Guardant®

Система защиты от компьютерного пиратства

Руководство пользователя

Устройство электронных ключей
Guardant Sign/Time/Code

Издание 6.1

Содержание

Глава 8
Технологии Guardant

Базовый элемент – микроконтроллер

Современные электронные ключи Guardant (Guardant Sign / Time / Code) базируются на высокопроизводительном 32-разрядном микроконтроллере. По сравнению с предшественниками, новые ключи получили гораздо больше вычислительных ресурсов и обеспечивают высокую производительность и скорость обмена данными с компьютером по шине USB.

Этот факт открывает множество новых возможностей защиты, как, например, вычисление хеш-функций, шифрование данных симметричными алгоритмами, электронная цифровая подпись на основе эллиптических кривых.

Благодаря наличию микроконтроллера электронный ключ Guardant представляет собой интеллектуальное устройство, способное обрабатывать данные по сложным алгоритмам. На этапе изготовления ключа в память его микроконтроллера «прошивается» специальная микропрограмма, реализующая следующие функции:

- Доступ к функциям ключа по трем 32-битным паролям - ***кодам доступа***
- Интерфейсные функции, организация защищенного протокола обмена с драйвером
- Вычисление функций преобразования данных (симметричное кодирование, хэш-функции, асимметричное шифрование и ЭЦП) при помощи аппаратных алгоритмов
- Защита содержимого памяти: кодирование, аппаратные запреты

Для электронных ключей используются микроконтроллеры, защищенные от считывания и модификации микропрограммы. Таким образом, микроконтроллер представляет собой «черный ящик», скрывающий все происходящие в нем процессы.

Протокол обмена

Протокол обмена драйвера с ключом традиционно является одним из уязвимых звеньев программно-аппаратной защиты. Его изучение – один из основных этапов создания эмуляторов электронных ключей.

Разработчики электронных ключей тратят много сил на улучшение стойкости протоколов обмена. Протокол обмена с современными ключами Guardant имеет ряд оригинальных свойств, повышающих его стойкость.

Свойства протокола обмена с ключом

Шифрование трафика. Все данные, курсирующие между Guardant API и электронным ключом Guardant, непрерывно шифруются при помощи алгоритма AES-128. Это значительно осложняет задачу анализа протокола обмена.

Использование сеансовых ключей. Для противодействия атакам типа «man-in-the-middle» (MITM) на протокол обмена между электронным ключом и Guardant API используется принцип сеансовых ключей. Для каждого сеанса работы с электронным ключом вырабатывается уникальный ключ, на котором и производится шифрование трафика. В течение сеанса периодически вырабатывается новый сеансовый ключ. Таким образом, даже при обмене одинаковыми данными трафик не повторяется и задача построения табличного эмулятора не может быть решена.

Взаимная аутентификация. Кроме собственно шифрования трафика, протокол обеспечивает взаимную аутентификацию электронного ключа и Guardant API, основанную на использовании асимметричного криптографического алгоритма на базе эллиптических кривых ECC160, для защиты от атак типа MITM.

Защищенные ячейки

Защищенная ячейка представляет собой специальную структуру в памяти электронного ключа, предназначенную для безопасного хранения данных. Обращение к защищенным ячейкам осуществляется не по адресу в памяти ключа, а по логическому номеру. Следствием этого является более простая работа с памятью ключа, поскольку отсутствует необходимость вычислять адреса данных, к которым производится обращение. Этим самым облегчается модификация системы защиты.

Для обращения к защищенным ячейкам можно задавать пароли: отдельно для чтения данных и отдельно – для записи. Одновременно с этим возможна установка запретов на чтение или запись данных в ячейки, что дает возможность организовать ячейки только для чтения или только для записи.

Очень важной является возможность активации и деактивации защищенных ячеек по паролю. Она позволяет закладывать в структуру данных возможность безопасного оперативного изменения, например, для перехода на новую версию или для активации дополнительных функций приложения.

Защищенные ячейки используются для хранения дескрипторов аппаратных алгоритмов.

Детальную информацию об устройстве и использовании защищенных ячеек см. в **Главе 11. Защищенные ячейки**.

Аппаратные алгоритмы

Аппаратными алгоритмами принято называть алгоритмы, реализующие преобразования данных (кодирование и декодирование, вычисление хэш-функции и т.п.) внутри микроконтроллера ключа. При такой реализации алгоритмов ни сам метод преобразования, ни ключи шифрования не покидают памяти микроконтроллера, что осложняет анализ этих алгоритмов извне. По сути, такая реализация алгоритмов делает электронный ключ аппаратным «черным ящиком», анализ которого возможен только по входящим и исходящим данным.

Важная информация

Реализация возможности обработки данных вне процессора и памяти компьютера аппаратными алгоритмами является основной защитной функцией электронных ключей Guardant.

Детальную информацию об устройстве и использовании аппаратных алгоритмов см. в **Главе 12. Аппаратные алгоритмы**.

Безопасность памяти ключа

Все данные, которые хранятся внутри ключа (Flash, RAM, EEPROM) защищены от чтения несанкционированного изменения аппаратно архитектурой самого микроконтроллером. Получить доступ к этой памяти без уничтожения ее содержимого невозможно.

Аппаратные запреты

Часть энергонезависимой памяти ключей Guardant недоступна ни для чтения, ни для записи, часть доступна только для чтения. Остальная память доступна полностью и для чтения, и для записи.

Естественно, хакеры проявляют повышенный интерес к содержимому памяти любых ключей. Ведь считав информацию из ключа, можно создать его эмулятор или полную аппаратную копию. EEPROM-память расположена внутри микроконтроллера, а значит защищена так же хорошо, как и сама микропрограмма электронного ключа. С помощью Guardant API, Guardant Code API или GrdUtil к ней можно получать доступ.

Энергонезависимая память, используемая в ключах Guardant, позволяет устанавливать аппаратные запреты чтения/записи ее содержимого. Скопировать содержимое области памяти, на которую наложен запрет, программным способом невозможно – для этого попросту нет никаких средств. Ключ просто не отвечает на запрос на чтение/запись защищенных областей его памяти.

Установка аппаратных запретов производится на нижнем, аппаратном уровне – это гарантирует невозможность их обхода программными средствами.

Аппаратные запреты можно устанавливать на любую область доступной памяти ключей, снимать запреты, расширять или сужать границы защищенной памяти. На дескрипторы создаваемых в ключах Guardant аппаратных алгоритмов и защищенных ячеек аппаратные запреты чтения и записи утилитой программирования устанавливаются по умолчанию – в этом заключается гарантия того, что аппаратные алгоритмы ключа Guardant не могут быть считаны или продублированы. При программировании ключей утилитами собственной разработки необходимо заботиться о правильной установке аппаратных запретов.

Аппаратная блокировка терминальных сеансов

Наличие механизма шифрования трафика между электронным ключом и Guardant API дает возможность на аппаратном уровне ограничить использование локального ключа одновременно из нескольких терминальных сеансов.

В обычном режиме для каждого сеанса взаимодействия с ключом, ограниченного функциями GrdLogin() и GrdLogout(), вырабатывается отдельный сессионный ключ шифрования трафика. Количество сессионных ключей определяется количеством открытых сеансов. Соответственно, электронный ключ может одновременно

использоваться большим количеством одновременно работающих копий защищенного приложения, в том числе из терминальных сеансов. Такая ситуация нарушает принцип «один ключ – одна лицензия», поэтому возникает естественное желание не разрешать запуск более одной копии приложения с одним ключом. Как правило, эта задача решается программными методами, такими как создание глобальных объектов в памяти при запуске приложения и проверки их существования при запуске дополнительных копий. Также существуют методы отслеживания лишних запущенных копий, использующие запись случайных чисел в память ключа и проверку их наличия в процессе работы.

Технология, реализованная в ключах Guardant, позволяет обеспечивать работоспособность только одной копии запущенного приложения. Для этого электронный ключ при предпродажном программировании должен быть переведен в один из режимов ограничения количества сеансовых ключей:

- Один сессионный ключ для автозащиты
- Один сессионный ключ для API
- Два сессионных ключа (по одному для автозащиты и API)

Схема работы в режимах ограничения следующая:

При запуске 1-й копии приложения и выполнении функции **GrdLogin()** генерируется один или два сессионных ключа, которые хранятся в оперативной памяти электронного ключа на протяжении сеанса.

Если происходит запуск 2-й копии приложения, использующего тот же электронный ключ, при выполнении функции **GrdLogin()** происходит выработка нового сессионного ключа. При этом новый ключ заместит собой тот, что был сгенерирован при запуске 1-й копии. Первая копия приложения не сможет обмениваться данными с электронным ключом, поскольку будет использовать устаревший сессионный ключ, и станет неработоспособной.

Защищенное удаленное программирование по технологии Trusted Remote Update

При удаленном программировании электронных ключей, находящихся у конечных пользователей, неизбежно возникают проблемы, связанные с защищенным обменом данными.

При удаленном программировании разработчик должен убедиться, во-первых, в том, что конечный пользователь собирается обновить данные именно в том ключе, в котором нужно. Для этой цели в закодированное число-вопрос, генерируемое ключом в начале сессии удаленного программирования, обязательно включается ID ключа, на котором оно сгенерировано.

Во-вторых, разработчику необходимо удостовериться, что число-вопрос является подлинным, то есть конечный пользователь не прислал поддельное или измененное число-вопрос. Для контроля подлинности используется аппаратно реализованная хэш-функция с секретным ключом: HASH64 (для Guardant Sign/Time/Net) или SHA256 (Guardant Code /Code Time). Число-вопрос передается разработчику вместе с результатом вычисления хэш-функции. Имея электронный ключ с тем же алгоритмом, разработчик может убедиться в подлинности числа-вопроса, вычислив хэш-функцию и сравнив ее результат с присланным. Таким образом, реализуется своего рода цифровая электронная подпись данных.

В-третьих, разработчик должен быть уверенным в том, что число-ответ будет передано в электронный ключ конечного пользователя в неизменном виде. Для этого также используется аппаратное вычисление HASH64 (SHA256).

В-четвертых, конечному пользователю вовсе не обязательно знать, какие именно данные передаются. На месте конечного пользователя вполне может оказаться хакер, пытающийся проанализировать протокол удаленного программирования. По этой причине все данные, передаваемые от конечного пользователя разработчику и обратно, в обязательном порядке кодируются симметричным алгоритмом: GSII64 (для Guardant Sign/Time/Net) или AES128 (Guardant Code /Code Time).

Безопасность протокола удаленного программирования Trusted Remote Update обеспечивается именно аппаратной реализацией алгоритмов кодирования и хеширования, а также тем, что секретные ключи этих преобразований в процессе работы не извлекаются из электронного ключа. Все операции, связанные с декодированием и проверкой целостности данных, осуществляются внутри аппаратного устройства. Это исключает возможность компрометации или подмены данных, которые записываются в электронный ключ.

Важная информация

Для того чтобы механизм Trusted Remote Update работал, в ключ конечного пользователя на этапе предпродажной подготовки прошиваются секретные пароли (секретные ключи GSII64 (AES128)). Их копии вместе с ID электронных ключей, в которые они были прошиты, обязательно должны сохраняться разработчиком в базе данных, доступ к которой должен иметь только авторизованный персонал.

Псевдокод

Наиболее серьезная проблема при защите программного обеспечения – это противодействие различным средствам статического и динамического анализа кода. Цель разработчика защиты состоит в том, чтобы как можно более затруднить работу по реверсингу.

На сегодняшний день технология псевдокода, работающего на виртуальной машине, является наиболее актуальной и эффективной.

Суть технологии состоит в том, что определенные фрагменты исполняемых файлов дизассемблируются, анализируются и преобразуются в защищенный код некоторой уникальной защищенной виртуальной машины. Сама виртуальная машина генерируется тут же. Анализировать логику работы защищенного подобным образом кода существенно сложнее, чем стандартные инструкции Intel-совместимых процессоров, поскольку для него не существует никакого стандартного инструментария (отладчиков, дизассемблеров). Поэтому взломщику приходится все делать вручную, что занимает на несравнимо больше времени, чем использование готовых инструментов.

В каждой копии виртуальной машины уникальным образом реализуются:

- Набор внутренних команд псевдокода
- Множественный взаимный контроль целостности (для затруднения внесения изменений и установки точек останова)
- Обфускация кода виртуальной машины (замусоривание реального кода вторичным)
- Преобразование кода самой виртуальной машины
- Преобразование самого псевдокода
- Параметры многих команд рассчитываются только во время выполнения (защита от статического дизассемблирования)
- Отсутствие постоянных сигнатур в защищенном псевдокодом коде (чтобы затруднить поиск такого псевдокода в защищенном приложении)

При помощи технологии псевдокода защищен код драйверов и Guardant API.

Глава 9
Организация EEPROM-памяти ключей Guardant

Важная информация

Обратите особое внимание на отличия методов адресации памяти и на то, какие области памяти доступны для использования.

EEPROM современных ключей Guardant имеет объем 4096 байт. Часть этой памяти недоступна ни для чтения, ни для записи, часть доступна только для чтения, часть может изменяться только специальными операциями. Остальная память доступна для чтения и записи. При необходимости можно создавать области памяти, защищенные от чтения и (или) записи.

EEPROM может хранить объекты данных двух основных типов:

- ***Предопределенные объекты данных***, существующие в памяти электронного ключа всегда. Их назначение и расположение в памяти определено разработчиками электронных ключей.
- ***Объекты данных, создаваемые разработчиком приложения***. Их назначение, размерность и расположение в памяти определяются при создании. Такие объекты данных могут создаваться только в области памяти свободного назначения

Предопределенные объекты можно разделить на два вида:

Системные поля	Используются микропрограммой ключа, а также в диагностических целях и для идентификации ключей. Адресуются в режиме SAM
Поля общего назначения	Используются, в основном, для выбора нужного ключа, удовлетворяющего заданным критериям поиска. Адресуются в режимах SAM и UAM

Объекты разработчика также могут быть трех видов:

Поля памяти, не защищенные аппаратными запретами	В полях памяти хранится различная информация, которую условно можно отнести к различным типам данных, подобно переменным в языках программирования. Чтение и запись в поля памяти производится функциями **GrdRead** и **GrdWrite** с использованием соответствующих кодов доступа. Никаких дополнительных сервисов не предусмотрено. Адресуются в режимах SAM и UAM

Защищенные ячейки и дескрипторы алгоритмов	В защищенных ячейках могут храниться как обычные данные, так и дескрипторы аппаратных алгоритмов. Для доступа к данным, хранящимся в защищенных ячейках, могут быть назначены специальные пароли, индивидуальные для каждой ячейки. Предусмотрены специальные сервисы активации, деактивации и обновления данных. Ячейки адресуются номерами в соответствии с информацией, которая хранится в таблице размещения защищенных ячеек
Загружаемый код (только для Guardant Code/ Code Time)	Разработчик приложения имеет возможность поместить в электронный ключ собственный код, который должен нести функциональную нагрузку и производить вычисления над данными приложения

Методы адресации памяти

Обращаться к EEPROM-памяти можно в двух режимах:

- ***System Address Mode*** (SAM, системный режим адресации). В режиме SAM доступны 3968 байт информации (но на некоторые области памяти установлены запреты на чтение и (или) запись). Режим SAM рекомендуется использовать только для чтения полей, недоступных в режиме UAM, например, ID ключа. В остальных случаях лучше использовать режим UAM.
- ***User Address Mode*** (UAM, пользовательский режим адресации) В этом режиме доступна память, начиная с адреса 30 SAM. Т. е., например, ячейка памяти с адресом 33 в SAM имеет адрес 3 в UAM, а ячейка памяти с адресом 20 SAM недоступна в UAM.

Два режима адресации нужны для того, чтобы в будущем, при совершенствовании ключей, если будут добавлены новые поля перед полем с адресом 30, не нужно было изменять свою программу, т. к. в режиме UAM адреса ячеек памяти пользователя не изменятся. Электронный ключ умеет сообщать начальный адрес для режима UAM. Это создает возможность сохранить совместимость версий ключей снизу вверх и сверху вниз.

Карта EEPROM

EEPROM-память можно логически разбить на следующие области (все адреса даны в системе адресации SAM):

- **Байты с 0 по 9 включительно** недоступны для записи. Эта память используется ключом для собственных нужд. Содержимое этой памяти программируют на стадии производства ключей; она может использоваться только для чтения. При попытке записи в нее функцией API возвращается ошибка **GrdE_OK**. Эта память недоступна в режиме UAM.

- **Байты с 14 по 25 включительно** доступны только для чтения. Там содержатся Общий код, аппаратная версия ключа, сетевой ресурс ключа, биты типов и моделей ключа и ID. Поля этой памяти полезны, когда надо определить, например, ID ключа. Данные в эту память заносятся на стадии производства ключей. При попытке записи в нее возвращается **GrdE_OK**. Эта память недоступна в режиме UAM.
- **Байты с 10 по 13 и с 26 по 29 включительно** доступны только для операций **GrdProtect** и **GrdInit** или **GrdTRU_SetKey**. Содержат адреса запретов на чтение/запись, количество аппаратных алгоритмов и защищенных ячеек в ключе. При попытке записи в эту память не через **GrdProtect** возвращается **GrdE_OK**. Эта память недоступна в режиме UAM.
- **Байты с 30 по 43 включительно** всегда доступны для чтения и записи и содержат т. н. поля общего назначения: номер программы, версию, серийный номер, маску, счетчик запусков GP (счетчик #1), максимальный ресурс сетевых лицензий (счетчик #2), индекс для дистанционного программирования ключа. Эти поля доступны в режиме UAM. Эти поля в основном используются для формирования параметров поиска нужного ключа и активно используются утилитами автоматической защиты.
- **Байты с 44 по 3959 включительно** доступны для чтения и записи, если на них не установлен запрет, и могут использоваться для хранения данных пользователя, дескрипторов аппаратных алгоритмов и защищенных ячеек. Эти байты обнуляются при выполнении функции **GrdInit** или **GrdTRU_SetKey**.
- **Байты с 3960 по 3967 включительно** зарезервированы, поэтому их не рекомендуется использовать. Доступны в режиме UAM.

Описание полей памяти ключа Guardant:

Адрес (SAM)	Адрес (UAM)	Длина в байтах	Запрет: R-read W-write	Имя поля	Описание поля
Системные поля					
0	N/A	1	W	kmModel	Код модели ключа
1	N/A	1	W	kmMemSize	Код объема памяти. Значение поля соответствует степени, в которую нужно возвести число 2, чтобы получить объем памяти ключа в байтах
2	N/A	1	W	kmProgVer	Версия программы в ключе. Это поле может быть использовано для идентификации ключей с заказными микропрограммами

Адрес (SAM)	Адрес (UAM)	Длина в байтах	Запрет: R-read W-write	Имя поля	Описание поля
3	N/A	1	W	kmProtocol	Версия протокола обмена с ключом. Для внутреннего применения. Используется утилитами диагностики
4	N/A	2	W	kmClientVer	Версия ядра нижнего уровня. Значение 0x104 соответствует версии 1.4. Для внутреннего применения. Используется утилитами диагностики
6	N/A	1	W	kmUserAddr	SAM-адрес начала пользовательской (UAM) памяти в двухбайтовых словах
7	N/A	1	W	kmAlgoAddr	Адрес таблицы размещения защищенных ячеек в словах
8	N/A	2	W	kmPrnPort	Адрес параллельного порта, к которому подсоединен ключ. Для USB-ключей значение равно 0
10	N/A	2	W	kmWriteProtectS3	Адрес SAM первого байта, доступного для записи. Число 0 означает отсутствие запретов на запись. Запрет может быть установлен только с адреса 44 SAM **(*)**. Для Guardant Stealth/Net III и старше
12	N/A	2		kmReadProtectS3	Адрес SAM первого байта, доступного для чтения. Число 0 означает отсутствие запретов на чтение. Запрет может быть установлен только с адреса 44 SAM. **(*)** Для Guardant Stealth/Net III и старше
14	N/A	4	W	kmPublicCode	Общий код в числовом виде.
18	N/A	1	W	kmVersion	Аппаратная версия ключа: биты 0-3: minor биты 4-7: major
19	N/A	1	W	kmLANRes	Максимальный LAN ресурс сетевого ключа, программируемый при продаже ключа. Для локальных ключей значение равно 0
20	N/A	2	W	kmType	Биты типа ключа. Поле может участвовать в условии поиска ключа.
22	N/A	4	W	kmID	ID (уникальный идентификационный номер) ключа. Может участвовать в условии поиска ключа. Записывается на стадии производства. Гарантируется, что двух ключей с одинаковыми ID не может быть. Можно настроить приложение на идентификатор ключа, и тогда оно будет работать только с ключом, содержащим этот ID.
26	N/A	1	W	kmWriteProtect	Не используется. Должно быть равно FFh (см. kmWriteProtectS3) **(*)**

Адрес (SAM)	Адрес (UAM)	Длина в байтах	Запрет: R-read W-write	Имя поля	Описание поля
27	N/A	1	W	kmReadProtect	Не используется. Должно быть равно FFh (см. kmReadProtectS3) **(*)**
28	N/A	1	W	kmAlgoNum	Количество защищенных ячеек и аппаратных алгоритмов **(*)**
29	N/A	1	W	kmTableLMS	№ защищенной ячейки, в которой хранится таблица лицензий. Используется только для сетевых ключей
Поля общего назначения					
30	0	1		kmNProg	Номер программы. Может участвовать в условии поиска ключа.
31	1	1		kmVer	Версия. Может участвовать в условии поиска ключа.
32	2	2		kmSN	Серийный номер. Может участвовать в условии поиска ключа.
34	4	2		kmMask	Битовая маска. Может участвовать в условии поиска ключа.
36	6	2		kmGP	Счетчик запусков программы GP (счетчик #1). Не может участвовать в условии поиска ключа. Не рекомендуется использовать
38	8	2		kmRealLANRes	Реальный сетевой ресурс ключа Guardant Net (счетчик #2). Не может участвовать в условии поиска ключа.
40	10	4		kmIndex	Индекс для удаленного программирования. Не может участвовать в условии поиска ключа. Не рекомендуется использовать
Поля свободного назначения					
44	14	3916		kmUserData	Начиная с этого адреса, могут располагаться данные, в том числе дескрипторы аппаратных алгоритмов
Поля специального назначения					
3960	3930	8			Это поле используется программами диагностики электронного ключа. СТРОГО не рекомендуется использовать. Не обнуляется операцией GrdInit и GrdTRU_SetKey. Доступны в режиме UAM

Важная информация

(*) Только функции GrdInit() и GrdTRU_SetKey() могут обнулить значение этого поля. Запись в поле возможна только после обнуления и только функцией GrdProtect(). При попытке записи в эту память не через GrdProtect() возвращается GrdE_OK, но запись не производится. Эта память недоступна в режиме UAM.

Использование EEPROM

Поля только для чтения

Эти поля можно использовать при помощи функций API. Некоторые поля используются также и автозащитой. Эта область всегда защищена от записи, поэтому изменить их содержимое нельзя.

Поле **kmModel**. Содержит код модели ключа. В настоящее время моделям ключей присвоены следующие коды (см. файл ***GrdAPI.H***):

Имя константы	Значение	Модель ключа
GrdDM_GS1L	0	Guardant Stealth/Net LPT
GrdDM_GS1U	1	Guardant Stealth/Net USB
GrdDM_GF1L	2	Guardant Fidus LPT
GrdDM_GS2L	3	Guardant Stealth II /Net II LPT
GrdDM_GS2U	4	Guardant Stealth II /Net II USB
GrdDM_GS3U	5	Guardant Stealth III / Net III
GrdDM_GS3SU	7	Guardant Sign/Time, Sign Net /Time Net
GrdDM_GCU	8	Guardant Code

Поле **kmMemSize** содержит информацию об объеме памяти ключа. Значение поля соответствует степени, в которую нужно возвести число 2, чтобы получить объем памяти ключа в байтах.

Поле **kmProgVer** содержит версию программы микроконтроллера ключа. Это поле может быть использовано для идентификации ключей с заказными микропрограммами.

Поле **kmProtocol** содержит версию протокола обмена с ключом. Для внутреннего применения. Используется утилитами диагностики.

Поле **kmClientVer** содержит версию ядра нижнего уровня. Значение 0x104 соответствует версии 1.4. Для внутреннего применения. Используется утилитами диагностики.

Поле **kmUserAddr** содержит SAM-адрес начала пользовательской (UAM) памяти в двухбайтовых словах.

Поле **kmAlgoAddr** содержит начальный адрес таблицы размещения аппаратных алгоритмов в двухбайтовых словах.

Поле **kmPrnPort** содержит адрес параллельного порта, к которому подсоединен ключ. Для USB-ключей Guardant значение этого поля равно 0.

Поле **kmWriteProtectS3**. Содержит адрес (в режиме SAM) первого байта, который доступен для записи. Число 0 означает отсутствие запретов на запись. Запрет может быть установлен только с адреса 44 SAM. Запись в это поле возможна только операцией Protect и только если содержимое поля равно 0. Обнулить содержимое поля можно только операцией **GrdInit** или **GrdTRU_SetKey**.

Поле **kmReadProtectS3**. Содержит адрес (в режиме SAM) первого байта, который доступен для чтения. Число 0 означает отсутствие запретов на чтение. Запрет может быть установлен только с адреса 44 SAM. Запись в это поле возможна только операцией Protect и только если содержимое поля равно 0. Обнулить содержимое поля можно только операцией **GrdInit** или **GrdTRU_SetKey**.

Поле **kmPublicCode**. Содержит Общий код ключа в числовом виде (4 байта), однозначно идентифицирующий владельца данного электронного ключа.

Поле **kmVersion**. Содержит аппаратную версию данного ключа. Т. к. ключи Guardant постоянно совершенствуются, приложение может проверять аппаратную версию для выяснения возможностей ключа, с которым оно сейчас работает.

Поле **kmLANRes**. Содержит максимальный сетевой ресурс ключа Guardant Net. Максимальный сетевой ресурс ключа программируется на стадии производства. В локальных ключах Guardant любого поколения значение этого поля равно нулю.

Поле **kmType**. Индикатор типа ключа. Позволяет программному обеспечению определять конкретный тип ключа. Можно использовать это поле при проверках наличия ключа.

Таким образом, существуют следующие флаги типов ключей (см. файл ***GrdAPI.H***):

Имя константы	Значение	Комментарий. Тип ключа
GrdDT_DOS	0x0000	Ключ поддерживает защиту приложений DOS
GrdDT_Win	0x0000	Ключ поддерживает защиту приложений Windows
GrdDT_LAN	0x0001	Ключ поддерживает защиту приложений, созданных для работы в локальных сетях
GrdDT_Time	0x0002	Ключ имеет возможность ограничивать время работы защищенного приложения без использования RTC
GrdDT_GSII64	0x0008	В ключе есть алгоритм GSII64: Guardant Stealth II / Net II, Stealth III/ Net III, Sign /Time, Sign Net /Time Net. В Guardant Code этот флаг отсутствует
GrdDT_PI	0x0010	Ключ поддерживает защищенные ячейки: Guardant Stealth III/ Net III, Sign, Time, Code
GrdDT_TRU	0x0020	Ключ поддерживает защищенное удаленное обновление: Guardant Stealth III/ Net III, Sign, Time, Code
GrdDT_RTC	0x0040	Ключ содержит таймер реального времени: Guardant Time / Time Net, Guardant Code Time
GrdDT_AES	0x0080	Ключ содержит аппаратный алгоритм AES: Guardant Sign / Time / Net / Code
GrdDT_ECC	0x0100	Ключ содержит аппаратную реализацию алгоритма ECC160: Guardant Sign / Time / Net / Code
GrdDT_LoadableCode	0x0400	Ключ поддерживает загружаемый код: Guardant Code / Code Time

Поле **kmID**. Содержит идентификационный номер (ID) ключа. ID записывается в это поле на стадии производства. Гарантируется, что двух ключей с одинаковыми ID не может быть. Это поле используется в специальном режиме автоматической защиты. Можно настроить приложение на ID ключа, и тогда оно будет работать только с ключом, содержащим этот ID.

Поле **kmWriteProtect**. В современных ключах поле содержит значение FFh. Запись в это поле возможна только операцией **GrdProtect** и только если содержимое поля равно 0. Обнулить содержимое поля можно только операцией **GrdInit** или **GrdTRU_SetKey**.

Поле **kmReadProtect**. В современных ключах это поле должно содержать значение FFh. Запись в это поле возможна только операцией **GrdProtect** и только если содержимое поля равно 0. Обнулить содержимое поля можно только операцией **GrdInit** или **GrdTRU_SetKey**.

Поле **kmAlgoNum**. Содержит количество аппаратных алгоритмов и защищенных ячеек. Значение в это поле записывается только при выполнении операции **GrdProtect**. Обнулить содержимое поля можно только операцией **GrdInit** или **GrdTRU_SetKey**.

Поле **kmTableLMS**. В локальных ключах Guardant не используется. Значение в это поле записывается только при выполнении операции **GrdProtect**. Обнулить содержимое поля можно только операцией **GrdInit** или **GrdTRU_SetKey**.

Поля общего назначения

Поля общего назначения используются в основном для того, чтобы приложение могло найти ключ с нужными для его работы параметрами. Это особенно актуально в тех случаях, когда разработчик поставляет несколько приложений, защищенных электронными ключами с одинаковым Общим кодом, а также в тех случаях, когда каждая копия приложения персонализирована под конкретного пользователя. Информация, записанная в полях общего назначения, носит скорее справочный характер, чем влияет на защищенность приложения.

Большинство этих полей активно используется автоматической защитой (см. Руководство пользователя, часть 1, глава **Автоматическая защита**). Также возможно изменять содержимое полей и использовать их с помощью функций Guardant API.

kmNProg	**Номер программы** – поле содержит номер защищенного программного продукта. Используется для «привязки» копии приложения к электронному ключу в случае, если программных продуктов несколько. Диапазон допустимых значений от 0 до 255. По умолчанию поле содержит значение 0.
kmVer	**Версия программы** - поле содержит номер версии защищенного программного продукта. Может использоваться для «привязки» копии приложения к электронному ключу в случае выпуска новых версий программы. Диапазон допустимых значений от 0 до 255.
kmSN	**Серийный номер** - поле содержит номер копии защищенного программного продукта. Используется для «привязки» конкретной копии приложения к электронному ключу. Диапазон допустимых значений от 0 до 65535. При записи маски в ключ утилитой GrdUtil.exe значение поля автоматически увеличивается на 1 при каждой записи информации в память ключа.
kmMask	**Битовая маска** - поле полезно в случае, если программный продукт состоит из комплекса независимых приложений. Используется для разрешения/запрета работы с отдельными модулями программного продукта (комплекса). Диапазон допустимых значений от 0 до 65535.
kmGP	**Счетчик запусков программ (GP, счетчик #1).** Диапазон допустимых значений от 0 до 65535. Содержимое этого поля уменьшается на 1 при каждом вызове специальной функции API, либо при каждом запуске приложения, защищенного автоматической защитой в специальном режиме. **В настоящее время использование этого поля не актуально. Оно существует исключительно для обеспечения совместимости. Для ограничения времени необходимо использовать ключи с часами реального времени, для ограничения числа запусков - счетчики в дескрипторах аппаратных алгоритмов**
kmReal LANRes	**Реальный сетевой ресурс ключей Guardant (счетчик #2)** – поле содержит реальное значение сетевого ресурса электронного ключа. Используется для лицензирования – ограничения количества одновременно используемых в ЛВС копий программного продукта. Доступно только для сетевых ключей. Диапазон допустимых значений от 0 до 65535. В ключах Guardant Sign/Time Net и Guardant Net III значение этого поля должно дублироваться в таблице лицензий LMS.
kmIndex	**Индекс** – поле используется утилитами дистанционного программирования ключа. Нельзя использовать **Индекс** для хранения данных!!!

Поля свободного назначения

При активном использовании функций Guardant API и аппаратных алгоритмов, эта область EEPROM является основной. Именно в ней хранятся наиболее критичные для защиты данные.

В области полей свободного назначения хранятся:

- Системные таблицы
- Дескрипторы аппаратных алгоритмов и защищенных ячеек
- Другие данные, необходимые для защиты

При предпродажной подготовке электронных ключей сюда записываются дескрипторы стандартных аппаратных алгоритмов и защищенных ячеек. При помощи утилиты GrdUtil.exe можно создать свои дескрипторы и добавить их к существующим, либо изменить стандартные дескрипторы, также сделав их уникальными (подробнее см. в **Главе 11. Защищенные ячейки**).

Кроме того, здесь могут находиться любые другие данные, которые будут использованы при защите программного обеспечения. Это могут быть: ключевые слова, наборы данных, фрагменты программного кода и т. п. Получить к ним доступ, можно выполнив операцию чтения при помощи функций API.

Область полей свободного назначения должна быть сформирована таким образом: в начало области записываются системные таблицы, все дескрипторы алгоритмов и защищенных ячеек, и лишь потом – все остальные данные. Это необходимо потому, что дескрипторы должны быть защищены от чтения и записи, а запреты устанавливаются на непрерывную область памяти с адреса 44 SAM (поле **kmUserData**).

Пример кода, реализующего запись таблицы размещения, дескрипторов защищенных ячеек и аппаратных алгоритмов находится среди примеров, входящих в комплект разработчика.

Поля специального назначения

Последнюю, самую «дальнюю» область памяти ключа используют утилиты Комплекта разработчика. Поэтому, начиная с адреса 3960 SAM, 8 байт зарезервированы, и **их использование строго не рекомендуется во избежание конфликта!**

Глава 10
Защищенные ячейки

Защищенная ячейка (Protected Item) – это специальный объект, предназначенный для хранения разного рода данных. В качестве содержимого защищенных ячеек могут выступать как обычные данные (числа, строки, дампы), так и секретные ключи (определители) аппаратных алгоритмов. В отличие от обычных полей памяти, защищенные ячейки предоставляют дополнительные сервисы: чтение и обновление данных, активация и деактивация, доступ к данным по паролям.

Дескриптор защищенной ячейки

Защищенная ячейка определяется структурой в памяти электронного ключа, называемой ***дескриптором***. Дескриптор состоит из полей, описывающих тип данных, которые хранятся в защищенной ячейке, ее свойства, состояние, пароли активации/деактивации и на выполнение операций с данными.

Обращение к защищенной ячейке / аппаратному алгоритму производится по ***числовому имени***. Числовое имя – это идентификатор длиной 2 байта, который хранится в специальной таблице числовых имен ячеек и алгоритмов (Algorithm Root Table, ART). Числовое имя позволяет идентифицировать ячейку вне зависимости от того, какую область памяти она занимает, поскольку «физически» ячейки могут располагаться в памяти в произвольном порядке.

Смещение от начала дескриптора	**Длина поля в байтах**	**Обозначение поля**	**Описание поля**
00h	1	rs_LoFlags	Младший байт флагов, см. Определения **nsafl_xxx**
01h	1	rs_algo	Код типа алгоритма (см. определения **rs_algo_XXXX**)
02h	2	ReservedForEven	Зарезервировано для выравнивания
04h	4	rs_HiFlags	Дополнительные флаги, см. определения **nsafh_xxx**
08h	4	rs_klen	Длина данных ячейки или определителя алгоритма в байтах (**rs_K[]**). Нужно учитывать необходимость оставить 16 свободных байт в конце доступной памяти. Для алгоритмов SHA256 равно 0. Для алгоритмов с загружаемым кодом соответствует размеру структуры **TgrdLoadableCodeData**

Смещение от начала дескриптора	Длина поля в байтах	Обозначение поля	Описание поля
0C	4	rs_blen	Размер блока данных для аппаратного алгоритма. Для алгоритма SHA256 равно 0. Для защищенных ячеек и алгоритмов с загружаемым кодом содержимое поля игнорируется
10	8	rs_hash	Зарезервировано. Поле должно быть заполнено нулями
18	4	rs_ActivatePwd	Пароль активации (если установлен флаг nsafl_ActivationSrv)
1C	4	rs_DeactivatePwd	Пароль деактивации (если установлен флаг nsafl_DeactivationSrv)
20	4	rs_ReadPwd	Пароль для чтения полей rs_GP, rs_ErrorCounter, rs_K[] с помощью функции GrdPI_Read (если установлен флаг nsafh_ReadPwd)
24	4	rs_UpdatePwd	Пароль для изменения полей rs_GP, rs_ErrorCounter, rs_K[] с помощью функции GrdPI_Update (если установлен флаг nsafh_UpdateSrv)
28	6	rs_BirthTime	Дата и время автоматической активации ячейки (если установлен флаг nsafh_BirthTime). Дата и время хранятся в структуре TGrdTime. Размер поля равен sizeof(TGrdTime)
2E	6	rs_DeadTime	Дата и время автоматической деактивации ячейки (если установлен флаг nsafh_DeadTime). Дата и время хранятся в структуре TGrdTime. Размер поля равен sizeof(TGrdTime)
34	8	rs_Lifetime	Время, в течение которого ячейка будет оставаться активной после 1-го обращения к ней или вызова алгоритма (если установлен флаг nsafh_Lifetime) Дата и время хранятся в структуре TGrdLifeTime. Размер поля равен sizeof(TGrdLifeTime)
3C	8	rs_FlipTime	Если указан флаг nsafh_FlipTime - алгоритм меняется каждые rs_DaysGap дней, начиная с даты rs_ChangeFlipTimeStart Размер поля равен sizeof(TgrdFlipTime). Для алгоритмов с загружаемым кодом и SHA256 не используется

Смещение от начала дескриптора	Длина поля в байтах	Обозначение поля	Описание поля
44	4	rs_GP	Обратный счетчик. Если установлен флаг nsafl_GP_dec, счетчик уменьшается на 1 при каждом вызове алгоритма через функцию GrdTransform(). При использовании других функций (например, GrdHash() или GrdCrypt() на больших объемах данных) счетчик может уменьшаться быстрее, согласно количеству вызовов GrdTransform() внутри вызывающей функции. По достижении счетчиком нулевого значения, алгоритм переходит в деактивированное состояние и перестает преобразовывать данные. При дальнейших обращениях возвращается код ошибки GrdE_GPis0. Значение счетчика можно увеличить, только записав весь дескриптор заново Если установлен флаг nsafl_GP, алгоритм становится зависимым от значения счетчика. См. описание флага. Если ни один из этих флагов не установлен, значение игнорируется.
48	4	rs_ErrorCounter	Допустимое количество ошибок ввода паролей (если установлен хотя бы один из флагов: nsafl_ActivationSrv, nsafl_DeactivationSrv, nsafl_UpdateSrv). Используется только 1 байт этого поля. По достижении счетчиком нуля защищенная ячейка переходит в состояние с неизменяемым статусом. При вводе правильного пароля значение счетчика не восстанавливается
4C	rs_klen	rs_K[]	Данные защищенной ячейки или определитель алгоритма длиной rs_klen. Для алгоритмов с загружаемым кодом в этом поле хранится структура TgrdLoadableCodeData

Поле **rs_LoFlags** содержит младший байт флагов, определяющих свойства защищенной ячейки. Возможна установка следующих флагов (приведенные ниже названия флагов используются в Guardant API):

Имя флага	Значение	Комментарий
nsafl_ID	1	Уникальность алгоритма по ID ключа. При одинаковых определителях алгоритмы в разных ключах кодируют данные по-разному. Флаг работает только для симметричных алгоритмов (AES128 и GSII64), остальными типами алгоритмов не используется

Имя флага	Значение	Комментарий
nsafl_GP_dec	2	Уменьшать счетчик GP при каждой обращении к алгоритму. По достижении счетчиком GP 0, алгоритм/ ячейка автоматически деактивируется и при дальнейших обращениях возвращается код ошибки GrdE_GPis0. Для алгоритмов SHA256 флаг игнорируется
nsafl_GP	4	Зависимость преобразования данных от значения счетчика GP. В современных ключах не используется
nsafl_ST_III	8	В современных ключах флаг установлен
nsafl_ActivationSrv	16	Сервис активации доступен
nsafl_DeactivationSrv	32	Сервис деактивации доступен
nsafl_UpdateSrv	64	Сервис изменения данных ячеек rs_K[] по паролю доступен (функция GrdPI_Update поддерживается)
nsafl_InactiveFlag	128	Признак, что в данный момент алгоритм/ячейка деактивирован. Операции GrdTransform, GrdPI_Read, GrdPI_Update недоступны.

Возможность применения флагов свойств в алгоритмах и защищенных ячейках:

Имя флага	AES128, GSII64	ECC160	SHA256	Загружаемый код	Ячейка с данными
nsafl_ID	+	-	-	-	-
nsafl_GP_dec	+	+	+	+	-
nsafl_GP	-	-	-	-	-
nsafl_ST_III	+	+	+	+	-
nsafl_ActivationSrv	+	+	+	+	+
nsafl_DeactivationSrv	+	+	+	+	+
nsafl_UpdateSrv	+	+	+	+	+
nsafl_InactiveFlag	+	+	+	+	+
nsafh_ReadSrv	+	+	+	+	+
nsafh_ReadPwd	+	+	+	+	+
nsafh_BirthTime	+	+	+	+	-
nsafh_DeadTime	+	+	+	+	-
nsafh_LifeTime	+	+	+	+	-
nsafh_FlipTime	+	-	-	-	-

Поле **rs_algo** содержит код типа защищенной ячейки.

а) Для Guardant Sign/Time/Net доступны следующие коды типов данных защищенных ячеек:

Имя кода	Значение	Комментарий
	0-4	Зарезервировано
rs_algo_GSII64	5	Симметричный алгоритм кодирования данных. Секретный ключ размером 128 или 256 бит
rs_algo_HASH64	6	Вычисление 64-битного хэша. Секретный ключ размером 128 или 256 бит
rs_algo_RND64	7	Генерация 64-битного случайного числа

Имя кода	Значение	Комментарий
rs_algo_PI	8	Защищенная ячейка данных
rs_algo_GSII64_ENCRYPT	10	Аналогичен rs_algo_GSII64, но возможно только зашифрование
rs_algo_GSII64_DECRYPT	11	Аналогичен rs_algo_GSII64, но возможно только расшифрование
rs_algo_ECC160	12	Цифровая подпись по алгоритму ECC 160 бит. Размер определителя при этом 20 байт
rs_algo_AES128	13	AES-128 (режимы шифрования поддерживаются)
rs_algo_SHA256	15	Алгоритм хэширования SHA256. Определитель не содержит данных в поле rs_K[]

б) Для Guardant Code/Code Time допустимы следующие значения:

Имя кода	Значение	Комментарий
rs_algo_PI	8	Защищенная ячейка, содержащая данные
rs_algo_ECC160	12	Цифровая подпись по алгоритму ECC 160 бит. Размер определителя при этом 20 байт
rs_algo_AES128	13	Симметричное шифрование по AES с длиной ключа 128 бит (режимы шифрования поддерживаются)
rs_algo_LoadableCode	14	Алгоритм, содержащий загружаемый код
rs_algo_SHA256	15	Алгоритм хэширования SHA256. Определитель не содержит данных в поле rs_K[]
rs_algo_AES128Encode	16	Симметричное шифрование по AES с длиной ключа 128 бит. Данные могут только зашифровываться. Возможны режимы шифрования только ECB и CBC
rs_algo_AES128Decode	17	Симметричное шифрование по AES с длиной ключа 128 бит. Данные могут только расшифровываться. Возможны режимы шифрования только ECB и CBC

Поле **rs_HiFlags** содержит 4 байта флагов, определяющих свойства защищенной ячейки. Возможна установка следующих флагов (приведенные названия флагов используются в Guardant API):

Имя флага	Значение	Комментарий
nsafh_ReadSrv	1	Сервис чтения данных ячеек rs_K[] доступен (функция GrdPI_Read поддерживается)
nsafh_ReadPwd	2	Чтение осуществляется по паролю rs_ReadPwd
nsafh_BirthTime	4	Включен режим активации в указанное время (хранится в поле rs_BirthTime)
nsafh_DeadTime	8	Включен режим деактивации в указанное время (хранится в поле rs_DeadTime)
nsafh_LifeTime	16	Включен режим деактивации через указанное время после первого обращения к ячейке (оставшееся до деактивации время хранится в ячейке rs_LifeTime) Одновременное использование с флагами nsafh_DeadTime и nsafh_BirthTime не допускается!

Имя флага	Значение	Комментарий
nsafh_FlipTime	32	Включен режим автоматического изменения определителя каждые rs_DaysGap дней, начиная с даты rs_ChangeFlipTimeStart. Можно комбинировать с флагами nsafh_DeadTime, nsafh_BirthTime и nsafh_LifeTime. Этот режим поддерживается только для симметричных алгоритмов (AES128 и GSII64).

Поле **rs_klen** содержит размер данных **rs_K[]**, хранящихся в защищенной ячейке (секретного ключа алгоритма) в байтах.

Поле **rs_blen** содержит минимальный размер блока данных для аппаратного алгоритма. Допустимые значения:

Тип аппаратного алгоритма	rs_klen - длина секретного ключа, байт	rs_blen - минимальная длина блока данных, байт
GSII64	GrdADS_GSII64=16/32	GrdARS_GSII64=8
HASH64	GrdADS_HASH64=16/32	GrdARS_HASH64=8
RND64	GrdADS_RAND64=16/32	GrdARS_RAND64=8
AES128	GrdADS_AES128=16	GrdARS_AES128=16
ECC160	GrdADS_ECC160=20	GrdARS_ECC160=20
SHA256	-	GrdARS_HASH_SHA256=0
Загружаемый код	sizeof(TGrdLoadableCodeData)	Не используется

Для алгоритмов GSII64, HASH64 и RND64 при ошибочном значении длины ключа, его длина устанавливается равной 16 байт.

Поле **rs_hash** зарезервировано для будущего использования.

Поле **rs_ActivatePwd** содержит 4-байтовый пароль активации защищенной ячейки, в том случае, если разрешен сервис активации установкой флага **nsafl_ActivationSrv**.

Поле **rs_DeactivatePwd** содержит 4-байтовый пароль деактивации защищенной ячейки, в том случае, если разрешен сервис деактивации установкой флага **nsafl_DeactivationSrv**.

Поле **rs_ReadPwd** содержит 4-байтовый пароль чтения данных защищенной ячейки, в том случае, если разрешен сервис чтения данных по паролю установкой флага **nsafh_ReadPwd**.

Поле **rs_UpdatePwd** содержит 4-байтовый пароль на обновление данных защищенной ячейки в том случае, если разрешен сервис обновления данных по паролю установкой флага **nsafh_UpdateSrv**.

Поле **rs_BirthTime** содержит дату и время автоматической активации алгоритма. Если не установлен флаг **nsafh_BirthTime**, значение поля игнорируется. Дата и время хранятся в структуре **TGrdTime**:

Смещение	Длина	Название	Комментарий
0	BYTE	BSeconds	Секунды от 0 до 59
1	BYTE	BMinute	Минуты от 0 до 59

Смещение	Длина	Название	Комментарий
2	BYTE	BHour	Часы от 0 до 23
3	BYTE	BDay	Дни от 1 до 31
4	BYTE	BMonth	Месяцы от 1 до 12
5	BYTE	BYear	Годы от 0 до 99, начиная с 2000 года

Поле **rs_DeadTime** содержит дату и время автоматической деактивации алгоритма. Если не установлен **nsafh_DeadTime**, значение поля игнорируется. Дата и время хранятся в структуре **TgrdTime** (см. выше).

Поле **rs_LifeTime** содержит время, в течение которого алгоритм будет оставаться в активном состоянии после первого обращения к нему. Если не установлен флаг **nsafh_LifeTime**, значение поля игнорируется. Дата и время хранятся в структуре **TGrdLifeTime**:

Смещение	Длина	Название	Комментарий
0	TGrdTime	LifeTime	Срок жизни аппаратного алгоритма после первого обращения (в виде разницы дат)
6	BYTE	State	0 – обращений к алгоритму еще не было, **это исходное значение**. При предпродажном программировании ключа в поле rs_DeadTime рекомендуется помещать текущее время на момент программирования. 1 – алгоритм уже активировался, это значение устанавливается **автоматически** после первого обращения к алгоритму. В поле rs_DeadTime автоматически записывается время, когда алгоритм должен деактивироваться
7	BYTE	ReservedForEven	Зарезервировано для выравнивания

Поле **rs_FlipTime** содержит время **rs_ChangeFlipTimeStart**, когда должна начаться автоматическая модификация определителя алгоритма и количество дней **rs_DaysGap**, составляющих период смены.

Если не установлен флаг **nsafh_FlipTime**, значение поля игнорируется. Дата и время хранятся в структуре **TGrdFlipTime**:

Смещение	Длина	Название	Комментарий
0	TGrdTime	rs_ChangeFlipTimeStart	Дата и время, с которого начинается отсчет
6	BYTE	rs_DaysGap	Алгоритм меняется каждые rs_DaysGap дней, начиная с даты rs_ChangeFlipTimeStart. Должно быть значение от 1 до 255
7	BYTE	ReservedForEven	Зарезервировано для выравнивания

Поле **rs_GP** содержит счетчик алгоритма. Если указан флаг **nsaf_GP_dec**, то это поле задает то количество раз, которое алгоритм может быть выполнен. По достижении счетчиком нулевого значения, такой алгоритм переходит в деактивированное состояние и перестает преобразовывать данные. При дальнейших обращениях возвращается код ошибки **GrdE_InactiveItem**. Значение счетчика можно увеличить, только записав весь дескриптор заново.

Поле **rs_ErrorCounter** содержит счетчик неудачных попыток подбора паролей на доступ к защищенной ячейке. При каждой неудачной попытке счетчик уменьшается на 1. По достижении счетчиком нуля защищенная ячейка переходит в состояние с неизменяемым статусом.

Поле **rs_K[]** содержит данные защищенной ячейки. В зависимости от типа ячейки здесь может храниться секретный ключ алгоритма, либо какие-нибудь иные данные. Длина данных обязательно должна соответствовать значению поля **rs_klen**. Данные защищенной ячейки можно считывать или изменять только с помощью специальных функций. Главное отличие защищенной ячейки от простых данных заключается в том, что ее данные можно менять, только указав специальный пароль. Этот пароль программируется разработчиком и хранится в дескрипторе защищенной ячейки. Такой механизм позволяет производить более безопасное изменение части памяти ключа, не затрагивающее остальную память.

В дескрипторах алгоритмов с загружаемым кодом в поле **rs_K[]** должна содержаться структура **TGrdLoadableCodeData**, описывающая свойства модуля загружаемого кода. Эта структура разделена на публичную часть, данные которой можно прочитать, если сервис чтения разрешен, и приватную часть, данные которой закрыты от чтения.

Длина, байт	Имя поля	Комментарий
Публичная часть (структура TGrdCodePublicData), данные доступны для чтения		
1	bLoadableCodeVersion	Версия загружаемого кода
1	bReserved0	Зарезервировано
1	bState	Состояние загружаемого кода (GrdCodeState_XXXXX)
1	bReserved	Зарезервировано
4	dwLoadingDate	Дата загрузки кода. Заполняется автоматически функцией GrdCodeLoad

Длина, байт	Имя поля	Комментарий
Закрытая часть структуры, данные не доступны для чтения		
40	abLoadableCodePublicKey4VerifySign	Открытый ключ для проверки подписи кода
20	abLoadableCodePrivateKey4DecryptKey	Закрытый ключ для расшифрования ключа AES, на котором зашифрован загружаемый код
4	dwBegFlashAddr	Начальный адрес Flash-памяти, зарезервированной для загружаемого кода
4	dwEndFlashAddr	Конечный адрес Flash-памяти, зарезервированной для загружаемого кода
4	dwBegMemAddr	Начальный адрес RAM ключа, зарезервированной для загружаемого кода
4	dwEndMemAddr	Конечный адрес RAM ключа, зарезервированной для загружаемого кода

Таблица лицензий

Таблица лицензий (License Management System, LMS)– разновидность защищенной ячейки, в которой хранится общий сетевой ресурс ключей Guardant Sign Net/ Time Net, а также сетевые ресурсы многомодульных программных комплексов.

В сетевом ключе может быть только одна таблица лицензий.

Адрес таблицы в памяти ключа определяется по значению, указанному в поле **kmTableLMS (29 SAM)**.

Заголовок таблицы имеет размер 2 байта.

Формат таблицы лицензий современных сетевых ключей

Для Guardant Sign Net / Time Net наличие таблицы лицензий обязательно, т. к. в ней хранится реальный сетевой ресурс ключа. Реальный ресурс ключа лежит в первой по счету записи LMS. За ним следуют записи с ресурсами модулей.

Данные защищенной ячейки типа **таблица лицензий**, размещаются в поле **rs_K[]** (см. описание формата дескриптора защищенной ячейки):

Смещение*	Размер	Значение	Комментарии
0000	2 байта	'LT'	Сигнатура таблицы
0002	2 байта	0x0510	Версия таблицы

* Смещение от начала поля **rs_K[]**

Смещение*	Размер	Значение	Комментарии
0004	4 байта	CRC	Контрольная сумма таблицы**, подсчитанная при помощи GrdCRC от -1
0008	2 байта	LMS_FLAG_LICENCE_SIZE_2BYTES	Флаги таблицы. LMS_FLAG_LICENCE_SIZE_2BYTES установлен, если размер записи – 2 байта, иначе -1
0010	1 байт	Количество записей таблицы***	Общий размер LMS не может превышать 254 байта
0011	5 байтов	Зарезервировано	Не используется
0016	1 или 2 байта	0-255 или 0-65535	Реальный сетевой ресурс ключа
0017 или 0018	1 или 2 байта	0-255 или 0-65535	0-й модуль LMS

При программировании ключа с помощью Guardant API в поле **Счетчик #2** необходимо прописать значение, равное значению основного ресурса в таблице лицензий, т. к. **Счетчик #2** нужен утилитам диагностики для отображения информации о ключе.

Формат таблицы лицензий Guardant Net II/ Net

Для ключей Guardant Net II/ Net наличие таблицы лицензий обязательно только в случае, если необходимо лицензировать приложение, состоящее из нескольких модулей.

Здесь таблица лицензий отвечает только за ресурсы модулей, основной ресурс сетевого ключа содержится в поле **Счетчик #2**.

Формат таблицы Guardant Net II/ Net:

Адрес	Размер	Значение	Комментарии
0000	1 байт	Размер модуля + Количество записей в таблице	Если старший бит байта установлен в 1, то ресурсы модулей имеют размер – 2 байта, иначе – 1 байт. Остальные 7 бит содержат количество записей от 1 до 127 или насколько хватит памяти ключа. Для старых клиентов в новом сервере можно использовать лицензирования по копиям приложения, если в старшем бите модуля указать 1. Так 128 – лицензирования по основному ресурсу в режиме лицензирования по копиям. 129 – модуль №0 по копиям и т. д.
0001	1 байт	Зарезервировано	Не используется

** Для подсчета CRC таблицы лицензий входные параметры функции GrdCRC должны быть следующими:

pData = начиная с флагов таблицы (0008 Dec) - до конца таблицы

dwLng = зависит от количества записей таблицы лицензий; выравнивающий байт не учитывается

dwPrevCRC = -1

*** При наличии в таблице одного модуля LMS (помимо записи с реальным сетевым ресурсом ключа), значение этого поля будет равно двум – и т. д.

Адрес	Размер	Значение	Комментарии
0002	1 или 2 байта	Ресурс 0-го модуля	Размер может быть 1 или 2 байта (максимальное значение 255 и 63355 соотв.), следующая запись лежит прямо за данной записью. Если указан максимальный ресурс, то ресурс модуля считается неограниченным.
0003 или 0004	1 или 2 байта	Ресурс 1-го модуля	Адрес зависит от размера записи

Системные таблицы

Защищенные ячейки хранятся в нижних адресах пользовательской памяти EEPROM и нумеруются, начиная с нулевой.

Числовые имена ячеек хранятся в Таблице числовых имен защищенных ячеек (Protected Item Root Table, PIRT). На каждый элемент таблицы отводится по 2 байта. Числовые имена ячеек не должны совпадать. Числовые имена от 0xFF00 и выше зарезервированы, их использование крайне не рекомендуется.

Адреса памяти EEPROM, по которым располагаются дескрипторы защищенных ячеек, записаны в таблице размещения защищенных ячеек (Protected Item Allocation Table, PIAT). Каждому 16-битовому элементу этой таблицы соответствует защищенная ячейка с соответствующим номером.

На область памяти, занятую дескрипторами, обычно устанавливаются аппаратные запреты на чтение и запись, что препятствует изучению, повторению или модификации защищенных ячеек.

Таблица размещения имеет следующий формат:

Algorithm System Table (AST) – системная таблица описания алгоритмов и ячеек

Адрес*	Размер, байт	Допустимые Значения	Описание
Данные ниже защищены от чтения/записи, если kmAlgoNum!=0 (в памяти ключа присутствует хотя бы одна защищенная ячейка), вне зависимости от установки аппаратных запретов			
Protected Item Main Table (PIMT) – Таблица описания таблиц			
0	1	0	Признак формата современных ключей
1	1	0 или 1	Код формата таблиц и дескрипторов: 0 – все как в Stealth III 1 – формат Guardant Sign/Time/Code. Все описанное ниже относится к этому этот формату
2	2	0-макс.	SAM-адрес первого байта после ВСЕХ алгоритмов и защищенных ячеек. Обязательно для заполнения

*Адрес относительно начала таблицы AMT

Адрес*	**Размер, байт**	**Допустимые Значения**	**Описание**
4	2	Обязательно в пределах AST	SAM-адрес первого байта таблицы PIAT
6	10		Зарезервировано
Данные ниже можно прочитать с помощью GrdRead вне зависимости от установки запретов. Запись запрещена			
16	2	Обязательно в пределах AST	SAM-адрес первого байта таблицы PIRT
Protected Item Root Table (PIRT) – таблица числовых имен ячеек и алгоритмов			
18	2	0 - 0xFF00, остальные имена зарезервированы	Числовое имя первого по счету алгоритма/ячейки (N=0)
18+2	2	0 - 0xFF00, остальные имена зарезервированы	Числовое имя ячейки номер N=1
18+(N)*2	2	0 - 0xFF00, остальные имена зарезервированы	Числовое имя ячейки с номером N
Данные ниже нельзя прочитать с помощью GrdRead, если установлены запреты. Запись запрещена			
Protected Item Allocation Table (PIAT) – таблица числовых имен ячеек/алгоритмов			
18+ kmAlgoNum*2	2	0 - макс. адрес SAM	SAM-адрес первого дескриптора (N=0)
18+ kmAlgoNum*2+ 2	2	0 - макс. адрес SAM	SAM-адрес второго дескриптора (N=1)
18+ kmAl- goNum*2+N*2	2	0 - макс. адрес SAM	SAM-адрес N-го дескриптора
И т. д.			
18+ kmAlgoNum*4	8		Зарезервировано. Адрес определяется исходя из количества защищенных ячеек kmAlgoNum, с учетом того, что на символьное имя и адрес ячейки отводится по 2 байта (суммарно – 4 байта)

Активация/деактивация защищенных ячеек

Защищенные ячейки могут находиться в активированном либо деактивированном состоянии. Состояние ячейки определяется значением 8 бита поля **rs_LoFlags** (**nsafl_InactiveFlag**). Если этот бит установлен, ячейка находится в деактивированном состоянии.

Если защищенная ячейка находится в активированном состоянии, с ней можно осуществлять все действия, которые запрограммированы для этой ячейки. Если ячейка находится в деактивированном состоянии, для нее запрещены любые действия, кроме активации.

Активация и деактивация защищенных ячеек производится при помощи специальных функций Guardant API. Для выполнения операции необходимо указывать соответствующий ей пароль. Для защиты от подбора паролей существует специальный счетчик **rs_ErrorCounter**, который уменьшается на 1 при каждой неудачной попытке. Как только этот счетчик дойдет до нуля, ячейка блокируется в том состоянии, в котором находилась и становится ячейкой с неизменяемым статусом.

Важная информация

Для сервисов активации и деактивации используется один и тот же счетчик, который декрементируется при использовании неправильного пароля в любом из сервисов.

При вводе правильного пароля после нескольких неудачных попыток значение счетчика не восстанавливается!

Ячейка с неизменяемым статусом находится в том состоянии, в котором она находилась на момент исчерпания счетчика неудачных попыток. С такой ячейкой нельзя выполнять никаких операций, кроме операции **GrdTransform** и родственных ей (**GrdCrypt**, **GrdCryptEx**, **GrdHash**, **GrdSign**, **GrdTransformEx** и т. д.), и только в том случае, если ячейка до блокирования находилась в активированном состоянии.

Данное состояние является дополнительным к активному/неактивному, иначе при неудачной попытке обновить данные в алгоритме пользователь потеряет не только возможность работать с новыми функциями программы, но и с теми, что у него есть уже сейчас.

Способы создания защищенных ячеек

Для того чтобы создать защищенную ячейку, нужно сконструировать ее дескриптор. Это можно сделать двумя способами:

При помощи программы GrdUtil.exe

Это наиболее простой и предпочтительный способ создания защищенных ячеек. GrdUtil.exe предоставляет для этого удобный оконный интерфейс. Для создания защищенной ячейки или дескриптора аппаратного алгоритма нужно лишь указать свойства будущего объекта – всю остальную работу программа выполнит сама. Она создаст системные таблицы, запишет в ключ дескрипторы ячеек и установит запреты на чтение/запись занимаемой ими области памяти.

При помощи GrdUtil.exe можно также добавлять новые защищенные ячейки, изменять их свойства, и, при необходимости, удалять.

При помощи специальных функций Guardant API

Этот способ значительно сложнее. Он состоит из следующих этапов:

1. В случае если в ключе уже есть алгоритмы, все их нужно сначала удалить при помощи **GrdInit** или **GrdTRU_SetKey** (при использовании технологии защищенного удаленного программирования ключей Trusted Remote Update).
2. С адреса **kmAlgoAddr** записываются системные таблицы.
3. Со следующего после таблицы байта записываются дескрипторы защищенных ячеек.
4. Операцией **GrdProtect** устанавливаются запреты на чтение и запись для области памяти, занимаемой дескрипторами, а в поле **kmAlgoNum** записывается количество защищенных ячеек. С этого момента дескрипторы невозможно будет считать.

Этот способ имеет серьезные ограничения. По сути, с его помощью можно лишь удалить все существующие защищенные ячейки и создать вместо них новые. Ни добавить новую ячейку к уже существующим, ни изменить дескриптор существующей ячейки, ни произвести выборочное удаление невозможно.

Важная информация

Необходимо принимать во внимание тот факт, что смещения адресов дескрипторов зависит от количества элементов в таблице размещения. Поэтому адреса нужно внимательно рассчитывать во избежание ошибок.

Глава 11
Аппаратные алгоритмы

В этой части подробно описаны аппаратные алгоритмы: для защиты от каких методов взлома они нужны, каково их устройство, как с их помощью можно преобразовать данные и усложнить логику работы ключа.

Важная информация

1. Хотя дескрипторы аппаратных алгоритмов и являются частным случаем защищенных ячеек, аппаратные алгоритмы являются основным защитным механизмом, предоставляемым электронными ключами. Поэтому понимание сути работы аппаратных алгоритмов и правильное использование их для защиты программного обеспечения играет решающую роль в построении надежных систем защиты и повышает защищенность приложений на порядки

2. Обратите особое внимание на устройство алгоритмов и на особенности методов преобразования данных с их помощью.

Назначение

Аппаратные алгоритмы ключей Guardant предназначены для преобразования информации. С их помощью можно шифровать любые данные, используемые защищенным приложением. При правильной организации защиты, использование аппаратных алгоритмов делает бессмысленным удаление хакером вызовов функций Guardant API из кода приложения: в этом случае попросту нечем будет декодировать данные. Кроме того, при правильном использовании аппаратных алгоритмов можно достичь достаточно высокой степени защищенности от эмуляции.

Эмулятор – это программный модуль или драйвер, умеющий воспроизводить процесс обмена с тем или иным ключом и «подсовывающий» защищенному приложению те данные, которые оно ожидает получить. То есть программный эмулятор, по сути, становится полноценной заменой (с точки зрения защищенного приложения, конечно) электронного ключа.

Важная информация

Единственный эффективный путь борьбы с эмуляцией – это усложнение логики работы с ключом. Только в том случае, если ключ обменивается с приложением большими объемами каждый раз разных данных на протяжении как можно больших периодов времени, написание эмулятора становится слишком трудоемким. Только в том случае, если эти данные невозможно ни подсмотреть, ни вычислить заранее, написание эмулятора становится задачей очень трудно реализуемой.

К ним относятся симметричные алгоритмы типа GSII64 и AES128, однонаправленные алгоритмы HASH64 и SHA256, вычисляющие хэш-функцию от исходных данных, алгоритм электронной цифровой подписи на базе эллиптических кривых ECC160, алгоритм генерации случайных чисел RND64.

Особенности

Аппаратные алгоритмы ключей Guardant имеют следующие особенности:

- Преобразование данных происходит не в приложении, а в электронном ключе, что исключает возможность изучения алгоритмов при помощи отладчика и делает бессмысленным удаление из программы модулей защиты.
- Данные преобразуются стойким алгоритмом, секретный ключ которого находится в памяти электронного ключа и не покидает его при выполнении преобразования. Стойкость алгоритма подразумевает, что даже если удастся извлечь сам алгоритм, то на подбор секретного ключа уйдет много вычислительных ресурсов, а вычислить его аналитическими методами не удастся.
- Разработчику приложения известен только секретный ключ шифрования аппаратного алгоритма, а создателям ключей только вид микропрограммы, обрабатывающей этот дескриптор. Таким образом, конкретный вид аппаратного алгоритма, не может быть известен никому.
- Одно и то же защищенное приложение может использовать несколько уникальных дескрипторов аппаратных алгоритмов для преобразования разной информации. Это заставит хакера подбирать вид каждого из них.

Устройство

Дескриптор

Каждый аппаратный алгоритм описывается дескриптором. Дескрипторы хранятся в памяти ключа в защищенных ячейках специального вида и защищены от чтения и модификации. Часть дескриптора описывает свойства алгоритма. Другая его часть представляет собой определитель аппаратного алгоритма. Он играет важнейшую роль в реализации конкретного алгоритма, будучи секретным ключом преобразования.

Детальную информацию об устройстве дескриптора см. в разделе **Дескриптор защищенной ячейки**.

Свойства и их использование

В процессе создания дескриптора участвует также и комбинация свойств аппаратного алгоритма. Свойства задаются специальными флагами, входящими в состав дескриптора. Задавая то или иное сочетание свойств, можно добиться от аппаратного алгоритма нужного «поведения». Свойства аппаратного алгоритма можно комбинировать в зависимости от необходимости.

Алгоритм с ограниченным числом запусков

Если в дескрипторе алгоритма установлен флаг **nsaf_GP_dec**, перед каждым выполнением этого алгоритма значение его счетчика (4-байтовое поле **km_ad_GP** дескриптора) будет уменьшаться на 1. Когда значение счетчика алгоритма достигнет нулевого значения, алгоритм автоматически деактивируется и перестает преобразовывать данные. Это прекрасный способ создания приложения, защищенного с ограничением количества запусков.

Для обратной активации алгоритма, счетчик выполнений которого достиг нуля, необходимо либо перепрограммировать ключ локально, либо записать новое значение в поле **km_ad_GP** дескриптора этого алгоритма, используя TRU.

Можно рассчитать примерное количество обращений к алгоритму за время предполагаемой жизни приложения (или его очередной версии) и ограничить количество выполнений алгоритма соответствующим числом. Это может служить одним из способов противодействия попыткам брутфорса алгоритма.

Активация и деактивация алгоритма

Если в определителе алгоритма заданы эти свойства, то появляется возможность в нужный момент активировать или деактивировать алгоритм, используя специальную функцию . Для активации и деактивации можно задавать пароли, которые также хранятся в дескрипторе алгоритма. При помощи активации и деактивации можно эффективно управлять конкретным набором дескрипторов алгоритмов, участвующих в работе системы защиты.

Более подробно см. **Активация/деактивация защищенных ячеек**.

Зависимость от ID

Флаг **nsaf_ID** устанавливает зависимость аппаратного алгоритма от идентификационного номера (ID) данного ключа. Это означает, что такой алгоритм в каждом ключе будет преобразовывать данные уникальным образом, даже если все значения его дескриптора будут одинаковыми во всех ключах.

Предупреждение

В случае если ключ с таким аппаратным алгоритмом будет случайно испорчен, его будет невозможно заменить ключом с таким же алгоритмом, т. к. не может существовать двух ключей с одинаковыми ID.

Ограничение алгоритма по времени работы

Электронные ключи Guardant с часами реального времени позволяют управлять активностью аппаратных алгоритмов при помощи таймера реального времени с автономным источником питания. Для этого в дескрипторе аппаратного алгоритма предусмотрены специальные поля, в которых хранятся ограничивающие значения времени.

Более подробно см. **Использование таймера для управления статусом аппаратных алгоритмов**.

Секретный ключ (определитель) алгоритма

Определитель аппаратного алгоритма – это набор байтов, записанных в особом поле дескриптора алгоритма и интерпретируемых микропрограммой как секретный ключ алгоритма.

Определители алгоритмов являются критическими данными, относящимися к защите приложения и всегда должны храниться в секрете. Доступ к ним может осуществлять только авторизованный персонал.

В процессе кодирования/декодирования или вычисления хэш-функции определитель никогда не покидает памяти контроллера. Дескриптор алгоритма, содержащий определитель должен быть защищен от чтения/записи аппаратными запретами.

Для генерации определителей лучше всего пользоваться надежными защищенными алгоритмами генерации случайных чисел, например, аппаратным алгоритмом типа RND64. Не рекомендуется использовать для этой цели встроенные функции языков программирования.

Определители полезно время от времени менять. Это очень хорошая и распространенная практика, повышающая защищенность системы. Определители алгоритмов желательно менять при выпуске каждой новой версии защищенного приложения.

Симметричное шифрование

Электронные ключи работают с аппаратными алгоритмами по следующей общей схеме: от защищенного приложения к ключу поступает блок данных (вопрос к ключу), и он при помощи симметричного аппаратного алгоритма преобразует (кодирует или декодирует) эту последовательность. Таким образом, получается ответ ключа, посылаемый защищенной программе.

Чаще всего разработчик системы защиты для какого-либо приложения ограничивается весьма небольшим количеством возможных вопросов-ответов, которые используются на протяжении недолгого периода времени. Это сильно облегчает задачу построения табличных эмуляторов, поскольку для отслеживания всех запросов и ответов требуется то самое недолгое время. Это одна из самых распространенных ошибок при разработке систем защиты.

Для эффективной борьбы с созданием табличных эмуляторов число различных запросов и ответов должно быть как можно большим, а время, за которое они будут все использованы, должно измеряться месяцами.

Алгоритмы типа GSII64, с длиной ключа 128 или 256 бит, а также AES, с длиной ключа 128 бит, реализованные в ключах Guardant, симметрично кодируют и декодируют информацию в самом электронном ключе. Это позволяет значительно расширить возможности электронных ключей Guardant и повысить защищенность программного обеспечения за счет того, что данные для кодирования могут динамически изменяться.

Симметричные алгоритмы семейства GSII64

Важная информация

Аппаратный алгоритм GSII64 есть только в ключах Guardant Sign/Time и их разновидностях. Он не реализован в ключах с загружаемым кодом (Guardant Code и его модификации).

GSII64 – блочный симметричный алгоритм, устойчивый к криптоанализу. Длина секретного ключа GSII64 может составлять 16 или 32 байта (128 или 256 бит). Алгоритм может преобразовать за один цикл блок данных длиной 8 байт, имеет возможность преобразования блоков данных длиной, кратной 8 байтам, а также имеет режимы для преобразования блоков данных произвольной длины.

Алгоритм является симметричным, его можно использовать как для кодирования данных, так и для декодирования.

К семейству GSII64 относятся еще два алгоритма: GSII64_ENCRYPT, GSII64_DECRYPT. Эти алгоритмы по своим свойствам идентичны GSII64, за исключением того, что первый может выполнять только прямое преобразование (зашифрование), а второй – обратное (расшифрование).

Симметричные алгоритмы семейства AES

Алгоритм AES (Advanced Encryption Standard) – это блочный симметричный алгоритм шифрования, принятый в качестве стандарта шифрования США. Длина секретного ключа AES составляет 16 байт (128 бит). Минимальная длина блока данных, преобразуемых алгоритмом за один цикл, – 16 байт. У алгоритма имеются режимы, позволяющие шифровать блоки данных, кратные по длине 16 байтам, и блоки произвольной длины. Симметричность алгоритма означает использование одного и того же секретного ключа шифрования, как для прямого преобразования, так и для обратного.

Подробное описание алгоритма AES можно найти на сайте NIST: http://csrc.nist.gov/archive/aes/index.html

К разновидностям AES, реализованным в ключах Guardant, относятся еще два алгоритма: AES128_ENCRYPT, AES128_DECRYPT. Эти алгоритмы по своим свойствам идентичны AES128, за исключением того, что первый может выполнять только прямое преобразование (зашифрование), а второй – обратное (расшифрование).

Кроме того, в Guardant API реализован также алгоритм AES с длиной ключа 256 бит. В отличие от аппаратного AES128, он реализован программно, то есть выполняется в оперативной памяти и на процессоре компьютера, а не электронного ключа.

Программный алгоритм AES256

Для аппаратных алгоритмов существуют и ограничения, связанные с тем, что скорость аппаратных преобразований, выполняемых электронным ключом, сравнительно невысока. Преобразование больших объемов данных – от десятков килобайт до сотен мегабайт (разного рода базы данных, текстовые и графические файлы и т. п.), повлечет очень существенные задержки в работе защищенного приложения: все же мощность процессоров современных компьютеров во много раз выше, чем процессора электронного ключа.

Это ограничение отчасти можно преодолеть, используя программно реализованные алгоритмы шифрования. Такие алгоритмы используют всю вычислительную мощность компьютера, а потому работают на порядки быстрее, чем алгоритмы электронного ключа. В настоящее время можно легко найти достаточно большое количество реализаций различных алгоритмов. В состав Guardant API включена реализация симметричного алгоритма AES с длиной ключа 256 бит. Этот стойкий алгоритм с успехом можно использовать для надежного кодирования больших объемов данных.

Аппаратное преобразование в этом случае можно эффективно применять для кодирования или генерации ключей шифрования, используемых программно реализованными алгоритмами.

Режимы работы алгоритмов AES и GSII64

Режим ECB

Режим «электронной кодовой книги». Это простейший режим работы блочного симметричного алгоритма. В режиме ECB каждый блок открытого текста, подаваемый на вход алгоритма, преобразуется с одним и тем же определителем в блок шифртекста. Поэтому преобразование 2-х одинаковых блоков даст идентичный результат.

Если длина блока данных превышает минимальную длину блока (16 байт для AES и 8 байт для GSII64), он должен быть разбит на блоки, равные минимальной длине блока. При необходимости, к последнему блоку нужно добавить недостающие байты. Сильно желательно, чтобы байты-заполнители не были постоянными. В качестве байтов-заполнителей можно использовать случайные числа. В этом случае последний закодированный блок требуется хранить полностью, вместе с зашифрованными байтами-заполнителями (а не отбрасывать эти байты). Иначе полезные байты данных из этого блока невозможно будет расшифровать.

Режим ECB подходит для шифрования небольших объемов данных, например, векторов инициализации, используемых в других режимах алгоритма или ключей шифрования других алгоритмов.

Режим CBC

Режим сцепления блоков по шифртексту. В режиме CBC, как и в ECB, каждый блок открытого текста преобразуется в блок шифртекста той же длины. Преобразование в режиме CBC для всех блоков осуществляется с одним и тем же ключом. Режим CBC чаще используется и лучше подходит для преобразования блоков данных, превышающих по длине минимальную длину блока.

Однако в отличие от ECB, преобразование двух одинаковых блоков открытого текста, находящихся в разных позициях исходного блока данных, не даст идентичного результата. Это осуществляется благодаря тому, что на каждом следующем шаге шифруется не сам блок, а его сумма по модулю 2 с предыдущим блоком шифртекста. Для получения первого зашифрованного блока используется сумма по модулю 2 первого зашифрованного блока и некоторого вектора инициализации IV. Значение IV должно быть сохранено для корректного обратного преобразования, но желательно, если оно будет защищено (например, зашифровано в режиме ECB).

Преобразование получается позиционно-зависимым, поскольку результат шифрования зависит не только от самого блока открытого текста, но и от предшествующего ему.

Обратное преобразование также производится поблочно.

Суммарная длина исходного набора данных должна быть кратна минимальной длине блока. В противном случае, к последнему блоку нужно добавить байты-заполнители, так же, как и в режиме ECB.

Режим CBC можно использовать для вычисления надежных контрольных сумм, аутентификации и проверки подлинности данных. В качестве такой контрольной суммы используется последний блок шифротекста. Этот блок зависит от всех предыдущих блоков, а также от вектора инициализации, и вычисляется на основе секретного ключа алгоритма. Он не дает информации об исходных данных, но идентифицирует их практически однозначно. Подделать этот блок так же трудно, как подобрать ключ алгоритма.

Режим CFB

Режим с кодированной обратной связью. Режим CFB позволяет преобразовывать блоки данных произвольного размера, не обязательно кратного минимальной длине блока. Это избавляет от необходимости дополнять исходные данные до целого количества 8 блоков. В этом режиме длина зашифрованной последовательности будет равна длине исходной.

В режиме CFB, подобно режиму CBC, происходит сцепление блоков исходных данных, поэтому каждый закодированный блок будет зависеть от всех предыдущих блоков исходных данных, поскольку для зашифрования каждого следующего блока исходных данных используется предыдущий блок шифртекста.

В этом режиме для преобразования используется вектор инициализации IV (см. режим CBC).

Для однонаправленных алгоритмов, выполняющих только зашифрование или только расшифрование этот режим недоступен, поскольку математически прямое и обратное преобразования в этом режиме эквивалентны.

Важная информация

Если при декодировании указан неверный вектор инициализации, все данные, кроме первого блока, все равно расшифруются правильно. Если это критично для приложения, предпочтительно использовать режим OFB.

Режим OFB

Режим с обратной связью по выходу. Этот режим имеет много общего с режимом CFB.

Главное отличие состоит в том, что для зашифрования следующего блока используется не предыдущий блок шифртекста, а результат преобразования вектора инициализации.

Преимущество данного режима заключается в том, что при передаче зашифрованных данных снижается зависимость от искажений предыдущих блоков. То есть при повреждении одного блока, остальные блоки при расшифровании не пострадают.

У этого преимущества есть и обратная сторона – режим OFB обеспечивает меньшую защиту от искажения данных, поскольку при изменении одного бита шифртекста в расшифрованных данных будет изменен тот же бит. Для проверки подлинности данных в таком случае нужно пользоваться надежной контрольной суммой.

В этом режиме, так же как и в 2-х предыдущих, для преобразования используется вектор инициализации IV (см. режимы CBC и CFB).

Для однонаправленных алгоритмов, выполняющих только зашифрование или только расшифрование этот режим недоступен, поскольку математически прямое и обратное преобразования в этом режиме эквивалентны.

Рекомендации по работе с вектором инициализации IV

Для корректного преобразования данных симметричными алгоритмами GSII64 и AES необходимо принимать во внимание что:

- Вектор инициализации IV должен иметь одинаковые значения перед началом зашифрования и перед началом расшифрования
- Необходимо сохранять значение вектора инициализации IV между обращениями к GrdCrypt (GrdCryptEx) или GrdTransform при продолжении зашифрования/расшифрования больших блоков (больше 248 байт для ECB и CBC и 255 байт для CFB и OFB для алгоритма GSII64)
- При шифровании данных типа разных записей БД или секторов диска, инициализировать IV этим номером записи/ сектора для того, чтобы каждая из них кодировалась всегда одинаково, а разные записи с одинаковыми значениями кодировались по-разному.

Однонаправленное преобразование (вычисление хэш-функции)

Наряду с симметричными преобразованиями электронные ключи могут вычислять однонаправленные функции (хэш-функции): HASH64 с секретным ключом длиной 128 или 256 бит (только для Guardant Sign/ Time / Net) и SHA256 с ключом 256 бит.

Такие преобразования характеризуются тем, что по результату преобразования практически невозможно восстановить исходные данные. Наличие секретного ключа обеспечивает уникальность преобразования, что часто используется при защите данных от подделки.

Эти свойства однонаправленных функций широко используются для проверки подлинности данных, например, паролей, а также для вычисления надежных контрольных сумм, аутентификации и других задач. Поскольку в память электронных ключей можно записывать одинаковые дескрипторы аппаратных алгоритмов, однонаправленное преобразование можно использовать и для проверки подлинности данных, которыми обмениваются разные защищенные приложения или разные части одного и того же приложения.

Алгоритм хэширования SHA256

Алгоритм хэширования SHA256 представляет собой разновидность алгоритма типа SHA-2 (Secure Hash Algorithm v.2). Число 256 в названии алгоритма говорит о том, что длина результирующего хэша составляет 256 бит (32 байта).

Минимальная длина блока данных, хэш которых может быть вычислен, составляет 0 байт.

Если длина массива входных данных составляет менее 32 байт, к блоку данных по специальным правилам, оговоренным в описании стандарта, автоматически добавляются биты-заполнители. Разработчику защиты не нужно реализовывать этот алгоритм самостоятельно.

Более подробно об алгоритме SHA256 можно прочитать здесь: http://www.csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

Алгоритм хэширования HASH64

Важная информация

Аппаратный алгоритм HASH64 есть только в ключах Guardant Sign/Time и их разновидностях. Он не реализован в ключах с загружаемым кодом (Guardant Code и его модификации).

HASH64 – это алгоритм вычисления 64-битной хэш-функции на основе блочного симметричного алгоритма с секретным ключом. Длина секретного ключа алгоритма HASH64 может составлять 16 или 32 байта (128 или 256 бит). Минимальная длина блока данных, хэш которых должен быть вычислен, составляет 16 байт. Длина блока данных должна быть кратной 8 байтам.

Основное свойство алгоритма HASH64 состоит в его однонаправленности. Это означает, что, даже зная значение хэш-функции для какого-либо произвольного набора данных, нельзя восстановить сами данные.

Алгоритм HASH64 используется для вычисления надежных контрольных сумм, аутентификации и проверки подлинности данных.

Если длина массива данных не кратна 8 байтам или составляет менее 16 байт, к блоку данных нужно добавить недостающие байты. Сильно желательно, чтобы байты-заполнители не были постоянными. В качестве байтов-заполнителей можно использовать случайные числа. В этом случае весь блок данных требуется хранить полностью, вместе с байтами-заполнителями (а не отбрасывать эти байты).

Рекомендации по работе с вектором инициализации IV

Для корректного преобразования данных алгоритмом HASH64 необходимо принимать во внимание следующее:

- Чтобы результаты вычисления хэша одних и тех же данных алгоритмами с одинаковыми определителями совпали, вектор инициализации IV должен иметь одинаковые значения;
- Необходимо сохранять значение вектора инициализации IV между обращениями к алгоритму при продолжении вычисления хэша больших блоков (больше 248 байт);
- При вычислении, к примеру, разных записей БД или секторов диска, рекомендуется инициализировать IV этим номером записи/сектора для того, чтобы вычисление хэша каждой из них выполнялось всегда одинаково, а разные записи с одинаковыми значениями имели разные значения хэш-функции.

Вычисление электронной цифровой подписи: алгоритм ECC160

ECC160 – алгоритм электронной цифровой подписи, в основе которого лежит математический аппарат эллиптических кривых. Этот алгоритм был специально разработан для использования в электронных ключах. Длина секретного ключа ECC160 составляет 20 байт (160 бит). Секретный ключ этого алгоритма хранится в определителе аппаратного алгоритма соответствующего типа. Минимальная длина блока данных, преобразуемых алгоритмом за один цикл, составляет 20 байт, при этом на выходе алгоритма получается блок данных, содержащий подпись, длиной 40 байт. Поскольку алгоритм является асимметричным, для проверки подписи используется открытый ключ длиной 40 байт (320 бит). Этот ключ должен храниться в защищенном приложении. Во избежание подмены не следует хранить открытый ключ в чистом виде (см. **Рекомендации программисту**).

Асимметричные функции типа ECC160 позволяют осуществлять электронную цифровую подпись данных на секретном ключе, который хранится в дескрипторе аппаратного алгоритма, и последующую проверку подписи на открытом ключе чисто программными методами. Проверка подписи позволяет убедиться в том, что подпись выполнена именно электронным ключом, поскольку к секретному ключу доступа нет. Применение асимметричных алгоритмов сильно затрудняет построение эмуляторов за счет того, что для проверки можно использовать не заранее известные данные, а любые, сгенерированные случайным образом в процессе работы приложения.

Таким образом, мы избавляемся от «классической» схемы «вопрос-ответ», свойственной симметричным и однонаправленным функциям, и подразумевающей использование полученных заранее пар исходных и преобразованных данных, на которых, собственно, и основаны табличные эмуляторы.

О том, как правильно использовать асимметричные алгоритмы, см. раздел Приложение А. **Рекомендации программисту**.

Генерация случайных чисел: алгоритм RND64

Важная информация

Аппаратный алгоритм RND64 есть только в ключах Guardant Sign/Time и их разновидностях. Он не реализован в ключах с загружаемым кодом (Guardant Code и его модификации).

RND64 представляет собой алгоритм генерации 64-битных случайных чисел на основе блочного симметричного алгоритма с секретным ключом. Длина секретного ключа алгоритма RND64 может составлять 16 или 32 байта (128 или 256 бит). Минимальная длина блока данных, инициализирующих генератор, составляет 8 байтов. Длина блока данных должна быть кратной 8 байтам.

Для получения случайных чисел при помощи алгоритма RND64 используется функция **GrdTransform** (или **GrdTransformEx**), при вызове которой необходимо указать в параметре **dwAlgoNum** числовое имя соответствующего алгоритма. Содержимое буфера **pData** на результат влияния не оказывает.

Загружаемый код (только Guardant Code / Code Time)

Электронный ключ с возможностью загрузки кода приложения представляет собой своего рода доверенную платформу, позволяющую производить значимые вычисления вне центрального процессора и оперативной памяти компьютера, на котором работает защищенное приложение.

Традиционно электронные ключи «умели делать» две основные вещи: а) хранить какие-либо данные, критичные для работы защищаемого приложения, б) выполнять внутри ключа алгоритмы, преобразующие данные по неким алгоритмам.

Эти алгоритмы, как правило, представляют собой либо симметричные алгоритмы шифрования, либо однонаправленные функции с секретным ключом преобразования.

Разработчики систем защиты, основанных на «традиционных» электронных ключах, зачастую испытывают сложности с выбором данных, которые можно было бы подвергать преобразованиям. Как правило, такие данные приходится генерировать искусственно, а не использовать поток реальных данных, на которых производятся вычисления.

В связи с этим существуют проблемы, связанные с тем, что преобразованные аппаратным алгоритмом данные, во-первых, далеко не всегда можно использовать напрямую в приложении. Во-вторых, нужно очень постараться, чтобы поток этих данных был разнообразным на протяжении достаточно длительного времени.

Защитный механизм загружаемого кода основан на том, что алгоритм, запрограммированный в ключ самим разработчиком, обрабатывает натуральные (естественные) данные, получаемые в процессе работы приложения. Обработанные данные можно использовать в приложении напрямую, исключая проверки валидности, которые, как правило, сводятся к одной-двум ассемблерным командам. Поток натуральных данных не постоянен и разнообразен. Поэтому алгоритм загружаемого кода с большой вероятностью будет производить вычисления на постоянно меняющихся данных, если алгоритм этот верно выбран и корректно реализован.

Использование аппаратных алгоритмов

Guardant API – основной инструмент для работы с аппаратными алгоритмами из приложения. В зависимости от задачи, стоящей перед разработчиком, выбирается тот или иной тип алгоритма и соответствующая функция GrdAPI.

Симметричное преобразование: функция GrdCryptEx

С помощью функции **GrdCryptEx** происходит вызов симметричных алгоритмов типа GSII64 или AES.

Преобразование при помощи этой функции имеет следующие свойства:

- Взаимообратность
- Стойкость к криптоанализу
- Выполнение преобразования электронным ключом

Такое преобразование используется для кодирования и декодирования данных, используемых приложением, внутри ключа. Объем преобразуемых данных должен быть относительно невелик.

Однонаправленное преобразование: операция GrdHashEx

Преобразование при помощи операции **GrdHashEx** обычно производится при анализе легальности копии приложения. Главное предназначение этого метода – усложнить логику работы ключа и тем самым предотвратить его эмуляцию. Это преобразование можно использовать для контроля целостности и подлинности данных.

Преобразование имеет два важных свойства:

- Однонаправленность, т. е. для функции F(X) не существует функция $F^{-1}()$ такая, что $X=F^{-1}(F(X))$.
- Стойкость к криптоанализу
- Выполнение преобразования электронным ключом

Исходя из этих свойств, можно очертить круг задач, которые можно решать при помощи такого преобразования. Его можно использовать в случае:

- Если объем преобразуемой информации невелик (десятки и сотни байт)
- Для кодирования данных, которые не потребуется затем декодировать в исходный вид. Например, в простейшем случае приложение может преобразовать некую произвольную последовательность данных, подсчитать хэш ответной последовательности и сравнить его с эталонным хэшем, полученным на этапе установки защиты. Если значения совпали, то копия защищенного приложения считается легальной.

- В комплексе с программно реализованными симметричными алгоритмами. Например, для получения ключа алгоритма AES.

Кодировать же данные, используемые приложением, этим методом не следует, т. к. преобразование однонаправленное.

Выполнение загружаемого кода: функция GrdCodeRun (только для Guardant Code/Code Time)

Вызовы загружаемого кода осуществляются при помощи функции GrdCodeRun(). Если загружаемый код разработан в соответствии с рекомендациями и удовлетворяет требованиям, перечисленным в начале следующей главы, он реализует алгоритм, результаты работы которого можно использовать в приложении напрямую.

Таким образом, отсутствие электронного ключа лишает приложение части важной функциональности, без которой его работа перестает быть возможной.

Приемы работы с аппаратными алгоритмами

Очень важно использовать аппаратные алгоритмы творчески. Ведь преобразовывать информацию можно целым рядом способов, преследуя при этом самые разные цели. От того, насколько оригинально и необычно сконструированная защита использует аппаратные алгоритмы и преобразуемые ими данные, напрямую зависит надежность защиты.

Несколько аппаратных алгоритмов в одном ключе

В одном ключе Guardant можно создавать дескрипторы аппаратных алгоритмов, число которых ограничивается только размером памяти ключа. В любой момент времени можно использовать любой из активированных дескрипторов. Для защиты одного и того же приложения можно использовать все созданные дескрипторы: например, какая-то часть данных кодируется одним алгоритмом, какая-то – другим и т. д.

Конструирование и запись в элетронный ключ дескрипторов алгоритмов происходит при помощи утилиты программирования **GrdUtil.exe**. Также возможна разработка собственных альтернативных утилит, выполняющих аналогичные функции, но программирующей ключи согласно технологии, принятой разработчиками для конкретной компании.

Возможность активации и деактивации аппаратных алгоритмов позволяет записывать их дескрипторы «впрок», для последующего использования в будущих версиях приложения. До выхода новой версии эти алгоритмы могут находиться в деактивированном состоянии, а при переходе на новую версию – активироваться.

Если компания производит несколько программных продуктов, можно и нужно каждый из них защищать с использованием своих уникальных дескрипторов аппаратных алгоритмов. Также возможен вариант, когда в каждом электронном ключе создаются свои уникальные дескрипторы – и тогда каждая копия приложения будет защищена уникальным методом. Этим исключается опасность создания универсальных эмуляторов для разных программных продуктов (или для разных версий одного продукта).

Комбинация аппаратных и программных алгоритмов

Этот метод рекомендуется использовать при необходимости работы с большим объемом данных, которые использует защищенное приложение.

При этом аппаратный алгоритм должен декодировать или преобразовывать ключ программного алгоритма AES256, который будет использоваться для кодирования основного объема данных.

Смысл преобразования ключа перед его использованием – повысить уровень защищенности кодированных данных. В этом случае данные преобразуются не по паролю, вводимому пользователем, а по его видоизмененной форме, которая не хранится нигде.

Случайные числа и анализ вероятностной функции

На вероятностном принципе можно строить надежные схемы защиты. Например, создается алгоритм, который возвращает случайную последовательность размером 100 байт. Она разбивается на 50 чисел по 2 байта, рассчитывается математическое ожидание, среднеквадратичное отклонение и определяется, действительно ли вероятностная функция имеет равномерное распределение. Если это не так, значит, хакер пытается эмулировать защиту от копирования, т. к. ему сложно (да и негде) взять столько правильных случайных последовательностей с равномерным распределением. Если приложение периодически обращается к ключу, то статистика может накапливаться и уточняться при каждом обращении, что будет давать с каждым разом все более точную статистическую картину.

Т. о., чтобы взломать защиту, надо написать весьма громоздкую программу-надстройку, которую даже неясно как внедрять в приложение. Либо придется анализировать всю систему защиты в совокупности, на что может уйти очень много времени, если система сбора и анализа статистики имеет распределенный по приложению характер.

Использование таймера для управления статусом аппаратных алгоритмов

Наличие в моделях Guardant Time/Net Time/ Code Time таймера реального времени (Real Time Clock, RTC) дает возможность управлять активацией и деактивацией аппаратных алгоритмов в этих ключах, а также периодическим изменением определителя алгоритма в зависимости от времени.

Эти возможности используются для управления сроками лицензий на защищаемые приложения, а также для автоматической перестройки защитных механизмов с течением времени.

Деактивация по таймеру

При решении задач по ограничению срока действия лицензии могут возникнуть следующие потребности:

- завершить действие лицензии в заданное время (дата окончания действия лицензии)
- завершить действие лицензии через определенное время, начиная с первого использования (срок действия лицензии)

Эффективным методом ограничения срока лицензии является ограничение времени работы одного или нескольких аппаратных алгоритмов, используемых в системе защиты. В период действия лицензии аппаратный алгоритм должен корректно преобразовывать данные, а по окончании этого срока алгоритм блокируется (переводится в неактивное состояние) и перестает обрабатывать данные. Т. о., выполнение защищенного приложения также блокируется.

Для ограничения срока лицензии конкретным временем в дескрипторе аппаратного алгоритма должно быть установлено значение поля **rs_DeadTime** и флаг **nsafh_DeadTime** в поле флагов **rs_HiFlags**. При наступлении времени **rs_DeadTime** алгоритм будет переведен в деактивированное состояние. Изначально алгоритм должен находиться в активном состоянии.

Для ограничения срока лицензии заданным периодом в дескрипторе аппаратного алгоритма должно быть установлено значение поля **rs_LifeTime** и флаг **nsafh_LifeTime** в поле флагов **rs_HiFlags**. По истечении времени **rs_LifeTime** алгоритм будет переведен в деактивированное состояние. Условием использования данного ограничения является отсутствие установленных флагов **nsafh_DeadTime** и **nsafh_BirthTime** в поле флагов **rs_HiFlags**, то есть должна быть запрещена активация и деактивация алгоритма иными методами.

Активация в заданное время

Для активации алгоритма в заданное время в его дескрипторе должно быть установлено значение поля **rs_BirthTime** и флаг **nsafh_BirthTime** в поле флагов **rs_HiFlags**. При наступлении времени **rs_BirthTime** алгоритм будет автоматически переведен в активное состояние. При этом сам алгоритм изначально должен находиться в неактивном состоянии.

С этого момента аппаратный алгоритм сможет преобразовывать данные, и, соответственно, может быть задействован в защитном механизме. Сама схема такого механизма может выглядеть как «передача эстафеты» от автоматически деактивированного алгоритма к активированному. Эти алгоритмы могут обрабатывать данные поразному, а потому изменится и сам алгоритм защиты, что увеличивает стойкость.

Кроме того, автоматическая активация позволяет при необходимости задавать момент начала действия лицензии.

Автоматическое изменение определителя алгоритма

Другим способом динамически изменять защитные механизмы является возможность модифицировать определитель алгоритма по таймеру. При этом алгоритм постоянно остается активным, и «передачи эстафеты», как и в случае, рассмотренном выше, не происходит.

Для использования данного свойства алгоритма необходимо задать периодичность смены определителя **rs_DaysGap** и время **rs_ChangeFlipTimeStart**, начиная с которого будет производиться периодическая смена.

Можно комбинировать автоматическое изменение определителя с режимами ограничения времени жизни при помощи флагов **nsafh_DeadTime**, **nsafh_BirthTime** и **nsafh_LifeTime**.

Этот сервис доступен только для симметричных алгоритмов шифрования (AES128 и GSII64).

Глава 12
Работа с ключами Guardant Code /Code Time

Важная информация

Существует два поколения ключей Guardant Code – на микроконтроллере архитектуры ARM7 и микроконтроллере архитектуры Cortex-M3. Ключи на ARM7 сняты с производства 1 июля 2011 года. Вся информация в данной главе относится только к новой модели.

Электронные ключи Guardant Code и Guardant Code Time представляют собой новое поколение аппаратных средств защиты программного обеспечения семейства Guardant. Основным защитным механизмом, реализуемым этими ключами, является возможность разработки и размещения в электронном ключе собственных алгоритмов, реализующих полезную для защищаемого приложения функциональность. Эти алгоритмы в дальнейшем будут называться загружаемым кодом. Выполнение загружаемого кода происходит внутри ключей Guardant Code / Code Time, выполняющих роль своего рода доверенной платформы.

Устройство ключей Guardant Code

Важная информация

Обратите особое внимание на отличия методов адресации памяти и на то, какие области памяти доступны для использования.

Память ключа Guardant Code

В контроллере электронных ключей Guardant Code существует два вида памяти: **оперативная** (RAM) и **постоянная** (Flash).

Оперативная память (далее – RAM) используется для хранения переменных, стека, буферов ввода-вывода. Часть RAM используется системной микропрограммой (System RAM), а другая часть – загружаемым кодом (20 Кб).

Память, которая используется для хранения загружаемого кода и его констант, в дальнейшем будет называться **Flash-памятью**. Часть Flash-памяти занята системной микропрограммой (System Flash), другая часть предназначена для загружаемого кода (128 Кб или 352 Кб, в зависимости от модификации ключа).

В предыдущих моделях ключей Guardant энергонезависимая память для хранения ключевой информации защищаемого приложения была выполнена в виде отдельной микросхемы EEPROM (электрически перепрограммируемая постоянная память). Поэтому для сохранения преемственности и совместимости EEPROM эмулируется во Flash-памяти контроллера Guardant Code. В ней (точно так же, как в обычных ключах Guardant) хранятся системные поля данных, дескрипторы защищенных ячеек и аппаратных алгоритмов. Для ясности и краткости эта эмулируемая память в дальнейшем будет называться просто **EEPROM**.

Организация EEPROM

Ключи с загружаемым кодом обладают EEPROM объемом 4096 байт. Организация этой области памяти идентично устройству EEPROM ключей Guardant Sign/Time/Sign Net/ Time Net и детально рассматривается в **Главе 10. Организация EEPROM памяти ключей Guardant**.

Карта Flash-памяти и RAM

При работе с RAM-памятью и Flash-памятью для хранения загружаемого кода используется иная, чем в случае с EEPROM, адресация, основанная на карте памяти контроллера ключа.

Карта памяти, доступная для загружаемого кода, выглядит так:

Адреса	Назначение
00020000h-0003FFFFh	Flash-память для размещения загружаемого кода, страницы 1-4 (для варианта с 128 кб Flash-памяти)
00040000h-00077FFFh	Flash-память для размещения загружаемого кода, страницы 5-11 (для варианта с 352 кб Flash-памяти)
40003000h-40007FDFh	RAM (ОЗУ), доступная загружаемому коду. Тут размещаются: стек, буфер ввода-вывода, переменные загружаемого кода

Flash-память разделена на страницы по 32 Кб. При этом страница может быть занята только целиком. Вследствие такой организации памяти в Guardant Code может существовать от 1 до 4 (или до 11, в зависимости от объема доступной Flash-памяти) отдельных сегментов загружаемого кода.

Как правило, необходимости в нескольких независимых сегментах загружаемого кода не возникает, а потому в подавляющем большинстве случаев размещения для загружаемого кода резервируется вся доступная Flash-память.

Разработка приложений для Guardant Code

Выбор кода для размещения в ключе

Это самый ответственный и нетривиальный этап разработки загружаемого кода. Суть его заключается в том, что разработчику нужно решить, какой именно код будет выполняться в электронном ключе.

Код должен быть устроен таким образом, чтобы выполнять определенную конечную задачу, которая в общем виде выглядит так:

- Получение буфера с входными данными
- Вычисления над данными из этого буфера
- Возврат буфера с выходными данными

Существует целый ряд требований к этому коду, налагающих достаточно серьезные ограничения на выбор. Требования условно можно разделить на несколько типов:

Требования по безопасности

Загружаемый код должен быть достаточно сложным, чтобы брутфорс или иные (более эффективные и продвинутые) методы анализа черных ящиков не сделали возможным создания эмулятора в короткое время.

Этот код должен отсутствовать в более ранних версиях приложения. Несоблюдение этого условия делает возможным сравнение версий приложения и нахождение перенесенного кода для внедрения его в эмулятор.

Требования по производительности

Контроллер ключа обладает достаточно большими вычислительными возможностями, однако мощность его все же гораздо ниже, чем у современных процессоров. Поэтому важно, чтобы код не был слишком ресурсоемким, в противном случае время его выполнения может возрасти до неприемлемого уровня.

Кроме того, код, загружаемый в ключ, не должен вызываться слишком часто, например, в цикле. Если при единичном вызове задержка при выполнении не будет значительной, то при циклическом вызове она может оказаться очень существенной.

Требования по реализуемости

Код, размещаемый в электронном ключе, не должен:

- Содержать вызовов API, которые нельзя было бы перенести в электронный ключ, или иметь внешние зависимости
- Использовать потоки ввода-вывода
- Использовать вывод на консоль
- Использовать динамическое распределение памяти

Коду, исполняемому внутри ключа, доступны лишь стандартные библиотеки C и функции для работы с электронным ключом.

Ключи Guardant Code позволяют исполнять алгоритмы до 20 тысяч строк кода на C (до 60 тысяч строк в моделях ключей с увеличенным размером памяти). Соответственно, размер кода должен укладываться в эти пределы.

Средства разработки

Несмотря на то, что для процессоров CORTEX-M3 существуют компиляторы многих языков программирования, эффективнее всего использовать язык C, а точнее его подмножество, основой для которого является стандарт ANSI C.

Требование соблюдения стандарта связано с тем, что первоначальная разработка и отладка загружаемого кода может выполняться с использованием различных компиляторов. Т. е. предполагается, что код разрабатывается в привычной интегрированной среде, а затем компилируется для использования в Guardant Code. Поэтому для разработки загружаемого кода пригодятся навыки кроссплатформенного программирования.

Код перед загрузкой в ключ должен быть скомпилирован. Поскольку система команд процессора архитектуры CORTEX-M3, используемого в ключе, отличается от системы команд x86-совместимых процессоров, для работы потребуется компилятор, способный генерировать двоичный код, совместимый с архитектурой CORTEX-M3.

Для компиляции загружаемого кода можно применять как коммерческие средства, так и распространяемые под различными «свободными» лицензиями. К последним можно отнести GCC (включая различные решения на его основе, такие как YAGARTO).

Большинство средств разработки, распространяемых под свободными лицензиями, рассчитаны на работу под Linux. Тем не менее, существуют решения и для Windows. Кроме того, есть возможность использовать средства разработки, предназначенные для Linux под операционными системами семейства Windows, применяя для их запуска Cygwin или MinGW.

Примеры и makefile, входящие в комплект разработчика Guardant, рассчитаны на использование компилятора **GCC** и инструментария **YAGARTO**, как на самый доступный вариант.

Хотя для большинства высокоуровневых языков программирования перенос кода на C достаточно несложен, но альтернативно, можно использовать и другие языки программирования высокого уровня, для которых есть компилятор CORTEX-M3. Однако в данном документе эта возможность не описывается.

Guardant Code API. Интерфейс прикладного программирования загружаемого кода

При разработке загружаемого кода с большой вероятностью может возникнуть необходимость обращаться к ресурсам ключа, находящимся в области **EEPROM** (защищенным ячейкам, алгоритмам), или к таймеру. Поэтому был разработан специальный интерфейс прикладного программирования **Guardant Code API** (см. Справочную систему Guardant API, файл **GrdAPI.chm**) Библиотека этого API содержит большинство функций Guardant API, адаптированных для использования из среды загружаемого кода.

Основной нюанс при работе с **Guardant Code API** состоит в том, что хэндл защищенного контейнера в загружаемом коде теряет смысл, поскольку этот код, во-первых, имеет доступ только к одному ключу, а во-вторых, не существует ситуации конкурентного доступа к ресурсам ключа из разных потоков одного приложения и из разных приложений.

Вместе с тем, функциям внутренного API загружаемого кода передается параметр типа HANDLE. Это сделано для соблюдения единообразия и удобства отладки загружаемого кода.

Guardant Code API поддерживает основные функции Guardant API, связанные с хранением данных и работой с алгоритмами.

Кроме того, в API загружаемого кода существует возможность вызывать криптографические алгоритмы, не используя дескрипторы, а напрямую, подобно тому, как в Guardant API вызываются программно-реализованные алгоритмы. Для этого вместо числового имени ячейки, содержащей дескриптор, указывается специальное зарезервированное имя алгоритма.

Если в загружаемом коде присутствуют функции Guardant API (например, в ключ переносится алгоритм, который раньше защищался ключами Guardant), то для большинства этих функций существуют аналоги в Guardant Code API, и портирование будет заключаться в смене префикса с GrdXXX на GcaXXX или GccaXXX.

Название функции	Краткое описание
GcaCrash()	Инициировать ошибку среды исполнения Guardant Code
GcaExit()	Выйти из загружаемого кода. В вызывающее приложение передается код возврата
GcaLedOn()	Включить светодиодный индикатор
GcaLedOff()	Выключить светодиодный индикатор
GcaRead()	Прочитать данные из EEPROM, аналогично GrdRead()
GcaWrite()	Записать данные в EEPROM, аналогично GrdWrite()
GcaPI_Read()	Прочитать данные из защищенной ячейки, аналогично GrdPI_Read()
GcaPI_Update()	Изменить данные защищенной ячейки, аналогично GrdPI_Update()
GcaPI_GetTimeLimit()	Получить время жизни защищенной ячейки, аналогично GrdPI_GetTimeLimit(). Для Guardant Code Time
GcaPI_GetCounter()	Получить значение счетчика оставшихся выполнений алгоритма, аналогично GrdPI_GetCounter()
GcaGetTime()	Получить текущее время из RTC ключа, аналогично GrdGetTime(). Для Guardant Code Time
GcaGetRTCQuality()	Проверить валидность значения RTC, аналогично GrdGetRTCQuality(). Для Guardant Code Time
GcaGetLastError()	Получить последний код ошибки, аналогично GrdGetLastError(). Возвращает код ошибки, которая произошла при последнем вызове функции Guardant Code API
GccaCryptEx()	Шифровать данные симметричным алгоритмом, аналогично GrdCryptEx(). Для программного алгоритма AES128 требуется другой размер контекста. Контекст должен быть выровнен по границе в 4 байта (!) применением макроса ALIGNED
GccaSign()	Вычислить ЭЦП, аналогично GrdSign(). Присутствует дополнительный параметр – ключ подписи и вариант вызова, работающий в обход таблицы дескрипторов
GccaVerifySign()	Проверить ЭЦП, аналогично GrdVerifySign()
GccaGenerateKeyPair()	Генерировать ключевую пару для алгоритма ЭЦП ECC160
GccaHash()	Вычислить хэш-функцию, аналогично GrdHash()
GccaGetRandom()	Генерировать случайное однобайтовое число
GcaSetTimeout	Установить разрешенное время работы загружаемого кода
GcaCodeGetInfo	Запросить информацию из дескриптора загружаемого кода
GcaCodeRun	Выполнить код из другого участка загружаемого кода

Важная информация

1. Подробную информацию по функциям внутреннего Guardant Code API см. в Справочной системе по Guardant API (файл **GrdAPI.chm**).

2. Поскольку в Guardant Code не реализован алгоритм GSII64 и производные от него (HASH64, RAND64 и т. д.), возможно придется немного переработать существующую схему защиты на использование алгоритмов AES128 для шифрования и SHA256 для хэширования. Все остальные возможности предыдущих поколений ключей присутствуют и в Guardant Code.

Сервисные функции GrdAPI для работы с загружаемым кодом

Загружаемый код, в отличие от ячеек с данными и дескрипторов аппаратных алгоритмов, хранится в другой области памяти ключа, для которой используются иные принципы адресации. Поэтому для загрузки кода в память ключа, его выполнения и других операций существуют специальные функции Guardant API.

Для записи в ключ загружаемый код преобразуется в файл специального формата **GCEXE** (Guardant Code Executable), обеспечивающего защиту от подделки, подмены и анализа. Эта защита обеспечивается шифрованием криптостойкими алгоритмами и ЭЦП. Такая защита делает возможным безопасное обновление загружаемого кода в ключе, в том числе при пересылке файлов по открытым каналам и через сети общего пользования.

Файл формата GCEXE генерируется на основе данных дескриптора загружаемого кода, скомпилированного кода и map-файла утилитой программирования ключей **GrdUtil.exe**. Однажды сгенерированный файл может быть использован для записи в тиражируемые ключи той же утилитой. Если предпродажное программирование осуществляется при помощи собственных специально разработанных инструментов, файл с загружаемым кодом может быть записан в ключ при помощи функции GrdCodeLoad().

Список сервисных функций Guardant API:

Название функции	Код доступа	Краткое описание
GrdCodeGetInfo	Private Read	Получить информацию из дескриптора загружаемого кода
GrdCodeLoad	Private Read	Записать GCEXE-файл, содержащий загружаемый код, во Flash-память ключа
GrdCodeRun	Private Read	Выполнить загружаемый код
GrdSetDriverMode	Private Read	Задать USB-режим работы ключа

Подробнее см. в Справочной системе по Guardant API (файл **GrdAPI.chm**).

Компиляция загружаемого кода

Инсталляция и настройка компилятора GCC

Для компиляции загружаемого кода используются компилятор и линкер **GCC**, стандартная библиотека C для встраиваемых систем **newlib**, утилита **make** и несколько сервисных утилит.

Существует два варианта использования GCC:

1. Загрузить исходный код GCC, make, newlib из соответствующих репозиториев и самостоятельно скомпилировать.
2. Воспользоваться готовым комплектом инструментов для разработки под ARM.

Для работы с ключами Guardant Code рекомендуется использовать свободно распространяемый инструментарий **YAGARTO**. Все примеры из комплекта разработчика, тестировались именно на нем.

Чтобы приступить к работе с YAGARTO требуется выполнить несколько простых шагов:

- Загрузить набор утилит YAGARTO с сайта разработчика (http://www.yagarto.de/download/yagarto/yagarto-tools-20100703-setup.exe)
- Загрузить компилятор с сайта разработчика (http://www.yagarto.de/download/yagarto/yagarto-bu-2.21_gcc-4.6.0-c-c++_nl-1.19.0_gdb-7.2_eabi_20110429.html)
- Установить загруженные пакеты в директорию C:\YAGARTO

После выполнения этих шагов компилятор полностью готов к работе. Дальнейшая работа с ним будет выполняться посредством вызова утилиты **make** на заранее созданных и настроенных **makefile**.

Общие сведения о компиляции и сборке

Все инструкции для компилятора и линкера, а также команды для обработки скомпилированного кода, содержатся в конфигурационном файле утилиты **make** (имя этого файла по умолчанию - **makefile**).

Соответственно, для компиляции приложения необходимо:

1. Отредактировать настройки в секции **Main Configuration** внутри **makefile**:
   - Задать желаемое имя точки входа, адреса RAM, ROM
   - Задать требуемый размер стека
   - Задать имена и размеры буферов ввода-вывода
   - Задать путь к системным утилитам **hex2bin.exe** и **map_parse.exe**
   - Задать путь к папке с заголовочными файлами **GrdAPI.h** и **GcaAPI.h**
   - Задать путь к папке с файлом libgcaapi.lib

2. Сгенерировать шаблон проекта: **make template** (для использования компиляторов, отличных от GCC, требуется ручная настройка проекта). Проект должен иметь определенный набор файлов и быть настроен для работы в среде Guardant Code

3. Скопировать в папку проекта файлы с исходными текстами с модулями на C/C++ и добавить имена этих файлов в переменные makefile **SRC** и **CPPSRC** соответственно
4. В скопированных модулях должна присутствовать функция с именем, совпадающим с именем заданной точки входа (и соответствующим прототипом)
5. После этого можно выполнить команду: **make** и приложение будет собрано
6. В папке проекта будет создана папка **.out** с двумя файлами, имеющими расширения **bin** и **bmap**. Эти файлы требуются для генерации GCEXE-файла и загрузки его в электронный ключ при помощи утилиты GrdUtil.exe

Важная информация

В makefile нельзя использовать символ «\» и пути с пробелами. В качестве разделителя необходимо указывать «/». Чтобы избежать пробелов, можно задавать относительные пути, либо копировать необходимые для компиляции файлы в отдельную директорию.

Команды утилиты make

Приложение собирается при помощи GNU-утилиты **make**, которая использует конфигурационный файл с именем **makefile**.

Для утилиты **make** доступны следующие команды:

1. Сборка проекта

Если конфигурационный файл имеет имя по умолчанию (makefile):

```
make
```

Если конфигурационный файл имеет имя, отличное от имени по умолчанию:

```
make –f confname
```

2. Удаление всех файлов, создаваемых при сборке (файлы, создаваемые при make template не удаляются)

```
make clean
```

или

```
make -f confname clean
```

3. Создание шаблона приложения

```
make template
```

или

```
make -f confname template
```

Важная информация

Если в сгенерированные при создании шаблона проекта файлы **Startup.S** и **rom.ld** были внесены изменения, они будут потеряны при повторном создании шаблона!

Полная пересборка приложения

```
make clean
make all
```

или

```
make -f confname clean
make -f confname all
```

Пересборка может требоваться при изменении уровня оптимизации и при добавлении новых файлов (см. следующий раздел).

Важная информация

Если в системе одновременно с GCC установлены другие компиляторы, использующие собственные утилиты make (например, Borland C), при вызове make следует указывать полный путь, поскольку пути к другим утилитам make могут быть прописаны в переменной среды PATH. Можно придумать и иные способы дифференциации.

Настройка универсального makefile

Универсальный **makefile** содержит секции настроек с параметрами:

- Генерации шаблона проекта командой **make template**
- Сборки приложения по команде **make/make all**

При внесении изменений в первую секцию требуется перегенерация шаблона проекта (см. соответствующий раздел). При внесении изменений во вторую секцию требуется пересборка проекта путем подачи команды **make clean** и затем **make all**.

Настройки секции генерации шаблона:

Имя параметра	Значение
CFG_ENTRYPOINT_NAME	Имя точки входа (по умолчанию функция main)
CFG_PROGRAM_ADDR *)	Адрес Flash-памяти, по которому располагается приложение
CFG_PROGRAM_SIZE *)	Размер приложения во Flash-памяти
CFG_RAM_ADDR *)	Адрес начала RAM, резервированной для загружаемого кода
CFG_RAM_SIZE *)	Размер RAM, зарезервированной для загружаемого кода
CFG_INPUT_BUFFER_NAME	Имя буфера ввода, через который данные передаются в загружаемый код
CFG_INPUT_BUFFER_SIZE	Размер буфера ввода
CFG_OUTPUT_BUFFER_NAME	Имя буфера вывода, данные из которого возвращаются вызывающему приложению
CFG_OUTPUT_BUFFER_SIZE	Размер буфера вывода
CFG_STACK_SIZE	Размер программного стека
CFG_INCLUDE_DIR	Путь до директории, содержащей заголовочные файлы GcaAPI.h и GrdAPI.h
CFG_SYS_DIR	Путь до директории, содержащей служебные утилиты
CFG_TARGET_NAME	Имя двоичного bin-файла, получаемого при компиляции

*) Поскольку в ключе под загружаемый код по умолчанию резервируется вся Flash-память и вся RAM, значения в этих пунктах изменять не нужно.

В **makefile** доступны следующие настройки сборки проекта:

Имя параметра	Значение
OPT	Уровень оптимизации. Рекомендуемые значения 2 или s (так же допустимые значения 0 и 1, значение 3 крайне не рекомендуется)
SRC	Набор C-файлов, используемых в проекте
ASRC	Набор ASM-файлов, используемых в проекте

Важно заметить, что задаваемые имена файлов зависят от регистра. К примеру, при несовпадении регистра в имени файла **main.c**, при сборке может возникнуть следующая ошибка:

```
        -------- begin --------
make: *** No rule to make target `main.o', needed by
                        `elf'.  Stop.
```

Несовпадение имен как таковых проверяется отдельно и вызывает более внятную ошибку:

```
        -------- begin --------
File main.c not found! Please check makefile (SRC, ASRC
and CPPSRC values).
```

Точка входа в приложение

При старте приложения, в самом начале начинает исполняться код, находящийся в файле **Startup.S**. Он инициализирует стек и С-окружение (предварительно инициализированные переменные) и обнуляет неинициализированные переменные и область стека, при необходимости вызывает конструкторы глобальных объектов С. После этого он передает управление в приложение на С. Этот файл генерируется автоматически из универсального **makefile**.

По умолчанию точка входа в С-приложении на GCC имеет стандартное имя **main**. Прототип, однако, отличается от стандартного ANSI C и имеет следующий вид:

```
int main(
DWORD dwInDataLng, DWORD dwOutDataLng, DWORD dwP1);
```

Где:

dwInDataLng – размер данных поступивших из PC,

dwOutDataLng – размер данных, который PC запрашивает назад,

dwP1 – параметр dwP1, переданный функции GrdCodeRun().

Если требуется изменить адрес точки входа, то в файле Startup.S требуется исправить строчки:

```
.global main
…
LDR     R4, =main
```

Адресное пространство

В микроконтроллерах на основе ядра CORTEX-M3, на которых построен ключ Guardant Code, имеется единое адресное пространство в 4Гб. Для загружаемого кода доступны следующие диапазоны адресов:

Адреса	Назначение
00020000h-0003FFFFh	Flash-память для размещения загружаемого кода и ROM-секции микропрограммы (для варианта с 128 кб Flash-памяти)
00020000h-00077FFFh	Flash-память для размещения загружаемого кода и ROM-секции микропрограммы (для варианта с 352 кб Flash-памяти)
40003000h-40007FDFh	RAM (ОЗУ), доступная загружаемому коду. Тут размещаются: стек, буфер ввода-вывода, переменные загружаемого кода

Диапазон используемых адресов указывается в **makefile** (параметры **CFG_PROGRAM_ADDR**, **CFG_PROGRAM_SIZE**, **CFG_ RAM, CFG_RAM_SIZE**). Задаваемые адреса должны быть кратны 0x8000 байт, и быть выровнены по границе 32768 байт.

Диапазон адресов, доступных загружаемому коду, описывается в соответствующем дескрипторе аппаратного алгоритма. GrdUtil автоматически заполняет соответствующие поля дескриптора информацией из файла ***.bmap**.

Поскольку по умолчанию под загружаемый код резервируется вся Flash-память и вся RAM, значения этих настроек без насущной необходимости изменять не нужно.

Буферы ввода-вывода

Имена буферов ввода и вывода в **makefile** могут быть разными, а могут и совпадать.

В случае, когда они совпадают, выделяется один буфер, который работает одновременно и на ввод, и на вывод. В С-коде буфер ввода-вывода может быть объявлен так:

```
extern BYTE iodata[];
```

При этом в параметрах **CFG_INPUT_BUFFER_NAME** и **CFG_OUTPUT_BUFFER_NAME** указывается значение **iodata**.

Если же используются раздельные буферы, то каждый из них объявляется в С-коде отдельно, и имеет собственное имя и размер.

При объявлении буферов допустимо использование любых типов данных, однако в случае структур рекомендуется добавлять в определение макрос **ALIGNED**, например:

```
extern struct
    {
     double x;
     …
    } iodata ALIGNED;
```

Это указывает компилятору, что структура выровнена в памяти, и позволяет генерировать более эффективный код для доступа к полям структуры.

По умолчанию размер буфера ввода-вывода установлен равным 1024 байта. Для ввода-вывода в примерах используется единый буфер. Перед запуском загружаемого приложения данные в него помещаются, а после окончания работы возвращаются обратно в PC.

Максимальный суммарный размер буферов для ввода-вывода составляет 0x3F00 байт (16128 байт). Так же в объявлении переменной желательно указание макроса ALIGNED, который говорит компилятору, что буфер выровнен в памяти, и, в некоторых случаях, оптимизировать доступ к данной переменной.

Стек

Размер программного стека для GCC указывается в **makefile**. За это отвечает параметр:

```
CFG_STACK_SIZE = 0x800;
```

Т. е., размер стека по умолчанию равен 2кБ (0x800 байтам).

Поскольку размер RAM достаточно сильно ограничен, рекомендуется небольшие и простые, но часто используемые функции оформлять как **inline**. Можно использовать макрос **INLINE** из **syscalls_public.h**. Например:

```
INLINE void add(int a, int b)
{
return a+b;
}
```

За счет этого происходит экономия памяти стека и увеличивается быстродействие кода.

Устройство загружаемого кода

Прямой перенос кода из исходного приложения может быть сопряжен с определенными трудностями. В общем случае, код, перенесенный в том же виде, как он существует в приложении, будет неработоспособен в электронном ключе. Поэтому код должен быть модифицирован и оптимизирован для выполнения на платформе CORTEX-M3. Желательно, чтобы этот код был написан заново и реализовывал функции, которых в ранних версиях приложения не было, либо эти функции должны быть видоизменены.

Параметры функции main()

Прототип функции **main()** объявляется следующим образом:

```
DORD main(
       DWORD dwInDataLng, DWORD dwOutDataLng, DWORD dwP1)
```

Параметры **dwInDataLng** и **dwOutDataLng** устанавливают количество данных, считываемых из буфера ввода и возвращаемых в буфер вывода. Параметр **dwP1** используется для передачи кода подфункции загружаемого кода.

Параметр **dwP1** передается функции **GrdCodeRun()** и его можно получить в загруженном коде в виде третьего параметра функции **main** (функции, которая первой получает управление в С-коде):

```
DWORD func1(dwInDataLng, dwOutDataLng)
     {
      // Логика работы 1:
      return 101;
     }
DWORD func2(dwInDataLng, dwOutDataLng)
     {
      // Логика работы 2:
      return 102;
     }
DWORD func3(dwInDataLng, dwOutDataLng)
     {
      // Логика работы 3:
      return 103;
     }

DWORD main(DWORD dwInDataLng, DWORD dwOutDataLng, DWORD
                        dwP1) {
     switch (dwP1)
     {
      case 0x01:
      return func1(dwInDataLng, dwOutDataLng);
      case 0x02:
      return func2(dwInDataLng, dwOutDataLng);
      case 0x03:
      return func3(dwInDataLng, dwOutDataLng);
      case 0x04:
      // ...
      default:
      return -1;
     }
}
```

Статические и глобальные переменные

По возможности, переменные лучше объявлять глобально (хотя это и противоречит принципам функционального программирования), а не в теле функции, чтобы не передавать данные через стек. Этим экономится память стека и увеличивается быстродействие.

В загружаемом коде можно создать глобальные переменные, содержимое которых не будет обнуляться между вызовами. Такие переменные требуется объявлять с макросом **NO_INIT**:

```
DWORD buffer[100] NO_INIT
```

Эти переменные являются аналогами статических переменных.

Польза от них может заключаться в возможности запоминания некоторых состояний загружаемого кода. Это делает анализ «черного ящика» гораздо более сложным.

Возврат из загружаемого кода

Выход из приложения можно осуществлять следующим образом:

- Возврат из main при помощи **return**. Код возврата будет помещен в параметр **dwRetCode** функции **GrdCodeRun()**
- Вызов функции **GcaExit()**. Ей также передается код возврата

Кроме того, принудительное завершение приложения происходит в следующих случаях:

- Наступление таймаута времени выполнения загружаемого кода (3 секунды)
- Попытка выполнения приложением недопустимого действия (обращение к недопустимым адресам памяти и т.д.)

В примерах в качестве кода возврата с ошибкой используется значение -1. Для упрощения отладки можно возвращать значения макроса **__LINE__** или пользоваться вызовом **GcaExit(0, __LINE__)**. Этот способ поможет определить строку, на которой произошел выход из приложения. Оставлять в конечных версиях возвраты в данном виде нежелательно, так как это может дать дополнительную информацию для злоумышленника.

Для отладки можно, к примеру, использовать следующий макрос:

```
#define ASSERT(cond){if(cond)GcaExit(0,__LINE__);}
```

ASSERT(x != 0); // Если x!=0, осуществит возврат из программы с указанием номера строки, в которой вставлен ASSERT.

Использование арифметики с плавающей точкой

За вычисления с плавающей точкой отвечает библиотека **libm** из комплекта GCC. Полное описание математических функций, доступных в ней, можно найти в документации к данной библиотеке. Для каждой функции имеется 2 варианта: обычный, для вычислений с двойной точностью (тип double), а также с приставкой «f», для вычислений с половинной точностью (тип float).

Отладка загружаемого кода

Разработка и первоначальная отладка загружаемого кода производится на компьютере. Для этого можно использовать любую IDE и отладчик языка C.

Основная проблема состоит в том, что отлаживать уже загруженный код затруднительно, поскольку нет возможности «залезть» отладчиком в контроллер ключа. Поэтому первоначально отлаживают сам алгоритм загружаемого кода.

Загруженный в ключ код имеет ограниченные возможности для трассировки. Например, нельзя вывести трассу на консоль или записать в файл. Однако некоторые средства все же есть. Для этой цели можно использовать **функции управления светодиодом**. Сигналы, подаваемые с его помощью, можно применять в качестве признаков прохождения тех или иных веток кода.

Как вариант, можно использовать принудительный возврат из загружаемого кода с соответствующим кодом возврата и передачей необходимых для отладки данных через буфер вывода.

Методы отладки кода такие же, как и при разработке на PC. Если в загружаемом коде используются вызовы Guardant Code API, то для отладки не нужно загружать этот код в ключ: можно использовать входящую в комплект разработчика отладочную библиотеку.

Описание отладочной библиотеки

Отладочная библиотека представлена двумя частями:

- Модуль для загрузки в электронный ключ,
- Динамическая библиотека, содержащая функции, прототипы которых аналогичны тем, что доступны для загружаемого кода внутри электронного ключа (GcaXXX и GccaXXX)

Порядок работы с отладочной библиотекой таков:

1. При помощи GrdUtil.exe создается файл маски, в котором один из алгоритмов представляет собой отладочный модуль загружаемого кода – **DebugModule.bin**. При этом нужно убедиться, что соответствующий модулю **bmap-**файл находится в той же директории. Можно взять готовый файл маски **DebugMask.nsd** из примера и изменить его для использования в собственном приложении. От файла маски зависит, какой номер будет у алгоритма, содержащего загружаемый код.
2. Полученный файл маски с отладочным модулем прошивается в электронный ключ при помощи GrdUtil.
3. К проекту загружаемого кода на PC подключается отладочная библиотека **gcaapidll.dll**. Для этого используется библиотека экспортов **gcaapidll.lib**. Файл **GcaAPIdll.h** содержит описание прототипов функций GcaXXX/GccaXXX.
4. Перед вызовами функций GcaXXX/GccaXXX из **gcaapidll.dll** в исходном коде следует разместить вызов макроса **DEBUGDLL_INIT(hHandle, dwAlgoNum)**, который настраивает библиотеку для работы с текущим контекстом Guardant API. Макрос осуществляет привязку библиотеки к используемому контексту Guardant API и открытому ключу, также ему передается номер аппаратного алгоритма, в который был загружен отладочный модуль.
5. Если код, предполагаемый для размещения в электронном ключе, использует вызовы внешнего Guardant API, то их требуется заменить на соответствующие вызовы Guardant Code API. В данном случае функции импортируются из отладочной библиотеки. Если же код содержит функции Guardant API, не имеющие прямых аналогов в Guardant Code API, требуется создать эквивалентные им конструкции из доступных функций.

Следует принимать во внимание, что ни сама отладочная библиотека, ни отладочный модуль не содержат логики работы функций. Они являются всего лишь своеобразным «туннелем», через который параметры вызова функций передаются в электронный ключ и возвращаются обратно.

Пример использования макроса **DEBUGDLL_INIT**:

```
main()
{
// Хэндл ключа, в котором запускается загруженный
                        пользователем код:
HADNLE hGrd;

// Инициализация API и подключение к электронному ключу
...

#ifdef DEBUG
// Инициализация DLL
DEBUGDLL_INIT(hGrd, 1);
// hGrd - хендл открытого электронного ключа.
// 1 - номер аппаратного алгоритма, в котором находится
//     загруженный отладочный модуль
#endif

// Передача параметра hGrd необязательна.
GcaGetRandom(0, &iodata[i]);

return 0;
}
```

Также стоит отметить, что функции **GcaExit()** и **GcaLedOn()/ GcaLedOff()** не могут работать в отладочном режиме. Первая – из-за того, что результат ее работы просто нельзя зафиксировать, а функции управления светодиодом – из-за того, что сразу после их вызова работа кода будет завершаться, при этом индикатор просто зажигается вновь.

Использование отладочной библиотеки демонстрируется в примере №9 (см. **Краткая характеристика примеров**).

Загрузка кода в электронный ключ

Для загрузки кода в электронный ключ первоначально используется GrdUtil. При помощи этой утилиты создается дескриптор аппаратного алгоритма типа **Загружаемый код**.

В свойствах алгоритма указывается бинарный файл, который содержит скомпилированный загружаемый код. Этому файлу должен сопутствовать файл bmap, содержащий настройки адресов памяти.

Бинарный файл перед загрузкой должен быть преобразован в файл типа GCEXE (Guardant Code executable). Преобразование осуществляется в автоматическом режиме утилитой программирования ключей GrdUtil.

При выполнении преобразования GrdUtil генерирует ключевые пары:

- **Для зашифрования и расшифрования загружаемого кода**
  Зашифрование производится на открытом ключе, который хранится в маске и не записывается в электронный ключ.
  Расшифрование – на закрытом ключе, который храниться и в файле маски, и в дескрипторе алгоритма, записанного в электронный ключ
- **Для электронной цифровой подписи загружаемого кода**
  Подписывание производится – на закрытом ключе, который хранится только в маске и не записывается в сам ключ. Проверка – на открытом, который будет храниться и в маске, и в дескрипторе алгоритма, который будет записан в ключ

Перед загрузкой бинарный файл зашифровывается на сеансовом ключе и подписывается ЭЦП. Это гарантирует возможность загрузки кода только разработчиком. При необходимости файл GCEXE можно сгенерировать таким образом, чтобы он мог быть загружен только в ключ с указанным ID. Эта возможность полезна для создания адресных обновлений, например – платных.

При записи данных в ключ первоначально записывается дескриптор алгоритма, а уже затем – файл GCEXE.

Однажды сгенерированный файл GCEXE может быть в дальнейшем записан и в другие ключи, содержащие соответствующие ключи шифрования и подписи. Для этого используется функция **GrdCodeLoad()**.

Отладка защищенного приложения

Отладке приложений, использующих загружаемый код, следует уделить особое внимание, поскольку поиск ошибок при работе с «черным ящиком» является непростым делом.

Очень важным является итоговое быстродействие загруженного кода. Если оно получается неудовлетворительным, требуется принять меры по приведению кода к обозначенным в начале этой главы требованиям.

Дистанционное обновление загружаемого кода

Проблема обновления информации в ключах, уже находящихся у пользователей приложения, актуальна и для загружаемого кода. Рано или поздно в этот код может потребоваться внести изменения или исправления.

Для успешного обновления загружаемого кода необходимо выполнение следующих условий:

- У разработчика должна храниться прошивка (файл маски), содержащая ключевые пары для шифрования и подписи загружаемого кода,
- У конечного пользователя должен находиться электронный ключ Guardant Code, содержащий дескриптор алгоритма с загружаемым кодом, а также закрытый ключ для расшифрования кода и открытый ключ для проверки ЭЦП
- Ключевые пары в маске и ключе должны быть идентичны.

Для обновления загружаемого кода необходимо сгенерировать новый GCEXE-файл с обновленным кодом, зашифрованным и подписанным на соответствующих ключах.

Само обновление может производиться как при помощи технологии TRU, так и прямой загрузкой GCEXE-файла из защищаемого приложения функцией GrdCodeLoad().

При желании можно сделать процедуру обновления загружаемого кода «прозрачной» для пользователя. Тогда от него потребуется только получить обновление, поместить его рядом с исполняемым файлом приложения (или в специально для этого предназначенную директорию) и запустить приложение.

Примеры использования загружаемого кода

Каждый пример находится в директории установки комплекта разработчика Guardant (по умолчанию **%Program Files%\ Guardant\Guardant 5\%Общий код%\Samples\ARM**), в отдельной папке вида **XX - SampleName**, и состоит из двух частей:

- *Проект Visual Studio 2005 .NET* в папке **Win32**, демонстрирующий использование Guardant API (т. е. необходимую инициализацию и запуск загруженного кода)
- *Пример загружаемого кода* в папке **Loadable Code**, содержащий проект GCC-ARM в виде исходного кода на C и **makefile**. При сборке примера получаются ***.bin** и ***.bmap** файлы, которые необходимо указывать GrdUtil для генерации GCEXE-файла соответствующего алгоритма.

Структура используемого файла маски

Файл маски (**Mask.nsd**), используемый в примерах для ключей Guardant Code / Code Time, отличается от файла маски GrdUtil по умолчанию (**default.nsd**) и имеет следующую структуру, одинаковую для всех примеров:

1. Алгоритм #0 AES128
2. Алгоритм #01 SHA256
3. Алгоритм #02 AES128, зависит от ID ключа
4. Защищенная ячейка #03 – только для чтения
5. Защищенная ячейка #04 – для чтения и записи
6. Алгоритм #05 AES128 Demo
7. Алгоритм #06 SHA256 Demo
8. Защищенная ячейка #07 – Таблица лицензий
9. Алгоритм #08 – ECC160
10. Алгоритм #09 Загружаемый код № Занимает 4 сектора. В него по очереди загружаются все примеры

По умолчанию файл маски **Mask.nsd** находится в директории **%Program Files%\ Guardant\Guardant 5\%Общий код%\ Samples\ARM**.

Важная информация

Для корректной работы примеров Guardant Code необходимо записать **Mask.nsd** в ключ при помощи GrdUtil.

Краткая характеристика примеров

Важная информация

Для корректной работы примеров Guardant Code необходимо записать **Mask.nsd** в ключ при помощи GrdUtil.

0. Шаблон проекта загружаемого кода. На его основе можно разрабатывать собственные модули.
1. Функция main(), возвращающая определенный код (демонстрация передачи параметров dwRet и dwP1).
2. Демонстрация использования буферов ввода-вывода.
3. Демонстрация работы необнуляемого буфера (внутренняя память). В одном вызове можно записать данные, а в следующем – считать обратно.
4. Разделение загружаемого приложения на несколько логических частей (обработка команд).
5. Управление индикатором. Демонстрация работы функций GcaLedOn()/GcaLedOff().
6. Создание интерфейса к EEPROM. Обеспечивает чтение/запись участка в 512 байт EEPROM блоками по 16 байт. Блоки шифруются на AES-128 в режиме ECB, на ключе, хранящемся внутри загружаемого приложения.
7. Демонстрация работы с защищенными ячейками. Проход по аппаратным алгоритмам и получение информации о них (сервисы времени, чтения, модификации, счетчик запусков).
8. Модификация содержимого заданной защищенной ячейки.
9. Демонстрация функций работы с таймером (только для Guardant Code Time). Получение текущего времени и валидности RTC. Функция установки времени отсутствует. Единственный способ установить время в электронном ключе – при помощи GrdUtil.
10. Получение времени жизни защищенной ячейки.
11. Функции криптографии. Демонстрация генерации ключевой пары, подписи и проверки этой подписи по ECC160 с возвратом всех буферов в PC.
12. Использование аппаратного алгоритма для подписи.
13. Подсчет хэша.
14. Возврат блока случайных данных.
15. Отличие форматов хранения чисел с плавающей точкой в GCC ARM и PC.
16. Демонстрация использования встроенных математических функций.

17. Пример реализации вычислительной задачи (вычисление заданной точки на кривой Безье порядка N, параметры кривой хранятся внутри загружаемой микропрограммы).
18. Реализация целочисленной задачи. Шифрование на алгоритме Blowfish для блока данных на заданном ключе.
19. Демонстрация использования отладочной библиотеки.
    Для запуска примера необходимо запустить проект **Demo.sln**

    Пример демонстрирует, как можно использовать отладочную DLL в проектах. В подкаталоге **LoadableCode** содержится проект загружаемого модуля, который собирается в виде самостоятельного модуля, так и в составе проекта **Demo.sln**.

    По умолчанию в проекте задано использование отладочной библиотеки, при этом код из файла **lc.c** собирается в составе VS-проекта. Если убрать опцию **DEBUG_DLL** и прошить в ключ маску из подпапки **LoadableCode**, код из файла **lc.c** будет выполняться уже внутри ключа. Как видно, в **lc.c** допустимо использование вызовов Guardant Code API.

    Однако в случае исполнения в составе VS-проекта налагается ограничение на функцию GcaExit(), и она не вызовет выход из кода в **lc.c**.

Характеристики и вычислительные возможности Guardant Code

1. Быстродействие 120 MIPS 32-битных операций
2. 20 кб ОЗУ для программы пользователя
3. Около 128 кб (с возможностью расширения до 352 кб) для программы пользователя (около 20000 строк на Си для варианта 128кб и около 60000 строк на Си для варианта 384 кб)
4. Приблизительная скорость операций с плавающей точкой:

Приложение А
Краткий обзор Guardant API

Примечание

Детальная информация по Guardant API содержится в Справочной системе по Guardant API (файл **GrdAPI.chm**, по умолчанию расположен в директории **%Program Files%\Guardant\Guardant 5\%Общий код%\Doc**)

Интерфейс прикладного программирования Guardant представляет собой набор функций для работы с ключом, собранных в библиотечных файлах. Также в состав ПО Guardant входят исходные тексты примеров использования Guardant API для основных языков программирования и сред разработки.

Защита с использованием функций API заключается во встраивании в исходный код приложения вызовов к ключу согласно предварительно разработанной схеме (см. **рекомендации по эффективному применению Guardant API**).

Основные достоинства защиты при помощи API – высокий уровень стойкости защищенного приложения к взлому при правильном использовании функций API; возможность создания эффективной и нетривиальной защиты любой категории сложности; широкие возможности для использования аппаратных алгоритмов ключа – основного инструмента, с помощью которого усложняется логика работы защищенного приложения.

Для достижения большей защищенности рекомендуется комбинировать автозащиту и защиту при помощи Guardant API.

Особенности Guardant API

Псевдокод

Псевдокод - это технология защиты исполняемых файлов от изучения логики их работы. Суть ее состоит в том, что определенные фрагменты исполняемых файлов дизассемблируются, анализируются и преобразуются в защищенный код виртуальной машины, которая генерируется тут же.

Анализировать логику работы защищенного подобным образом кода на порядок сложнее, чем инструкции Intel-совместимых процессоров, поскольку для псевдокода не существует никакого стандартного инструментария (отладчиков, дизассемблеров). Взломщику придется все делать вручную или создавать свои собственные инструменты.

В каждой копии виртуальной машины реализуется уникальным образом:

- Набор внутренних команд псевдокода
- Множественный взаимный контроль целостности (для затруднения модификации и установки точек останова)
- Обфускация кода виртуальной машины (замусоривание реального кода вторичным)
- Преобразование кода самой виртуальной машины
- Преобразование самого псевдокода
- Параметры многих команд рассчитываются только во время выполнения (защита от статического дизассемблирования)
- Отсутствие постоянных сигнатур в защищенном коде (затруднение поиска псевдокода в защищенном приложении)

Handle-ориентированность

Для работы с ключом создается контекст, который сохраняется в специальном защищенном контейнере. Доступ к контейнеру осуществляется через хэндл. Хэндл является идентификатором контекста, с которым и оперируют функции Guardant API.

Это позволяет упростить обмен с API за счет минимизации числа вызовов и параметров функций, а также сократить время работы функций.

Поддержка многопоточности

В защищенном контейнере сохраняются параметры поиска и прочие внутренние переменные. Это существенно расширяет возможности разработчиков приложений, поскольку наличие контекста позволяет работать с несколькими ключами из нескольких потоков одновременно.

Это дает возможность:

- Существенно повысить защищенность, поскольку отлаживать и анализировать многопоточные приложения гораздо сложнее
- Легче интегрировать API в сложные многопоточные приложения
- Работать с несколькими ключами одновременно

Защищенность, контроль целостности и аутентификация

Содержимое контейнера закодировано. Осуществляется контроль целостности данных, хранящихся в контейнере. Коды доступа тоже хранятся в контейнере в закодированном виде и их не нужно указывать при каждом вызове функций Guardant API. В защищенном

контейнере хранятся только те коды доступа, которые действительно используются.

Guardant API контролирует целостность своего кода во избежание вмешательств извне. Код Guardant API защищен от статического и динамического анализа при помощи псевдокода.

Драйвер Guardant подписан при помощи асимметричной криптографии. Защищенное приложение проверяет подпись драйвера (автоматически при вызове функций Guardant API 5.x). Взломщик не сможет создать драйвер-эмулятор электронного ключа, чтобы у него сходилась подпись, поскольку закрытый ключ ему не известен.

Так же как и код Guardant API, драйвер защищен от анализа с использованием технологии псевдокода. Это подразумевает, в том числе, и контроль целостности кода драйвера.

Унификация

Для унификации и упрощения написания кода все функции Guardant API сделаны универсальными для работы, как с локальными, так и с сетевыми ключами. При инициализации библиотеки необходимо указать, с какими ключами будет работать приложение: локальными, сетевыми, или с обоими типами.

Это существенно упрощает процесс программирования систем защиты, рассчитанных на работу и с сетевыми и с локальными ключами, поскольку код защиты можно написать единый. Также максимально упрощается переход на сетевые ключи: по сути надо будет поменять всего один флаг в функции **GrdSetFindMode()**.

Список функций Guardant API

Функции Guardant API общего назначения

Функции общего назначения позволяют осуществлять сервисные операции, связанные с инициализацией, настройкой и деинициализацией Guardant API, а также выполнять основные операции с электронными ключами:

- Искать электронный ключ, удовлетворяющий заданным условиям поиска
- Инициализировать память ключа
- Читать данные из памяти ключа
- Записывать данные в память ключа
- Устанавливать запреты на чтение/запись памяти ключа
- Преобразовывать информацию при помощи аппаратных алгоритмов ключа

Перечень функций Guardant API общего назначения:

Название функции	Код доступа	Краткое описание
GrdStartup	Не нужен	Инициализировать Guardant API
GrdStartupEx	Не нужен	Инициализировать GrdAPI + задать путь к GnClient.ini
GrdCleanup	Не нужен	Деинициализировать Guardant API
GrdDllMain	Не нужен	Инициализировать сетевые функции для использования в DLL
GrdGetLastError	Не нужен	Получить информацию о последней ошибке из защищенного контейнера
GrdFormatMessage	Не нужен	Преобразовать код ошибки в текстовое сообщение
GrdGetInfo	Не нужен	Получить информацию из защищенного контейнера
GrdIsValidHandle	Не нужен	Проверить валидность хэндла защищенного контейнера
GrdCreateHandle	Не нужен	Создать хэндл защищенного контейнера
GrdCloseHandle	Не нужен	Закрыть хэндл защищенного контейнера
GrdSetAccessCodes	Public и Read обязательно, остальные по необходимости	Поместить коды доступа в защищенный контейнер
GrdSetWorkMode	Не нужен	Установить режимы работы Guardant API
GrdSetFindMode	Не нужен	Установить режимы поиска электронных ключей
GrdFind	Private Read	Найти электронный ключ, удовлетворяющий критериям поиска
GrdLogin	Private Read	Зарегистрироваться на электронном ключе
GrdLogout	Private Read	Снять регистрацию с электронного ключа
GrdLock	Private Read	Заблокировать ключ на время выполнения операций, требующих монопольного режима работы с ключом
GrdUnlock	Private Read	Разблокировать электронный ключ
GrdCheck	Private Read	Проверить наличие электронного ключа
GrdDecGP	Private Read+ PrivateWrite	Декрементировать счетчик GP
GrdRead	Private Read	Прочитать данные из памяти электронного ключа
GrdWrite	Private Write	Записать данные в память электронного ключа
GrdSign	Private Read	Вычислить ЭЦП массива данных
GrdVerifySign	Не нужен	Проверить ЭЦП массива данных
GrdSeek	Не нужен	Переместить указатель на текущий адрес памяти электронного ключа
GrdInit	Private master	Инициализировать память электронного ключа
GrdProtect	Private master	Установить аппаратные запреты чтения/записи и записать количество аппаратных алгоритмов и защищенных ячеек
GrdTransform	Private Read	Преобразовать данные аппаратно- или программно-реализованным алгоритмом
GrdTransformEx	Private Read	Преобразовать данные аппаратно- или программно-реализованным алгоритмом

Название функции	Код доступа	Краткое описание
GrdCrypt	Private Read	Закодировать/раскодировать данные аппаратно или программно-реализованным алгоритмом
GrdCryptEx	Private Read	Закодировать/раскодировать данные аппаратно или программно-реализованным алгоритмом
GrdHash	Private Read	Вычислить значение хэш-функции программно или аппаратно-реализованным алгоритмом
GrdHashEx	Private Read	Расширенный вариант **GrdHash**
GrdCodeInit	PrivateRead	Инициализировать пароль быстрого взаимообратного преобразования
GrdEnCode	не нужен	Закодировать данные быстрым взаимообратным преобразованием
GrdDeCode	не нужен	Раскодировать данные быстрым взаимообратным преобразованием
GrdCRC	не нужен	Вычислить CRC

Функции Guardant API для работы с защищенными ячейками

Для работы с защищенными ячейками существуют специальные функции Guardant API:

Название функции	Код доступа	Краткое описание
GrdPI_Activate	Private Read и пароль на данную операцию, если он задан в ключе	Активировать алгоритм / защищенную ячейку
GrdPI_Deactivate		Деактивировать аппаратный алгоритм или защищенную ячейку
GrdPI_Read		Прочитать данные из защищенной ячейки
GrdPI_Update		Обновить данные в защищенной ячейке
GrdPI_GetCounter		Получить значение счетчика запусков алгоритма

Функции Guardant API для работы с ключами с RTC

Для работы с ключами, поддерживающими технологию Time, существуют специальные функции Guardant API:

Название функции	Код доступа	Краткое описание
GrdSetTime	Private Master	Установить время микросхемы таймера ключа
GrdGetTime	Private Read	Получить время таймера ключа
GrdPI_GetTimeLimit	Private Read	Получить оставшееся время работы алгоритма
GrdMakeSystemTime	Не нужен	Заполнить поля структуры TgrdSystemTime
GrdSplitSystemTime	Не нужен	Получить значение полей структуры TgrdSystemTime
GrdTRU_ GenerateQuestionTime	Private Read	Сгенерировать зашифрованное число-вопрос
GrdTRU_ GenerateQuestionTimeEx	Private Read	Расширенная версия **GrdTRU_GenerateQuestionTime**
GrdTRU_ DecryptQuestionTime	Private Read	Расшифровать и проверить подлинность числа-вопроса
GrdTRU_ DecryptQuestionTimeEx	Private Read	Расширенная версия **GrdTRU_ DecryptQuestionTime**

Функции Guardant API для работы с Trusted Remote Update

Технология удаленного обновления Trusted Remote Update может быть реализована не только при помощи утилит, входящих в Комплект разработчика. При желании разработчики могут встраивать поддержку этой технологии непосредственно в свои приложения, используя набор предназначенных для этой цели функций:

Название функции	Код доступа	Краткое описание
GrdTRU_SetKey	Private Master	Инициализировать память электронного ключа и записать секретный ключ TRU
GrdTRU_GenerateQuestion	Private Read	Сгенерировать число-вопрос
GrdTRU_GenerateQuestionEx	Private Read	Расширенная версия **GrdTRU_GenerateQuestion**
GrdTRU_DecryptQuestion	Private Read	Декодировать и проверить подлинность числа-вопроса
GrdTRU_DecryptQuestionEx	Private Read	Расширенная версия **GrdTRU_DecryptQuestion**
GrdTRU_SetAnswerProperties	Не нужен	Установить параметры числа-ответа
GrdTRU_EncryptAnswer	Private Read	Сгенерировать и кодировать число-ответ
GrdTRU_EncryptAnswerEx	Private Read	Расширенная версия **GrdTRU_EncryptAnswer**
GrdTRU_ApplyAnswer	Private Read	Записать число-ответ в ключ Guardant

Функции Guardant API для работы с ключами Guardant Code

См. **Сервисные функции для работы с загружаемым кодом**.

Внутреннее Guardant API Code для загружаемого кода

См. **Guardant Code API**.

Функции Guardant API для управления драйверами

Для того чтобы инсталлировать, деинсталлировать и настраивать драйверы Guardant из собственных приложений можно использовать функции управления драйверами, которые реализованы в виде динамически загружаемой библиотеки.

Название функции	Краткое описание
GrdDrvInstall	Инсталлировать драйверы Guardant
GrdDrvUnInstall	Деинсталлировать драйверы Guardant
GrdDrvIsInstalled	Проверить наличие драйверов Guardant в системе
GrdDrvGetVersion	Получить версию драйверов Guardant, установленных в системе
GrdDrvSetPortUseState	Установить режимы использования порта
GrdDrvGetPortUseState	Получить текущие установленные режимы использования порта

Название функции	Краткое описание
GrdDrvSetPrnPortTimeOut	Установить значение таймаута для LPT порта
GrdDrvGetPrnPortTimeOut	Получить текущее установленное значение таймаута для LPT порта
GrdDrvSetPortProperties	Установить параметры работы с LPT портом
GrdDrvGetPortProperties	Получить текущие установленные параметры работы с LPT портом
GrdDrvSetPortPropertiesAuto	Установить параметры работы с LPT портом по умолчанию
GrdDrvGetSystemInfo	Получить системную информацию о конфигурации LPT и USB-портов

Местонахождение файлов API

Категории файлов	Каталог с API
Заголовочные файлы	Guardant\Guardant 5\%PublicCode%\Include
Библиотеки и объектные модули	Guardant\Guardant 5\%PublicCode%\Lib
Примеры работы с Guardant API	Guardant\Guardant 5\%PublicCode%\Samples\

Коды ошибок Guardant API

Функции Guardant API, предназначенные для работы с электронными ключами, могут возвращать следующие коды ошибок:

Название ошибки	Код	Краткое описание
GrdE_Ok	0	Операция выполнена успешно
GrdE_DongleNotFound	1	Не найден ключ, отвечающий заданным условиям поиска
GrdE_AddressTooBig	3	Указанный адрес слишком велик
GrdE_Gpis0	5	Счетчик запусков GP исчерпан (значение равно нулю)
GrdE_InvalidCommand	6	Неверная команда обращения к ключу
GrdE_VerifyError	8	Ошибка верификации записи в память ключа
GrdE_NetProtocolNotFound	9	Сетевой протокол не найден
GrdE_NetResourceExhaust	10	Сетевой ресурс ключа Guardant Net исчерпан
GrdE_NetConnectionLost	11	Потеряно соединение с сервером Guardant Net
GrdE_NetDongleNotFound	12	Сервер Guardant Net не найден
GrdE_NetServerMemory	13	Ошибка распределения памяти сервера Guardant Net
GrdE_DPMI	14	Ошибка DPMI
GrdE_Internal	15	Внутренняя ошибка сервера Guardant Net
GrdE_NetServerReloaded	16	Сервер Guardant Net был перезагружен
GrdE_VersionTooOld	17	Данная команда не поддерживается данной версией ключа (ключ старой версии)
GrdE_BadDriver	18	Необходим драйвер Windows NT
GrdE_NetProtocol	19	Ошибка сетевого протокола
GrdE_NetPacket	20	Получен сетевой пакет недопустимого формата
GrdE_NeedLogin	21	Требуется регистрация на сервере Guardant Net

Название ошибки	Код	Краткое описание
GrdE_NeedLogout	22	Необходимо снять регистрацию на сервере Guardant
GrdE_DongleLocked	23	Ключ Guardant Net занят другим приложением
GrdE_DriverBusy	24	Драйвер не может захватить порт
GrdE_CRCError	30	Ошибка CRC при обращении к ключу
GrdE_CRCErrorRead	31	Ошибка CRC при чтении данных из ключа
GrdE_CRCErrorWrite	32	Ошибка CRC при записи данных в ключ
GrdE_Overbound	33	Выход за границу памяти ключа
GrdE_AlgoNotFound	34	Аппаратный алгоритм с таким номером в ключе не найден
GrdE_CRCErrorFunc	35	Ошибка CRC аппаратного алгоритма
GrdE_AllDonglesFound	36	Все ключи перебраны
GrdE_ProtocolNotSup	37	Слишком старая версия Guardant API
GrdE_InvalidCnvType	38	Задан несуществующий метод взаимообратного преобразования
GrdE_UnknownError	39	Неизвестная ошибка при работе с алгоритмом/ячейкой, операция могла не завершиться
GrdE_AccessDenied	40	Неверный пароль доступа к защищенной ячейке
GrdE_StatusUnchangeable	41	Статус защищенной ячейки изменить нельзя
GrdE_NoService	42	Для алгоритма/ячейки сервис не предусмотрен
GrdE_InactiveItem	43	Алгоритм/ячейка находятся в состоянии Inactive, команда не выполнена
GrdE_DongleServerTooOld	44	Попытка выполнить операцию, которую не поддерживает эта версия сервера Guardant Net
GrdE_DongleBusy	45	В данный момент ключ не может выполнять никаких операций
GrdE_InvalidArg	46	Задано недопустимое значение одного из аргументов функции
GrdE_MemoryAllocation	47	Ошибка распределения памяти
GrdE_InvalidHandle	48	Недопустимый хендл
GrdE_ContainerInUse	49	Этот защищенный контейнер уже используется
GrdE_Reserved50	50	Зарезервировано
GrdE_Reserved51	51	Зарезервировано
GrdE_Reserved52	52	Зарезервировано
GrdE_SystemDataCorrupted	53	Нарушена целостность системных данных
GrdE_NoQuestion	54	Вопрос для удаленного обновления не был сгенерирован
GrdE_InvalidData	55	Недопустимый формат данных для удаленного обновления
GrdE_QuestionOK	56	Вопрос для удаленного обновления уже сгенерирован
GrdE_UpdateNotComplete	57	Процедура записи при удаленном обновлении не завершена

Название ошибки	Код	Краткое описание
GrdE_InvalidHash	58	Неверное знач. хэша данных удаленного обновления
GrdE_GenInternal	59	Внутренняя ошибка
GrdE_AlreadyInitialized	60	Эта копия Guardant API уже инициализирована
GrdE_LastError	61	Неизвестная ошибка
GrdE_DuplicateNames	63	Числовое имя аппаратного алгоритма / защищенной ячейки уже существует
GrdE_AATFormatError	64	Несуществующий адрес в AAT-таблице
GrdE_SessionKeyNtGen	65	Сессионный ключ не создан
GrdE_InvalidPublicKey	66	Недействительный открытый ключ
GrdE_InvalidDigitalSign	67	Недействительная ЭЦП
GrdE_SessionKeyGenError	68	Ошибка при создании сессионного ключа
GrdE_InvalidSessionKey	69	Недействительный сессионный ключ
GrdE_SessionKeyTooOld	70	Просроченный сессионный ключ
GrdE_NeedInitialization	71	Необходима инициализация
GrdE_gcProhibitCode	72	Ошибка при проверке загружаемого кода. Адрес точки входа некорректен или обнаружены запрещенные команды или обращение к недопустимым адресам
GrdE_gcLoadableCodeTimeOut	73	Тайм-аут при выполнении загружаемого кода
GrdE_gcFlashSizeFrom DescriptorTooSmall	74	В дескрипторе загружаемого кода указан недостаточный размер Flash-памяти
GrdE_Reserved75	75	Зарезервировано
GrdE_Reserved76	76	Зарезервировано
GrdE_Reserved77	77	Зарезервировано
GrdE_Reserved78	78	Зарезервировано
GrdE_Reserved79	79	Зарезервировано
GrdE_gcIncorrectMask	80	В дескрипторе загружаемого кода указан недостаточный размер памяти для структуры TGrdLoadableCodeData
GrdE_gcRamOverboundInProtect	81	В дескрипторе загружаемого кода указан неверный размер RAM
GrdE_gcFlashOverboundInProtect	82	В дескрипторе загружаемого кода указан неверный размер Flash-памяти
GrdE_gcIntersection OfCodeAreasInProtect	83	Обнаружено пересечение областей Flash-памяти, заданных в нескольких дескрипторах
GrdE_gcBmapFileTooBig	84	Слишком большой размер BMAP-файла
GrdE_gcZeroLengthProgram	85	Загружаемый код имеет нулевую длину
GrdE_gcDataCorrupt	86	Ошибка при контроле целостности данных
GrdE_gcProtocolError	87	Ошибка в протоколе Guardant Code
GrdE_gcGCEXENotFound	88	Нет загруженной программы пользователя
GrdE_gcNotEnoughRAM	89	Объявленный в программе пользователя RAM-буфер ввода-вывода недостаточен для передачи\приема данных

Название ошибки	Код	Краткое описание
GrdE_gcException	90	При выполнении загружаемого кода произошло нарушение защиты виртуальной среды
GrdE_gcRamOverboundInCodeLoad	91	Буфер ввода-вывода, заданный в загружаемом коде, выходит за допустимую область RAM
GrdE_gcFlashOverboundInCodeLoad	92	Загружаемый код выходит за пределы допустимой области Flash-памяти
GrdE_gcIntersection OfCodeAreasInCodeLoad	93	Адресное пространство загружаемого кода пересекается с уже загруженным. Необходима инициализация памяти
GrdE_gcGCEXEFormatError	94	Неверный формат GCEXE файла
GrdE_gcRamAccessViolation	95	Заданы пересекающиеся области ОЗУ для запускающего и запускаемого кода
GrdE_gcCallDepthOverflow	96	Превышен уровень вложенности вызовов GcaCodeRun (<1)
GrdE_LastError	97	Общее число кодов возврата

Специфика структуры программ, использующих Guardant API

Использование Guardant API требует соблюдения определенных условий, касающихся структуры программ. Это связано с тем, что для работы Guardant API требуется выполнение обязательных процедур, которые инициализируют API перед его использованием и деинициализируют после. Все функции, работающие непосредственно с электронным ключом, требуют, чтобы была выполнена регистрация на конкретном локальном или сетевом ключе.

Общая структура приложения, использующего GrdAPI, имеет вид:

Блок инициализации	**GrdStartup( )**	Инициализировать данную копию Guardant API
	GrdCreateHandle( )	Создать хэндл. Хэндлов может быть создано несколько, в том числе и для одного и того же ключа.
	GrdSetAccessCodes( )	Установить коды доступа в защищенный контейнер. Устанавливать следует только те коды, которые будут реально использоваться. Неиспользуемые коды можно заменить случайными числами.
	GrdSetFindMode( )	Установить режимы и критерии поиска ключа
	GrdFind( )	Выполнить поиск электронного ключа по заданным критериям. Следует учитывать, что критериям может соответствовать не один ключ, поэтому следует выполнять поиск до тех пор, пока не будут обнаружены все ключи, соответствующие критериям.
	GrdLogin( )	Из всех обнаруженных ключей следует выбрать один и выполнить процедуру регистрации. Без регистрации остальные функции не смогут обращаться к ключу.

Основной блок	Основная работа с электронным ключом. В этом блоке можно вызывать функции работы с электронным ключом: **GrdLock, GrdUnlock, GrdCheck, GrdDecGP, GrdRead, GrdWrite, GrdSeek, GrdInit, GrdProtect, GrdTransform, GrdCrypt, GrdHash, GrdPI_Activate, GrdPI_Deactivate, GrdPI_Read, GrdPI_Update** – и др.	
Блок деинициализации	**GrdLogout( )**	Снять регистрацию с текущего ключа.
	GrdCloseHandle( )	Закрыть хэндл
	GrdCleanup( )	Деинициализировать данную копию Guardant API

Использование Guardant API в DLL

При использовании Guardant API в динамических библиотеках к блоку инициализации в обязательном порядке до вызова **GrdStartup( )** добавляется вызов функции **GrdDllMain( )**.

Подробное описание функций Guardant API см. в справочном файле GrdAPI.chm, который находятся в директории:

%WinDir%\Program Files\Guardant\Guardant 5xx\%Public Code%\Doc

Защита Guardant для ОС GNU/Linux

Современные ключи Guardant, начиная с Guardant Sign, позволяют защищать приложения, запускаемые в ОС GNU/Linux на аппаратных платформах i386 и x86_64. Для этого в комплект разработчика включены статические библиотеки (ELF) **libgrdapi.a** соответствующей разрядности, которые реализуют Guardant API.

Кроме того, поддерживается запуск защищенных Windows-приложений с использованием Wine (www.winehq.org) - свободной реализации Windows API. Для этого в комплект разработчика включен проект динамической библиотеки для Wine — **GrdWine**, распространяемый под свободной лицензией GNU Lesser General Public License version 2.1 (поставляется в виде пакета с исходными кодами – **grdwine-0.5.4.tar.gz**).

Ключи работают в ОС GNU/Linux (в том числе, в **HID-режиме**) без установки дополнительных драйверов и демонов, требуя лишь обеспечить имя и разрешение доступа к файлу устройства. Для обращения к ключу используются соответственно Linux USB Device Filesystem или Linux USB HID Device Interface (в случае HID-режима). В комплект разработчика включены наборы правил для систем регистрации устройств и описание требований для случаев нетипичного конфигурирования (см. ниже раздел **Имена и доступ к устройствам**).

Подготовка к работе с ключами Guardant

См. раздел **Установка ключа в среде Linux** в первой части Руководства пользователя.

Защита Native-приложений GNU/Linux

Для сборки защищаемого приложения, необходимо скомпоновать (слинковать) защищаемое приложение со статической библиотекой Guardant API.

Рекомендуется использовать компилятор GCC 4-ой версии, однако, возможно использовать и более ранние версии GCC, и другие компиляторы, например, Intel C++ Compiler (ICC).

Для компиляции с библиотекой Guardant API необходимо выполнить следующее (на примере файла с исходным текстом программы — foobar.c):

```
$ gcc [-I<путь_к_заголовочному_файлу_GrdAPI.h>]
-c foobar.c -o foobar.o
$ gcc [-L<путь_к_библиотеке_libgrdapi.a>] foobar.o
-o foobar -lpthread -lgrdapi
```

или

```
$ gcc [-I<путь_к_заголовочному_файлу_GrdAPI.h>]
[-L<путь_к_библиотеке_libgrdapi.a>] foobar.c -o foobar
-lgrdapi –lpthread
```

Обратите внимание, что библиотека Guardant API использует библиотеку **pthread** - POSIX Threads, поэтому для компоновки приложений необходимо использовать соответствующую библиотеку.

Подсоедините ключ Guardant к USB-порту компьютера, защищенное приложение готово к работе.

Запуск защищенных Windows-приложений под Wine

Библиотека **grdwine.dll.so**, реализующая работу с ключами Guardant для защищенных Windows-приложений под Wine, поставляется в виде исходных кодов (см. **grdwine-0.5.4.tar.gz)**. Это, в принципе, позволяет ее использовать с любой версией Wine, достаточно просто собрать библиотеку из исходных кодов.

Важная информация

Рекомендуемая к использованию версия Wine — 1.x.x. Корректная работа с более ранними версиями Wine не гарантируется. Загрузить последнюю версию Wine можно по адресу: http://www.winehq.org/site/download

Компиляция библиотеки

```
$ tar xvf grdwine-0.5.4.tar.gz
$ cd grdwine-0.5.4
$ ./configure --with-wineincs=/usr/include/wine
--with-winedlls=/usr/lib/wine
$ make
# make install
```

Важная информация

Указанные в примере пути к заголовочным файлам и библиотекам Wine (**/usr/ include/wine** и **/usr/lib/wine**) могут меняться в зависимости от версии Wine, используемого дистрибутива или заданного префикса для установки (в случае, если Wine устанавливался из исходных кодов)

Подсоедините ключ Guardant к USB-порту компьютера, защищенное приложение готово к работе.

Удаление библиотеки из системы

```
$ cd grdwine-0.5.4
# make uninstall
```

Имена и доступ к устройствам

Для ключей, работающих в драйверном режиме

Обращение к ключу происходит через Linux USB Device Filesystem.

Подробную информацию см. в **linux/Documentation/usb/proc_usb_info.txt**.

Для успешной работы с ключом в системе нужно разрешить доступ на чтение/запись к файлу устройства.

Для ключей, работающих в HID-режиме

Обращение к ключу происходит через Linux USB HID Device Interface (драйвер **usbhid**). Подробную информацию см. в **linux/ Documentation/usb/hiddev.txt**.

Для успешной работы с ключом в системе нужно изменить имена соответствующих устройств на **/dev/grdhid[№]** и разрешить доступ на чтение/запись к файлу устройства.

Пример для hotplug или hotplug-ng

```
# cp etc/grdnt.usermap /etc/hotplug/usb/grdnt.usermap
# cp etc/grdnt /etc/hotplug/usb/grdnt
```

Пример для udev

Для ключей в драйверном режиме, и в случае использования файлов-устройств USB Device Filesystem

```
# cp etc/grdnt.udev /etc/udev/rules.d/95-grdnt.rules
```

Для ключей в HID-режиме

```
# cp etc/grdnt_hid.udev /etc/udev/rules.d/95-grdnt_hid.rules
```

Переменные окружения

Для настройки Guardant API под GNU/Linux следует пользоваться следующими переменными окружения:

GRD_IPC_NAME	директория, в которой процессы будут создавать/открывать для чтения и записи файлы, используемые для синхронизации доступа к ключу. Если переменная не задана, используется значение по умолчанию (**/tmp)**
USB_DEVFS_PATH	директория **Linux USB Device Filesystem** (точка монтирования или директория, содержащая дерево соответствующих устройств). Если переменная не задана, будет использоваться **/dev/bus/usb** (если существует), иначе – **/proc/bus/usb**

Приложение В Приемы повышения стойкости защиты

Электронные ключи Guardant – высокоэффективное средство программно-аппаратной защиты ПО, которое позволяет создать защиту практически любого уровня сложности и стойкости к взлому.

Автоматическая защита

Native-приложения (Win32)

Для защиты готовых приложений от изучения логики их работы пользуйтесь утилитой автоматической защиты. При этом целесообразно использовать следующие опции:

Защита от вирусов. Эта опция позволяет защитить приложение не только от файловых вирусов, но и от нелегальной модификации файла приложения (дописывания каких-то программных модулей, изменения копирайтов и т. п.).

Кодирование загружаемой части и внутренних оверлеев приложения. Это предохранит приложение от попыток использования другого метода изучения – дизассемблирования.

Используйте опции **проверки электронных ключей по времени**. В этом случае даже снятие полного дампа памяти приложения окажется для хакера пустым развлечением.

Используйте опции **защиты импортов и извлечения кода**. Эти опции позволяют автозащите извлечь ряд инструкций из тела защищаемого приложения и перенести их в тело виртуальной машины. Это позволит противостоять технологиям автоматической деактивации навесных защит, что существенно увеличит сложность и стоимость взлома.

Важная информация

Если применение опций /RIP_CODE и /IMPORT_HOOK сильно замедляют работу приложения, рекомендуется использовать опции /RIP_CODE_LIST и /IMPORT_HOOK_LIST для оптимизации быстродействия защищеной программы. Подробнее см. Руководство пользователя, Часть 1, глава Автоматическая защита

В случае, когда осуществляется привязка к Sign или Time-ключу, используйте **опцию работы с ассиметричным алгоритмом ECC160**. При этом в ходе работы защищенного приложения будут генерироваться случайные данные и подписываться цифровой подписью на эллиптических кривых непосредственно на электронном ключе. Затем подпись будет проверяться функцией Guardant API, защищенной псевдовокодом, шифрованием трафика и другими защитными механизмами.

Приложения на платформе .NET (версии 2.0 – 4.0)

Пользуйтесь символьным обфускатором и утилитой защиты MSIL-кода. При этом целесообразно использовать следующие опции:

Символьная обфускация. Позволяет производить запутывание кода приложения, что вместе с другими мерами серьезно повышает уровень защищенности приложения. Используйте обфускацию для всех *.exe и *.dll сборок в приложении, за исключением сторонних и подписанных цифровой подписью.

Шифрование строк. Привязывает обфусцируемое приложение к ключу Guardant и шифрует строковые константы в защищаемых сборках. Рекомендуется использовать эту опцию, если в приложении нет логических элементов, сильно зависящих от быстродействия строковых констант.

При использовании **опции обфускации публичных интерфейсов** происходит более полная обфускация всей сборки. Однако будьте осторожны, использование этой опции возможно лишь при замкнутости системы (все сборки обфусцируются в одном сеансе, из других приложений никакие методы обфусцируемых сборок не используются). В большинстве случаев безопасно проводить обфускацию публичных интерфейсов для exe-сборок (кроме случаев использования технологии Reflection на самой сборке или экспорта типов для других приложений).

Используйте файл с исключениями. Для его генерации можно воспользоваться утилитой ExclusionUtility.exe, входящей в комплект разработчика. В общем, разработчик приложения должен отчетливо понимать какие типы, методы и свойства необходимо вносить в исключения символьного обфускатора. Как правило, это все те языковые элементы, что могут быть использованы извне. Обратить внимание следует на использование технологий *Serialization, Reflection, Data Dinding.*

При использовании утилиты автоматической защиты MSIL-кода необходимо помнить:

Используйте разумно **опцию с указанием процента защищаемых методов**. Чем больше и сложнее защищаемое приложение, тем меньше необходимо указывать процент. В общем случае, процент 100 можно выставлять для самых простых и нетребовательных к скорости выполнения сборок, содержащих 2 – 3 типа по 10 – 15 методов каждый. Для больших проектов % следует понижать до 10.

Не защищайте **чувствительные к скорости работы** сборки, типы и методы. Помните, что вынос MSIL-кода в шифрованную область и выполнение его на виртуальной машине может в отдельных случаях существенно замедлять его выполнение (особенно при первом вызове). При дальнейших вызовах также присутствуют накладные расходы, и чем меньше размер метода, тем заметнее падает скорость ее выполнения.

Пользуйтесь **опцией и утилитой для задания исключений**. Исключайте из защиты все небольшие по объему методы, не содержащие логики, имеющую коммерческую ценность.

При работе совместно с символьным обфускатором необходимо использовать **опцию создания MAP-файла** (подробнее см. Часть 1 настоящей документации).

Исключайте вся языковые элементы, построенные на **асинхронной передаче данных**. Защита MSIL-кода таких элементов может приводить к непредсказуемым результатам.

Защита при помощи API

Общие рекомендации

Под надежным прикрытием автоматической защиты вы можете строить свою внутреннюю защиту на основе Guardant API. При этом целесообразнее следовать следующим советам:

1. Не храните в явном виде коды доступа к ключу. Не следует хранить в «чистом» виде Личные коды, по которым производится доступ к электронному ключу. Для того чтобы осложнить хакерам их поиск в теле приложения, Личные коды можно закодировать (например, при помощи операции XOR). Непосредственно перед вызовом функции API раскодируйте их, а после того как функция отработала – сразу же удаляйте из памяти раскодированный вариант Личного кода. Также можно хранить Личные коды частями в разных переменных.

2. Не храните «лишние» коды доступа в защищенном приложении. Например, если приложение использует только чтение из ключа и запуск его аппаратных алгоритмов, нужно хранить в приложении только Личный код для чтения (Private Read Code). Остальные Личные коды не будут нужны, а их присутствие в теле приложения может сильно облегчить хакеру задачу получения доступа к ключу.

3. Располагайте вызовы функций API по всему телу приложения. Это резко увеличивает объем кода, который надо изучить хакеру для локализации кода проверок и его модификации.

4. Организовывайте взаимодействие с электронным ключом из **разных потоков**. Многопоточные приложения гораздо труднее отлаживать, а значит и взламывать. При этом полезно усложнить взаимодействие между потоками, организуя его так, чтобы потоки выполняли часть логики приложения, и нельзя было их просто остановить или завершить.

5. Очень хорошим приемом являются **«вероятностные» вызовы**, когда функция API вызывается не всегда, а с какой-то вероятностью – например 1/7. В разных местах приложения можно использовать различные вероятности вызовов (причем в жизненно важных местах приложения вероятность может быть больше, в менее важных местах – меньше). В результате хакер может попросту не найти те вызовы, которые происходят с малой вероятностью – и тогда, якобы «взломанное», приложение рано или поздно преподнесет сюрприз нелегальным пользователям.

6. Усложните логику обработки кодов возврата функций API. Если вы будете проверять код возврата функции API простым **сравнением**, хакеру не составит труда уничтожить это сравнение прямо в теле приложения – и таким образом снять защиту. Разработайте более сложную логику. Например, используйте коды возврата в качестве индексов каких-то массивов данных, стартовых значений для генераторов псевдослучайных чисел, используйте их в алгоритмах математических вычислений, реализованных в приложении и т. п. Таким образом, точка, в которой защищенное приложение принимает решение о своей дальнейшей работе, в чистом виде будет отсутствовать. Хакеру придется разбираться не только в логике работы защиты, но и в хитросплетениях процессов, происходящих в самом приложении. И защита станет полноправной частью приложения, без которой станут неверными его вычисления, будут использоваться не те данные и т. д.

7. Откладывайте момент реакции приложения на коды возврата функций API. Хорошо зарекомендовал себя прием, когда приложение принимает решение о своей дальнейшей работе не сразу по получении кодов возврата функции API, а значительно позже. Например, функция API вызывается при запуске приложения, а анализ возвращенных ею значений производится при выполнении пользователем каких-либо действий (открытие или сохранение файла, вызов меню настройки приложения и т. п.). В таком случае хакеру будет сложно проследить причинно-следственную связь поведения приложения. Кроме того, этот прием заставит хакера исследовать в отладчике большие фрагменты кода приложения.

8. Ограничивайте возможности приложения, если электронный ключ не найден. Этот прием можно использовать в дополнение к описанному выше. Если функция API возвращает ошибку, можно не принимать кардинальных решений относительно дальнейшей работы приложения, а лишь ограничить его возможности. Например, перестать давать возможность экспортировать файлы в другие форматы, сохранять настройки, распечатывать отчеты и т. п. Причем, в ответ на неудачу при вызове API из разных мест приложения можно «лишать» его все новых и новых возможностей, все более превращая в демонстрационное. Помимо важных, можно блокировать и несколько второстепенных возможностей – это увеличит вероятность того, что хакер попросту не заметит их отсутствия, и приложение в таком виде попадет на «черный рынок». Пользователей же такой «демо-версии» отсутствие части ее возможностей подтолкнет на приобретение легальной копии приложения.

9. Подсчитывайте хэш важных участков кода приложения. Это мощное средство защиты от модификации хакером кода приложения. Предварительно вы подсчитываете хэш тех мест в программе, в которых производятся вызовы функций API, анализируются выходные данные функций и принимаются решения о дальнейшей работе приложения. В процессе работы приложения вы еще раз производите подсчет этих участков – и получаете достоверную информацию о том, были ли эти участки нелегально модифицированы. Производить подсчет хэшей лучше в отдельном потоке, причем делать это следует значительно раньше или значительно позже того, как управление попадет в контролируемый участок кода. Один и тот же участок целесообразно контролировать из нескольких мест приложения. А реакция на факт модификации программного кода должна быть примерно такой же, как и на ошибку при вызове функции API. В результате хакер будет вынужден дополнительно искать по всему телу приложения места, в которых вы проверяете контрольные суммы. Для вычисления хэш-функции можно воспользоваться функцией Guardant API GrdHash.

10. Меняйте логику работы приложения с модулями защиты. Очень полезно бывает время от времени (оптимальный вариант – в каждой новой версии защищенного продукта) менять логику его работы с модулями защиты. Используйте новые приемы, подобные описанным здесь, по-иному комбинируйте их и т. п. – и тогда при выходе новой версии продукта хакер будет вынужден начинать работу по ее исследованию, что называется, «с нуля». Все его прежние наработки мгновенно теряют актуальность – ведь теперь приложение взаимодействует с модулями защиты совершенно по-иному.

11. Используйте обфускаторы. Какой бы совершенный код защиты вы не написали, он почти всегда доступен хакеру для исследования посредством отладчиков, дизассемблеров. Есть специальные программы для затруднения анализа программ и понимания смысла того, что они делают. Их называют обфускаторами. Эти программы обрабатывают готовые приложения (exe-файлы) и модифицируют их таким образом, что стандартными средствами невозможно их понять, надо создавать специальные средства именно для данного обфускатора, а то и для данной защищенной программы (обфускаторы, как правило, полиморфны). Есть неплохие сторонние обфускаторы, которые можно использовать для этих целей. Дополнительно обфускаторы часто защищают приложения от изменения таким образом, что невозможно изменить в них даже один бит, сохранив работоспособность программы. Для взлома в таком случае приходится тоже создавать специальные инструменты, что долго и очень дорого.

Использование аппаратных алгоритмов ключей Guardant

Если вы остановили свой выбор на электронных ключах Guardant, в вашем распоряжении есть целый ряд эффективных приемов повышения стойкости защиты к взлому.

1. Задействуйте аппаратные алгоритмы. Это обязательное условие для построения по-настоящему стойкой защиты. Правильное использование ответов аппаратных алгоритмов не только делает практически невозможным написание универсальных эмуляторов ключей Guardant. Удаление из защищенного приложения вызовов функций API также теряет смысл – ведь если аппаратный алгоритм не был запущен, то и важные для приложения данные не были декодированы.

2. Задавайте ключу больше вопросов. Если для защиты используется преобразование только одного массива данных (т. е. если приложение задает ключу всегда один и тот же вопрос), есть вероятность, что хакер все же подсмотрит верный ответ ключа и создаст подпрограмму-«заглушку», которая вместо функции API «подсовывает» приложению этот ответ. Поскольку в данном случае ключ

возвращает всегда один и тот же ответ, такая подпрограмма-«заглушка» может свести роль аппаратного алгоритма в защите данного приложения «на нет». Чтобы избежать этого, нужно задавать аппаратному алгоритму большое количество разных вопросов. Создайте массив различных вопросов (т. е. блоков кодированных данных), и в разных местах приложения декодируйте аппаратным алгоритмом разные кодированные блоки. Кстати, наряду с действительно важными для приложения данными в состав такого массива могут входить и «лишние» данные, на самом деле не нужные приложению – это только дезориентирует хакера. Очень хорошо было бы организовать процесс случайной выборки вопроса – тогда в одном и том же месте приложения, но в разные моменты времени будут обрабатываться разные вопросы. Это сделает практически невозможным для хакера создание, как подпрограммы-«заглушки», так и эмулятора ключа.

3. Используйте разные вопросы в разных версиях приложения. Если в разных приложениях или разных версиях одного и того же продукта будут использоваться разные вопросы к алгоритму (т. е. разные блоки кодированных данных), это даст гарантию того, что хакер не сможет создать универсальный инструмент для взлома всех продуктов (или всех их версий). Даже если хакер исхитрится-таки создать эмулятор для приложения, выход его новой версии заставит хакера производить свою работу заново – ведь новая версия приложения работает уже с другими кодированными данными.

4. Задействуйте разные аппаратные алгоритмы в разных версиях приложения. В электронном ключе создайте сразу несколько различных алгоритмов (например, 4). Затем, в первой версии программного продукта кодируйте данные, скажем, 1-м и 3-м алгоритмом, во второй – 2-м и 4-м, в третьей – 1-м и 2-м, и т. п. Эффект будет аналогичен рассмотренному выше, да и в самом электронном ключе ничего перепрограммировать будет не нужно – все необходимые аппаратные алгоритмы будут в нем присутствовать с самого начала.

5. Вносите изменения в определители алгоритмов. По своей сути определители алгоритмов аналогичны паролям или цифровым сертификатам, поэтому их тоже полезно время от времени менять. Это очень хорошая и распространенная во всем мире практика, повышающая защищенность системы. В ключах с часами реального времени полезно использовать для этого **возможность автоматического изменения определителя каждые N дней**. Алгоритм смены определителя подробно описан в документации, разработчику лишь остается подготовить массивы кодированных данных на несколько месяцев или лет вперед. Как и в предыдущих примерах, однажды взломанная программа может внезапно перестать рабо-

тать т. к. определитель алгоритма сам изменился в какой-то момент времени.

6. Воспользуйтесь алгоритмом цифровой подписи. Алгоритм ECC160 позволяет подписывать произвольные данные с помощью функции GrdSign. Лучше всего, чтобы это были данные, которые возникают в приложении естественным образом (так называемая естественная энтропия): данные, которые пользователь вводит в программу с клавиатуры, движения мышки, значения из баз данных, полученные по запросу пользователя. Тогда простая табличная эмуляция станет невозможной. Проверять правильность подписи можно с помощью функции GrdVerifySign. Однако не следует использовать простое сравнения возвращаемого ею значения, т. к. сравнение по принципу «да/нет» легко сломать с помощью т. н. битхака (исправляется всего 1 бит в программе). Лучше всего собирать возвращаемые значения (например, из битов возврата GrdVerifySign делать 32-битные числа), проверять подпись случайных данных (создавать ложные цели для анализа хакером) и на основе большого числа вызовов GrdSign/ GrdVerifySign создавать значения, которые в дальнейшем можно использовать в работе защищенного приложения.

7. Не храните ответы аппаратных алгоритмов в защищенном приложении. Сам принцип использования аппаратных алгоритмов основан на предпосылке, что хакер не знает заранее ответов алгоритма. В противном случае алгоритмы не имеют особого смысла: хакер, зная все его ответы, может создать и эмулятор ключа, и подпрограмму-«заглушку». Поэтому ни в коем случае не следует хранить ответы ключа в приложении или в файлах данных! Просто используйте ответы ключа по назначению, без предварительной проверки правильности их декодирования алгоритмом, а сразу после использования – по возможности удаляйте из памяти. Данные могут быть декодированы неверно лишь в одном случае – если либо с защищенным приложением, либо с электронным ключом производились какие-то сомнительные манипуляции. Но, в таком случае, неверная работа (и даже «подвисание») приложения – событие весьма естественное. В самом деле, чего еще можно ожидать от приложения, «ломаемого» хакером, особенно если защита «надета» на приложение качественно? Но если проверка правильности ответов все же необходима, используйте для этого функцию подсчета CRC. На этапе установки защиты, перед тем как закодировать данные, вычислите их CRC, а в процессе работы приложения подсчитайте CRC этих же данных после их декодирования алгоритмом ключа. Если эти два значения совпали, то данные декодированы верно.

Важная информация

В связи с постоянно растущими вычислительными возможностями современных компьютеров настоятельно рекомендуется задавать размер вопроса/ответа и определителя аппаратных алгоритмов ключей Guardant не менее чем 8 байт. Это позволит минимизировать шансы атаки методом brute force (полный перебор).

Особенности использования ключей Guardant Sign / Time / Code

При работе с электронных ключами Guardant необходимо учитывать несколько важных особенностей, связанных с активным использованием ассиметричной криптографии, как в алгоритмах ключа, так и в защитных механизмах. Эти моменты необходимо учитывать при проектировании защиты.

1. Безопасный обмен сеансовыми ключами. При выполнении функции GrdLogin происходит взаимная аутентификация электронного ключа и Guardant API и безопасный обмен сеансовыми ключами. Слово «обмен» - это просто термин. На самом деле API и электронный ключ рассчитывают это значение независимо на основе безопасной ассиметричной схемы. В протоколе обмена его подсмотреть невозможно. Поэтому команда GrdLogin занимает существенно больше времени, чем остальные.

2. Периодическая смена сеансовых ключей. Если хакеру удастся как-то извлечь из защищенного псевдокодом Guardant API сеансовый ключ, то это тоже не принесет ему много пользы. Электронный ключ устроен так, что он работает на любом сеансовом ключе не больше нескольких минут. По его истечении, электронный ключ начинает возвращать специальный код ошибки в ответ на любую команду, полученную с использованием просроченного сеансового ключа, до тех пор, пока данная копия API опять не пройдет взаимную аутентификацию и не перегенерирует его заново. Соответственно, при проектировании защиты нужно учитывать тот факт, что эта перегенерация может занять какое-то дополнительное время. И если логика работы приложения не допускает таких задержек в выполнении, то лучше организовывать запросы к электронному ключу из параллельно работающих потоков.

3. Использование GrdSign и GrdVerifySign. Это очень мощный инструмент, который при правильном применении дает возможность создать защиту беспрецедентно высокого уровня. Для этого нужно во время работы защищенной программы аппаратно подписать (GrdSign) и программно проверить подпись (GrdVerifySign) от неких каждый раз разных случайных данных. Проверка подписи специально реализована чисто программно, так как это лишает хакера даже гипотетической возможности перехватить ее в эмуляторе на уровне драйвера или USB-шины. А ключ шифрования в ней ис-

пользуется публичный, который не может быть скомпрометирован.

4. Атака на генератор случайных чисел. Если хакер сможет сделать в электронный ключ запросы не случайными, а постоянными, то у него появится возможность сделать табличный эмулятор, который будет знать ответы на все вопросы, задаваемые программой. Чтобы посылаемые данные были действительно случайными, настоятельно рекомендуется использовать какой-либо хеш (например, SHA-256), от действительно случайных данных, которые использует программа и без которых она теряет смысл или львиную долю функциональности. Например, введенные пользователем данные, ID созданных потоков, адреса аллокированной памяти, и т. п. Иначе хакер сможет подсовывать приложению любые нужные ему значения в ответ на любые вызовы Guardant API.

5. Атака на подмену публичного ключа. Хотя публичный ключ ассиметричного алгоритма не является секретным, но, тем не менее, необходимо принять все меры по предотвращению его модификации и/или подмены.

a. Не хранить его просто инициализированным в сегменте данных (на Си это инициализированная глобальная или статическая переменная или константа), так как там его проще всего подменить

b. Рассчитывать его значение непосредственно перед использованием и сразу после использования затирать и удалять.

c. Контролировать целостность кода, который работает с ним.

6. Атака на подмену Guardant API. Чтобы хакер не мог просто заменить код функций Guardant API на свой эмулятор, необходимо:

a. Вызывать GrdVerifySign так, чтобы она в некой части возвратов возвращала другие коды ошибок, которые также надо проверять. Например, подавать на вход случайные данные в сообщении или подписи или неверный общий ECC ключ.

b. Среди этих вызовов очень полезно время от времени вызывать GrdVerifySign с запросом на проверку валидности подписи одного из валидных сообщений, подписанного на этом же ключе, но не в процессе работы программы, а на этапе защиты. Для этого при защите приложения в нем необходимо сохранить какое-то количество подписанных случайных сообщений с подписями. Аналогично тому, как это делается с различными массивами вопросов и ответов, используемыми при работе с симметричными алгоритмами.

c. Результат проверки подписи GrdVerifySign по сути один бит. Можно вместо того, чтобы ставить явную проверку кода возврата, составить из множества вызовов одну константу, которая будет использоваться в дальнейших вычислениях. Благо скорость работы подписи в ключе достаточно высока.

7. Использование однонаправленных симметричных алгоритмов шифрования AES и GSII64. Можно, например, сделать так, чтобы в электронном ключе для защиты приложения некий AES или GSII64 алгоритм работал в обоих направлениях, а в попадающем к конечным пользователям ключе он мог только расшифровывать. Тогда можно хранить в некой защищенной ячейке (или файле) какую-либо конфигурационную или лицензионную информацию, зашифрованную на этом алгоритме при продаже софта не боясь, что хакер сможет подменять ее. Ведь для подмены нужно записать в эту ячейку информацию, зашифрованную на этом алгоритме. А зашифровать ее на ключе пользователя, на этом же алгоритме хакер просто не сможет, из-за его однонаправленности. Аналогичные механизмы можно применять с распространением зашифрованных обновлений, которые хакер, имея ключ конечного пользователя с однонаправленным алгоритмом, не сможет зашифровать, модифицировать и подсунуть программе.

Важная информация

Однонаправленные алгоритмы AES128 можно использовать только в блочных режимах ECB и CBC. Поточные режимы CFB и OFB использовать нельзя.

8. Атака на замену GrdAPI.DLL. Обычно настоятельно рекомендуется всем нашим пользователям по возможности использовать статическую линковку, т. е. по сути, использовать Guardant API в виде OBJ вместо DLL. Если же это невозможно из-за ограничений средств разработки, то крайне рекомендуется сделать следующее:

a. Перенести часть функциональности во внешнюю DLL, к которой статически прилинкована библиотека Guardant API (OBJ).

b. Использовать традиционные механизмы работы с симметричными алгоритмами. Массивы вопросов и ответов, случайный выбор таблицы и запроса. Проверки при выполнении какой-либо функциональности.

c. Время от времени вызывать GrdVerifySign с запросом о проверке валидности подпись одного из валидных случайных сообщений, подписанного на этом же ключе, но не в процессе работы программы, а на этапе защиты. Если хакер даже перехватит все вызовы Guardant API, то он не будет знать, что вернуть на такой вопрос.

9. Time-ключи. Появилась возможность надежно задавать ограничение работы, как всего электронного ключа, так и любого конкретного алгоритма. Это дает возможность сдавать ПО в аренду. Однако рекомендуется уведомлять конечного пользователя о близости прекращения работа программы заранее, для того, чтобы он успел продлить лицензию до того, как программа перестанет работать. Для этого в Guardant API есть все необ-

ходимые функции типа GrdPI. См. документацию на Guardant API. Аналогичные опции предусмотрены в новом мастере автозащиты.

10. Счетчик запусков. Аналогично, в Sign-ключах появилась возможность получить оставшееся значение счетчика запусков алгоритма. Соответственно, о приближении его завершения нужно информировать конечного пользователя заранее. Для этого есть специальная функция типа GrdPI_. См. документацию на Guardant API. Эти же опции предусмотрены в новом мастере автозащиты.

11. Используйте драйвер Guardant. Хотя использования ключей в HID-режиме очень удобно для распространения защищенной программы, все же наличие драйвера Guardant увеличивает защищенность, и по возможности мы рекомендуем все же использовать ключи не в HID-режиме. Тем не менее, надо обратить внимание, что программа, защищенная ключами в HID-режиме, в любом случае обеспечивает защиту гораздо более высокую, чем предыдущие поколения электронных ключей.

12. Используйте Time-ключи не только для ограничения времени работы, но и для повышения уровня защищенности. В отличие от других типов Time-ключей, электронные ключи Guardant Time можно использовать для повышения уровня защищенности. Такая уникальная возможность как BirthTime позволяет создавать такие системы защиты, которые активируются спустя время и анализировать которые хакер никак не может заранее. А технология FlipTime позволяет создавать алгоритмы, определители которых меняются через заданное количество дней, что опять затруднит анализ защиты заранее. Утилита для генерации определителей FlipTime также включена в комплект разработчика вместе с исходным кодом.

13. Используйте высокую скорость работы новых ключей. Guardant Sign / Time / Code работают до 10 раз быстрее, чем их предшественники. Поэтому с их помощью можно шифровать и расшифровывать гораздо больше информации, и это не отразится на быстродействии защищенной программы.

14. Защита баз данных. Можно реализовать вариант защиты, при котором база данных шифруется с помощью программного AES. При продаже база комплектуется электронным ключом, содержащим аппаратный алгоритм AES с теми же параметрами, как у программного алгоритма, на котором база данных была зашифрована. Отличием алгоритма в электронном ключе будет то, что он работает только в режиме расшифрования (без зашифрования) и расшифровывает не всю базу данных, а некую ее часть, нужную пользователю. В этом случае, содержимое базы данных нельзя будет модифицировать.

Этот вариант подходит для случаев, когда у конечного пользователя нет необходимости пополнять базу данных.

Рекомендации по написанию загружаемого кода (Code)

1. Основная угроза для систем защиты, основанных на электронных ключах – это эмуляторы. Всегда существует шанс, что в результате анализа «черного ящика», которым является электронный ключ, злоумышленник сможет построить его эмулятор. Это может быть либо табличный эмулятор, либо универсальный эмулятор алгоритма.

При анализе черного ящика злоумышленник будет пытаться определить, каким образом изменения входных данных повлияют на выходные данные. Если этих днных мало, либо если алгоритм слишком простой, с высокой вероятностью можно построить алгоритм, функционально эквивалентный «черному ящику».

Поэтому алгоритм загружаемого кода должен быть нетривиальным и с достаточно большой сложностью. Он должен принимать на вход и давать на выход достаточно много данных.

2. Что делать, если функционально алгоритму требуется мало данных? Можно ввести дополнительные данные, не связанные напрямую с функциональностью алгоритма. По сути, это будут «мусорные» данные, однако их использование будет затруднять анализ «черного ящика». Аналитику придется разбираться, какие данные и для чего нужны, и нужны ли вообще.

3. Существует возможность неявного вызова аппаратных алгоритмов через загружаемый код. Например, можно возвращать данные из загружаемого кода зашифрованными на симметричном алгоритме AES128. Перед использованием они должны быть расшифрованы с использованием того же секретного ключа. Это увеличивает разнообразие данных, проходящих через интерфейс электронного ключа, и затрудняет эмуляцию. Для большего усложнения можно периодически менять ключ шифрования AES128 изнутри загружаемого кода.

4. Защищенные ячейки можно использовать для неявного обмена данными между загружаемым кодом и приложением. В них можно помещать как часть входных данных для загружаемого кода, так и принимать через них данные от загружаемого кода.

5. Загружаемый код должен контролировать данные, принимаемые от приложения, по размеру и корректности. Это поможет избегать определенного класса атак.

6. При возможности можно использовать сохранение данных в ключе между вызовами, так как размер таблицы вопросов-ответов в этом случае получается на порядок больше. Очевидно, такой подход сильно затруднит процесс построения эмуляторов.

7. Если в защищаемом приложении нет фрагментов со значительными вычислениями (только такие разумно размещать в Guardant Code), можно использовать идею трудности анализа большого количества простых множеств. Т.е. построить защиту на том, что размещаемые в ключе фрагменты небольшие, они делают простые вещи, но зато их много. Правда, это, по-прежнему, должны быть фрагменты, которые используются не слишком часто. Количество фрагментов должно быть действительно большим. Их точное количество зависит от сложности вычислений в них. Например, если во фрагментах только простые операции сложения или вычитания – их должно быть около пары сотен. Если есть умножение или деление – достаточно несколько десятков. Если функции типа логарифмов или синусов/косинусов, достаточно 10. Данные цифры действительны на ближайшие несколько лет (стойкость всегда оговаривается сроками).

8. Поскольку загрузка кода в Guardant Code – операция настолько безопасная, что может осуществляться у конечного пользователя, и быстрая даже для больших кусков загружаемого кода, можно создать защищенную программу, которую будет так же легко обновлять, как и незащищенную программу. Для этого не потребуется что-либо специальное делать с электронным ключом при обновлении – достаточно просто отправить клиенту обновление.

Более того, существует возможность даже не записывать в Guardant Code перед продажей загружаемый код! Это можно сделать в момент 1-го запуска программы, проанализировав значения, возвращаемые функцией **GrdGetCodeInfo()**:

- Перед продажей программы записать в ключ маску без загружаемого кода
- В защищенной программе добавить проверку – загружен ли код в Guardant Code (точнее какой именно код загружен – существуют поля версии загруженного кода). Если кода нет, или он не тот, то загрузить его в момент инициализации приложения. Это произойдет только один раз при 1-м запуске программы. Поскольку загружаемый код зашифрован и подписан ЭЦП, в него невозможно внести изменений и невозможно подменить другим кодом или кодом от другой программы

- Поскольку бывают программы разной стоимости, и обновления тоже могут быть платными или бесплатными, нужно предусмотреть политику ограничения вариантов загрузки кода. То есть, надо сделать так, чтобы Update, предназначенный для дорогой программы, нельзя было загрузить в ключ, который был куплен для дешевой программы. Сделать это просто: ключи Guardant Code для разных программ должны иметь разные пары закрытых ключей для проверки ЭЦП загружаемого кода

Нужно обратить внимание, что контроль версии загружаемого кода самим ключом не осуществляется. Это может сделать только программа, которая загружает код. Поэтому данный метод не годится для платных обновлений, в отличие от бесплатных. Не годится потому, что обновление можно загрузить бесплатно в любой ключ, который содержит необходимые ключи шифрования и подписи для загрузки кода. Бесплатные обновления традиционно служат для исправления ошибок в ПО. Платные обновления применяются чаще всего тогда, когда пользователь программы получает новые возможности. Для платных обновлений надо пользоваться механизмом TRU, т. к. он настроен для работы с конкретным ключом и позволяет реализовать процедуру адресного обновления.