长，因此取加密后的前 8 位保存在 code 中。然后将 code 进行加减乘除运算，加入组员学号的信息，再将 code 以字符串的形式传入 name 中。然后将 name 与 rCode 进行比较，若相同，则注册成功，反之则注册失败。如图 2.10、图 2.11 所示。

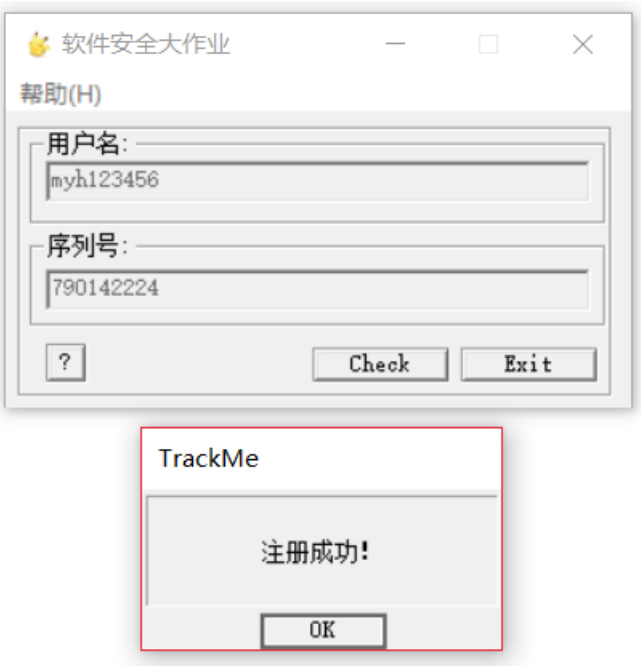

图 2.10 注册成功

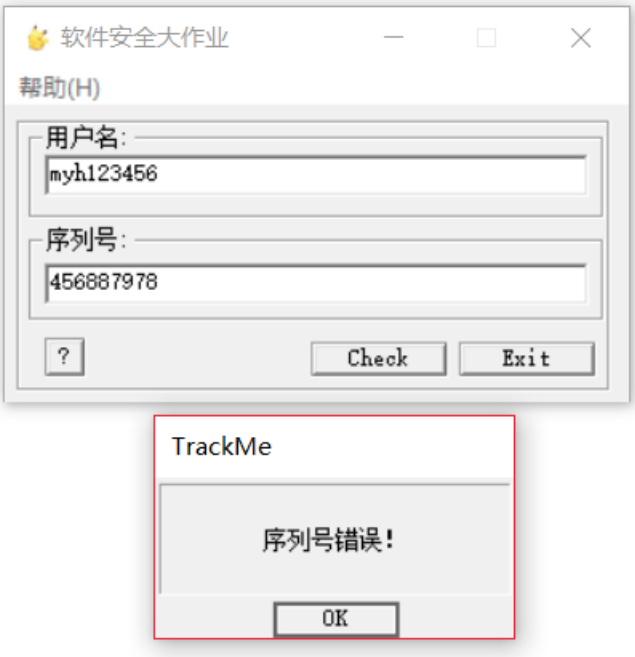

图 2.11 注册失败

此外，本小组还针对该程序编写了一个注册机软件。使用与原程序相同的加密算法，通过输入的用户名计算出序列号，如图 2.12 所示。

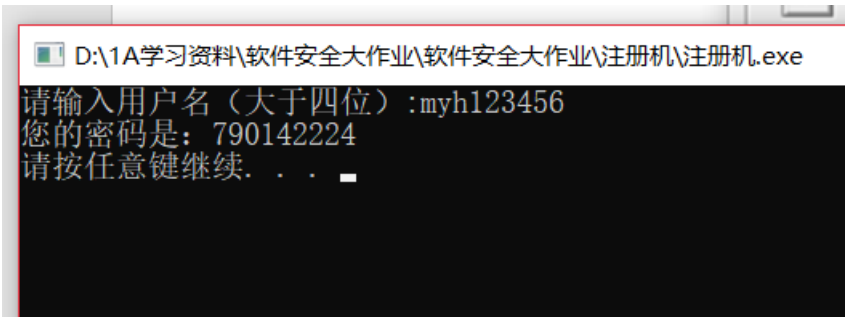

图 2.12 注册机

三. 程序功能测试

本次大作业实现的具体功能为通过用户输入用户名和注册码判断是否注册成功。其中加入花指令保护、SMC 保护和反动态调试，程序的主页面如图 3.1 所示。

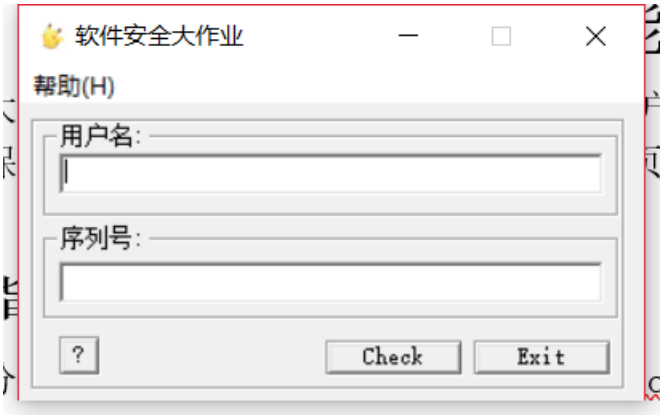

图 3.1 程序主页面

程序的帮助页面为本小组四人的学号姓名信息，如图 3.2 所示。

图 3.2 本小组成员信息

3.1 花指令测试

首先分析第一套花指令对 GetVolumeInformation 函数进行保护为例进行分析，如图所示。将该花指令插在 GetVolumeInformation 函数的前面，通过 IDA 静态分析加花指令前后的区别，如图 3.2 所示。

```
_FLOWER_XX00;
::GetVolumeInformation("C:\\", namebuf, namesize, &serialnumber, &maxlen, &fileflag, sysnamebuf, sysnamesize);
itoa(serialnumber, Table, 10);
```

图 3.2 对 GetVolumeInformation 函数进行保护

IDA 对比分析：

加花指令前，在 IDA 中可以清晰地看到 GetVolumeInformation 函数的调用位置以及参数位置。如图 3.3 所示。

```
.text:00401110                push    ebp
.text:00401111                mov     ebp, esp
.text:00401113                sub     esp, 18h
.text:00401116                mov     eax, ___security_cookie
.text:0040111B                xor     eax, ebp
.text:0040111D                mov     [ebp+var_4], eax
.text:00401120                push    ebx
.text:00401121                mov     eax, [ebp+arg_0]
.text:00401124                push    esi
.text:00401125                mov     ebx, [ebp+arg_4]
.text:00401128                push    edi
.text:00401129                push    nFileSystemNameSize ; nFileSystemNameSize
.text:0040112F                mov     [ebp+lpString1], eax
.text:00401132                xor     edi, edi
.text:00401134                push    lpFileSystemNameBuffer ; lpFileSystemNameBuffer
.text:0040113A                push    offset FileSystemFlags ; lpFileSystemFlags
.text:0040113F                push    offset MaximumComponentLength ; lpMaximumComponentLength
.text:00401144                push    offset VolumeSerialNumber ; lpVolumeSerialNumber
.text:00401149                push    nVolumeNameSize ; nVolumeNameSize
.text:0040114F                push    lpVolumeNameBuffer ; lpVolumeNameBuffer
.text:00401155                push    offset RootPathName ; "C:\\"
.text:0040115A                call    ds:GetVolumeInformationA
.text:00401160                push    0Ah
.text:00401162                lea     eax, [ebp+var_10]
```

图 3.3 加花前 IDA 函数调用分析

加花指令后：在 TEXT 页面发现了花指令中带有的指令，添加了错误的数据入栈，函数的参数被成功混淆，无法判断调用函数之前的语句的作用以及数据的含义。添加花指令后，无法显示当前的图形界面，如图 3.4 所示。

```
.text:0040112A                jz      near ptr loc_401136+2
.text:00401130                jnz     near ptr loc_401136+2
.text:00401136
.text:00401136 loc_401136:                            ; CODE XREF: .text:0040112A
.text:00401136                                        ; .text:00401130↑j
.text:00401136                call    near ptr 0ECCB6653h
.text:0040113B                push    0FFFFFFFFh
.text:0040113D                push    0A2C2Ah
.text:00401142                push    0D9038h
.text:00401147                mov     eax, large fs:0
.text:0040114D                nop
.text:0040114E                mov     large fs:0, esp
.text:00401155                nop
.text:00401156                mov     large fs:0, eax
.text:0040115C                pop     eax
.text:0040115D                pop     eax
.text:0040115E                nop
.text:0040115F                pop     eax
.text:00401160                pop     eax
.text:00401161                nop
.text:00401162                mov     ebp, eax
.text:00401164                mov     eax, dword_449004
.text:00401169                push    eax
.text:0040116A                mov     ecx, dword_449A4C
.text:00401170                push    ecx
.text:00401171                push    offset unk_449A44
.text:00401176                push    offset unk_449A40
.text:0040117B                push    offset dword_449A3C
.text:00401180                mov     edx, dword_449000
.text:00401186                push    edx
.text:00401187                mov     eax, dword_449A48
.text:0040118C                push    eax
.text:0040118D                push    offset unk_440228
.text:00401192                call    ds:GetVolumeInformationA
```

图 3.4 加花后函数调用分析

分析第二套花指令对字符串比较判断进行保护为例进行分析，如图 3.5 所示。

```
_FLOWER_XX02;
wsprintf(name,TEXT("%1d"),code);
_FLOWER_XX01;
if(lstrcmp(rCode, name)==0)
    return TRUE;
else
    return FALSE;
```

图 3.5 对字符串比较判断进行保护

IDA 对比分析：

加花指令前，在 IDA 中可以清晰地看到 IstrcmpA 函数的调用位置以及参数保存位置以及函数流程，如图 3.6 所示。

```
.text:004011A0                push    edi
.text:004011A1                push    offset aLd      ; "%ld"
.text:004011A6                push    ebx             ; LPSTR
.text:004011A7                call    ds:wsprintfA
.text:004011AD                add     esp, 0Ch
.text:004011B0                push    ebx             ; lpString2
.text:004011B1                push    [ebp+lpString1] ; lpString1
.text:004011B4                call    ds:lstrcmpA
.text:004011BA                mov     ecx, [ebp+var_4]
.text:004011BD                neg     eax
.text:004011BF                pop     edi
.text:004011C0                sbb     eax, eax
.text:004011C2                xor     ecx, ebp
.text:004011C4                pop     esi
.text:004011C5                inc     eax
.text:004011C6                pop     ebx
.text:004011C7                call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:004011CC                mov     esp, ebp
.text:004011CE                pop     ebp
.text:004011CF                retn
.text:004011CF sub_401110     endp
```

图 3.6 加花前 IDA 函数调用分析

加花指令后，发现了花指令中带有的指令，函数的参数被成功混淆，混淆了入栈的数据，无法判断数据的作用。添加花指令后，无法显示当前的图形界面，如图 3.7、图 3.8 所示。

```
.text:00401317                call    ds:wsprintfA
.text:0040131D                add     esp, 0Ch
.text:00401320                call    $+5
.text:00401325                pop     eax
.text:00401326                call    loc_40132E
.text:00401326 ; ---------------------------------------------------------------------------
.text:0040132B                db 0EAh
.text:0040132C ; ---------------------------------------------------------------------------
.text:0040132C                jmp     short loc_401337
.text:0040132E ; ---------------------------------------------------------------------------
.text:0040132E
.text:0040132E loc_40132E:                            ; CODE XREF: .text:00401326↑j
.text:0040132E                pop     ebx
.text:0040132F                inc     ebx
.text:00401330                push    ebx
.text:00401331                mov     eax, 11111111h
.text:00401336                retn
.text:00401337 ; ---------------------------------------------------------------------------
.text:00401337
.text:00401337 loc_401337:                            ; CODE XREF: .text:0040132C↑j
.text:00401337                call    sub_401343
.text:0040133C                mov     ebx, 33333333h
.text:00401341                jmp     short loc_401350
```

图 3.7 加花后 IDA 函数调用分析

```
.text:00401343 ; =============== S U B R O U T I N E =======================================
.text:00401343
.text:00401343
.text:00401343 sub_401343      proc near              ; CODE XREF: .text:loc_401337↑p
.text:00401343                 mov     ebx, 11111111h
.text:00401348                 pop     ebx
.text:00401349                 mov     ebx, offset loc_401350
.text:0040134E                 push    ebx
.text:0040134F                 retn
.text:0040134F sub_401343      endp
.text:0040134F
.text:00401350 ; ---------------------------------------------------------------------------
.text:00401350
.text:00401350 loc_401350:                            ; CODE XREF: .text:00401341↑j
.text:00401350                                        ; DATA XREF: sub_401343+6↑o
.text:00401350                 mov     ebx, 22222222h
.text:00401355                 mov     eax, [ebp+0Ch]
.text:00401358                 push    eax
.text:00401359                 mov     ecx, [ebp+8]
.text:0040135C                 push    ecx
.text:0040135D                 call    ds:lstrcmpA
.text:00401363                 test    eax, eax
.text:00401365                 jnz     short loc_401370
.text:00401367                 mov     eax, 1
```

图 3.8 加花后 IDA 函数调用分析

对第三套花指令对函数 DialogBox 判断进行保护为例进行分析，如图 3.9 所示。

```
_FLOWER_XX02;
DialogBox (hInst, MAKEINTRESOURCE (IDD_CHECK), hDlg, CheckDlgProc ) ;
```

图 3.9 对 DialogBox 进行花指令保护

IDA 对比分析：

加花指令前：在 IDA 中可以清晰地看到 DialogBox 函数的调用位置以及参数保存位置，如图 3.10 所示。

```
.text:0040145B                call    ds:MessageBeep
.text:00401461                push    0               ; dwInitParam
.text:00401463                push    offset DialogFunc ; lpDialogFunc
.text:00401468                push    edi             ; hWndParent
.text:00401469                push    79h             ; lpTemplateName
.text:0040146B                push    hInstance       ; hInstance
.text:00401471                call    ds:DialogBoxParamA
```

图 3.10 加花前 IDA 函数调用分析

加花指令后：在参数入栈前发现了错误的跳转和运算。函数的某些参数被成功混淆，对保存参数的寄存器进行了错误的判断。无法判断调用函数之前的语句的作用。添加花指令后，无法显示当前的图形界面，如图 3.10 所示。

```
.text:0040174E                call    ds:MessageBeep
.text:00401754                nop
.text:00401755                push    eax
.text:00401756                inc     eax
.text:00401757                push    edx
.text:00401758                nop
.text:00401759                pop     eax
.text:0040175A                dec     eax
.text:0040175B                pop     edx
.text:0040175C                nop
.text:0040175D                jz      near ptr loc_401771+1
.text:00401763                jnz     near ptr loc_401771+1
.text:00401769                call    near ptr 22988BC5h
.text:0040176E                nop
.text:0040176F                nop
.text:00401770                pusha
.text:00401771
.text:00401771 loc_401771:                            ; CODE XREF: .text:0040175D↑j
.text:00401771                                        ; .text:00401763↑j
.text:00401771                ja      short loc_4017DD
.text:00401773                add     (byte_43FFFF-44005Fh)[eax], ch
.text:00401776                adc     [eax+0], al
.text:00401779                mov     eax, [ebp+8]
.text:0040177C                push    eax
.text:0040177D                push    79h
.text:0040177F                mov     ecx, dword_449A38
.text:00401785                push    ecx
.text:00401786                call    ds:DialogBoxParamA
.text:0040178C                jmp     short loc_40179E
```

图 3.11 加花后 IDA 函数调用分析

对第四套花指令对函数 EnableWindow 判断进行保护为例进行分析，如图 3.12 所示。

```
_FLOWER_XX03;
EnableWindow(GetDlgItem(hDlg,IDC_TXT0),FALSE);
EnableWindow(GetDlgItem(hDlg,IDC_TXT1),FALSE);
```

图 3.12 对 EnableWindow 进行花指令保护

IDA 对比分析：

加花指令前：在 IDA 中可以清晰地看到 EnableWindow 函数的调用位置以及参数保存位置，如图 3.13 所示。

```
text:004013F4                call    ds:lstrcpyA
text:004013FA                push    0               ; bEnable
text:004013FC                push    6Eh             ; nIDDlgItem
text:004013FE                push    edi             ; hDlg
text:004013FF                call    ebx ; GetDlgItem
text:00401401                mov     esi, ds:EnableWindow
text:00401407                push    eax             ; hWnd
text:00401408                call    esi ; EnableWindow
text:0040140A                push    0               ; bEnable
text:0040140C                push    3E8h            ; nIDDlgItem
text:00401411                push    edi             ; hDlg
text:00401412                call    ebx ; GetDlgItem
text:00401414                push    eax             ; hWnd
text:00401415                call    esi ; EnableWindow
```

图 3.13 加花前 IDA 函数调用分析

加花指令后，函数参数入栈时，对保存参数的寄存器进行了错误的判断。添加了错误的跳转指令，对函数的流程进行了保护，且无法判断调用函数之前的语句的作用。添加花指令后，无法显示当前的图形界面，如图 3.14 所示。

```
.text:004016A6                call    ds:lstrcpyA
.text:004016AC                jz      near ptr loc_4016B8+2
.text:004016B2                jnz     near ptr loc_4016B8+2
.text:004016B8
.text:004016B8 loc_4016B8:                            ; CODE XREF: .text:004016AC↑j
.text:004016B8                                        ; .text:004016B2↑j
.text:004016B8                call    near ptr 0ECCB6BD5h
.text:004016BD                inc     ecx
.text:004016BE                push    edx
.text:004016BF                pop     edx
.text:004016C0                dec     ecx
.text:004016C1                pop     ebp
.text:004016C2                inc     ecx
.text:004016C3                push    0
.text:004016C5                push    6Eh
.text:004016C7                mov     edx, [ebp+8]
.text:004016CA                push    edx
.text:004016CB                call    ds:GetDlgItem
.text:004016D1                push    eax
.text:004016D2                call    ds:EnableWindow
.text:004016D8                push    0
.text:004016DA                push    3E8h
.text:004016DF                mov     eax, [ebp+8]
.text:004016E2                push    eax
.text:004016E3                call    ds:GetDlgItem
.text:004016E9                push    eax
.text:004016EA                call    ds:EnableWindow
```

图 3.14 加花后 IDA 函数调用分析

3.2 SMC 功能测试

生成解决方案之后，通过 SMC pack 对程序进行 SMC 加密，加密成功后如图 3.15 所示。

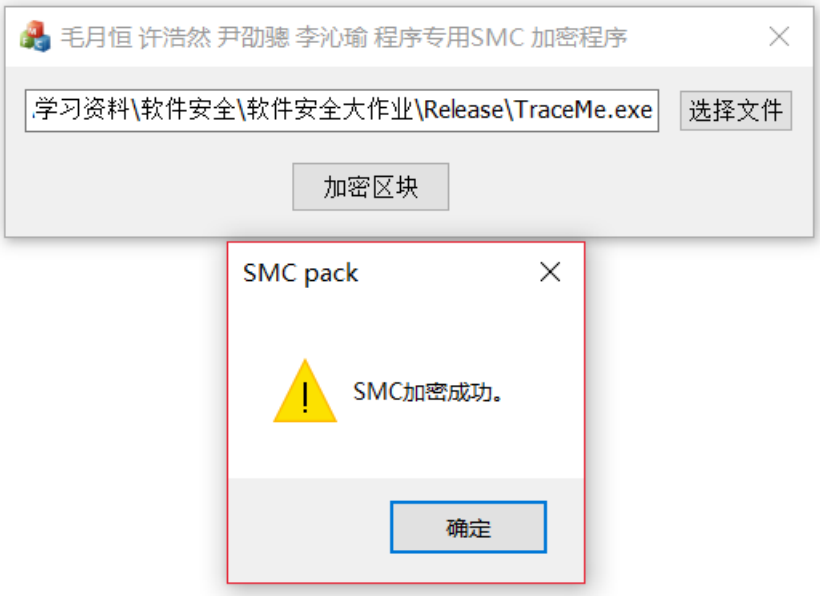

图 3.15 SMC 加密成功

通过 LordPE 查看该程序，可以发现新增加了 SMC 区段，如图 3.16 所示。

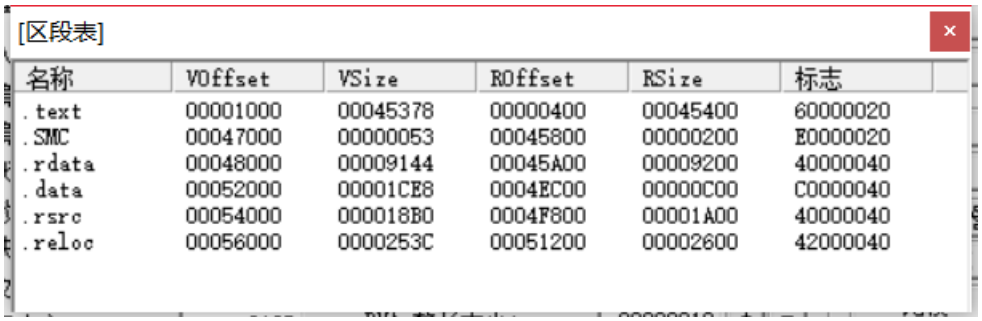

图 3.16 新增 SMC 区段

使用 IDA 查看，可以发现 IDA 无法分析出该程序的 SMC 区段的代码逻辑，只能看到一系列操作数，如图 3.17 所示。

```
.SMC:00447000 ; Section 2. (virtual address 00047000)
.SMC:00447000 ; Virtual size                  : 00000053 (     83.)
.SMC:00447000 ; Section size in file          : 00000200 (    512.)
.SMC:00447000 ; Offset to raw data for section: 00045800
.SMC:00447000 ; Flags E0000020: Text Executable Readable Writable
.SMC:00447000 ; Alignment     : default
.SMC:00447000 ; ===========================================================================
.SMC:00447000
.SMC:00447000 ; Segment type: Pure code
.SMC:00447000 ; Segment permissions: Read/Write/Execute
.SMC:00447000 .SMC            segment para public 'CODE' use32
.SMC:00447000                 assume cs:.SMC
.SMC:00447000                 ;org 447000h
.SMC:00447000                 assume es:nothing, ss:nothing, ds:.data, fs:nothing, gs:nothing
.SMC:00447000                 dd 9820469Bh, 0FFC19341h, 46999C02h, 0EE74C3B1h, 0E1CBCCCDh
.SMC:00447000                 dd 0B53A4839h, 77D943DDh, 5DDE9740h, 9C47C947h, 41CF45C9h
.SMC:00447000                 dd 0B702F68Fh, 0C0904539h, 324B0AF8h, 8341D3B2h, 7BDF41DDh
.SMC:00447000                 dd 0CB8CD2C1h, 414EC241h, 448ECE99h, 9F13786h, 94943EB1h
.SMC:00447000                 dd 0CB0F9095h, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh, 0CECACBCCh, 0CACBCCCDh
.SMC:00447000                 dd 0CBCCCDCEh, 0CCCDCECAh, 0CDCECACBh
.SMC:00447200                 dd 380h dup(?)
.SMC:00447200 .SMC            ends
.SMC:00447200
```

图 3.17 使用 IDA 分析 SMC 区段

3.3 反调试功能测试

1. 枚举进程列表

在使用带插件的 OD 进行调试时，立即被检测出来，弹出对话框后退出程序，实际保护中可以不弹框，或者隐藏字符串，为方便演示和检测反调试效果故弹出对话框，如图 3.18 所示。

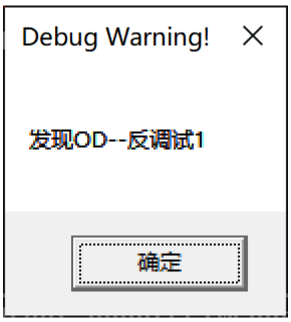

图 3.18 反调试 1 实现

2. 父进程检测

可检测出原版 OD，在插件保护下的 OD 不能被发现，如图 3.19 所示。

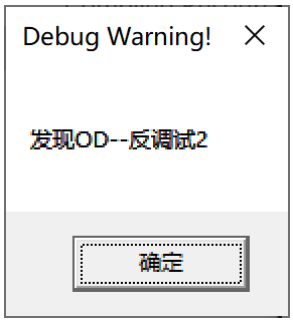

图 3.19 反调试 2 实现

3.检测 BeingDebugged 字段

可检测出原版 OD，在插件保护下的 OD 不能被发现，如图 3.20 所示。

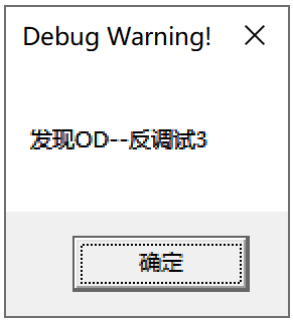

图 3.20 反调试 3 实现

4.检查时序 GetTickCount

弱保护，能通过带插件的 OD，但在不同电脑上不一定能发现 OD，在某些有时间处理的程序中不易被调试出来，如图 3.21 所示。

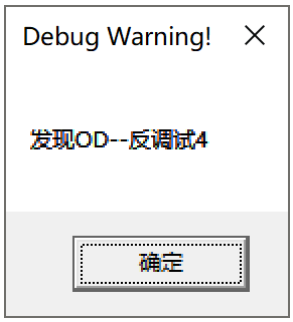

图 3.21 反调试 4 实现

5.检测 STARTUPINFO

可检测出原版 OD，在插件保护下的 OD 不能被发现，如图 3.22 所示。

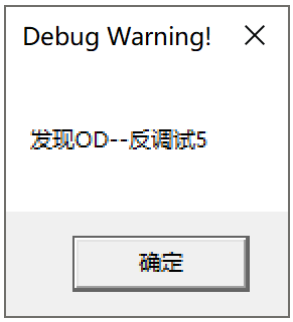

图 3.22 反调试 5 实现

6.检测硬件断点

能通过有插件 OD 的反调试，下硬件断点即被检测，如图 3.23 所示。

图 检测硬件断点

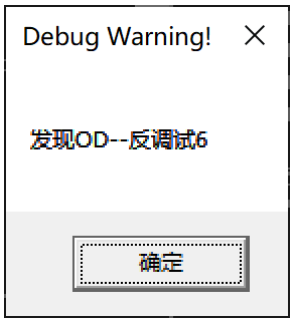

图 3.23 反调试 6 实现

3.4 注册码测试

用户输入的用户名和序列号不匹配时，程序会弹出序列号错误的对话框，如图 3.24 所示。

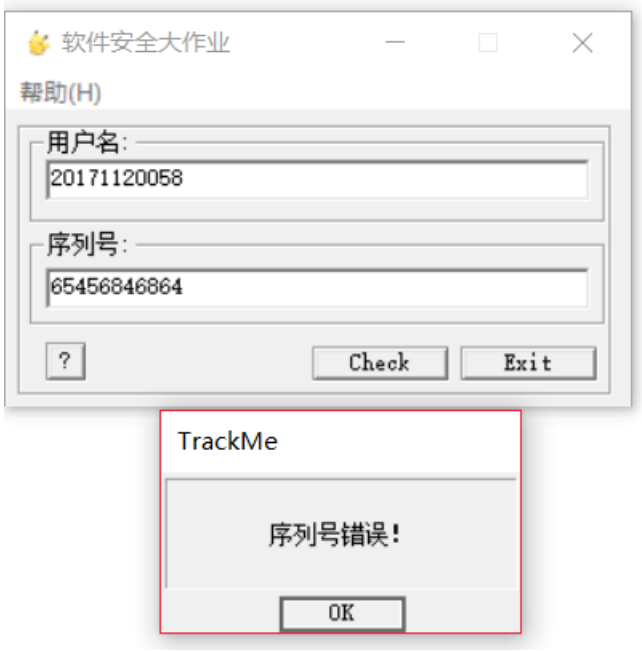

图 3.24 序列号错误

使用注册机计算出用户名对应的正确的序列号时，程序就会弹出注册成功的对话框，如图 3.25、图 3.26 所示。

图 3.25 注册机

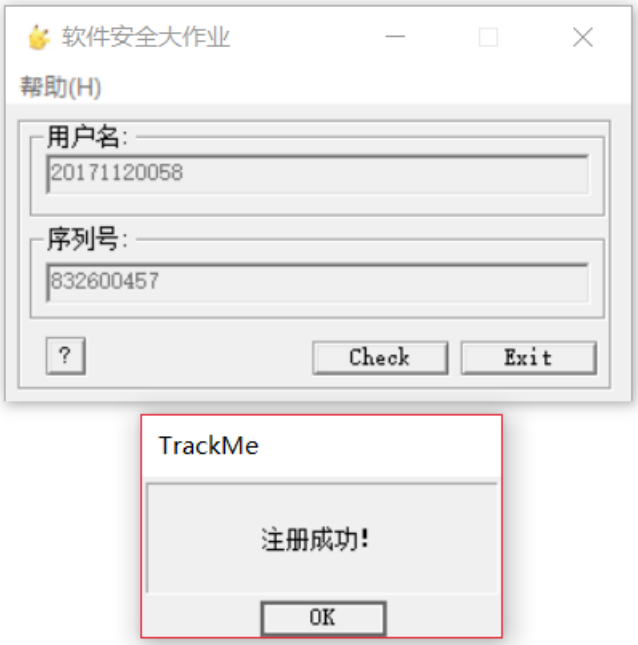
软件安全大作业
帮助(H)
用户名:
20171120058
序列号:
832600457
?
Check
Exit
TrackMe
注册成功!
OK

图 3.26 注册成功

四.大作业总结

本次大作业本小组实现了基于花指令保护、SMC 保护、反动态调试和一机一码的注册码对软件进行保护。共定义了四套花指令宏，使用了一个 SMC 保护，使用了 6 个反动态调试功能，使用了基于 CPU 序列号和 MD5 加密算法的一机一码注册码保护。

通过本学期的学习和本次大作业的学习，我们学到了很多软件保护的知识，对于软件保护的认识更加深刻了。本学期学会了可执行文件的 PE 结构、导入表导出表、花指令保护、SMC 保护、反动态调试等软件安全的基础知识。通过大作业的实践操作，自己动手编写花指令代码、反动态调试代码、SMC 保护代码和注册码算法，我们可以自己实现一些针对软件保护的功能。

软件安全对于社会的发展有着巨大的意义，不仅仅是可以保护作者的软件知识产权，更能提高人们的软件正版意识。今后在开发程序时，可以使用软件保护技术，对软件进行多种保护，防止他人破解，保护自己的知识产权。

附录

[1]详解反调试技术. https://blog.csdn.net/qq_32400847/article/details/52798050
[2]软件保护技术概述. https://blog.csdn.net/fuhanghang/article/details/84036545
[3]加密与解密.第四版.段钢著
[4]有趣的二进制.[日]爱甲健二著
[5]看雪安全论坛. https://bbs.pediy.com/
[6]逆向工程核心原理.[韩]李承远著
[7]恶意代码分析实战.[美]Micheal Sikorski, Andrew Honig 著
[8] 计算机软件注册码的法律保护.https://wenku.baidu.com/view/e69163814531b90d6c85ec3a87c24028905f854d.html

源码

1）TraceMe.cpp

```
#include<iostream>
#include <stdio.h>
#include "resource.h"
#include <windows.h>
#include <stdlib.h>
#include <string.h>


#include <Tlhelp32.h>
#include <process.h>

using namespace std;

/*-------------------------------------------------------------*/
/* 定义子程序与全局变量、常量                                  */
/*-------------------------------------------------------------*/
#define _WJQ_USED_FLOWER

#define MAXINPUTLEN 80

INT_PTR    CALLBACK MainDlg   (HWND, UINT, WPARAM, LPARAM) ;
INT_PTR    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;
INT_PTR    CALLBACK CheckDlgProc  (HWND, UINT, WPARAM, LPARAM) ;

BOOL GenRegCode( TCHAR  *rCode, TCHAR  *name ,int len) ;
TCHAR szBuffer[30];

HINSTANCE hInst;

LPTSTR namebuf = new char[12];
DWORD namesize = 12;
DWORD serialnumber;
DWORD maxlen;
DWORD fileflag;
LPTSTR sysnamebuf = new char[10];
DWORD sysnamesize = 10;



/*各函数声明*/
```

```
void shizhuaner(int in, int n, int* md5);
void shizhuaner_weishu(int in, int* md5);
void shiliuzhuaner(char* t, int* temp);
int c_out(int* a);
void abcd_out(int* a);
void move(int step, int* temp1, int* temp2);
void F(int* b, int* c, int* d, int* temp1, int* temp2);
void G(int* b, int* c, int* d, int* temp1, int* temp2);
void H(int* b, int* c, int* d, int* temp);
void I(int* b, int* c, int* d, int* temp);
void yu(int* a, int* b, int* temp);
void huo(int* a, int* b, int* temp);
void fei(int* a, int* temp);
void yihuo(int* a, int* b, int* temp);
void jia(int* a, int* b, int* temp);
int MD5(char x);
void xorPlus(char* , int , char* , int );

void SMC(char* , char* );


void UnPack();


/*------------------------------------------------------------*/
/* 定义花指令                                                 */
/*------------------------------------------------------------*/



#define _FLOWER_XXOO _asm{\
__asm jz  $+0xe  \
__asm jnz $+8 \
__asm _emit 0xe8 \
__asm _emit -0xe8 \
__asm push ebp \
__asm mov ebp,esp \
__asm push -1 \
__asm push 666666 \
__asm push 888888 \
__asm mov eax,dword ptr fs:[0] \
__asm nop \
__asm mov dword ptr fs:[0],esp \
__asm nop \
```

```
__asm mov dword ptr fs:[0],eax \
__asm pop eax \
__asm pop eax \
__asm nop \
__asm pop eax \
__asm pop eax \
__asm nop \
__asm mov ebp,eax \
}

#define _FLOWER_XX01 __asm{\
__asm call l1 \
__asm l1: \
__asm pop eax \
__asm call f1 \
__asm _EMIT 0xEA \
__asm jmp l2 \
__asm f1 : \
__asm pop ebx \
__asm inc ebx \
__asm push ebx \
__asm mov eax, 0x11111111 \
__asm ret \
__asm l2 : \
__asm call f2 \
__asm mov ebx, 0x33333333 \
__asm jmp e \
__asm f2 : \
__asm mov ebx, 0x11111111 \
__asm pop ebx \
__asm mov ebx, offset e \
__asm push ebx \
__asm ret \
__asm e : \
__asm mov ebx, 0x22222222 \
}

#define _FLOWER_XX02 __asm{\
__asm nop \
__asm push eax \
__asm inc eax \
__asm push edx \
__asm nop \
__asm pop eax \
```

```
__asm dec eax \
__asm pop edx \
__asm nop \
__asm jz $+0x15 \
__asm jnz $+0xf \
__asm _emit 0xe8 \
__asm _emit 0x57 \
__asm _emit 0x74 \
__asm _emit 0x58 \
__asm _emit 0x22 \
__asm _emit 0x90 \
__asm _emit 0x90 \
__asm _emit 0x60 \
__asm _emit 0x77 \
}

#define _FLOWER_XX03 __asm{\
__asm jz  $+0xe  \
__asm jnz $+8 \
__asm _emit 0xe8 \
__asm _emit -0xe8 \
__asm push ebp \
__asm mov ebp,esp \
__asm inc ecx \
__asm push edx \
__asm pop edx \
__asm dec ecx \
__asm pop ebp \
__asm inc ecx \
}


/*------------------------------------------------------------*/
/* 反调试                                                     */
/*------------------------------------------------------------*/

BOOL CheckDebug1()//调试器进程检测
{
    DWORD ID;
    DWORD ret = 0;
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(pe32);
    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE)
```

```
        {
            return FALSE;
        }
        BOOL bMore = Process32First(hProcessSnap, &pe32);
        while (bMore)
        {
            if (stricmp(pe32.szExeFile, "OllyDBG.EXE") == 0 ||
stricmp(pe32.szExeFile, "吾爱破解[LCG].exe") == 0 || stricmp(pe32.szExeFile,
"x64_dbg.exe") == 0 || stricmp(pe32.szExeFile, "windbg.exe") == 0 ||
stricmp(pe32.szExeFile, "ImmunityDebugger.exe") == 0)
            {
               MessageBox(NULL, TEXT("发现OD--反调试1"), TEXT("Debug Warning!"),
MB_OK);
                system("Pause");
                exit(-1);
            }
            bMore = Process32Next(hProcessSnap, &pe32);
        }
        CloseHandle(hProcessSnap);
        return FALSE;
    }

    BOOL CheckDebug2()//父进程检测
    {
       if (FindWindowA("OllyDbg", NULL) != NULL || FindWindowA("WinDbgFrameClass",
NULL) != NULL || FindWindowA("吾爱破解[LCG]", NULL) != NULL)
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 2"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
        else
        {
            return FALSE;
        }
    }
    BOOL CheckDebug3()//BeingDebugged字段检测
    {
        BOOL isDebuggerPresent = false;
        if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPresent))
        {
            if (isDebuggerPresent)
            {
```

```
                MessageBox(NULL, TEXT("发现OD--反调试3"), TEXT("Debug Warning!"),
MB_OK);
                system("Pause");
                exit(-1);
            }
        }
    }

    BOOL CheckDebug4()//利用时间判断
    {
        DWORD time1 = GetTickCount();
        __asm
        {
            mov     ecx, 10
            mov     edx, 6
            mov     ecx, 10
        }
        DWORD time2 = GetTickCount();
        if (time2 - time1 > 0xA)
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 4"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
        else
        {
            return FALSE;
        }
    }

    BOOL CheckDebug5()//系统打开程序下列值为0，OD打开则不为0
    {
        STARTUPINFO si;
        GetStartupInfo(&si);
        if (si.dwX != 0 || si.dwY != 0 || si.dwFillAttribute != 0 || si.dwXSize !=
0 || si.dwYSize != 0 || si.dwXCountChars != 0 || si.dwYCountChars != 0)
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 5"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
        else
```

```
        {
            return FALSE;
        }
    }

    BOOL CheckDebug6()//硬件断点检测
    {
        CONTEXT context;
        HANDLE hThread = GetCurrentThread();
        context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
        GetThreadContext(hThread, &context);
        if (context.Dr0 != 0 || context.Dr1 != 0 || context.Dr2 != 0 || context.Dr3 !=
0)
        {
            MessageBox(NULL, TEXT("发现 OD--反调试 6"), TEXT("Debug Warning!"),
MB_OK);
            system("Pause");
            exit(-1);
        }
        return FALSE;
    }


    /*-------------------------------------------------------------*/
    /* WinMain — 基于 WIN32 的程序的入口                            */
    /*-------------------------------------------------------------*/
    int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        PSTR szCmdLine, int iCmdShow)
    {

        hInst = hInstance;
        DialogBoxParam (hInstance, MAKEINTRESOURCE(IDD_MAINDLG), NULL, MainDlg,
NULL);
        return 0;
    }

    /*-------------------------------------------------------------*/
    /* AboutDlgProc — 关于窗口                                     */
    /*-------------------------------------------------------------*/

    INT_PTR CALLBACK AboutDlgProc (HWND hDlg, UINT message,
                                   WPARAM wParam, LPARAM lParam)
    {
        switch (message)
```

```
        {

        case WM_COMMAND :
            switch (LOWORD (wParam))
            {
            case IDOK :
            case IDCANCEL :
                EndDialog (hDlg, 0) ;
                return TRUE ;
             }
              break ;
         }
         return FALSE ;
    }

    /*------------------------------------------------------------*/
    /* CheckDlgProc — 提示信息窗口                                */
    /*------------------------------------------------------------*/

    INT_PTR CALLBACK CheckDlgProc (HWND hDlg, UINT message,
                                   WPARAM wParam, LPARAM lParam)
    {
        switch (message)
        {
        case WM_INITDIALOG:

            SendMessage(GetDlgItem(hDlg,     IDC_STATIC1),     WM_SETTEXT,     0,
(LPARAM)szBuffer);//初始化提示信息
            return TRUE;
        case WM_COMMAND :
            switch (LOWORD (wParam))
            {
            case IDOK :
            case IDCANCEL :
                EndDialog (hDlg, 0) ;
                return TRUE ;
             }
              break ;
         }
         return FALSE ;
    }


    /*------------------------------------------------------------*/
```

```
/*  MainDlg — 主对话窗口                                          */
/*--------------------------------------------------------------*/

INT_PTR CALLBACK MainDlg (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)

{

    TCHAR cName[MAXINPUTLEN]={0};
    TCHAR cCode[100]={0};
    int len;
    TCHAR szEnchar[] = TEXT ("你输入字符要大于四个！") ;
    _FLOWER_XX00;
    TCHAR szSucc[] = TEXT ("注册成功！") ;
    _FLOWER_XX02;
    TCHAR szFail[] = TEXT ("序列号错误！") ;
    _FLOWER_XX03;

    CheckDebug1();
    CheckDebug2();
    CheckDebug3();
    CheckDebug4();
    CheckDebug5();
    CheckDebug6();


    switch (message)
    {
    case WM_INITDIALOG:

SendMessage(hDlg,WM_SETICON,ICON_BIG,LPARAM(LoadIcon(hInst,MAKEINTRESOURCE(
IDI_ICON)))); //设置图标
        SendDlgItemMessage(hDlg, IDC_TXT0, EM_LIMITTEXT, MAXINPUTLEN, 0);
//初始化 edit 控件 IDC_TXT0 字符长度
        break;

    case WM_CLOSE:
        DestroyWindow(hDlg);
        break;

    case WM_COMMAND:
        switch (LOWORD (wParam))
        {

        case IDC_ABOUT :
```

```
            case IDM_HELP_ABOUT :
                DialogBox    (hInst,    MAKEINTRESOURCE    (IDD_ABOUT),    hDlg,
AboutDlgProc) ;
                break;

            case IDC_OK:

    len=GetDlgItemText(hDlg,IDC_TXT0,cName,sizeof(cName)/sizeof(TCHAR)+1);

    GetDlgItemText(hDlg,IDC_TXT1,cCode,sizeof(cCode)/sizeof(TCHAR)+1);
                if (cName[0] == 0||len<5)
                {
                    lstrcpy(szBuffer,szEnchar);
                    SetFocus (GetDlgItem(hDlg,IDC_TXT0));
                }

                else
                {
                    _FLOWER_XX00;
                    if(GenRegCode(cCode, cName ,len))   //此处调用序列号计算的
子程序
                    {   lstrcpy(szBuffer,szSucc);
                        _FLOWER_XX03;
                        EnableWindow(GetDlgItem(hDlg,IDC_TXT0),FALSE);
                        EnableWindow(GetDlgItem(hDlg,IDC_TXT1),FALSE);
                        _FLOWER_XX02;
                    }

                    else
                        lstrcpy(szBuffer,szFail);
                SetFocus (GetDlgItem(hDlg,IDC_TXT1));
                }
                _FLOWER_XX03;
                MessageBeep (MB_OK);
                _FLOWER_XX02;
                DialogBox    (hInst,    MAKEINTRESOURCE    (IDD_CHECK),    hDlg,
CheckDlgProc ) ;

                break;

            case IDC_EXIT:
                SendMessage(hDlg, WM_CLOSE, 0, 0);
                break;
            }
```

```
            return TRUE;
            break;
        }

        return FALSE;
    }
```

```
/*十进制转二进制函数*/
void shizhuaner(int in, int n, int* md5)
{
    int j, s, w;
    s = n / 4 + 1;  //s 是 md5 里面组的排位数，w 是该组里面的位数
    w = n % 4;
    j = 1;
    do
    {
        md5[32 * s - 8 * w - j] = in % 2;
        in = in / 2;
        j++;
    } while (in != 0);
    while (j <= 8)  //二进制不够八位时补零
    {
        md5[32 * s - 8 * w - j] = 0;
        j++;
    }
}
```

```
/* 位数填充时所用到的十进制转二进制函数 */
void shizhuaner_weishu(int in, int* md5)
{
    int i, j, temp, a[64];
    for (i = 0; in != 0; i++)
    {
        a[i] = in % 2;
        in = in / 2;
    }
    while (i % 8 != 0)  //二进制位数不够八的整数倍时补零
    {
        a[i] = 0;
        i++;
    }
    for (j = 0; j < i / 2; j++)
    {
        temp = a[i - j - 1];
        a[i - j - 1] = a[j];
        a[j] = temp;

    }
    temp = i / 8;
    for (i = i - 1; i < 64; i++)
        a[i] = 0;
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 8; j++)
            md5[512 - temp * 8 + j - 32] = a[i * 8 + j];
        temp = temp - 1;
    }
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 8; j++)
            md5[512 - (i + 1) * 8 + j] = a[i * 8 + j + 32];
    }
}

/* 十六进制转二进制函数 */
void shiliuzhuaner(char* t, int* temp)
{
    int i;
    for (i = 0; i < 8; i++)
    {
        switch (t[i])
```

```
            {
            case '0': {temp[4 * i] = 0; temp[4 * i + 1] = 0; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 0; }break;
            case '1': {temp[4 * i] = 0; temp[4 * i + 1] = 0; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 1; }break;
            case '2': {temp[4 * i] = 0; temp[4 * i + 1] = 0; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 0; }break;
            case '3': {temp[4 * i] = 0; temp[4 * i + 1] = 0; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 1; }break;
            case '4': {temp[4 * i] = 0; temp[4 * i + 1] = 1; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 0; }break;
            case '5': {temp[4 * i] = 0; temp[4 * i + 1] = 1; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 1; }break;
            case '6': {temp[4 * i] = 0; temp[4 * i + 1] = 1; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 0; }break;
            case '7': {temp[4 * i] = 0; temp[4 * i + 1] = 1; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 1; }break;
            case '8': {temp[4 * i] = 1; temp[4 * i + 1] = 0; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 0; }break;
            case '9': {temp[4 * i] = 1; temp[4 * i + 1] = 0; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 1; }break;
            case 'a': {temp[4 * i] = 1; temp[4 * i + 1] = 0; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 0; }break;
            case 'b': {temp[4 * i] = 1; temp[4 * i + 1] = 0; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 1; }break;
            case 'c': {temp[4 * i] = 1; temp[4 * i + 1] = 1; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 0; }break;
            case 'd': {temp[4 * i] = 1; temp[4 * i + 1] = 1; temp[4 * i + 2] = 0;
temp[4 * i + 3] = 1; }break;
            case 'e': {temp[4 * i] = 1; temp[4 * i + 1] = 1; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 0; }break;
            case 'f': {temp[4 * i] = 1; temp[4 * i + 1] = 1; temp[4 * i + 2] = 1;
temp[4 * i + 3] = 1; }break;
            }
        }
    }



    /* 密文输出函数 */

    int c_out(int* a)
    {
        int i, add;
```

```
        int x[8];
        int result=0;
        for (i = 1; i <= 4; i++)  //二进制转换成十六进制输出
        {
            add = a[32 - i * 8] * 8 + a[32 - i * 8 + 1] * 4 + a[32 - i * 8 + 2] *
2 + a[32 - i * 8 + 3];
            if (add >= 10)
            {
                switch (add)
                {
                case 10:x[i - 1] = 2; break;
                case 11:x[i - 1] = 0; break;
                case 12:x[i - 1] = 1; break;
                case 13:x[i - 1] = 7; break;
                case 14:x[i - 1] = 1; break;
                case 15:x[i - 1] = 1; break;
                }
            }
            else
                x[i - 1] = add;
            add = a[32 - i * 8 + 4] * 8 + a[32 - i * 8 + 5] * 4 + a[32 - i * 8 +
6] * 2 + a[32 - i * 8 + 7];
            if (add >= 10)
            {
                switch (add)
                {
                case 10:x[i + 3] = 2; break;
                case 11:x[i + 3] = 0; break;
                case 12:x[i + 3] = 0; break;
                case 13:x[i + 3] = 5; break;
                case 14:x[i + 3] = 8; break;
                case 15:x[i + 3] = 1; break;
                }
            }
            else
                x[i + 3] = add;
        }

        for (int j = 0; j < 8; j++)
        {
            result += x[j];
        }
```

```
    return result;
}


/* 中间过程的输出函数 */
void abcd_out(int* a)
{
    int i, add;
    for (i = 0; i < 4; i++)  //二进制转换成十六进制输出
    {
        add = a[i * 8] * 8 + a[i * 8 + 1] * 4 + a[i * 8 + 2] * 2 + a[i * 8 +
3];
        if (add >= 10)
        {
            switch (add)
            {
            case 10:printf("a"); break;
            case 11:printf("b"); break;
            case 12:printf("c"); break;
            case 13:printf("d"); break;
            case 14:printf("e"); break;
            case 15:printf("f"); break;
            }
        }
        else
            printf("%d", add);
        add = a[i * 8 + 4] * 8 + a[i * 8 + 5] * 4 + a[i * 8 + 6] * 2 + a[i *
8 + 7];
        if (add >= 10)
        {
            switch (add)
            {
            case 10:printf("a"); break;
            case 11:printf("b"); break;
            case 12:printf("c"); break;
            case 13:printf("d"); break;
            case 14:printf("e"); break;
            case 15:printf("f"); break;
            }
        }
        else
            printf("%d", add);
    }
}
```

```
/* 与函数 */
void yu(int* a, int* b, int* temp)
{
    int i;
    for (i = 0; i < 32; i++)  //同为1为1，否则为0
    {
        if (a[i] == 1 && b[i] == 1)
            temp[i] = 1;
        else
            temp[i] = 0;
    }
}

/* 或函数 */
void huo(int* a, int* b, int* temp)
{
    int i;
    for (i = 0; i < 32; i++)  //同0为0，否则为1
    {
        if (a[i] == 0 && b[i] == 0)
            temp[i] = 0;
        else
            temp[i] = 1;
    }
}

/* 非函数 */
void fei(int* a, int* temp)
{
    int i;
    for (i = 0; i < 32; i++)
    {
        if (a[i] == 0)
            temp[i] = 1;
        else
            temp[i] = 0;
    }
}

/*异或函数*/
void yihuo(int* a, int* b, int* temp)
{
    int i;
```

```
    for (i = 0; i < 32; i++)  //相同为0，不同为1
    {
        if (a[i] != b[i])
            temp[i] = 1;
        else
            temp[i] = 0;
    }
}

/* 模二的32次加 */
void jia(int* a, int* b, int* temp)
{
    int i, jin;
    jin = 0;
    for (i = 0; i < 32; i++)
    {
        if (a[31 - i] + b[31 - i] + jin > 1)
        {
            temp[31 - i] = a[31 - i] + b[31 - i] + jin - 2;
            jin = 1;
        }
        else
        {
            temp[31 - i] = a[31 - i] + b[31 - i] + jin;
            jin = 0;
        }
    }
}

/* F函数 */
void F(int* b, int* c, int* d, int* temp1, int* temp2)
{
    /* F(x,y,z)=(x∧y)∨(?x∧z) */
    yu(b, c, temp1);
    fei(b, temp2);
    yu(temp2, d, temp2);
    huo(temp1, temp2, temp2);
}

/* G函数 */
void G(int* b, int* c, int* d, int* temp1, int* temp2)
{
    /* G(x,y,z)=(x∧z)∨(y∧?z) */
    yu(b, d, temp1);
```

```
    fei(d, temp2);
    yu(temp2, c, temp2);
    huo(temp1, temp2, temp2);
}

/* H 函数 */
void H(int* b, int* c, int* d, int* temp)
{
    /* H(x,y,z)=x⊕y⊕z */
    yihuo(b, c, temp);
    yihuo(temp, d, temp);
}

/* I 函数 */
void I(int* b, int* c, int* d, int* temp)
{
    /* I(x,y,z)=y⊕(x∨?z) */
    fei(d, temp);
    huo(b, temp, temp);
    yihuo(c, temp, temp);
}

/*左移函数*/
#pragma code_seg(".SMC")
void move(int step, int* temp1, int* temp2)
{
    int i;
    for (i = 0; i < 32 - step; i++)
        temp2[i] = temp1[i + step];
    for (i = 0; i < step; i++)
        temp2[32 - step + i] = temp1[i];
}

#pragma code_seg()
#pragma comment(linker, "/SECTION:.SMC,ERW")

/*每一大轮的 16 小轮循环函数*/
void round(int* a, int* b, int* c, int* d, int* m, int* md5, int r, char* t1,
    char* t2, char* t3, char* t4, char* t5, char* t6, char* t7, char* t8, char*
t9,
    char* t10, char* t11, char* t12, char* t13, char* t14, char* t15, char* t16)
{
    int i, j, in, step, temp1[32], temp2[32];
    for (i = 0; i < 16; i++)
```

```
        {
            switch (r)  //根据 r 判断所选的逻辑函数
            {
            case 1:F(b, c, d, temp1, temp2); break;
            case 2:G(b, c, d, temp1, temp2); break;
            case 3:H(b, c, d, temp2); break;
            case 4:I(b, c, d, temp2); break;
            }
            in = m[i];
            for (j = 0; j < 32; j++)
                temp1[j] = md5[in * 32 + j];
            jia(temp2, temp1, temp2);
            switch (i + 1)  //选择 t[]
            {
            case 1:shiliuzhuaner(t1, temp1); break;
            case 2:shiliuzhuaner(t2, temp1); break;
            case 3:shiliuzhuaner(t3, temp1); break;
            case 4:shiliuzhuaner(t4, temp1); break;
            case 5:shiliuzhuaner(t5, temp1); break;
            case 6:shiliuzhuaner(t6, temp1); break;
            case 7:shiliuzhuaner(t7, temp1); break;
            case 8:shiliuzhuaner(t8, temp1); break;
            case 9:shiliuzhuaner(t9, temp1); break;
            case 10:shiliuzhuaner(t10, temp1); break;
            case 11:shiliuzhuaner(t11, temp1); break;
            case 12:shiliuzhuaner(t12, temp1); break;
            case 13:shiliuzhuaner(t13, temp1); break;
            case 14:shiliuzhuaner(t14, temp1); break;
            case 15:shiliuzhuaner(t15, temp1); break;
            case 16:shiliuzhuaner(t16, temp1); break;
            }
            jia(temp2, temp1, temp2);
            jia(temp2, a, temp2);
            switch (r)  //根据 r 为左移步数 step 赋值
            {
            case 1:switch (i % 4 + 1) { case 1:step = 7; break; case 2:step = 12;
break; case 3:step = 17; break; case 4:step = 22; break; }break;
            case 2:switch (i % 4 + 1) { case 1:step = 5; break; case 2:step = 9;
break; case 3:step = 14; break; case 4:step = 20; break; }break;
            case 3:switch (i % 4 + 1) { case 1:step = 4; break; case 2:step = 11;
break; case 3:step = 16; break; case 4:step = 23; break; }break;
            case 4:switch (i % 4 + 1) { case 1:step = 6; break; case 2:step = 10;
break; case 3:step = 15; break; case 4:step = 21; break; }break;
            }
```

```
        UnPack();
        move(step, temp2, temp1);
        UnPack();

        jia(temp1, b, temp2);
        for (j = 0; j < 32; j++)
        {
            a[j] = d[j];
            d[j] = c[j];
            c[j] = b[j];
            b[j] = temp2[j];
        }

        /*若想输出每轮 a、b、c、d 的值，把下面的注释取消即可*/
        /*printf("第%d 大轮的第%d 小轮\n", r, i);
        abcd_out(a);
        printf("  ");
        abcd_out(b);
        printf("  ");
        abcd_out(c);
        printf("  ");
        abcd_out(d);
        printf("\n");*/

    }
}

/* 主函数 */
int MD5(char x)
{
    char ch,
        /* 一大坨 t[] */
        t1[8] = { 'd', '7', '6', 'a', 'a', '4', '7', '8' },
        t2[8] = { 'e', '8', 'c', '7', 'b', '7', '5', '6' },
        t3[8] = { '2', '4', '2', '0', '7', '0', 'd', 'b' },
        t4[8] = { 'c', '1', 'b', 'd', 'c', 'e', 'e', 'e' },
        t5[8] = { 'f', '5', '7', 'c', '0', 'f', 'a', 'f' },
        t6[8] = { '4', '7', '8', '7', 'c', '6', '2', 'a' },
        t7[8] = { 'a', '8', '3', '0', '4', '6', '1', '3' },
        t8[8] = { 'f', 'd', '4', '6', '9', '5', '0', '1' },
        t9[8] = { '6', '9', '8', '0', '9', '8', 'd', '8' },
        t10[8] = { '8', 'b', '4', '4', 'f', '7', 'a', 'f' },
```

```
t11[8] = { 'f', 'f', 'f', 'f', '5', 'b', 'b', '1' },
t12[8] = { '8', '9', '5', 'c', 'd', '7', 'b', 'e' },
t13[8] = { '6', 'b', '9', '0', '1', '1', '2', '2' },
t14[8] = { 'f', 'd', '9', '8', '7', '1', '9', '3' },
t15[8] = { 'a', '6', '7', '9', '4', '3', '8', 'e' },
t16[8] = { '4', '9', 'b', '4', '0', '8', '2', '1' },
t17[8] = { 'f', '6', '1', 'e', '2', '5', '6', '2' },
t18[8] = { 'c', '0', '4', '0', 'b', '3', '4', '0' },
t19[8] = { '2', '6', '5', 'e', '5', 'a', '5', '1' },
t20[8] = { 'e', '9', 'b', '6', 'c', '7', 'a', 'a' },
t21[8] = { 'd', '6', '2', 'f', '1', '0', '5', 'd' },
t22[8] = { '0', '2', '4', '4', '1', '4', '5', '3' },
t23[8] = { 'd', '8', 'a', '1', 'e', '6', '8', '1' },
t24[8] = { 'e', '7', 'd', '3', 'f', 'b', 'c', '8' },
t25[8] = { '2', '1', 'e', '1', 'c', 'd', 'e', '6' },
t26[8] = { 'c', '3', '3', '7', '0', '7', 'd', '6' },
t27[8] = { 'f', '4', 'd', '5', '0', 'd', '8', '7' },
t28[8] = { '4', '5', '5', 'a', '1', '4', 'e', 'd' },
t29[8] = { 'a', '9', 'e', '3', 'e', '9', '0', '5' },
t30[8] = { 'f', 'c', 'e', 'f', 'a', '3', 'f', '8' },
t31[8] = { '6', '7', '6', 'f', '0', '2', 'd', '9' },
t32[8] = { '8', 'd', '2', 'a', '4', 'c', '8', 'a' },
t33[8] = { 'f', 'f', 'f', 'a', '3', '9', '4', '2' },
t34[8] = { '8', '7', '7', '1', 'f', '6', '8', '1' },
t35[8] = { '6', 'd', '9', 'd', '6', '1', '2', '2' },
t36[8] = { 'f', 'd', 'e', '5', '3', '8', '0', 'c' },
t37[8] = { 'a', '4', 'b', 'e', 'e', 'a', '4', '4' },
t38[8] = { '4', 'b', 'd', 'e', 'c', 'f', 'a', '9' },
t39[8] = { 'f', '6', 'b', 'b', '4', 'b', '6', '0' },
t40[8] = { 'b', 'e', 'b', 'f', 'b', 'c', '7', '0' },
t41[8] = { '2', '8', '9', 'b', '7', 'e', 'c', '6' },
t42[8] = { 'e', 'a', 'a', '1', '2', '7', 'f', 'a' },
t43[8] = { 'd', '4', 'e', 'f', '3', '0', '8', '5' },
t44[8] = { '0', '4', '8', '8', '1', 'd', '0', '5' },
t45[8] = { 'd', '9', 'd', '4', 'd', '0', '3', '9' },
t46[8] = { 'e', '6', 'd', 'b', '9', '9', 'e', '5' },
t47[8] = { '1', 'f', 'a', '2', '7', 'c', 'f', '8' },
t48[8] = { 'c', '4', 'a', 'c', '5', '6', '6', '5' },
t49[8] = { 'f', '4', '2', '9', '2', '2', '4', '4' },
t50[8] = { '4', '3', '2', 'a', 'f', 'f', '9', '7' },
t51[8] = { 'a', 'b', '9', '4', '2', '3', 'a', '7' },
t52[8] = { 'f', 'c', '9', '3', 'a', '0', '3', '9' },
t53[8] = { '6', '5', '5', 'b', '5', '9', 'c', '3' },
t54[8] = { '8', 'f', '0', 'c', 'c', 'c', '9', '2' },
```

```
    t55[8] = { 'f', 'f', 'e', 'f', 'f', '4', '7', 'd' },
    t56[8] = { '8', '5', '8', '4', '5', 'd', 'd', '1' },
    t57[8] = { '6', 'f', 'a', '8', '7', 'e', '4', 'f' },
    t58[8] = { 'f', 'e', '2', 'c', 'e', '6', 'e', '0' },
    t59[8] = { 'a', '3', '0', '1', '4', '3', '1', '4' },
    t60[8] = { '4', 'e', '0', '8', '1', '1', 'a', '1' },
    t61[8] = { 'f', '7', '5', '3', '7', 'e', '8', '2' },
    t62[8] = { 'b', 'd', '3', 'a', 'f', '2', '3', '5' },
    t63[8] = { '2', 'a', 'd', '7', 'd', '2', 'b', 'b' },
    t64[8] = { 'e', 'b', '8', '6', 'd', '3', '9', '1' };
int in, n = 0, i, j, addup;
int md5[512],
    /*每一大轮 m[]的调用顺序*/
    m1[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
    m2[16] = { 1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12 },
    m3[16] = { 5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2 },
    m4[16] = { 0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9 },
    /* a[]、b[]、c[]、d[]的初始值(已经过大小端处理) */
    /* 把 a[]、b[]、c[]、d[]赋值给 a1[]、b1[]、c1[]、d1[]*/
    a[32] = { 0, 1, 1, 0, 0, 1, 1, 1,
    0, 1, 0, 0, 0, 1, 0, 1,
    0, 0, 1, 0, 0, 0, 1, 1,
    0, 0, 0, 0, 0, 0, 0, 1 },
    a1[32] = { 0, 1, 1, 0, 0, 1, 1, 1,
    0, 1, 0, 0, 0, 1, 0, 1,
    0, 0, 1, 0, 0, 0, 1, 1,
    0, 0, 0, 0, 0, 0, 0, 1 },
    b[32] = { 1, 1, 1, 0, 1, 1, 1, 1,
    1, 1, 0, 0, 1, 1, 0, 1,
    1, 0, 1, 0, 1, 0, 1, 1,
    1, 0, 0, 0, 1, 0, 0, 1 },
    b1[32] = { 1, 1, 1, 0, 1, 1, 1, 1,
    1, 1, 0, 0, 1, 1, 0, 1,
    1, 0, 1, 0, 1, 0, 1, 1,
    1, 0, 0, 0, 1, 0, 0, 1 },
    c[32] = { 1, 0, 0, 1, 1, 0, 0, 0,
    1, 0, 1, 1, 1, 0, 1, 0,
    1, 1, 0, 1, 1, 1, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 0 },
    c1[32] = { 1, 0, 0, 1, 1, 0, 0, 0,
    1, 0, 1, 1, 1, 0, 1, 0,
    1, 1, 0, 1, 1, 1, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 0 },
    d[32] = { 0, 0, 0, 1, 0, 0, 0, 0,
```

```
        0, 0, 1, 1, 0, 0, 1, 0,
        0, 1, 0, 1, 0, 1, 0, 0,
        0, 1, 1, 1, 0, 1, 1, 0 },
        d1[32] = { 0, 0, 0, 1, 0, 0, 0, 0,
        0, 0, 1, 1, 0, 0, 1, 0,
        0, 1, 0, 1, 0, 1, 0, 0,
        0, 1, 1, 1, 0, 1, 1, 0 };

    ch = x;
    while (ch != '\n' && n < 57)
    {
        in = (int)ch;
        shizhuaner(in, n, md5);
        n++;
        ch = x;
    }
    i = 0;
    addup = n;
    while (n % 4 != 0 && n < 56)  //长度不是4的倍数，补一个1和0直到长度为
4的倍数,，最终实现用1与0使其长度模512与448同于，在这个程序里也就是448
    {
        int s, w, j;
        s = n / 4 + 1;
        w = n % 4;
        j = 1;
        do
        {
            md5[32 * s - 8 * w - j] = 0;
            j++;
        } while (j <= 7);
        if (i == 0)
        {
            md5[32 * s - 8 * w - j] = 1;
            i = 1;
        }
        n++;
    }
    if (i == 0)  //长度不是4的倍数，补一个1和31个0
    {
        for (j = 0; j < 32; j++)
            md5[n * 8 + j] = 0;
        md5[8 * n + 24] = 1;
    }
    for (i = 0; i < 512; i++)  //补零，任何不为1的数都设为0
```

```
        {
            if (md5[i] == 1)
                md5[i] = 1;
            else
                md5[i] = 0;
        }

        shizhuaner_weishu(addup * 8, md5);  //64 位数填充

        /*若想看 m[0]~m[15],把下面注释去掉即可*/
        /*printf("m[0]~m[15]如下:\n");
        for (i = 0; i < 512; i++)
        {
            printf("%d ", md5[i]);
            if (i % 8 == 7)
                printf("\n");
            if (i % 32 == 31)
                printf("\n");
        }
        printf("\n");*/

        /* 第一、二、三、四大轮，每一大轮下有 16 小轮 */
        round(a, b, c, d, m1, md5, 1, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11,
t12, t13, t14, t15, t16);
        round(a, b, c, d, m2, md5, 2, t17, t18, t19, t20, t21, t22, t23, t24, t25,
t26, t27, t28, t29, t30, t31, t32);
        round(a, b, c, d, m3, md5, 3, t33, t34, t35, t36, t37, t38, t39, t40, t41,
t42, t43, t44, t45, t46, t47, t48);
        round(a, b, c, d, m4, md5, 4, t49, t50, t51, t52, t53, t54, t55, t56, t57,
t58, t59, t60, t61, t62, t63, t64);

        /* 最终的 a、b、c、d 分别与最初的 a、b、c、d 相加 */
        jia(a, a1, a);
        jia(b, b1, b);
        jia(c, c1, c);
        jia(d, d1, d);
        /*密文输出*/
        //printf("密文:\n");
        int xxx;
        xxx=c_out(a);
        //c_out(b);
        //c_out(c);
        //c_out(d);
        //printf("\n");
```

```
    return xxx;
}
```

```
/*-------------------------------------------------------------*/
/*            SMC                                              */
/*-------------------------------------------------------------*/

void xorPlus(char* soure, int dLen, char* Key, int Klen)
{
    if (!Klen)
        return;
    for (int i = 0; i < dLen;)
    {
        for (int j = 0; (j < Klen) && (i < dLen); j++, i++)
        {
            soure[i] = soure[i] ^ Key[j];
            soure[i] = ~soure[i];
        }
    }//
}
void SMC(char* pBuf, char* key)
{
    // SMC 加密 XX 区段
    const char* szSecName = ".SMC";
    short nSec;
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS pNtHeader;
    PIMAGE_SECTION_HEADER pSec;
```

```
        pDosHeader = (PIMAGE_DOS_HEADER)pBuf;
        pNtHeader = (PIMAGE_NT_HEADERS)& pBuf[pDosHeader->e_lfanew];
        nSec = pNtHeader->FileHeader.NumberOfSections;
        pSec   =   (PIMAGE_SECTION_HEADER)&   pBuf[sizeof(IMAGE_NT_HEADERS)   +
pDosHeader->e_lfanew];
        for (int i = 0; i < nSec; i++)
        {
            if (strcmp((char*)& pSec->Name, szSecName) == 0)
            {
                int pack_size;
                char* packStart;
                pack_size = pSec->SizeOfRawData;
                packStart = &pBuf[pSec->VirtualAddress];

    //VirtualProtect(packStart,pack_size,PAGE_EXECUTE_READWRITE,&old);
                xorPlus(packStart, pack_size, key, strlen(key));
                //MessageBox(_T("SMC 解密成功。"));
                //MessageBox(NULL, TEXT("解密成功"), TEXT("1"), MB_OK);
                return;
            }
            pSec++;
        }
    }

    void UnPack()
    {
        char* Key = "12345";
        char* hMod;
        hMod = (char*)GetModuleHandle(0);
        SMC(hMod, Key);
    }


    /*-------------------------------------------------------------*/
    /* GenRegCode — 注册算法主函数                                 */
    /*-------------------------------------------------------------*/


    BOOL GenRegCode( TCHAR  *rCode, TCHAR  *name ,int len)
    {
    int i,j;
    unsigned long code=0;
    char Table[10];
    long x=0;
```

```
    _FLOWER_XX00;
    ::GetVolumeInformation("C:\\", namebuf, namesize, &serialnumber, &maxlen,
&fileflag, sysnamebuf, sysnamesize);
    itoa(serialnumber, Table, 10);
    _FLOWER_XX00;
    for (i = 0, j = 0; j < 10; i++, j++)
    {
        _FLOWER_XX02;
        if (i > len - 1)
        {
            _FLOWER_XX00;
            i = 0;
            _FLOWER_XX00;
        }
        _FLOWER_XX03;
        code += ((BYTE)name[i]) ^ Table[j];
    }


    code = 2017 * code;
    x=MD5((char)code);
    code = x * code;
    code = code + 58;
    code = code + 69;
    code = code + 83;
    code = code + 107;

    code = code * code;

        _FLOWER_XX02;
        wsprintf(name,TEXT("%ld"),code);
        _FLOWER_XX01;
        if(lstrcmp(rCode, name)==0)
            return TRUE;
        else
        {
        //  MessageBox(NULL,  name, TEXT("序列号"), MB_OK);
                return FALSE;
        }
    }
```

2）SMC pack.cpp

```
    // SMC packDlg.cpp : 实现文件
    //
```

```
#include "stdafx.h"
#include "SMC pack.h"
#include "SMC packDlg.h"
#include "afxdialogex.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// 用于应用程序“关于”菜单项的 CAboutDlg 对话框

class CAboutDlg : public CDialogEx
{
public:
    CAboutDlg();

// 对话框数据
    enum { IDD = IDD_ABOUTBOX };

    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV 支持

// 实现
protected:
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialogEx::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialogEx)
END_MESSAGE_MAP()


// CSMCpackDlg 对话框
```

```
CSMCpackDlg::CSMCpackDlg(CWnd* pParent /*=NULL*/)
    : CDialogEx(CSMCpackDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CSMCpackDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialogEx::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CSMCpackDlg, CDialogEx)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BUTTON1, &CSMCpackDlg::OnBnClickedButton1)
    ON_BN_CLICKED(IDC_BUTTON2, &CSMCpackDlg::OnBnClickedButton2)
END_MESSAGE_MAP()


// CSMCpackDlg 消息处理程序

BOOL CSMCpackDlg::OnInitDialog()
{
    CDialogEx::OnInitDialog();

    // 将“关于...”菜单项添加到系统菜单中。

    // IDM_ABOUTBOX 必须在系统命令范围内。
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        BOOL bNameValid;
        CString strAboutMenu;
        bNameValid = strAboutMenu.LoadString(IDS_ABOUTBOX);
        ASSERT(bNameValid);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
```

```
                pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
            }
        }

        // 设置此对话框的图标。当应用程序主窗口不是对话框时，框架将自动
        //  执行此操作
        SetIcon(m_hIcon, TRUE);          // 设置大图标
        SetIcon(m_hIcon, FALSE);         // 设置小图标

        // TODO: 在此添加额外的初始化代码

        return TRUE;  // 除非将焦点设置到控件，否则返回 TRUE
    }

    void CSMCpackDlg::OnSysCommand(UINT nID, LPARAM lParam)
    {
        if ((nID & 0xFFF0) == IDM_ABOUTBOX)
        {
            CAboutDlg dlgAbout;
            dlgAbout.DoModal();
        }
        else
        {
            CDialogEx::OnSysCommand(nID, lParam);
        }
    }

    // 如果向对话框添加最小化按钮，则需要下面的代码
    //  来绘制该图标。对于使用文档/视图模型的 MFC 应用程序，
    //  这将由框架自动完成。

    void CSMCpackDlg::OnPaint()
    {
        if (IsIconic())
        {
            CPaintDC dc(this); // 用于绘制的设备上下文

            SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

            // 使图标在工作区矩形中居中
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
```

```
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // 绘制图标
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialogEx::OnPaint();
    }
}

//当用户拖动最小化窗口时系统调用此函数取得光标
//显示。
HCURSOR CSMCpackDlg::OnQueryDragIcon()
{
    return static_cast<HCURSOR>(m_hIcon);
}



void CSMCpackDlg::OnBnClickedButton1()
{
    // TODO: 在此添加控件通知处理程序代码
    CFileDialog f(TRUE);
    if(f.DoModal()==IDOK)
        SetDlgItemText(IDC_EDIT1,f.GetPathName());

}
void xorPlus(char *soure,int dLen,char *Key,int Klen)
{

    for (int i=0;i<dLen;)
    {
        for (int j=0;(j<Klen) && (i<dLen);j++,i++)
        {
            soure[i]=soure[i] ^ Key[j];
            soure[i]=~soure[i];
        }
    }
}
void SMC(HANDLE hFile,char *key)
{
```

```
        // SMC 加密 XX 区段
        HANDLE hMap;
        const char *szSecName = ".SMC";
        char *pBuf;
        int size;
        short nSec;
        PIMAGE_DOS_HEADER pDosHeader;
        PIMAGE_NT_HEADERS pNtHeader;
        PIMAGE_SECTION_HEADER pSec;

        size = GetFileSize(hFile,0);
        hMap=CreateFileMapping(hFile,NULL,PAGE_READWRITE,0,size,NULL);
        if (hMap==INVALID_HANDLE_VALUE)
        {
    _viewf:
            AfxMessageBox(_T("映射失败"));
            return ;
        }
        pBuf=(char *)MapViewOfFile(hMap,FILE_MAP_WRITE|FILE_MAP_READ,0,0,size);
        if(!pBuf) goto _viewf;
        pDosHeader=(PIMAGE_DOS_HEADER)pBuf;
        pNtHeader=(PIMAGE_NT_HEADERS)&pBuf[pDosHeader->e_lfanew];
        if (pNtHeader->Signature!=IMAGE_NT_SIGNATURE)
        {
            AfxMessageBox(_T("不是有效的 win32 可执行文件"));
            goto _clean;
        }
        nSec=pNtHeader->FileHeader.NumberOfSections;
        pSec=(PIMAGE_SECTION_HEADER)&pBuf[ sizeof(IMAGE_NT_HEADERS)+pDosHeader-
>e_lfanew];
        for (int i=0;i<nSec;i++)
        {
            if (strcmp((char *)&pSec->Name,szSecName)==0)
            {
                int pack_size;
                char *packStart;
                pack_size=pSec->SizeOfRawData;
                packStart = &pBuf[pSec->PointerToRawData];
                xorPlus(packStart,pack_size,key,strlen(key));
                AfxMessageBox(_T("SMC 加密成功。"));
                goto _clean;
            }
            pSec++;
        }
```

```
        AfxMessageBox(_T("未找到 .SMC 段"));
    _clean:
        UnmapViewOfFile(pBuf);
        CloseHandle(hMap);
        return ;
    }

    void CSMCpackDlg::OnBnClickedButton2()
    {
        // TODO: 在此添加控件通知处理程序代码
        HANDLE hFile;
        CString filepath;
        char* KeyBuffer="12345";
        GetDlgItemTextW(IDC_EDIT1,filepath);
        //GetWindowTextA(GetDlgItem(IDC_EDIT2)->m_hWnd,KeyBuffer,MAX_PATH);
        //if (strlen(KeyBuffer)==0)
        //{
        //  AfxMessageBox(_T("请输入 Key"));
        //  return ;
        //}
        hFile=CreateFile(filepath.GetBuffer(),GENERIC_READ|GENERIC_WRITE,0,NULL
,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
        if (hFile==INVALID_HANDLE_VALUE)
        {
            AfxMessageBox(_T("打开文件失败!"));
            return ;
        }
        SMC(hFile,KeyBuffer);
        CloseHandle(hFile);
    }
```