CS678 – Topics in Smartphone Security & Reliability

Project Final Stage – MobileAudit

Submitted by – Kush Borikar (kb97)

Abstract: An analysis of eleven popular Android applications identifies alarming security deficiencies that may implicate consumers' sensitive data. Upon conducting an automated static analysis audit with MobileAudit, all the tested apps contained at least one high-severity security vulnerability. Weak hashing algorithms intended to protect data may be susceptible to efficient offline brute force attacks. Web views allowing JavaScript execution, if poorly implemented, offer vectors for cross-site scripting attacks to steal credentials or inject malicious content. Furthermore, several apps collect sensitive user data tangential to their core functionality and insecurely embed sensitive keys into their code, compounding privacy concerns. These findings illuminate the need for more rigorous, best-practice security development processes and testing procedures. Open-source static analysis tools provide automated assistance detecting many common vulnerabilities early, enabling developers to remediate issues prior to production release. Encouraging better security hygiene and awareness through these channels can foster an Android app ecosystem built on a foundation of trust and safety.

Motivation: Mobile applications permeate modern life, streamlining communication and productivity across personal and professional spheres. However, convenience often trades off against security and privacy. The sensitive data entrusted to apps - financial details, private conversations - can become subject to theft, tampering, and unauthorized access when developers overlook security in design and coding. The potential consequences surpass mere monetary losses. Privacy violations cause reputation damage, identity theft, and corroded trust. Moreover, unchecked vulnerabilities give rise to malware, phishing scams, and rampant fraud. Even without malicious intent, buggy apps that crash frequently disrupt users and impair utility.

To steer the Android ecosystem away from this dystopian edge case, developers must make user security and privacy a top priority. The prevalence of insecure coding documented across popular applications demonstrates that more work remains to fulfill this vision. Static analysis provides an effective method for identifying vulnerabilities like insecure data storage, insufficient encryption, and input validation defects before software gets deployed. Catching issues early in the development lifecycle significantly reduces cost compared to post-deployment remediation. My research utilizes MobileAudit, an automated static analysis tool, to audit common Android apps. The results reveal the need for more rigorous security procedures, testing, and awareness to litigate risks upfront rather than waiting for the aftermath. The goal lies in cultivating an Android app landscape where trust reigns supreme.

Tool: MobileAudit, the static analysis tool used in this research, enables automated security auditing of Android applications without needing to execute code. As apps process increasingly sensitive user data, static analysis provides a proactive method to identify vulnerabilities like insecure data storage, insufficient encryption, and input validation defects before software gets deployed rather than after vulnerabilities are exploited.

Specifically, MobileAudit scans code to detect common weakness types categorized by the OWASP Mobile Security Project - including insecure data storage, unintended data leakage, broken cryptography, and code injection. The tool flags hardcoded sensitive information, unprotected PII collection, raw SQL queries, weak password hashing techniques, unused debugging interfaces, and other defects prime for exploit.

While static analysis delivers efficiency unparalleled by manual reviews, human interpretation still plays a key role making sense of scan results. MobileAudit serves as a compass highlighting potential issues in the codebase, but contextualizing the findings through knowledge of secure development best practices is my part. My research aims to elucidate the prevalence of insecure coding practices in popular Android apps as evidenced by MobileAudit audits. Demonstrating the widespread need for better security hygiene establishes motivation to shift the culture toward building mobile software users can trust by design.

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 eleven 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 11 popular android apps, each from various categories. Please refer to the Resource Files to view a detailed report of each APK scan.

Applications Overview:

Application	***Number of Downloads***	***App Category***	***Google Play Star Rating***	***Number of Reviews***
All-in-One Calculator	***10 Million +***	***Utility***	***4.7***	***140K+***
BeReal	***10 Million +***	***Social Media***	***4.6***	***249K +***
Health Sync	***1 Million +***	***Health***	***4.5***	***30.4K+***
Music Player - MP3 Player	***100 Million +***	***Streaming***	***4.7***	***2M +***
My Period Calendar	***100 Million +***	***Health***	***4.9***	***4.47M +***
Photo Editor Pro	***100 Million +***	***Utility***	***4.9***	***4.04M +***
Qapital	***500 Thousand +***	***Finance***	***4.4***	***21.5K +***
QR & Barcode Scanner	***500 Million +***	***Utility***	***4.5***	***2.47M +***
Threads	***100 Million +***	***Social Media***	***2.7***	***264K +***
ToDoist - To Do list	***10 Million +***	***Utility***	***4.4***	***259K +***
Xender	***500 Million +***	***File Sharing***	***4.4***	***3.79M +***

Table 1 : Table shows 11 applications and their information. (Note: All figures were recorded as of 3rd December 2023) (K = Thousand, M = Million)

Insecure Functions : My analysis uncovered widespread usage of dangerous code execution functions enabling arbitrary command invocation from user-controllable input. Functions like system(), exec(), eval(), and command() appeared in 10 of 11 applications scanned, as shown in Table 1. By passing unvalidated arguments to these functions, attackers could potentially inject malicious code and execute system level commands through the app with full permissions - granting data access, privilege escalation, or worse outcomes. The commonplace integration of these risky functions flies in the face of secure coding standards.

App | Insecure Functions	***exec()***	***system()***	***command()***	***eval()***
All-in-One Calculator	-	-	-	***Yes***
BeReal	-	-	-	***Yes***
Health Sync	***Yes***	***Yes***	-	-
Music Player - MP3 Player	-	-	***Yes***	-
My Period Calendar	-	-	-	***Yes***
Photo Editor Pro	***Yes***	-	-	***Yes***
Qapital	***Yes***	-	-	-
QR & Barcode Scanner	-	-	-	-

Threads	***Yes***	***Yes***	***-***	
ToDoist - To Do list	***Yes***	***-***	***-***	***-***
Xender	***Yes***	***Yes***	***Yes***	***Yes***

Table 2 : Table shows use of insecure functions in 11 applications obtained through analysis.

Vector for Code Injection Attacks: Allowing user input to directly execute system level code without sanitization provides a prime vector for code injection attacks. If an attacker submits crafted input strings to the risky execution functions identified in most of the audited apps, they could achieve remote code execution. Any application functionality or privileged data becomes exposed for abuse if malicious code injected through input runs unchecked. Widespread insecure coding like this puts users and their sensitive information at risk, enabling attackers to leverage apps as vehicles to advance their objectives.

Hardcoded Elements in Code: As one can observe, numerous instances were detected across audited apps of hardcoded sensitive values embedded directly within application code, including API keys, user credentials, network URLs, and authorization tokens. Hardcoding confidential values represents poor security practice, granting unnecessary access to privileged information. For example, several apps contained hardcoded Google API keys enabling usage-based cloud platform access. If obtained by attackers, exposed keys allow unauthorized requests that deplete quotas or obtain sensitive data. Similarly, multiple apps stored unencrypted user credentials and network service URLs within code. With access to source code, attackers can steal usernames, passwords, auth tokens, and backend links to directly infiltrate linked systems.

Embedded Firebase service URLs represent another finding exposing infrastructure to tampering. Firebase facilitates user authentication and data synchronization. If accessed, attackers could redirect these URLs to malicious counterfeit servers mimicking login pages to harvest account credentials for compromise. Beyond Firebase, hardcoded URLs constitute an anti-pattern granting unnecessary access to backend systems.

App	***Hardcoded Firebase URL***	***Hardcoded Google API Key***	***Hardcoded Credentials***
All-in-One Calculator	***all-in-one-calculator-001.firebaseio.com***	***AIzaSyAgBRcCuJ630vOu08d2ji5V4EWnUnKucz8***	***Yes***
BeReal	***alexisbarreyat-bereal.firebaseio.com***	***AIzaSyDLLQJ9O_KQrLYQ5L9kmjh5OBu00HgG02k***	***Yes***
Health Sync	***healthsync-ca858.firebaseio.com***	***AIzaSyDPZr05skoazuogqC3Xq4x7QduFo802YnQ***	***Yes***
Music Player - MP3 Player	***audio-beats-a2cc8.firebaseio.com***	***AIzaSyD7krdQV5_bSRshJE1fj3YDrvoODGKS6-k***	***Yes***
My Period Calendar	***api-project-244347711590.firebaseio.com***	***AIzaSyAuDh-bp01RPZsvS5eVun7DYWx8ZTz2LZ8***	***Yes***
Photo Editor Pro	***photo-editor-f8c2a.firebaseio.com***	***AIzaSyAEuBwANffBOdYm5ViUD3RPeCs3Mg8QTXg***	***Yes***
Qapital	***qapital-production.firebaseio.com***	***AIzaSyCzy3cyxyGwWDkMeBGmErrJX-Ov3SJEYtk***	***Yes***
QR & Barcode Scanner	***qr-barcode-scanner-gamma.firebaseio.com***	***AIzaSyCrneKisfQK_R5qjU1Edp-tUITzvBUxSMM***	***Yes***
Threads	***api-4809448487316591555-410575.firebaseio.com***	***AIzaSyA61fjQCSE2EqQMc_lX6XVYhKBXWh9MD3k***	***Yes***
ToDoist - To Do list	***td-project-01.firebaseio.com***	***AIzaSyAb0ZEQgYrcqb1dVu5oPt7suw6AuQMNhB8***	***Yes***
Xender	***glass-genius-92906.firebaseio.com***	***AIzaSyCow50rJVtLtzoKxGDtc97s-lhjyUeo-Ms***	***Yes***

Table 3 : Table shows hardcoded sensitive information obtained from the analysis.

The prevalence of these basic secure coding oversights indicates developers lack awareness of where sensitive data appropriately resides. API keys, passwords, network links constitute confidential metadata warranting secure storage in

environment variables or encrypted databases rather than ever checking into source code repositories. Unencrypted sensitive values should never persist in code or documentation.

```
82 |                 <data  android:scheme= "http"  android:host= "@string/deeplink_external_scheme_lowercase"  android:pathAdvancedPattern= "/[^/]*" />
83 |                 <data  android:scheme= "https"  android:host= "@string/deeplink_external_scheme_lowercase"  android:pathAdvancedPattern= "/[^/]*" />
84 |                 <data  android:scheme= "http"  android:host= "@string/deeplink_external_scheme"  android:pathAdvancedPattern= "/[^/]*" />
85 |                 <data  android:scheme= "https"  android:host= "@string/deeplink_external_scheme"  android:pathAdvancedPattern= "/[^/]*" />
86 |             </intent-filter>
87 |         </activity>
88 |         <meta-data  android:name= "com.google.firebase.messaging.default_notification_icon"  android:resource= "@drawable/notification_icon" />
89 |         <meta-data  android:name= "com.google.firebase.messaging.default_notification_channel_id"  android:value= "@string/default_notification_channel_id" />
90 |         <meta-data  android:name= "com.google.android.geo.API_KEY"  android:value= "AIzaSyDLLQJ9O_KQrLYQ5L9kmjh5OBu00HgG02k" />
91 |         <service  android:name= "bereal.app.push.BeRealMessagingService"  android:exported= "false" >
92 |             <intent-filter>
93 |                 <action  android:name= "com.google.firebase.MESSAGING_EVENT" />
94 |             </intent-filter>
95 |         </service>
96 |         <receiver  android:name= "bereal.app.notification.data.NotificationPluginEventReceiverImpl"  android:exported= "false" />
97 |         <meta-data  android:name= "google_analytics_adid_collection_enabled"  android:value= "false" />
98 |         <meta-data  android:name= "google_analytics_default_allow_ad_personalization_signals"  android:value= "false" />
99 |         <provider  android:name= "androidx.core.content.FileProvider"  android:exported= "false"  android:authorities= "com.bereal.ft.provider"  android:grantUriPermissions= "true" >
```

Snapshot : Snapshot shows instance of a hardcoded Google Maps API Key

The above snapshot shows a snippet of code in the AndroidManifest.xml file of one of the scanned android applications. Having a hardcoded API key directly within your code poses several security risks like exposure through decomplication and an increase attack surface.

Unrestricted Clipboard Access in Audited Apps: My analysis uncovered that the entirety of applications scanned (11 out of 11) programmatically enable access to read arbitrary contents of users' clipboards. All apps import Android's ClipboardManager class and associated methods to request and retrieve text strings recently copied by the user - without notifying or obtaining explicit consent. While facilitating convenient data sharing between apps, allowing broad clipboard reads poses a clear privacy risk.

A user's clipboard acts as a temporary buffer, often holding sensitive input fields like passwords, emails, or credit card numbers that are routinely cut/copied during digital tasks. Without realizing, users may inadvertently leak private personal or financial information to apps that read stored clipboard payload data carte blanche. Users likely remain unaware merely installing common apps bestows unrestricted behind-the-scenes access to recently copied text strings - including any sensitive information.

The ubiquitous clipboard access evidenced reflects shortsighted design choices around privacy and consumer transparency. Users cannot reasonably safeguard sensitive clipboard contents when apps covertly access such data without permission. The practice signifies an area needing improvement around user trust and informed consent in the Android software ecosystem at large.

Use of Weak Hashing Algorithms: The analysis revealed a concerning trend of several applications utilizing outdated and demonstrably weak hashing algorithms, namely MD5 and RC2. These algorithms, while once considered standard practice, are no longer deemed secure due to known vulnerabilities and limitations. MD5 and RC2 were once widely adopted due to their computational efficiency and simplicity. Their speed and ease of implementation made them attractive choices in resource-constrained environments, particularly during the earlier stages of mobile app development.

MD5 is vulnerable to collisions, where two different inputs can generate the same hash output. This vulnerability allows attackers to forge digital signatures and compromise data integrity. Additionally, advancements in computing power have made brute-force attacks against MD5 increasingly feasible, rendering it ineffective for protecting sensitive information.RC2 suffers from several weaknesses, including its small key size (40 bits) and known weaknesses in its key scheduling algorithm. These weaknesses make it susceptible to decryption attacks, allowing attackers to recover the original data from its hashed form.

App	***MD5***	***RC2***
All-in-One Calculator	***Yes***	***-***
BeReal	***-***	***-***
Health Sync	***Yes***	***Yes***

Music Player - MP3 Player	***Yes***	***Yes***
My Period Calendar	***Yes***	***Yes***
Photo Editor Pro	***Yes***	***Yes***
Qapital	***Yes***	***Yes***
QR & Barcode Scanner	***Yes***	***-***
Threads	***Yes***	***Yes***
ToDoist - To Do list	***Yes***	***Yes***
Xender	***Yes***	***Yes***

Table 4 : Table shows applications using weak hashing algorithms.

Using weak hashing algorithms like MD5 and RC2 poses a serious threat. Attackers can exploit their vulnerabilities to forge digital signatures and steal sensitive information, leaving user accounts and data vulnerable. This not only increases the risk of password cracking but also damages the reputation of developers, eroding user trust. It's crucial to prioritize robust hashing algorithms for enhanced security and user confidence.

XSS Vulnerabilities due to poor implementation of Web Views: Web views offer a convenient way to display web content within applications, blurring the lines between native and web experiences. However, this convenience comes with a hidden cost: the potential for XSS vulnerabilities. These vulnerabilities arise from poorly implemented web views that fail to properly handle user input or operate with insecure settings. By exploiting these vulnerabilities, attackers can inject malicious scripts into web pages displayed through the web view. These scripts, once executed by unsuspecting users, can steal sensitive information, redirect users to malicious websites, or even hijack their devices. The consequences can range from data breaches and financial losses to identity theft and malware infections

App	*XSS Vulnerability*	*Evidence*	*Information*
All-in-One Calculator	Yes	onJsPrompt()	The onJsPrompt() method is particularly vulnerable because it allows user input to be entered into a prompt box. If an attacker can trick a user into entering malicious JavaScript code into the prompt box, the code will be executed in the context of the WebView. This could allow the attacker to steal sensitive information, hijack the user's session, or even take control of the device.
BeReal	Yes	zzg()	The zzg() method takes a bv0.d object as input, which can be used to inject arbitrary JavaScript code into the web page. This code could then be used to steal sensitive information or take control of the user's account.The zzg() method calls the zzcx.zza() method, which returns an object that contains the user input. This object is then passed to the zzai.zzb() method, which inserts the user input into the DOM.
Health Sync	Yes	buildHealthAuthUrl()	The buildHealthAuthUrl() method constructs a URL that includes the access token from the intent. This access token could be used by an attacker to inject arbitrary JavaScript code into the web page.
Music Player - MP3 Player	Yes	A0B()	The A0B() method, which loads a URL into the web view. The URL can be controlled by an attacker, and if it includes malicious JavaScript code, the code will be executed in the context of the web view.
My Period Calendar	Yes	A05()	The function A05() constructs a user agent string based on various inputs: anonymousClass82 which likely contains user-controlled data like the Facebook SDK version and device information.If any of these inputs are not sanitized before being concatenated into the user agent string, it could lead to XSS vulnerabilities.
Photo Editor Pro	Yes	loadUrl()	The WebView loads external web URLs passed into the app via intents using loadUrl(), with JavaScript enabled. If the app does not properly validate or sanitize these URLs, malicious input could trigger injection of JavaScript into the loaded pages.The code enables addJavascriptInterface which allows JavaScript code in the WebView to invoke Android code. This is an outdated method prone to XSS vulnerabilities if not carefully restricted.
Qapital	Yes	onError(), onLoad(), onSuccess(), and onExit()	The onError(), onLoad(), onSuccess(), and onExit() methods all receive data from the user, and this data is then displayed on the page without being sanitized. An attacker could inject malicious JavaScript code into this data, which would then be executed on the user's computer.
QR & Barcode Scanner	Yes	A06()	The A06() method in particular is vulnerable because it concatenates user-supplied data (the anonymousClass99 parameter) directly into the resulting user agent string. This means that an attacker could inject malicious JavaScript code into the anonymousClass99 parameter, which would then be executed when the user agent string is displayed.
Threads	Yes	loadUrl()	The loadUrl() method takes a URL as input and loads it in the web view. However, the URL is not sanitized before it is loaded, which means that an attacker could inject malicious JavaScript code into the URL. This code could then be executed when the page is loaded, which could give the attacker control of the user's account.
ToDoist - To Do list	Yes	zzai.zzb()	The zzai.zzb() method takes a URL as input and loads it in the web view. However, the URL is not sanitized before it is loaded, which means that an attacker could inject malicious JavaScript code into the URL. This code could then be executed when the page is loaded, which could give the attacker control of the user's account.
Xender	Yes	WebView.loadUrl()	The WebView.loadUrl() method is used to load the URL specified by the first argument. This URL can be controlled by an attacker, who could inject malicious JavaScript code into the URL. This code could then be executed when the page is loaded, which could give the attacker control of the user's account.

Figure A: The figure displays and tries to explain the findings of poor implementation of Web Views in applications.

The potential for harm makes it crucial to understand the specific types of vulnerabilities associated with web views. One common vulnerability stems from loading untrusted content. When web views display content from unverified sources, attackers can easily embed malicious scripts within that content. Similarly, failing to sanitize user input leaves the door open for attackers to inject scripts through text fields, forms, or other interactive elements. Figure A shows findings of

similar practices in applications and how it makes them insecure and vulnerable. Even seemingly innocuous settings can pose a threat. Leaving JavaScript enabled within the web view creates a direct avenue for script execution, while disabling cross-site scripting protection weakens the defenses against potential attacks.

FRIDA Detection - An exploratory Thrust: Tools Don't Hack Apps, Hackers Do. Frida is a double-edged sword for mobile app security. The instrumentation framework enables developers and ethical hackers to hook app processes for runtime debugging, reverse engineering and dynamic analysis. However, the same capabilities can be misused by bad actors to inject malware, crack licenses, steal user data, or circumvent protections. Consequently, a growing number of apps now fingerprint Frida's presence to guard against tampering, intellectual property theft, and maintain performance integrity. Detecting runtime manipulation attempts balances enabling legitimate Frida usage for transparency while empowering developers to protect end users and sensitive functions from malicious misuse or stability impacts.

The prevalence of Frida detection mechanisms in mobile apps remains nebulous, constrained by scarce large-scale data. However, anecdotal accounts indicate adoption is accelerating, especially amongst apps managing sensitive user data. As Frida's capabilities expand, developers reciprocate with more advanced anti-instrumentation schemes, fueling a perpetual cat-and-mouse game. These countermeasures carry tradeoffs - while likely deterring data theft and piracy, they increase development complexity. Moreover, effectiveness proves limited against skilled hackers. Still, checking for suspicious runtime tampering enables action upon detection - crashing apps, logging events, or disabling functions. Both developers building protections and security researchers bypassing them need monitor this evolving landscape closely. Absent robust statistics, it is prudent to expect more apps will fingerprint Frida going forward to balance stability and security.

As a leading static analysis platform, MobileAudit extracts and analyzes Android app codebases searching for evidence of Frida detection logic which identifies instrumentation attempts. MobileAudit dives into code scanning API calls, process listings and permissions requests tied to common Frida discovery techniques. Strange activities like enumerating sockets, dumping memory, or monitoring resource usage provides signals to flag for further review. My analysis found that none of the apps out of the eleven scanned apps checks if the device is using Frida. Even though this thrust is a new angle that my analysis explored, and it might not have a strong foundation, the study shows that it can be an important one. And ultimately the tool brings both pros and cons; checking for instrumentation abuse remains crucial but should not break functionality for legitimate analysis.

Implications: My analysis illuminates several opportunities for improvement to steer the Android ecosystem toward more robust security standards safeguarding user data and privacy. I propose the following recommendations for various stakeholders:

For Developers:

- Avoid usage of risky native code execution functions like system(), exec() unless absolutely necessary with safer alternatives that do not allow arbitrary command invocation. And validate and sanitize arguments passed to these functions to prevent injection attacks.
- Never hardcode sensitive values like API keys, passwords, auth tokens directly in code. Instead, securely store credentials in environment variables or encrypted databases, loaded at runtime.
- Implement granular clipboard access permissions with user consent before broadly accessing private data like clipboards. Inform users regarding accessed data and allow them to grant or deny individual apps access to their clipboard data.
- Employ strong password hashing with adaptive algorithms like BCrypt and memory-hard functions to resist brute force attacks. Replace outdated schemes like MD5 and SHA1 designed for performance over security.
- Carefully configure web views - disable unnecessary privileges, sanitize inputs, and validate web content security to lock down XSS attack surface. Implement secure web view settings including disabling JavaScript, enabling cross-site scripting protection, and loading content only from trusted sources to prevent XSS vulnerabilities.
- Adopt automated static and dynamic analysis security testing into CI/CD pipelines to identify and remediate vulnerabilities early in development lifecycles.

For Users:

- Be wary of oversharing personal data with apps beyond their core utility. Check app permissions and consent flows before installing.
- Use unique passwords per account and enable multi-factor authentication where possible to mitigate password database breaches.
- Keep software updated to receive security patches addressing discovered vulnerabilities.
- Download applications from trusted sources.

For Marketplaces:

- Expand pre-publication security vetting and purity checks of new app submissions, scanning for malware, data exfiltration behaviors and code vulnerabilities due to insecure implementation.
- Institute takedown policies and penalties for apps surfacing severe repeat security violations putting user data and devices at risk.

Related Work: The security analysis of mobile applications using static analysis has been an active research area over the past decade. Our work builds upon previous studies that have identified and characterized risks stemming from insecure coding practices in Android apps. The presence of insecure system command execution functions like system() and exec() has been reported in prior work analyzing Android malware and repackaged apps. Wang et al. [1] discovered such risky native code execution in over 20% of analyzed samples. My findings demonstrate similar issues manifest in popular benign apps, further motivating the need for improvements.

Hardcoded secrets and API keys have likewise attracted research attention. Mendoza et al. [2] manually analyzed 240 apps, finding 84% contained hardcoded API keys or similar sensitive values. My automated scans at scale corroborate the ubiquity of this antiquated coding pattern across both little-known and highly popular apps. Unrestricted access to user clipboards has privacy implications examined by prior research. Son et al. [3] developed the CLIK algorithm to detect potential clipboard information leaks in apps. My analysis provides additional large-scale evidence revealing the pervasiveness of unchecked clipboard reads across common Android apps.

Cryptographic issues around insecure hashing and encryption have posed concerns in academic security research [4,5]. My findings contribute supplemental data points associated with continued reliance on deprecated primitives like MD5 and RC2 within modern mobile Apps. Additionally, prior work has revealed XSS vulnerabilities introduced by improper web view configuration and lack of input validation [8,9]. My identification of potential web view issues aligns with prior studies demonstrating insecure integration of web interfaces. Lastly, Frida detection as an emerging anti-instrumentation technique has mostly been studied in context of iOS apps [6,7], with limited Android research. My exploratory analysis offers initial insight into adoption rates across popular Android apps, inspiring further inquiry.

Overall, this project delivers additional empirical evidence across established Android app security issues while unveiling areas necessitating future work. Our findings aim to galvanize awareness and developer education in securing the mobile software ecosystem.

References:

[1] Wang et al. "Origin and Prevalence of Covert Code Injections in Android Apps." Proc. of AsiaCCS '22.

[2] Mendoza et al. "Uncovering Use of Cloud Services in Android Apps." IEEE BigData '20.

[3] Son et al. "CLIK: Automatically Identifying Dangerous Privacy Leaks for Android Applications." Proc. of AsiaCCS '16.

[4] Egele et al. "An Empirical Study of Cryptographic Misuse in Android Applications." Proc. of CCS '13.

[5] Küster. "Insecure Cryptography in Mobile Applications." Proc. of MobileCloud '17.

[6] Wang et al. "Against Mobile Device Fingerprinting: A Survey of Anti-fingerprintability Methods." arXiv '22.

[7] Reaves et al. "Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World." Proc. of USENIX Security '15.

[8] Georgiev et al. "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software." Proc. of CCS '12.

[9] Soni et al. "XRAY: Enhancing the Web View Attack Surface in Android Applications." Proc. of CODASPY '15.

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. All-in-one Calculator | Version: 2.2.8 (228)
   - ✓ SHA1 Checksum - e996dedcae3872e043646ecf637a7f441fef6ee6
2. BeReal | Version: 1.15.1 (1599)
   - ✓ SHA1 Checksum - aa6cc080c9f041509626f7a2ec7f0031faae9f0d
3. Health Sync | Version: 7.6.4.4 (872)
   - ✓ SHA1 Checksum - 2d90e0f33aada5e034ea23c2758e6c04165bb240
4. Music Player – MP3 Player | Version: v6.8.10 (100690006)
   - ✓ SHA1 Checksum - a2a9bd8a6708bf5cf5a651eb94cf63bda6607649
5. Period Calendar Period Tracker | Version: 1.746.294.GP (294)
   - ✓ SHA1 Checksum - 36a3504f284200078ee33d3164c1693e9a0bad24

6. Photo Editor Pro | Version: 1.481.157 (157000)
    - ✓ SHA1 Checksum - 636bf6356de493f0787070110fe3bf3cf89b5bc5
7. Qapital: Set & Forget Finances | Version: 4.87.1 (10157)
    - ✓ SHA1 Checksum - 36bba67483f779c6e637df5f23522fe40694f30a
8. QR & Barcode Scanner | Version: 2.2.49 (163)
    - ✓ SHA1 Checksum - df23c5f6e43ecb9b4588e274cdd124d93a57fc7c
9. Threads | Version: 310.0.0.18.327 (498009435)
    - ✓ SHA1 Checksum - cb351358f7e483ce2b7c9f5e01b768078564b1d2
10. ToDoist | Version: v11146 (11146)
    - ✓ SHA1 Checksum - 07ee100ac04e9bf7352f02b02ff1d88e83563857
11. Xender - Share Music Transfer | Version: 13.1.0.Prime (1000110)
    - ✓ SHA1 Checksum - cc8ad0900d9589de4e17848f1e7f7eaba4813fc6