79. An Android Fingerprint Authentication Tutorial

Fingerprint authentication uses the touch sensor built into many Android devices to identify the user and provide access to both the device and application functionality such as in-app payment options. The implementation of fingerprint authentication is a multi-step process which can, at first, seem overwhelming. When broken down into individual steps, however, the process becomes much less complex. In basic terms, fingerprint authentication is primarily a matter of encryption involving a key, a cipher to perform the encryption and a fingerprint manager to handle the authentication process.

This chapter provides both an overview of fingerprint authentication and a detailed, step by step tutorial that demonstrates a practical approach to implementation.

79.1 An Overview of Fingerprint Authentication

There are essentially 10 steps to implementing fingerprint authentication within an Android app. These steps can be summarized as follows:

Request fingerprint authentication permission within the project Manifest file.

1. Verify that the lock screen of the device on which the app is running is protected by a PIN, pattern or password (fingerprints can only be registered on devices on which the lock screen has been secured).
2. Verify that at least one fingerprint has been registered on the device.
3. Create an instance of the FingerprintManager class.
4. Use a Keystore instance to gain access to the Android Keystore container. This is a storage area used for the secure storage of cryptographic keys on Android devices.
5. Generate an encryption key using the KeyGenerator class and store it in the Keystore container.
6. Initialize an instance of the Cipher class using the key generated in step 5.
7. Use the Cipher instance to create a CryptoObject and assign it to the

FingerprintManager instance created in step 4.

8. Call the *authenticate* method of the FingerprintManager instance.
9. Implement methods to handle the callbacks triggered by the authentication process. Provide access to the protected content or functionality on completion of a successful authentication.

Each of the above steps will be covered in greater detail throughout the tutorial outlined in the remainder of this chapter.

79.2 Creating the Fingerprint Authentication Project

Begin this example by launching the Android Studio environment and creating a new project, entering *FingerprintDemo* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 23: Android 6.0 (Marshmallow). Continue through the setup screens, requesting the creation of an Empty Activity named *FingerprintDemoActivity* with a corresponding layout named *activity_fingerprint_demo*.

79.3 Configuring Device Fingerprint Authentication

Fingerprint authentication is only available on devices containing a touch sensor and on which the appropriate configuration steps have been taken to secure the device and enroll at least one fingerprint. For steps on configuring an emulator session to test fingerprint authentication, refer to the chapter entitled *"Using and Configuring the Android Studio AVD Emulator"*.

To configure fingerprint authentication on a physical device begin by opening the Settings app and selecting the *Security & Location* option. Within the Security settings screen, select the *Fingerprint* option. On the resulting information screen click on the *Next* button to proceed to the Fingerprint setup screen. Before fingerprint security can be enabled a backup screen unlocking method (such as a PIN number) must be configured. If the lock screen is not already secured and follow the steps to configure either PIN, pattern or password security.

With the lock screen secured, proceed to the fingerprint detection screen and touch the sensor when prompted to do so (Figure 79-1), repeating the process

to add additional fingerprints if required.

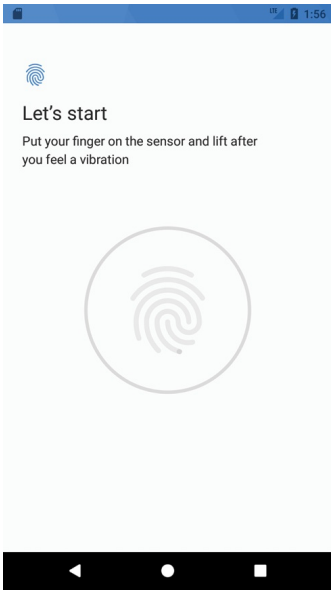

Figure 79-1

79.4 Adding the Fingerprint Permission to the Manifest File

Fingerprint authentication requires that the app request the *USE_FINGERPRINT* permission within the project manifest file. Within the Android Studio Project tool window locate and edit the *app -> manifests -> AndroidManifest.xml* file to add the permission request as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.fingerprintdemo1">

    <uses-permission
        android:name="android.permission.USE_FINGERPRINT" />
.
.
```

79.5 Adding the Fingerprint Icon

Google provides a standard icon (Figure 79-2) which should be displayed whenever an app requests authentication from a user.

Figure 79-2

A copy of this icon is included in the *project_icons* folder of the sample code download available from the following URL:

http://www.ebookfrenzy.com/retail/androidstudio30/index.php

Open the filesystem navigator for your operating system, select the *ic_fp_40px.png* image file and press Ctrl-C (Cmd-C on macOS) to copy the file. Return to Android Studio, right-click on the *app -> res -> drawable* folder and select the *Paste* menu option to add a copy of the image file to the project. When the Copy dialog appears, click on the *OK* button to use the default settings.

79.6 Designing the User Interface

In the interests of keeping the example as simple as possible, the only elements within the user interface will be a TextView and an ImageView. Locate and select the *activity_fingerprint_demo.xml* layout resource file to load it into the Layout Editor tool.

Delete the sample TextView object, drag and drop an ImageView object from the *Images* category of the palette and position it in the center of the layout canvas.

After the ImageView widget has been placed within the layout, the *Resources* dialog will appear. From the left-hand panel of the dialog select the *Drawable* option. Within the main panel, enter *ic_fp* into the search box as illustrated in Figure 79-3 to locate the fingerprint icon. Select the icon from the dialog and click on *OK* to assign it to the ImageView object. Resize the ImageView instance if necessary.

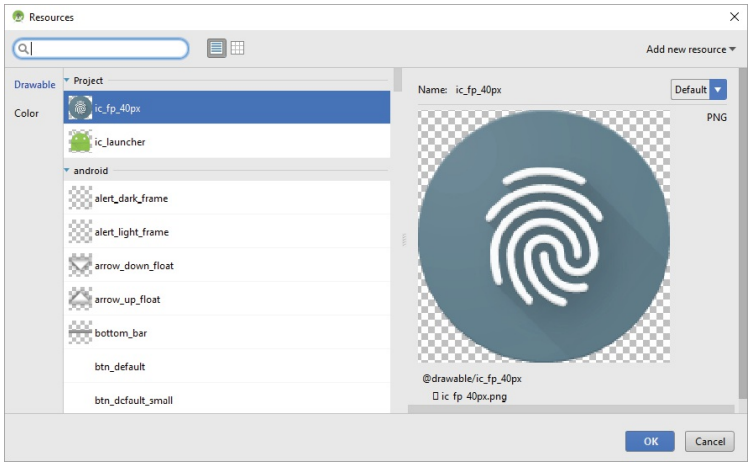

Figure 79-3

Locate the TextView widget from the palette and drag and drop it so that it is positioned in the horizontal center of the layout and beneath the bottom edge of the ImageView object. Using the Attributes tool window, change the text property to "Touch Sensor" and increase the font size to 24sp. Finally, extract the string to a resource named *touch_sensor*.

On completion of the above steps the layout should match that shown in Figure 79-4:

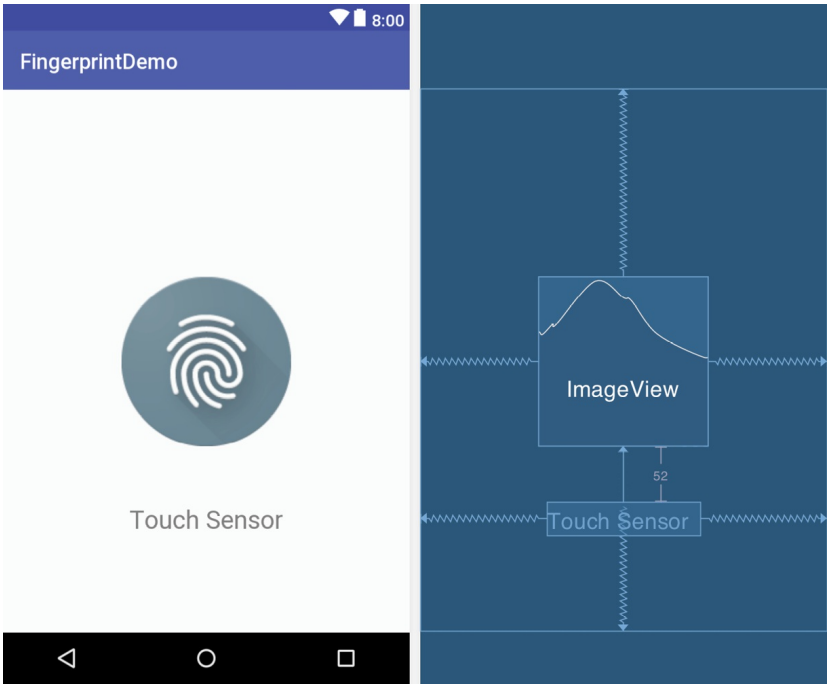

Figure 79-4

79.7 Accessing the Keyguard and Fingerprint Manager Services

Fingerprint authentication makes use of two system services in the form of the KeyguardManager and the FingerprintManager. Edit the *onCreate* method located in the *FingerprintDemoActivity.java* file to obtain references to these two services as follows:

```
package com.ebookfrenzy.fingerprintdemo;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.app.KeyguardManager;
import android.hardware.fingerprint.FingerprintManager;
```

```
public class FingerprintDemoActivity extends AppCompatActivity {

    private FingerprintManager fingerprintManager;
    private KeyguardManager keyguardManager;

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

        if (getManagers()) {

        }
    }

    private Boolean getManagers() {
        keyguardManager =
                (KeyguardManager) getSystemService(KEYGUARD_SERVICE);
        fingerprintManager =
                (FingerprintManager)
                     getSystemService(FINGERPRINT_SERVICE);
    }
}
```

79.8 Checking the Security Settings

Earlier in this chapter steps were taken to configure the lock screen and register fingerprints on the device or emulator on which the app is going to be tested. It is important, however, to include defensive code in the app to make sure that these requirements have been met before attempting to seek fingerprint authentication. These steps will be performed within the *onCreate* method residing in the *FingerprintDemoActivity.java* file, making use of the Keyguard and Fingerprint manager services. Note that code has also been added to verify that the USE_FINGERPRINT permission has been configured for the app:

```
.
.
import android.widget.Toast;
import android.Manifest;
import android.content.pm.PackageManager;
import android.support.v4.app.ActivityCompat;
```

```
public class FingerprintDemoActivity extends AppCompatActivity {

    private FingerprintManager fingerprintManager;
    private KeyguardManager keyguardManager;
.
.
    private Boolean getManagers() {

        keyguardManager =
                (KeyguardManager) getSystemService(KEYGUARD_SERVICE);
        fingerprintManager =
                (FingerprintManager)
                     getSystemService(FINGERPRINT_SERVICE);

        if (!keyguardManager.isKeyguardSecure()) {

            Toast.makeText(this,
                    "Lock screen security not enabled in Settings",
                    Toast.LENGTH_LONG).show();
            return false;
        }

        if (ActivityCompat.checkSelfPermission(this,
                Manifest.permission.USE_FINGERPRINT) !=
                PackageManager.PERMISSION_GRANTED) {
            Toast.makeText(this,
                    "Fingerprint authentication permission not
enabled",
                    Toast.LENGTH_LONG).show();

            return false;
        }

        if (!fingerprintManager.hasEnrolledFingerprints()) {

            // This happens when no fingerprints are registered.
            Toast.makeText(this,
                    "Register at least one fingerprint in Settings",
                    Toast.LENGTH_LONG).show();
            return false;
        }
        return true;
    }
```

```
.
.
}
```

The above code changes begin by using the Keyguard manager to verify that a backup screen unlocking method has been configured (in other words a PIN or other authentication method can be used as an alternative to fingerprint authentication to unlock the screen). In the event that the lock screen is not secured the code reports the problem to the user and returns from the method.

The fingerprint manager is then used to verify that at least one fingerprint has been registered on the device, once again reporting the problem and returning from the method if necessary.

79.9 Accessing the Android Keystore and KeyGenerator

Part of the fingerprint authentication process involves the generation of an encryption key which is then stored securely on the device using the Android Keystore system. Before the key can be generated and stored, the app must first gain access to the Keystore. A new method named *generateKey* will now be implemented within the *FingerprintDemoActivity.java* file to perform the key generation and storage tasks. Initially, only the code to access the Keystore will be added as follows:

```
.
.
import java.security.KeyStore;

public class FingerprintDemoActivity extends AppCompatActivity {

    private FingerprintManager fingerprintManager;
    private KeyguardManager keyguardManager;
    private KeyStore keyStore;
.
.
.
    protected void generateKey() {
        try {
            keyStore = KeyStore.getInstance("AndroidKeyStore");
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

A reference to the Keystore is obtained by calling the *getInstance* method of the Keystore class and passing through the identifier of the standard Android keystore container ("AndroidKeyStore"). The next step in the tutorial will be to generate a key using the KeyGenerator service. Before generating this key, code needs to be added to obtain a reference to an instance of the KeyGenerator, passing through as arguments the type of key to be generated and the name of the Keystore container into which the key is to be saved:

```
.
.
import android.security.keystore.KeyProperties;

import java.security.KeyStore;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;

import javax.crypto.KeyGenerator;

public class FingerprintDemoActivity extends AppCompatActivity {

    private FingerprintManager fingerprintManager;
    private KeyguardManager keyguardManager;
    private KeyStore keyStore;
    private KeyGenerator keyGenerator;

    protected void generateKey() {
        try {
            keyStore = KeyStore.getInstance("AndroidKeyStore");
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            keyGenerator = KeyGenerator.getInstance(
                                KeyProperties.KEY_ALGORITHM_AES,
                                "AndroidKeyStore");
        } catch (NoSuchAlgorithmException |
                                   NoSuchProviderException e) {
```

```
            throw new RuntimeException(
                    "Failed to get KeyGenerator instance", e);
        }
    }
}
```

79.10 Generating the Key

Now that we have a reference to the Android Keystore container and a KeyGenerator instance, the next step is to generate the key that will be used to create a cipher for the encryption process. Remaining within the *FingerprintDemoActivity.java* file, add this new code as follows:

```
.
.
import android.security.keystore.KeyGenParameterSpec;
.
.
import java.security.cert.CertificateException;
import java.security.InvalidAlgorithmParameterException;
import java.io.IOException;
.
.
public class FingerprintDemoActivity extends AppCompatActivity {

    private static final String KEY_NAME = "example_key";
.
.
    protected void generateKey() {
        try {
            keyStore = KeyStore.getInstance("AndroidKeyStore");
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            keyGenerator = KeyGenerator.getInstance(
                                KeyProperties.KEY_ALGORITHM_AES,
                                "AndroidKeyStore");
        } catch (NoSuchAlgorithmException |
                                   NoSuchProviderException e) {
            throw new RuntimeException(
                    "Failed to get KeyGenerator instance", e);
        }
```

```
        try {
            keyStore.load(null);
            keyGenerator.init(new
               KeyGenParameterSpec.Builder(KEY_NAME,
                    KeyProperties.PURPOSE_ENCRYPT |
                    KeyProperties.PURPOSE_DECRYPT)
                    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                    .setUserAuthenticationRequired(true)
                    .setEncryptionPaddings(
                         KeyProperties.ENCRYPTION_PADDING_PKCS7)
                            .build());
            keyGenerator.generateKey();
        } catch (NoSuchAlgorithmException |
                  InvalidAlgorithmParameterException
                 | CertificateException | IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The above changes require some explanation. After importing a number of additional modules the code declares a string variable representing the name (in this case "example_key") that will be used when storing the key in the Keystore container.

Next, the keystore container is loaded and the KeyGenerator initialized. This initialization process makes use of the KeyGenParameterSpec.Builder class to specify the type of key being generated. This includes referencing the key name, configuring the key such that it can be used for both encryption and decryption, and setting various encryption parameters. The *setUserAuthenticationRequired* method call configures the key such that the user is required to authorize every use of the key with a fingerprint authentication. Once the KeyGenerator has been configured, it is then used to generate the key via a call to the *generateKey* method of the instance.

79.11 Initializing the Cipher

Now that the key has been generated the next step is to initialize the cipher that will be used to create the encrypted FingerprintManager.CryptoObject instance. This CryptoObject will, in turn, be used during the fingerprint authentication process. Cipher configuration involves obtaining a Cipher

instance and initializing it with the key stored in the Keystore container. Add a new method named *cipherInit* to the *FingerprintDemoActivity.java* file to perform these tasks:

```
.
.
import android.security.keystore.KeyPermanentlyInvalidatedException;
.
.
import java.security.InvalidKeyException;
import java.security.KeyStoreException;
import java.security.UnrecoverableKeyException;
.
.
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;

public class FingerprintDemoActivity extends AppCompatActivity {
.
.
    private Cipher cipher;
.
.
.
    public boolean cipherInit() {
        try {
            cipher = Cipher.getInstance(
                      KeyProperties.KEY_ALGORITHM_AES + "/"
                    + KeyProperties.BLOCK_MODE_CBC + "/"
                    + KeyProperties.ENCRYPTION_PADDING_PKCS7);
        } catch (NoSuchAlgorithmException |
                            NoSuchPaddingException e) {
            throw new RuntimeException("Failed to get Cipher", e);
        }

        try {
            keyStore.load(null);
            SecretKey key = (SecretKey) keyStore.getKey(KEY_NAME,
                                                        null);
            cipher.init(Cipher.ENCRYPT_MODE, key);
            return true;
        } catch (KeyPermanentlyInvalidatedException e) {
```

```
            return false;
        } catch (KeyStoreException | CertificateException
                    | UnrecoverableKeyException | IOException
                | NoSuchAlgorithmException | InvalidKeyException e) {
            throw new RuntimeException("Failed to init Cipher", e);
        }
    }
}
```

The *getInstance* method of the Cipher class is called to obtain a Cipher instance which is subsequently configured with the properties required for fingerprint authentication. The previously generated key is then extracted from the Keystore container and used to initialize the Cipher instance. Errors are handled accordingly and a true or false result returned based on the success or otherwise of the cipher initialization process.

Work is now complete on both the *generateKey* and *cipherInit* methods. The next step is to modify the *onCreate* method to call these methods and, in the event of a successful cipher initialization, create a CryptoObject instance.

79.12 Creating the CryptoObject Instance

Remaining within the *FingerprintDemoActivity.java* file, modify the *onCreate* method to call the two newly created methods and generate the CryptoObject as follows:

```
public class FingerprintDemoActivity extends AppCompatActivity {

    private static final String KEY_NAME = "example_key";
    private FingerprintManager fingerprintManager;
    private KeyguardManager keyguardManager;
    private KeyStore keyStore;
    private KeyGenerator keyGenerator;
    private Cipher cipher;
    private FingerprintManager.CryptoObject cryptoObject;

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

        if (getManagers()) {
            generateKey();
```

```
            if (cipherInit()) {
                cryptoObject =
                        new FingerprintManager.CryptoObject(cipher);
            }

        }
    }
.
.
}
```

The final task in the project is to implement a new class to handle the actual fingerprint authentication.

79.13 Implementing the Fingerprint Authentication Handler Class

So far in this chapter most of the work has involved preparing for the fingerprint authentication in terms of the key, cipher and crypto object. The actual authentication is triggered via a call to the *authenticate* method of the FingerprintManager instance. This method call, however, will trigger one of a number of callback events depending on the success or failure of the authentication. Both the *authenticate* method call and the callback handler methods need to be implemented in a class that extends the FingerprintManager.AuthenticationCallback class. Such a class now needs to be added to the project.

Navigate to the *app -> java -> com.ebookfrenzy.fingerprintdemo* entry within the Android Studio Project tool window and right-click on it. From the resulting menu, select the *New -> Java Class* option to display the Create New Class dialog. Name the class *FingerprintHandler* and click on the *OK* button to create the class.

Edit the new class file so that it extends FingerprintManager.AuthenticationCallback, imports some additional modules and implements a constructor that will allow the application context to be passed through when an instance of the class is created (the context will be used in the callback methods to notify the user of the authentication status):

```
package com.ebookfrenzy.fingerprintdemo;
```

```
import android.Manifest;
import android.content.Context;
import android.content.pm.PackageManager;
import android.hardware.fingerprint.FingerprintManager;
import android.os.CancellationSignal;
import android.support.v4.app.ActivityCompat;
import android.widget.Toast;

public class FingerprintHandler extends
             FingerprintManager.AuthenticationCallback {

    private CancellationSignal cancellationSignal;
    private Context appContext;

    public FingerprintHandler(Context context) {
        appContext = context;
    }
}
```

Next a method needs to be added which can be called to initiate the fingerprint authentication. When called, this method will need to be passed the FingerprintManager and CryptoObject instances. Name this method *startAuth* and implement it in the *FingerprintHandler.java* class file as follows (note that code has also been added to once again check that fingerprint permission has been granted):

```
public void startAuth(FingerprintManager manager,
       FingerprintManager.CryptoObject cryptoObject) {

    cancellationSignal = new CancellationSignal();

    if (ActivityCompat.checkSelfPermission(appContext,
              Manifest.permission.USE_FINGERPRINT) !=
                     PackageManager.PERMISSION_GRANTED) {
        return;
    }
    manager.authenticate(cryptoObject, cancellationSignal, 0, this,
null);
}
```

Next, add the callback handler methods, each of which is implemented to display a toast message indicating the result of the fingerprint authentication:

```
@Override
public void onAuthenticationError(int errMsgId,
```

```
                            CharSequence errString) {
    Toast.makeText(appContext,
            "Authentication error\n" + errString,
            Toast.LENGTH_LONG).show();
}

@Override
public void onAuthenticationHelp(int helpMsgId,
                            CharSequence helpString) {
    Toast.makeText(appContext,
            "Authentication help\n" + helpString,
            Toast.LENGTH_LONG).show();
}

@Override
public void onAuthenticationFailed() {
    Toast.makeText(appContext,
            "Authentication failed.",
            Toast.LENGTH_LONG).show();
}

@Override
public void onAuthenticationSucceeded(
              FingerprintManager.AuthenticationResult result) {

    Toast.makeText(appContext,
            "Authentication succeeded.",
            Toast.LENGTH_LONG).show();
}
```

The final task before testing the project is to modify the *onCreate* method so that it creates a new instance of the FingerprintHandler class and calls the *startAuth* method. Edit the *FingerprintDemoActivity.java* file and modify the end of the *onCreate* method so that it reads as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_fingerprint);
.
.
.
    if (cipherInit()) {
        cryptoObject = new FingerprintManager.CryptoObject(cipher);
```

```
        FingerprintHandler helper = new FingerprintHandler(this);
        helper.startAuth(fingerprintManager, cryptoObject);
    }
}
```

79.14 Testing the Project

With the project now complete run the app on a physical Android device or emulator session. Once running, either touch the fingerprint sensor or use the extended controls panel within the emulator to simulate a fingerprint touch as outlined the chapter entitled *"Using and Configuring the Android Studio AVD Emulator"*. Assuming a registered fingerprint is detected a toast message will appear indicating a successful authentication as shown in Figure 79-5:

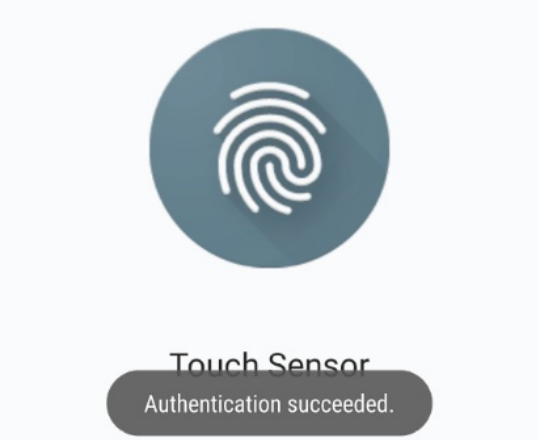

Figure 79-5

Stop the running app and relaunch it, this time using an unregistered fingerprint to attempt the authentication. This time a toast message should appear indicating that the authentication failed.

79.15 Summary

Fingerprint authentication within Android is a multi-step process that can initially appear to be complex. When broken down into individual steps, however, the process becomes clearer. Fingerprint authentication involves the use of keys, ciphers and key storage combined with the features of the FingerprintManager class. This chapter has provided an introduction to these steps and worked through the creation of an example application project intended to show the practical implementation of fingerprint authentication within Android.