Fingerprint Authentication in Action

Ben Oberkfell

@benlikestocode

Introduction

- About Me:
  - Android at American Express
  - GDG St Louis Co-Organizer
  - @benlikestocode
    - These slides are pinned there
  - Disclaimer: Opinions are my own and not my employer's

My name is Ben Oberkfell, I'm an Android developer at American Express on the US cardmember servicing app. I also help run the St Louis chapter of Google Developers Group. I've been doing Android in some fashion since 2010; I also used to write bioinformatics software for the Human Genome Project and cancer research.

Also on the advice of my attorney I should inform you that my opinions are my own and not those of my employer!

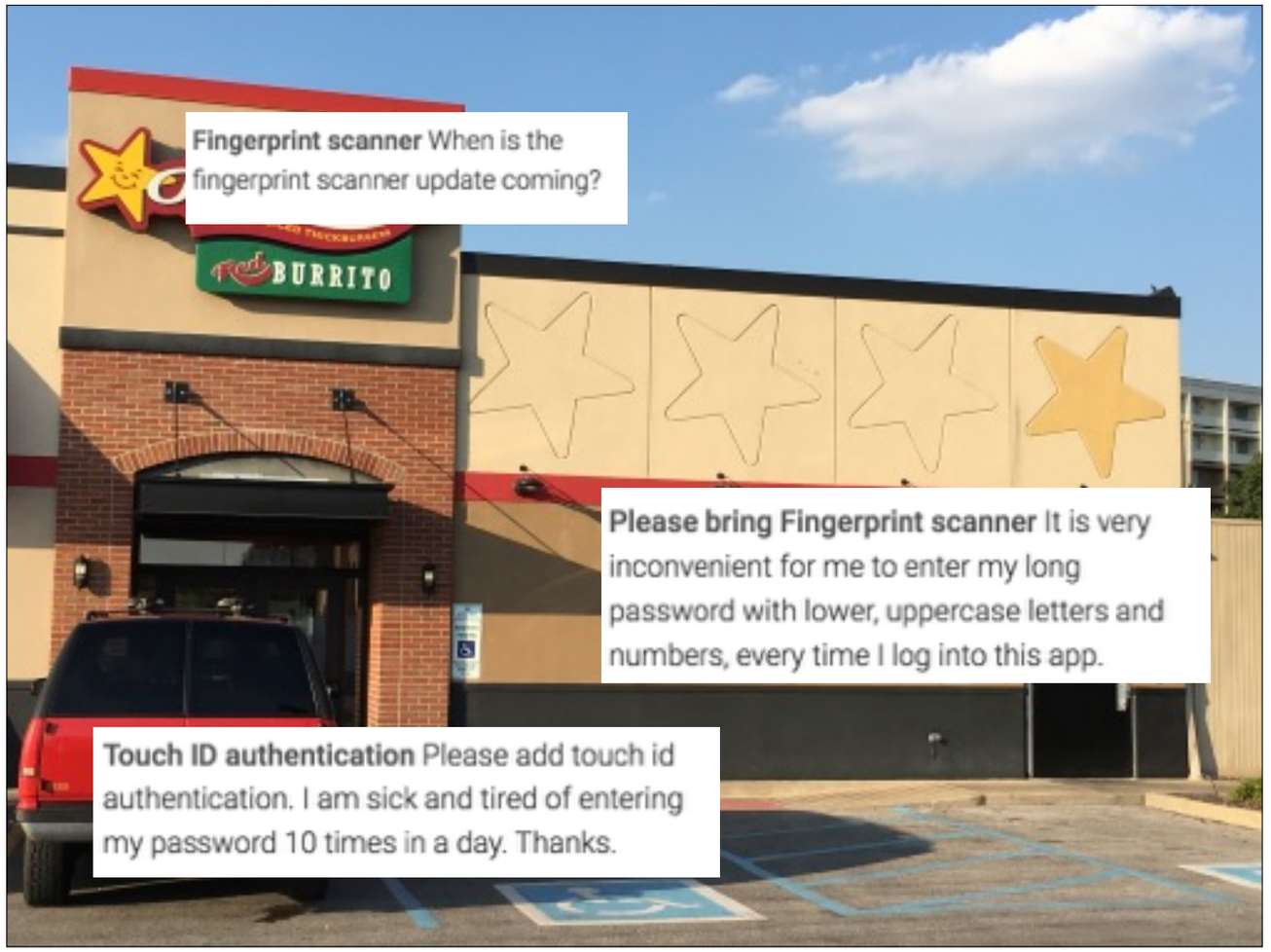
Today!

- Why/Where to Use Fingerprint
- How to Authenticate
- Designing it Right

So today we'll talk about three important things. Why and where to use fingerprint, how to do it, and how to design it right per Google's guidelines.

So with the Galaxy S6, Samsung introduced a fingerprint reader, but it didn’t get widespread acceptance until Google released an official API and reference devices like this lovely Nexus 6P.

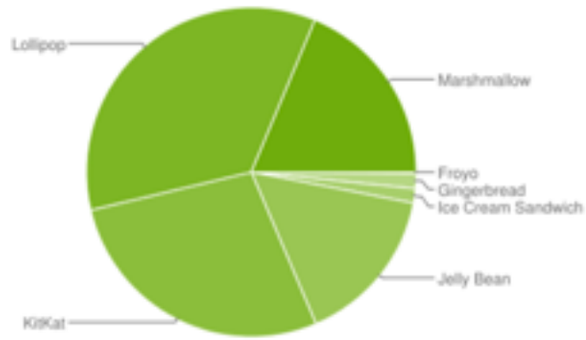
Show of hands, how many of you have a phone that supports Android Marshmallow and has a fingerprint reader? Keep them up.



Leave your hand up if you've written a one-star review because an app didn't have fingerprint authentication.

Or put your hand up if you've gotten reviews like this.

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	1.5%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.4%
4.1.x	Jelly Bean	16	5.6%
4.2.x		17	7.7%
4.3		18	2.3%
4.4	KitKat	19	27.7%
5.0	Lollipop	21	13.1%
5.1		22	21.9%
6.0	Marshmallow	23	18.7%



Data collected during a 7-day period ending on September 5, 2016.

Any versions with less than 0.1% distribution are not shown.

developer.android.com/resources/dashboard/screens.html

As of this month nearly one in five Android devices are already running Marshmallow, and devices with fingerprint readers have been out for well over a year. That means that implementing fingerprint is a feature that’s going to have a return on your time investment.

Bottom Line

This is becoming table stakes, so let’s learn how to do it!

Your customers are going to expect fingerprint interactions. If you’re in the room with me today you probably don’t need any more convincing, so let’s learn how to build this out!

Where to Use Fingerprint?

So where would we want to use fingerprint?

Login

Save the Trouble of Entering Passwords

For one, we may want to use it for logging in, in lieu of a password. In our world of data breaches it’s becoming more important to use long and complicated passwords that are a pain to type in.

Protect Critical User Flows

Things that Cost Money, Personal Security, Etc

We’ve seen too many stories of kids buying $100 in-app purchases. You can use a fingerprint check to authorize critical flows that cost your user real money.

You may also want to confirm a critical action and protect it from being replayed by an attacker.

The Hard Part

How Do You Communicate a Successful Fingerprint Scan?

The question is, if you have a flow in your app that is secured by a fingerprint scan, how do you communicate that to your backend?

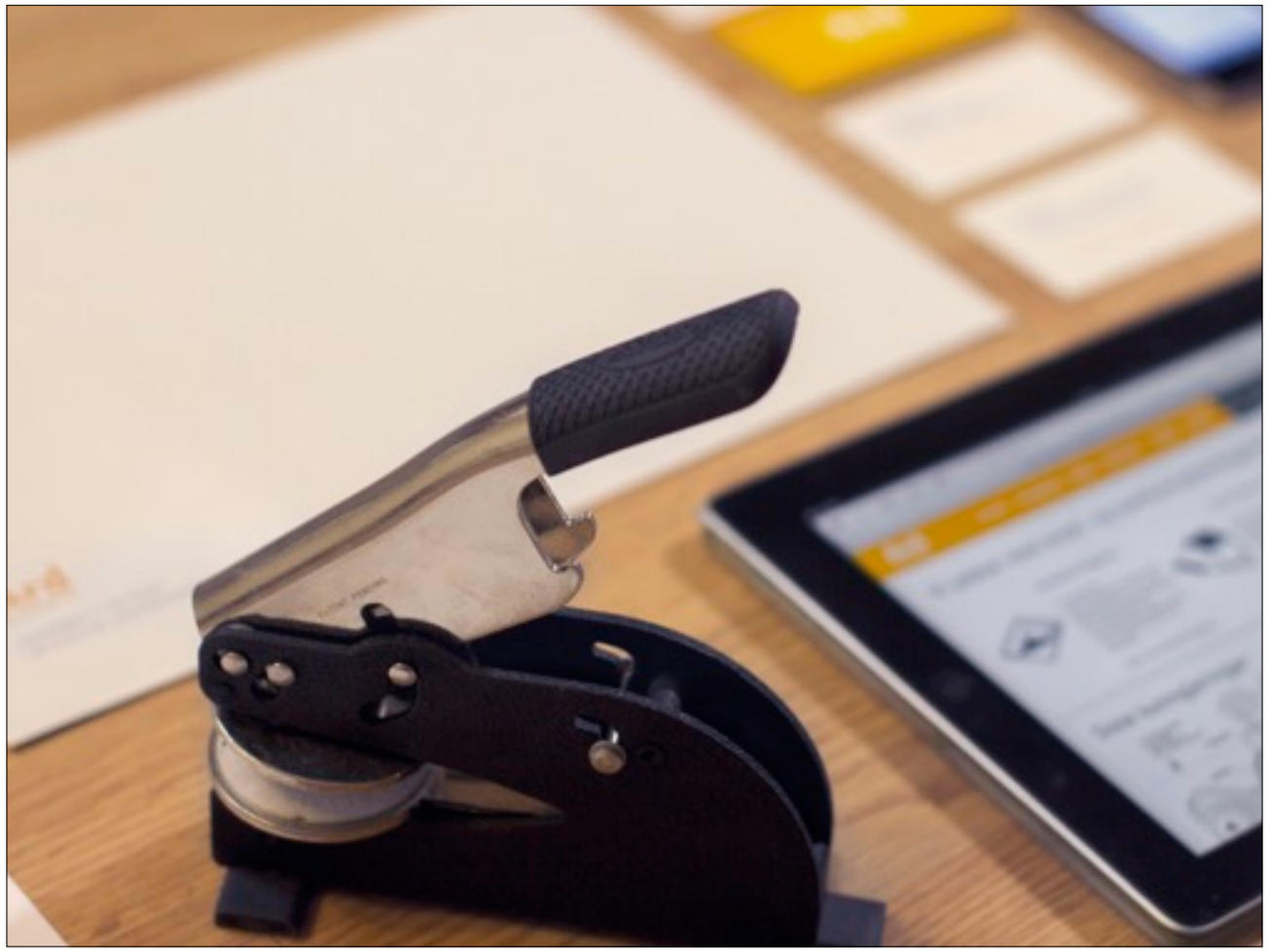
Would You Trust This?

```
{
    "item": "Pepperoni Pizza",
    "quantity": 500,
    "deliveryAddress" : "1600 Pennsylvania Avenue",
    "fingerprintValidated" : true
}
```

🤔

If the goal is to secure the user flow, nope!

Suppose you’re Domino’s Pizza, and you want to only send pizzas when your customer authorized with a fingerprint. Would you trust this?



I sold my house about two weeks ago, and as you do, I signed a bazillion forms at the closing. The county recorder and the bank all want to trust that it was me who signed, so those forms were notarized. How can I notarize my pizza order?

What's In The Fingerprint Scanner?

If a device has a fingerprint reader and provides a developer-facing API, the Compatibility Definition Document says it has to:

- have a hardware backed KeyStore
- gate usage of keys via fingerprint

To help us with that answer, let's get a look at some useful pieces of the Android platform. Android is open source and anyone can go build a device with the open source parts, but to get the Google goodies, you need to follow the Compatibility Definition Document. The document says we have to have a hardware backed keystore, and gate usage of these keys via fingerprint auth.

What Does This Mean

We can create encryption keys where the key cannot be used without authenticating with the fingerprint reader.

… which means if we can use the key, we know we successfully scanned an authorized fingerprint.

… so how do we use this for authentication?

We can create encryption keys where the key is locked out from use before the fingerprint is signed. So we know if we can use that key, we successfully signed. So how do we use this for auth?

A 10,000 Foot View

Create a public/private key pair, and make it require authentication to use. 🔑

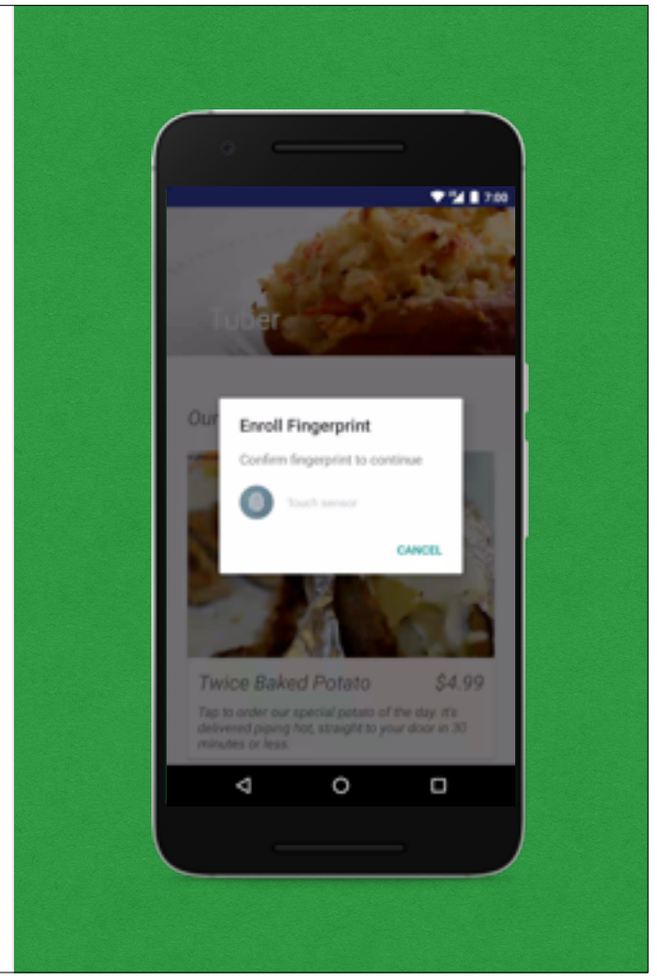
Register the public key with your backend, and associate it with your user. 💁‍♀️

Sign your critical requests with the private key. ✒️

Your backend verifies the signed request. ✅

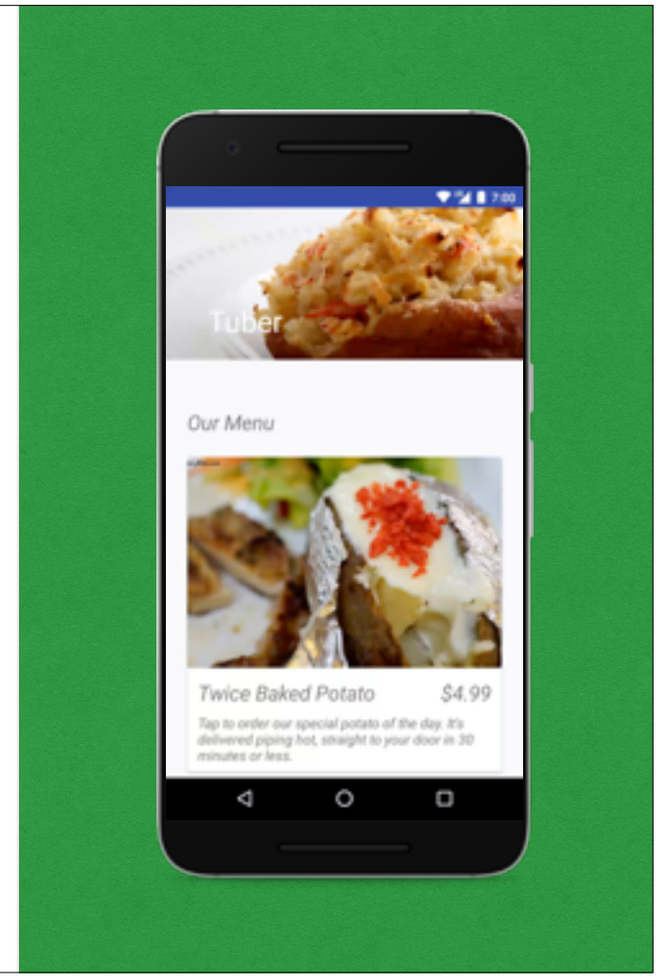
So what we're going to do is create a public/private key pair, and make it so that you need to auth with the reader to use. We'll register that public key with the backend, and associate with your user. We'll sign all our critical requests with that private key. Finally, your backend will verify the signed request.

The Goal



So here’s what we’re building today. It’s like Uber, but for baked potatoes. I call it Tuber. And we register our user’s public key with the backend on launch….

The Goal



… and then they can authorize their purchases with their fingerprint as well.

Creating the Key Pair

```
keyPairGenerator = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_RSA,
                                                "AndroidKeyStore");
keyPairGenerator.initialize(
            new KeyGenParameterSpec.Builder("DemoKey", PURPOSE_SIGN)
                    .setKeySize(2048)
                    .setDigests(DIGEST_SHA256)
                    .setSignaturePaddings(KeyProperties.
                                          SIGNATURE_PADDING_RSA_PKCS1)
                    .setUserAuthenticationRequired(true)
                .build());

keyPairGenerator.generateKeyPair();
```

Our first step is to go and create that key pair. I’m going to create a key that uses RSA and SHA256.

Creating the Key Pair

```
keyPairGenerator = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_RSA,
                                                "AndroidKeyStore");
keyPairGenerator.initialize(
            new KeyGenParameterSpec.Builder("DemoKey", PURPOSE_SIGN)
                    .setKeySize(2048)
                    .setDigests(DIGEST_SHA256)
                    .setSignaturePaddings(KeyProperties.
                                          SIGNATURE_PADDING_RSA_PKCS1)
                    .setUserAuthenticationRequired(true)
                .build());

keyPairGenerator.generateKeyPair();
```

The important thing to pay attention to here is this bit in our parameters for the key generator. setUserAuthenticationRequired is what puts the fingerprint reader as the gatekeeper in front of this key.

Signing A Request

```
public String signString(String data) {
    KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load(null);
    PrivateKey privateKey = keyStore.getKey(KEY_NAME, null);

    Signature signature = Signature.getInstance("SHA256withRSA");
    signature.initSign(getPrivateKey());
    signature.update(dataToSign.getBytes("UTF8"))

    byte[] bytes = cryptoObject.getSignature().sign();
    return Base64.encodeToString(bytes, Base64.NO_WRAP);
}
```

But this won’t work right off the bat, since we’re not authenticated
by the fingerprint reader..

So now with some code like this, we could go out and sign some data with that key. But this isn’t going to work, and will throw an exception instead, because the user hasn’t authenticated.

Authenticating First

Some prereqs:

- Call for the USE_FINGERPRINT permission in your manifest. It's auto-granted.
- We'd use the FingerprintManager class to work with the fingerprint sensor.
- But to make our lives easier for minSdkVersion < 23, use FingerprintManagerCompat instead.

So let's get ourselves authenticated, shall we? First things first, we'll need the USE_FINGERPRINT permission. It's auto granted so no need to ask the user for it. The FIngerprintManager class in the platform handles this for us, but if we're below minSdkVersion 23 then we'll use FingerprintManagerCompat, which levels the playing field between pre-Marshmallow devices and any device with no fingerprint reader at all.

Detecting Fingerprint Status

Does the device have fingerprint hardware?

```
boolean hasHardware = fingerprintManager.isHardwareDetected();
```

Did the user enroll any fingerprints yet?

```
boolean hasFingerprints = fingerprintManager.hasEnrolledFingerprints();
```

So now, we can ask the fingerprint manager if we have hardware, and if they do, we can ask the user to go enroll fingerprints if they haven’t done so already.

Authenticating

```
public void requestFingerprintAuth(Signature signature,
                                   FingerprintManagerCompat.AuthenticationCallback callback) {

    CryptoObject cryptoObject = new CryptoObject(signature);
    CancellationSignal cancellationSignal = new CancellationSignal();

    FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.from(context);

    fingerprintManager.authenticate(cryptoObject, 0, cancellationSignal,
                                    callback, null);

}
```

Now let’s get on to how we can use the FingerprintManager to do authentication. Let’s break it down.

Authenticating

```
public void requestFingerprintAuth(Signature signature,
                                   FingerprintManagerCompat.AuthenticationCallback callback) {

    CryptoObject cryptoObject = new CryptoObject(signature);
    CancellationSignal cancellationSignal = new CancellationSignal();

    FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.from(context);

    fingerprintManager.authenticate(cryptoObject, 0, cancellationSignal,
                                    callback, null);

}
```

The CryptoObject wraps the type of crypto operation we're going to be using with the fingerprint auth. In addition to signing data using asymmetric keys, you can use the fingerprint scanner to gate asymmetric encryption, as well as symmetric encryption where the whole key is stored on the device. Today we're signing, though, so let's wrap the signature.

Authenticating

```
public void requestFingerprintAuth(Signature signature,
                                   FingerprintManagerCompat.AuthenticationCallback callback) {

    CryptoObject cryptoObject = new CryptoObject(signature);
    CancellationSignal cancellationSignal = new CancellationSignal();

    FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.from(context);

    fingerprintManager.authenticate(cryptoObject, 0, cancellationSignal,
                                    callback, null);

}
```

We’re going to call authenticate on our fingerprint manager, with our crypto object, flags, a cancellation signal, our callback, and optionally a Handler for events. More on the callback in a bit, let’s talk about the cancellation signal.

Authenticating

```
public void requestFingerprintAuth(Signature signature,
                                   FingerprintManagerCompat.AuthenticationCallback callback) {

    CryptoObject cryptoObject = new CryptoObject(signature);
    CancellationSignal cancellationSignal = new CancellationSignal();

    FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.from(context);

    fingerprintManager.authenticate(cryptoObject, flags, cancellationSignal,
                                    callback, null);

}
```

The cancellation signal is how you tell the fingerprint reader to stop listening. Obviously in real code we don’t want it to be locally scoped to this method. It’s very important to pay attention to your activity lifecycle and stop listening when you’re not active.

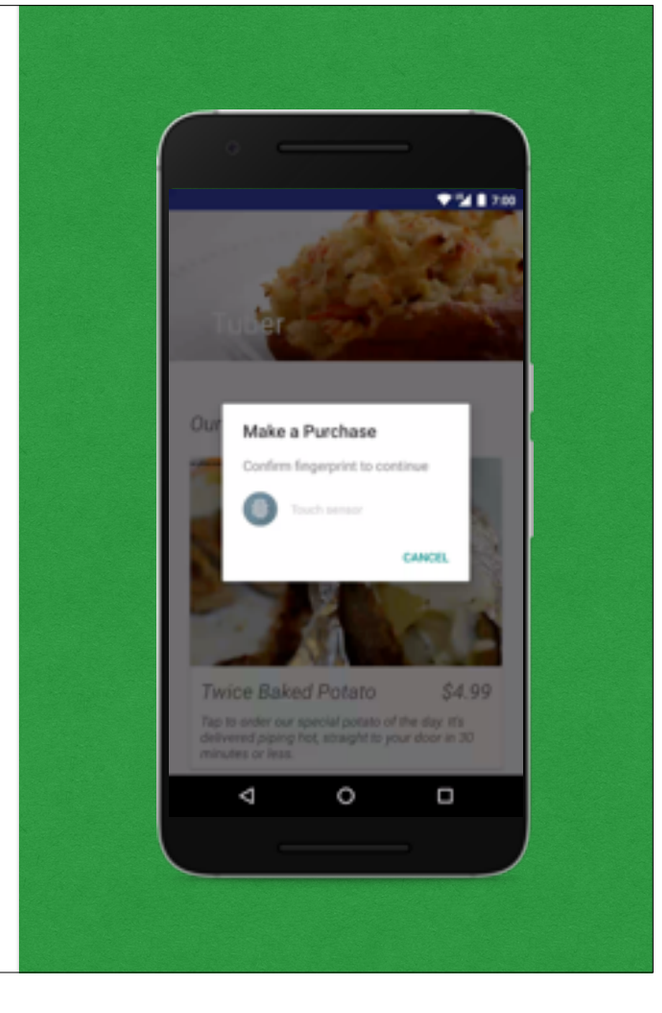
Reacting to Successful Auth

```
FingerprintManagerCompat.AuthenticationCallback callback =
    new FingerprintManagerCompat.AuthenticationCallback() {
            @Override
            public void onAuthenticationSucceeded(FingerprintManagerCompat.
                                                  AuthenticationResult result) {
                super.onAuthenticationSucceeded(result);
                CryptoObject cryptoObject = result.getCryptoObject();

                // NOW we can use this Signature to sign!
                Signature signature = cryptoObject.getSignature();
                signTheThing(signature, data)
            }

            // ....
}
```

Now let’s get a look at the callbacks. The AuthenticationResult gives us back the crypto object we passed in, this time blessed by the fingerprint authentication. Now we can unpack the signature from it and use the signature to sign some data.

Handling Unsuccessful Authentication

Wrong Fingerprint



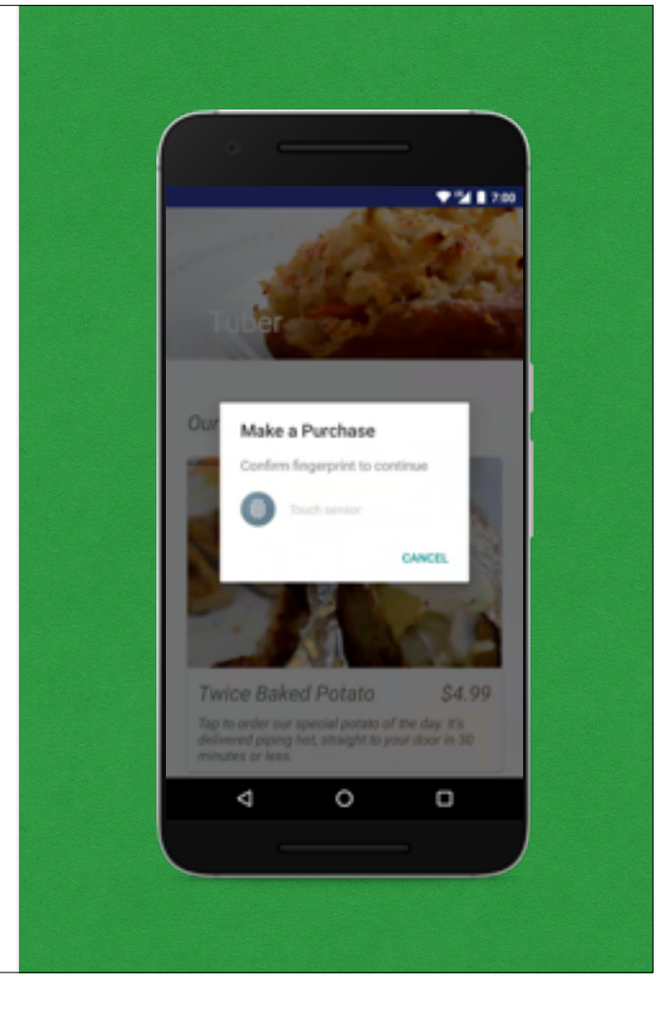
One scenario we can get back is a failed authentication, which is where we've scanned a fingerprint, but it's not an authorized one. We should display a wrong fingerprint message in the UI.

Wrong Fingerprint

```
FingerprintManagerCompat.AuthenticationCallback callback =
    new FingerprintManagerCompat.AuthenticationCallback() {

            // Bad Fingerprint
             @Override
             public void onAuthenticationFailed() {
                 super.onAuthenticationFailed();
                 fingerprintView.onError("Fingerprint Not Recognized");
             }
        // ...
    }
}
```

onAuthenticationFailed is how we'd handle this.

Soft Error



Another scenario is a soft error. This might be, for instance, the user moved their finger too fast, or the sensor was dirty.

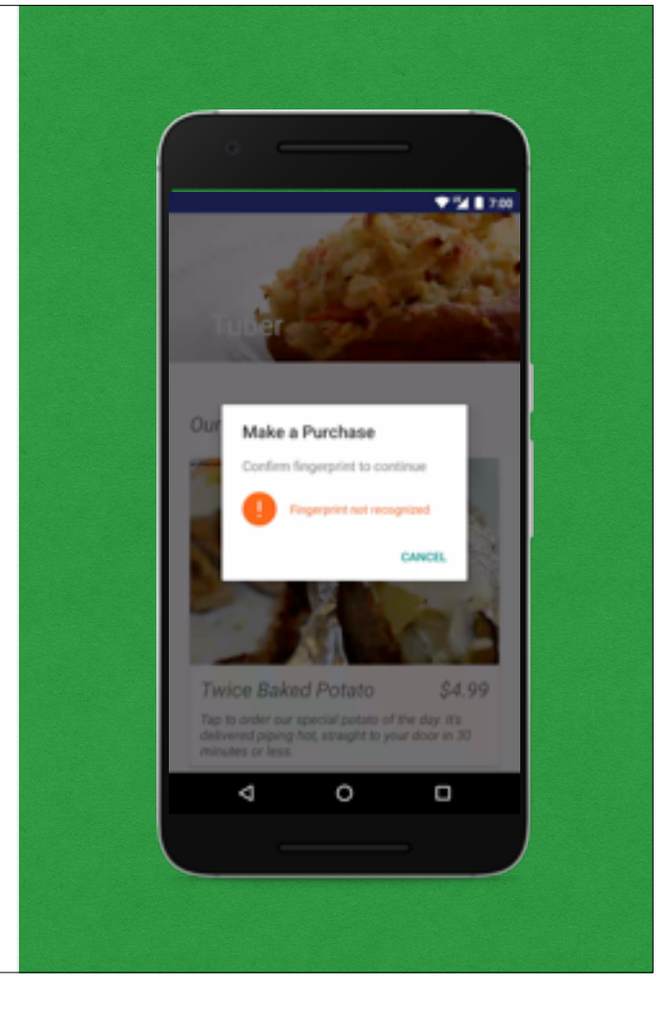
Soft Error

```
FingerprintManagerCompat.AuthenticationCallback callback =
    new FingerprintManagerCompat.AuthenticationCallback() {

            // ....

            @Override
            public void onAuthenticationHelp(int helpMsgId,
                                             CharSequence helpString) {
                  super.onAuthenticationHelp(errMsgId, errString);
                  fingerprintView.onHelp(helpString.toString());

            // ....

    }
}
```

The framework gives you a help message to use — use it!

The onAuthenticationHelp callback is to help you disambiguate between these soft errors and hard errors. The framework even gives you a help string to present to your user. In the interest of consistency for the user, you should probably use it.

Hard Error



Another error condition is a hard error, after which point no further calls into your callback will happen. An example here is that you had too many wrong finger attempts and the fingerprint reader locked out the hardware. If the user locked their reader out in another app, you may even get this callback right off the bat.

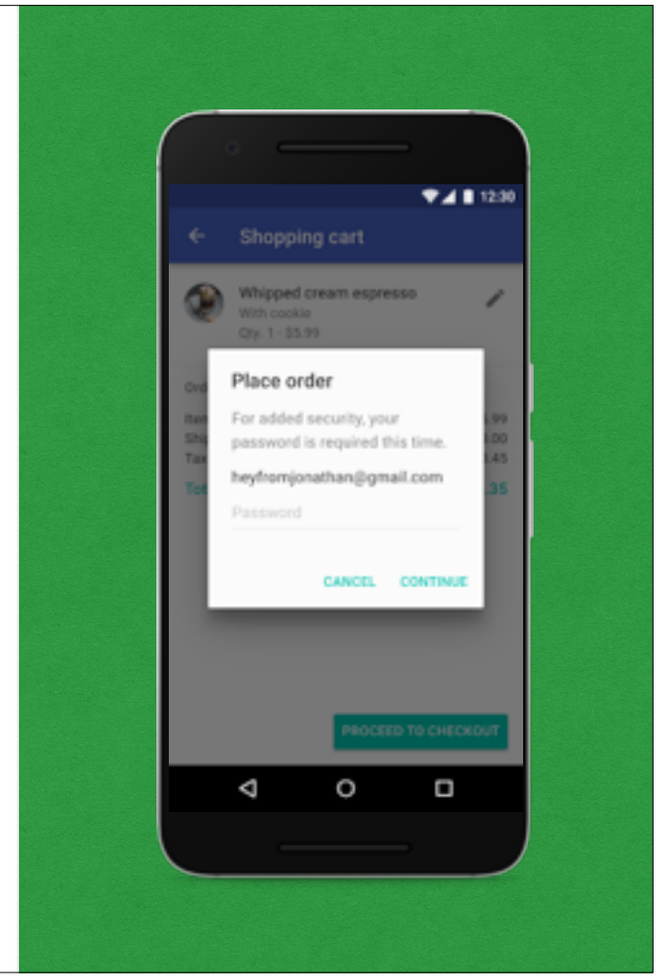
Hard Error

```
FingerprintManagerCompat.AuthenticationCallback callback =
    new FingerprintManagerCompat.AuthenticationCallback() {

            // ....

            public void onAuthenticationError(int errMsgId, CharSequence errString) {
                super.onAuthenticationHelp(errMsgId, errString);
                fingerprintView.onError(errString.toString());
                fingerprintView.handleHardError();
            }

            // ....
    }
}
```

The framework gives you an error message to use — use it!

So here onAuthenticationError is how you're going to deal with this. And again, your friendly framework gives you a message to use as to what went wrong.

Key Invalidation



If a user adds a new fingerprint, old keys tied to fingerprint get invalidated. Same goes for if they turn off their lockscreen.

You'll get an InvalidKeyException when you try to create a signature with an invalid key. Handle it and present a message to the user to fall back to password.

On the Backend Side

Responding to Requests

Registering the Key

```
{
    'registerRequest' : {
        'username' : 'jdoe',
        'password' : 'p4ss4w0rd',
        'timestamp' : 1474000105154,
        'publicKey' : '123456789ABC…'
    },
    'signature' : '123456789ABC…'
}
```

Sign this request, to prove ownership of the private key

Here is a really stupid simple and contrived example. You may well want to do this differently, for instance you authenticate normally with user name and password, get a session token, and then pass that session token in instead. You want to make sure the user has ownership of the private key, so when you post your public key, make sure to sign the whole request. Validate it
in your backend, too.

Now, Make the Request

```
{
    'purchaseRequest' : {
        "item": "Pepperoni Pizza",
        "quantity": 500,
        "deliveryAddress" : "1600 Pennsylvania Avenue",
        'timestamp' : 1474000105154,
    },
    'signature' : '123456789ABC…'
}
```

So here we can actually order a pizza, and we'll post this to our backend. We've authenticated in our user flow, signed the request, and send this up.

Validate It

```
fun isRequestValidlySigned(request: SignedRequest<T>, publicKey: String) :
                           Boolean {
        val pubKeyBytes = Base64.getDecoder().decode(publicKey)
        val signatureBytes = Base64.getDecoder().decode(request.signature)

        val kf = KeyFactory.getInstance("RSA")
        val key = kf.generatePublic(X509EncodedKeySpec(pubKeyBytes))
        val verify = Signature.getInstance("SHA256withRSA")
        verify.initVerify(key)
        verify.update(request.payload.messageForVerification().toByteArray())
        return verify.verify(signatureBytes)
    }
```

So here we can get our message back and validate it. This is in the example code (it’s in Kotlin, yay!). The important thing here too is we need to normalize our data for both signing and verification.

Normalize It

You may find yourself needing to transform the data to normalize it when both signing and validating.

Pepperoni Pizzal500l1600 Pennsylvania Avenuel1474000105154

However you choose to do it, do it the same on either side.

Signing can be tough with json. You likely will need to normalize the data in some fashion, especially if your frameworks handle serialization for you. Whatever you do, do the same thing on both the client and the backend.

Prevent Relay Attacks

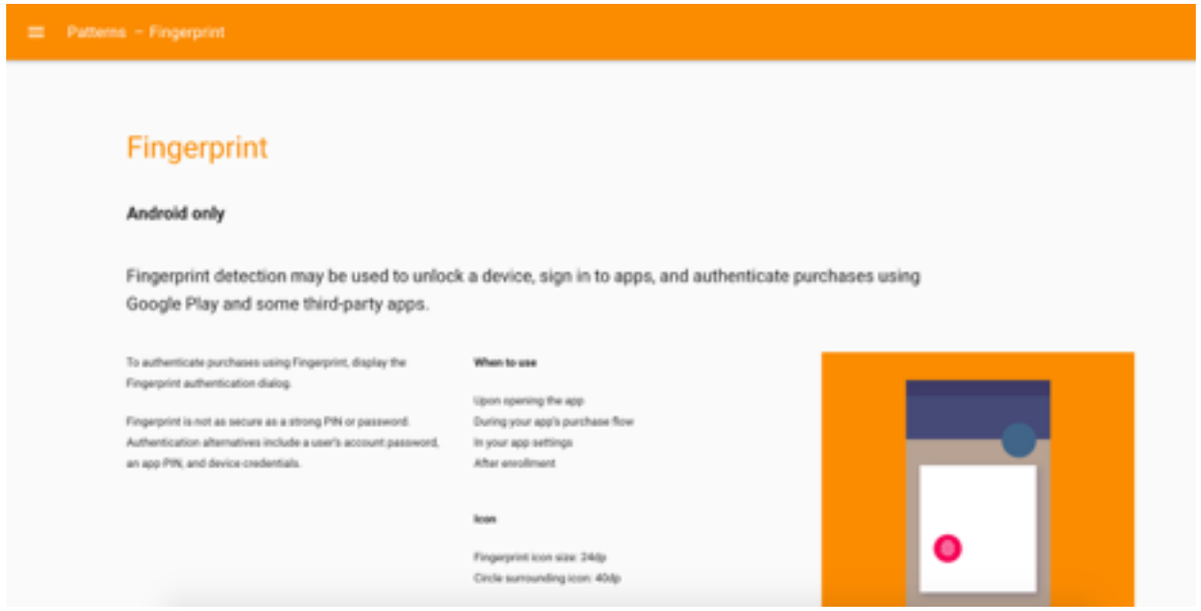
- Use a "nonce."
- For instance, a timestamp in the request prevents replaying the same request later. Tampering with the timestamp would invalidate the signature.
- You could also use other business data available to you if you can verify that the value hasn't been used multiple times.

If we aren't careful, we could allow the same request to be re-posted and next thing you know we've ordered a pizza the user did not authorize. We should add a piece of uniquely valid data to ensure you can't replay the request.

Onto UI

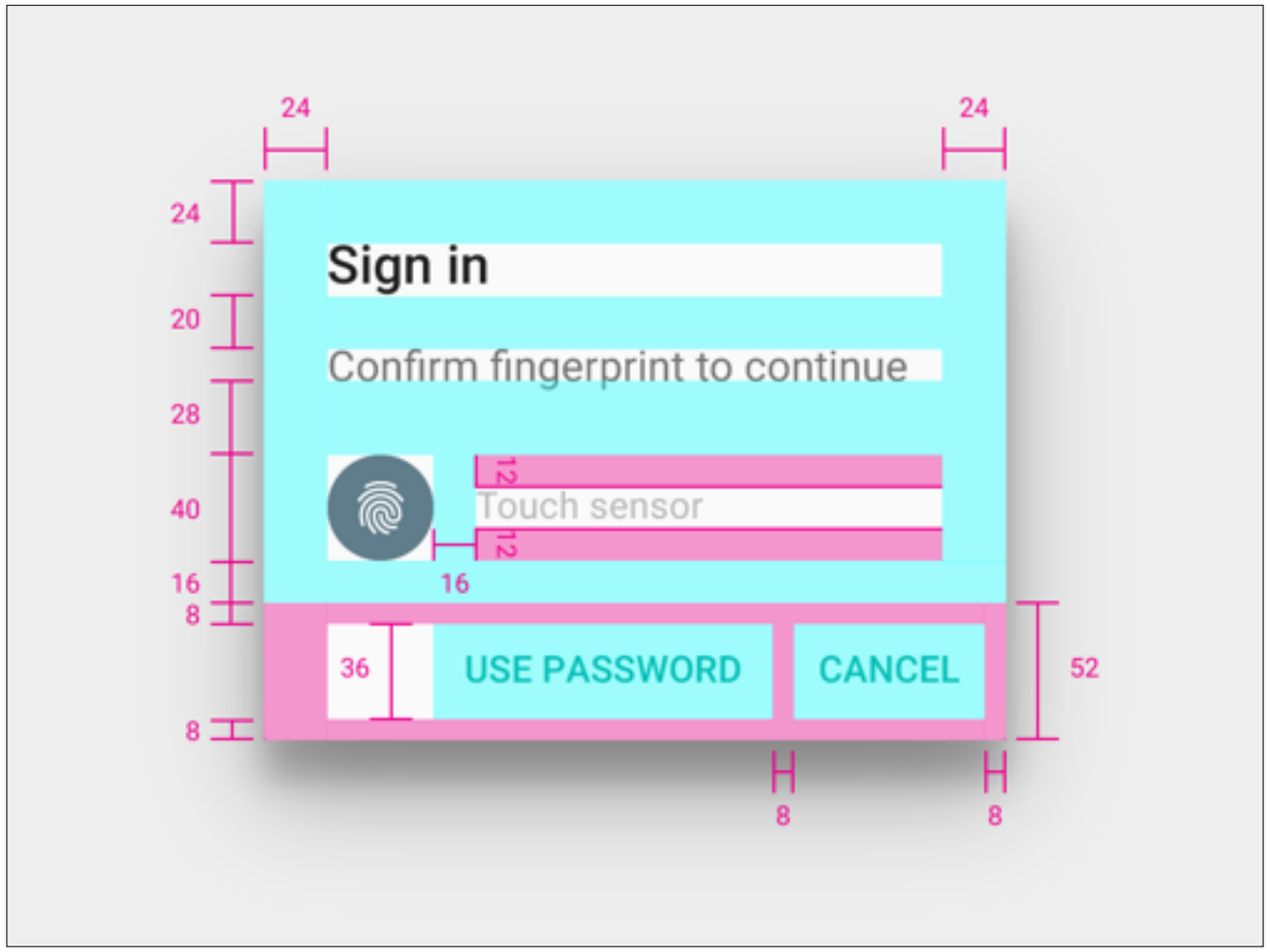
Let’s make it look good

Now let’s talk design.



https://material.google.com/patterns/fingerprint.html

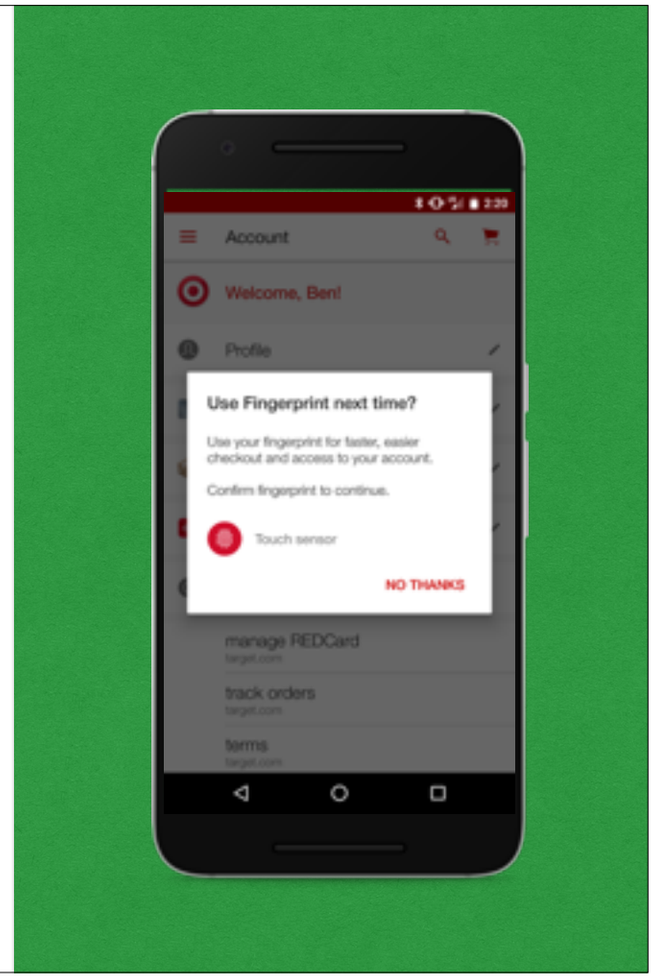
The Google Material Guidelines page has some discussion about how to design for fingerprint authentication.



The guidelines include some redlines for how to design the fingerprint dialog.

The header should indicate what you're going to do - sign in, make a purchase, etc. You should use the phrase "confirm fingerprint." And you should use the Material fingerprint icon.

If you redesign, keep the icon!



Users are instructed to look for the material fingerprint icon as a hint to use their fingerprint. If you re-theme your app, you should still use the icon, even if you change the fonts or color scheme.

Example App

Example App & Backend

https://goo.gl/oqtmgF

Thank You!

Image Credits

- One Star Restaurant (own photo)
- Nexus 6p https://flic.kr/p/C7keQW
- Notary Embosser https://flic.kr/p/dNMmAi