

# What is the most appropriate way to store user settings in Android application

Asked 15 years, 8 months ago   Modified 2 years, 6 months ago

Viewed 161k times    Part of [Mobile Development](#) Collective



322



I am creating an application which connects to the server using username/password and I would like to enable the option "Save password" so the user wouldn't have to type the password each time the application starts.

I was trying to do it with Shared Preferences but am not sure if this is the best solution.

I would appreciate any suggestion on how to store user values/settings in Android application.

MD

java

android

encryption

preferences

credentials

Share Follow

edited Jun 5, 2014 at 6:25



[chiragkyada](#)

3,615 ● 3 ● 18 ● 18

asked Apr 24, 2009 at 14:03



[Niko Gamulin](#)

66.5k ● 96 ● 228 ● 293

**241**

In general SharedPreferences are your best bet for storing preferences, so in general I'd recommend that approach for saving application and user settings.



The only area of concern here is what you're saving. Passwords are always a tricky thing to store, and I'd be particularly wary of storing them as clear text. The Android architecture is such that your application's SharedPreferences are sandboxed to prevent other applications from being able to access the values so there's some security there, but physical access to a phone could potentially allow access to the values.

If possible I'd consider modifying the server to use a negotiated token for providing access, something like [OAuth](#). Alternatively you may need to construct some sort of cryptographic store, though that's non-trivial. At the very least, make sure you're encrypting the password before writing it to disk.

Share Follow

answered Apr 24, 2009 at 16:20

**Reto Meier****96.9k** ● 18 ● 101 ● 72

---


3 Could you please explain what you mean by sandboxed?  
– [Abhijit](#) Jul 3, 2012 at 15:00

---



16 a sandboxed program is any application whose process and information (such as those shared preferences) remains hidden from the rest of the applications. An android

application running in a package cannot directly access to anything inside another package. That's why applications in the same package (which are always yours) could access to information from other ones – [Sergi Juanola](#) Jul 12, 2012 at 20:46

---

@Reto Meier my requirement is to protect the publicly available web services for that I am using a token, is storing it on shared preferences is safe? i have a bootup broadcast receiver in my application which will delete all sharedpreferences data if it found device as rooted. Is this enough to protect my token. – [Piyush Agarwal](#) Mar 11, 2013 at 19:47 

---

- 1 Per [android-developers.blogspot.com/2013/02/...](http://android-developers.blogspot.com/2013/02/...), *User credentials should be stored with the MODE\_PRIVATE flag set and stored in internal storage* (with the same caveats about storing any sort of password locally ultimately open to attack). That said, is using `MODE_PRIVATE` with `SharedPreferences` equivalent to doing the same with a file created on internal storage, in terms of effectiveness to obfuscate locally stored data? – [qix](#) Oct 18, 2013 at 20:06 
  - 9 Do not store a password in shared preferences. If the user ever loses the phone, they've lost the password. It will be read. If they used that password elsewhere, everyplace they used it is compromised. In addition, you've permanently lost this account because with the password they can change your password. The correct way to do this is to send the password up to the server once, and receive a login token back. Store that in shared preference and send it up with each request. If that token is compromised, nothing else is lost. – [Gabe Sechan](#)  Jul 27, 2014 at 2:31
- 



I agree with Reto and fiXedd. Objectively speaking it doesn't make a lot of sense investing significant time and



effort into encrypting passwords in SharedPreferences since any attacker that has access to your preferences file is fairly likely to also have access to your application's binary, and therefore the keys to unencrypt the password.

However, that being said, there does seem to be a publicity initiative going on identifying mobile applications that store their passwords in cleartext in SharedPreferences and shining unfavorable light on those applications. See <http://blogs.wsj.com/digits/2011/06/08/some-top-apps-put-data-at-risk/> and <http://viaforensics.com/appwatchdog> for some examples.

While we need more attention paid to security in general, I would argue that this sort of attention on this one particular issue doesn't actually significantly increase our overall security. However, perceptions being as they are, here's a solution to encrypt the data you place in SharedPreferences.

Simply wrap your own SharedPreferences object in this one, and any data you read/write will be automatically encrypted and decrypted. eg.

```
final SharedPreferences prefs = new ObscuredSharedPref
    this, this.getSharedPreferences(MY_PREFS_FILE_NAME
);

// eg.
prefs.edit().putString("foo", "bar").commit();
prefs.getString("foo", null);
```

Here's the code for the class:

```
/**
 * Warning, this gives a false sense of security. If
access to
 * acquire your password store, then he almost certain
acquire your
 * source binary and figure out your encryption key.
casual
 * investigators from acquiring passwords, and thereby
negative
 * publicity.
 */
public class ObscuredSharedPreferences implements Shared
    protected static final String UTF8 = "utf-8";
    private static final char[] SEKRET = ... ; // INSE
HERE.

                                                                    // Don'
wouldn't want to

                                                                    // get
decompiled

                                                                    // your

    protected SharedPreferences delegate;
    protected Context context;

    public ObscuredSharedPreferences(Context context,
delegate) {
        this.delegate = delegate;
        this.context = context;
    }

    public class Editor implements SharedPreferences.E
        protected SharedPreferences.Editor delegate;

        public Editor() {
            this.delegate = ObscuredSharedPreferences.
        }

        @Override
        public Editor putBoolean(String key, boolean v
            delegate.putString(key, encrypt(Boolean.to
```

```
        return this;
    }

    @Override
    public Editor putFloat(String key, float value) {
        delegate.putString(key, encrypt(Float.toString(value)));
        return this;
    }

    @Override
    public Editor putInt(String key, int value) {
        delegate.putString(key, encrypt(Integer.toString(value)));
        return this;
    }

    @Override
    public Editor putLong(String key, long value) {
        delegate.putString(key, encrypt(Long.toString(value)));
        return this;
    }

    @Override
    public Editor putString(String key, String value) {
        delegate.putString(key, encrypt(value));
        return this;
    }

    @Override
    public void apply() {
        delegate.apply();
    }

    @Override
    public Editor clear() {
        delegate.clear();
        return this;
    }

    @Override
    public boolean commit() {
        return delegate.commit();
    }

    @Override
```

```

        public Editor remove(String s) {
            delegate.remove(s);
            return this;
        }
    }

    public Editor edit() {
        return new Editor();
    }

    @Override
    public Map<String, ?> getAll() {
        throw new UnsupportedOperationException(); //
the reader
    }

    @Override
    public boolean getBoolean(String key, boolean defV
        final String v = delegate.getString(key, null)
        return v!=null ? Boolean.parseBoolean(decrypt(v))
    }

    @Override
    public float getFloat(String key, float defValue)
        final String v = delegate.getString(key, null)
        return v!=null ? Float.parseFloat(decrypt(v))
    }

    @Override
    public int getInt(String key, int defValue) {
        final String v = delegate.getString(key, null)
        return v!=null ? Integer.parseInt(decrypt(v))
    }

    @Override
    public long getLong(String key, long defValue) {
        final String v = delegate.getString(key, null)
        return v!=null ? Long.parseLong(decrypt(v)) :
    }

    @Override
    public String getString(String key, String defValu
        final String v = delegate.getString(key, null)

```

```

        return v != null ? decrypt(v) : defValue;
    }

    @Override
    public boolean contains(String s) {
        return delegate.contains(s);
    }

    @Override
    public void
registerOnSharedPreferenceChangeListener(OnSharedPrefe
onSharedPreferenceChangeListener) {

    delegate.registerOnSharedPreferenceChangeListener(onSh
    }

    @Override
    public void
unregisterOnSharedPreferenceChangeListener(OnSharedPre
onSharedPreferenceChangeListener) {

    delegate.unregisterOnSharedPreferenceChangeListener(on
    }

    protected String encrypt( String value ) {

        try {
            final byte[] bytes = value!=null ? value.g
byte[0];

            SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance("PBESWithMD5AndDES");
            SecretKey key = keyFactory.generateSecret(
            Cipher pbeCipher = Cipher.getInstance("PBE
            pbeCipher.init(Cipher.ENCRYPT_MODE, key, n
PBESParameterSpec(Settings.Secure.getString(context.get
20)));

            return new String(Base64.encode(pbeCipher.
Base64.NO_WRAP),UTF8);

        } catch( Exception e ) {
            throw new RuntimeException(e);
        }
    }

```



```

    }

    }

    protected String decrypt(String value){
        try {
            final byte[] bytes = value!=null ?
Base64.decode(value,Base64.DEFAULT) : new byte[0];
            SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance("PBESWithMD5AndDES");
            SecretKey key = keyFactory.generateSecret(
Cipher pbeCipher = Cipher.getInstance("PBE
pbeCipher.init(Cipher.DECRYPT_MODE, key, n
PBEPParameterSpec(Settings.Secure.getString(context.get
20));

            return new String(pbeCipher.doFinal(bytes)

        } catch( Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

Share Follow

edited Apr 13, 2016 at 17:09



**JulienGenoud**

630 ● 10 ● 22

answered Jun 18, 2011 at 2:42



**emmby**

100k ● 66 ● 194 ● 251

---

3 FYI Base64 is available in API level 8 (2.2) and later. You can use [i harder.sourceforge.net/current/java/base64](http://i harder.sourceforge.net/current/java/base64) or something else for earlier OSs. – **emmby** Jun 20, 2011 at 22:19

---

38 Yes, I wrote this. Feel free to use, no attribution necessary – **emmby** Sep 10, 2012 at 16:21

---

8 I agree with you. But if the password is only used on the server, why not use Public/private key encryption? Public key on client when saving the password. The client will never have to read the clear text password again, right? The server can then decrypt it with the private key. So even if somebody goes through your app source code, they can't get the password, except they hack your server and get the private key. – [Patrick Boos](#) Sep 24, 2012 at 1:43 ✎

---

4 I've added a few features to this code and placed it on github at [github.com/RightHandedMonkey/WorxForUs\\_Library/blob/master/src/...](https://github.com/RightHandedMonkey/WorxForUs_Library/blob/master/src/...). It now handles migrating a non-encrypted preferences to the encrypted one. Also it generates the key at runtime, so decompiling the app does not release the key. – [RightHandedMonkey](#) Apr 23, 2014 at 15:15

---

3 Late addition, but the comment by @PatrickBoos is a great idea. One problem with this, though, is that even though you've encrypted the password, an attacker that stole that cipher would still be able to log in to your servers, because your servers do the decryption. One addition to this approach is to encrypt the password together with a timestamp. That way you can decide, for example, to only allow passwords saved in the recent past (like adding an expiration date to your "token"), or even requiring certain users to have a timestamp since a particular date (let's you "revoke" old "tokens"). – [adevine](#) Oct 23, 2014 at 21:55

---



About the simplest way to store a single preference in an Android Activity is to do something like this:

29



```
Editor e = this.getSharedPreferences(Context.MODE_PRIVATE).e
e.putString("password", mPassword);
e.commit();
```



If you're worried about the security of these then you could always encrypt the password before storing it.

Share Follow

edited Jun 18, 2014 at 17:26

answered May 4, 2009 at 8:50



[Jeremy Logan](#)

47.5k ● 39 ● 124 ● 145

---

9 I couldn't agree with you more about this simplistic approach; however, you should always be worried about the security of passwords that you store? Depending on your application, you have potential liabilities for stolen personal information. Just pointing this out for anybody trying to store actual passwords to such things as bank accounts or something equally important. I still vote you though. – [While-E](#) Jun 16, 2011 at 4:21 ✎

---

2 Where would you store the key that stored the password? If the shared preferences are accessible by other users, so is the key. – [OrhanC1](#) Aug 7, 2014 at 14:08

---

@OrhanC1 did you get the answer.? – [eRaisedToX](#) Aug 19, 2017 at 12:04 ✎

---

@eRaisedToX The answer is there is no answer. [EncryptedSharedPreferences](#) even runs into this problem when it stores the encryption key in KeyStore. As long as the encryption key is on the device, the data is vulnerable. The only semi-solution is to put the key on a server, but that means the key must exist in the device at some point (either in storage or in memory). – user19309143 Jul 1, 2022 at 21:26 ✎

---



10



Using the snippet provided by Richard, you can encrypt the password before saving it. The preferences API however doesn't provide an easy way to intercept the value and encrypt it - you can block it being saved via an `OnPreferenceChange` listener, and you theoretically could modify it through a `preferenceChangeListener`, but that results in an endless loop.

I had earlier suggested adding a "hidden" preference in order to accomplish this. It's definitely not the best way. I'm going to present two other options that I consider to be more viable.

First, the simplest, is in a `preferenceChangeListener`, you can grab the entered value, encrypt it, and then save it to an alternative preferences file:

```
public boolean onPreferenceChange(Preference preference) {
    // get our "secure" shared preferences file.
    SharedPreferences secure = context.getSharedPreferences(
        "SECURE",
        Context.MODE_PRIVATE
    );
    String encryptedText = null;
    // encrypt and set the preference.
    try {
        encryptedText = SimpleCrypto.encrypt(Preference
            (String)newValue);

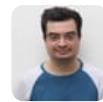
        Editor editor = secure.getEditor();
        editor.putString("encryptedPassword", encryptedText);
        editor.commit();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
// always return false.  
return false;  
}
```

The second way, and the way I now prefer, is to create your own custom preference, extending `EditTextPreference`, @Override'ing the `setText()` and `getText()` methods, so that `setText()` encrypts the password, and `getText()` returns null.

Share Follow

edited Mar 14, 2016 at 12:36



Praful Bhatnagar

7,445 ● 2 ● 37 ● 44

answered Mar 2, 2011 at 14:56



Mark

2,542 ● 19 ● 37

---

I know this is pretty old, but would you mind posting your code for your custom version of `EditTextPreference`, please?

– RenniePet Sep 6, 2013 at 2:50

---

Never mind, I found a usable sample here

[groups.google.com/forum/#!topic/android-developers/pMYNEVXMa6M](https://groups.google.com/forum/#!topic/android-developers/pMYNEVXMa6M) and I've got it working now.

Thanks for suggesting this approach. – RenniePet Sep 6, 2013 at 5:04



6

Okay; it's been a while since the answer is kind-of mixed, but here's a few common answers. I researched this like crazy and it was hard to build a good answer



1. The `MODE_PRIVATE` method is considered generally safe, if you assume that the user didn't root the device. Your data is stored in plain text in a part of the file system that can only be accessed by the original program. This makes grabbing the password with another app on a rooted device easy. Then again, do you want to support rooted devices?
2. AES is still the best encryption you can do. Remember to look this up if you are starting a new implementation if it's been a while since I posted this. The largest issue with this is "What to do with the encryption key?"

So, now we are at the "What to do with the key?" portion. This is the hard part. Getting the key turns out to be not that bad. You can use a key derivation function to take some password and make it a pretty secure key. You do get into issues like "how many passes do you do with PKDF2?", but that's another topic

1. Ideally, you store the AES key off the device. You have to figure out a good way to retrieve the key from the server safely, reliably, and securely though
2. You have a login sequence of some sort (even the original login sequence you do for remote access). You can do two runs of your key generator on the same password. How this works is that you derive the key twice with a new salt and a new secure initialization vector. You store one of those generated

passwords on the device, and you use the second password as the AES key.

When you log in, you re-derive the key on the local login and compare it to the stored key. Once that is done, you use derive key #2 for AES.

1. Using the "generally safe" approach, you encrypt the data using AES and store the key in `MODE_PRIVATE`. This is recommended by a recent-ish Android blog post. Not incredibly secure, but way better for some people over plain text

You can do a lot of variations of these. For example, instead of a full login sequence, you can do a quick PIN (derived). The quick PIN might not be as secure as a full login sequence, but it's many times more secure than plain text

Share Follow

edited May 14, 2013 at 13:30

answered May 14, 2013 at 13:21



Joe Plante

6,368 ● 2 ● 31 ● 23



5

I know this is a little bit of necromancy, but you should use the Android [AccountManager](#). It's purpose-built for this scenario. It's a little bit cumbersome but one of the things it does is invalidate the local credentials if the SIM card changes, so if somebody swipes your phone and



throws a new SIM in it, your credentials won't be compromised.



This also gives the user a quick and easy way to access (and potentially delete) the stored credentials for any account they have on the device, all from one place.

[SampleSyncAdapter](#) is an example that makes use of stored account credentials.

Share Follow

answered Apr 4, 2012 at 18:04



Jon O

6,591 ● 1 ● 48 ● 58

- 
- 2 Please note that using the AccountManager is not more secure than any other method provided above!  
[developer.android.com/training/id-auth/...](#) – Sander Versluys  
Nov 8, 2012 at 10:14
- 
- 1 The use case for AccountManager is when the account has to be shared between different apps, and apps from different authors. Storing the password and giving it to any requesting app would not be appropriate. If the usage of the user/password is only for a single app, don't use AccountManager. – dolmen Nov 12, 2012 at 0:34
- 
- 1 @dolmen, that's not quite correct. The AccountManager won't give the account password to any app whose UID doesn't match the Authenticator's. The name, yes; the auth token, yes; the password, no. If you try, it'll throw a SecurityException. And the use case is much broader than that. [developer.android.com/training/id-auth/identify.html](#)  
– Jon O Nov 12, 2012 at 15:37
-





5



I'll throw my hat into the ring just to talk about securing passwords in general on Android. On Android, the device binary should be considered compromised - this is the same for any end application which is in direct user control. Conceptually, a hacker could use the necessary access to the binary to decompile it and root out your encrypted passwords and etc.

As such there's two suggestions I'd like to throw out there if security is a major concern for you:

1) Don't store the actual password. Store a granted access token and use the access token and the signature of the phone to authenticate the session server-side. The benefit to this is that you can make the token have a limited duration, you're not compromising the original password and you have a good signature that you can use to correlate to traffic later (to for instance check for intrusion attempts and invalidate the token rendering it useless).

2) Utilize 2 factor authentication. This may be more annoying and intrusive but for some compliance situations unavoidable.

Share Follow

answered Oct 30, 2014 at 17:00



dcgregorya

53 ● 1 ● 3



*This is a supplemental answer for those arriving here based on the question title (like I did) and don't need to*



# How to use Shared Preferences

User settings are generally saved locally in Android using `SharedPreferences` with a key-value pair. You use the `String` key to save or look up the associated value.

## Write to Shared Preferences

```
String key = "myInt";
int valueToSave = 10;

SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(context)
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(key, valueToSave).commit();
```

Use `apply()` instead of `commit()` to save in the background rather than immediately.

## Read from Shared Preferences

```
String key = "myInt";
int defaultValue = 0;

SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(context)
int savedValue = sharedPref.getInt(key, defaultValue);
```

The default value is used if the key isn't found.

## Notes

- Rather than using a local key String in multiple places like I did above, it would be better to use a constant in a single location. You could use something like this at the top of your settings activity:

```
final static String PREF_MY_INT_KEY = "myInt";
```

- I used an `int` in my example, but you can also use `putString()`, `putBoolean()`, `getString()`, `getBoolean()`, etc.
- See the [documentation](#) for more details.
- There are multiple ways to get SharedPreferences. See [this answer](#) for what to look out for.

Share Follow

edited Jun 20, 2020 at 9:12



Community Bot

1 • 1

answered Oct 25, 2017 at 2:32



Suragch

510k • 335 • 1.4k • 1.4k



You can also check out this little lib, containing the functionality you mention.



<https://github.com/kovmarci86/android-secure-preferences>



It is similar to some of the other approaches here. Hope helps :)

Share Follow

answered Sep 21, 2013 at 21:51



Marcell

981 ● 1 ● 6 ● 13



1



This answer is based on a suggested approach by Mark. A custom version of the EditTextPreference class is created which converts back and forth between the plain text seen in the view and an encrypted version of the password stored in the preferences storage.



As has been pointed out by most who have answered on this thread, this is not a very secure technique, although the degree of security depends partly on the encryption/decryption code used. But it's fairly simple and convenient, and will thwart most casual snooping.

Here is the code for the custom EditTextPreference class:

```
package com.Merlinia.OutBack_Client;

import android.content.Context;
import android.preference.EditTextPreference;
import android.util.AttributeSet;
import android.util.Base64;

import com.Merlinia.MEncryption_Main.MEncryptionUserPa
```

```

/**
 * This class extends the EditTextPreference view, providing
 * decryption services for
 * OutBack user passwords. The passwords in the preferences are
 * encrypted using the
 * MEncryption classes and then converted to strings so that the
 * preferences store can not
 * store byte arrays.
 *
 * This is largely copied from this article, except for the
 * encryption/decryption parts:
 * https://groups.google.com/forum/#!topic/android-dev
 */
public class EditPasswordPreference extends EditTextPreference {

    // Constructor - needed despite what compiler says
    public EditPasswordPreference(Context context) {
        super(context);
    }

    // Constructor - needed despite what compiler says
    public EditPasswordPreference(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    // Constructor - needed despite what compiler says
    public EditPasswordPreference(Context context, AttributeSet attrs,
int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    /**
     * Override the method that gets a preference from the
     * for display by the
     * EditText view. This gets the base64 password, converts it to a
     * array, and then decrypts
     * it so it can be displayed in plain text.
     * @return OutBack user password in plain text
     */
    @Override
    public String getText() {

```

```

        String decryptedPassword;

        try {
            decryptedPassword = MEncryptionUserPasswor
                Base64.decode(getSharedPreferences
""), Base64.DEFAULT));
        } catch (Exception e) {
            e.printStackTrace();
            decryptedPassword = "";
        }

        return decryptedPassword;
    }

    /**
     * Override the method that gets a text string fro
     stores the value in
     * the preferences storage. This encrypts the pass
     and then encodes that
     * in base64 format.
     * @param passwordText OutBack user password in p
     */
    @Override
    public void setText(String passwordText) {
        byte[] encryptedPassword;

        try {
            encryptedPassword =
MEncryptionUserPassword.aesEncrypt(passwordText);
        } catch (Exception e) {
            e.printStackTrace();
            encryptedPassword = new byte[0];
        }

        getSharedPreferences().edit().putString(getKey
Base64.encodeToString(encryptedPassword, Base64.DEFAUL
        .commit());
    }

    @Override
    protected void onSetInitialValue(boolean restoreVa

```

```

{
    if (restoreValue)
        getEditText().setText(getText());
    else
        super.onSetInitialValue(restoreValue, defa
}
}

```

This shows how it can be used - this is the "items" file that drives the preferences display. Note it contains three ordinary EditTextPreference views and one of the custom EditPasswordPreference views.

```

<PreferenceScreen xmlns:android="http://schemas.android

    <EditTextPreference
        android:key="@string/useraccountname_key"
        android:title="@string/useraccountname_title"
        android:summary="@string/useraccountname_summa
        android:defaultValue="@string/useraccountname_
    />

    <com.Merlinia.OutBack_Client.EditPasswordPreferenc
        android:key="@string/useraccountpassword_key"
        android:title="@string/useraccountpassword_tit
        android:summary="@string/useraccountpassword_s
        android:defaultValue="@string/useraccountpassw
    />

    <EditTextPreference
        android:key="@string/outbackserverip_key"
        android:title="@string/outbackserverip_title"
        android:summary="@string/outbackserverip_summa
        android:defaultValue="@string/outbackserverip_
    />

    <EditTextPreference
        android:key="@string/outbackserverport_key"
        android:title="@string/outbackserverport_title
        android:summary="@string/outbackserverport_sum
        android:defaultValue="@string/outbackserverpor

```

```
/>
```

```
</PreferenceScreen>
```

As for the actual encryption/decryption, that is left as an exercise for the reader. I'm currently using some code based on this article

<http://zenu.wordpress.com/2011/09/21/aes-128bit-cross-platform-java-and-c-encryption-compatibility/>, although with different values for the key and the initialization vector.

Share Follow

answered Sep 6, 2013 at 5:49



RenniePet

11.6k ● 8 ● 83 ● 107



1



First of all I think User's data shouldn't be stored on phone, and if it is must to store data somewhere on the phone it should be encrypted with in the apps private data. Security of users credentials should be the priority of the application.



The sensitive data should be stored securely or not at all. In the event of a lost device or malware infection, data stored insecurely can be compromised.

Share Follow

answered Dec 3, 2013 at 10:40



Yauraw Gadav

1,746 ● 1 ● 19 ● 39





1

I use the Android KeyStore to encrypt the password using RSA in ECB mode and then save it in the SharedPreferences.



When I want the password back I read the encrypted one from the SharedPreferences and decrypt it using the KeyStore.



With this method you generate a public/private Key-pair where the private one is safely stored and managed by Android.

Here is a link on how to do this: [Android KeyStore Tutorial](#)

Share Follow

answered Nov 7, 2018 at 13:23



Braveblacklion

11 ● 1



1

As others already pointed out you can use SharedPreferences generally but if you would like to store data encrypted it's a bit inconvenient. Fortunately, there is an easier and quicker way to encrypt data now since there is an implementation of SharedPreferences that encrypts keys and values. You can use **EncryptedSharedPreferences** in Android JetPack Security.



Just add AndroidX Security into your build.gradle:

```
implementation 'androidx.security:security-crypto:1.0.
```

And you can use it like this:

```
String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.  
  
SharedPreferences sharedPreferences = EncryptedSharedPreferences.  
    "secret_shared_prefs",  
    masterKeyAlias,  
    context,  
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.  
    EncryptedSharedPreferences.PrefValueEncryptionScheme.  
);  
  
// use the shared preferences and editor as you normal  
SharedPreferences.Editor editor = sharedPreferences.ed
```

See more details: <https://android-developers.googleblog.com/2020/02/data-encryption-on-android-with-jetpack.html>

Official docs:

<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>

Share Follow

answered Feb 18, 2021 at 5:30



becgabi

51 ● 3



This is how I am doing it.

0

This does not give errors in strict mode.



```
public class UserPreferenceUtil  
{  
  
    private static final String THEME = "THEME";
```



```
private static final String LANGUAGE = "LANGUAGE"

public static String getLanguagePreference(Context context)
{
    String lang = getPreferenceByKey(context, LANGUAGE);
    if( lang==null || "System".equalsIgnoreCase(lang) )
    {
        return null;
    }

    return lang;
}

public static void saveLanguagePreference(Context context, String value)
{
    savePreferenceKeyValue(context, LANGUAGE, value);
}

public static String getThemePreference(Context context)
{
    return getPreferenceByKey(context, THEME);
}

public static void saveThemePreference(Context context, String value)
{
    savePreferenceKeyValue(context, THEME, value);
}

public static String getPreferenceByKey(Context context, String preferenceKey )
{
    SharedPreferences sharedPreferences = PreferenceManager.getDefaultSharedPreferences(context);

    String value = sharedPreferences.getString(preferenceKey, null);

    return value;
}
```

```
private static void savePreferenceKeyValue(Context
preferenceKey, String value)
{
    SharedPreferences sharedPreferences =
PreferenceManager.getDefaultSharedPreferences(context)
    SharedPreferences.Editor editor = sharedPrefer
editor.putString(preferenceKey, value);
editor.apply();

}

}
```

My app does not need a password. However, rather than saving passwords or encrypted passwords, I would save a one-way hash. When the user logs in, I will hash the input the same way and match it with the stored hash.

Share Follow

edited Jun 11, 2022 at 15:24

answered May 28, 2022 at 15:37



Sandeep Dixit

1,026 ● 9 ● 14



-3



you need to use the sqlite, security apit to store the passwords. here is best example, which stores passwords, -- passwordsafe. here is link for the source and explanation -- <http://code.google.com/p/android-passwordsafe/>



Share Follow

edited Dec 11, 2009 at 8:45



Artem Russakovskii

22k ● 18 ● 97 ● 118

answered Apr 27, 2009 at 6:09



xyz

- 
- 4 The OP needs to store one username and password pair. It would be ridiculous to consider creating an entire database table for this one use – [HXCaine](#) May 25, 2010 at 14:27
- 

@HXCaine i respectfully disagree - i can see at least 1 other use of a user/passwords sqlite table. IF YOU CONSIDER THE RISK (of using sqlite) ACCEPTABLE, besides simple application login authentication, you could use the table to store multiple ftp passwords (if your app uses ftp - mine do sometimes), for example. plus, creating a sqlite adapter class for this manipulation is boilerplate simple. – [tony gil](#) Jul 22, 2012 at 14:40

---

Nice resurrection of a two-year-old comment! To be fair, my comment was a year after the answer :) Even with a handful of FTP passwords, the overhead is much larger with an SQLite table than with SharedPreferences both in terms of space and coding. Surely that can't be necessary – [HXCaine](#) Jul 24, 2012 at 21:01

---



-3



shared preferences is easiest way to store our application data. but it is possible that anyone can clear our shared preferences data through application manager.so i don't think it is completely safe for our application.

Share Follow



answered Oct 18, 2013 at 8:41



Rohit Jain

229 ● 1 ● 4 ● 7

