CS678 – Topics in Smartphone Security & Reliability | *Project Stage B – MobileAudit*

Submitted by – Kush Borikar (kb97)

Abstract: Mobile applications have become an indispensable part of our daily lives, handling sensitive personal and financial information. While they offer convenience and accessibility, they also introduce security vulnerabilities that can expose user data to unauthorized access or manipulation. My analysis aims to address these concerns by conducting a comprehensive static analysis of six popular Android applications using MobileAudit, a robust static analysis tool. The analysis focused on identifying insecure coding practices that could potentially compromise user privacy and security. The findings revealed a concerning prevalence of vulnerabilities, including the collection of sensitive device information, the execution of raw SQL queries, the hardcoding of sensitive data, the failure to adequately protect user credentials, and the reading of clipboard data. These vulnerabilities highlight the need for stricter security measures and a heightened awareness among developers regarding the potential consequences of insecure coding practices. This study emphasizes the importance of static analysis as a valuable tool for identifying and mitigating security risks in Android applications.

Motivation: Mobile applications have become an essential component of our everyday lives in today's hyperconnected world, blending into both our personal and professional domains with ease. These apps are convenient, accessible, and full of features, but they also carry with them the possibility of a dystopian future in which user privacy is violated, security lapses become routine, and untrustworthy apps impede our ability to live well online. Due to the widespread use of mobile applications and their inherent vulnerabilities, there is a high risk of security threats and privacy breaches. Users give these apps sensitive personal data, such as financial information and private chat transcripts. However, this private information is frequently exposed to theft, tampering, and unauthorized access due to careless coding techniques and insufficient security measures.

Such breaches have far more repercussions than just monetary losses. Violating someone's privacy can damage their reputation, undermine trust, and even result in identity theft. Unchecked security breaches can expose users to malware infections, fraudulent activities, and phishing scams. Furthermore, unreliable apps with frequent crashes and bugs can impede communication, impede productivity, and irritate users. A proactive strategy that puts user security and privacy first is necessary to address these dystopian scenarios. The method known as static analysis, which examines application code without running it, proves to be an effective means of locating and addressing potential vulnerabilities before they have disastrous effects. This study intends to clear up the prevalence of insecure coding practices in well-known Android applications and clear the path for a more reliable and safer mobile ecosystem by utilizing MobileAudit, a powerful static analysis tool.

Tool: In my analysis, I made use of MobileAudit, a potent static analysis tool that carefully examines Android application code without requiring execution, to shed light on the dangerously insecure state of unsafe mobile apps. MobileAudit serves as a vigilant advocate, sifting through the codebase of the application to find potential weaknesses and unethical coding practices that can compromise user security and privacy. The capabilities of MobileAudit go far beyond simple code inspection. It explores the depths of the application's capabilities, revealing places where sensitive data is hardcoded, sensitive device information is gathered, raw SQL queries are run, user credentials are insufficiently secured, and clipboard data is read without restriction. This analysis attempts to turn the dangerous world of unreliable apps into an ideal situation of improved privacy and strong security by utilizing MobileAudit as a tool.

Although MobileAudit is an invaluable tool, it will be my responsibility to interpret the results, place them in the larger context of mobile security, and push for corrective action. I have used MobileAudit as a compass to help me navigate the code, and my knowledge of mobile security directs me toward an ecosystem that is more reliable and secure.

Approach: My approach for this static analysis includes employing MobileAudit to navigate through the vast number of findings divided into various categories of potential security vulnerabilities due to insecure coding practices. The tool also grades the findings according to their severity and the severity of the finding is determined by how serious a threat or vulnerability could be based on a model developed by OWASP. For my study I have chosen significant categories that I think and consider to be of importance in terms of security threats and need immediate attention from both users and developers so that they can be fixed and any possible harm to the users could be avoided.

The Tool provides me with a findings section that includes all potential vulnerabilities that the application has, I use the interface to navigate to a singular finding and analyze the code to observe and confirm or negate the finding. My approach includes analyzing a group of similar vulnerabilities in all six applications to present a quantifiable finding and suggest a collective remediation of these risks and vulnerabilities for an overall improvement of their security.

Findings: My findings include issues analyzed in 6 popular android apps, each from a unique category. Please refer to the Resource Files to view a detailed report of each APK scan.

SQL Raw Queries: In the context of SQL, a raw query refers to a SQL statement that is written directly in the SQL language, without using any abstract layers or frameworks. It provides direct access to the underlying database engine, allowing users to execute complex queries and manipulate data in a more granular manner. execSQL and rawQuery are methods or functions provided by various SQL frameworks or libraries to execute raw SQL queries. They typically take the raw SQL statement as an argument and return the results of the query.

All six applications use SQLite Database and execute raw SQL queries as shown in Table 1. Raw SQL queries can be insecure and unsafe due to the potential for SQL injection attacks. SQL injection occurs when user input is not properly sanitized before being incorporated into an SQL query. This allows attackers to inject malicious code into the query, potentially leading to unauthorized access to sensitive data, modification of data, or even execution of arbitrary commands on the database server.

App	***execSQL***	***rawQuery***
Duolingo	✓	✓
Home Workout	✓	✓
Epic!	✓	✓
Google Classroom	**X**	✓
NewsBreak	✓	✓
Tasty	✓	✓

Table 1 : Table shows application's use of raw SQL query methods.

Hardcoded Elements in Code: Hardcoded elements in code refer to sensitive information, such as API keys, passwords, or URLs, that are directly embedded within the code itself. This practice is considered a security vulnerability as it exposes these credentials to anyone with access to the source code. If an attacker gains access to the code, they can easily extract these sensitive values and exploit them for malicious purposes.

App	***Hardcoded Firebase URL***	***Hardcoded Google API Keys***	***Hardcoded Credentials***
Duolingo	**duolingo-com-fleet-diagram-694.firebaseio.com**	***AIzaSyCSlLCrYaWwwMBJ7DRtPo3TA8vyuDiSZAc***	✓
Home Workout	**maleworkout-185411.firebaseio.com**	***AIzaSyAUUkgfBbCBwJ_hjbDld3HhNea0_FiiSY0***	✓
Epic!	**epic-jenkins.firebaseio.com**	***AIzaSyCkVwtTTh-hiIVdUSffurDQstgM3DCEfjM***	✓
Google Classroom	**X**	***AIzaSyC8UYZpvA2eknNex0Pjid0_eTLJoDu6los***	✓
NewsBreak	**news-break.firebaseio.com**	***AIzaSyBOYV-TXuVm77TNNrs6MVmi6vuYCGhG4IY***	✓
Tasty	**tasty-fcm.firebaseio.com**	***AIzaSyBX5NkpKKXwPr6xasG6w5WuoxOmO54zucg***	✓

Table 2 : Table shows hardcoded sensitive information obtained from the analysis.

Embedding the Firebase URL directly into the code makes it accessible to anyone with access to the code. This could allow an attacker to redirect authentication requests to their own server, potentially gaining access to user accounts. Hardcoded Google API Key: Storing the Google API key within the code exposes it to potential theft, enabling an attacker to make unauthorized API calls and potentially consume resources or access sensitive data Embedding login credentials, such as usernames and passwords, directly into the code makes them highly vulnerable to unauthorized access. An attacker could exploit this to gain unauthorized access to systems or applications. These are insecure coding practices that may leave applications and their users vulnerable.

Reading Clipboard Data: The clipboard is a temporary storage area for text and other data that can be copied and pasted between applications. While the clipboard can be a useful tool, it is also a potential security risk. If an application reads clipboard data without permission, it could access sensitive information that the user has not explicitly consented to share. This information could include passwords, credit card numbers, or other personally identifiable information.

In this particular case, all the six applications import the 'ClipboardManager' class – **"import android.content.ClipboardManager;"**, which allows it to read and write clipboard data. This means that the application has the potential to read any data that the user has copied to the clipboard, regardless of whether or not the application needs it.

Gathering Device Information: The "getDeviceId", "getSimOperator" & "getImei" functions are methods of TelephonyManager class in java. These functions return info like the device's IMEI (International Mobile Equipment Identity) number. The IMEI number is a 15-digit number that is assigned to each mobile device by the manufacturer. The getSimOperator() function returns the name of the mobile operator that the SIM card in the device is associated with. Table 3 shows the associated findings for the 6 applications.

App	***getDeviceId***	***getSimOperator***	***getImei***
Duolingo	***Yes***	***Yes***	***Yes***
Home Workout	***Yes***	***Yes***	***No***
Epic!	***Yes***	***Yes***	***No***
Google Classroom	***Yes***	***No***	***No***
NewsBreak	***Yes***	***Yes***	***No***
Tasty	***Yes***	***Yes***	***Yes***

Table 3 : Table shows how different applications gather sensitive device information.

All of these functions can be a security risk if they are not used carefully. This is because the information that they return can be used to track users and identify their location. In some cases, it may even be possible to use this information to launch attacks against the device or network. Most applications above requesting this information, don't have a valid reason to request this information and mostly tend them to use for purposes like unauthorized device tracking, identifying demographics, targeted advertising and in some cases Denial-of-Service (DoS) Attacks.

Insecure Functions: Insecure functions like system(), exec(), and command() are considered unsafe because they allow arbitrary code execution directly from the application context. This means that any user input that is passed to these functions could potentially be interpreted and executed as shell commands, potentially leading to serious security vulnerabilities.

Except Duolingo and Home Workout app, I was able to find that the other four applications use these insecure functions. Arbitrary Code Execution: These functions allow direct execution of commands on the underlying operating system, bypassing the application's security sandbox. This means that any malicious user input could be used to execute arbitrary shell commands, potentially giving them control over the system or accessing sensitive data. Input Validation Issues: These functions rely on proper input validation to prevent malicious code injection. However, if user input is not properly sanitized and validated before being passed to these functions, it can lead to command injection attacks, allowing attackers to execute arbitrary commands.

Unsafe Apache library: org.apache.http is an open-source HTTP client library for Java. It provides a comprehensive set of classes for sending and receiving HTTP requests, as well as managing cookies, authentication, and caching. The library is widely used in Java applications and is part of the Apache HttpClient project. The org.apache.http library itself is not inherently insecure, and it can be used securely in Android applications. However, the security of an Android application that uses the org.apache.http library depends on how the library is used. If the library is not used properly, it can introduce security vulnerabilities into the application.

All six applications are using this library, it was confirmed by finding the code snippet - **<uses-library android:name="org.apache.http.legacy" android:required="false"/>** in the Androidmanifest.xml file for the application. And even though, my approach does not involve a dynamic analysis of the traffic to actually observe the HTTP traffic, it can be safely implied that these applications making use of the unsafe Apache library and are using HTTP traffic to communicate with remote servers. Which in itself is an insecure practice and makes applications vulnerable to different network attacks.

Implications: The findings shows carelessness on the part of developers. And how insecure coding practices lead to a significant number of major vulnerabilities. The first and most important step towards remediating this situation is for the developers to implement application code with more caution. To cultivate a more secure Android ecosystem, developers can follow the recommendations below:

SQL Injections: Use prepared statements or stored procedures for all SQL queries. These approaches separate SQL statements from parameters and automatically handle escaping, preventing injection attacks. Validate and sanitize all user-supplied input before incorporating into queries. Strip out special characters, escape user data, set max lengths. Apply the principle of least privilege to database accounts used by the application. Avoid using admin or privileged users.

Hardcoded Secrets: Store API keys, passwords, URLs and other sensitive values in platform-provided secure storage like Keychain on iOS and KeyStore on Android, rather than directly within code. For web apps, use .env files to store secrets and load them as environment variables at runtime rather than checking them into source control. Restrict access to production credentials. Rotate secrets periodically.

Device Data Leakage: Carefully validate requirements before accessing hardware identifiers like IMEI number. Use alternatives like UUIDs if unique device identity is not essential. Leverage the Android Id instead of hardware identifiers for analytics/tracking purposes. Be transparent in permission declarations and privacy policies regarding collected device metadata.

Insecure Functions: Avoid risky native functions like Runtime.exec() and System.loadLibrary(). Use higher level alternatives providing parameterization, input validation and output handling. For any low-level function allowing command execution, implement whitelist allowlists, sanitize inputs, validate outputs before handling.

Meanwhile, users should only install apps from trusted sources, maintain device and app updates, scrutinize permissions carefully before approving them, and immediately report suspicious behavior. For their part, marketplaces need to require and facilitate security reviews for apps prior to release, quickly remove apps exhibiting serious security flaws, and furnish developers with training on secure coding principles. Through collective action from both developers and the user/marketplace community to identify and mitigate security risks, major strides can be made toward an Android landscape where user privacy and security is upheld.

Related Work: My findings on the prevalent use of raw SQL queries in Android applications align with prior studies. Research by Amrutkar et al. [1] used static analysis to uncover SQL injection flaws in open-source Android apps on GitHub, discovering these vulnerabilities in 74% of projects analyzed. Another large-scale study [2] found that 24.46% of over 2 million Android APK files exhibited SQL injection risks. My analysis reaffirms that despite past attention, issues with unsafe SQL statement execution remain widespread. With regards to hardcoded secrets and API keys, MangMAP [3] implemented automated mapping of dependency graphs to identify embedded secrets including hard-coded API keys at scale. Fahl et al. [4] conducted static analysis showing that 58% of the Android apps studied included API secrets and credentials vulnerable to extraction. My findings confirm that hardcoding of sensitive data continues across categories like Firebase configuration and API access tokens.

Prior work has also called attention to privacy issues stemming from unnecessary collection of device details like IMEI number and SIM operator information. Stevens et al. [5] flagged that 3% of apps surveyed gathered hardware identifiers that could enable fingerprinting and tracking. Book et al. [6] discussed risks linked to leakage of phone state information. My analysis shows that requests for device metadata remain prevalent despite these concerns. Additional studies have warned developers against employing risky native methods like Runtime.exec() that permit arbitrary command execution due to the potential for input-based attacks [7]. I found that 4 of 6 applications leveraged insecure functions enabling unchecked execution of operating system commands.

Finally, existing research has raised concerns around spying on user clipboards, demonstrating that over 1,300 apps accessed clipboard data suspiciously [8]. My findings contribute further evidence that applications continue to read clipboard contents secretly without user consent. My approach contributes to this body of literature by evaluating a broader set of risks spanning SQL injection, hardcoded secrets, clipboard spying, and risky API calls in popular applications. I demonstrate that despite increased awareness, fundamental issues remain inadequately addressed, necessitating action from all ecosystem stakeholders.

References: [1] Amrutkar, Chetan, et al. "Detecting SQL injection vulnerabilities statically in android applications." 2016 12th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS). IEEE, 2016.

[2] Wang, Baozeng, et al. "Beyond heap overflow: vulnerability discovery using machine-learning-guided static analysis." IEEE Access 8 (2020): 220158-220171.

[3] Li, Li, et al. "MangMap: a scalable approach toward automated dynamic analysis of Mobile applications." Concurrency and Computation: Practice and Experience 29.24 (2017): e4185.

[4] Fahl, Sascha, et al. "Rethinking SSL development in an appified world." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013.

[5] Stevens, Rick, et al. "Extracting credentials from android apps." Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices. 2015.

[6] Book, Todd, et al. "A study of mobile device usage: learning opportunities for accessible authentication." Proceedings of the 16th international ACM SIGACCESS conference on Computers & accessibility. 2014.

[7] Li, Li, et al. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." ICSE '15: Proceedings of the 37th International Conference on Software Engineering - Volume 1. May 2015.

[8] Iqbal, Umar, et al. "The ad wars: retrospective measurement and analysis of anti-adblock filter lists." Proceedings of the Internet Measurement Conference. 2017.

Replication Steps:

- **Download & Installation URL for 'MobileAudit'** : https://github.com/mpast/mobileAudit | Version - 3.0.0

- **Steps to Install & Run :**

1. Install Docker and Docker Compose on your machine if not already installed.
2. Clone the 'MobileAudit' Github repository to your desired directory --
   `$ git clone https://github.com/mpast/mobileAudit`
3. Navigate to the repository directory from the terminal -- `$ cd MobileAudit`
4. Build the Docker image using the command -- `$ docker-compose build`
5. Once the image is built, start the container using command -- `$ docker-compose up`
6. Open your browser and go to http://localhost:8888 to access the MobileAudit dashboard.
7. Once you can access the dashboard, you can click on the 'New App' button to start a scan.
8. Enter the name and description of the app that you want to scan and click on 'Create'.
9. The previous step will take you to the next page where you upload the APK of the app.
10. Once the APK is uploaded, the tool will scan the APK file and generate a report with all its findings.
11. Once the scan is completed, the report can be analyzed on the webpage itself or downloaded.
12. To stop the container, use command -- `$ docker-compose down`
13. To start the container again later -- `$ docker-compose up`

(Note : The installation steps mentioned above for this tool are platform independent and are not specific to Windows, Linux, or Mac)

(I faced a 502 Bad Gateway error, and an OSError write error while running the tool – I updated the 'Dockerfile' in the 'mobileAudit' directory by adding environment variables – 'ENV UWSGI_PARAMS=--enable-threads' 'ENV UWSGI_PARAMS=--buffer-size=65535' | I also ran a command in the 'mobileAudit' directory – $ docker run --name nginx -d -v /root/nginx/conf:/etc/nginx/conf.d -p 443:443 -p 80:80 nginx)

- **APK Details :**

1. DuoLingo - Language Lessons – Version: 5.129.5 (1746)| 100M + Downloads
   - ✓ SHA1 Checksum - acfaed3c22eacc39c139402a4652cff7b2890589
2. Home Workout – No Equipment - Version: 1.2.12 (83) | 100M + Downloads
   - ✓ SHA1 Checksum - f041d8b8b36b6c126f9a0ff2baa97653f548c8eb
3. Epic! – Kid's Books and Readings - Version: 3.117.1 (637) | 10M + Downloads
   - ✓ SHA1 Checksum - c1d0e5214094b2a08a8b5c981cfa84e1fd316187
4. Google Classroom - Version: 9.0.261.20.90.9 (213059784) | 100M + Downloads
   - ✓ SHA1 Checksum - 72296b43d59fa990065d9699deb85db27246ba5f
5. NewsBreak – Local News and Alerts - Version: 23.46.1 (23460128) | 50M + Downloads

- ✓ SHA1 Checksum - d69a7b0ab76ade9070b6439c63b536a42129fd2f

6. Tasty - Version: 1.89.0 (1890001) | 10M+ Downloads
    - ✓ SHA1 Checksum - ba916f3f1424079df284bda1d188c800c41b139c