# Leaking Private Data on the SD Card

```java
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    public void leakPrivateDataToSDCard(String user, String pass) throws
IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

# Leaking Private Data on the SD Card

```
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    public void leakPrivateDataToSDCard(String user, String pass) throws 
IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

# Leaking Private Data on the SD Card

```
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    public void leakPrivateDataToSDCard(String user, String pass) throws
IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

# Leaking Private Data on the SD Card

Security Issue! Other apps can read files stored on external storage.

```
public class PrivateDataExposerActivity

    @Override
    protected void onCreate(Bundle savec
        super.onCreate(savedInstanceStat
    }

    public void leakPrivateDataToSDCard(String user, String pass) throws
IOException {
        // Get the location of the SD card
        File sdCard = Environment getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

# Leaking Private Data on the SD Card

You should also avoid storing passwords on disk!

```
public class PrivateDataExposerActivity

    @Override
    protected void onCreate(Bundle save
        super.onCreate(savedInstanceSta
    }

    public void leakPrivateDataToSDCard(String user, String pass) throws
IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

# Leaking Private Data on the SD Card

If you must store something sensitive to disk, always encrypt or hash it and store it in a private location!

```java
public class PrivateDataExposerActivity

    @Override
    protected void onCreate(Bundle save
        super.onCreate(savedInstanceSta
    }

    public void leakPrivateDataToSDCard(String user, String pass) throws
IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

# Leaking Private Data on the SD Card

```
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCreate(Bu                        {
        super.onCreate(savedI
    }

    public void leakPrivateDat                        tring pass) throws
IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File privateFile = new File (sdCard, "myData.hidden");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileWriter f = new FileWriter(privateFile);
        f.write("user="+user+"\npass="+pass);
    }
```

Only write non-private / sensitive data to external storage.

# An OK Use of External Storage

```
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    public void storePhotoToSDCard(byte[] data) throws IOException {
        // Get the location of the SD card
        File sdCard = Environment.getExternalStorageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File publicPhoto = new File (sdCard, ”benignPhoto.jpg");

        // Make sure that the photo is really something that the
        // user wants to be public!
        FileOutputStream fout = new FileOutputStream(publicPhoto);
        fout.write(publicPhoto);
        …
    }
```

# An OK Use of External Storage

```
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCre                          tate) {
        super.onCreate(s
    }

    public void storePho                          hrows IOException {
        // Get the locat
        File sdCard = En                          ageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File publicPhoto = new File (sdCard, ”benignPhoto.jpg");

        // Make sure that the photo is really something that the
        // user wants to be public!
        FileOutputStream fout = new FileOutputStream(publicPhoto);
        fout.write(publicPhoto);
        …
    }
```

Make sure that anything you save is really OK for any app to read or potentially steal!

# Be Certain the Data is Benign!

```
public class PrivateDataExposerActivity extends Activity {

    @Override
    protected void onCre                          tate) {
        super.onCreate(s
    }

    public void storePho                          rows IOException {
        // Get the locat
        File sdCard = En                          ageDirectory();

        // Choose a file name for the data to be saved to the SD card
        File publicPhoto = new File (sdCard, ”privatePhoto.jpg");

        // Security issue! Private data is being written to the SD card, which
        // can be read by any app!
        FileOutputStream fout = new FileOutputStream(publicPhoto);
        fout.write(publicPhoto);
        …
    }
```

Saving photos to the SD card probably doesn’t make sense for SnapChat!

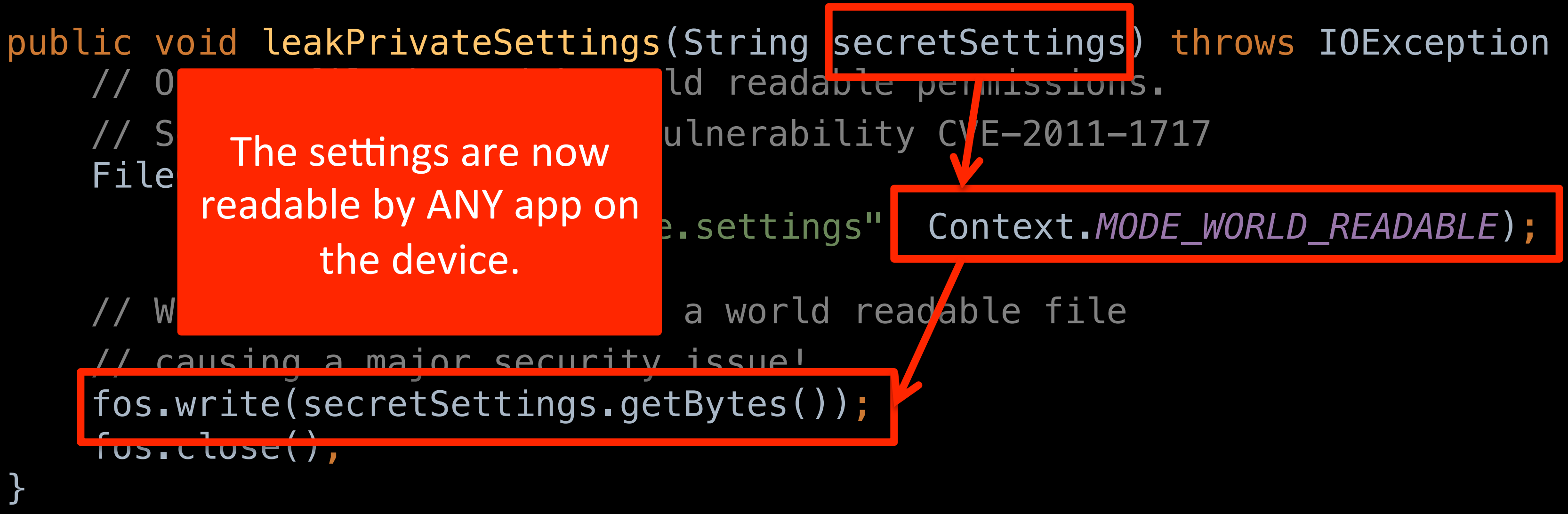
# Leaking Data with Bad File Permissions

```
public void leakPrivateSettings(String secretSettings) throws IOException {
    // Open a file but with world readable permissions.
    // See the Skype security vulnerability CVE-2011-1717
    FileOutputStream fos =
         openFileOutput(“private.settings", Context.MODE_WORLD_READABLE);

    // Write secret settings to a world readable file
    // causing a major security issue!
    fos.write(secretSettings.getBytes());
    fos.close();
}
```

# Leaking Data with Bad File Permissions

```
public void leakPrivateSettings(String secretSettings) throws IOException {
    // Open a file but with world readable permissions.
    // See the Skype security vulnerability CVE-2011-1717
    FileOutputStream fos =
        openFileOutput(“private.settings", Context.MODE_WORLD_READABLE);

    // Write secret settings to a world readable file
    // causing a major security issue!
    fos.write(secretSettings.getBytes());
    fos.close();
}
```

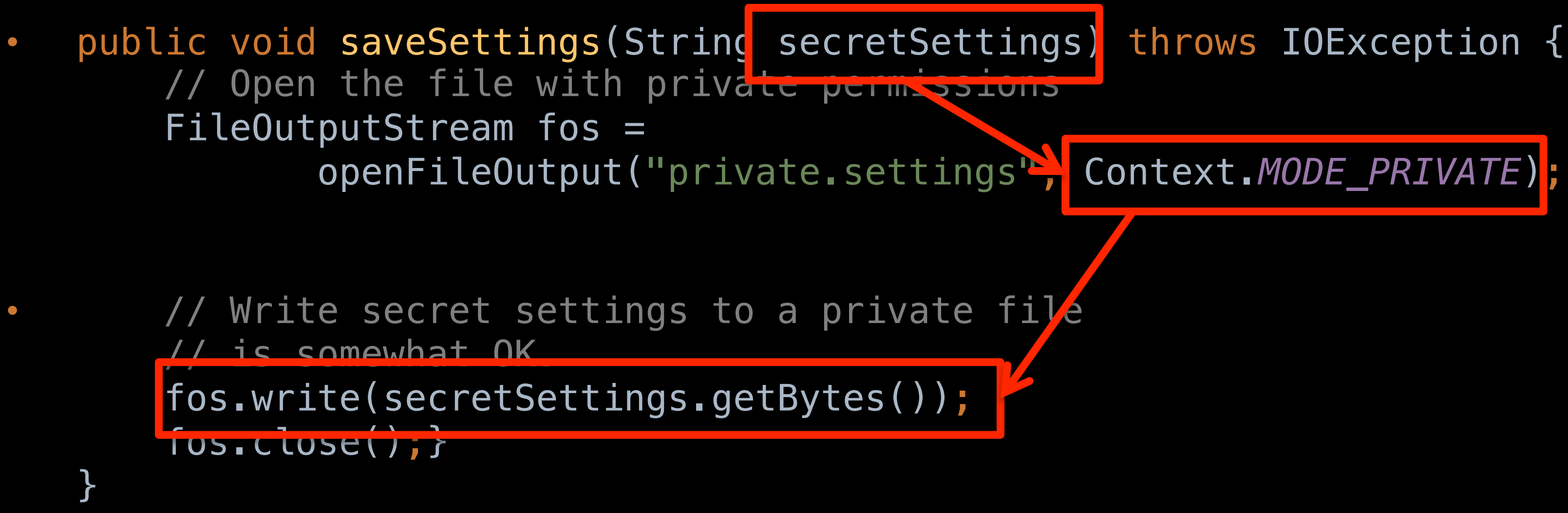
# Leaking Data with Bad File Permissions



```
public void leakPrivateSettings(String secretSettings) throws IOException {
    // O                        ld readable permissions.
    // S                        ulnerability CVE-2011-1717
    File
                                e.settings"  Context.MODE_WORLD_READABLE);

    // W                          a world readable file
    // causing a major security issue!
    fos.write(secretSettings.getBytes());
    fos.close();
}
```

The settings are now readable by ANY app on the device.

# A Better Version of Saving Settings



```
public void saveSettings(String secretSettings) throws IOException {
    // Open the file with private permissions
    FileOutputStream fos =
           openFileOutput("private.settings", Context.MODE_PRIVATE);


    // Write secret settings to a private file
    // is somewhat OK.
    fos.write(secretSettings.getBytes());
    fos.close();}
}
```

# A Better Version of Saving Settings

```
• public void saveSettings(String secretSettings) throws IOException {
      /                          ivate permissions
      F
                                 rivate.settings", Context.MODE_PRIVATE);

•     /                          to a private file
      // is somewhat OK
      fos.write(secretSettings.getBytes());
      fos.close();}
  }
```

Remember, your private data is potentially readable from the user of the device!

# Developer Data is Not Secure on User Devices

```
public void saveSettings(String secretSettings) throws IOException {
    // Open the file with private permissions
    FileOutputStream fos =
           openFileOutput("private.settings", Context.MODE_PRIVATE);

    // Private from other apps but NOT the user of the device!
    secretSettings += "Developer's AmazonAWSPrivateKey=123456";

    // Write secret settings to a private file
    // is somewhat OK.
    fos.write(secretSettings.getBytes());
    fos.close();}
}
```

# Developer Data is Not Secure on User Devices

Private data is still potentially accessible by the user of the device!

```
public void saveSettings(String s                    OException {
    // Open the file with private
    FileOutputStream fos =
           openFileOutput("privat                    DE_PRIVATE);

    // Private from other apps but NOT the user of the device!
    secretSettings += "Developer's AmazonAWSPrivateKey=123456";

    // Write secret settings to a private file
    // is somewhat OK.
    fos.write(secretSettings.getBytes());
    fos.close();}
}
```

# Developer Data is Not Secure on User Devices

```
public void saveSettings(St                       rows IOException {
    // Open the file with p
    FileOutputStream fos =
           openFileOutput("                       ext.MODE_PRIVATE);

    // Private from other apps but NOT the user of the device!
    secretSettings += "Developer's AmazonAWSPrivateKey=123456";

    // Write secret settings to a private file
    // is somewhat OK.
    fos.write(secretSettings.getBytes());
    fos.close();}
}
```

Don’t store your private developer data on other people’s devices!

# But I Would Never Accidentally Leak Data…

# Can You Spot the Security Flaw Below?

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new PrivateStorageAdapter());
    handlerMapping_.put("group", new PublicStorageAdapter());
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the data is stored privately
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

# Can You Spot the Security Flaw Below?

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append ".puser" to the key to ensure that the
    handlerMapping_.put("puser", new PrivateStorageAdapt
    handlerMapping_.put("group", new PublicStorageAdapte
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append ".puser" to the key to ensure that the data is stored privately
    storeData("name.puser", "private stuff…");
    storeData("private.groups.puser", "private stuff…");
    storeData("address.puser", "private stuff…");

    // We append ".group" to the key to allow the data to be stored publicly
    storeData("profilephoto.group", "public stuff…");
    storeData("homepageurl.group", "public stuff…");
}
```

Find the suffix
(e.g., .puser or .group)

# Can You Spot the Security Flaw Below?

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new Private
    handlerMapping_.put("group", new PublicS
}

public void storeData(String key, String dat
    int start = key.indexOf(".");
    String type = key.substring(start + 1, s

    // Find and use the appropriate storag
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the data is stored privately
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

Based on the suffix, decide where to store the data (e.g., .puser is private)

# Can You Spot the Security Flaw Below?

```java
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append ".puser" to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new PrivateStorageAdapter());
    handlerMapping_.put("group", new PublicStorageAdapter());
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append ".puser" to the key to ensure that the data is stored privately
    storeData("name.puser", "private stuff…");
    storeData("private.groups.puser", "private stuff…");
    storeData("address.puser", "private stuff…");

    // We append ".group" to the key to allow the data t
    storeData("profilephoto.group", "public stuff…");
    storeData("homepageurl.group", "public stuff…");
}
```

This data is supposed to be stored privately (hence the ".puser")

# Can You Spot the Security Flaw Below?

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append ".puser" to the key to ensure that the
    handlerMapping_.put("puser", new PrivateStorageAdapt
    handlerMapping_.put("group", new PublicStorageAdapte
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append ".puser" to the key to ensure that the data is stored privately
    storeData("name.puser", "private stuff…");
    storeData("private.groups.puser", "private stuff…");
    storeData("address.puser", "private stuff…");

    // We append ".group" to the key to allow the data t
    storeData("profilephoto.group", "public stuff…");
    storeData("homepageurl.group", "public stuff…");
}
```

But this expression resolves to "group" when applied to "private.groups.puser"

This data is supposed to be stored privately (hence the ".puser")

# Can You Spot the Security Flaw Below?

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append ".puser" to the key to ensure that the
    handlerMapping_.put("puser", new PrivateStorageAdapt
    handlerMapping_.put("group", new PublicStorageAdapte
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append ".puser" to the key to ensure that the data is stored privately
    storeData("name.puser", "private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append ".group" to the key to allow the data t
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

Which causes the data to be stored on the SD card, which isn't private

But this expression resolves to "group" when applied to "private.groups.puser"

This data is supposed to be stored privately (hence the ".puser")

# Rules for More Secure Android Coding

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new PrivateStorageAdapter());
    handlerMapping_.put("group", new PublicStorageAdapter());
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the data is stored privately
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

# 1. Avoid Coupling Security State to Data State

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append ".puser" to the key to ensure that the
    handlerMapping_.put("puser", new PrivateStorageAdapt
    handlerMapping_.put("group", new PublicStorageAdapte
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append ".puser" to the key to ensure that the data is stored privately
    storeData("name.puser", "private stuff…");
    storeData("private.groups.puser", "private stuff…");
    storeData("address.puser", "private stuff…");

    // We append ".group" to the key to allow the data to be stored publicly
    storeData("profilephoto.group", "public stuff…");
    storeData("homepageurl.group", "public stuff…");
}
```

Data and security are combined in the "key" variable

# 1. Avoid Coupling Security State to Data State

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the
    handlerMapping_.put("puser", new PrivateStorageAdapt
    handlerMapping_.put("group", new PublicStorageAdapte
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for the data
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the data is stored privately
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

The key has both a data storage and a security meaning

# 1. Avoid Coupling Security State to Data State

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new PrivateStorageAdapter());
    handlerMapping_.put("group", new PublicStorageAdapter());
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for th
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the data is stored privately
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

Changing the data key changes the security!

# 1. Avoid Coupling Security State to Data State

```
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new PrivateStorageAdapter());
    handlerMapping_.put("group", new PublicStorageAdapter());
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for th
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the da
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

It is very hard to prove that this code will work for every possible data value! Tightly coupling data and security state leads to hard to spot security issues.

# 1. Avoid Coupling Security State to Data State

```java
private interface StorageHandler {
    public void store(String data);
}

private class PublicStorageAdapter implements StorageHandler{
    public void store(String data){ storeOnSDCard(data); }
}

private class PrivateStorageAdapter implements StorageHandler{
    public void store(String data){ storeInPrivateData(data); }
}

private Map<String, StorageHandler> handlerMapping_ = new HashMap<String, StorageHandler>();

public void initHandlers(){
    // We append “.puser” to the key to ensure that the data is stored privately
    handlerMapping_.put("puser", new PrivateStorageAdapter());
    handlerMapping_.put("group", new PublicStorageAdapter());
}

public void storeData(String key, String data){
    int start = key.indexOf(".");
    String type = key.substring(start + 1, start + 6);

    // Find and use the appropriate storage handler for th
    handlerMapping_.get(type).store(data);
}

public void saveSettings(){
    // We append “.puser” to the key to ensure that the da
    storeData("name.puser", ”private stuff…");
    storeData("private.groups.puser", ”private stuff…");
    storeData("address.puser", ”private stuff…");

    // We append “.group” to the key to allow the data to be stored publicly
    storeData("profilephoto.group", ”public stuff…");
    storeData("homepageurl.group", ”public stuff…");
}
```

Coupling data to security state also allows attackers to potentially manipulate your security by changing input data

# 1. Avoid Coupling Security State to Data State

Data State

Security State

```
public void storeData(String key, String data, boolean isSecure){

    if(isSecure){
        getSecureHandler().store(key,data);
    }
    else {
        getPublicHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, true);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, false);
    storeData("homepageurl.group", publicUrl, false);
}
public StorageHandler getSecureHandler(){ return handlerMapping_.get("secure");}
public StorageHandler getPublicHandler(){ return handlerMapping_.get("public");}
```

This variant is an improvement because the security state isn't coupled to the data state

# 1. Avoid Coupling Security State to Data State

```
…
public void storeData(String key, String data, boolean isSecure){

    if(isSecure){
        getSecureHandler().store(key,data);
    }
    else {
        getPublicHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, true);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, false);
    storeData("homepageurl.group", publicUrl, false);
}
public StorageHandler getSecureHandler(){ return handlerMapping_.get("secure");}
public StorageHandler getPublicHandler(){ return handlerMapping_.get("public");}
```

But…we could still do better…

# 2. Make Highest Security the Default Level

```
…
public void storeData(String key, String data, boolean isSecure){

    if(isSecure){
        getSecureHandler().store(key,data);
    }
    else {
        getPublicHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, true);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, false);
    storeData("homepageurl.group", publicUrl, false);
}
public StorageHandler getSecureHandler(){ return handlerMapping_.get("secure");}
public StorageHandler getPublicHandler(){ return handlerMapping_.get("public");}
```

The default value of a boolean is false, so we default to insecure storage

# 2. Make Highest Security the Default Level

```
…
public void storeData(String key, String data, boolean isPublic){

    if(isPublic){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, false);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, true);
    storeData("homepageurl.group", publicUrl, true);
}
public StorageHandler getSecureHandler(){ return handlerMapping_.get("secure");}
public StorageHandler getPublicHandler(){ return handlerMapping_.get("public");}
```

This variant is better because secure storage is the default unless the data is explicitly declared public

# 2. Make Highest Security the Default Level

```
…
public void storeData(String key, String data, boolean isPublic){

    if(isPublic){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, false);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, true);
    storeData("homepageurl.group", publicUrl, true);
}
public StorageHandler getSecureHandler(){ return handlerMapping_.get("secure");}
public StorageHandler getPublicHandler(){ return handlerMapping_.get("public");}
```

But… we could still do better…

# 3. Make the Security Level Clear in the Interface / Naming

```
…
public void storeData(String key, String data, boolean isPublic){

    if(isPublic){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, false);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, true);
    storeData("homepageurl.group", publicUrl, true);
}
```

It isn't obvious that this is storing data securely

# 3. Make the Security Level Clear in the Interface / Naming

```
public enum SecurityLevel{ MAX, NONE }

public void storeData(String key, String data, SecurityLevel security){

    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, SecurityLevel.MAX);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, SecurityLevel.NONE);
    storeData("homepageurl.group", publicUrl, SecurityLevel.NONE);
}
```

This variant is better because you can more easily see the security level when auditing the data storage code

# 3. Make the Security Level Clear in the Interface / Naming

```
public enum SecurityLevel{ MAX, NONE }

public void storeData(String key, String data, SecurityLevel security){

    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, SecurityLevel.MAX);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, SecurityLevel.NONE);
    storeData("homepageurl.group", publicUrl, SecurityLevel.NONE);
}
```

It also still defaults to secure storage unless security is explicitly turned off

# 4. Bound Security State & Strongly Type It

```java
private static final int MAX_SECURITY = 2;
private static final int NO_SECURITY = 1;

public void storeData(String key, String data, int security){

    if(security == NO_SECURITY){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

What if we had fixed the naming problem like this?

# 4. Bound Security State & Strongly Type It

```
private static final int MAX_SECURITY = 2;
private static final int NO_SECURITY = 1;

public void storeData(String key, String data, int security){

    if(security == NO_SECURITY){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

Our security state can now be any integer value and this code needs a lot of testing of boundary conditions!

# 4. Bound Security State & Strongly Type It

```java
private static final int MAX_SECURITY = 2;
private static final int NO_SECURITY = 1;

public void storeData(String key, String data, int security){

    if(security == NO_SECURITY){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

But why would we test every value, it is obvious what will happen…

# 4. Bound Security State & Strongly Type It

```
private static final int MAX_SECURITY = 2;
private static final int NO_SECURITY = 1;
private static final int SECURITY_LEVEL_REQUIRES_ENCRYPTION = 3;

public void storeData(String key, String data, int security){

    if(security < SECURITY_LEVEL_REQUIRES_ENCRYPTION){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

Unless someone begins refactoring the code and makes a mistake…

# 4. Bound Security State & Strongly Type It

```java
private static final int MAX_SECURITY = Integer.MAX_VALUE;
private static final int NO_SECURITY = 1;

public void storeData(String key, String data, int security){

    if(security == NO_SECURITY){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){
    int security = MAX_SECURITY;
    // I am really paranoid, let’s increase security!
    security++;

    // Secure stuff
    storeData("private.groups.puser", privateGroups, security);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

Our security state can be any integer value… So what happens when someone provides an undefined value?

# 4. Bound Security State & Strongly Type It

```
private static final int MAX_SECURITY = Integer.MAX_VALUE;
private static final int NO_SECURITY = 1;

public void storeData(String key, String data, int security){

    if(security == NO_SECURITY){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){
    int security = privateGroups.size();

    // Secure stuff
    storeData("private.groups.puser", privateGroups, security);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

Also, the loose typing doesn’t enforce separation of security state and data state!

# 4. Bound Security State & Strongly Type It

```
public enum SecurityLevel{ MAX, NONE }
```

```
public void storeData(String key, String data, SecurityLevel security){
    // The security variable has at most 2 possible values (and its provable)
    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){
    SecurityLevel level = 3; // Compile error
    SecurityLevel groups = privateGroups.size() // Compile error
    SecurityLevel paranoid = SecurityLevel.MAX + 1; // Compile error
    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

With this approach, our security state is strongly typed, bounded, and can't easily be mixed with data.

# 4. Bound Security State & Strongly Type It

```java
public enum SecurityLevel{ MAX, NONE }

public void storeData(String key, String data, SecurityLevel security){
    // The security variable has at most 2 possible values (and its provable)
    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){
    SecurityLevel level = 3; // Compile error
    SecurityLevel groups = privateGroups.size() // Compile error
    SecurityLevel paranoid = SecurityLevel.MAX + 1; // Compile error
    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

Attempts to use incorrect values or mix with data state are compile errors

# 4. Bound Security State & Strongly Type It

```
public enum SecurityLevel{ MAX, NONE }

public void storeData(String key, String data, SecurityLevel security){
    // The security variable has at most 2 possible values (and its provable)
    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){
    SecurityLevel level = 3; // Compile error
    SecurityLevel groups = privateGroups.size() // Compile error
    SecurityLevel paranoid = SecurityLevel.MAX + 1; // Compile error
    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

The enum also simplifies testing since there are at most 2 possible values that can be provided to this method

# 4. Bound Security State & Strongly Type It

```
public enum SecurityLevel{ MAX, NONE }

public void storeData(String key, String data, SecurityLevel security){
    // The security variable has at most 2 possible values (and its provable)
    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){
    SecurityLevel level = 3; // Compile error
    SecurityLevel groups = privateGroups.size() // Compile error
    SecurityLevel paranoid = SecurityLevel.MAX + 1; // Compile error
    // Secure stuff
    storeData("private.groups.puser", privateGroups, MAX_SECURITY);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, NO_SECURITY);
    storeData("homepageurl.group", publicUrl, NO_SECURITY);
}
```

…But…we could still do better…

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeData(String key, String data, SecurityLevel security){

    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    SecurityLevel groupsec = SecurityLevel.NONE;
    SecurityLevel profilesec = SecurityLevel.MAX;

    // Secure stuff
    storeData("private.groups.puser", privateGroups, groupsec);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, profilesec);
    storeData("homepageurl.group", publicUrl, profilesec);
}
```

This conditional logic makes it possible that a developer could accidentally pass the wrong SecurityLevel or that the conditional logic could be wrong

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeData(String key, String data, SecurityLevel security){

    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    SecurityLevel groupsec = SecurityLevel.NONE;
    SecurityLevel profilesec = SecurityLevel.MAX;

    // Secure stuff
    storeData("private.groups.puser", privateGroups, groupsec);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, profilesec);
    storeData("homepageurl.group", publicUrl, profilesec);
}
```

It is also harder to look at the code and know that it isn't storing data inappropriately

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeData(String key, String data, SecurityLevel security){

    if(security == SecurityLevel.NONE){
        getPublicHandler().store(key,data);
    }
    else {
        getSecureHandler().store(key,data);
    }
}

public void saveSettings(){

    // Secure stuff
    SecurityLevel groupsec = SecurityLevel.NONE;
    SecurityLevel profilesec = SecurityLevel.MAX;

    // Secure stuff
    storeData("private.groups.puser", privateGroups, groupsec);


    // Insecure stuff
    storeData("profilephoto.group", publicPhoto, profilesec);
    storeData("homepageurl.group", publicUrl, profilesec);
}
```

Finally, a new user of the API might not know what the "levels" mean or pass the wrong level

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String key, String data){
    getPublicSDCardStorageHandler().store(key,data);
}



public void saveSettings(){

    // Secure stuff
    storeDataPrivatelyAndEncrypted("private.groups.puser", privateGroups);


    // Insecure stuff
    storeDataPubliclyOnSDCard("profilephoto.group", publicPhoto);
    storeDataPubliclyOnSDCard("homepageurl.group", publicUrl);
}
```

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String k
    getPublicSDCardStorageHandler().store(key,
}



public void saveSettings(){

    // Secure stuff
    storeDataPrivatelyAndEncrypted("private.groups.puser", privateGroups);


    // Insecure stuff
    storeDataPubliclyOnSDCard("profilephoto.group", publicPhoto);
    storeDataPubliclyOnSDCard("homepageurl.group", publicUrl);
}
```

This variant has less conditional logic to mess up

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPublicly(
    getPublicSDCardStorageHan(
}



public void saveSettings(){

    // Secure stuff
    storeDataPrivatelyAndEncrypted("private.groups.puser", privateGroups);


    // Insecure stuff
    storeDataPubliclyOnSDCard("profilephoto.group", publicPhoto);
    storeDataPubliclyOnSDCard("homepageurl.group", publicUrl);
}
```

It is also clear to developers what the code is doing and a new API user is less likely to make a mistake

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String key, St
    getPublicSDCardStorageHandler().store(key,data);
}



public void saveSettings(){

    // Secure stuff
    storeDataPrivatelyAndEncrypted("private.groups.puser", privateGroups);


    // Insecure stuff
    storeDataPubliclyOnSDCard("profilephoto.group", publicPhoto);
    storeDataPubliclyOnSDCard("homepageurl.group", publicUrl);
}
```

A security audit of this code is going to be a lot easier

# 5. Avoid Conditional Logic in Security Pathways

```
…
public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String key, St
    getPublicSDCardStorageHandler().store(key,data);
}



public void saveSettings(){

    // Secure stuff
    storeDataPrivatelyAndEncrypted("private.groups.puser", privateGroups);


    // Insecure stuff
    storeDataPubliclyOnSDCard("profilephoto.group", publicPhoto);
    storeDataPubliclyOnSDCard("homepageurl.group", publicUrl);
}
```

But..we could still do better…

# 6. Don't Allow Secure Pathways to be Compromised by Configuration Changes

```
private Map<String, StorageHandler> handlerMapping_ = new HashMap<String,
StorageHandler>();

public StorageHandler getPrivateEncryptedStorageHandler(){
     return handlerMapping_.get("secure");
}
public StorageHandler getPublicSDCardStorageHa
      return handlerMapping_.get("public");
}


public void initHandlers(){
    handlerMapping_.put("secure", new PrivateStorageAdapter());
    handlerMapping_.put("public", new PublicStorageAdapter());
}

public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String key, String data){
    getPublicSDCardStorageHandler().store(key,data);
}
```

Our getPrivateEncryptedStorageHandler () method relies on proper configuration of the handlerMapping_

# 6. Don’t Allow Secure Pathways to be Compromised by Configuration Changes

```
private Map<String, StorageHandler> handlerMapping_ = new HashMap<String,
StorageHandler>();

public StorageHandler getPrivateEncryptedStorageHandler(){
     return handlerMapping_.get("secure");
}
public StorageHandler getPublicSDCardStorageHandler(){
      return handlerMapping_.get("public");
}


public void initHandlers(){
    handlerMapping_.put("secure", new PrivateStorageAdapter());
    handlerMapping_.put("public", new PublicStorageAdapter());
}

public void storeDataPrivatelyAndE
    getPrivateEncryptedStorageHand
}

public void storeDataPubliclyOnSDC
    getPublicSDCardStorageHandler(
}
```

If someone makes a mistake initializing this mapping or if it is connected to a configuration file, it could compromise our secure pathway

# 6. Don’t Allow Secure Pathways to be Compromised by Configuration Changes

```
private Map<String, StorageHandler> handlerMapping_ = new HashMap<String,
StorageHandler>();

public StorageHandler getPrivateEncryptedStorageHandler(){
     return handlerMapping_.get("secure");
}
public StorageHandler getPublicSDCardStorageHa
      return handlerMapping_.get("public");
}


public void initHandlers(){
    handlerMapping_.put("secure", new PrivateStorageAdapter());
    handlerMapping_.put("public", new PublicStorageAdapter());
}

public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String key, String data){
    getPublicSDCardStorageHandler().store(key,data);
}
```

The handlerMapping_ is also mutable and might be changed at runtime allowing a possible attack

# 6. Don’t Allow Secure Pathways to be Compromised by Configuration Changes

```
public StorageHandler getPrivateEncryptedStorageHandler(){
     return new PrivateStorageAdapter();
}
public StorageHandler getPublicSDCardStorageHandler(){
     return new PublicStorageAdapter();
}
```

This version makes security a compile-time decision and not a runtime or installation configuration decision

```
public void storeDataPrivatelyAndEncrypted(String key, String data){
    getPrivateEncryptedStorageHandler().store(key,data);
}

public void storeDataPubliclyOnSDCard(String key, String data){
    getPublicSDCardStorageHandler().store(key,data);
}
```