Guardant®

Система защиты от компьютерного пиратства

Эффективная
защита приложений

Урок 3.2: Guardant API
Использование симметричных алгоритмов

Содержание

Общее описание метода защиты

Базовыми элементами маски ключа, с использованием которых могут быть организованы защитные механизмы приложения, являются:

- Симметричные алгоритмы шифрования
- Асимметричные алгоритмы шифрования
- Защищенные ячейки

Данный урок посвящен защите функционала приложения при помощи **симметричных алгоритмов**, а именно, алгоритма **AES128**, поддерживаемого всеми ключами поколения Sign/Time.

Обращение к симметричным аппаратным алгоритмам может происходить при помощи функций **GrdTransformEx()** и **GrdCryptEx()**.

Классическая схема использования алгоритма шифрования AES128 выглядит следующим образом:

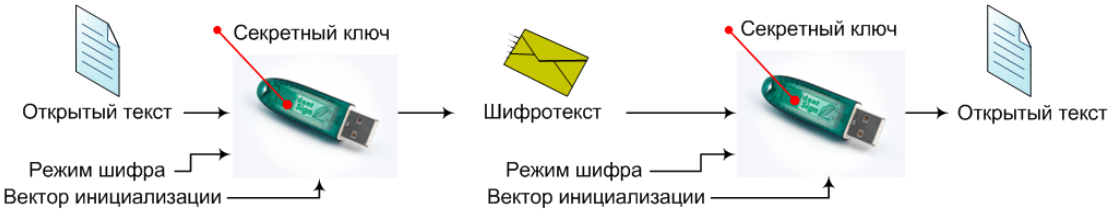

Рисунок 1. Классическая схема использования симметричного алгоритма шифрования

Вектор инициализации определяет начальное состояние алгоритма и выступает в качестве т. н. «соли» для повышения криптографической стойкости схемы шифрования в целом. Важно следить, чтобы при шифровании и расшифровании на вход алгоритма подавалось одно и то же значение вектора инициализации. Его значение не является секретным и может передаваться вместе с шифротекстом по открытому каналу.

Выбор **режима работы** шифра имеет принципиальное значение. Он целиком зависит от целей и способа использования алгоритма шифрования. Так, к примеру, для шифрования сессионных ключей достаточно режима ECB, однако, во многих других случаях его использование недопустимо.

Симметричный аппаратный алгоритм электронного ключа можно использовать в целях защиты одним из следующих **способов**:

- Для шифрования статических данных приложения (к примеру, текстовых строк), в т.ч. использование таблиц вопросов-ответов
- Для шифрования исполняемого кода приложения
- Для защиты конфиденциальности передаваемых/хранимых данных

Простой и, в то же время, наиболее распространенной **атакой**, реализуемой на **шифрование статических данных**, является построение таблицы вопросов-ответов вызовов алгоритма в процессе работы защищенного приложения. Использование различных значений вектора инициализации может усложнить предполагаемому противнику проведение указанной атаки.

Шифрование исполняемого кода защищаемого приложения в нашем случае неприменимо, так как после компиляции приложения предполагается установка автозащиты.

Таким образом, наиболее эффективным способом защиты приложения при помощи симметричного алгоритма является использование его «по прямому назначению», то есть для шифрования передаваемого трафика, сессионных ключей или других, периодически изменяющихся данных.

В общем случае защищаемое приложение может не содержать функционала передачи или хранения данных, поэтому будем рассматривать несколько искусственный вариант использования алгоритма шифрования – для защиты информации, хранимой и передаваемой внутри защищаемого приложения в процессе его работы.

Контрольный пример

На примере простого консольного приложения, реализующего выбор напитка, рассмотрим использование симметричного алгоритма блочного шифрования AES128 для защиты данных алгоритма принятия решений и реализации лицензионных ограничений. Выбор напитка происходит из вектора значений, сориентированного случайным образом. Ниже приведён листинг программы на языке C++:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <ctime>


using namespace std;
ptrdiff_t random (ptrdiff_t i) { return (rand()% i);}
ptrdiff_t (*p_random)(ptrdiff_t) = random;

int main(int argc, char* argv[])
{
      vector<string> ans;
      string q="";
      srand ( unsigned ( time(NULL) ) );
      ans.push_back("Coffee");
      ans.push_back("Tea");
      random_shuffle(ans.begin(),ans.end(),p_random);
      cout << "============ Tea or Coffee ...=============="<< endl;
      cout << "Heads or tails ? Enter you choice ( \"h\" or \"t\" ):";
      while ((q!="t") & (q!="h"))      cin >>q;
      cout << "Today your drink is ";
      if (q=="t")
            cout<< ans[0];
      else
            cout<< ans[1];
      cout << " !!!" << endl;
      cout << "=========================================="<< endl;
      return 0;
}
```

Реализация защиты

Будем защищать входные данные алгоритма принятия решений о выборе напитка – пользовательский ввод.

Прежде всего, необходимо определить номер (т.н. **Числовое имя**) алгоритма типа AES128 в маске ключа. Сделать это можно при помощи утилиты программирования ключей GrdUtil, порядок работы с которой описан в уроке 1.2. При отсутствии указанного алгоритма в последней маске, записанной в ключ, его необходимо добавить и записать маску в ключ:

096	r w	Алгоритм 08 (ECC160)	ECC 160 digital sign	F5 C1 4B 1F 18 8D 24 54 F2 43 DE B9 39 7C
092	r w	Алгоритм 09 (AES128)	AES 128 Demo	AC 5F A9 AF 80 67 DD 90 CA B9 D3 8A 7D B8
092	r w	Алгоритм 10 (GSII64 Encode)	GSII64 Encode	78 54 A3 22 17 E2 FE 61 23 AA AD A9 79 9A

Числовое имя алгоритма будет передаваться функции **GrdCryptEx()** при обращении к алгоритму AES128 с заданным в маске секретным ключом.

При программировании ключа в свойствах аппаратного алгоритма устанавливаются **лицензионные ограничения** защищаемого функционала (временные зависимости или счетчик запусков алгоритма). Так как автозащита подразумевает использование симметричного алгоритма, то необходимо уделять внимание заданию номера алгоритма при наложении автозащиты. Не следует указывать для использования автозащитой алгоритмы с ограниченным числом запусков.

Важным моментом является установка значения **вектора инициализации** перед обращением к алгоритму. Будем использовать текущее системное время с точностью до 2 секунд.

Ниже приведем пример тестового приложения с добавленным в него функционалом шифрования:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <ctime>
#include "grddongle.h"

using namespace std;
ptrdiff_t random (ptrdiff_t i) { return (rand()% i);}
ptrdiff_t (*p_random)(ptrdiff_t) = random;

#define MESSAGE_SIZE 0x10

char    szMessage[MESSAGE_SIZE];    // Защищаемые данные
char    pIV[GrdARS_AES128];         // Вектор инициализации
string      q="";

int main(int argc, char* argv[])
{
      vector<string> ans;
      srand ( unsigned ( time(NULL) ) );
      memset(szMessage, 0, MESSAGE_SIZE);

      int nRet = 0;
```

```
	CGrdDongle GrdDongle( GrdFMR_Local );
	nRet = GrdDongle.Create(GrdDC_DEMONVK, GrdDC_DEMORDO, 0, 0);
	nRet = GrdDongle.Login( -1l );
	if(nRet)
		cout << "GrdLogin() failed!" << endl;
	else
	{
		ans.push_back("Coffee");
		ans.push_back("Tea");
		cout << "============ Tea or Coffee ...=============="<< endl;
		cout << "Heads or tails ? Enter you choice ( \"h\" or \"t\" ):";
		while ((q!="t") & (q!="h"))
		{
			cin >>q;
			// Значение помещаем в сереседину массива для подписи
			szMessage[MESSAGE_SIZE / 2] = *q.begin();
			// Устанавливаем значение вектора инициализации
			memset(pIV, 0, GrdARS_AES128);
			time((time_t*)pIV);
			if(pIV[0]%2) pIV[0]--;
			// Шифруем массив при помощи алгоритма AES128 в режиме OFB
			nRet = GrdDongle.CryptEx(0x0009,     // Номер алгоритма AES128
						sizeof(szMessage), szMessage,
						GrdAM_OFB|GrdAM_Encode, sizeof(pIV), pIV );
			if(nRet)
				cout << "GrdCryptEx() failed!" << endl;
		}

		// Сохраняем или передаем зашифрованный массив
		// ...

		// Устанавливаем значение вектора инициализации
		memset(pIV, 0, GrdARS_AES128);
		time((time_t*)pIV);
		if(pIV[0]%2) pIV[0]++;
		// Расшифровываем массив с использованием обратного преобразования
		nRet = GrdDongle.CryptEx(0x0009,              // Номер алгоритма AES128
						sizeof(szMessage), szMessage,
						GrdAM_OFB|GrdAM_Decode, sizeof(pIV), pIV );
		if(nRet)
			cout << "GrdCryptEx() failed!" << endl;
		else
		{
			// Алгоритм принятия решения о выборе напитка
			random_shuffle(ans.begin(),ans.end(),p_random);
			cout << "Today your drink is ";
			if (q=="t")
				cout<< ans[0];
			else
				cout<< ans[1];
			cout << " !!!" << endl;
		}
		cout << "=========================================="<< endl;

	}
	return 0;
}
```

После компиляции приложения настоятельно рекомендуется установить автозащиту с целью защиты кода приложения от анализа и несанкционированной модификации. Порядок наложения автозащиты с самостоятельным программированием ключа был подробно описан в **уроке 1.3**.

Заключение

В процессе выполнения данного урока были рассмотрены основные действия, выполняемые при защите функционала приложения при помощи симметричного алгоритма ключа.

Описанный подход защиты при помощи симметричного алгоритма AES128 может применяться для противодействия автоматизированным средствам построения эмуляторов, однако, не является эффективным против анализа кода приложения и исключения из него вызов Guardant API.

Таким образом, рекомендуется дополнительно контролировать корректность работы алгоритма шифрования, используя как «положительный», так и «отрицательный» (расшифрование заведомо некорректных данных) результат шифрования, а также реализовывать любые другие механизмы, усложняющие анализ защиты. Для этого могут быть полезны «однонаправленные» симметричные алгоритмы, средства автоматической генерации распределенных таблиц вопросов-ответов и другие инструменты, поставляемые в составе комплекта разработчика Guardant.

Защита приложения, основанная на уникальных механизмах, реализованных разработчиком приложения и известных только ему, потенциально имеет значительно более высокую стойкость по сравнению с защитой на основе любых шаблонных методов.

Дополнительные источники информации

При возникновении вопросов, на которые вам не удалось найти ответа в этом пособии, рекомендуем обратиться к следующим дополнительным источникам информации:

WWW: http://www.guardant.ru

Web-сайт разработчика содержит большой объем справочной информации об электронных ключах Guardant.

Служба технической поддержки:

e-mail: hotline@guardant.ru

тел. +7(495)925-77-90