Godfrey Nolan

# Bulletproof Android™

## Practical Advice for Building Secure Apps

FREE SAMPLE CHAPTER

# Bulletproof Android™

# Bulletproof Android™

## Practical Advice for Building Secure Apps

Godfrey Nolan

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

*Library of Congress Cataloging-in-Publication Data*

Nolan, Godfrey.
  Bulletproof Android : practical advice for building secure apps / Godfrey Nolan.
    pages cm
  Includes index.
  ISBN 978-0-13-399332-5 (pbk. : alk. paper)
 1. Android (Electronic resource) 2. Application software—Development.
3. Computer security. I. Title.
  QA76.774.A53N654 2014
  005.8—dc23
2014039900



ISBN-13: 978-0-13-399332-5
ISBN-10: 0-13-399332-9

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2014

**Editor-in-Chief**
Mark L. Taub

**Executive Editor**
Laura Lewin

**Senior Development Editor**
Chris Zahn

**Managing Editor**
John Fuller

**Full-Service Production Manager**
Julie B. Nahil

**Project Editor**
Eclipse Publishing Services

**Copy Editor**
Diane Freed

**Indexer**
Jack Lewis

**Proofreader**
Melissa Panagos

**Technical Reviewers**
Matt Insko
Dave Truxall

**Editorial Assistant**
Olivia Basegio

**Cover Designer**
Chuti Prasertsith

**Compositor**
Eclipse Publishing Services

*This book is dedicated to my son and daughter, Rory and Dayna,*
*for making me laugh so much over the years.*
*I'm hoping you too will get to write your own books and plays,*
*and have the pleasure of one day dedicating them to your own kids.*

# Contents at a Glance

# Contents

# Preface

Why another Android security book? Right now I know of a half dozen books or so about hacking Android. I personally wrote one a few years ago called *Decompiling Android*. In the world of hacking we use the term *white hat* for someone who is trying to improve the security of a system and *black hat* for someone who is trying to exploit the weaknesses of a system. In my opinion, most of the existing Android hacking books are either black hat books or they tread the line between white hat and black hat. Sometimes they benefit a black hat hacker and sometimes the information is useful for someone who wants to write a more secure app. Black hat books are still a great resource for understanding how to secure your app, but the focus is on how to attack rather than how to protect an app.

## What This Book Is About

This book is firmly in the white hat category. It is an Android security book for developers, for managers, and for security professionals who want to write more secure Android apps. It uses examples from the many hundreds of Android apps that we (the company I run) have audited over the past three years, and it uses real-world examples of what works and doesn't work from a security perspective. In each chapter we'll look at some examples of how naive coding practices expose apps and how other developers have found more secure solutions.

This book is also written to complement the *Android Security Essentials LiveLessons* video that covers the OWASP (Open Web Application Security Project) Mobile Top 10 Risks in detail. The OWASP Mobile Top 10 is the de facto standard for Android security. And because all security projects are a moving target, the book uses the latest OWASP Mobile Top 10 that has been updated since the LiveLessons video first appeared.

## What This Book Is Not About

If you own an Android phone you're probably worried about apps with hidden malware, or what permissions you should or shouldn't accept. We won't be covering those issues as the focus of the book is on Android developers who want to write more secure Android apps, not someone who owns an Android phone. What's more, we're not going to discuss how to root your phone because that really doesn't have much to do with writing secure code. We will touch on its implications for secure apps, but we won't be showing you how to root your phone. From a developer's perspective, that's why you have an emulator.

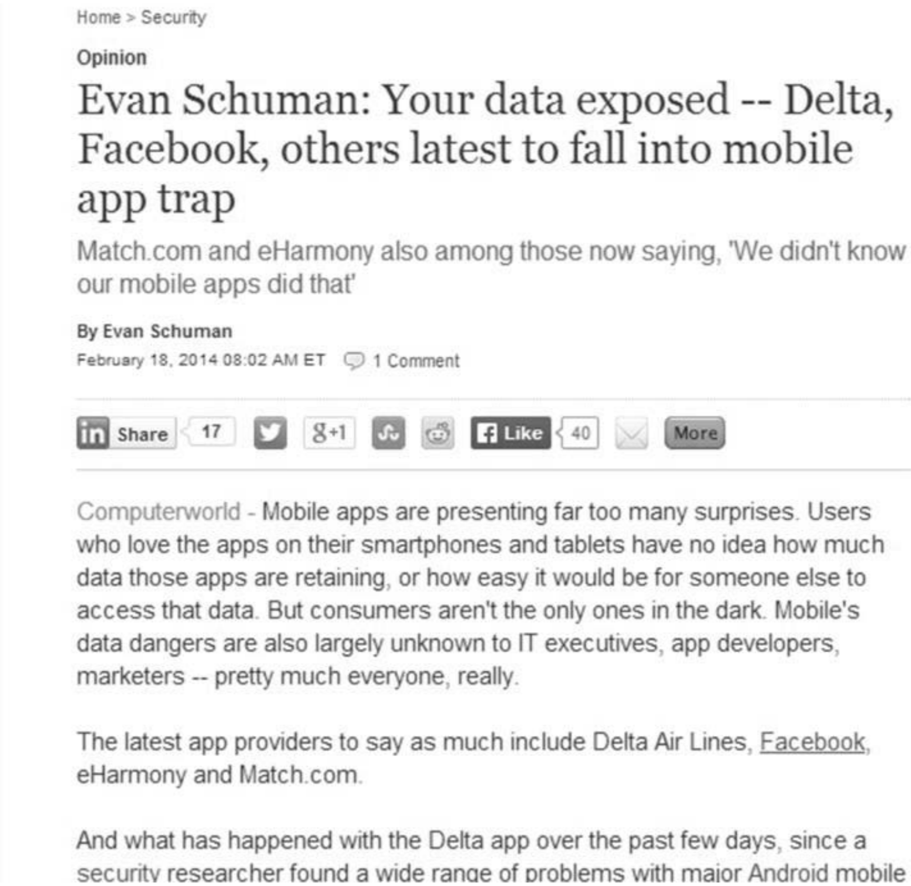
## Why Care?

Over the past two or three years we've downloaded a large number of Android APKs and examined them for any security holes. We've uncovered a wide range of security issues; see Figure P-1 for some examples. These generally fall into the following categories:

1. Keys or API information hard coded in the app (static information)
2. Usernames and passwords and other credentials that are stored insecurely (dynamic information)
3. Sensitive data sent insecurely across the network to a back-end server
4. Third-party libraries collecting and transmitting back to base ad hoc information that they don't need to perform their job
5. Test data or other extraneous information stored in the production APK

It's customary to notify companies that their apps have security issues and are leaking information before releasing the information to the press. This gives the developers some time to fix it and release an update before it goes public. Many times in the past when we contacted the developers responsible for the security issues, we found that security really isn't on their radar as something to worry about. If you're developing mobile apps, then security needs to become part of your development process.

This book comes from what we've seen in our audits of different Android apps. The aim here is to provide you with a book of security anti-patterns where you can see other people's mistakes and hopefully not repeat (m)any of them, thereby keeping your users more secure than your competition.



Home > Security

Opinion

# Evan Schuman: Your data exposed -- Delta, Facebook, others latest to fall into mobile app trap

Match.com and eHarmony also among those now saying, 'We didn't know our mobile apps did that'

By Evan Schuman

February 18, 2014 08:02 AM ET 1 Comment

Share 17 g+1 Like 40 More

Computerworld - Mobile apps are presenting far too many surprises. Users who love the apps on their smartphones and tablets have no idea how much data those apps are retaining, or how easy it would be for someone else to access that data. But consumers aren't the only ones in the dark. Mobile's data dangers are also largely unknown to IT executives, app developers, marketers -- pretty much everyone, really.

The latest app providers to say as much include Delta Air Lines, Facebook, eHarmony and Match.com.

And what has happened with the Delta app over the past few days, since a security researcher found a wide range of problems with major Android mobile

Figure P-1 Dating app insecurity

## What This Book Covers

Here is a breakdown of the book by chapter.

### Chapter 1: Android Security Issues

Chapter 1 is an introduction to the security issues on the Android platform. We'll show how to decompile an Android APK and look at some of the industry standard guidelines for securing the Android platform.

### Chapter 2: Protecting Your Code

In Chapter 2, we'll look at how to download and reverse engineer an Android APK back into Java source in more detail. We'll also cover how to best protect your code using different types of obfuscation tools and techniques that we've encountered during our audits. We'll look at the implications of being able to disassemble your code into bytecode. And we'll show how you can use the NDK to hide your algorithms and business rules.

### Chapter 3: Authentication

Providing a secure login mechanism for your mobile users is harder than on the Web. The trend with mobile devices is to make things as easy as possible for the user. Mobile keyboards are small, so it's unlikely that someone is going to enter more than six characters to log in to an app. But if you make it too easy to log in to your app, then you run the risk of unauthorized users gaining access to sensitive data by going around your authentication. In Chapter 3 we'll look at how some of the authentication mechanisms in our audits have failed, and we're also going to look at what developers have been using to log in to mobile apps that have been a lot more effective.

### Chapter 4: Network Communication

In modern browsers, if you connect via secure HTTP, or HTTPS over a secure sockets layer, you'll get a little green lock, or a gold one depending on your browser, to indicate that you're in a secure encrypted transaction. Developers pay a Certificate Authority (CA) to make sure that they are who they say they are. And if you happen to come across a site that isn't a valid site, your web browser will alert you pretty quickly that something is wrong. Unfortunately, there isn't anything similar in mobile computing—there is no lock or key to comfort the user that any network communication is encrypted.

In this chapter we'll first take a look at how to send information securely across the network using SSL. In the second part of the chapter we'll look at how hackers might perform a man-in-the-middle attack using an SSL Proxy that intercepts the communication and sees whether it's really secure.

## Chapter 5: Android Databases

One of the most basic questions about Android security and mobile security in general is, "What information should you store on a device, and where can you store it securely?" Ideally, you would not store or cache anything on the device. But if someone doesn't have any mobile service—for example, when on an airplane without wi-fi—then you're going to cause some frustration if this person can't log into the app for a number of hours. In this chapter we'll talk about where you can store data and how using the wrong permissions can allow other apps to read your data. Finally, we'll explain how to write data securely to an SD card as well as a SQLite database.

## Chapter 6: Web Server Attacks

Most mobile apps that do real work will in some way connect to a back-end web server. If the communication is via a web service, this can either be via SOAP or, more commonly, by using a REST web service. In this chapter it's a case of what's old is new again. We'll explore how the same security best practices that have applied to web servers for the past 20 years apply to web servers used in mobile apps. We'll also look at how we can use logins from other website break-ins to help secure our authentication.

## Chapter 7: Third-Party Library Integration

Data leakage from third-party apps is perhaps a less obvious way that someone can recover a user's information from your app. In this chapter we'll explain the meaning behind side channel data leakage and learn how to track what information is being passed by your app to other services, with or without your knowledge.

## Chapter 8: Device Security

Running your APK on different versions of Android can have different security problems. In this chapter we'll look at how Android device fragmentation needs to be considered when you're writing a secure app. Different environments have different requirements: Corporations have different requirements than individuals, health care needs HIPAA compliance, and government work probably means that your Android phone needs to be FIPS compliant. In this chapter we'll also look at how Samsung Knox and SELinux or SEAndroid are being used to make your device more secure.

## Chapter 9: The Future

There aren't many certainties about where Android security is going. But in Chapter 9 we're going to look into the crystal ball: Using Android L as well as some open source ideas, we'll do our best to predict what future versions of Android will provide from a security perspective. This way, you'll know what existing security challenges will be solved and what new challenges lie ahead. We'll also look at how Android attacks are likely to get more sophisticated in the near future.

# Tools

There are lots of tools that we'll be using again and again throughout this book. Most of them are listed here for convenience.

- 010, a hex editor that includes a template for disassembling classes.dex files. 010 does a great job of parsing the classes.dex file (see Figure P-2). It can be found at www.sweetscape.com/010editor/.
- Abe, the Android Backup Extractor. It is used to convert an Android backup into a tar format so that it can be unzipped. It's available from https://github.com/nelenkov/android-backup-extractor.
- adb, the Android debug bridge. It comes as part of the Android SDK.
- apktool, a collection of tools. It includes Smali and Baksmali as well as AXMLPrinter2.
- AXMLPrinter2, which converts the compressed AndroidManifest.xml in an APK back into a readable format. It's available at https://code.google.com/p/android4me/downloads/list.

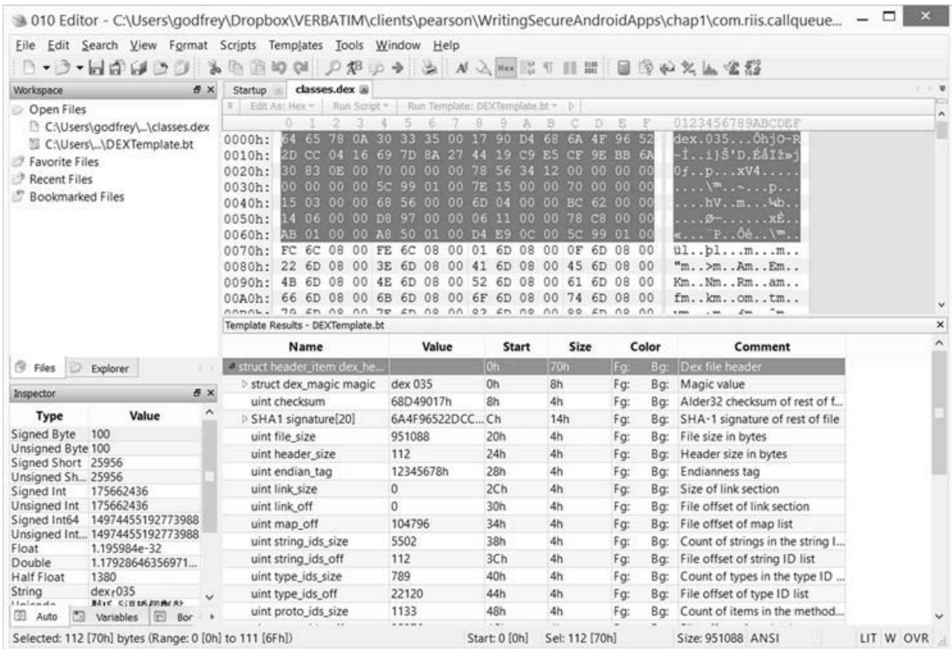



Figure P-2 010 Editor parsing classes.dex file

- Baksmali and Smali, the Android disassembler and assembler. You can find them at https://code.google.com/p/smali/ or as part of apk-tool.
- Charles Proxy, a tool for testing for man-in-the-middle attacks. It's available from http://www.charlesproxy.com/.
- Dedexer, a classes.dex dump file. Written by Gabor Paller in Hungary, it's available from http://dedexer.sourceforge.net/.
- dex2jar, which converts APKs to Java jar files for decompilation. You can find it at https://code.google.com/p/dex2jar/.
- Drozer, an attack tool for Android apps. It's available from https://www.mwrinfosecurity.com/products/drozer/.
- JD-GUI, one of many Java decompilers. You can find it at http://jd.benow.ca/.
- Jadx, one of a new breed of Android decompilers. It's available at https://github.com/skylot/jadx.
- Keyczar, which we use for our public/private key encryption. You can download it from http://keyczar.org.
- Lint, which comes with the Android SDK.
- ProGuard and DexGuard, which are obfuscators. ProGuard ships with the Android SDK, and DexGuard is available at www.saikoa.com/.
- sqlitebrowser, a GUI for SQLite databases. It's available from http://sqlitebrowser.org/.

*This page intentionally left blank*

# Acknowledgments

**Laura Lewin**—I lost count of the number of times Laura hounded me on items that were due or, more often than not, overdue. I sincerely appreciate your patience.

**David Truxall and Matt Insko**—Thanks to my two technical reviewers. I've worked with good reviewers and bad reviewers in the past. The better ones try the code, make suggestions for things you missed, and help get you to the finish line without losing your mind. Dave and Matt are the best.

**Cameron Beyer and Paul Moon**—Thanks for your help with the coding, especially when I wasn't very specific about what I was trying to do. ☺

**Chris Zahn**—Thanks for the editing. Your quality and speed are amazing.

*This page intentionally left blank*

# About the Author

**Godfrey Nolan** is founder and president of the mobile and web development company RIIS LLC based in Troy, Michigan, and Belfast, Northern Ireland. This is his fourth book. He has had a healthy obsession with reverse engineering bytecode since he wrote "Decompile Once, Run Anywhere," which first appeared in *Web Techniques* magazine way back in September 1997. Godfrey is originally from Dublin, Ireland.

*This page intentionally left blank*

*This page intentionally left blank*

# 5

# Android Databases

In Android development, when we say "databases" we primarily mean SQLite and all of its variants. These are typically small databases used to store or cache user information locally on the device. It would be fair to say that databases and shared preferences contain the bulk of an application's dynamic data that is stored on a phone. In this chapter we're going to look at how developers have used SQLite and, more importantly, how they have tried to secure that data in progressively more secure ways so you don't make the same mistakes.

## Android Database Security Issues

Android databases are typically used to cache application data so that it can be retrieved more quickly than doing a web service call to a back-end database server across the Internet. Every app will have its own databases folder. So if the app's package name is com.riis.sqlite3, then you can find all its databases in the /data/data/com.riis.sqlite3 /databases folder. You can see this in Figure 5-1 where we're doing an `adb shell` command to get us a list of the files in the database folder.

Android databases are not a good place to store sensitive information. As we'll see later in the chapter, it is all too easy for someone to do a backup command and quickly find what you're trying to hide.

```
C:\Users\Admin>adb shell ls /data/data/com.riis.sqlite3/databases
tasks.db
tasks.db-journal

C:\Users\Admin>
```

Figure 5-1 SQLite databases on your phone

However, many apps ignore this issue because using SQLite is so convenient for storing data. Facebook keeps a lot of its user information in SQLite databases, which they have openly admitted is for performance reasons. Figure 5-2 shows a Facebook database that's been taken off an Android device using the `adb backup` command. The "text" column in the threads.db database shows all the thread messages that a user has sent and received in Facebook via the website as well as on the mobile app.

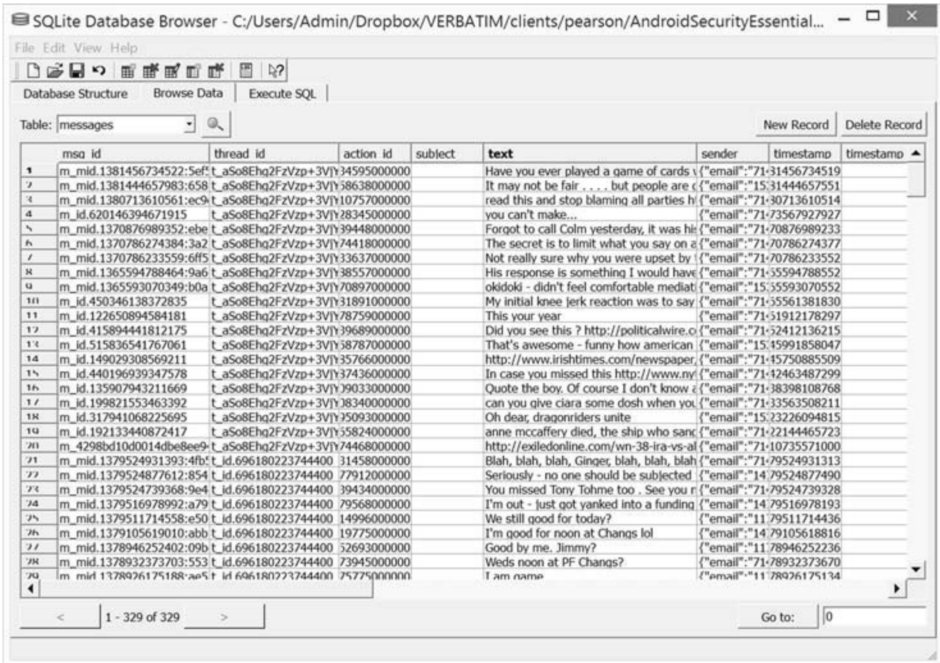



Figure 5-2 Viewing SQLite databases on your PC using SQLitebrowser

# SQLite

SQLite is a fully functional database. It has many of the features you would expect in a modern database, such as indexes and stored procedures. You can even do an explain plan for optimizing your queries to find out exactly where your SQL code is spending most of its time.

Any and all of your runtime app information—which includes all the shared preference files and databases—can be backed up by anyone with access to your phone using a USB cable. Because of an oversight at Google, no one running Android after version 4.0 even needs root access—they just need physical access to the phone. To be fair, I think this was an intentional feature, not an oversight. The feature just has significant unintended consequences.

> **Note**
>
> Section §164.312 of the HIPAA standards says the following:
>
> > (a)(1) Standard: Access control. Implement technical policies and procedures for electronic information systems that maintain electronic protected health information to allow access only to those persons or software programs that have been granted access rights as specified in §164.308(a)(4).

Putting any personal health information unencrypted in a SQLite database is not HIPAA compliant because we cannot be sure that only persons that have been granted access have access to the databases. Under most circumstances encrypted information in a SQLite database is also not compliant. A quick way to check whether you have an issue is to put the phone in Airplane mode and then see whether there is any sensitive information, or what is known as Protected Health Information (PHI), being displayed by the application. This will typically tell you that the information is either not encrypted or the encryption key is somewhere on the phone, neither of which is HIPAA compliant.

## Backing Up the Database Using `adb`

Let's look at how to write to a SQLite application and how someone can pull the database off the phone. To begin, we need to add a SQLite database to the Android HelloWorld app. Listing 5-1 shows how to add a SQLite database to your Android app.

Listing 5-1 **Adding SQLite to your code**

```
package com.riis.sqlite3;

import java.io.File;
import android.os.Bundle;
import android.app.Activity;
import android.database.sqlite.SQLiteDatabase;                    // line 7
public class MainActivity extends Activity {

@Override
protected void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.activity _ main);

          InitializeSQLite3();                                     // line 16

     }

     private void InitializeSQLite3() {

          File databaseFile = getDatabasePath("names.db");
          databaseFile.mkdirs();
          databaseFile.delete();
```

```
        SQLiteDatabase database =                                          // line 26
            SQLiteDatabase.openOrCreateDatabase(databaseFile, null);

        database.execSQL("create table user(id integer primary key autoincrement,
        " +
                    "first text not null, last text not null, " + //    line 28
                    "username text not null,  password text not null)");

        database.execSQL("insert into user(first,last,username, password) " +
                    "values('Bertie','Ahern','bahern','celia123')");
                    // line 31
    }

}
```

To add SQLite to your application, import the library (see line 7), initialize the SQLite database (see line 26), and then create your tables (see line 28) as well as add any initial data (see line 31).

In the example shown we are adding just a single row of data to the database. We are adding a first name, a last name, and a corresponding username and password to our database.

We can now recover the database using the following steps on a compatible phone:

1. Compile the code, push it to your phone or emulator, and make sure it executes.
2. Run the app.
3. Back up the databases using the following command:

   ```
   adb backup com.riis.sqlite3
   ```
4. If all is working, device will respond with "Now unlock your device and confirm the backup operation."
5. On the device or emulator, click Back up my data to enable it to be backed up (see Figure 5-3).
6. The backup file is a tar file with a custom header. We need to download the Android Backup Extractor from https://github.com/nelenkov/android-backup-extractor to get it into a tar format.
7. Convert your backup.ab file using the following command:

   ```
   java -jar abe.jar unpack backup.ab backup.tar
   ```
8. Uncompress your tar file using `tar -xvf` or 7zip if you're on a Windows machine.


Figure 5-3 Back up my data

9. Change directory to apps/com.riis.sqlite3/db, where you can now find your names.db database.
10. Open names.db in sqlitebrowser from http://sqlitebrowser.org (see Figure 5-4). As you see, the user information is in cleartext.

If you don't have sqlitebrowser, you can always gain access to the sqlite database from the command line (refer ahead to Figure 5-6).

Note that if your backup.ab file is empty, then it's likely that you have used the wrong package name. For commercial apps the best way to find the correct package name is to look at the target ID in the app's Google Play URL (see Figure 5-5 for Facebook's target ID). In this example, to back up the Facebook database you would type the following:

```
adb backup com.facebook.katana
```

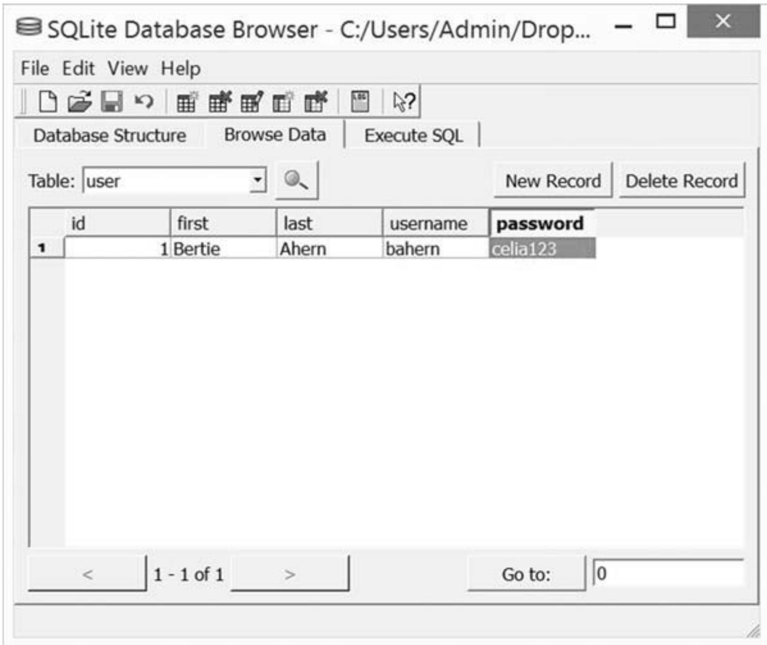



Figure 5-4 View the backup database data using the SQLite browser.

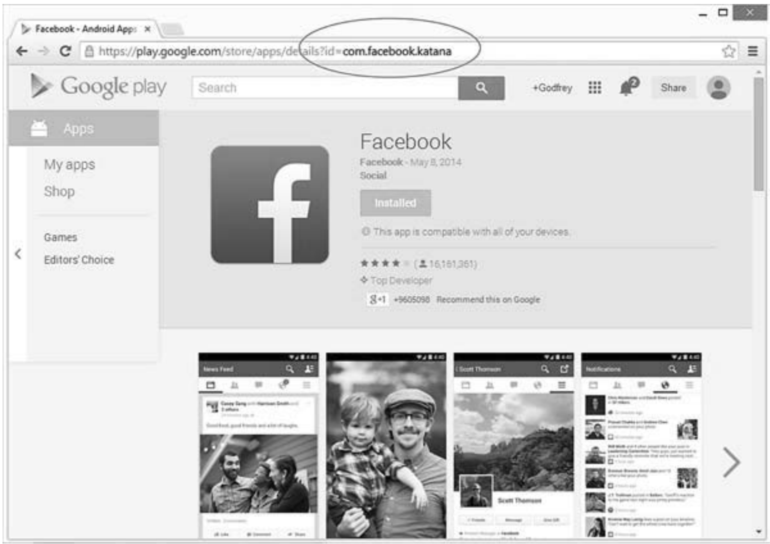



Figure 5-5 Finding an App's package name

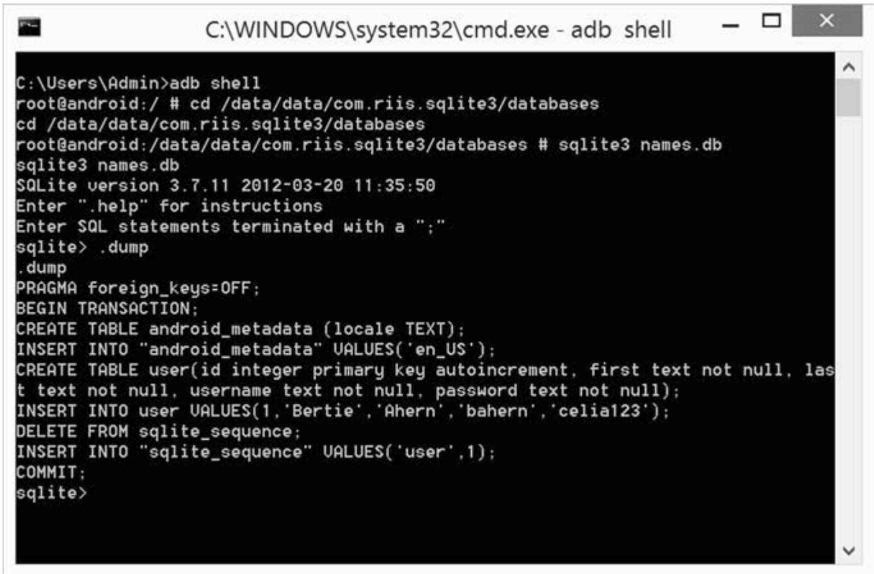



Figure 5-6 Viewing the backup database data from command line SQLite

## Disabling Backup

If anyone with access to your phone can back it up, then we'll need some way to hide the information if we're going to be HIPAA compliant.

We can start with something simple by disabling backups using the `allowBackup` attribute in the Android Manifest file. By default this is set to `true`. Changing it to `false`, as in Listing 5-2, will stop the `adb` backup command working for any phone, even for a full system backup.

However, it would be a mistake to solely rely on this, as a rooted phone has access to databases and can still remove them from the phone via Unix commands. Figure 5-6 shows how someone can shell onto the phone, `cd` to the databases directory, and then dump the database table to view the data.

`adb pull` can also be used to get the database off the phone. But you may also need to run a `chmod 777` <filename> to fully open the file's permissions before you can retrieve them.

Listing 5-2 **Disabling backup**

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.riis.sqlite3"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="16" />
    <application
        android:allowBackup="false"
        android:icon="@drawable/ic _ launcher"
        android:label="@string/app _ name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.riis.sqlite3.MainActivity"
            android:label="@string/app _ name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

## SQLCipher

We've seen that it doesn't take a degree in computer science to gain access to an APK's source code, the static information, and an app's backup data, the dynamic information. Ideally you wouldn't store any important customer information locally, but this isn't always an option. But, as we've seen, any data that is stored in cleartext can be found easily. So if you do have to store any sensitive data, it is important to encrypt the data in either shared preferences or in a database—or store it somewhere else.

> **Note**
>
> Apps using SQLCipher are restricted from export/distribution in certain countries and require additional export registration with the US government if the app is distributed outside the United States because SQLCipher contains strong encryption. The Play Store asks about US export law compliance when you publish an app. This can be a gotcha when using SQLCipher for developers who are unaware. The Android OS encryption functionality is already compliant via Google's filings, which no doubt explains why Android ships with a cut-down version of Bouncy Castle that does not use strong encryption.

One of the more promising ways to store data securely in a database is using SQLCipher, which is an open source library used in conjunction with SQLite. SQLCipher can be downloaded from www.sqlcipher.net.

In Listing 5-3 we show how to use SQLCipher to encrypt the data in the database. First, add the sqlcipher.jar, commons-codec.jar and guava-r09.jar libraries, which can also be found on the sqlcipher.net website. Then change the import statement (line 7) to import SQLCipher, add a new `loadLibs` command (line 21) and, as you can see, the `openOrCreateDatabase` now takes a password (line 27).

**Listing 5-3 Adding SQLCipher to your SQLite code**

```
package com.riis.sqlite3;

import java.io.File;

import android.os.Bundle;
import android.app.Activity;
import net.sqlcipher.database.SQLiteDatabase;                         // line 7

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity _ main);

         InitializeSQLite3();

    }

    private void InitializeSQLite3() {
        SQLiteDatabase.loadLibs(this);                                // line 21

        File databaseFile = getDatabasePath("names.db");
        databaseFile.mkdirs();
        databaseFile.delete();
        SQLiteDatabase database =                                     // line 27
        SQLiteDatabase.openOrCreateDatabase(databaseFile,"pass123",
        null);

        database.execSQL("create table user(id integer primary key autoincrement,
        " +
                    "first text not null, last text not null, " +
                    "username text not null, password text not null)");

        database.execSQL("insert into user(first,last,username, password) " +
                   "values('Bertie','Ahern','bahern','celia123')");

    }
}
```

Compile and push the app to the phone. Repeat the earlier steps to back up the database onto our computer. You will probably notice that it takes noticeably longer to push the app to the phone, as well as to back it up. This is because of the size of the added libraries.

Again, try to open it in sqlitebrowser or by using the SQLite command line tool. This time the database won't open because it's encrypted with the key pass123.

The best way to open the database is to use the sqlite3 command line tool that comes with SQLCipher. A new step is required whereby we need to tell the database what the key is before it will allow us to do any SQL queries on the tables.

```
sqlite> PRAGMA key='pass123';
```

Figure 5-7 shows how to view the database using the new password.

You may also encounter databases that were created with earlier versions of the SQLCipher libraries. These can be opened using the following PRAGMA command after the PRAGMA key command.

```
sqlite> PRAGMA key='pass123';
sqlite> PRAGMA kdf_iter = 4000;
```

This tells the sqlite tool that the key definition file has a lower iteration count than the current version.

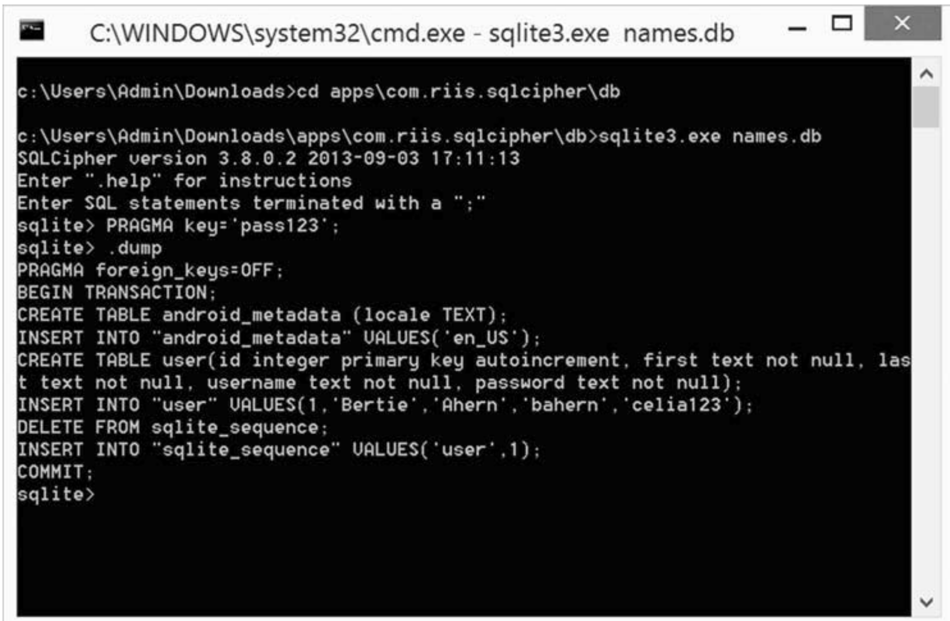



Figure 5-7 Viewing an encrypted database from command line SQLite

## Finding the Key

Now that SQLCipher has encrypted the database, our security problem shifts to "Where can we hide the key?" If we can find the key, then we're going to be able to open the database, just like we did in Chapter 2. We can take the following steps to pull the APK off the device.

1. The APK is in the /data/app folder on the phone. It will also be called the same package name we used in the `adb backup` command but with `-1.apk` appended. The complete command to get the APK off the phone is the following:

   ```
   adb pull /data/app/com.riis.sqlcipher-1.apk
   ```

2. Convert the APK back into a jar file using the `dex2jar` command:

   ```
   dex2jar com.riis.sqlcipher-1.apk
   ```

3. We can now view the source using a Java decompiler, in this case JD-GUI. Figure 5-8 shows the code for the MainActivity.java file and clearly shows that the password is pass123.

In the next section we'll look at our options for hiding the key.

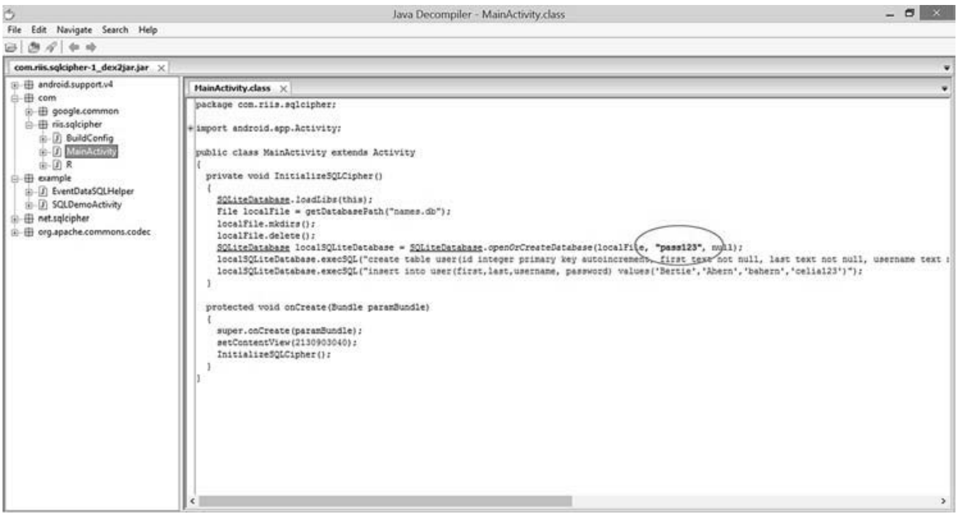



Figure 5-8 Viewing the SQLCipher key using JD-GUI

# Hiding the Key

One of the most fundamental decisions that you're going to face as a mobile developer is what encryption to use to hide sensitive information and whether you're going to leave the information on the phone or not.

In this section we're going to look at a number of different ways that other developers have tried to solve this problem. These examples come from real-world Android apps that we've audited over the years. They each get progressively better at hiding an encryption key for the database itself or for fields in the database, such as the password.

Security on Android is almost always a battle between security and ease of use. App developers want to make it easy for people to use, and they don't think it's a good idea to make someone log into the phone multiple times.

And while many of these examples look like very naive implementations, we have the benefit of hindsight and can probably assume that the developers were not aware that someone could gain access to their code and encryption keys so easily. If you're using some sort of symmetrical key encryption where the encrypted data, as well as the encrypted key, are on the phone, then you're leaving yourself open to attack.

## Ask Each Time

Possibly the safest way to encrypt your database is to ask for the key each time, either using a PIN code or a password. The first time the user opens the app they're asked for the key, which is then used to encrypt the database.

If the user wants to access any data on the app, then the next time they use the app they have to remember their key and reenter it. The key is stored in the user's head and not on your phone.

The downside of this is that the user has to log in to the phone each time they open your app. And depending on the key size it may also be open to a brute-force attack. Certainly a four-digit pin code is not very secure.

Listing 5-4 shows an example of how to use a login password to encrypt the database. The password is captured as the user is logging in on line 31; it's then passed to `initializeSQLCipher` as a string on line 35 and used as the SQLCipher key when we open the database on line 45.

Listing 5-4 **Using a Login password to encrypt the database**

```
public class LoginActivity extends Activity {

    private Button loginButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login _ screen);
        initializeViews();
        bindListenersToViews();

    }

    private void initializeViews() {
        loginButton = (Button) findViewById(R.id.login _ button);

    }

    private void bindListenersToViews() {
        loginButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                    loginToApp();
            }
    });
    }

    private void loginToApp() {
        EditText usernameField = (EditText) findViewById(R.id.username _ field);
        EditText passwordField =         // line 31
(EditText) findViewById(R.id.password _ field);
        EditText emailField = (EditText) findViewById(R.id.email _ field);


        InitializeSQLCipher(passwordField.getText().toString());         // line 35

    }

    private void InitializeSQLCipher(String pwd) {
        SQLiteDatabase.loadLibs(this);
        File databaseFile = getDatabasePath("names.db");
        databaseFile.mkdirs();
        databaseFile.delete();

        SQLiteDatabase database =                                    // line 45
SQLiteDatabase.openOrCreateDatabase(databaseFile, pwd, null);

        database.execSQL("create table user(id integer primary key autoincrement,
        " +
                    "first text not null, last text not null, " +
                    "username text not null,  password text not null)");

        database.execSQL("insert into user(first,last,username, password) " +
                    "values('Bertie','Ahern','bahern','celia123')");
      }
}
```

## Shared Preferences

The next implementation is to hide the key in the shared preferences and then load it each time the app is opened. There are two variations on this theme. A typical app will ask the user to encrypt the app the first time and save the key in the shared preferences. Listing 5-5 shows how to write and load our encryption key from a shared preferences file.

Listing 5-5 **Storing passwords in the shared preferences file**

```
private void saveLastSuccessfulCreds() {
    String username =
((EditText) findViewById(R.id.username_field)).getText().toString();
    String password =                                              // line 3
((EditText) findViewById(R.id.password_field)).getText().toString();

    SharedPreferences.Editor editor = sharedPrefs.edit();
    editor.putString(SettingsActivity.LAST_USERNAME_KEY, username);
    editor.putString(SettingsActivity.LAST_PASSWORD_KEY, password);    // line 7
    editor.commit();
}

private void loadLastSuccessfulCreds() {
    String lastUsername =
sharedPrefs.getString(SettingsActivity.LAST_USERNAME_KEY, "");
    String lastPassword =                                         // line 13
sharedPrefs.getString(SettingsActivity.LAST_PASSWORD_KEY, "");

     ((EditText) findViewById(R.id.username_field)).setText(lastUsername);
     ((EditText) findViewById(R.id.password_field)).setText(lastPassword); //line 16
}
```

The `adb backup` command will not only recover the databases, it will also recover the shared preferences files. Figure 5-9 shows a screenshot of someone viewing a shared preferences file on the phone itself.

Alternatively, the app can load an app-specific username and password when the app is first opened. Android will load data from the resources/xml folder and store it in shared preferences. Listing 5-6 shows how to load the key from the resources folder.

Listing 5-6 **Loading the SQLCipher key from the resources folder**

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >

<EditTextPreference
    android:defaultValue="pass1234"
    android:key="myKey" />

</PreferenceScreen>
```

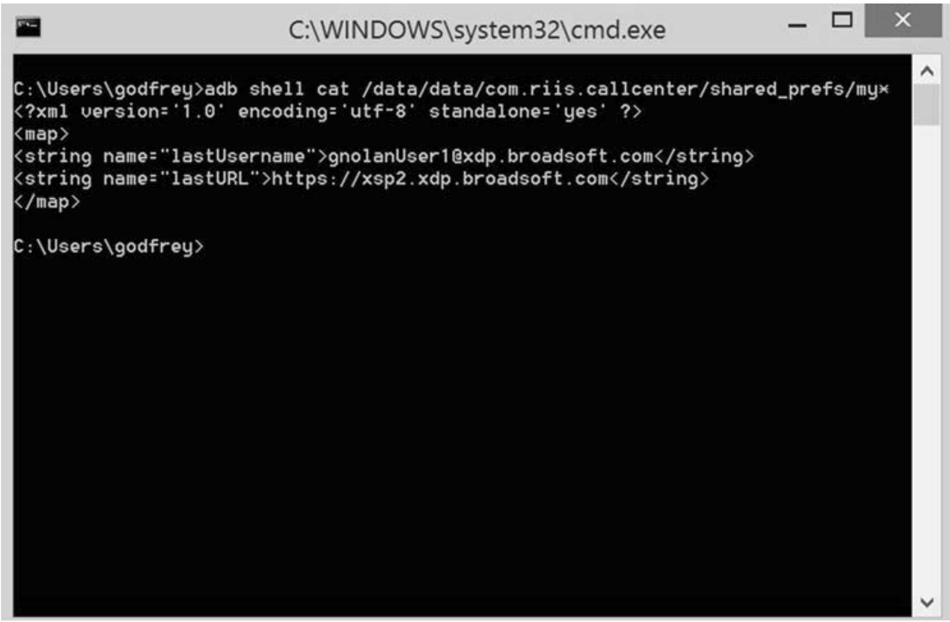



Figure 5-9 Viewing shared preferences files

The advantage of this is that it's very easy to use; it encrypts the database without any user input. The disadvantage is that it's very easy for someone to find the key and decrypt the phones. For example, the `apktool`—available from https://code.google.com/p/android-apktool/—will convert an APK's resources back into xml using the following command:

```
java -jar apktool.jar d com.riis.sqlcipher-1.apk
```

## In the Code

We can see from the SQLCipher code example earlier in Figure 5-8 that we can't simply hard code our key in the SQLCipher class because someone is going to find it when they decompile your APK. If we create a security scale showing level of difficulty—from 1 to 10, where 1 is your kid brother and 10 is a foreign government—then we're close to 1 or 2 in the level of difficulty to reverse engineer an APK to decompile the code.

A couple of years ago, using a single security key for everyone's app was common practice in Android development. More recently, developers have moved to generating the key and making it device-specific using the device's attributes, such as device_id, android_id, and any number of phone-specific attributes such as BUILD ID's, and Build.MODEL and Build.MANUFACTURER. This is then concatenated together and is a unique key for that phone or tablet. Listing 5-7 shows how you might do that. It takes the device's unique Android ID and the Device ID (assuming it's not a tablet) as well as a whole array of phone information. All of this information is concatenated together and converted into an md5 digest or hash value.

So far, so good. It protects the app from any potential targeted malware that would use a decompiled key to attack the app on lots of different phones. However, although the key isn't the same on every device, the algorithm is the same. And it's a small step if the code can be decompiled to figure out how to recreate the recipe for generating the key, so ultimately it's only slightly more secure than using the same key.

Listing 5-7 **Device-specific keys**

```
android_id =
     Secure.getString(getBaseContext().getContentResolver(),Secure.ANDROID _ ID);
tManager = (TelephonyManager) this.getSystemService(Context.TELEPHONY _ SERVICE);
device _ id = tManager.getDeviceId();


String str1 = Build.BOARD + Build.BRAND + Build.CPU _ ABI + Build.DEVICE +
     Build.DISPLAY + Build.FINGERPRINT + Build.HOST + Build.ID + Build.MANUFACTURER
+
     Build.MODEL + Build.PRODUCT + Build.TAGS + Build.TYPE + Build.USER;
String key2 = md5(str1 + device _ id + android _ id);
```

## In the NDK

If the Java code in Android can be reverse engineered so easily, then it makes sense to write it in some other language that isn't so easily decompiled. Some developers hide their keys in C++ using the Native Developer Kit (NDK). The NDK enables developers to write code as a C++ library. This can be useful if you want to try to hide any keys in binary code. And, unlike Java code, C++ cannot be decompiled, only disassembled.

Listing 5-8 shows some simple C++ code for returning the "pass123" key to encrypt the database.

Listing 5-8 **Hiding the key in the NDK**

```
#include <string.h>
#include <jni.h>

jstring Java _ com _ riis _ sqlndk _ MainActivity _ invokeNativeFunction(JNIEnv* env,
jobject javaThis) {
  return (*env)->NewStringUTF(env, "pass123");
}
```

Listing 5-9 shows the Android code to call the NDK method correctly. Line 11 does the JNI library call, the function is defined on line 14, and then we call the function that returns the key on line 21. The `sqlndk.c` file needs to be in a `jni` folder. And because it's C++ code, we're going to need a make file.

Listing 5-9 **Calling the NDK code from Android**

```
import java.io.File;

import net.sqlcipher.database.SQLiteDatabase;
import android.os.Bundle;
import android.app.Activity;
import android.app.AlertDialog;

public class MainActivity extends Activity {

    static {
        System.loadLibrary("sqlndk");                                   // line 11
        }

    private native String invokeNativeFunction();                      // line 14

    @Override
    protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_main);

         String sqlkey = invokeNativeFunction();                       // line 21
         new AlertDialog.Builder(this).setMessage(sqlkey).show();

        InitializeSQLCipher(sqlkey);


    }

    private void InitializeSQLCipher(String initKey) {
        SQLiteDatabase.loadLibs(this);
        File databaseFile = getDatabasePath("tasks.db");
        databaseFile.mkdirs();
        databaseFile.delete();
        SQLiteDatabase database =
           SQLiteDatabase.openOrCreateDatabase(databaseFile, initKey, null);
        database.execSQL("create table tasks" +
                  " (id integer primary key autoincrement,title text not null)");
        database.execSQL("insert into tasks(title) values('Placeholder 1')");
    }
}
```

Listing 5-10 shows the corresponding Android.mk file. The C++ code is compiled using the `ndk-build` command that comes with the Android NDK tools. `ndk-build` is run from a `cgywin` command line if you're on Windows.

Listing 5-10 **NDK makefile**

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

# Here we give our module name and source file(s)
LOCAL_MODULE     := sqlndk
LOCAL_SRC_FILES := sqlndk.c

include $(BUILD_SHARED_LIBRARY)
```

But we're not there yet. Even though we can no longer decompile the code, we can disassemble it. Looking at Figure 5-10 you can see where the library, opened up in a hexadecimal editor, shows the key very clearly at the end of the hexidecimal strings in the file.

If you're going to use the NDK, then choose hexadecimal-like text so that it doesn't stand out in a hex editor. We can also take the earlier approach and use some device-specific or app-specific characteristic and generate a unique app key in NDK just like we can in native Android code. Listing 5-11 shows how you can use the app

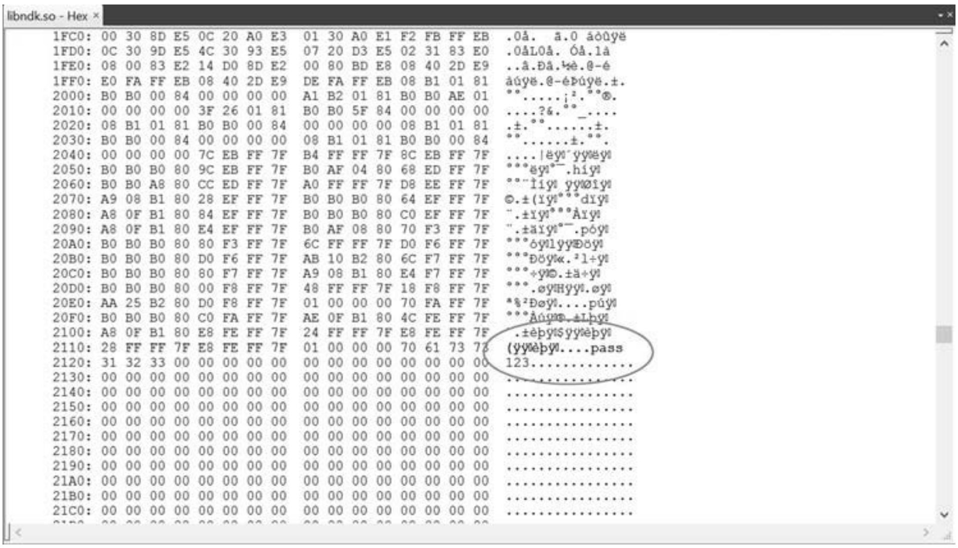



Figure 5-10 Viewing the NDK password

ID as a unique key, which will be different every time the app is installed on a different phone. It uses a function called `getlogin()` to find out the login ID, which in this case is the `app_id`.

Listing 5-11 **Using the App ID for the database key**

```
#include <string.h>
#include <jni.h>
#include <unistd.h>

jstring Java_com_riis_sqlndk_MainActivity_invokeNativeFunction(JNIEnv* env,
jobject javaThis) {

    return (*env)->NewStringUTF(env, (char *)getlogin());

}
```

However, neither of these approaches is ultimately enough to stop someone from reading the binary. But it is a better option to consider if you have no other choice than to put the API or encryption keys on the device. Disassembled code rapidly becomes more difficult to understand as it gets further away from these simple hello-world examples.

### Web Services

The safest option for any type of device is to store the key, or the algorithm for generating your key, remotely and to access it via secure web services. This has already been covered in previous chapters. The disadvantage to this is that the Android device will need to be connected to the Internet when you open the database, which might not be acceptable to the end user.

But the message should be clear by now that any keys stored on the phone are open to being hacked in ways similar to what we've shown in this section. We'll go into more detail in the next chapter about what to do to protect your web server and your web server traffic from prying eyes.

## SQL Injection

SQL injection refers to when the attacker taints the data with a SQL statement. We said earlier that SQLite is a fully functional database, so, just like your SQL Server or MySQL box, it is just as susceptible to SQL injection if you are not careful. SQL injection typically works by adding data to the querystring or adding data in a form field to give the hacker access to the database or unauthorized logins. And while SQL injection is usually something used for attacking a web view or a web service, it can also be an attack on an Activity. Figure 5-11 shows a simple SQL injection example.


Figure 5-11 Classic SQL injection attack

If we look at the `checkLogin` code in Listing 5-12 we can see that the SQL query is passed directly to the database. So if we log in with a username of `' OR 1=1 --'` and password of `test`, the query to SQLite will be the following string:

```
select * from login where USERNAME = '' OR 1=1 --' and PASSWORD = 'test'
```

Listing 5-12 **Login unprotected from SQL injection**

```
public boolean checkLogin(String param1, String param2)
{
    boolean bool = false;

    Cursor cursor = db.rawQuery("select * from login where USERNAME = '" +
    // line 5
        param1 + "' and PASSWORD = '" + param2 + "';", null);

    if (cursor != null) {
        if (cursor.moveToFirst())
             bool = true;
        cursor.close();
    }
    return bool;
}
```

Because of the `OR 1=1` portion of the string and the `--`, which comments out the rest of the SQL query, this will always be a true condition. The result is that the user can log in without needing a real username and password.

To fix this we need to sanitize any user-entered data and assume it can't be trusted. We can do this either by using regular expressions to check that it's what we're expecting—for example, a valid email address—or by using SQL prepared statements. Or better still, we can do both.

To fix our `checkLogin` code we're going to change the SQL to use prepared statements. Listing 5-13 shows a modified `checkLogin`, which now uses prepared statements on line 5. Here the injected SQL becomes a parameter and can no longer cut off the SQL statement.

Listing 5-13 **Protecting code using prepared statements**

```
public boolean checkSecureLogin(String param1, String param2)
  {
      boolean bool = false;

      Cursor cursor = db.rawQuery("select * from login where " +      // line 5
           "USERNAME = ? and PASSWORD = ?", new String[]{param1, param2});

      if (cursor != null) {
           if (cursor.moveToFirst())
                bool = true;
           cursor.close();
      }
      return bool;
  }
```

# Conclusion

In this chapter we've looked at options to make your databases more secure. If you're going to store customer information, we've covered how to use SQLCipher to encrypt the data as well as the various schemes developers have used to hide the key and keep the data safely encrypted.

The only 100 percent secure way to hide any encryption key is to keep it off the phone, and even then you must make sure it's transmitted securely and not cached anywhere. Every other alternative that we looked at had limitations, some more obvious than others. None of these alternatives would be HIPAA compliant. Ask yourself the question, "Would the security of my app be compromised if someone could read my code?" If the answer is yes, then the app is not HIPAA compliant.

*This page intentionally left blank*

# Index

## Numbers



## A

## E

## F



## G



## H

## M



## N

## X