Evaluating Mobile Banking Application Security Posture Using the OWASP's MASVS Framework

Trevor Henry Chiboora
tchiboor@andrew.cmu.edu
Cylab-Africa/Upanzi Network
Kigali, Rwanda

Lenah Chacha
lchacha@andrew.cmu.edu
Cylab-Africa/Upanzi Network
Kigali, Rwanda

Theoneste Byagutangaza
tbyaguta@andrew.cmu.edu
Cylab-Africa/Upanzi Network
Kigali, Rwanda

Assane Gueye
assaneg@andrew.cmu.edu
Cylab-Africa/Upanzi Network
Kigali, Rwanda

ABSTRACT

In the context of financial gain, hackers are motivated to exploit vulnerabilities that could result in financial or data loss. Therefore, it is crucial for financial applications to undergo thorough testing to identify and address such vulnerabilities. Regrettably, many financial institutions neglect proper testing procedures and sometimes even fail to establish a suitable security release baseline. This report presents an analysis of 18 mobile applications, each belonging to a different financial institution in Africa. The selection of these applications was carefully executed, considering institutions of varying sizes, to enable a comparative assessment of security practices across different organizational scales. The assessment was conducted by evaluating the sampled applications against the Mobile Application Security Verification Standard v2.0. This is a set of checklists and guidelines by the Open Web Application Security Project (OWASP) used as a baseline for mobile application security. Due to the extensive nature of the project, the testing scope was limited to the application itself, as experienced by the end user. This included examining the application's interaction with the back-end server and observing its behavior on the user's mobile device. It is important to note that this report does not provide a comprehensive analysis, as it excludes the assessment of the server-side API and testing of business logic that requires elevated privileges within the application. Furthermore, a survey was conducted to gain insights into why developers may neglect baseline security thereby introducing potential vulnerabilities in mobile applications. The findings of this survey are also included in a short summary at the end of this document.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

VAPT, Financial Inclusion, Android Applications, OWASP MASVS v2.0

ACM Reference Format:
Trevor Henry Chiboora, Lenah Chacha, Theoneste Byagutangaza, and Assane Gueye. 2023. Evaluating Mobile Banking Application Security Posture Using the OWASP's MASVS Framework. In *ACM SIGCAS/SIGCHI Conference on Computing and Sustainable Societies (COMPASS '23), August 16–19, 2023, Cape Town, South Africa.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3588001.3609367

COMPASS '23, August 16–19, 2023, Cape Town, South Africa

ACM ISBN 979-8-4007-0149-8/23/08.
https://doi.org/10.1145/3588001.3609367

1 ABOUT THIS REPORT

Financial inclusion has been identified as an enabler for 7 of the 17 Sustainable Development Goals (SDGs) [1]. According to the World Bank, financial inclusion means that individuals and businesses have access to useful and affordable financial products and services that meet their needs – transactions, payments, savings, credit, and insurance – delivered in a responsible and sustainable way [1].

The World Bank Group considers financial inclusion a key enabler to reduce extreme poverty and boost shared prosperity[9]. The world bank also considers financial products as one of the pillars and strategies for achieving financial Inclusion. In Africa, these products have been in the form of mobile wallets, banking applications, saving applications, etc.

The primary objective of this project is to understand the vulnerability landscape with fintech mobile apps and to help organizations achieve their goal of delivering secure applications to their users by testing their software security posture. In today's context of financial transactions, ensuring software security is crucial to delivering products in a responsible and sustainable way. While it may be expensive to incorporate security into software development, it is necessary to protect users' sensitive information and build trust in the market. Hence, the project aims to support organizations in enhancing their software security posture and delivering secure applications to their users, enabling them to establish a reputation as a trustworthy and reliable company in the market.

2 INTRODUCTION

According to a report published in August 2022 by McKinsey & Company [10], Fintech is the fastest-growing startup industry in Africa, garnering 54% of known startup funding in 2021. Agile technology startups are innovating to meet the needs of large and diverse markets to get a piece of the revenues in this growing industry. As a result, Africa has had growth in financial services offering retail/SME lending, insurance, account management, payments, remittances, wealth management, wallets, and blockchain/crypto services [10]. Any industry with large financial gains is prone to hackers trying to gain it by hacking digital financial providers or their users. This has resulted in financial fraud, data theft, and identity theft crimes. If a digital financial service is compromised personal information may be disclosed putting the organization and its customers at risk. In addition, financial losses might be incurred.

A report that was published in 2018 [2] showed that these applications collect personally identifiable information (PII) and about 90% of these applications requested dangerous permissions (permissions where the app requests data or resources that involve the user's private information or could potentially affect the user's stored data or operation of other applications) [2]. For example, an app can request to access a camera and if the request is not done securely, attackers can record or take pictures. Also, since resources are shared a vulnerability in one application can be used as initial access to attack other applications on the phone or their data[3].

2.1 Objective

This research aims to enhance the security of financial applications by assessing the level of security built-in, investigating permissions and data collection, identifying vulnerabilities in Android applications, and providing recommendations to improve security. A rigorous methodology will be employed, encompassing both manual and automated testing, analysis of source code, and surveys conducted among developers. Our findings will be presented in a detailed report to aid financial organizations, application developers, and policymakers.

3 METHODOLOGY

To ensure the diversity of the sample, applications were carefully selected from the Play Store from financial organizations of varying sizes. Additionally, the apps were classified based on their risk levels, with designations of top, middle, and low tiers. Table 1 provide more details on each tier. Moreover, considering that the applications are related to finance, the strictest level of the MASVS, L2, was applied.

In order to gather data on Android applications in the financial sector, a Google Play Store scraper script was utilized. The script had been configured to retrieve a list of applications according to specific parameters, such as the application type and country code. This enabled the collection of information such as the app name, ID, country, number of downloads, ratings, and other relevant data. The researchers ensured the inclusion of at least one application from each African region, considering the earlier mentioned three-tier classification system. By employing this methodology, a diverse range of data on financial applications throughout the African continent was successfully collected.

Category	Description
Top Tier	Application with more than 50,000 Downloads or used in more than one country. When the latter is true, the number of downloads is also more than 50, 000. These applications pose the highest risk due to the user base
Middle Tier	Applications with downloads between 5000 and 49, 999
Low Tier	Applications with downloads between 500 and 4, 999

Table 1: Description of the application classification groups

3.1 Framework

To ensure a methodical and thorough comparison of applications, the team sought a framework that would provide a comprehensive set of criteria to be evaluated. The chosen framework needed to allow for the creation of a checklist that could be used to test the applications. The (MASVS)[4] is one such framework specifically designed to be a guide and checklist for evaluating the security of mobile applications. Developed by the OWASP Mobile Security Project, it is a standard set of security requirements that covers a broad range of categories.

3.1.1 Architecture, Design, and Threat Modeling Requirements. It mainly focuses on building an application with a secure architecture that is resilient to common attacks and vulnerabilities.

3.1.2 Data Storage and Privacy Requirements. Ensure that mobile applications handle user data securely. It is concerned about secure storage and transfer of user data, PII, credentials, etc and it suggests encrypting data at rest and in transit, not storing sensitive data on the device, avoiding hard-coding sensitive data, and not logging sensitive data.

3.1.3 Cryptography Requirements. Ensures that sensitive data is properly protected while in transit and at rest by using strong cryptographic algorithms for encryption, using secure random number generators, using secure protocols, such as HTTPS or SSL/TLS, for communication, and using secure key storage management practices.

3.1.4 Network Communication. Requires that applications implement secure network communication to protect against eavesdropping, tampering, and other types of network-based attacks.

3.1.5 Platform Interaction Requirements. Outlines a set of guidelines and best practices for how mobile apps should interact with various platforms, including operating systems, mobile devices, and other software. These requirements are designed to help developers ensure that their apps are secure and can be used safely by users.

3.1.6 Code Quality and Build Setting Requirements. Focuses on the overall level of excellence and maintainability of the code base in a

software project and the specific configurations and settings that must be in place in order to build and deploy the software.

3.1.7 Resilience Requirements. Refers to the measures taken to ensure that an application can continue to function in the face of unexpected events or conditions, such as network outages or device failures

3.2 Tools Used

The analysis made use of the following non-exhaustive list of tools; Burp Suite, MobSf, Jdx-gui, Objection, Genymotion, apktool, ADB

3.3 Scope

A thorough security assessment of an application requires tests on both the client and server-side components. However, for the purpose of this paper, the authors have chosen to focus solely on unauthenticated client-side testing, following the black-box testing approach. As a result, the scope of this project is limited, and the team was unable to perform comprehensive testing of all application flows since they lacked access to test user credentials for each application.
The table below summarizes the number of sections and sub-items that were tested out of the total as listed under the MASVS v2 framework. While the testing may not be comprehensive, the researchers have made every effort to provide a comprehensive overview of the security risks and vulnerabilities discovered during the testing process. All other tests required privileges access to the application.

MASVS Section	Tested	Sub-sections tested
Architecture, Design and Threat Modeling Requirements	yes	5 of 12
Data Storage and Privacy Requirements	yes	7 of 15
Cryptography Requirements	yes	3 of 6
Network Communication Requirements	yes	4 of 6
Platform Interactions	yes	1 of 11
Code Quality and Build Settings	yes	9 of 9
Resilience Requirements	yes	5 of 13

Table 2: Number of sub-items tested in each category of the MASVS Framework.

4 ANALYSIS

4.1 Applications Chosen

The analysis looked at a total of 18 applications downloaded from the Play Store. The versions of the apps were the latest as per the time of testing. Only applications in e-banking and mobile wallets were selected in this sample size. These banks operate in east, south, and west Africa. The sample size included financial services of various sizes including national, regional, and small SACCOs (Community banks and Savings and Credit Cooperative Organisations or Society). These applications were further categorized according to fig 1 (Each category is described in table 1). For confidential purposes, the paper will not identify the banking applications but a full report will be provided to the app owners.

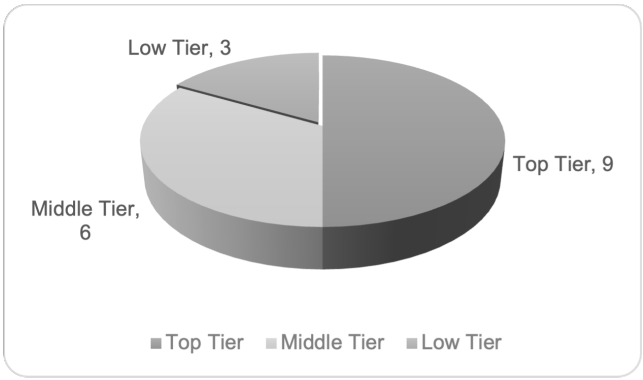

Figure 1: Number of applications per risk category

4.2 Detailed Security Analysis

In this section, an analysis of each requirement and how many apps passed the test has been done. We have further classified the issues into four tiers based on how many apps did not pass the test criteria as the MASVS is designed to be used as a security baseline. All secure banking apps should meet most if not all baselines. In some occasions, this rating has been overridden to escalate an important issue such as transmitting information in plain text.

- **Critical:** number of apps that failed this test were more 70%
- **high:** number of apps that failed this test were below 70%
- **Medium:** number of apps that failed this test were below 55%
- **Low:** less than 30% or no apps failed this test

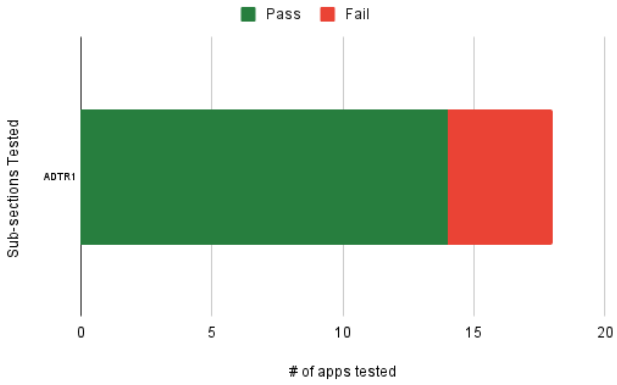

Figure 2: Number of applications that passed/failed the Architecture, Design and Threat Modeling Requirements.

4.2.1 ***Architecture, Design and Threat Modeling Requirements(ADTR).***
ADTR1: Automated security and functional app updates

Severity: Critical
A well-maintained software should contain occasional updates to address existing software bugs, security vulnerabilities in native code or third-party libraries, and functional updates. In the case of security vulnerabilities, an app should be forced to upgrade once a newer version is available on Play/App Store or otherwise discontinue its use. In our analysis, 81% of the applications did not enforce in-app updates despite its ease of implementation. This leads to the use of outdated APIs or app crashes and in-turn frustrations to the users. Automated security and functional app updates.

4.2.2 ***Data Storage and Privacy(DSP).*** Figure 3 summarizes the tests in this section.

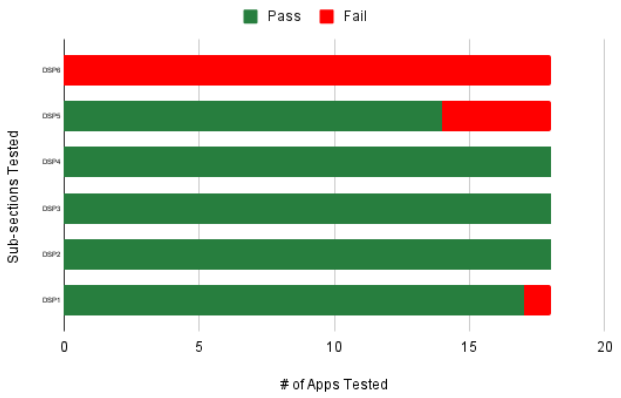

Figure 3: Number of applications that passed/failed the Data Storage and Privacy requirements.

DSP1: Log Management
Severity: Low
Sometimes in a software development project, developers would use logs as a way of monitoring if the application flows are correct. In some cases, sensitive information is coded into the logs as part of the debugging process and later forgotten. A secure application should not have this sensitive data written to application logs. Our limited analysis (the tests only looked at the interfaces accessible before credentials were required) only found one of the apps that logged network requests in clear.

DSP2: Protect the keyboard cache:
Severity: Low
Text input fields that process sensitive data should have the keyboard cache disabled. This measure protects usernames, passwords, and other frequently typed sensitive data from being accessed by other malicious apps running on the phone which may continuously capture the screen and expose data. 100% of the applications tested passed this test.

DSP3: Exposure of sensitive data, such as passwords or pins, through the user interface
Severity: Low
A user needs to enter data at some point be it username, passwords, and in our case, credit card information and transaction information. App developers need to mask this sensitive information in order to protect users from vulnerabilities such as shoulder surfing. 100% of the applications tested passed this test.

DSP4: Screenshots with sensitive data
Severity: Critical
It highlights the risk of exposing passwords, API keys, and personal data through screenshots taken by apps or the operating system. To mitigate this risk, it is advised to avoid storing sensitive information in screenshot-captured views and implement security measures like obscuring data or disabling screenshot functionality. Apps should also remove sensitive data from views when in the background. All tested applications allowed the capture of screenshots containing sensitive information, potentially enabling unauthorized access. For example, capturing a screenshot of a banking application may reveal information about the user's account, credit, transactions, and so on.

DSP5: Local database backups
Severity: Low
Most apps will usually keep a local database that can be used to keep a copy of API keys, usernames, and bank accounts and reduce the number of connections to the backend. In this case, encryption should be applied to protect the databases. In addition, backups of these databases should be disabled to prevent one from getting a copy of the database in its encrypted or unencrypted formats. Three apps failed this test by allowing data storage with the "allowBackup" setting enabled in the manifest file. Financial apps must disable this feature before going live due to the sensitive nature of banking information.

DSP6: Data exposure through InterProcess Mechanisms
Severity: Low
The goal of this check is to ensure that sensitive data stored on a device is not accessible to unauthorized third-party apps or malicious attackers. This is achieved by analyzing the IPC mechanisms (content providers) used to access and share data between different processes on a mobile device. The implications are that other apps on the device and malicious users can take advantage of the content providers to access sensitive and unauthorized data from the application. A static analysis of the code revealed that all (100%) of the tested applications did not export their content providers for all apps to read.

4.2.3 ***Cryptography Requirements (CR).*** Only 3 out of 6 tests were completed under this requirement as summarized in figure 4.

CR1: Symmetric cryptography with hardcoded strings
Severity: Low
This guide helps developers ensure secure and reliable cryptographic operations in mobile apps by focusing on symmetric key usage, generation, and storage. It aims to protect user data and application integrity. However, one application failed in this aspect due to its use of symmetric algorithms and hardcoded keys.

CR2: Cryptographic protocols
Severity: High
78% of the applications failed to use secure cryptographic protocols, compromising their security and integrity. One common cause is the extensive use of outdated cryptographic protocols like MD5, SHA-1, and DES. Outdated protocols may also be used to maintain

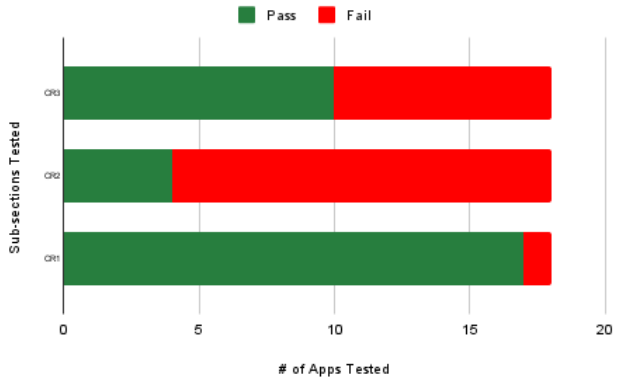

Figure 4: Number of applications that passed/failed the cryptography requirement

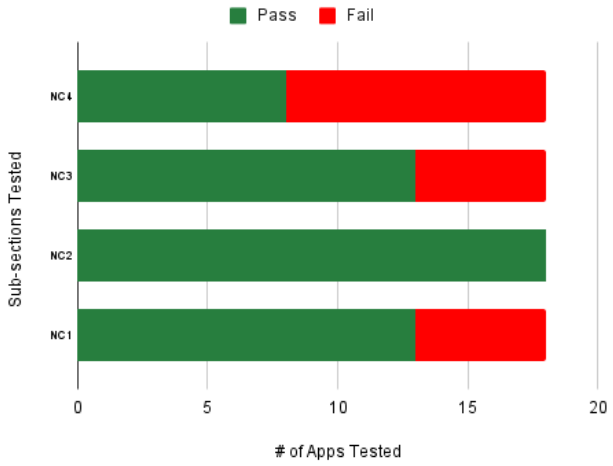

Figure 5: Number of applications that passed/failed the Network Communication requirements.

compatibility with older devices and software that lack support for newer cryptographic protocols. While this may seem like a practical solution, it ultimately compromises the overall security of the application.

CR3: Random number generation
Severity: High
True random number generation is essential for secure cryptographic systems. Unfortunately, eight (8) applications were using insecure random number generators like the java.util.random library, which is known to be insecure. This leads to predictability in cryptographic keys and algorithms, enabling attackers to guess the next generated value. This allows them to impersonate users or access sensitive information. To ensure security, developers should utilize secure random number generators that provide true randomness, such as java.security.SecureRandom.

4.2.4 ***Network Communication (NC).*** Only 4 of 11 tests were completed under this requirement as summarized in figure 5.

NC1: Protect data on transit
Severity: Low
Testing data encryption on the network involves checking the proper use of encryption protocols like TLS and SSL, as well as encryption keys and certificates. The objective is to secure data on transit.

Five out of the 18 tested applications failed by allowing cleartext traffic to their endpoints. Two of these apps allowed a mix of encrypted and cleartext traffic, while one app permitted cleartext traffic to and from a specific subdomain. One of these apps even exposed sensitive information like phone numbers and OTPs in cleartext.

NC2: Use of trusted CA certificates
Severity: Low
The endpoint authentication test verifies the correct identification and authentication of remote endpoints in a network using trusted CA-signed certificates. All tested applications were developed with a target SDK version >= 24, which should only accept trusted certificates by default from the SYSTEM certificates in Android phones. This default setting was also not overridden using the Network Security configuration file or by implementing trust anchors for user-supplied CA certificates in Android devices. Developers can comply with this check by ensuring the app is compiled targeting SDK version >= 24.

NC3: Restricting Trust, Identity Pinning
Severity: Medium
In the previous check, a check is done to ensure only SYSTEM CAs are trusted, placing a responsibility on the Android system to have trusted CAs. This check further builds on this to restrict the number of CAs that an app can trust and furthermore, a specific certificate on public key hardcode in the code such that only a specific key pair is trusted. This helps to prevent Man-in-the-Middle (MITM) attacks where an attacker presents a fake certificate to intercept and eavesdrop on a secure connection. 5 of the 18 applications failed this test because they did not implement certificate pinning.

NC4: Preventing supply chain attacks in communication
Severity: High
Android relies on a security provider to provide SSL/TLS- based connections. The problem with this kind of security provider (one example is OpenSSL), which comes with the device, is that it often has bugs and/or vulnerabilities. To avoid known vulnerabilities, developers need to make sure that the application will install a proper security provider. The app should only depend on up-to-date connectivity and security libraries. Applications based on the Android SDK should depend on GooglePlayServices for security provider updates. 10 of the 18 applications failed this test, thus the apps are at risk of using security libraries that are outdated and have known vulnerabilities.

4.2.5 ***Platform Interactions (PI).***

PI1: App Permissions
Severity: Low
This control ensures that mobile application only requests the necessary permissions to perform its intended functionality and that it does not overstep its bounds by requesting unnecessary or excessive permissions. 28% of the tested applications failed the test. As an example, one application allowed reads from the system's various log files. This allows it to discover general information about what the user is doing with their phone, potentially including personal or private information.

4.2.6 ***Code Quality and Build (CQB).*** Figure summarises tests in this section.

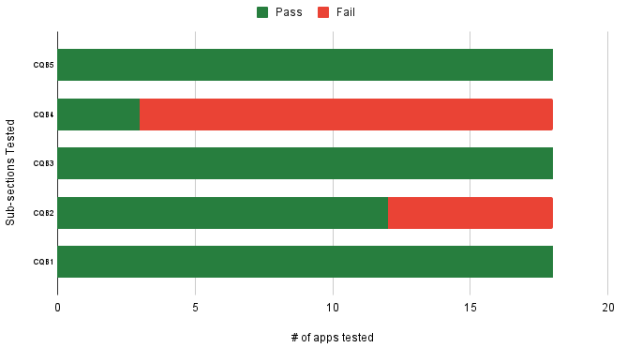

Figure 6: Number of applications that passed/failed the Code Quality and Build requirements.

CQB1: Debuggable Apps
Severity: Low This test checks if the app can be attached to a debugging tool. A debuggable app is vulnerable to attacks like code injection and memory manipulation, enabling attackers to access sensitive information and perform malicious activities. While the debuggable variable in the manifest file is useful for app development, it should be set to false before release and upload to the Play Store.

All 18 tested applications comply with the Play Store's requirements. However, app owners should be aware that some users download apps from unofficial sources that could have been repackaged and debuggable.

CQB2: Debugging Symbols
Severity: Low
The test verifies the removal of debugging symbols from native binaries [6]. Developers should be careful about including excessive information in compiled code. While certain metadata like line numbers and descriptive names can aid code comprehension, they can also assist reverse engineers in understanding the app's structure. Debugging symbols, containing sensitive information like function and variable names, are especially vulnerable to attackers aiming to exploit app vulnerabilities. All tested applications (100%) successfully passed this test.

CQB3: Debugging Code and Error Logging (Strict Mode)
Severity: Low
This test detects apps with strict mode enabled. Strict mode is a developer tool that enforces code rules and catches potential bugs during development. For example, it detects disk or network access on the main thread that can cause application freezing. Strict mode should not be used in production code as it degrades performance. All tested applications (100%) passed this test.

CQB4: Run-time protection against buffer overflows
Severity: Critical
This can be achieved by ensuring all native libraries the canary and PIC are both set to true. Disabling canaries can make the app more vulnerable to exploits such as buffer overflows and heap spraying. PIC makes it more difficult to reverse-engineer and attack an app. Opting to disable PIC and canary as a means to boost performance may come at the expense of compromising the security of your system. It's crucial to consider the trade-offs involved and implement alternative solutions that minimize the impact on security while still achieving the desired performance gains. 83% of the applications failed this test, and developers may need to recompile these applications and set canary and PIC to true.

CQB5: Exception Handling
Severity: Low
Testing exception handling ensures that the app handles exceptions securely without exposing sensitive information. The process checks error messages and stack traces for confidential data and verifies that the app transitions to a safe state without crashing. Due to limited access, the testing followed a black-box methodology, interacting with a limited number of views and functionalities, and reviewing code. Despite these limitations, all tested applications passed the exception-handling requirement, giving appropriate messages and avoiding crashes when unexpected input was provided.

4.2.7 ***Resilience (R).*** Figure 7 summarises tests in this section.

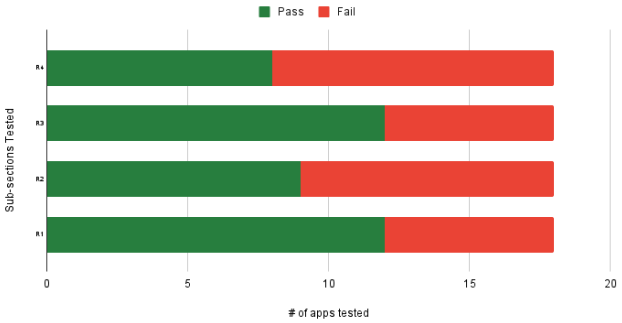

Figure 7: Number of applications that passed/failed the Resilience requirements.

R1: Root detection (Jailbroken device)
Severity: Medium

The test checks if the app can detect and respond to a jailbroken operating system. The expected response is to either warn the user or terminate the app. Unfortunately, 33% of the tested applications failed this test. For financial applications, it is recommended to restrict their usage on jailbroken devices, and it is advisable to enforce this control on the Play Store.

R2: Anti-Debugging
Severity: Medium
Debugging is a powerful method for analyzing app behavior at runtime, allowing reverse engineers to examine code, pause execution, inspect variables, and manipulate memory. Setting the debuggable flag to true enables attackers to assume app privileges and access its data by executing arbitrary code. 50% of the tested applications failed to detect an attached debugger.

R3: A financial app should not run on an emulator
Severity: Medium
Emulators are used by most hackers because they are controlled environments for hackers to learn application behavior. Developers need to put in controls to prevent an app from installing and running on an emulator. This research found that 33% of the apps could run on an emulator, allowing illegitimate users to run applications, and learn their behavior without the protections put in place when running the app on the phone.

R4: Code obfuscation
Severity: High
When hackers cannot run the application with a valid account, they opt to read the code and infer the functionalities of the app. Obfuscation is a measure to change the code in ways that make it harder for hackers to read the code once reverse-engineered. 55% of the apps tested did not have any obfuscation in place. However, none of the applications implemented any other methods of obfuscation like anti-debug obfuscation.

5 PROGRAMMER SURVEY REPORT

In many cases, the researchers had some hypotheses for the observed patterns that were not verified. A questionnaire inspired by the technical work was developed and presented to application developers in the form of an online survey. These hypotheses were presented in the form of a question and multiple-choice answers. For example: The researchers thought that some of the reasons the developers were not implementing security in their applications were time, performance and compatibility issues, lack of knowledge, and costs. Thus, they came up with the following multiple-choice question.

- What challenges have you encountered when implementing security measures in your application? (Check all that apply):
  - ☐ Lack of expertise or knowledge.
  - ☐ Cost of security solutions
  - ☐ Difficulty in integrating security solutions with existing app architecture
  - ☐ App performance issues
  - ☐ User experience issues
  - ☐ Tight Deadlines to deliver the applications

The researchers also wanted to find more generic responses separate from the ones presented in the above multi-choice questions above. In such cases, the question was presented as open-ended questions before the multiple-choice was.
Further examples of the questions asked are listed below:

- What are some of the challenges that make it difficult to think about security measures and implement them while developing an application?
- In your opinion, what can be done to improve the implementation of security measures in applications?
- How important do you think it is to educate developers on security best practices, and what methods do you think are most effective for doing so?
- As a developer, at what point do you think of security and start implementing security measures in your application?
- Are you aware that there are minimum security standards required for an application before deploying for use? (Yes/No)
- Have you ever sacrificed security for the sake of improving the performance of your application? (Yes/No)

To gather more informed responses, we specifically targeted developers with 1 year of experience and above, students enrolled in master's programs. Additionally, the survey was shared within the researcher's circles and on social media pages.
The data collected were manually analyzed by examining each response individually and taking note of key nouns and adjectives. For instance, when participants were asked about the challenges they faced while considering and implementing security measures during application development, keywords identified from the responses included time, expertise, awareness, and complexity. Additionally, analysis results included percentages representing the distribution of responses for the predefined answer options on the list. Below are some of the survey questions that were posed in the survey.

5.1 Result Analysis

Within a period of 2 months, a total of 46 responses were received. Although 26% of respondents were unaware of the necessary security requirements, 74% recognized the existence of minimum security requirements for an application prior to its public deployment. When asked about their approach to security implementation, most developers stated that they consider security during requirements gathering and system design, as well as just before production deployment (develop and test later approach). A majority (72%) claimed to implement security measures during the application development phase, while 26% implemented security measures before deployment and 2% did so after initial public deployment.
The developers identified several challenges they encountered when attempting to integrate security measures into their applications. These difficulties made them to disregard security altogether, employ incomplete measures or shortcuts, or utilize non-standard approaches that render the application vulnerable and unable to meet standard requirements. Some of the challenges they cited include insufficient knowledge and expertise in security implementation, time constraints as they focused on the delivery of the application's functional requirements before deadlines, security was regarded as secondary to functionality by their supervisors, insufficient awareness of security's importance and associated risks, the complexity of security implementation, a lack of community support for new

security mechanisms resulting in outdated mechanisms being used, and compatibility issues. 78% of respondents said they avoided implementing security due to a lack of expertise and knowledge about security. 63% thought that implementing security measures in applications is too complex or/and time-consuming.
The participants also proposed multiple measures to address these difficulties, including organizing workshops and seminars to educate developers on security, providing training sessions for them to learn about the most up-to-date security mechanisms, setting aside resources and time to incorporate security into the application before releasing it, increasing awareness among developers and their supervisors about the significance of security and associated risks, establishing and enforcing security policies and standards for development teams, and adopting a development pipeline that intentionally incorporates security and follows best practices.

5.2 Weaknesses of the Survey

The researchers however noted some weaknesses in the survey used. For example, the study could do better at profiling the developers e.g. information on fine-grained years of experience. Our survey only required a respondent to proceed with the survey if they had at least one year of experience. Also, the survey did not go into detail about the programming languages that the developers were experts in and what region the developers worked from. The questionnaire was also generic to software development practices in general.

5.3 Recommendations

Our suggestion is for supervisors to prioritize the significance of security and acknowledge it as a form of progress, even though it may not be as tangible as functional implementation. We propose that they establish security-oriented sprints that force developers and their managers to solely concentrate on integrating security mechanisms into their applications, covering all potential adversarial scenarios. Additionally, we advise organizations to seek a third-party evaluation and recommendation of their applications' security posture in the case where an internal security testing team does not exist and is too expensive to obtain.

6 CONCLUSION

In this report, our objective was to test financial applications used by users in Africa using OWASP's MASVS v2 and make a report of the security practices and measures in place to protect their users.
Our findings indicate that it is imperative for organizations to ensure their applications meet the minimum security requirements. By doing so, they can instill confidence in their users, who are more likely to trust and continue using an application that guarantees security and reliability.
We believe that ensuring the security of financial applications is the sole responsibility of the application owners. They must implement robust processes and practices that encourage software engineers to integrate security reinforcement into the software development lifecycle. While this may be costly, product owners would benefit much more if security was interpreted as programming techniques and code like in the MASVS. This approach will yield better results and completeness than simply relying on general web application vulnerabilities such as the OWASP Top 10.

In Summary, application owners should watch out to put in place automated security and functional app updates, put controls for screenshots with sensitive data, use secure Cryptographic protocols, use trully random number generators, Preventing supply chain attacks in communication by relaying on Google Play as a library provider, package apps for all architectures with run-time protection against buffer overflows in place, and employ multiple level of code obfuscation as these were the most ignored checks in our sample size of apps.

REFERENCES

[1] https://www.worldbank.org/en/topic/financialinclusion/overview
[2] Mobile Privacy: What Do Your Apps Know About You?", Symantec-enterprise-blogs.security.com https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/mobile-privacy-apps
[3] Application sandbox Android Open Source Project. https://source.android.com/docs/security/app-sandbox
[4] OWASP MASVS (Mobile Application Security Verification Standard) https://mas.owasp.org/MASVS/
[5] Android API Levels https://apilevels.com/
[6] Android ABIs https://developer.android.com/ndk/guides/abis
[7] android:debuggable https://developer.android.com/topic/security/risks/android-debuggable
[8] Binary Protection Mechanisms https://mas.owasp.org/MASTG/General/0x04h-Testing-Code-Quality/#dynamic-analysis-security-testing-considerations_1
[9] Financial Inclusion https://www.worldbank.org/en/topic/financialinclusion/overview#2
[10] Fintech in Africa: The end of the beginning August 2022 McKinsey & Company

7 APPENDIX

7.1 Disabling keyboard cache

This is achieved by:

- Adding the inputType attribute to your EditText view in XML code and setting its value to textNoSuggestions. This will disable suggestions for that specific view

```
<EditText
android:id="@+id/myEditText"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:inputType="textNoSuggestions" \/>
```

- Disabling text suggestions for specific input fields dynamically at runtime, you can do this by setting the inputType attribute of the EditText view in your code.

```
<EditText myEditText = findViewById
(R.id.myEditText);
myEditText.setInputType(InputType.
TYPE_CLASS_TEXT |
 InputType.TYPE_TEXT_FLAG_NO_SUGGESTIONS);
```

- Using a custom keyboard, you can disable text suggestions, change the appearance and behavior of the keyboard, and add additional features such as emojis or custom shortcuts.