

# Using SQL Databases as Universal Keystores

The following is a short extract showing how you can use an embedded SQL database in a mobile phone as a sophisticated user-keystore. How keys are protected varies but would in a perfect implementation be performed by a TPM or similar.

## SQL Definitions

```
/*=====*/
/*          PUKPOLICIES Table          */
/*=====*/

CREATE TABLE PUKPOLICIES
(
  PUKPolicyID    INT          NOT NULL  AUTO_INCREMENT,           -- Unique ID of PUK policy
  Created        TIMESTAMP    NOT NULL  DEFAULT CURRENT_TIMESTAMP, -- Creation time
  RetryLimit     SMALLINT     NOT NULL,                          -- PUK tries before locking the keys for good
  PUKTryCount    SMALLINT     NOT NULL,                          -- Decrementated for each error, locking at 0
  Format         SMALLINT     NOT NULL,                          -- Ordinal (0..n) of "PassphraseFormats"
  PUKValue       BLOB (32)    NOT NULL,                          -- The actual PUK value
  PRIMARY KEY (PUKPolicyID)
);

/*=====*/
/*          PINPOLICIES Table          */
/*=====*/

CREATE TABLE PINPOLICIES
(
  PINPolicyID    INT          NOT NULL  AUTO_INCREMENT,           -- Unique ID of PIN policy
  Created        TIMESTAMP    NOT NULL  DEFAULT CURRENT_TIMESTAMP, -- Creation time
  RetryLimit     SMALLINT     NOT NULL,                          -- PIN tries before locking the key(s)
  PUKPolicyID    INT          NOT NULL,                          -- For every PIN there is a governing PUK
  --
  --      User PIN set constraints
  --
  Format         SMALLINT     NOT NULL,                          -- Ordinal (0..n) of "PassphraseFormats"
  MinLength      SMALLINT     NOT NULL,                          -- Shortest acceptable PIN
  MaxLength      SMALLINT     NOT NULL,                          -- Longest acceptable PIN
  Grouping       SMALLINT     NOT NULL,                          -- Ordinal (0..n) of "PINGrouping"
  PatternRestr  BLOB (20)    NULL,                               -- "PatternRestrictions" [len + ordinals]
  --
  --      API control
  --
  InputMeth     SMALLINT     NOT NULL,                          -- Ordinal (0..n) of "InputMethods"
  CachingSupp   BOOLEAN      NOT NULL,                          -- Caching PIN support option
  --
  FOREIGN KEY (PUKPolicyID) REFERENCES PUKPOLICIES (PUKPolicyID),
  PRIMARY KEY (PINPolicyID)
);

/*=====*/
/*          USERKEYS Table          */
/*=====*/

CREATE TABLE USERKEYS
(
  KeyID          INT          NOT NULL  AUTO_INCREMENT,           -- Each key gets a unique ID in the database
  Created        TIMESTAMP    NOT NULL  DEFAULT CURRENT_TIMESTAMP, -- Creation time
  FriendlyName   VARCHAR (50) NULL,                               -- Optional human-oriented ID
  --
  --      Only defined for PIN-protected keys
  --
  ① PINPolicyID  INT          NULL,                               -- Unique ID of associated PIN policy
  PINValue       BLOB (32)    NULL,                               -- The PIN (password) value for the key
  PINTryCount    SMALLINT     NULL,                               -- Decrementated for each error, locking at 0
  --
  --      User-key cryptographic data
  --
  ② CertPath     BLOB         NOT NULL,                          -- Certificate path => Universal key-ID
  ③ PrivateKey   BLOB         NULL,                               -- Matching private key (symmetric key = NULL)
  ③ SecretKey    BLOB         NULL,                               -- [Symmetric key]
  SuppAlgs       TEXT        NULL,                               -- [For symmetric keys only]
  --
  FOREIGN KEY (PINPolicyID) REFERENCES PINPOLICIES (PINPolicyID),
  PRIMARY KEY (KeyID)
);
```

Notes: 1. A PIN policy may govern any number of keys. 2. Using the KeyGen2 object management facilities, all keys are associated with X.509 certificates (CertPath). 3. PrivateKey and SecretKey are either encrypted keys or handles to keys depending on the execution environment

## Java code

Note: This code was slightly retouched for brevity reasons

```
package org.webpki.android.keystore; // Google's Android

import java.io.IOException;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import java.security.cert.X509Certificate;

import org.webpki.keygen2.PassphraseFormats;

/**
 * Base class for the universal keystore. It must be extended by implementation classes
 * that either support asymmetric keys (PKI) or symmetric keys.
 */
abstract class UniversalKeyStore
{
    byte[] private_key_handle;

    byte[] secret_key_handle;

    String[] supported_algorithms; // For symmetric keys only

    X509Certificate[] cert_path; // For asymmetric keys only

    String key_alias;

    boolean cached_pin;

    abstract boolean wantAsymmetricKeys ();

    /**
     * Closes any open key (key handle). This method should only be necessary to call for keys
     * that support PIN caching.
     */
    public void close ()
    {
        private_key_handle = null;
        secret_key_handle = null;
    }

    /**
     * Opens a key (key handle) for cryptographic operations.
     * @param key_alias The internal database name of the key.
     * @param pin A PIN or password value needed for opening the key. For keys that
     * are not PIN or password protected this value should be <code>null</code>.
     * @return <code>true</code> if successful else <code>false</code>.
     * @throws IOException If there are hard errors.
     */
    public boolean open (String key_alias, String pin) throws IOException
    {
        close ();
        this.key_alias = key_alias;
        try
        {
            Connection conn = Support.getDatabaseConnection ();
            String wanted_key = wantAsymmetricKeys () ? "PrivateKey" : "SecretKey";
            String wanted_ext = wantAsymmetricKeys () ? "CertPath" : "SuppAlgs";
            PreparedStatement pstmt = conn.prepareStatement ("SELECT USERKEYS.PINPolicyID IS NOT NULL, " +
                "USERKEYS.PINTryCount, " +
                "USERKEYS.PINValue, " +
                "PINPOLICIES.Format, " +
                "PINPOLICIES.RetryLimit, " +
                "PINPOLICIES.CachingSupp, " +
                "USERKEYS." + wanted_key + ", " +
                "USERKEYS." + wanted_ext + " " +
                "FROM USERKEYS LEFT JOIN PINPOLICIES " +
                "ON USERKEYS.PINPolicyID=PINPOLICIES.PINPolicyID " +
                "WHERE USERKEYS.KeyID=? AND " +
                "USERKEYS." + wanted_key + " IS NOT NULL");

            pstmt.setString (1, key_alias);
            ResultSet rs = pstmt.executeQuery ();
            boolean pin_protected = false;
            int pin_try_count = 0;
            int retry_limit = 0;
            PassphraseFormats format = null;
            byte[] pin_value = null;
            boolean found_key = rs.next (); // Introduced to make DB handle clean-up nicer
        }
    }
}
```

```

if (found_key)
{
/*=====*/
/* If this key has an associated PIN policy object, get it */
/*=====*/
if (pin_protected = rs.getBoolean (1))
{
pin_try_count = rs.getInt (2);
pin_value = rs.getBytes (3);
format = PassphraseFormats.values ()[rs.getInt (4)];
retry_limit = rs.getInt (5);
cached_pin = rs.getBoolean (6);
}

/*=====*/
/* Get/open handle to the actual key data of the selected key */
/*=====*/
if (wantAsymmetricKeys ())
{
private_key_handle = rs.getBytes (7);
cert_path = Support.restoreCertificatePathFromDB (rs.getBytes (8));
}
else
{
secret_key_handle = rs.getBytes (7);
supported_algorithms = Support.tokenVector (rs.getString (8));
}
}
rs.close ();
pstmt.close ();
conn.close ();
if (!found_key)
{
Support.error ("Missing key for \"" + key_alias + "\"");
}

/*=====*/
/* Does this key require a PIN? If not return success status */
/*=====*/
if (!pin_protected)
{
return true;
}

/*=====*/
/* PIN (password) required. Was it given? */
/*=====*/
if (pin == null)
{
Support.error ("PIN=null for \"" + key_alias + "\"");
}

/*=====*/
/* Has this PIN-protected key already locked-up? */
/*=====*/
if (pin_try_count == 0)
{
close ();
Support.error ("Key \"" + key_alias + "\" locked due to bad PINs");
}

/*=====*/
/* Key is available, now check that the PIN is correct */
/*=====*/
if (Support.compare (Support.getEncryptedPassphrase (pin, format), pin_value))
{
/*=====*/
/* PIN OK, but there may be a need to clear earlier errors */
/*=====*/
if (pin_try_count < retry_limit)
{
setPINTryCount (retry_limit);
}

/*=====*/
/* Return success */
/*=====*/
return true;
}
}

/*=====*/
/* Bad PIN, close key and update the PIN error counter */
/*=====*/
close ();
setPINTryCount (--pin_try_count);

```

```

/*=====*/
/* Return [soft] failure */
/*=====*/
return false;
}
catch (SQLException sqle)
{
    Support.error (sqle.getMessage ());
}
return false; // For the compiler only...
}

private void setPINtryCount (int value) throws SQLException
{
    Connection conn = Support.getDatabaseConnection ();
    PreparedStatement pstmt = conn.prepareStatement ("UPDATE USERKEYS SET PINtryCount=? WHERE KeyID=?");
    pstmt.setInt (1, value);
    pstmt.setString (2, key_alias);
    pstmt.executeUpdate ();
    pstmt.close ();
    conn.close ();
}
}

```

## How to Provision a Universal Keystore?

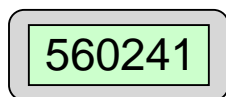
Although not shown in the very cut-down sample, the code above is intended to be augmented by the KeyGen2 universal provisioning system which currently supports:

- PKI
- Symmetric keys including OTP (One Time Password) “seeds”
- Issuer-specific PUKs (Personal Unlock Keys) and associated policies
- Issuer-specific PIN policies as well as preset PINs
- Property bags associated with provisioned keys
- Platform “negotiation” allowing for example controlled migration from RSA to ECC keys
- Downloaded algorithm code for use with a provisioned key
- Information Cards (formerly Microsoft CardSpace®)
- A generic extension mechanism allowing data and corresponding applications to be added (and discovered during platform negotiation) without changing the provisioning protocol or its implementation. This is enabled by the use a URI-based registry scheme and opaque extension “blobs”

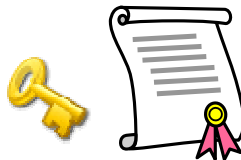
To facilitate life-cycle handling, provisioned objects can be remotely managed in a secure fashion. To simply the integration with the web, KeyGen2 is designed as an extension to Internet browsers.

## Rationale for Universal Keystores

It seems that the total cost for integrating cryptographic support in a platform could potentially be reduced by the code reuse typically offered by universal keystores. Although it is sometimes claimed that you only need a single authentication mechanism, this has in practice proved to be wrong; in the EU the majority of on-line banks use OTP since it works on any computer, while for example the US army has selected PKI. By using a universal keystore, a platform may fit *any* users’ need!



OTP



PKI



Information Cards