Insecure Data Storage in Android Applications

Introduction

Insecure data storage vulnerabilities occur when application store sensitive information such as username, password, and credit cards numbers in plain text.

Sensitive data is vulnerable when not properly protected by the app.,

Common Mistakes By Developers:

- Storing data in plain text without encryption.
- Insecurely storing keys, certificates, and passwords.
- Choosing weak encryption algorithms or custom encryption methods.

Android apps usually store data locally on these places:

- Shared preferences
- SQLite databases
- Internal storage
- External storage

Shared preferences

The shared preferences are stored in an XML file. They are stored in the app's data folder under the `shared_prefs` folder.

SQLite databases

SQLite databases are relational files that use a single file for storing data. The SQLite are suited for storing structured data on mobile. The database is only accessible to the app and not outside. To access this data a content provider is used.

Internal storage

Apps can also create folders under the their private folder in `/data/data/<package name>`. This folder is private to the app and other apps can not access it.

External storage

External storage is world readable and writable, any app with the `WRITE_EXTERNAL_STORAGE` and `READ_EXTERNAL_STORAGE` permissions.

Technical Impacts

Insecure data storage can result in data loss. at most, for an individual user. The important valuable data which are frequently stored:

- Usernames
- Authentication tokens
- Passwords
- Cookies
- Location Data
- Personal data: DoB, credit card etc.

Business Impacts

- Identity theft
- Reputation damage
- Violation of external policies
- Material loss

Practical Explationation

Part 1: Insecure Data Storage in DIVA

To see how the application stores data, we need to decompile the application have to look how application or specific activity saving data

```
public void saveCredentials(View view) {
    SharedPreferences spref = PreferenceManager.getDefaultSharedPreferences(this);
    SharedPreferences.Editor spedit = spref.edit();
    EditText usr = (EditText) findViewById(R.id.ids1Usr);
    EditText pwd = (EditText) findViewById(R.id.ids1Pwd);
    spedit.putString("user", usr.getText().toString());
    spedit.putString("password", pwd.getText().toString());
    spedit.commit();
    Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
```

FBy analyzing the vulnerable code in JADX or another decompilation tool. In the code, you will notice that the application is using an preferenceManager to store plaintext data.PreferenceManager saves data in XML in application path.

Exploiting:

Entering Random Data in the Application

Begin by entering random data into the application to simulate the storage of credentials.

Connecting Android Using ADB

Establish a connection with your Android device using the following command:

`adb connect [device_ip]`

Access your Android device's shell by running:

`adb shell`

Navigating to the Data Directory

Navigate to the application's data directory with the following commands:

`cd data/data/jakhar.aseem.diva`

Accessing the Shared Preferences Directory

Inside the application's data directory, you will find a folder named "shared_prefs." Navigate to this directory:

`cd shared_prefs`

Locating the XML File with Plain Text Credentials

Inside the "shared_prefs" directory, you will find an XML file. This file contains all the data and credentials saved by the app in plain text.

By following these steps, you can observe how the application stores data and credentials in an insecure manner, potentially exposing sensitive information.

Part 2: Vulnerable Code Using SQLite Database

To see how the application stores data, we need to decompile the application have to look how application or specific activity saving data

```
/* JADX INFO: Access modifiers changed from: protected */
@Override // android.support.v7.app.AppCompatActivity, android.support.v4.app.FragmentActivity, android.support.v4.app.BaseFragmentA
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
        this.mDB = openOrCreateDatabase("ids2", 0, null);
        this.mDB.execSQL("CREATE TABLE IF NOT EXISTS myuser(user VARCHAR, password VARCHAR);");
    } catch (Exception e) {
        Log.d("Diva", "Error occurred while creating database: " + e.getMessage());
    }
    setContentView(R.layout.activity_insecure_data_storage2);
}

public void saveCredentials(View view) {
    EditText usr = (EditText) findViewById(R.id.ids2Usr);
    EditText pwd = (EditText) findViewById(R.id.ids2Pwd);
    try {
        this.mDB.execSQL("INSERT INTO myuser VALUES ('" + usr.getText().toString() + "', '" + pwd.getText().toString() + "');");
        this.mDB.close();
    } catch (Exception e) {
        Log.d("Diva", "Error occurred while inserting into database: " + e.getMessage());
    }
    Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
}
```

By analyzing the vulnerable code in JADX or another decompilation tool. In the code, you will notice that the application is using an SQLite database to store data.

To identify which database is being used, inspect the code, typically found around line 22, where the database name is specified (e.g., "ids2").

Exploiting:

Entering Random Data in the Application

Begin by entering random data into the application to simulate the storage of credentials.

Nevigate to directory:

Locate the directory where SQLite databases are stored. This can be done by running the following commands in the Android shell:

`adb shell cd /data/data/jakhar.aseem.diva/databases`

In the "databases" directory, you should see the database files. These files may have extensions or not.

Pull database file to local machine

Use `adb pull` to copy the database file to your local machine for further examination:

`adb pull /data/data/jakhar.aseem.diva/databases/ids2`

Using sqlite3 tool for opening database file:

Upon pulling the database file, you can open it in a terminal or SQLite client. Use the following command to open it:

`sqlite3 ids2`

Within the SQLite shell, you can issue commands to inspect the database. For instance, you can use `.help` to see available commands and `.tables` to list the tables within the database.

Identify the table of interest; for example, if the code refers to a "myuser" table on line 23, you can select its contents:

`sqlite> select * from myuser;`

Examine the data in the table. If you find that the data is stored in plain text, this indicates a significant security vulnerability.

Part 3: Insecure Data Storage Using Temporary Files

To see how the application stores data, we need to decompile the application have to look how application or specific activity saving data

```
public void saveCredentials(View view) {
    EditText usr = (EditText) findViewById(R.id.ids3Usr);
    EditText pwd = (EditText) findViewById(R.id.ids3Pwd);
    File ddir = new File(getApplicationInfo().dataDir);
    try {
        File uinfo = File.createTempFile("uinfo", "tmp", ddir);
        uinfo.setReadable(true);
        uinfo.setWritable(true);
        FileWriter fw = new FileWriter(uinfo);
        fw.write(usr.getText().toString() + ":" + pwd.getText().toString() + "\n");
        fw.close();
        Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
    } catch (Exception e) {
        Toast.makeText(this, "File error occurred", 0).show();
        Log.d("Diva", "File error: " + e.getMessage());
    }
```

1. Now, let's focus on examining the code that uses temporary files for storing credentials. Look for this code section in the decompiled source.
2. Typically, you'll find a method or function where the temporary file is created and data is saved. This method can be identified, often around line 30, as the one responsible for saving the credentials to a temporary file.

Exploiting:

Navigate to directory:

To conclude this practical part, let's return to the Android shell and navigate to the application's data directory where the temporary file is stored. This can be done with the following commands:

`cd /data/data/jakhar.aseem.diva`

Look for file and reed file:

Inside this directory, look for the new file named "uinfo" or "uinfotemp." The name may vary, but it should be related to the temporary storage of data.

To examine the contents of this temporary file, use the `cat` command. For example, if the file is named "uinfo4879648433137917139tmp," you can view it with:

`cat uinfo4879648433137917139tmp`

Upon viewing the file, you will likely find that all data stored in this temporary file is in plain text. This lack of encryption signifies a security vulnerability, as sensitive data can be easily accessed.

Part 4: Data Storage in External Storage

To see how the application stores data, we need to decompile the application have to look how application or specific activity saving data

```
public void saveCredentials(View view) {
    EditText usr = (EditText) findViewById(R.id.ids4Usr);
    EditText pwd = (EditText) findViewById(R.id.ids4Pwd);
    File sdir = Environment.getExternalStorageDirectory();
    try {
        File uinfo = new File(sdir.getAbsolutePath() + "/.uinfo.txt");
        uinfo.setReadable(true);
        uinfo.setWritable(true);
        FileWriter fw = new FileWriter(uinfo);
        fw.write(usr.getText().toString() + ":" + pwd.getText().toString() + "\n");
        fw.close();
        Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
    } catch (Exception e) {
        Toast.makeText(this, "File error occurred", 0).show();
        Log.d("Diva", "File error: " + e.getMessage());
    }
```

Review the code that stores data in external storage (e.g., SD card).

Look for the "unfo.txt" file in the external storage directory.

Exploiting:

Navigate to directory:

To practically demonstrate this scenario, return to the Android shell and navigate to the external storage directory. You can use the following commands:

```
cd /mnt/sdcard
```

View the file:

Lfile is not listing we can see in code file is saving by .unfo.txt so its hidden file so we have to add -a to view hidden file ls-a

```
ls -a
```