Seneca | SCHOOL OF INFORMATION TECHNOLOGY ADMINISTRATION & SECURITY

FINAL REPORT

APPLICATION OF OWASP TOP 10 FOR MOBILE APP

Prepared To :

JUAN CASTANEDA

Presented By:

PAVITHRAN COIMBATORE JAYARAJ

PRABHAKARAN RAMALINGAM

ABSTRACT

Mobile applications, which offer usefulness and convenience in a variety of fields, have become essential to modern living. But because of this pervasive dependence, there are now serious security issues when hackers take advantage of flaws to obtain illegal access, compromise private data, and interfere with services. To overcome these obstacles and improve our security capabilities, we held an internal red team training session at SecureWave Technologies, a company specialising in the development of safe mobile apps.

As part of this effort, we used the OWASP Mobile Top 10 methodology to analyse BugBazaar, an intentionally vulnerable application, in order to find and take advantage of vulnerabilities. This experiment gave participants useful insights into important security topics. Our method examined the application's code and simulated real-world attack scenarios by combining static and dynamic analysis. The results emphasised the necessity of thorough developer training, frequent security audits, secure coding procedures, and strong encryption. We hope to improve our overall security posture and reaffirm our dedication to creating safe mobile applications by fixing these vulnerabilities.

Table of Contents

Introduction *4*

Proposed Solution and Results *4*

Overview of the Solution **5**

Technologies and Tools Used **5**

Results and Outcomes **6**

Dynamic Analysis **6**

M1 Improper Credential Usage 6

M3- Insecure Authentication/Authorization 9

M4- Insufficient Input/Output Validation 13

M8- Security Misconfiguration 19

Static Analysis **23**

M6: Inadequate Privacy Controls 24

M10- Insufficient Cryptography 26

Other Vulnerabilities **28**

M2: Insufficient Supply Chain Security 28

M5- Insecure Communication 29

M7- Insufficient binary protection 31

M9- Insecure Data Storage 32

Conclusion *33*

Recommendations **33**

Reference *34*

Introduction

Mobile apps have become essential to contemporary living, offering convenience and functionality in multiple sectors. Nonetheless, the extensive dependence on mobile applications has introduced major security issues, as attackers take advantage of weaknesses to obtain unauthorized access, compromise sensitive information, and interrupt services. At SecureWave Technologies, a startup focused on secure mobile app development, we carried out an internal red team training session to tackle these issues and enhance our security skills.

In this initiative, we examined BugBazaar, a purposefully vulnerable application, to detect and exploit weaknesses in accordance with the OWASP Mobile Top 10 framework. The exercise offered practical perspectives on essential security concerns like unsafe data storage, weak cryptographic methods, and poor supply chain security. These weaknesses usually occur because of insufficient compliance with secure coding standards, inadequate validation processes, and mishandling of external libraries.

The initiative sought to identify these vulnerabilities, investigate their underlying causes, and create effective solutions. This practical approach not only deepened our knowledge of mobile app security but also strengthened SecureWave Technologies' dedication to creating secure applications from the ground up.

Proposed Solution and Results

To tackle the security issues found in the analysis of the BugBazaar application, we created a comprehensive strategy emphasizing both preventive and remedial actions. This included utilizing contemporary tools and techniques to detect, utilize, and address vulnerabilities efficiently, following the OWASP Mobile Top 10 framework.

We split our approach into two components: dynamic analysis to mimic real-world attack situations and static analysis to examine the application's code. The upcoming subsections describe the overall architecture of the solution, the technologies used, and the results of the analysis.

Overview of the Solution

1. **Dynamic Analysis**: Using tools like Frida, Jadx, Apktool, Android Studio and ADB, we conducted runtime manipulations, explored authentication mechanisms, and examined insecure configurations.

2. **Static Analysis**: Leveraging MobSF, we analyzed the app's code structure, cryptographic practices, and storage mechanisms to uncover vulnerabilities like inadequate privacy controls and weak cryptography.

3. **Vulnerability Testing Framework**: All findings were evaluated against the OWASP Mobile Top 10 risks to ensure comprehensive coverage.

Technologies and Tools Used

- **MobSF**: Conducted in-depth static analysis, including code integrity checks and cryptographic evaluations.

- **Frida**: Allowed runtime manipulation to test authentication and data security mechanisms.

- **Jadx**: Helped decompile and analyze application source code for potential weaknesses. We especially used this to view the java script of the source code.

- **ADB Commands**: Explored app behavior under different scenarios, simulating potential exploits.

- **Apktool**: Enabled reverse engineering of APK files to inspect and modify the app's resources and manifest file for misconfigurations and vulnerabilities. We used it to decompile our APK and view it on Android Studio.

- **Android Studio**: Provided an integrated development environment to debug and simulate the app's behavior in a controlled environment, enabling better analysis of its components. We used our emulator here and viewed our decompiled APK.

Results and Outcomes

Dynamic Analysis

M1 Improper Credential Usage

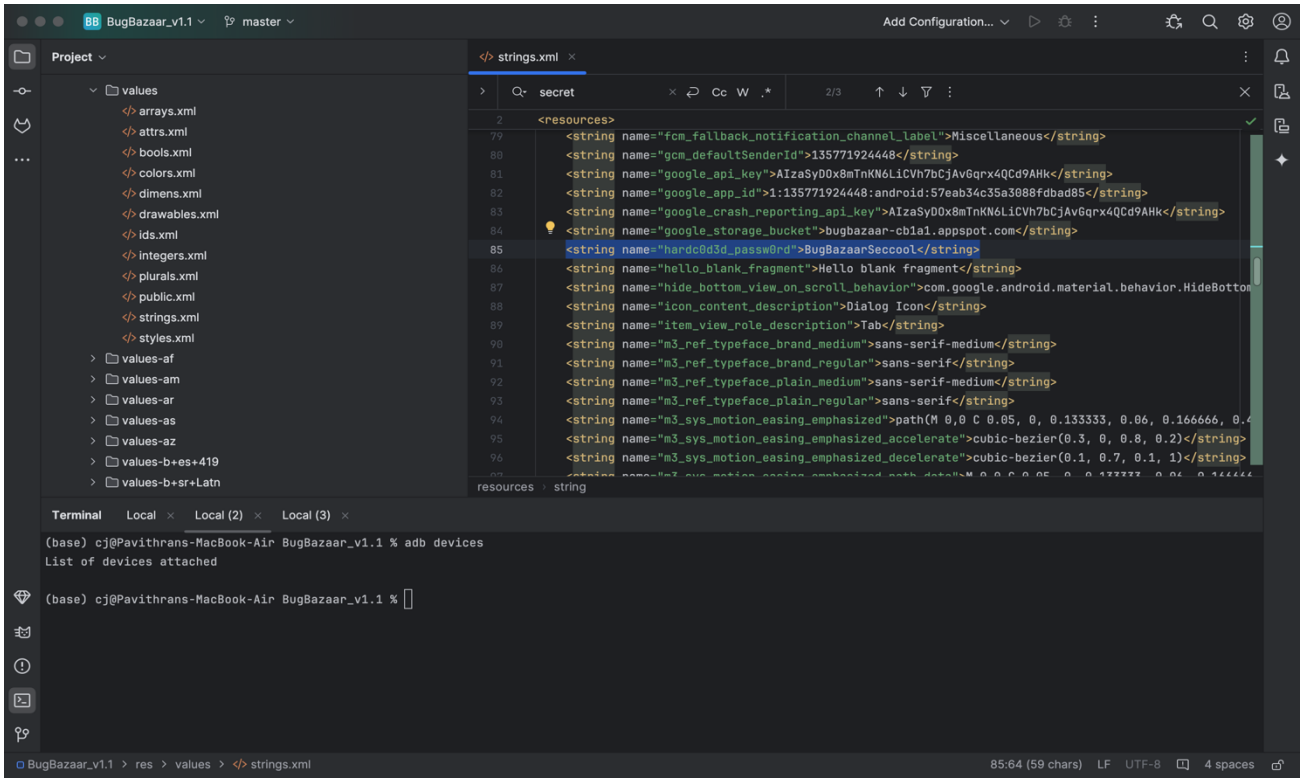
Inappropriate use of credentials is a common weakness in mobile applications. Threat actors can take advantage of this vulnerability by finding hardcoded secrets like passwords, API keys, or tokens that are embedded in the application's source code. This permits unauthorized entry to sensitive information, administrative capabilities, or other vital assets. Inadequate credential management presents serious risks, such as data breaches, privacy infringements, and possible fraud.

Approach and results

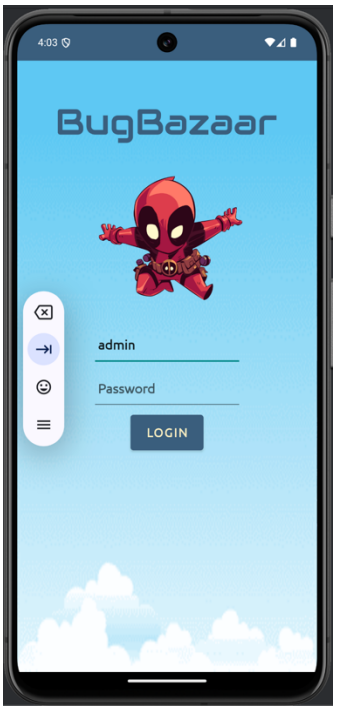
1. Decompiled the APK using **Apktool** to inspect the app's internal structure.

2. Reviewed the source code and XML files using **Jadx** and **Android Studio** to identify hardcoded secrets.

3. Located a **string.xml** file in the values folder containing a hardcoded password.



4. Found Hardcoded secrets like password.
5. Successfully logged into the application using the hardcoded credentials.
6. Asks us to create a passcode when we login.



Mitigation

- Refrain from embedding sensitive information such as passwords, API keys, or tokens directly into application code or configuration files. Rather, utilize secure environment variables or fetch them dynamically from the server.

- Secure credentials in transit by employing strong encryption protocols like TLS 1.2 or above.

- Implement strong user authentication mechanisms, such as OAuth2, and use multi-factor authentication (MFA) for added security.

- Establish robust user authentication methods, like OAuth2, and incorporate multi-factor authentication (MFA) for enhanced security.

- Consistently refresh and change API keys or tokens to reduce their vulnerability. Temporary tokens and the rapid revocation of compromised credentials further strengthen security.

- Safeguard sensitive user information securely. Refrain from saving credentials on the device, choose secure access tokens as an alternative. Utilize secure storage options such as Keystore (Android) or Keychain (iOS) for any sensitive information that is necessary.

M3- Insecure Authentication/Authorization

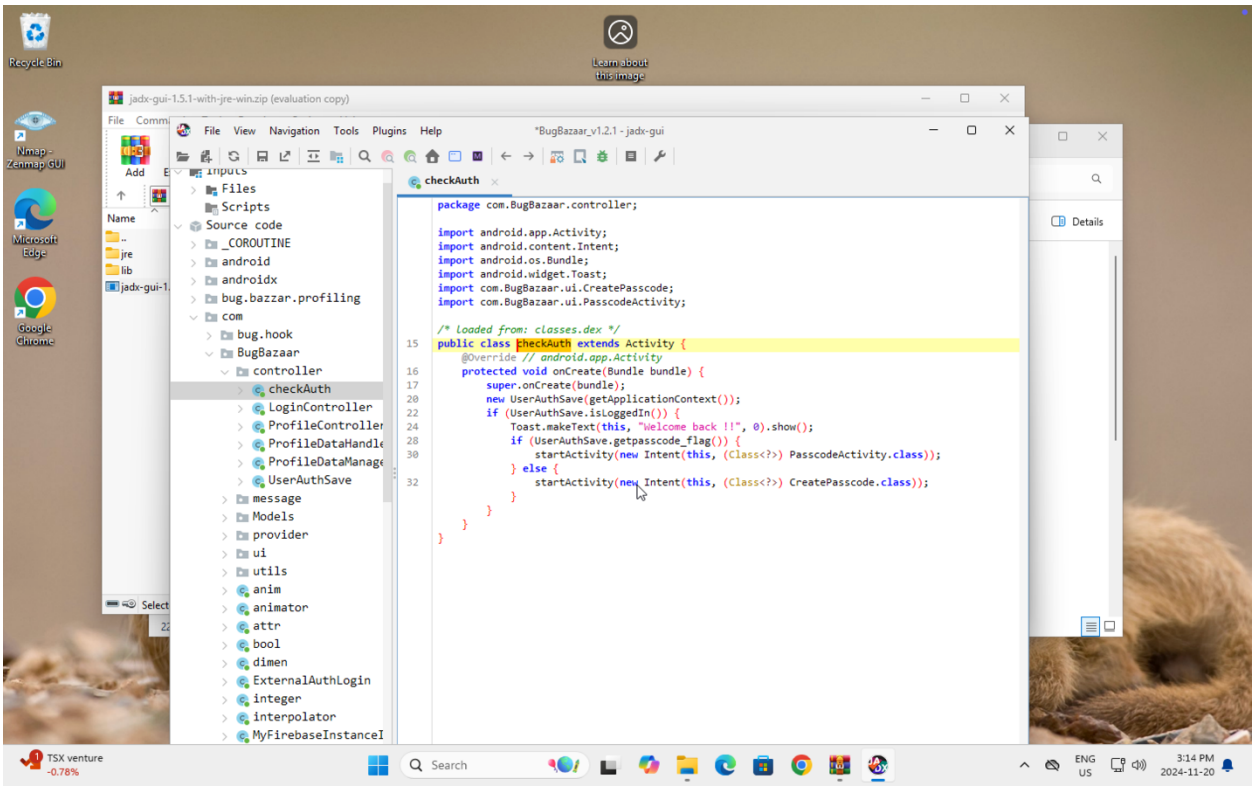
In mobile apps, weak authentication and authorization methods can result in account hijacking and illegal access to confidential information. Such problems usually occur when authentication protocols are inadequately executed, making the system susceptible to evasion attempts. Malicious individuals or automated attack tools can take advantage of these weaknesses to access features that are meant to be limited.

Impact: The abuse of authentication/authorization weaknesses can enable attackers to perform unauthorized actions, retrieve sensitive information, or elevate privileges. This may lead to serious outcomes, including data breaches, unsanctioned transactions, or harmful alterations to the system.

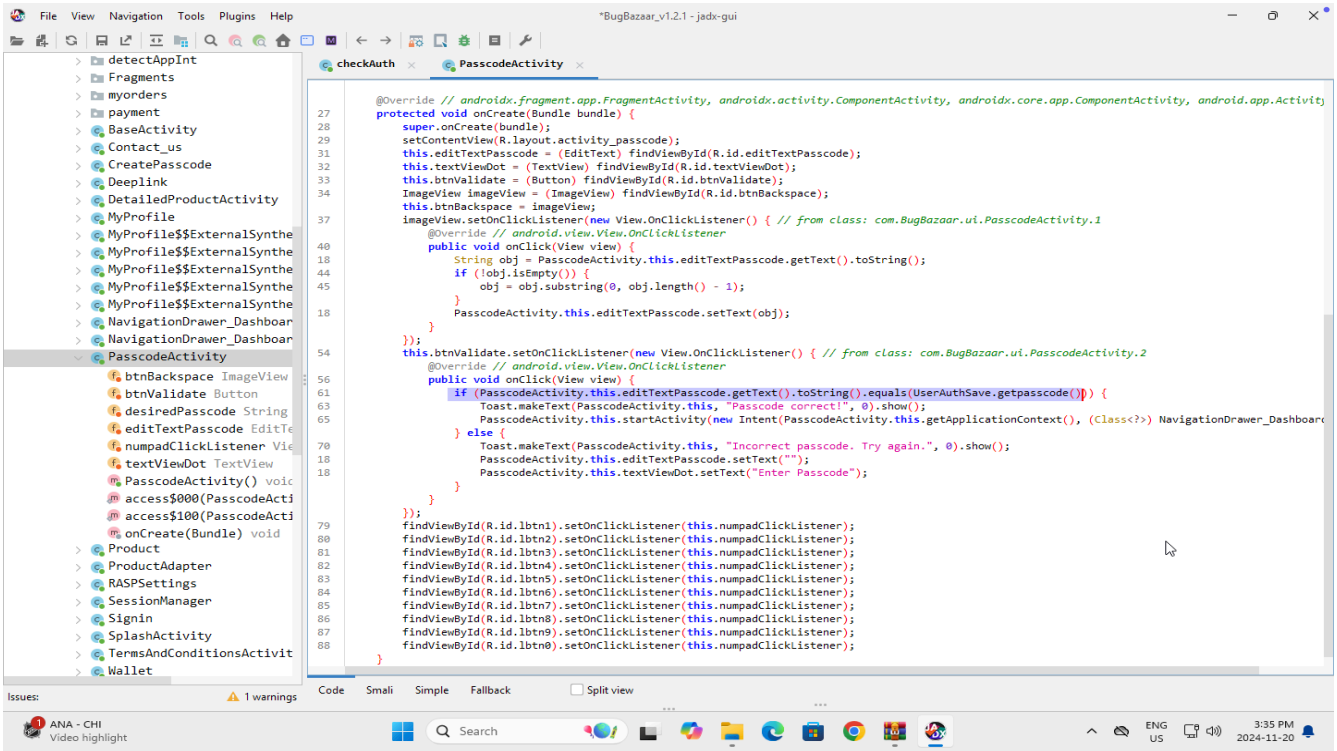
Approach and results

1. We first view the application source code in JADX.

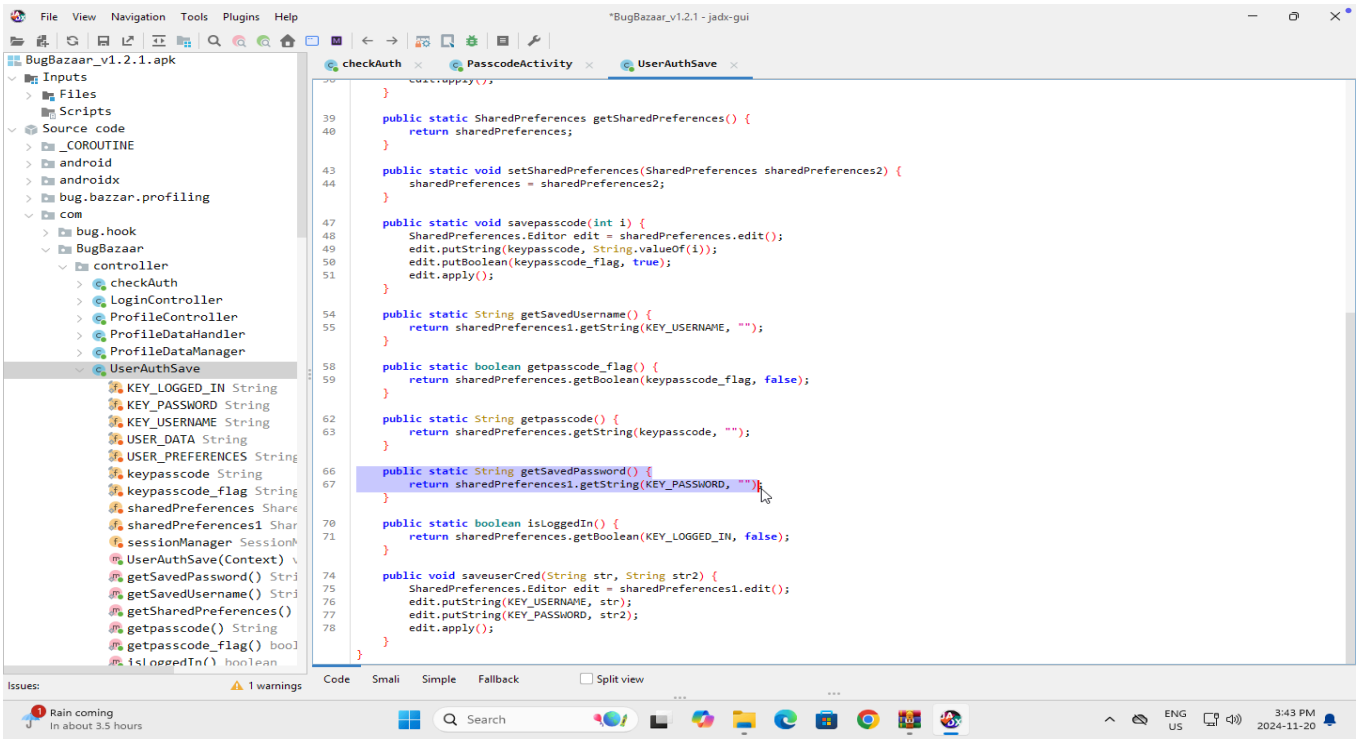
2. In the CheckAuth class it checks the pass code. If there's no passcode set it prompts to create a new passcode. Since we already created a passcode which is 1414, we don't have to create it.



3. Here in the PasscodeActivity class it compares the passcode on the screen with the UserAuthSave class.

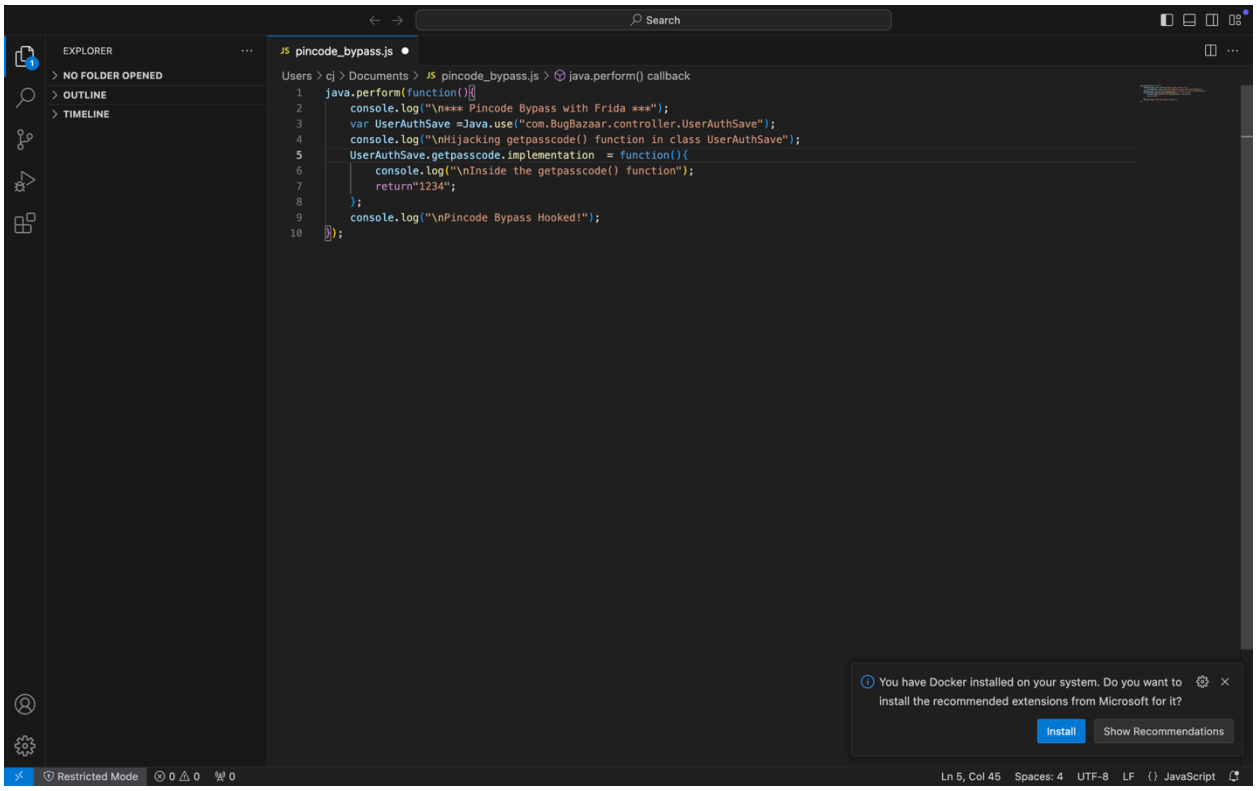


4. Now in the UserAuthSave class, the getpasscode method gets the passcode which is saved in the shared preference. So, the passcode is compared, and we are logged in.



5. Since there's a shared preference here, this is our vulnerability and here we can try to hook into this method using runtime manipulation using Frida to return whatever passcode we want. So basically, we are storing whatever passcode we want in the shared preference.

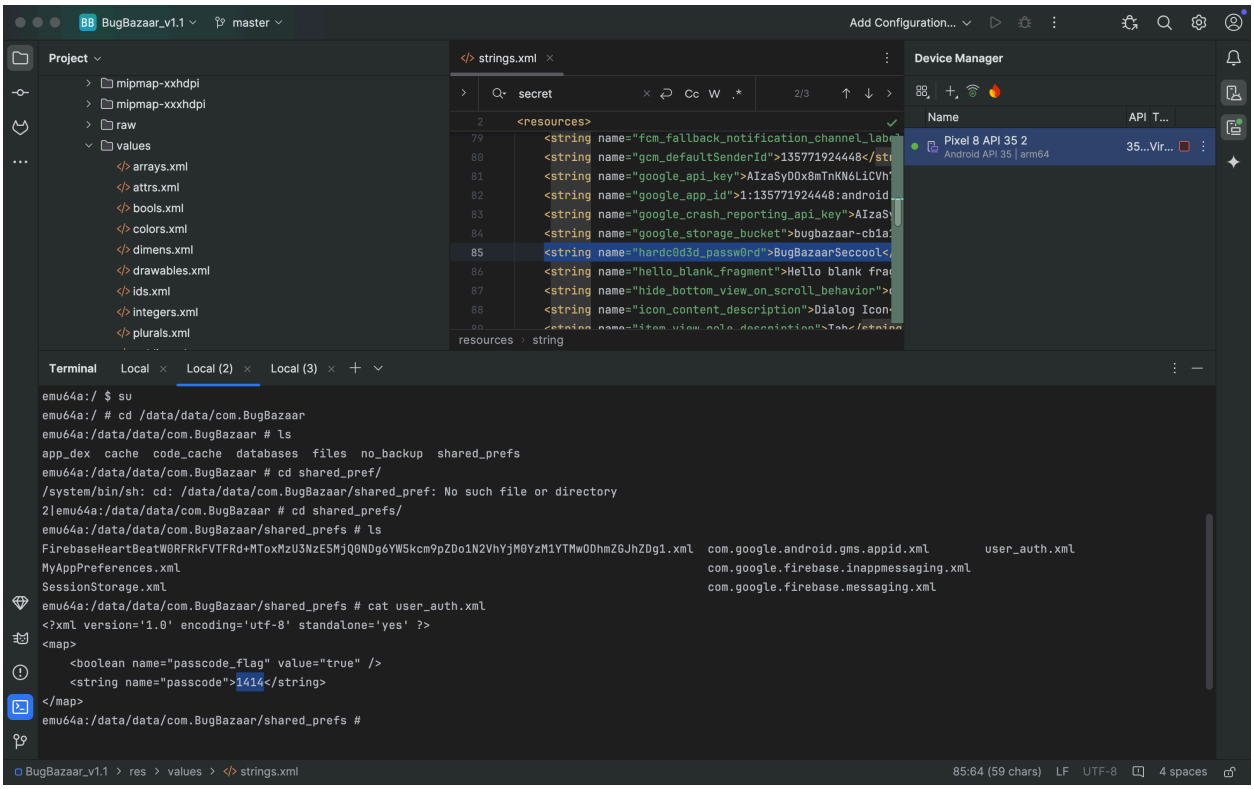
6. Then we write our java script to hook into the getpasscode method to return passcode 1234.



7. The exploitation using Freda could not be completed due to compatibility issues on my Macbook Air because I was using its terminal to execute Frida. We show this as a vulnerability because the authentication is done locally.

8. We can also view the shared preference file by viewing the data file stored in the directory.

9. First, we root our device using Magisk and then view the shared preference folder in the BugBazaar directory and look at the user_auth.xml and we can see the stored password.



Mitigation

- **Critical fix:** If the authentication procedure is managed on the server side, this vulnerability may be prevented. Sending the pincode or additional authentication information to the server for verification can help prevent local tampering.

- If local authentication is necessary, developers need to incorporate code integrity checks to identify unauthorized modifications. These verifications guarantee that the app's code remains untampered, assisting in reducing runtime manipulation attacks.

- Authentication must be implemented on the server side whenever feasible. The device must store data only following successful authentication.

- Make sure that mobile apps meet the same authentication standards as their web equivalents, indicating that multi-factor authentication should be considered.

- Any personally sensitive information kept locally, like passwords or authentication tokens, must be encrypted with robust encryption techniques and securely generated keys based on the user's login details.

- Create authentication tokens specific to devices that users can invalidate if their device is stolen or lost. This aids in reducing unauthorized access from lost or stolen devices.

- Refrain from utilizing device identifiers or geo-location as authentication methods, since attackers can easily spoof them.

- Whenever feasible, utilize FaceID or TouchID for enhanced security during authentication. These biometric factors can be employed to access securely held secrets such as session tokens or credentials.

M4- Insufficient Input/Output Validation

Inadequate input/output validation may result in serious security risks, including SQL injection, command injection, and cross-site scripting (XSS) attacks. If input data is not adequately validated and sanitized, malicious individuals can exploit it, leading to the exposure of sensitive information or the possibility to run arbitrary commands on the system. Moreover, insufficient output validation may result in data corruption, incorrect handling, or insecure data display, enabling attackers to influence how data is shown or managed by the application.

These vulnerabilities can lead to:

- Unauthorized access to sensitive data.

- Manipulation of app functionality.

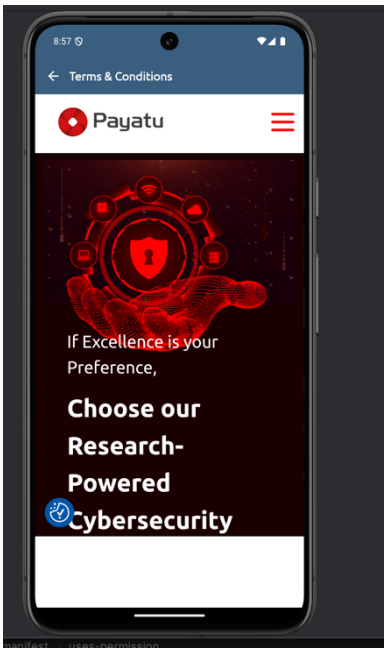
- Full compromise of the mobile system.

Approach and Result

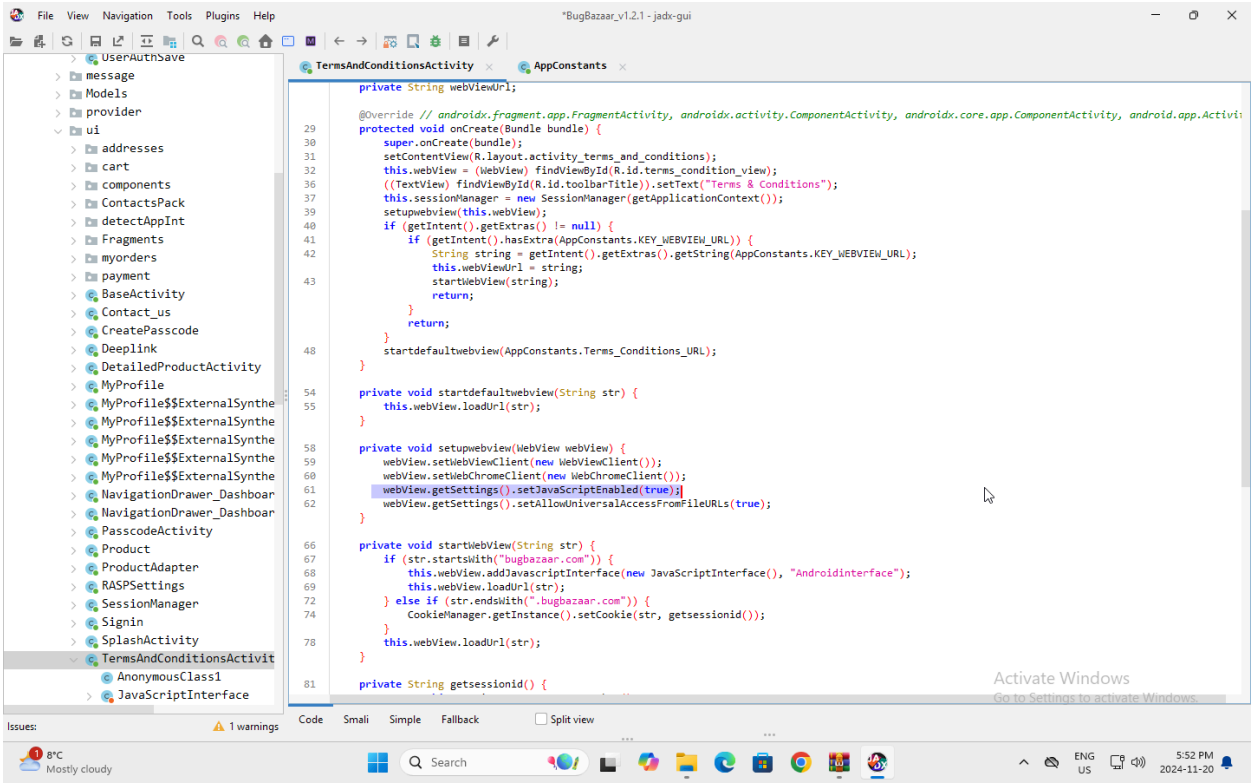
1. Crossite scripting can be done if WebView is enabled.

2. WebView is an extension of android application view class, that lets you display webpage in your activity. It’s a browser in the application.

3. This is how the Terms and condition page in the app looks like which displays the WebView Ui of whatever link was given in the source code.

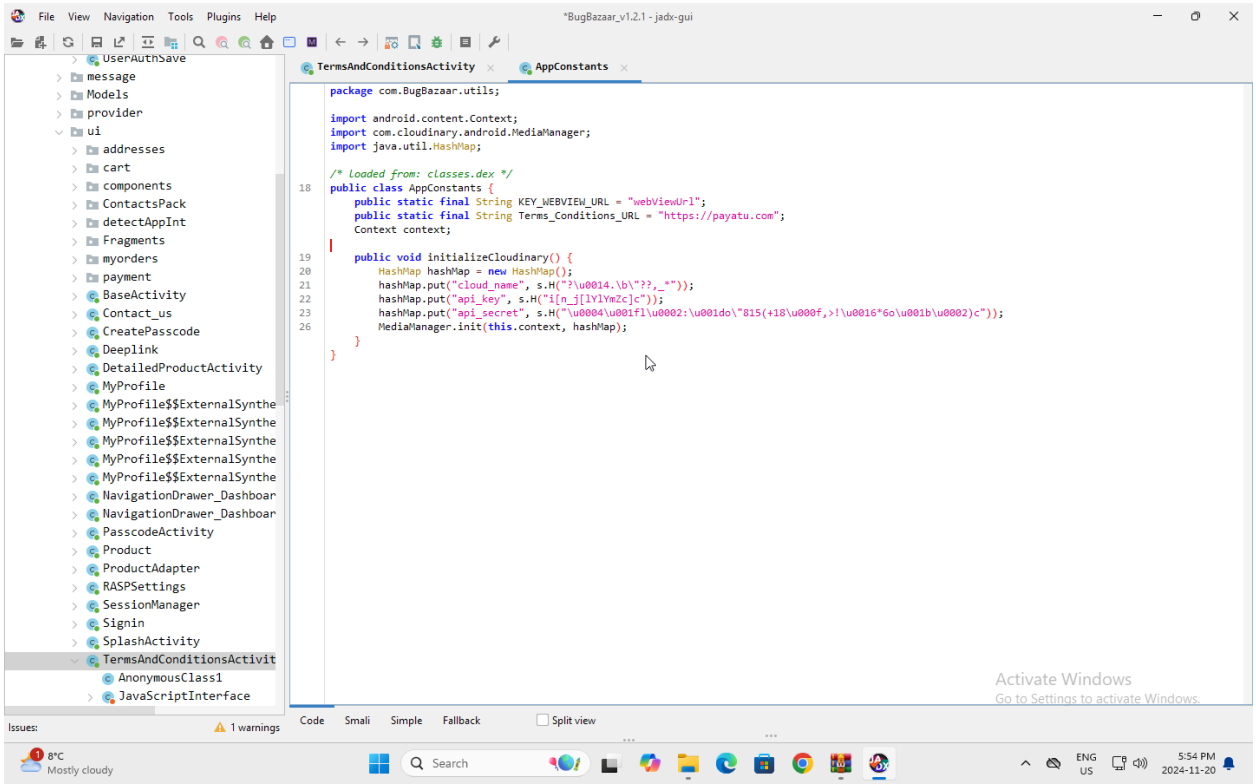


4. In the terms and condition class we can see that WebView is available with JavaScript enabled.



5. The WebView that is to be loaded is hardcoded in the application called webview_url. It is a string that can be passes to the activity through the intent.

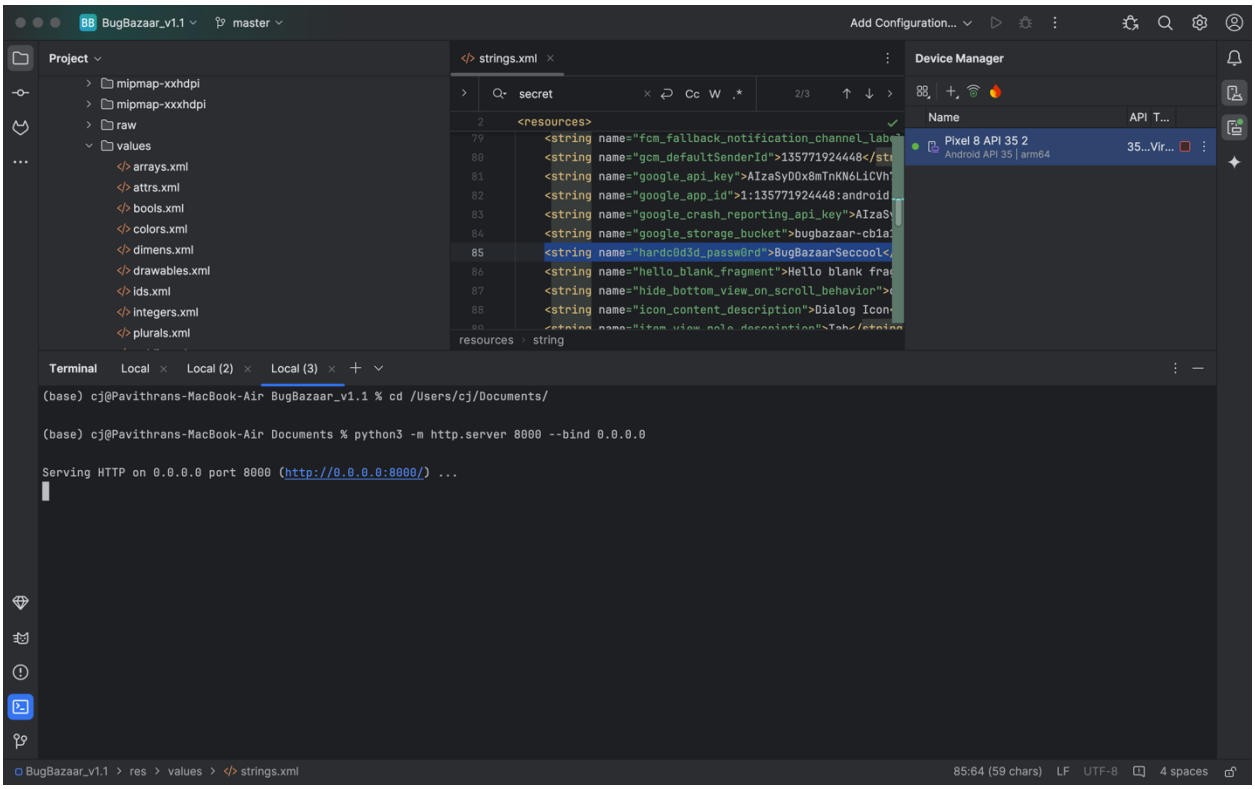
6. Here the WebView URL will be fetched and loaded in the WebView.



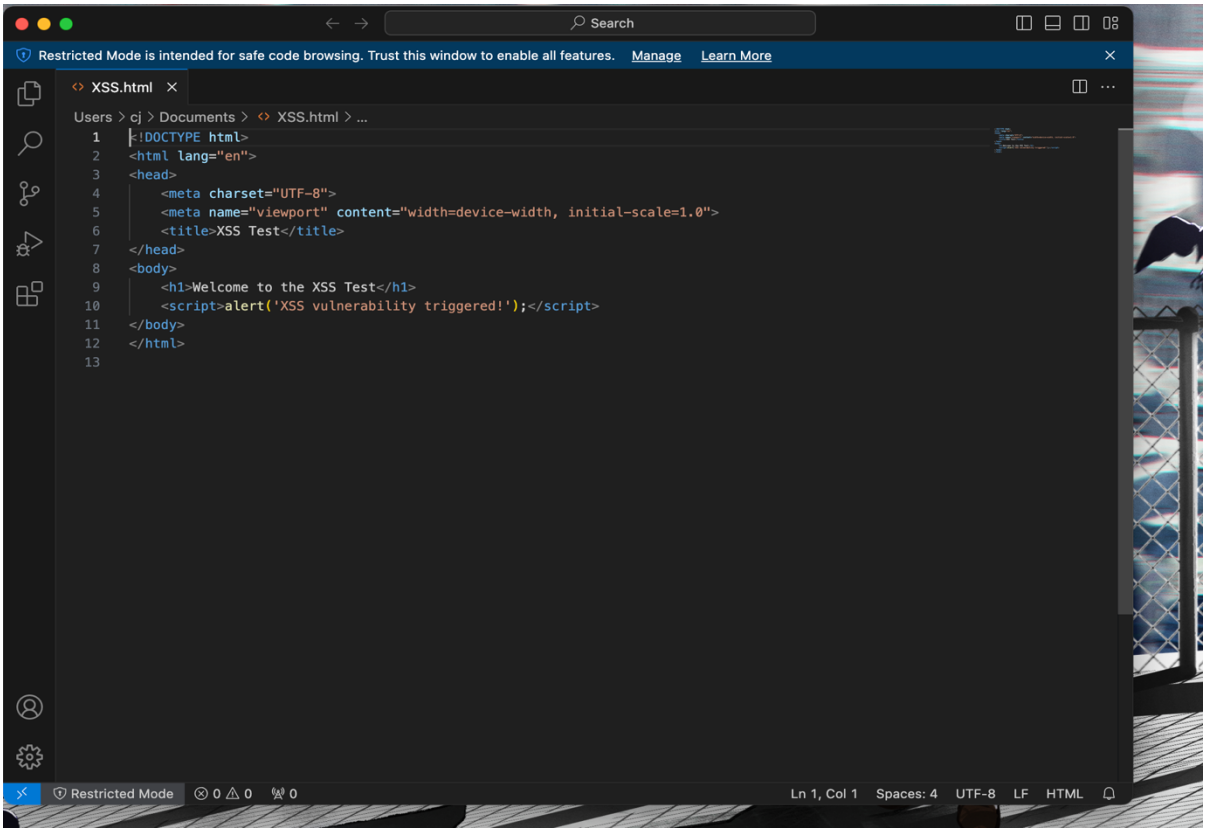
7. So, we can create a java script and load it in the URL of the activity and the java script will be loaded in the application.

8. To do that we need to host our server locally

9. To do that we can use ngrock or python. Here I've used python to host a local server.



10. We write our XXS file and save it as a HTML file.



11. Host the xss.html file on our server locally.

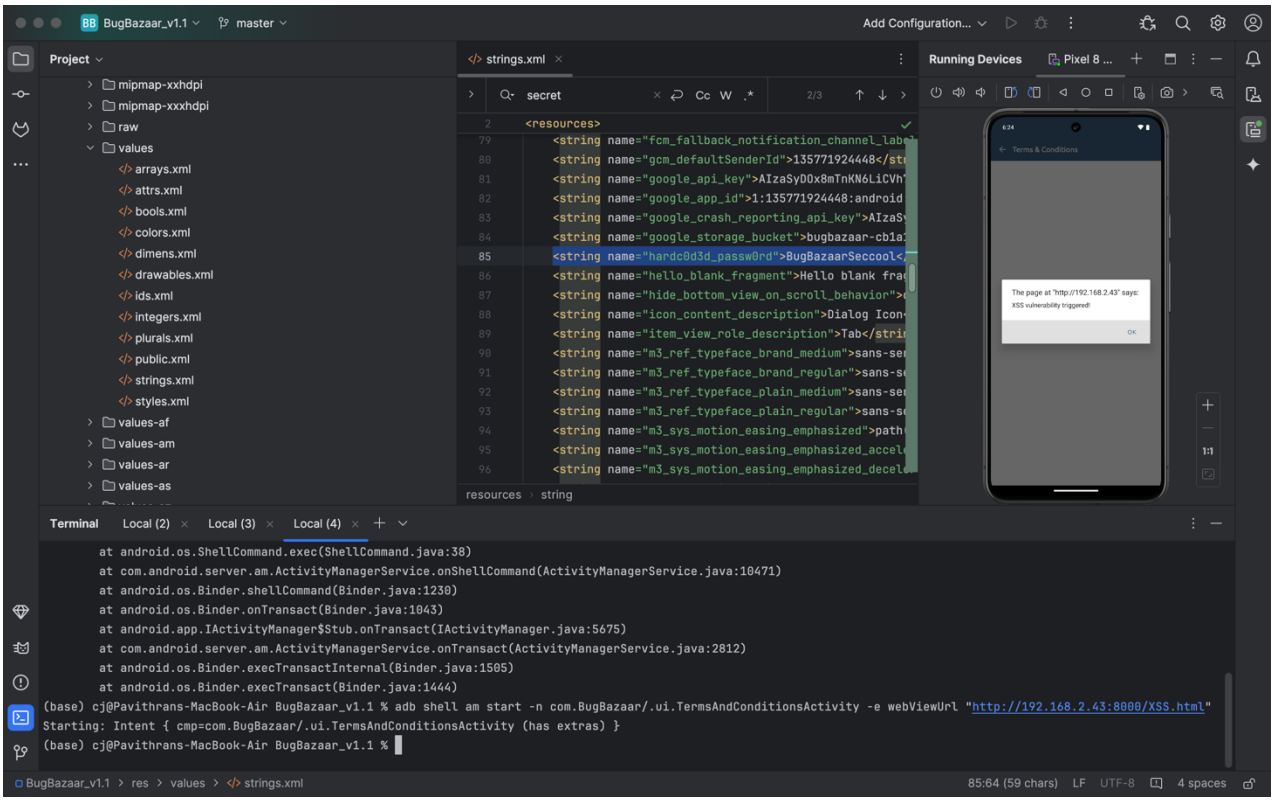
12. We pass our html file using the adb command **(adb shell am start -n com.BugBazaar/.ui.TermsAndConditionsActivity -e WebViewUrl http://192.168.2.43:8000/XSS.html)** The IP address is our IP address of our system that we are hosting the server.

13. This command uses ADB am(Activity Manager) to start the TermsAndConditionsActivity in the com.BugBazaar app and passes a URL (http://192.168.2.43:8000/XSS.html) as an extra parameter named webViewUrl.

14. If TermsAndConditionsActivity is set up to read and load this webViewUrl parameter, it should display the content from that URL in a WebView or handle it in some other way. This can be useful for testing purposes, such as checking how the app handles external URLs or testing for potential vulnerabilities like XSS.

15. Our html file will be loaded in the terms and condition page where the WebView is located.

16. So, when we load the terms and condition age we can see the output that was there in the html file.



17. Ideally for this vulnerability this exploit will work in a scenario where we already have another application installed on the device which will send the intent to the vulnerable app which will lead to this alert box or whatever is there in the html file we loaded.

Mitigation

- Turn off JavaScript in WebView to stop harmful scripts from running that may take advantage of weak validation flaws.

- Restrict the content shown in WebView to only reliable sources. This stops harmful external content from being loaded into the WebView and taking advantage of possible vulnerabilities.

- Enforce rigorous input validation and sanitization processes to guarantee that all data entered by the user is secure. This entails verifying the length, type, and content of user submissions, discarding any potentially harmful or dubious information.

- Ensure adequate validation of intents (e.g., starting activities, transferring data between components) to stop attackers from taking advantage of these communication pathways. This involves confirming that the information transmitted via intents cannot be altered by harmful individuals.

- Clean and verify all user inputs following stringent guidelines. Enforce limits on length and discard any unexpected or questionable data, stopping harmful input.

- Ensure that output data is correctly sanitized to avoid vulnerabilities such as cross-site scripting (XSS) when displaying user inputs or external data on the interface. Utilize appropriate output encoding methods to prevent the insertion of harmful scripts or content into the webpage.

- Verify data based on its context (such as file uploads, database queries) to avert attacks such as path traversal, SQL injection, or other types of data manipulation. Create tailored validation for various data types and actions to guarantee that only valid and anticipated data is handled.

- Conduct routine data integrity assessments to verify that confidential information remains unaltered and has not been interfered with by unauthorized entities.

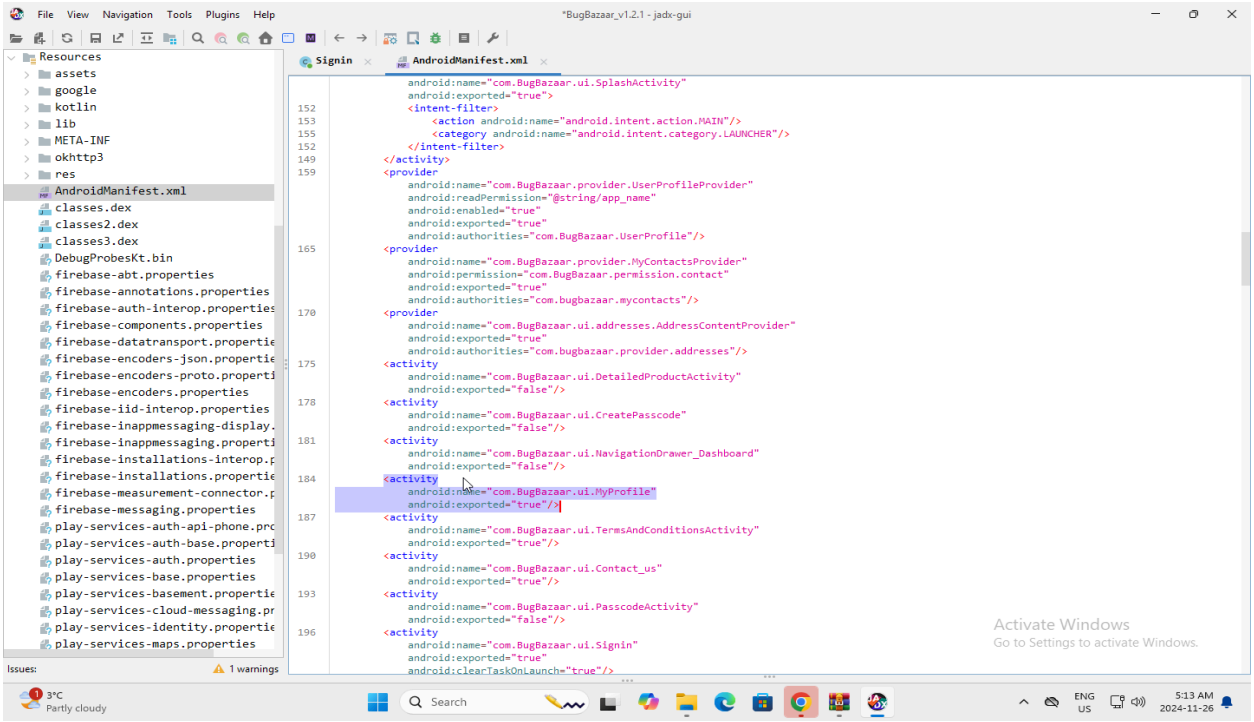
M8- Security Misconfiguration

Security Misconfiguration happens when an application, along with its components or environment, is not configured securely, making it susceptible to exploitation. This may occur because of incorrectly configured permissions, exposed services, weak security measures, or neglected security configurations. Typical instances involve the exportation of functions in Android lacking adequate security or permitting access to sensitive information because of misconfiguration. Attackers may take advantage of these misconfigurations to gain unauthorized access to confidential information or execute harmful activities, resulting in data breaches, system compromise, or unauthorized control over the application or device. Correct configuration, restricting unnecessary access, and safeguarding permissions are crucial to averting this vulnerability.

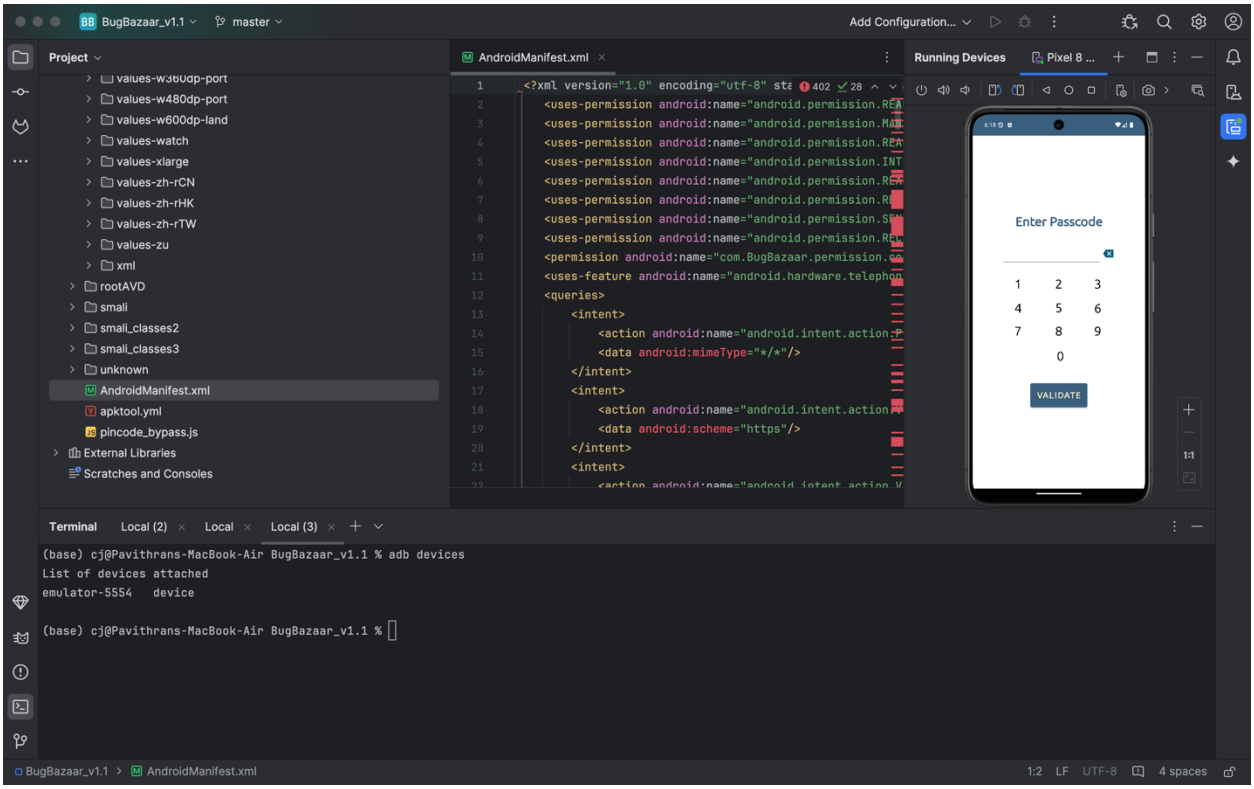
Approach and Results

1. For this risk we look at the manifest.xml file to look at the exported Activities.

2. For security misconfiguration, we are looking at profile activity. This contains PII of the user, some personal info of user. This is an example of this vulnerability.



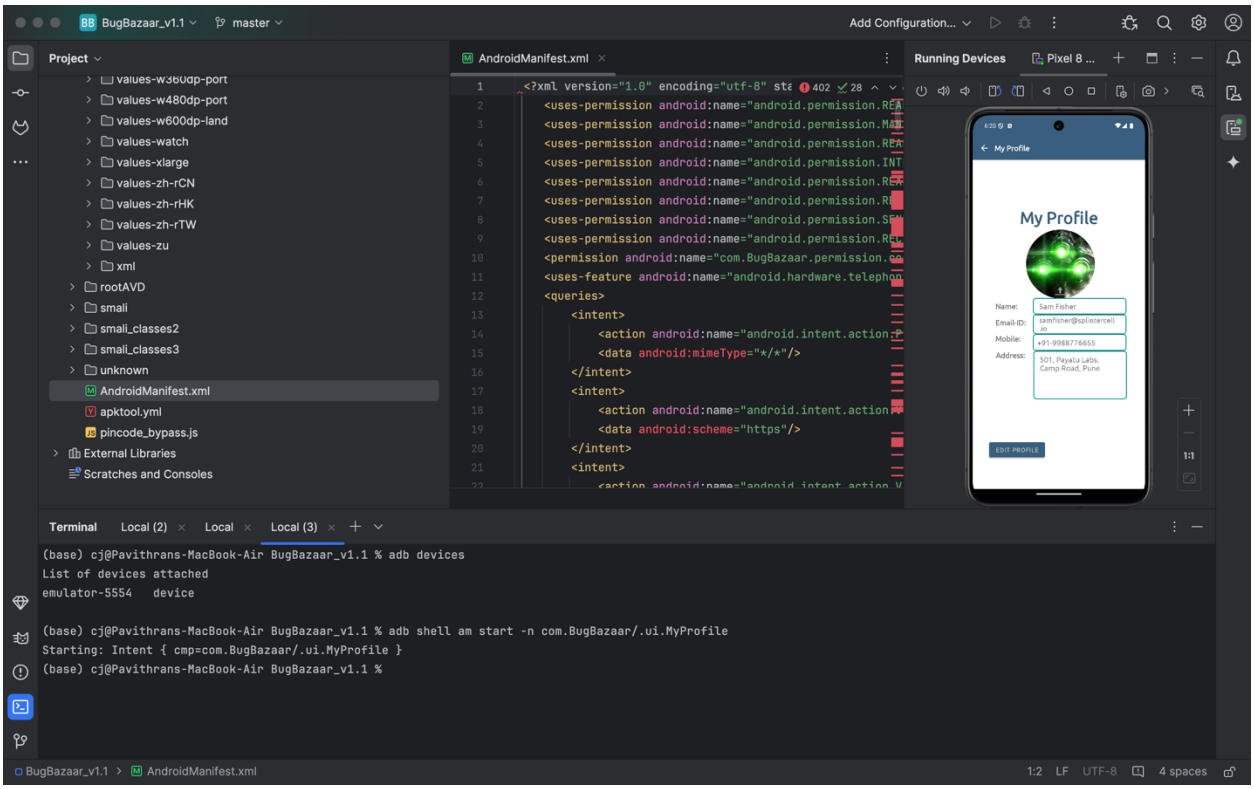
3. Now we are in the pin code screen of the BugBazaar application.



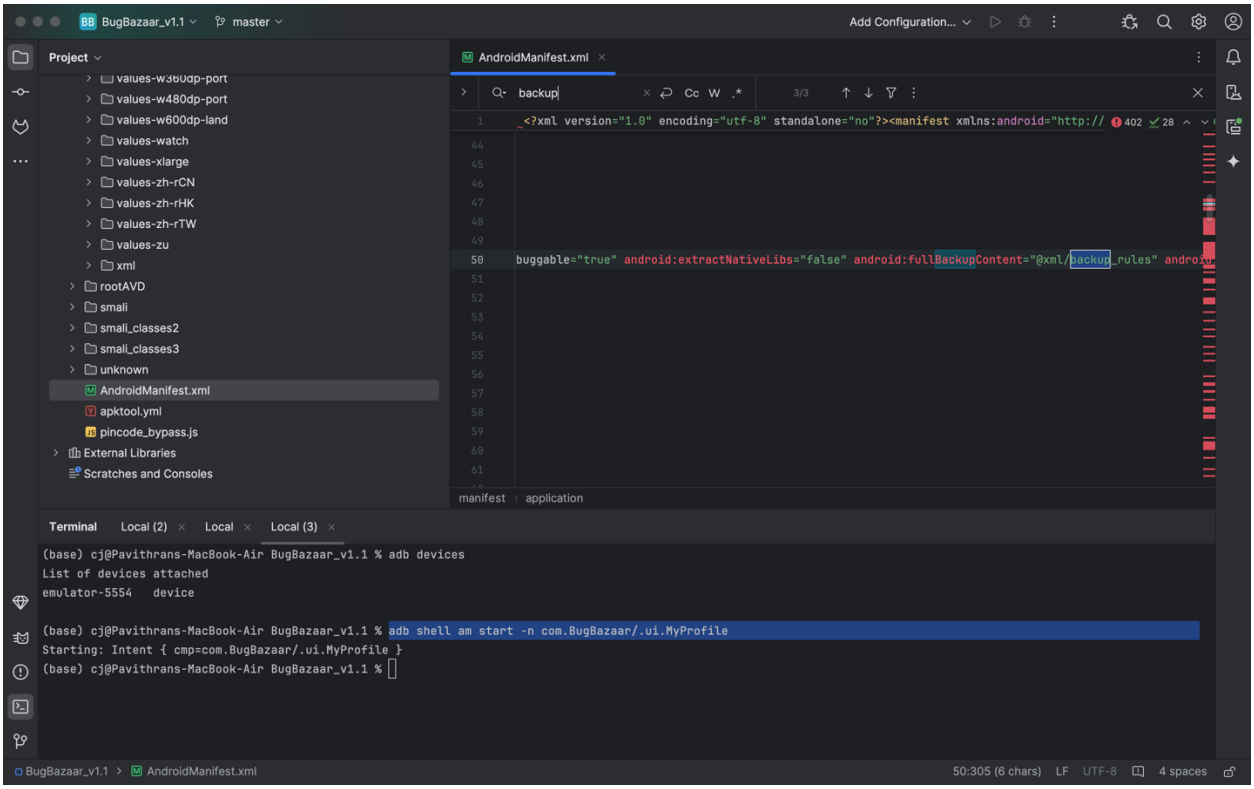
4. Using the adb shell am command we invoke this profile activity (**adb shell am start -n com.BugBazaar/.ui.MyProfile).**

5. Using this command, we can bypass the pin code screen to view the my profile screen without entering the pin code. Now we bypassed the pin code and can clearly navigate through the application. We can even change the details.

6. Here in our results, we can see that we are viewing our profile Ui which contains a lot of PII. We can see the name, email address, phone number and the user address.



7. Another example is that we can see in the manifest file that backup is enabled.
8. We can back up the data because this flag is enabled and when we backup we can get some sensitive information like the pin code etc.



Mitigation

- Make sure that activities are initiated safely by preventing access to sensitive elements by unauthorized applications or individuals. Implement appropriate authorization verifications when initiating activities to limit access solely to those who are entitled.

- Turn off backup options for confidential information. On Android, include the android : allowBackup="false" attribute in the application's manifest to stop app data from being backed up insecurely.

- Safeguard sensitive information by utilizing secure storage solutions like the Keychain on iOS or Keystore on Android. Refrain from saving sensitive data in unencrypted form or shared settings.

- Utilize runtime validations to identify any unauthorized activities or breaches. This might involve checking the app's integrity and confirming that it hasn't been altered or reverse engineered.

- Refrain from exporting activities, services, and content providers unless it is essential. By doing this, you reduce the attack surface presented to other applications or possible intruders.

- Make certain that the default settings of your app are secure, especially concerning permissions and the exposure of sensitive information.

- Avoid using hardcoded or default passwords in the application code. Establish secure methods for user verification and access control.

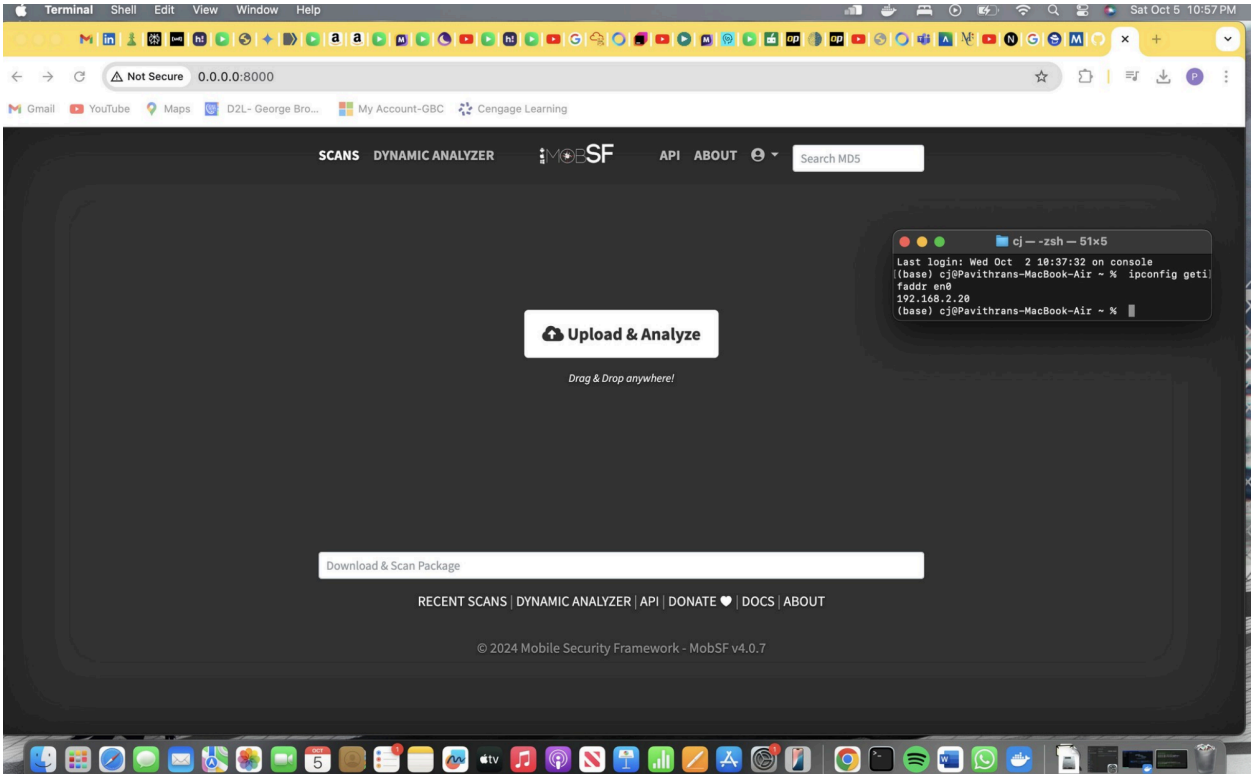
- Employ the least privilege principle by requesting only the permissions that are essential for the app's operation.

- Refrain from utilizing unencrypted traffic. Always utilize HTTPS to secure data while transmitted and think about applying certificate pinning to defend against man-in-the-middle (MITM) attacks.

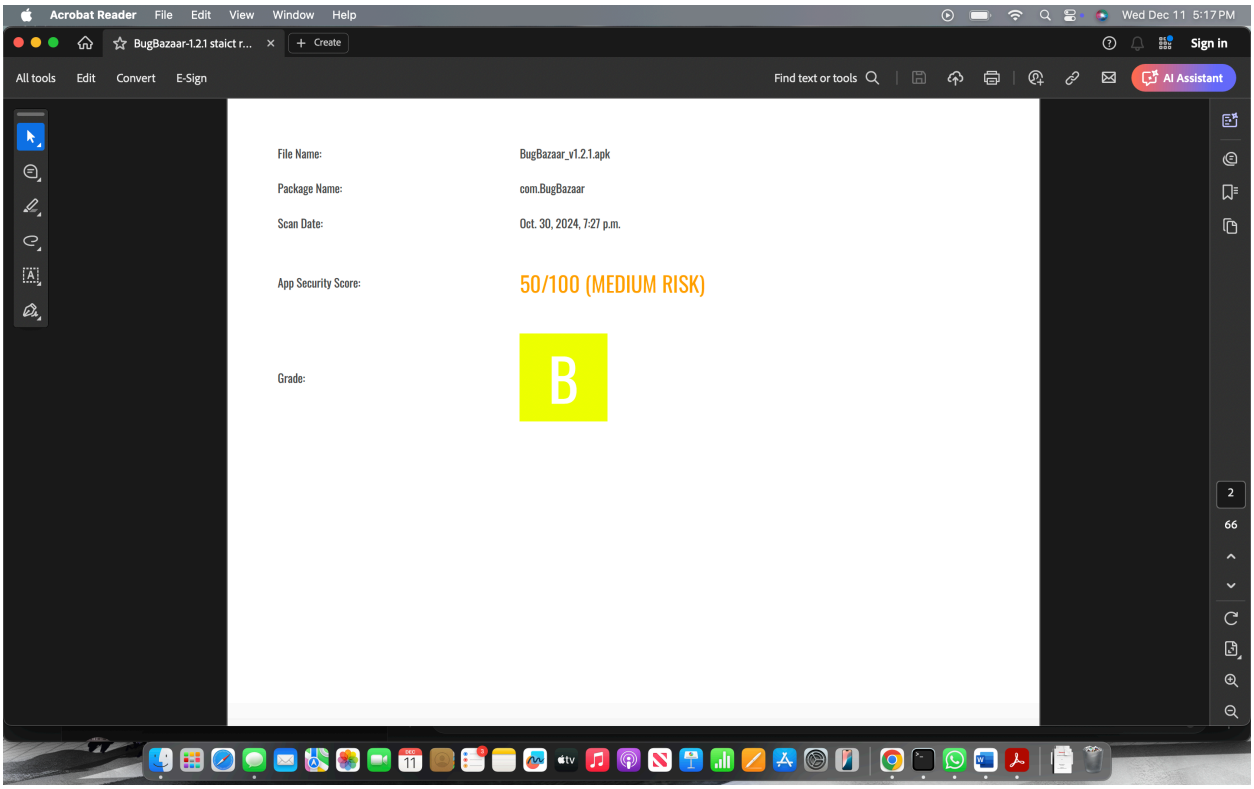
- Turn off debugging functions in the app's production versions. Make certain that debugging tools cannot be accessed by attackers.

Static Analysis

For static analysis, we utilized MobSF (Mobile Security Framework), a tool that conducts a comprehensive analysis of the APK without running it. Upon uploading the APK, MobSF produces a detailed static report that encompasses examinations for insecure permissions, hardcoded secrets, exported components, and issues related to the storage of sensitive data. It also examines third-party libraries for security flaws and makes sure that best practices, including data encryption and appropriate API access controls, are adhered to. The static report aids in spotting and addressing potential security threats prior to the app's launch, enhancing its overall security stance.



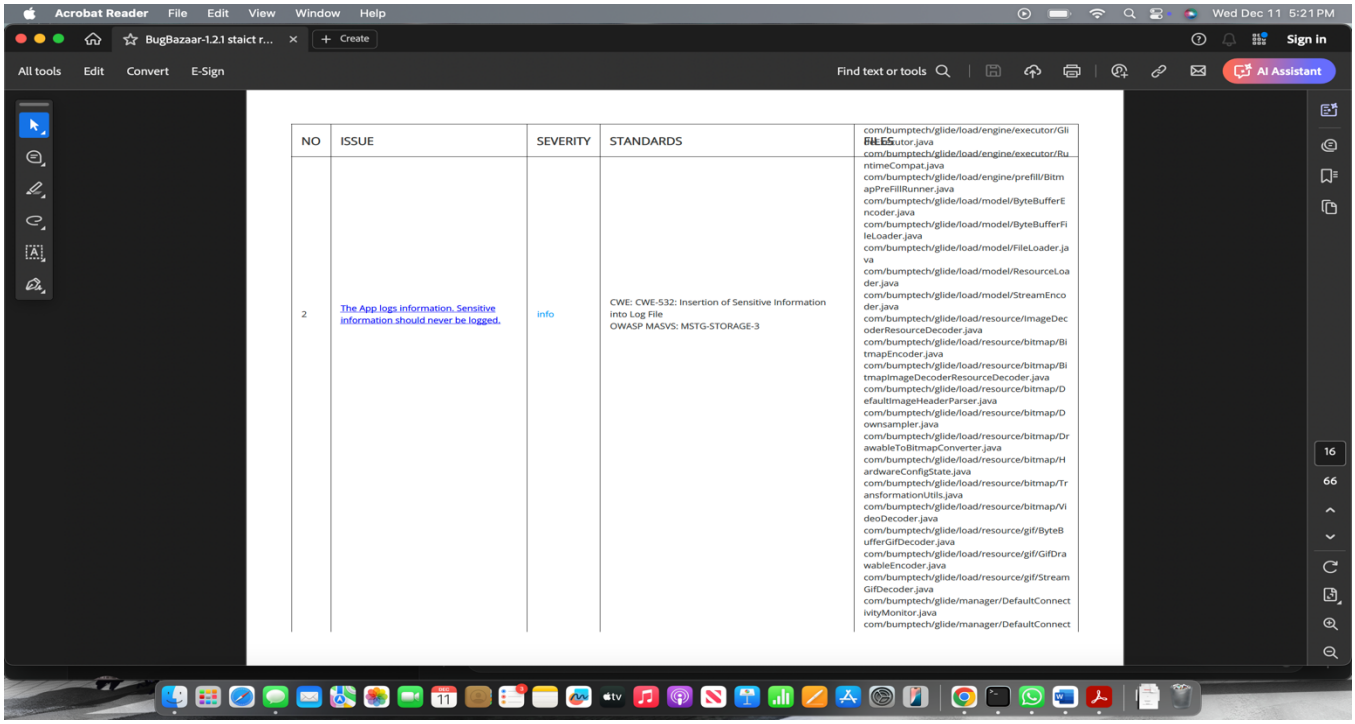
The scan took place on October 30, 2024, at 7:27 PM, resulting in a security score of 50/100, which shows a Medium Risk with a B grade. The assessment identified multiple security issues needing remediation, including insecure permissions, possible exported components, and compliance with best practices for protecting sensitive data and APIs. This assessment emphasizes aspects requiring enhancement to strengthen the app's security stance.



File Name:	BugBazaar_v1.2.1.apk
Package Name:	com.BugBazaar
Scan Date:	Oct. 30, 2024, 7:27 p.m.
App Security Score:	50/100 (MEDIUM RISK)
Grade:	B

A lot of other vulnerabilities were identified too but the two key vulnerabilities in the application we going to discuss are:

M6: Inadequate Privacy Controls



NO	ISSUE	SEVERITY	STANDARDS	FILES
2	The App logs information. Sensitive information should never be logged.	info	CWE: CWE-532: Insertion of Sensitive Information into Log File OWASP MASVS: MSTG-STORAGE-3	com/bumptech/glide/load/engine/executor/GlideExecutor.java com/bumptech/glide/load/engine/executor/RuntimeCompat.java com/bumptech/glide/load/engine/prefill/BitmapPreFillRunner.java com/bumptech/glide/load/model/ByteBufferEncoder.java com/bumptech/glide/load/model/ByteBufferFileLoader.java com/bumptech/glide/load/model/FileLoader.java com/bumptech/glide/load/model/ResourceLoader.java com/bumptech/glide/load/model/StreamEncoder.java com/bumptech/glide/load/resource/ImageDecoderResourceDecoder.java com/bumptech/glide/load/resource/bitmap/BitmapEncoder.java com/bumptech/glide/load/resource/bitmap/BitmapImageDecoderResourceDecoder.java com/bumptech/glide/load/resource/bitmap/DefaultImageHeaderParser.java com/bumptech/glide/load/resource/bitmap/Downsampler.java com/bumptech/glide/load/resource/bitmap/DrawableToBitmapConverter.java com/bumptech/glide/load/resource/bitmap/HardwareConfigState.java com/bumptech/glide/load/resource/bitmap/TransformationUtils.java com/bumptech/glide/load/resource/bitmap/VideoDecoder.java com/bumptech/glide/load/resource/gif/ByteBufferGifDecoder.java com/bumptech/glide/load/resource/gif/GifDrawableEncoder.java com/bumptech/glide/load/resource/gif/StreamGifDecoder.java com/bumptech/glide/manager/DefaultConnectivityMonitor.java com/bumptech/glide/manager/DefaultConnect

The static analysis identified a severe M6: Insufficient Privacy Controls vulnerability related to the recording of sensitive data. The application was discovered to record data across various files and classes associated with the Glide library, including but not limited to:

- GlideExecutor.java
- RuntimeCompat.java
- BitmapEncoder.java
- DefaultImageHeaderParser.java
- VideoDecoder.java
- DefaultConnectivityMonitor.java

This problem corresponds to CWE-532: Inclusion of Sensitive Data in Log Files and breaches the OWASP Mobile Application Security Verification Standard (MASVS) requirement MSTG-STORAGE-3, which states that sensitive information must never be recorded. Recording such information can subject users to considerable privacy threats, including possible data breaches and unauthorized access, particularly on shared or insecure devices.

Mitigation

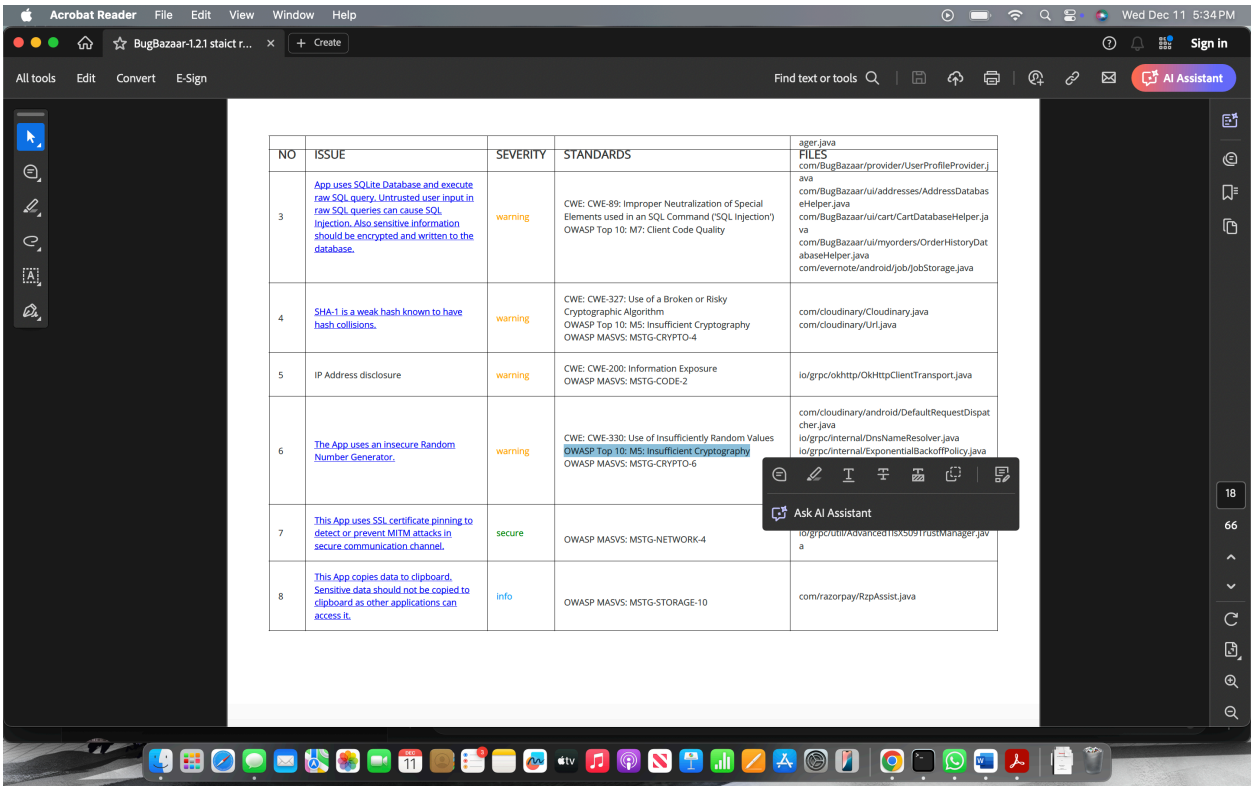
Core Mitigation Strategies

- Examine all logging systems to detect and eliminate any confidential data. Substitute confidential information in logs with anonymized placeholders (e.g., "User_ID: [REDACTED]"). Set up logging frameworks to omit sensitive data by default.
- Utilize secure logging libraries that provide encryption for log files. Enable logging solely in non-production environments. Turn off logging in production environments or restrict it to necessary operational information.
- Make sure logs are kept only for a required time frame, utilizing automated systems to remove logs after a predetermined period.
- Implement stringent access restrictions on log files to block unauthorized access.

OWASP general Suggestions

- Perform a comprehensive audit of all PII resources within the application.
- Substitute detailed information (e.g., GPS coordinates) with broad data.
- Employ anonymization methods like hashing or introducing noise to sensitive information. Implement expiration policies to remove PII once its intended use has been fulfilled.
- Enable users to agree to the collection of non-mandatory PII. Give explicit alerts regarding which PII is being gathered and the reasons for its collection.
- Encrypt PII both at rest and during transit, employing robust encryption standards (e.g., AES-256). For sensitive information (e.g., medical records), utilize multi-tier encryption, like employing hardware-backed keys specific to each device.

M10- Insufficient Cryptography



NO	ISSUE	SEVERITY	STANDARDS	FILES
				ager.java
3	App uses SQLite Database and execute raw SQL query. Untrusted user input in raw SQL queries can cause SQL injection. Also sensitive information should be encrypted and written to the database.	warning	CWE: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') OWASP Top 10: M7: Client Code Quality	com/BugBazaar/provider/UserProfileProvider.java com/BugBazaar/ui/addresses/AddressDatabaseHelper.java com/BugBazaar/ui/cart/CartDatabaseHelper.java com/BugBazaar/ui/myorders/OrderHistoryDatabaseHelper.java com/evernote/android/job/JobStorage.java
4	SHA-1 is a weak hash known to have hash collisions.	warning	CWE: CWE-327: Use of a Broken or Risky Cryptographic Algorithm OWASP Top 10: M5: Insufficient Cryptography OWASP MASVS: MSTG-CRYPTO-4	com/cloudinary/Cloudinary.java com/cloudinary/Url.java
5	IP Address disclosure	warning	CWE: CWE-200: Information Exposure OWASP MASVS: MSTG-CODE-2	io/grpc/okhttp/OkHttpClientTransport.java
6	The App uses an insecure Random Number Generator.	warning	CWE: CWE-330: Use of Insufficiently Random Values OWASP Top 10: M5: Insufficient Cryptography OWASP MASVS: MSTG-CRYPTO-6	com/cloudinary/android/DefaultRequestDispatcher.java io/grpc/internal/DnsNameResolver.java io/grpc/internal/ExponentialBackoffPolicy.java
7	This App uses SSL certificate pinning to detect or prevent MITM attacks in secure communication channel.	secure	OWASP MASVS: MSTG-NETWORK-4	io/grpc/util/AdvancedTlsX509TrustManager.java
8	This App copies data to clipboard. Sensitive data should not be copied to clipboard as other applications can access it.	info	OWASP MASVS: MSTG-STORAGE-10	com/razorpay/RzpAssist.java

The application shows cryptographic vulnerabilities that compromise its security. It utilizes the obsolete hashing algorithm SHA-1, which is susceptible to hash collision attacks, enabling attackers to generate different inputs yielding the same hash, undermining data integrity and authentication methods. Furthermore, the application employs an insecure random number generator, resulting in predictable results that can be leveraged in cryptographic tasks like key generation or token creation. These weaknesses undermine the application's cryptographic protections, risking sensitive information to threats like tampering, replay attacks, or unauthorized access.

Mitigation

- Substitute SHA-1 with cryptographically secure options like SHA-256 or SHA-3. These algorithms withstand hash collisions and offer improved data integrity.

- Substitute weak random number generators with cryptographically secure options such as java.security.SecureRandom. These guarantee unpredictable and high-entropy results, essential for secure key generation and token development.

- Utilize secure key storage methods like the Android Keystore or iOS Secure Enclave. Make certain that keys are encrypted, and that access is limited to only authorized components.

- Steer clear of unique cryptographic implementations and utilize well-known libraries such as BouncyCastle or OpenSSL. Make sure to use secure encryption standards like AES-256 for symmetric encryption properly.

- Make sure all communications utilize HTTPS with appropriate certificate verification. Employ TLS 1.2 or a newer version for secure data transfer between the application and servers.

- Conduct both static and dynamic security testing to uncover cryptographic weaknesses. Instruments such as MobSF can assist in identifying incorrect cryptographic practices within the application.

Other Vulnerabilities

M2: Insufficient Supply Chain Security

Insufficient Supply Chain Security highlights vulnerabilities stemming from flaws in the mobile app's supply chain, including insecure third-party libraries, SDKs, or hardcoded passwords. Malicious actors can take advantage of these vulnerabilities to insert harmful code or obtain unauthorized entry to the application or its backend systems, resulting in data leaks, service interruptions, or full control of the application.

We were unable to test this vulnerability directly in the APK since supply chain security problems usually arise during the development or build phase, rather than after the app is packaged. Our testing was restricted to the APK alone, which does not offer insight into the development environment, or the supply chain tools employed.

Mitigation from OWASP

- Adopt Secure Coding Practices: Embrace secure coding standards during the entire development lifecycle. Perform consistent code assessments and static/dynamic analyses to detect vulnerabilities early on. Adhere to recommended guidelines to avoid typical security vulnerabilities like injection attacks, buffer overflows, and inadequate cryptography.

- Secure App Signing and Distribution: Make sure your app is properly signed and utilize reliable distribution methods to stop attackers from inserting harmful code or changing the app. Utilize code-signing certificates to maintain the integrity of the code throughout the development, testing, and release stages.

- Utilize Reliable Third-Party Libraries: Only include libraries, SDKs, and components from credible and verified sources. Consistently assess and refresh dependencies to make certain they do not bring in vulnerabilities. Utilize tools such as Software Composition Analysis (SCA) to detect obsolete or susceptible dependencies.

- Create Firm Controls for App Updates and Releases: Establish strong governance for app updates, patches, and launches. This involves safeguarding the update procedure to guarantee that solely approved and verified updates are implemented. Employ methods like secure over-

the-air (OTA) updates to reduce the risk of adversaries interfering with the update procedure.

- Observe and Identify Supply Chain Security Events: Constantly supervise for possible vulnerabilities and attacks within the supply chain. Consistently conduct security assessments, including penetration tests and vulnerability scans, to identify possible problems in the supply chain. Establish a system for promptly addressing incidents, encompassing the tasks of patching or reverting affected updates and informing users about the breach. (OWASP, 2024)

M5- Insecure Communication

Insecure Communication denotes the exchange of information without adequate encryption, making it susceptible to interception and alteration by malicious actors. This may happen when information is transferred between the mobile application and distant servers without utilizing secure protocols such as HTTPS or when employing obsolete encryption techniques. If data is transmitted via HTTP rather than HTTPS, malicious actors on the same network can intercept and gather sensitive information like login details, personal information, or session tokens. Bluetooth communication can be susceptible to interception (sniffing) when not adequately protected, enabling attackers to access information sent wirelessly between devices.

The danger of unsecure communication may result in severe outcomes like account hijacking, user impersonation, and the revelation of private personal information.

This vulnerability is relatively simple to identify. Contemporary applications nearly always utilize HTTPS for secure communication, so we did not emphasize testing for insecure communication in this instance. Furthermore, communication technologies such as Bluetooth, which could be susceptible to interception, are not the main method of communication for the majority of mobile applications today. Thus, we presumed that secure communication protocols were already established, concentrating on more intricate vulnerabilities instead.

Mitigation from OWASP

- Utilize SSL/TLS for Every Data Transfer: Confirm that all information shared between the mobile application and the backend is sent through a secure protocol, like SSL/TLS. This stops attackers from capturing confidential information.
- Utilize Verified Certificates: Consistently employ certificates that are signed by a reputable Certificate Authority (CA). Always avoid self-signed, expired, or untrusted certificates. Carry out SSL chain verification to confirm the server's identity prior to establishing a connection.

- Utilize Certificate Pinning: Think about applying certificate pinning, which entails incorporating a particular certificate or public key into the app to guarantee that it only interacts with the designated server, minimizing the likelihood of man-in-the-middle attacks.

- Prevent Mixed SSL Sessions: Refrain from combining secure and insecure connections (e.g., SSL and HTTP). This could reveal session IDs along with other confidential information. Make sure that the complete communication session is protected with SSL/TLS.

- Encrypt Sensitive Information Prior to Sending: Whenever feasible, encrypt sensitive information before transmitting it via SSL/TLS. This provides an extra level of protection, guaranteeing data confidentiality even if there's a flaw in SSL/TLS.

- Ensure Certificates are Properly Validated: Avoid bypassing SSL verification processes while developing. Always verify certificates and terminate the connection if there are issues with the certificate, like it being invalid or untrusted.

- Observe Traffic for Unencrypted Channels: In security evaluations, examine whether any information is being sent through unencrypted channels, which should be prevented to maintain data security.

- For iOS, verify that certificates are legitimate and fail securely. Utilize the Secure Transport API to define trusted client certificates.

- For Android, eliminate any code that permits the app to accept all certificates and confirm that the certificate validation is correctly applied. (OWASP, 2024)

M7- Insufficient binary protection

Insufficient Binary Protections refer to vulnerabilities that make an app susceptible to reverse engineering or tampering. Attackers can decompile the app's binary to uncover sensitive information like API keys or hardcoded credentials. They might also modify the app's code to bypass security measures, remove licensing checks, or inject malicious code.

Such vulnerabilities can lead to unauthorized access to paid features, unauthorized data access, or harmful changes to the app's behavior.

We had already performed reverse engineering on the app for other vulnerabilities. Addressing this specific issue would have required additional steps, such as altering the smali code or applying runtime manipulations. Since other testing methods already covered some aspects of potential binary tampering and reverse engineering, I decided to exclude further detailed testing for this specific vulnerability at this stage.

Mitigation from OWASP

- **Threat Modeling**: Assess whether the app contains critical content or if its popularity necessitates binary protection. Conduct a threat modeling analysis to identify the highest risks and their potential financial impact.

- **Minimize Information Exposure**: Ensure apps only access the minimum necessary information to function, as this information is always at risk of being leaked or manipulated.

- **Prevent Reverse Engineering**: Use obfuscation tools (both free and commercial) to make the app binary incomprehensible. Compile parts of the app natively (for iOS and Android) or use interpreters/nested virtual machines to make reverse engineering harder, as many decompiling tools only support one language and binary format. Balance the complexity of the code with robustness against reverse engineering, as full obfuscation may affect libraries that rely on certain strings or symbols.

- **Enforce Security Mechanisms**: Combine obfuscation with backend-enforced security checks to prevent manipulation. Use integrity checks to detect code tampering and render the app unusable if tampering is detected, though these checks can also be found and deactivated by attackers. (OWASP, 2024)

M9- Insecure Data Storage

Insecure data storage in mobile applications can lead to significant security risks, including direct data extraction and interception of sensitive information. This vulnerability often arises from weak or nonexistent encryption, making it easy for attackers to access and decipher sensitive data. Additionally, storing data in easily accessible locations within the device's filesystem, such as plain text files or unprotected databases, exposes it to unauthorized extraction or manipulation. The impact of exploiting this vulnerability includes the disclosure of sensitive information to attackers, which can have severe consequences.

We did not focus on this vulnerability because it overlaps with other areas I have already tested. During the assessment of insecure authentication, I examined how shared preferences could be accessed. Additionally, in the security misconfiguration tests, I demonstrated how data could be backed up and potentially exposed. Since these tests already covered aspects of insecure data storage, I chose not to perform further detailed testing on this specific vulnerability. Moreover, conducting a demo would have required providing sensitive card details, which was not feasible.

Mitigation from OWASP

- Implement robust encryption algorithms to protect data at rest and in transit. Ensure encryption keys are securely stored and managed.
- Use secure communication protocols (e.g., HTTPS, SSL/TLS) to protect data during transmission.
- Store sensitive data in secure locations using platform-specific mechanisms like Keychain (iOS) or Keystore (Android).
- Implement strong access controls, authenticate users securely, and enforce role-based access controls.
- Use input validation and data sanitization to prevent injection attacks and ensure only valid data is stored.
- Use secure session management techniques, such as random session tokens and proper session timeouts.
- Keep libraries, frameworks, and third-party dependencies up to date with security patches. (OWASP, 2024)

Conclusion

Using the OWASP Mobile Top 10 methodology and the BugBazaar application, SecureWave Technologies' internal red team training has been essential in locating and fixing serious security flaws. In order to ensure the safe development of our applications, this effort sought to improve the expertise of our development and security staff. We learnt a lot about the security issues with mobile apps by identifying important risks such incorrect credential usage, insecure authentication/authorization, inadequate input/output validation, and security misconfiguration. The development of SecureWave's secure applications will be greatly influenced by these discoveries, reaffirming our dedication to creating dependable, data-protecting applications. Our knowledge of mobile app security has increased as a result of the training, and we are now more determined than ever to include security into every phase of our development process.

Recommendations

We advise using strong encryption techniques and secure key management to further fortify the security of our mobile apps and safeguard private information while it's in transit and at rest. Potential vulnerabilities can be found and fixed early in the development process by enforcing rigorous adherence to secure coding standards and carrying out frequent code reviews. Furthermore, regular penetration tests and security audits will find and fix new vulnerabilities as they appear. Giving developers continual security training will keep them abreast of developing dangers and best practices, ensuring they are equipped to tackle new problems.

Sensitive information kept on smartphones can be protected by using secure storage solutions like Keychain for iOS and Keystore for Android. By guaranteeing that only authorised users may access sensitive data, the implementation of robust access controls, such as role-based access controls and multi-factor authentication (MFA), will further improve security. The integrity and security of our apps will be preserved by verifying and cleaning input to stop injection attacks and making sure that only legitimate data is processed. Lastly, by carefully screening third-party libraries and dependencies, we may improve supply chain security and make sure they don't add risks to our apps. SecureWave Technologies may continue to improve its security procedures and guarantee the creation of reliable, secure mobile applications by implementing these steps.

Reference

Android BugBazaar: Your mobile appsec playground to Explore, Exploit, Excel. (2024, August 14). Retrieved from https://github.com/payatu/BugBazaar

JADX. (n.d.). Retrieved from https://github.com/skylot/jadx

OWASP. (2024). Retrieved from https://owasp.org/www-project-mobile-top-10/

Payatu. (n.d.). *Payatu*. Retrieved from https://payatu.com/mobile-application-security-testing/