Шаг №1. *Установка QtSdk.*

Скачиваем с официального сайта (http://qt.nokia.com/downloads) последнюю версию **Qt****S**dk.
Устанавливаем её в нашу систему. Пусть, для удобства описания, директорией для установки будет *~/qtsdk*
Qt**S**dk поставляется уже скомпилированная под конкретную платформу, поэтому сразу после установки готова к использованию. Главное не забываем при компиляции чётко указывать пути именно для этой версии **Q**t и **qmake**.

Шаг №2. *Подготовка к использованию.*

Создаём копию плагина sqlite с именем sqlcipher
$ cd ~/qtsdk/qt/src/plugins/sqldrivers
$ cp -rf sqlite sqlcipher

Шаг №3.1 *Установка и сборка SqlCipher (Linux).*

Клонируем из репозитория исходники **S**ql**C**ipher. Для этого в Вашей системе должен быть установлен git. Клонирование производится в любое удобное место.
$ git clone git://github.com/sjlombardo/sqlcipher.git

Далее, нам необходимо собрать **S**ql**C**ipher. Для этого выполняем следующие действия (не забываем изменить путь в --prefix)

$ ./configure --prefix=/home/user/qtsdk/qt/src/plugins/sqldrivers/sqlcipher CFLAGS="-DSQLITE_HAS_CODEC" LDFLAGS="-lcrypto"
$ make
$ make install

Теперь в *~/qtsdk/qt/src/plugins/sqldrivers* у нас появилось структура каталогов ***bin, include, lib*** - которые необходимы нам для сборки нового sql плагина для **Q**t.

Шаг №3.2 *Установка и сборка SqlCipher (Windows).*

3.2.1 Установка дополнительного ПО.

Скачиваем и устанавливаем на свой компьютер MSYS. В конце установки будет задано несколько вопросов, соглашаемся с ними и указываем путь до MinGW (входит в поставку Qt).

Скачиваем и устанавливаем Win32 OpenSSL v0.9.8m. Во время установки не менять место размещения dll-файлов.

Так же нам потребуется TclTk (специальная сборка для MinGW). Скачиваем, устанавливаем. Во время установки указываем путь до MinGW.
Находим файл MINGWPATH/bin/**tclsh84.exe** и переименовываем его в **tclsh.exe**

3.2.2 Конфигурирование и компиляция sqlcipher

Для сборки sqlcipher, запускаем MSYS и выполняем следущие действия (обязательно учитываем регистр при написании)

$ cd sqlcipher
$ ./configure --prefix=/QTPATH/src/plugins/sqldrivers/sqlcipher --disable-tcl --disable-amalgamation CFLAGS="-DSQLITE_HAS_CODEC -I../OpenSSL/include /c/Windows/System32/libeay32.dll"
$ make

```
$ make dll
$ make install
$ cp /OPENSSLPATH/lib/MinGW/libeay32.a /QTPATH/src/plugins/sqldrivers/sqlcipher/lib
```

Замечание: Для того что бы в MSYS перейти на диск C, необходимо набрать команду *cd /c (*для D - *cd /d).* Для удобства пользуйтесь автодополнением через TAB.

Шаг №4. *Создание нового sql плагина для Qt.*

Переходим в заранее подготовленную директорию для создания нового sql плагина для **Q**t

```
$ cd ~/qtsdk/qt/src/plugins/sqlcipher
```

Переименовываем файл проекта

```
$ mv sqlite.pro sqlcipher.pro
```

Открываем в любом текстовом редакторе файл *sqlcipher.pro*. Модифицируем его следующим образом.

```
TARGET = qsqlcipher

HEADERS = ../../../sql/drivers/sqlite/qsql_sqlite.h
SOURCES = smain.cpp \
../../../sql/drivers/sqlite/qsql_sqlite.cpp

!system-sqlite:!contains( LIBS, .*sqlite.* ) {
   CONFIG(release, debug|release):DEFINES *= NDEBUG
   DEFINES += SQLITE_OMIT_LOAD_EXTENSION SQLITE_OMIT_COMPLETE

   INCLUDEPATH += include

   win32 {
      LIBS += ./lib/libsqlite3.a ./lib/libeay32.a
   }
   unix {
      QMAKE_RPATHDIR += lib
      LIBS += -Llib -lsqlite3
   }

} else {
   LIBS *= $$QT_LFLAGS_SQLITE
   QMAKE_CXXFLAGS *= $$QT_CFLAGS_SQLITE
}

include(../qsqldriverbase.pri)
```

модифицируем файл *smain.cpp*

```
#include <qsqldriverplugin.h>
#include <qstringlist.h>
#include "../../../../src/sql/drivers/sqlite/qsql_sqlite.h"

QT_BEGIN_NAMESPACE

class QSqlCipherDriverPlugin : public QSqlDriverPlugin
{
public:
```

```
    QSqlCipherDriverPlugin();

    QSqlDriver* create(const QString &);
    QStringList keys() const;
};

QSqlCipherDriverPlugin::QSqlCipherDriverPlugin()
    : QSqlDriverPlugin()
{
}

QSqlDriver* QSqlCipherDriverPlugin::create(const QString &name)
{
    if (name == QLatin1String("QSQLCIPHER")) {
        QSQLiteDriver* driver = new QSQLiteDriver();
        return driver;
    }
    return 0;
}

QStringList QSqlCipherDriverPlugin::keys() const
{
    QStringList l;
    l  << QLatin1String("QSQLCIPHER");
    return l;
}

Q_EXPORT_STATIC_PLUGIN(QSQLiteDriverPlugin)
Q_EXPORT_PLUGIN2(qsqlcipher, QSqlCipherDriverPlugin)

QT_END_NAMESPACE
```

Компилируем плагин

```
$ ~/qtsdk/qt/bin/qmake
$ make
$ make install
```

В каталоге *~/qtsdk/qt/plugins/sqldrivers* у Вас должен появиться файл *libqsqlcipher.so*

Шаг №5. *Создание тестового приложения.*

```
$ cd /home/user/qtsdk/qt/examples/sql/sqlwidgetmapper
```

В качестве тестого приложения нам подойдет стандартный пример из поставки **Q**t *sqlwidgetmapper*. Необходимо только немного модифицировать в файл *window.cpp* метод **setupModel**

```
void Window::setupModel()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLCIPHER");

    if (!QFile::exists("test.db")) {
        db.setDatabaseName("test.db");
```

```
    if (!db.open()) {
        QMessageBox::critical(0, tr("Cannot open database"),
            tr("Unable to establish a database connection.\n"
               "This example needs SQLCipher support."),
                QMessageBox::Cancel);
        return;
    }

    QSqlQuery query;
    query.exec("pragma key = '12345';");
    query.exec("create table person (id int primary key, "
               "name varchar(20), address varchar(200), typeid int)");
    query.exec("insert into person values(1, 'Alice', "
               "'<qt>123 Main Street<br/>Market Town</qt>', 101)");
    query.exec("insert into person values(2, 'Bob', "
               "'<qt>PO Box 32<br/>Mail Handling Service"
               "<br/>Service City</qt>', 102)");
    query.exec("insert into person values(3, 'Carol', "
               "'<qt>The Lighthouse<br/>Remote Island</qt>', 103)");
    query.exec("insert into person values(4, 'Donald', "
               "'<qt>47338 Park Avenue<br/>Big City</qt>', 101)");
    query.exec("insert into person values(5, 'Emma', "
               "'<qt>Research Station<br/>Base Camp<br/>"
               "Big Mountain</qt>', 103)");
//! [Set up the main table]

//! [Set up the address type table]
    query.exec("create table addresstype (id int, description varchar(20))");
    query.exec("insert into addresstype values(101, 'Home')");
    query.exec("insert into addresstype values(102, 'Work')");
    query.exec("insert into addresstype values(103, 'Other')");
} else {
    db.setDatabaseName("test.db");
    if (!db.open()) {
        QMessageBox::critical(0, tr("Cannot open database"),
            tr("Unable to establish a database connection.\n"
               "This example needs SQLCipher support."),
                QMessageBox::Cancel);
        return;
    }

    QSqlQuery query;
    query.exec("pragma key = '12345';");

    {//===-Test-===
        query.exec("select * from addresstype;");
        if (query.lastError().type() != QSqlError::NoError) {
            QMessageBox::critical(0, tr("Cannot open database"),
                tr("%1").arg(query.lastError().text()), QMessageBox::Cancel);
            return;
        }
    }
}

model = new QSqlRelationalTableModel(this);
model->setTable("person");
```

```
    model->setEditStrategy(QSqlTableModel::OnManualSubmit);

    typeIndex = model->fieldIndex("typeid");

    model->setRelation(typeIndex,
          QSqlRelation("addresstype", "id", "description"));
    model->select();
}
```

```
$ ~/soft/qtsdk-2010.02/qt/bin/qmake
$ make

$ ./sqlwidgetmapper
```

После закрытия программы, открываем файл test.db и видим что все данный у на зашифрованы. При повторном запуске программы БД так же доступна. А вот если закомментировать query.exec("pragma key = '12345';"); то при запуске приложения мы получим сообщение об ошибке