Практический курс «Основы защиты приложений при помощи Guardant API»

Урок 1

Основы Guardant API для защиты приложений (C#)

Содержание

1. Постановка задачи и планирование ее решения

Тестовое приложение

В качестве тестового приложения выбрана программа, представляющая собой графический интерфейс работы с базой данных (БД) MS Access, содержащей список клиентов и информацию о них.

Постановка задачи

1. Начать работу с Guardant API.
2. Найти ключ по заданным параметрам.
3. Разработать простейшую систему защиты на основе одной защищенной ячейки.
4. Корректно завершить работу с Guardant API.
5. В случае возникновения ошибок корректно их обработать.

Действия необходимые для решения задачи

<u>При запуске программы</u>

1. Инициализация Guardant API (*GrdApi.GrdStartup*)
2. Создание хэндла контейнера Guardant (*GrdApi.GrdCreateHandle*)
3. Установка кодов доступа к ключу (*GrdApi.GrdSetAccessCodes*)
4. Построение списка доступных ключей (*GrdApi.GrdFind*)
5. Модификация поиска ключа. Использование флагов поиска (*GrdApi.GrdSetFindMode*)
6. Подключение к выбранному ключу (*GrdApi.GrdLogin*)

<u>При завершении работы программы</u>

1. Завершение работы с ключом (*GrdApi.GrdLogout*)
2. Закрытие хэндла контейнера Guardant. Освобождение памяти (*GrdApi.GrdCloseHandle*)
3. Деинициализация Guardant API (*GrdApi.GrdCleanup*)

<u>При вызове функции поиска телефона клиента по БД</u>

1. Первичная активация защищенной ячейки (*GrdApi.GrdPI_Activate*)
2. Чтение данных из защищенной ячейки (*GrdApi.GrdApi.GrdPI_Read*)
3. Проверка корректности данных в защищенной ячейке
4. Обновление данных в защищенной ячейке (*GrdApi.GrdPI_Update*)

<u>При вызове функции поиска имени клиента по БД</u>

1. Попытка чтения из защищенной ячейки (*GrdApi.GrdRead*)
2. Установка адресации памяти ключа в режим SAM (*GrdApi.GrdSetWorkMode*)
3. Чтение поля содержащего ID ключа (*GrdApi.GrdRead*)

2. Описание решения задачи

2.1 Инициализация API

Для начала работы с ключами Guardant необходимо иметь хотя бы один электронный ключ и коды доступа к нему, а так же скачать и установить Комплект разработчика. Он распространяется бесплатно и всегда доступен для скачивания с сайта www.guardant.ru. В процессе установки комплекта разработчика необходимо указать коды доступа к вашим электронным ключам.

Комплект разработчика установлен, ключ вставлен в USB-порт компьютера, можно приступать к разработке защиты программного обеспечения.

Чтобы обращаться к ключу Guardant, необходимо подключить к проекту специальную библиотеку. В различных средах разработки программного обеспечения это делается разными способами.

В частности, в Visual C# для этого нужно:

1. найти в установленном комплекте разработчика две библиотеки «*GuardantDotNetApi.dll*» и «GrdAPI32.dll» (они находятся в папке «Lib»), они нужны для работы с Guardant API. Данные библиотеки должны присутствовать в папке с откомпилированной программой. Возможен вариант, когда для удобства работы данные файлы при инсталляции будут скопированы в папку системы «Windows»;
2. через «*Solution Explorer*» добавить к блоку «*References*» ссылку на «*GuardantDotNetApi*». Для этого нужно нажать правой кнопкой мыши на «*References*», выбрать «*Add Reference..*», перейти в папку с библиотеками (пункт 1) и выбрать прокси библиотеку «*GuardantDotNetApi.dll*»;
3. добавить «*Guardant*» к списку используемых пространств имен (***using*** *Guardant*).

> Файлы для этой и других сред программирования можно найти в папке «Lib» Комплекта разработчика.

Перед началом работы с ключом Guardant нужно запрограммировать его под нужды нашей защиты, делается это с помощью утилиты программирования ключей Guardant (*GrdUtil.exe*). Программирование ключей происходит посредством создания маски ключа и последующей ее записи в один или несколько ключей. Говоря простым языком, маска ключа – это образ всех полей ключа (их структуры и содержимого). Более подробно с составом маски, мы познакомимся в следующих частях. Чтобы не тратить время на создание маски вручную, воспользуемся уже созданным для нашего урока файлом маски ключа «*lesson1.nsd*». Запустите утилиту «Программирование ключей Guardant» и откройте файл маски «*lesson1.nsd*» (*Файл → Открыть)*, после этого запишите данную маску в ваш ключ (*Ключ → Запись в ключ)*.

Теперь все готово для работы непосредственно с Guardant API.

Для начала работы с ключом защиты необходимо провести инициализацию нашей копии Guardant API, это делается с помощью метода **GrdApi.GrdStartup().** Единственный параметр данного метода позволяет выбрать тип ключей, с которыми мы будет работать. Это могут быть локальные ключи, удаленные ключи, или и те и другие. В данном уроке наша задача научиться использовать локальные ключи, поэтому в качестве параметра укажем «**GrdFMR.Local**».

```
RetCode = GrdApi.GrdStartup(GrdFMR.Local);
```

Завершение работы с Guardant API осуществляется методом **GrdApi.GrdCleanup()**, данный метод должен быть обязательно вызван перед закрытием приложения или когда более не планируется работать с ключами Guardant. В нашем случае, мы поместим вызов данного метода в блок закрытия главной формы приложения.

```
RetCode = GrdApi.GrdCleanup();
```

Обратите внимание, что после каждого вызова функции Guardant API рекомендуется проверять код возврата для обработки возможных ошибок. В качестве основы функции обработки ошибок, мы воспользуемся слегка модифицированной функцией **ErrorHandling**. Данная функция используется в демонстрационном приложении, которое можно найти в Комплекте разработчика. Все возможные коды ошибок с подробными описаниями можно найти в справке по Guardant API.

Грамотная обработка ошибок в программе – признак хорошего стиля программирования. Но стоит учесть, что при использовании одной и той же функции обработки ошибок вы можете дать взломщику подсказку к расположению всех вызовов Guardant API. Не забывайте про это! Используйте различные способы обработки ошибок, как с помощью специальных функций, так и в коде, непосредственно после вызова функции API.

Следующим шагом на пути к работе с электронным ключом является создание хэндла, по которому будет происходить обращение к ключу. Делается это методом **GrdApi.GrdCreateHandle().** Под хэндл необходимо выделить память, это можно сделать самостоятельно. Однако, в данном случае, из-за сборщика мусора возможен вариант, когда хэндл начнет «плавать» в памяти, что значительно затруднит работу с ним. Оставим выделение памяти на откуп Guardant API, который в Visual C# прекрасно справляется с этим самостоятельно. Важным параметром метода является определение режима, в котором будет использоваться хэндл. Это может быть монопольный или многопоточный режим. При использовании многопоточного режима внутри API создается критическая секция, через которую происходит синхронизация обращений к ключу из разных потоков, из-за чего приложение может работать медленнее. Поэтому, если использовать многопоточность не планируется, правильным будет выбор монопольного режима работы. В тестовом приложении выбран режим *GrdCHM.MultiThread.* Это связано с тем, что, возможно, в следующих уроках для демонстрации способов усиления защиты будет использована многопоточность.

```
GrdHandle = GrdApi.GrdCreateHandle(GrdCHM.MultiThread);
```

В конце работы с Guardant API необходимо будет закрыть хэндл и освободить выделенную под него память. Делается это методом **GrdApi.GrdCloseHandle()**. Чтобы не забыть, сразу поместим вызов в блок завершения работы программы.

```
RetCode = GrdApi.GrdCloseHandle(GrdHandle);
```

Хэндл, по которому мы будем обращаться к ключу, создан. Теперь нужно установить коды доступа к ключу (метод **GrdApi.GrdSetAccessCodes**). У ключа Guardant есть несколько уровней привилегий, каждый из которых снабжен собственным кодом доступа.

Данный метод позволяет установить 4 кода доступа:

1. Public Code – публичный код, используется для поиска ключа *(должен быть задан обязательно)*.
2. Private Read Code – код для чтения данных из ключа *(должен быть задан обязательно)*.
3. Private Write Code – код для записи данных в ключ, в большинстве случаев, его использование не требуется, поэтому в качестве параметра указываем произвольное число (к примеру, 0).
4. Private Master Code – мастер код, используется для редактирования параметров ключа. Не используйте данный код в защищаемых приложениях во избежание попадания его в руки злоумышленников.

Все коды передаются в метод в числовом виде. Чтобы затруднить злоумышленнику поиск вызовов Guardant API в коде защищенного приложения, замаскируем коды доступа простейшим способом: вычтем константу из кода доступа, а затем прибавим ее непосредственно в параметрах метода.

```
RetCode = GrdApi.GrdSetAccessCodes (GrdHandle, PublicCode + CryptPU,
ReadCode + CryptRD, 0, 0);
```

Для усиления защиты можно дополнительно встроить в программу проверку модификации кодов доступа к ключу. Хотя метод **GrdApi.GrdSetAccessCodes** относится к блоку инициализации, он может вызываться неоднократно, и устанавливать нужные коды доступа непосредственно перед вызовом нужных методов. Стоить отметить, что при повторных вызовах метода, если код в параметре установлен равным 0, то он останется в памяти неизмененным, а при установке любого другого числа, он будет перезаписан. Многократный вызов метода **GrdApi.GrdSetAccessCodes** с различными кодами может затруднить поиски взломщиком реальных кодов доступа.

2.2 Поиск и подключение к ключу

Для поиска всех ключей подключенных к компьютеру, в классе API Guardant предназначен метод **GrdApi.GrdFind()**. Воспользуемся им для построения списка доступных ключей. В качестве первого параметра метода указывается хэндл контейнера Guardant, второй параметр устанавливает тип поиска (*GrdF_First* – поиск первого ключа, *GrdF_Next* – следующие ключи).

> **RetCode = GrdApi.GrdFind(GrdHandle, GrdF.First, out DongleID, out GrdFindInfo1);**

В случае успешного вызова метода, в третьем параметре будет ID ключа и в четвертом - подробная информация о ключе. Для наглядной демонстрации просмотра подробной информации о ключе сохраним структуру **GrdFindInfo1** после первого вызова функции поиска.

Для наглядной демонстрации всех подключенных к компьютеру ключей добавим к программе вывод ID найденного ключа через MessageBox.Show().

Мы получили список ключей Guardant, подключенных к компьютеру, которые используют установленные нами коды доступа. Но что делать, если у вас целый ряд различных программных разработок и ключи к ним разные, а коды доступа одни и те же? Как выбрать ключ, который действительно нужен, и как к нему подключиться? Для этого в ключе Guardant предусмотрены целый ряд полей общего назначения, с помощью них можно задать дополнительные параметры поиска ключа. Полный список возможных параметров: «номер программы», «ID ключа», «серийный номер», «версия», «битовая маска», «тип ключа», «модель ключа», «интерфейс ключа».

Эти же параметры содержатся в подробной информации о ключе (**GrdFindInfo1)**, которую мы сохранили после первого запуска метода поиска. Выведем ее на экран и посмотрим, какие же данные записаны в нашем ключе. Для этого разберем структуру на составляющие и покажем ее через метод MessageBox.Show().

Один из параметров в данной структуре - это ID ключа. Он уникален для всех ключей линейки Guardant. Поэтому его можно использовать при построении системы защиты для дорогих программ. Если вы продаете единичные копии программного обеспечения, компилируете и защищаете программу для каждого конкретного клиента, то маска, помещаемая в электронный ключ также должна быть уникальна для каждого клиента. В этом случае необходимо будет искать ключ по его ID.

Давайте ограничим поиск ключа по номеру программы и битовой маске. В самом начале программы мы записали в память ключа маску из файла «*lesson1.nsd*». В ней заданы следующие параметры: *«Номер программы» = 5, «Битовая маска» = 77 (BIN=1001101).*

Параметр поиска «Битовая маска» удобно использовать, когда к ключу привязываются несколько разных приложений или несколько модулей одного приложения (то есть каждый электронный ключ может поддерживать произвольный набор из существующих модулей приложения). В таком случае в маске ключа создаются поля

(защищенные ячейки или дескрипторы алгоритмов) для каждого из модулей. Единицы в битовой маске ключа при этом соответствуют модулям, поля для которых были добавлены в маску, и которые будут реально работать с данным электронным ключом. К примеру, если в ключе задана битовая маска *1001101*, то в нем есть алгоритмы для 1,3,4 и 7 (справа-налево) функциональных модулей. Соответственно, если для работы приложения требуются 1-й и 4-й, то задав в параметрах поиска битовую маску *1001*, мы найдем ключ с маской *1001101*, так как в нем есть нужные нам поля. В нашем примере будет проверяться полное соответствие битовой маски (битовая маска в ключе равна битовой маске в параметрах поиска).

Чтобы установить ограничения поиска ключа воспользуемся методом **GrdApi.GrdSetFindMode ().** Добавим в наше приложение следующий код

```
RemoteMode = GrdFMR.Local;                  // Поиск только локальных ключей
DongleFlags = GrdFM.NProg + GrdFM.Mask;     // Ищем по номеру программы и
                                            битовой маске
DongleID = 645090921;                       // ID ключа
ProgramNumber = 5;                          // Номер нашей программы
SerialNumber = 0;                           // Серийный номер
Version = 0;                                // Версия программы
BitMask = 77;                               // Битовая маска
DongleType = GrdDT.GSII64;                  // Ключ с поддержкой алгоритма
GSII64
DongleModel = GrdFMM.ALL;                   // Все модели ключей
DongleInterface = GrdFMI.ALL;               // Интерфейс подключения любой
                                            (LPT или USB)
```

```
RetCode = GrdApi.GrdSetFindMode(GrdHandle,
                    RemoteMode,
                    DongleFlags,
                    ProgramNumber,
                    DongleID,
                    SerialNumber,
                    Version,
                    BitMask,
                    DongleType,
                    DongleModel,
                    DongleInterface);
```

Несмотря на то, что помимо битовой маски и номера программы в параметрах поиска были заданы еще ID, серийный номер и версия, в качестве критериев поиска будут использованы только номер программы и битовая маска, так как только для них указаны флаги параметра **DongleFlags.**

Теперь при вызове методов **GrdApi.GrdFind()** или **GrdApi.GrdLogin()** будут найдены только соответствующие заданным критериям поиска ключи.

В виде тренировки установите ID вашего ключа в качестве параметра поиска.

Не стоит забывать о том, что заданные параметры поиска ключа не являются защитным механизмом, и, равно как результат поиска ключа, не могут являться основой для организации логики защиты.

Использование метода поиска ключей **GrdApi.GrdFind()** в приложении необязательно. Вполне достаточно ограничить круг возможных ключей через метод **GrdApi.GrdSetFindMode(),** и после этого сразу же можно производить подключение к ключу с помощью **GrdApi.GrdLogin()**.

Метод подключения к электронному ключу **GrdApi.GrdLogin()** имеет ряд параметров. Первый - это хэндл контейнера, с которым мы уже встречались не раз. Второй и третий параметр для локальных ключей не используются. Но если они вас заинтересовали, то об этом всегда можно прочитать в справочной системе по Guardant API.

Выполним подключение к ключу.

```
RetCode = GrdApi.GrdLogin(GrdHandle, 0xFFFFFFFF, GrdLM.PerStation);
```

Наравне с методом подключения к ключу, есть метод отключения от ключа. Он особенно актуален при использовании сетевых ключей. При его вызове происходит освобождение лицензии, занятой копией нашего приложения в сетевом ключе при вызове **GrdLogin**. Добавим метод **GrdLogout** в блок завершения работы нашей программы.

```
RetCode = GrdApi.GrdLogout(GrdHandle);
```

На этом подготовительный этап работы с ключом завершен, и можно приступать непосредственно к программированию логики защиты приложения.

2.3 Чтение и запись информации. Начальный уровень защиты

Прежде чем говорить о методах чтении и записи информации в ключ Guardant, давайте познакомимся с защищенными ячейками. Защищенная ячейка - это область памяти в ключе Guardant, куда может быть помещена и откуда считана различная информация. В чем преимущество защищенной ячейки для хранения данных перед обычной памятью ключа? Защищенной ячейке доступны следующие сервисы: ячейка может быть активирована (доступна для чтения и записи) и деактивирована. Для активации-деактивации устанавливаются отдельные пароли. Возможна также установка пользовательских паролей на чтение и запись информации в ячейку. Защищенные ячейки - это отличный способ хранения каких-либо важных данных или лицензионной информации.

После того как в части 1 мы записали маску *«lesson1.nsd»* в ключ защиты, там находится одна защищенная ячейка с названием «*Урок 1»* и длинной 32 символа. У данной ячейки установлены следующие пароли доступа: активации 408956937,

чтения 1469608341, записи 843783688. В настоящее время в ячейке находится фраза «HelloWord!», ячейка деактивирована.

Все защищенные ячейки и аппаратные алгоритмы, находящиеся в памяти ключа, нумеруются, начиная с 0. Именно по этому номеру и происходит обращение. Наша первая защищенная ячейка имеет номер 0.

Эксперименты с защитой будем ставить на событии поиска по номеру телефона. Первый метод, которым мы научимся пользоваться, - это **GrdApi.GrdPI_Read()** – чтение защищенной ячейки ключа Guardant.

```
RetCode = GrdApi.GrdPI_Read(
    GrdHandle,          // Хэндл контейнера Guardant
    0,                  // Номер защищенной ячейки
    0,                  // Смещение, с которого будет производиться чтение в
                        защищенной ячейке
    32,                 // Количество байт для чтения
    out PIData,         // Указатель на буфер считанных данных
    1469608341);        // Пароль на чтение защищенной ячейки
```

Если попробовать прочитать ячейку сейчас, то будет получена ошибка **GrdE.InactiveItem**. Это произошло потому, что наша ячейка не активирована. Добавляем в код проверку на активацию ячейки:

```
    if (RetCode == GrdE.InactiveItem)
```

Теперь, если ячейка не активирована, мы должны активировать ее и еще раз вызвать функцию чтения. Для активации ячейки в Guardant API предназначен метод **GrdApi.GrdPI_Activate().** Данный метод по установленному паролю активирует, как защищенные ячейки, так и аппаратные алгоритмы. С аппаратными алгоритмами мы познакомимся во втором уроке, а сейчас добавим к исходному коду приложения метод активации защищенной ячейки.

```
    RetCode = GrdApi.GrdPI_Activate(
        GrdHandle,          // Хэндл контейнера Guardant
        0,                  // Номер защищенной ячейки или алгоритма
        408956937);         // Пароль для активации ячейки или алгоритма
```

Как вы видите, для активации защищенной ячейки достаточно указать ее номер и установленный Вами код активации.

Проверяем содержимое ячейки после активации. Там должна находиться строка «HelloWord!» без кавычек. Если это не так, значит, защита была нарушена, и мы должны отобразить сообщение об этом и завершить работу программы.

Если в ячейке все-таки, как мы и ожидали, находится строка «HelloWord!», то обновим содержимое ячейки с помощью метода **GrdApi.GrdPI_Update.** Запишем в ячейку информацию о случайном клиенте из Базы Данных. Формат ячейки при этом будет

следующий: первые 4 байта это ID клиента из Базы данных. Остальные 28 байтов имя клиента. Приведу пример записи в защищенную ячейку.

```
RetCode = GrdApi.GrdPI_Update(
GrdHandle,          //Хэндл контейнера
0,                  //Номер защищенной ячейки или алгоритма
0,                  //Смещение защищенной ячей
32,                 //Количество байтов для записи
PIData,             //Указатель на буфер с данными
843783688,          //Пароль на запись
GrdUM.MOV);         //Метод обновления (MOV или XOR)
```

Поля метода записи в защищенную ячейку, по сути, аналогичны методу чтения. Единственное изменение - поле «Метод обновления», данное поле может принимать 2 типа значений *GrdUM_MOV* или *GrdUM_XOR*. Названия говорят сами за себя, первый способ обновления перезаписывает ячейку данными, второй - складывает их с содержимым по модулю два.

Мы познакомились с методами чтения и записи защищенных ячеек. Неописанной осталась лишь логика защитных механизмов данного примера. Она приведена в виде блок-схемы (Рис. 1).

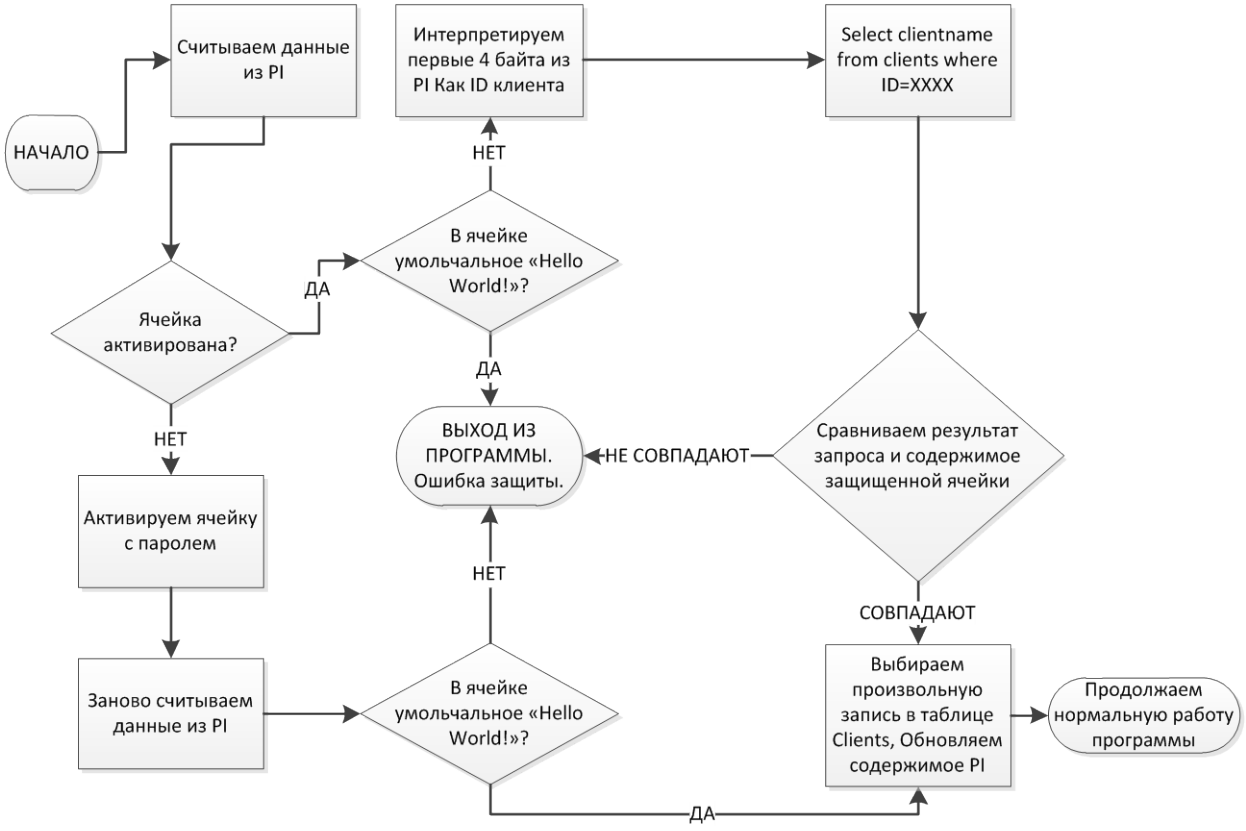

Рис. 1. Алгоритм работы с защищенной ячейкой (Protect Item – PI)

Внимание! Данный пример работы с БД является демонстрационным, не стоит необдуманно копировать все предлагаемые методы в реальное приложение. Обязательно учитывайте возможные сбои в работе реального приложения и базы данных. Помните, что защита программного обеспечения должна быть не только надежной, но и гибкой, устойчивой к любым сбоям реального приложения.

2.4 Дополнительная информация о чтении и записи

В предыдущей части мы познакомились с методами чтения и записи защищенных ячеек. Однако в Guardant API есть еще 2 метода чтения и записи **GrdApi.GrdRead()** и **GrdApi.GrdWrite()**. Они в основном используются для старых моделей ключей. Данные методы могут использоваться для считывания и записи адресов памяти, на которые не установлены программные запреты на чтение и запись.

Попробуем прочитать данные из нашей защищенной ячейки с помощью метода **GrdApi.GrdRead()**. Если взглянуть на маску нашего текущего ключа (сделать это можно в редакторе памяти ключа), видно, что защищенная ячейка «Урок 1» начинается в памяти с адреса 024. Добавим в блок поиска в БД по имени клиента следующий код

```
RetCode = GrdApi.GrdRead(GrdHandle,        / Хэндл контейнера
24,                  // Адрес памяти в ключе для начала считывания
52,                  // Размер блока данных для считывания (заголовок + содержимое ячейки)
out TempData);       // Переменная, куда будет помещен считанный блок
```

После выполнения данной функции, в переменной **ReadData** ничего не изменилось. Блок данных не прочитан. Ранее, для того чтобы получить всю информацию из ключа, достаточно было один раз запустить метод чтения и указать всю память ключа. В современных ключах Guardant перед взломщиками поставлен еще один рубеж обороны из защищенных ячеек.

Но метод **GrdApi.GrdRead** все-таки может вам пригодиться, например, как один из вариантов получения ID ключа. Чтобы это сделать для начала необходимо перевести адресацию памяти в режим SAM, для этого добавим в программу **GrdApi.GrdSetWorkMode().** В ключах Guardant используется два типа адресации памяти: SAM (System Address Mode) и UAM (User Address Mode). Чтобы понять разницу, достаточно представить себе память ключа. Первые 30 байт, записанные в ключе, относятся к системным (SAM) и не могут быть изменены, после них начинается область памяти, которая может изменяться пользователем (UAM). По умолчанию используется адресация UAM, поэтому, чтобы считать ID, нужно переключиться на SAM.

```
RetCode = GrdApi.GrdSetWorkMode(GrdHandle,      // Хэндл контейнера
GrdWM.SAM,                  // Тип адресации памяти
GrdWMFM.DriverAuto);        // Режим работы с драйвером
```

После этого можно производить чтение ID ключа. Длина ID составляет 4 байта, в качестве указателя на адрес памяти используем стандартную константу **GrdSAM.dwID**. С помощью таких же констант можно считать из ключа номер и версию программы, битовую маску и другие поля, рассмотренные нами в Части 2.

```
RetCode = GrdApi.GrdRead(GrdHandle,        //Хэндл контейнера
GrdSAM.dwID,               //Константа-указатель на ID
out TempDataID);           //Переменная, куда будет помещен считанный блок
```

2.5 Деинициализация Guardant API

Все моменты, касающиеся деинициализации и корректного завершения работы с Guardant API, упоминались в Части 1 данного урока. Рассмотрим их подробнее.

Завершение работы с Guardant API начинаем с метода закрытия сеанса работы ключом защиты.

```
RetCode = GrdApi.GrdLogout(GrdHandle);
```

Данный метод особенно актуален при работе с сетевыми ключами. В сетевом ключе для определения количества текущих пользователей используется переменная ресурса ключа, это количество пользователей, которые могут одновременно подключиться к данному ключу. Если завершение работы с сетевым ключом происходит некорректно, то Guardant API на протяжении 15 минут продолжит считать, что пользователь все еще работает с ключом. При этом возможна ситуация, что ресурс ключа будет исчерпан «мертвыми» соединениями и реальные пользователи не смогут к нему подключиться.

Следующий шаг – закрытие хэндла. При создании контейнера Guardant API происходит выделение памяти и, если работа завершена некорректно, то высока вероятность появления ошибок связанных с так называемой «утечкой памяти».

```
RetCode = GrdApi.GrdCloseHandle(GrdHandle);
```

Последний шаг – деинициализация копии Guardant API.

```
RetCode = GrdApi.GrdCleanup();
```

Только после выполнения всех этих методов, можно сказать, что работа с Guardant API завершена грамотно и корректно.

Для закрепления материала ознакомьтесь с примером кода корректного завершения работы с Guardant API.

```
if (GrdHandle.Address != 0) // Проверяем был ли создан хэндл
    {
        // Закрывает сеанс работы с ключом
        RetCode = GrdApi.GrdLogout(GrdHandle);
        ErrorHandling(GrdHandle, RetCode);

        // Закрытие хэндла контейнера Guardant.
        //Выход с сервера ключа и освобождение памяти
        RetCode = GrdApi.GrdCloseHandle(GrdHandle);
        ErrorHandling(GrdHandle, RetCode);
    }

    // Деинициализация копии GrdAPI.
    RetCode = GrdApi.GrdCleanup();
```

3. Несколько важных моментов

В завершение хочется еще раз напомнить несколько ключевых моментов в разработке защиты программного обеспечения на основе ключей защиты Guardant.

Во-первых! Обработка ошибок в программе – признак хорошего стиля программирования. Однако, при реализации защиты приложения, логика проверки ошибок не должна быть однообразной. Это позволит затруднить злоумышленнику анализ логики защитных механизмов.

Во-вторых! Параметры поиска ключа не являются защитным механизмом, и, равно как результат поиска ключа, не могут являться основой для организации логики защиты.

В-третьих! Не копируйте необдуманно все возможные варианты защитных механизмов. Помните, защита должна быть не только надежной, но и устойчивой к любым внешним чрезвычайным происшествиям, будь то сбои в работе базы данных или самой программы. В частности, аварийное завершение нашего приложения или нарушение целостности базы данных может привести к невозможности дальнейшей работы с программой.

Какие защитные механизмы будут использоваться и как – зависит от вас.