Практический курс «Основы защиты приложений при помощи Guardant API»

Урок 1

Основы Guardant API для защиты приложений (Delphi)

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

<u>При завершении работы программы</u>

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

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

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

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

1. Попытка чтения из защищенной ячейки (*GrdRead*)
2. Установка адресации памяти ключа в режим SAM (*GrdSetWorkMode*)
3. Чтение поля содержащего ID ключа (*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 модули Guardant API. В различных средах разработки программного обеспечения это делается разными способами. В частности, в Delphi для этого нужно подключить к программе модуль «*GrdDelphi*». Для этого в настройках Delphi необходимо добавить к списку поиска путь к папке «Lib» из установленного Комплекта разработчика.

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 функции **GrdStartupEx()**. Данная функция позволяет указать 3 параметра. Первый параметр 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**». Второй параметр, позволяет указать путь к файлу сетевых настроек клиента *«gnclient.ini»,* т.к. используется подключение к локальному ключу, в качестве параметра укажем **nil**. Третий параметр является резервным и не используется.

nRet := GrdStartupEx(GrdFMR_Local, nil, nil);

Завершение работы с Guardant API осуществляется функцией **GrdCleanup**, данная функция должна быть обязательно вызвана перед закрытием приложения или когда более не планируется работать с Guardant API. В нашем случае, мы поместим ее в блок закрытия главной формы приложения.

nRet := 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 с помощью функции **GrdCreateHandle.** Первым параметром данной функции является указатель на область памяти, где будет размещен контейнер Guardant. Память можно выделить вручную, но тогда придется при завершении работы программы также вручную ее освобождать. Выделение памяти можно отдать на откуп Guardant API, который прекрасно справляется с этим самостоятельно, поэтому в качестве параметра укажем *«nil»*. Вторым важным параметром данной функции 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 этого 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.

hGrd := GrdCreateHandle(nil, GrdCHM_MultiThread, nil);

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

nRet:=GrdCloseHandle(hGrd);

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

Данная функция 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 функцию 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 функции.

nRet:= GrdSetAccessCodes(hGrd, dwPublic + CryptPU, dwPrivRD + CryptRD, 0, 0)

Для усиления защиты можно дополнительно встроить в программу проверку модификации кодов доступа к ключу.
Хотя функция **GrdSetAccessCodes** относится к блоку инициализации, она может вызываться неоднократно, и устанавливать нужные коды доступа непосредственно перед вызовом нужных функций. Стоить отметить, что при повторных вызовах функции, если код в параметре установлен равным 0, то он останется в памяти неизмененным, а при установке любого другого числа, он будет перезаписан. Многократный вызов функции **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 Guardant API предназначена функция **GrdFind**. Воспользуемся ей для построения списка доступных ключей. В качестве первого параметра функции указывается хэндл контейнера Guardant, второй параметр устанавливает тип поиска (*GrdF_First* – поиск первого ключа, *GrdF_Next* – следующие ключи).

nRet:= GrdFind(hGrd, GrdF_First, @dwID, @GrdFindInfo);

Результатом работы функции будет информация, помещенная по указателям в третьем параметре - ID ключа и в четвертом - подробная информация о ключе. Для наглядной демонстрации просмотра подробной информации о ключе сохраним структуру **GrdFindInfo** после первого вызова функции поиска.

Для наглядной демонстрации всех подключенных к компьютеру ключей добавим к программе вывод ID найденного ключа через функцию «ShowMessage».

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 нужный ключ, который действительно нужен, и 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 (**GrdFindInfo)**, которую мы сохранили после первого поиска. Выведем ее на экран и посмотрим, какие же данные записаны в нашем ключе. Для этого разберем структуру на составляющие и покажем ее через функцию ShowMessage.

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 функцией **GrdSetFindMode.** Добавим в наше приложение следующий код

```
dwRemoteMode:= GrdFMR_Local;          { Поиск только локальных ключей }
dwFlags:= GrdFM_NProg + GrdFM_Mask        { Ищем по номеру программы и
                                          битовой маске }
dwID:=  0;                            { ID ключа }
byProg:= 5;                           { Номер нашей программы }
wSN:=   0;                            { Серийный номер }
byVer:= 0;                            { Версия программы }
wMask:= 77;                           { Битовая маска }
dwModel:= GrdFMM_ALL;                 { Все модели ключей }
dwInterface:= GrdFMI_ALL;             { Интерфейс подключения любой (LPT или
                                      USB)}
nRet := GrdSetFindMode(hGrd, dwRemoteMode, dwFlags, byProg, dwID, wSN,
byVer, wMask, wType, dwModel, dwInterface );
```

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 **dwFlags.**

Теперь при вызове функций GrdFind или GrdLogin будут найдены только ключи, соответствующие заданным критериям поиска.

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 функции поиска ключей **GrdFind** в приложении необязательно. Вполне достаточно ограничить круг возможных ключей через функцию **GrdSetFindMode,** и после этого сразу же можно производить подключение к ключу с помощью функции **GrdLogin**.

Функция подключения к электронному ключу **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.

```
nRet := GrdLogin(hGrd,  0xFFFFFFFF,   GrdLM_PerStation );
```

Наравне с функцией подключения к ключу, есть функция отключения от ключа. Она особенно актуальна при использовании сетевых ключей. При ее вызове происходит освобождение лицензии, занятой копией нашего приложения в сетевом ключе при вызове **GrdLogin**. Добавим функцию **GrdLogout** в блок завершения работы нашей программы. Второй параметр данной функции в настоящее время не используется и должен быть равен 0.

```
nRet := GrdLogout(hGrd, 0);
```

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 чтения 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 может быть 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 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 00. Именно по этому номеру и происходит обращение. Наша первая защищенная ячейка имеет номер 0.

Эксперименты с защитой будем ставить на функции поиска по номеру телефона. Первая функция, которую мы будет использовать это **GrdPI_Read** – чтение защищенной ячейки ключа Guardant.

```
nRet:= GrdPI_Read(hGrd,          { Хэндл контейнера Guardant }
        0,                       { Номер защищенной ячейки }
        0 ,                      { Смещение с которого будет производиться
                                   чтение}
        32,                      { Количество байтов для чтения }
        @ReadData[0],            { Указатель на буфер считанных данных  }
        1469608341,              { Пароль на чтение защищенной ячейки }
        nil );                   {Зарезервировано }
```

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 (nRet = GrdE_InactiveItem) then
```

Теперь, если ячейка не активирована, мы должны активировать ее и еще раз вызвать функцию чтения. Для активации ячейки в Guardant API предназначена функция **GrdPI_Activate**. Данная функция 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 функцию активации защищенной ячейки.

```
nRet:= GrdPI_Activate(hGrd,      { Хэндл контейнера Guardant}
          00,                    { 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 функции **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 ClientName.

```
nRet:= GrdPI_Update(hGrd,        { Хэндл контейнера }
            00,                  { Номер защищенной ячейки или алгоритма }
            4,                   { Смещение защищенной ячейки }
            28,                  { Количество байтов для записи }
            @ClientName[1],      { Указатель на буфер с данными }
            843783688,           { Пароль на запись }
```

```
                    GrdUM_MOV,      { Метод обновления (MOV или XOR) }
                    nil );          { Зарезервировано }
```

Поля функции записи в защищенную ячейку, по сути, аналогичны функции 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 функциями чтения и записи защищенных ячеек. Не описанной осталась лишь логика защитных механизмов данного примера. Она приведена в виде блок-схемы (Рис. 1).

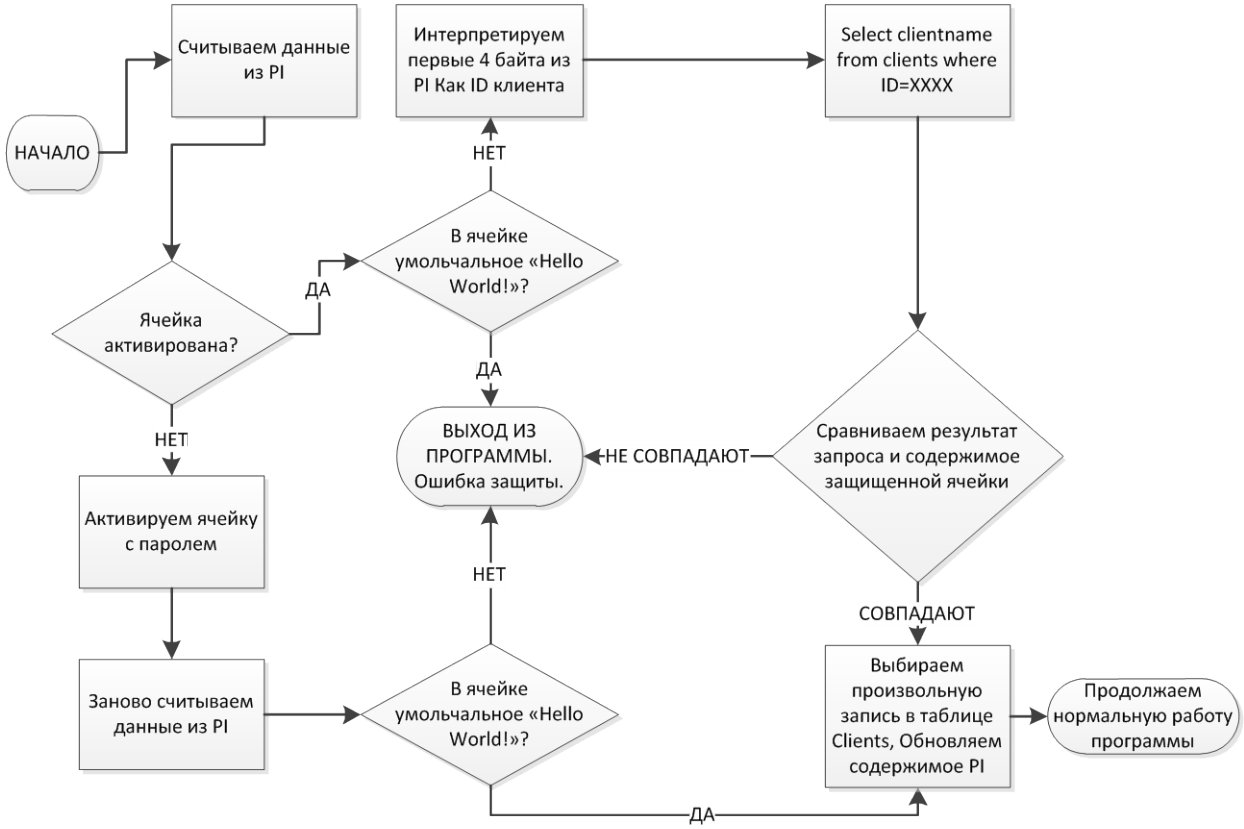

Рис. 1. Алгоритм работы с защищенной ячейкой (Protect Item – PI)

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 функциями чтения и записи защищенных ячеек. Однако в API Guardant есть еще 2 функции чтения и записи **GrdRead** и **GrdWrite**. Они в основном используются для старых моделей ключей. Данные функции 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 функции **GrdRead**. Если взглянуть на маску текущего ключа, видно, что защищенная ячейка «Урок 1» начинается в памяти с адреса 024. Добавим в функцию поиска по имени следующий код:

```
nRet:= GrdRead(hGrd,        { Хэндл контейнера}
       024,                 { Адрес памяти в ключе для начала считывания}
       52,                  { Размер блока данных для считывания (заголовок +
                              содержимое ячейки) }
       @ReadData[0],        { Указатель на переменную, куда будет помещен
                              считанный блок}
       nil);                { Зарезервировано}
```

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

Но функция **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 функцию **GrdSetWorkMode.** В ключе 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.

```
nRet:= GrdSetWorkMode (hGrd,        {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. С помощью таких же констант можно считать из ключа номер и версию программы, битовую маску и другие поля, рассмотренные нами в Части 2.

```
nRet:= GrdRead(hGrd,          {Хэндл контейнера}
      GrdSAM_dwID,            {Адрес памяти в ключе для начала
                               считывания}
      32,                     {Размер блока данных для считывания}
      @ReadData[0],           {Указатель на переменную, куда будет
                               помещен считанный блок}
      nil);                   {Зарезервировано}
```

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 функции закрытия сеанса работы ключом защиты.

```
nRet := GrdLogout(hGrd, 0);
```

Данная функция особенно актуальна 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».

```
nRet:= GrdCloseHandle( hGrd );
```

Последний шаг – деинициализация копии Guardant API.

```
nRet:= GrdCleanup();
```

Только после выполнения всех этих функций 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( hGrd <> nil )     Then        { Проверяем существование хэндла
контейнера }
      begin
            // Завершаем сеанс работы с ключом
```

```
nRet := GrdLogout(hGrd, 0);
        // Закрываем хэндл, освобождаем память
nRet:= GrdCloseHandle( hGrd );
end;
        // Деинициализация копии Guardant API
nRet:= 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.