An android applications vulnerability analysis using MobSF

Shujahat Ali Khan[1], Muhammad Adnan[2], Ahtasham Ali[1], Ali Raza[3], Asim Ali[2], Syed Zohaib Hassan Naqvi[2], and Tehseen Hussain[4]

[1]College of Engineering & IT, Charles Darwin University, Darwin 0800, Australia
[2]Department of Electronics Engineering, University of Engineering and Technology Taxila Pakistan.
[3]Department of Computer Engineering, National University of Technology (NUTECH) Pakistan.
[4]Department of Electrical Engineering, National University of Modern Languages Pakistan.
Email:adnanaslam90901,malikasim15329@gmaill.com[2], zohaib.naqvi@uettaxila.edu.pk[2], ali.raza@nutech.edu.pk[3]
Shujahatkhan345, tehseen.hussain@gmail.nyml.edu.pk[4],Ahtashamali@gmail.com[1]

2024 International Conference on Engineering &amp; Computing Technologies (ICECT) | 979-8-3503-4971-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICECT61618.2024.10581312

***Abstract*—The reality that so many third-party applications for Android have security flaws that allow thieves access is one of the main drawbacks of using them. The primary cause of the compromise is that, when implementing the source code, the developer focused more on the application's functionality than on its confidentiality, integrity, and authentication. As a result, safeguarding sensitive data used by users of mobile applications became extremely difficult due to privacy breaches. This study looks at how to analyze Android code to identify the fundamental causes of vulnerabilities that are discovered in it. A Mobile Security Platform (MobSF) was utilized to perform comprehensive and dynamic evaluations of Android applications. This required looking at the source code—the code written by developers—as well as the binary code, which is the code that can be executed by devices. It also involved using a method called common weakness enumeration to find known vulnerabilities. Identifying vulnerabilities in the source code itself and any additional security holes in the application was the aim. Typically, the adversary inserts malicious strings and backdoors into the applications to gain access and steal sensitive data. MobSF is an Android app analysis tool with a graphical user interface. MobSF is an Android app analysis tool with a graphical user interface. Although it can analyze apps both statically (by analyzing the code) and dynamically (by running the app), our primary goal is to detect potential security flaws or weaknesses in the Java code that developers write.**

***Index Terms*—Analysis in Motion Classification breaches pertaining to privacy Applications for Mobile Machine learning**

I. INTRODUCTION

Technology advances at a rapid pace in the modern digital world, right along with everything else. Thanks to the incredible Android operating system, phones have evolved from cumbersome desktop companions to potent, pocket-sized marvels. The reason Android has become so popular is that smartphones are no longer just phones. These are incredibly intelligent devices with incredible sensors, blazingly quick processing, and limitless potential. With Google's help, Android's open-source design has contributed to its rapid rise in the smartphone industry. But because of its widespread app ecosystem, cybersecurity researchers are also drawn to it to find any potential weaknesses. Exploiting these vulnerabilities in Android applications can be difficult. For instance, a research team [1] used a particular technique known as "static analysis" on multiple apps. Through this investigation, a data flow analysis technique was found that, if not addressed, could potentially lead to rights-based attacks, and significantly reduce security risks between applications. Viruses, worms, and trojans [2] are the three most well-known categories of malicious software, or malware. Even though these terms are frequently used synonymously, they differ significantly from one another. Every one of them has a different method for entering systems and having varied effects on them. As discussed in more detail in cited paper[2], once malware has successfully compromised a system, attackers can go after other nefarious objectives. [2]

- An intruder can easily view your past messages
- An intruder can effortlessly track your location
- An attacker can steal your passwords
- An attacker or intruder can steal your bank account information
- An attacker can send messages to others pretending to be you, without your awareness or consent
- An attacker can make calls without your knowledge, acting as a "man in the middle."
- An attacker can use up your data plan without your knowledge
- An attacker can reduce your device's battery life
- An attacker can record your calls and sell them.

A. Android Application Package

Android apps and other programs designed for middleware, interactive games, and mobile app development and distribution use a special format called the Android Application Package (APK). It's like a neatly packed suitcase containing everything an app needs to run on Android devices. This suitcase, in the form of a Java file, holds the app's code, its ID certificate, a detailed list of its contents (called a Manifest file), and various resources like images and sounds.

978-1-6654-6694-3/22/$31.00 ©2022 IEEE

B. Malware Identification

Multiple techniques are employed to detect malware within Android applications. One approach is signature-based detection, where an app's unique signature is compared to a database of known malware signatures. However, this method can sometimes miss new or unknown malware strains. To address this, behavior-based detection comes into play, where an app's behavior during execution is monitored and compared to profiles of both malicious and normal behavior. Android security techniques for malware detection can be broadly categorized into three types: static analysis, dynamic analysis, and hybrid analysis. [3] These are visually depicted in Figure 1. Detection programs often rely on various analytical methods, such as feature analysis, system call analysis, network traffic analysis, and more. [1]

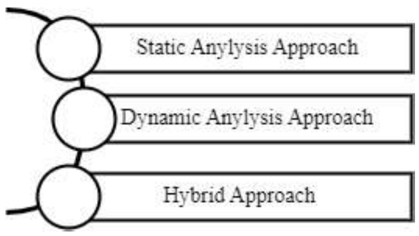

Fig. 1. Android Application Package Analysis Approach

We have selected MobSF as our preferred tool for this study to perform comprehensive and real-time analyses of Android source code. Android application packages are what we'll be feeding it to examine.

C. Mobile Security Framework

MobSF, a free open-source tool for your PC, helps you scan mobile apps for security weaknesses. It can perform comprehensive, effective, and malware-free analyses on mobile operating systems like Windows, iOS, and Android. OWASP MSTG advises using it for mobile application security statistical analysis [4]. With support for duplication (APK, IPA, and APPX) and ZIP source codes, MobSF is a useful tool for securely and quickly evaluating Android, iOS, and Windows Mobile applications. It can also perform impressive application testing while Android applications are running, jeopardising the capabilities of a standard Web API security scanner. It is expected to be integrated into your development pipeline or CI/CD system. There is a picture of the web management user interface (UI), which includes the site's design, an embedded emulator, a dashboard showing measurable results, and an API that allows tests to be started automatically. Your location information is unrelated to the cloud [5].

1) Running MobSF: To initiate MobSF on your local server, kindly proceed to http://localhost:8000/ in your web browser to gain access to MobSF's graphical user interface (GUI).The manage.py file in Python can be edited by users to change the port address and run MobSF on a specified port.

2) Android Log Analysis: An easy way to gather and review troubleshooting data is through the Android logging system. It collects logs from different applications and system parts and stores them in memory for examination later. [6]The logcat command can be used to view and filter these logs. We launched a malicious application after connecting to the emulator via ADB Shell. Logs started to appear as soon as the app launched, giving important information about how it behaved.

3) Web Interface of MobSF: You will see a range of symbols within the MobSF dashboard that relate to data visualisation, app iconography, app name, app size details, package name, and the disclosure of encryption algorithms like MD5 and SHA1. These selections serve a pivotal role in identifying potentially harmful attributes within Android applications.

4) Dynamic Analysis Approach: Running an app on a real device or an emulator to observe its behavior in action is what dynamic analysis is all about. Various data is collected during this process, including how the app accesses sensitive information, network traffic patterns, and any vulnerabilities it may have. While the app is running, this analysis examines file read-write operations, open network connections, and incoming/outgoing network traffic [7]. Dynamic analysis-based Android malware detection methods concentrate on features that are actively captured during app execution or interaction. V.M. Afonso et al. created a technique that takes advantage of this approach. It monitors factors such as device configuration and API calls before employing machine learning algorithms to determine whether an app is benign or malicious.

- Approach of Static Analysis
- Approach of Dynamic Analysis
- Hybrid Approach

MADAM is a detection algorithm that acts like a trained detective, picking out malware from the crowd of app source code. It does this by using a similarity matrix, which is a detailed map of what malware usually looks like, to spot suspicious patterns [8]. MADAM is designed to work on the host machine, meaning it analyzes apps right on your device. It keeps a close eye on app behavior at multiple levels—package, user, and application—to catch any malicious activity that might be lurking.

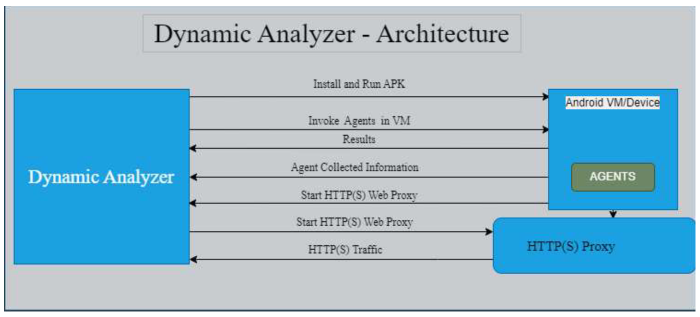

Fig. 2. Dynamic Analyzer Architecture

5) Static Analysis Approach: An approach that is frequently used to find problems in Android or computer systems without running the code is static code analysis. It entails a careful review of the code, akin to checking a blueprint for possible errors prior to building. Code is transformed into an abstract syntax tree (AST) by specialized tools, which expose the code's underlying logic and possible trouble spots. This transformation makes it possible to examine the code's structure in detail. Before conducting product testing, static code analysis is essential for spotting security holes and performance problems in both new and old systems. Finding source code flaws that could result in bugs or security breaches is the main objective. Because of their effectiveness and thoroughness, automated tools are typically chosen over manual analysis. [9] Static code analysis examines the application from the inside out, working backward from the finished product to its underlying code. This platform-agnostic approach avoids actual application execution in favor of a thorough examination of the source code and related components. Static analysis is especially useful during the development phase, allowing for the early detection and correction of vulnerabilities before they become embedded in the final product. Comprehensive security checkpoints should be implemented on a regular basis, such as daily, monthly, or upon each code submission or release. Static analysis provides the following benefits:

- Time expenditures associated with manual code reviews are significantly reduced through the utilization of automated tools, as they swiftly pinpoint code-based errors.
- The identification of bugs and potential errors that may elude unit or manual testing is facilitated by static code analysis.
- The customization and tailoring of project structures to align with specific requirements within Android applications are enabled through the definition of explicit project rules.
- Enhanced comprehension of Android applications is fostered through static code analysis.

During the static analysis process, the APK file is uploaded and then decompiled within the Mobile Security Framework's (MobSF) static analyzer engine. The source code is extracted from the APK through this process, and a variety of methods are applied to it to create a detailed report.

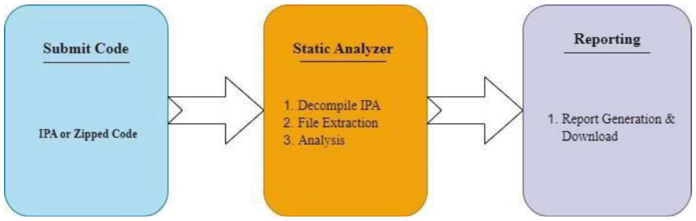

Fig. 3. Static Analysis Architecture

Malware Analysis is a process focused on unraveling the inner workings and intentions of suspicious files or URLs. Through this analysis, detection and assessment of potential threats are facilitated. A key advantage of malware analysis lies in its ability to aid incident responders and security analysts in various ways, including:

TABLE I
STANDARD SCORING SYSTEM OF VULNERABILITY

Score	Vulnerability Assessment
0.0	No Risk Present
0.1 - 3.9	Low Level of Risk value
4.0 - 6.9	Moderate Risk Level
7.0 - 8.9	High Risk Level
9.0 - 10.0	Extremely Critical Risk

- Sudden critical incidents are identified and responded to effectively.
- The accuracy and reliability of indicators of compromise (IOCs) warnings and alerts are enhanced.
- Preparedness for proactive threat hunting activities is bolstered.

6) Common Vulnerability Scoring System: Within the Common Vulnerability Scoring System (CVSS), code is assigned a vulnerability rating, referred to as a CVSS score. This system offers a thorough evaluation of security weaknesses, utilizing a numerical scale ranging from 0.0 to 10.0. Higher scores on this scale indicate more significant vulnerabilities. CVSS scores are assigned following a rigorous static analysis of the Android application. This analysis involves examining the application through various methods, including binary analysis and code analysis. Static analysis serves as a valuable tool for system validation, capable of detecting numerous issues within the code prior to its execution.

Among the potential issues that can be identified through static analysis are:

- Risks to Security
- Examples of Disruption and Collision Risk
- Disregarding Code Style Guidelines
- Issues of Homicide
- Inactive or Unutilized Code

D. Pegasus Malware

Pegasus Malware, a powerful form of spyware, allows attackers to infiltrate and hijack Android and iOS mobile devices. Pegasus, once implanted, can steal sensitive information from the infected device, such as text messages, instant messages, keystrokes, multimedia files, and data from targeted apps such as Instagram and Facebook. This advanced spyware not only extracts images from the compromised device's camera but also records videos and conversations with ease. Pegasus, created by the Israeli cybersecurity firm NSO Group in 2010, has been actively used in covert operations since mid-2016. The capabilities of Pegasus make it a coveted tool for hostile actors seeking to gather information from influential figures, such as leaders in public and private domains. This includes both readily available public data and more sensitive, privately held information.

E. *Functionality of the Pegasus Malware*

Pegasus attacks begin with a basic criminal technique intended to steal sensitive data: the attacker locates the victim and sends the URL of a website by text message, social media, email, or other means of communication. When an iOS user clicks on the link, the malware surreptitiously starts a three-day series of actions on the target device, which includes breaking into a remote jail and installing spyware. The only way to know that something happened is when the user closes the browser after clicking the link. There aren't any obvious alterations or the introduction of new procedures. Following its installation, Pegasus establishes a connection with the operator's command-and-control server infrastructure, enabling it to both receive and execute instructions issued by the operator [10]. Pegasus malware is comprised of nefarious code, procedures, and programs meticulously designed to surveil user behavior on the compromised device. This includes extensive data collection and transmission capabilities. Calls, emails, texts, and logs from a variety of apps, such as Facebook, FaceTime, Gmail, WhatsApp, Tango, Viber, Skype, and banking software, can be intercepted and forwarded by the malware [11]. Spyware uses the device's built-in applications to obtain data from infected devices, saving the installation of malicious copies of these applications [11].It is not necessary to use zero-day vulnerabilities to install Pegasus on Android in order to uninstall recognised devices and introduce malware. Rather, the virus uses a well-known and robust participatory strategy called frameproof. All attack sequences are rendered ineffective in the event that a zero-day attempt to jailbreak an iOS smartphone is unsuccessful. However, if the initial effort to disable the device proves unsuccessful, efforts are being made by hackers to develop an Android variant of Pegasus that requests permission to access and examine data.

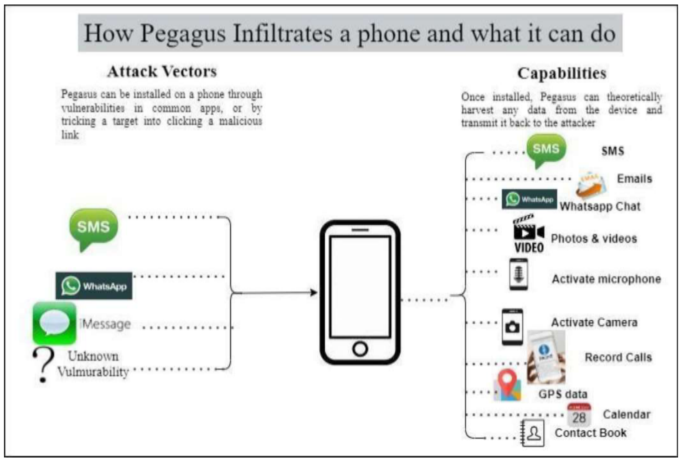
How Pegagus Infiltrates a phone and what it can do
Attack Vectors
Pegasus can be installed on a phone through vulnerabilities in common apps, or by tricking a target into clicking a malicious link
SMS
WhatsApp
Message
Unknown Vulmurability
Capabilities
Once installed, Pegasus can theoretically harvest any data from the device and transmit it back to the attacker
SMS
Emails
Whatsapp Chat
VIDEO
Photos & videos
Activate microphone
Activate Camera
Record Calls
GPS data
28
Calendar
Contact Book

Fig. 4. How Pegasus Malware Spreads?

II. LITERATURE REVIEW

Over the last year, malware detection on mobile phones has emerged as a prominent research area, particularly in Android devices. Two approaches have been introduced to address the issue of malware and its increasing prevalence. Static Analysis and Dynamic Analysis are two methods used to address this challenge [12]. Here is a list of the tools available for detecting mobile malware. The influence of static program analysis is used to identify Android malware. Malware detection entails disassembling using a static analyzer. This disassembly occurs in the absence of actual execution. During the de-compilation process, the device is protected and uninfected, allowing the analyzer to achieve high code coverage. The static analyzer examines the entire source code to obtain comprehensive code coverage. An instrument for the dynamic analysis of Android apps is called DroidBox [13]. The framework, called DroidBox 4.1.1, is used to evaluate Android applications automatically. All network communications, information exfiltration, the number of SMS messages sent to a specific address, and other activities related to Android applications are tracked. An altered version of the 4.1.1 rc6 Android emulator is used to complete these tasks. Framework operates on two levels: one within the guest operating system (Android emulator) that captures the ongoing activity of the Android application and then transmits the DroidBox records to the host machine via ADB; and one within the host machine that transmits the DroidBox records to the host machine via ADB. The host machine's second level decodes the ADB logs to extract the DroidBox log [14].It should be noted that the aforementioned release has only been tested on Mac and Linux operating systems. In the absence of the Android SDK, it is critical to download it and ensure that the necessary libraries are available: Matplotlib and Pylab, instrumental in rendering visual representations of the analytical outcomes presented in Table-II, will facilitate the visualization process. As cited in [15], the ANDRUBIS utility integrates both dynamic and static analyses on Dalvik virtual machines and system-level components. It employs diverse stimulation strategies to enhance code coverage.With the help of ANDRUBIS, we were able to compile a dataset of 1,000,000 android applications, 40% of which were malicious. In contrast to ANDRUBIS, DroidScope stands as an Android application analysis tool that upholds a comprehensive record of malware assessments through virtualization. DroidScope distinguishes itself from other desktop tools for analyzing malware by seamlessly and simultaneously reconstructing both OS-level and Java-level semantics [16]. The DroidScope tool exports a set of three-tiered APIs that mirror the three levels of an Android device: hardware, OS, and Virtual Machine, including Dalvik.Existing analysis systems with the capability to integrate dynamic and static analyses are limited, with none adept at adaptively monitoring actions within the Dalvik VM and beyond, in native libraries [17]. Moreover, many such systems are either unavailable for research purposes or have not undergone proper updates. This study aims to bridge this gap by leveraging MobSF. Another noteworthy tool, Mobile-Sandbox, employs a distinctive approach by integrating static and dynamic analysis methodologies. Native API calls can be effectively traced, and access is granted to individuals through a web interface [18]. In static analysis, fundamental information is extracted by scrutinizing the various modules of

TABLE II
ANALYSIS COMPARISON ANDROID APPLICATIONS TOOLS

Analysis Tool	Static Analysis Capability	Dynamic Analysis Capability	Network Analysis Feature	Malware Analysis Feature	Binary Analysis Feature	Manifest Analysis Feature	Code Analysis Feature	CVE
MOBSF	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Copper Droid	Yes	Yes	No	Yes	Yes	No	Yes	No
Sandbox for mobile	Yes	Yes	Yes	✓	No	No	No	Yes
Droidscope	No	Yes	Yes	No	Yes	Yes		No
Andrubis	Yes	Yes	No	Yes	Yes	No	Yes	No
DroidBox	No	Yes	Yes	Yes	Yes	No	No	No

the application. Initial static analyses are conducted, followed by parsing the manifest file, and ultimately decompiling the program to facilitate the identification of malicious code. Dynamic analysis, on the other hand, involves running the program in an emulator and logging every operation. Activities within the Java Virtual Machine and operations within native libraries, which might be incorporated into the application, are covered by this [19]. Compared to other analysis tools, CopperDroid [20] is a VM-based dynamic analysis system that is capable of reconstructing behaviours displayed by Android applications, which may include malicious content. This is made possible by keeping an eye on system calls and, if required, their arguments. Most notably, the tool records interactions that are essential to the Android operating system, like remote procedure calls (RPC) and inter-process communications (IPC). With this one point of observation, CopperDroid is able to extract the main behavioural traits of Android apps without getting into the nitty-gritty of SMALI instructions or the invocation of Dalvik VM methods. This unique methodology extends the life of suggested analyses by improving resistance against internal structural changes in Android, such as switching from Dalvik VM runtime to ART.After being abstracted into high-level semantics, the series of low-level system calls produces informative behavioural profiles. These profiles work well for large-scale data analysis and are helpful to human analysts [21]. MobSF is considered the most efficient of these tools. This conclusion stems from its thorough code analysis, which exposes hidden ulnerabilities in applications. MobSF effectively detects vulnerabilities using the CVSS score, enabling their remediation to strengthen the system against possible network attacks. A, Ali and a partner demonstrated a workable approach for performing source code analysis on smartphone platforms in order to find security flaws using in-depth investigation. In addition, we were the first to develop the platform for Source Code Analysis of Mobile Applications [22]. Threat Modelling [TM] is a technique that we can use to determine associated risks that could be triggered and determine the potential for danger [23], as described by Tatam et al. Vulnerable Android applications are now a prime target for current threats. It is critical to protect sensitive data from internal and external threats [24] [25].

III. EXPERIMENT AND RESULTS

As shown in Table 2, a purposeful selection of five distinct Android applications from different categories was made during this investigation: Knock Down, ANZ, Facebook Messenger, Bang GOOD, and Instagram. Following this selection, these applications were subjected to static and dynamic analyses in order to find vulnerabilities related to their network and code. Following post-testing, we discovered a wide range of vulnerabilities, including misconfigurations, the introduction of private data into log files, incorrect certificate validation, the use of dangerous cryptographic algorithms, Common Weakness Enumeration (CWE) incidents, Common Vulnerability Scoring System (CVSS) evaluations, and hidden strings in the application code. We used a variety of techniques in our analytical approach, including binary, source code, and security analysis. Using these approaches, we were able to pinpoint a number of flaws and vulnerabilities that developers had left behind, such as insecure objects and functions, CWE instances, insufficient certificate authentication, needless permissions that apps requested, and hardcoded strings that were hidden. These hidden strings could be a ticking time bomb, operating similarly to ransomware and dormant inside the program until the right opportunity presents itself to infect system files. The code and security analysis results are shown in Table 3, which includes a variety of outputs including hash values, CVSS scores, Security Scores, Binary Analysis results, and Log Analysis findings. Specifically, MobSF generates an APK hash for integrity checking. Laying the foundation for a secure software development cycle (SSDLC), MobSF stands as our first line of defense. Its potent capabilities meticulously scan our applications, diligently filtering out false alarms while pinpointing critical security vulnerabilities at the earliest stages of development. This proactive approach ensures that inherent security is woven into the very fabric of our applications, safeguarding them from the very beginning.

MobSF meticulously generates detailed logs within its directories, which can be meticulously examined to uncover additional vulnerabilities. These logs provide a complete picture of network activity, including the number of established connections and detailed monitoring of both outgoing and incoming traffic. Figure 5 illustrates the Common Vulnerability Scores (CVSS) meticulously generated by MobSF for each application following a thorough binary analysis of code weaknesses. As shown in Table 1, Know Down and ANZ are

TABLE III
STATIC & DYNAMIC ANALYSIS OF APPLICATIONS USING MOBSF

	Knock Down	**ANZ**	**Facebook Messenger**	**Bang good**	**Instagram**
Type	Gaming Application	Banking Application	Social Media Application	Online Store Application	Social Media Application
Hash	06c53fc33d451ef39fb 6dbe727e61873	40b0d41196a96 12d2e63110998b 22338	c614a09c8c8235feb7e aacd0f8904c0	e31c4c229f16130f9 27d1828f7f6917f	579e1f8764c2167f55 b9001f60762395
CVSS	7.5	7.5	6.4	6.3	6.6
Security Rating	90	75	55	10	5
Certificate Status	Authenticated	Authenticated	Authenticated	Authenticated	Authenticated
App Permissions	External Storage Access, Additional Android Permissions	External Storage Location Access	Mobile Contacts Location Access Multi-Account Ease	Read from External Storage	Allows request for authentication tokens
APK Analysis	Anti-VM code fund	Anti-VM code found	Anti-Debug Code found	Anti-VM code found	Anti-VM code fund
Code Evaluation	Random Value Usage (CWE-330)	Sensitive Information Inserted in Log File (CWE-532)	Configured insecurely allow clear text traffic to the domains, Risky or broken Cryptographic Algorithm is used (CWE-327)	Can easily bypass certificate pinning, Obfuscation of malicious strings, Encryption of data Inputs without Doing Integrity Checking (CWE-649)	CWE: CWE-276 Incorrect Default Permissions, Found Dangerous Method / Function (CWE-749), Certificate Validation is Improper (CWE-295)
Domain Malware Check	Good	Good	Good	Good	Good
Binary Analysis	Not found any trace	Exposed to Buffer overflow attack	Exposed to Cross Site Scripting attack	Exposed to Buffer overflow attack, and Obfuscation	Open to Authentication Breach, Spoofing attack
Log Analysis	Unauthorized access to resources not found	Unauthorized access to resources not found	Unauthorized access to resources not found	Encrypted hidden strings found	Data has been sent to authorized locations

classified as high-risk applications, while Facebook

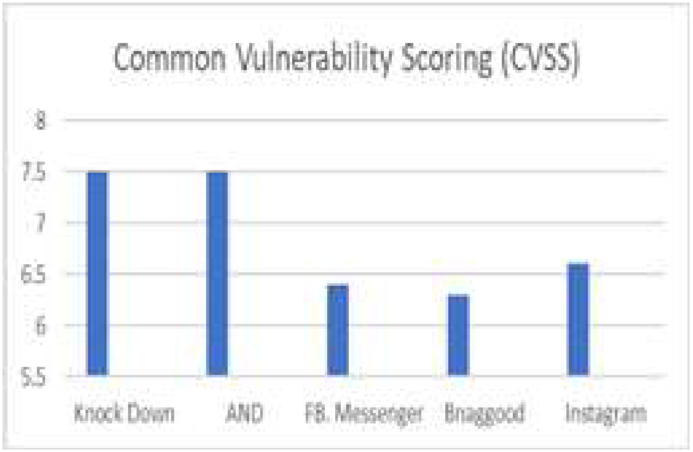

Fig. 5. Security Score Analysis

Messenger, Bang good, and Instagram are classified as medium-risk due to vulnerabilities discovered in their code. MobSF assesses security scores by scrutinizing an app's network security, including secure connections like SSL using encryption algorithms such as DES and AES for client-server communication. It examines encryption keys and certificates to ensure confidentiality and authentication. Additionally, MobSF applies Quark rules for malware analysis. The security score output indicates Instagram and Banggood as high risk, while ANZ, Facebook Messenger, and Knock Down are at low risk (Figure 6)

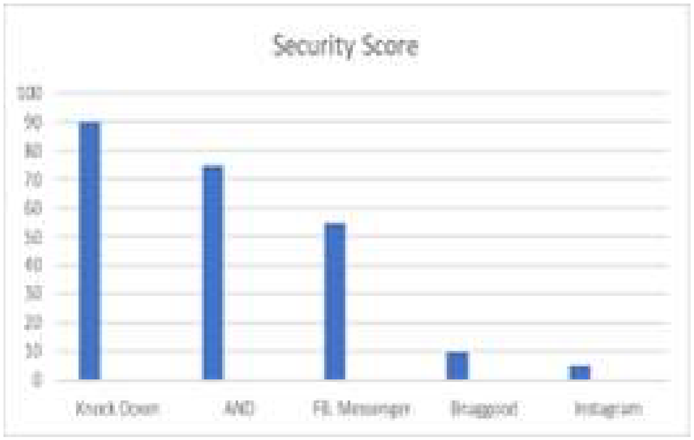

Fig. 6. CVSS Comparison of Different Applications

IV. Conclusion

Increased mobile app use heightens security risks, especially with unsafe apps, like unrated ones on the Play Store by inexperienced creators. These apps skip testing, leaving vulnerabilities that could be exploited. Inadequate code examination deprives developers of reliable bug detection methods. Users might download infected apps from untrusted sources or the Play Store, hiding vulnerabilities. Our research uses MobSF to analyze Android apps, revealing hidden vulnerabilities. Techniques like binary analysis uncover errors, insecure functions, Common Weakness Enumeration (CWE) occurrences, flawed authentication, unnecessary permissions, and risks from untrusted sources.Mitigating CVE/CWE vulnerabilities requires testing, often overlooked by developers. Considering CVE/CWE pre-coding and validating with tools like VirusTotal or MobSF strengthens app security.

The Android application is subjected to a thorough binary analysis by the Mobile Security Framework. This method reveals malicious code pieces that are hidden within the programs. These may subsequently be used to create rules for intrusion detection systems like Snort and Bro. These rules may also be synchronized with Quark to improve the Mobile Security Framework's network surveillance capabilities.

References

[1] H. Abdullah and S. R. Zeebaree, "Android mobile applications vulnerabilities and prevention methods: A review," *2021 2nd Information Technology To Enhance e-learning and Other Application (IT-ELA)*, pp. 148–153, 2021.

[2] A. Khandelwal and A. Mohapatra, "An insight into the security issues and their solutions for android phones," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 2015, pp. 106–109.

[3] Z. D. Patel, "Malware detection in android operating system," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE, 2018, pp. 366–370.

[4] D. Gökçeoğlu and Ş. Doğan, "Malware analysis on android devices-dynamic analysis," *PROCEEDINGS BOOKS*, p. 45.

[5] G. Suciu, C.-I. Istrate, R. I. Răducanu, M.-C. Diţu, O. Fratu, and A. Vulpe, "Mobile devices forensic platform for malware detection," in *6th International Symposium for ICS & SCADA Cyber Security Research 2019 6*, 2019, pp. 59–66.

[6] N. Kohli and M. Mohaghegh, "Security testing of android based covid tracer applications," in *2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*. IEEE, 2020, pp. 1–6.

[7] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dynalog: An automated dynamic analysis framework for characterizing android applications," in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE, 2016, pp. 1–8.

[8] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2016.

[9] R. Mahmood and Q. H. Mahmoud, "Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code," *arXiv preprint arXiv:1805.09040*, 2018.

[10] A. Mehrotra and C. Bhardwaj, "Pegasus has given privacy legislation a jab of urgency," *The Live Mint*, pp. 1–3, 2021.

[11] E. Downing, Y. Mirsky, K. Park, and W. Lee, "Deepreflect: Discovering malicious functionality through binary reconstruction." in *USENIX Security Symposium*, 2021, pp. 3469–3486.

[12] J. Shim, K. Lim, S.-j. Cho, S. Han, and M. Park, "Static and dynamic analysis of android malware and goodware written with unity framework," *Security and Communication Networks*, vol. 2018, 2018.

[13] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dynalog: An automated dynamic analysis framework for characterizing android applications," in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE, 2016, pp. 1–8.

[14] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[15] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," in *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. IEEE, 2014, pp. 3–17.

[16] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the seventh european workshop on system security*, 2014, pp. 1–6.

[17] L.-K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[18] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann, "Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques," *International Journal of Information Security*, vol. 14, pp. 141–153, 2015.

[19] M. Sharma, M. Chawla, and J. Gajrani, "A survey of android malware detection strategy and techniques," in *Proceedings of International Conference on ICT for Sustainable Development: ICT4SD 2015 Volume 2*. Springer, 2016, pp. 39–51.

[20] K. Tam, A. Fattori, S. Khan, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS Symposium 2015*, 2015, pp. 1–15.

[21] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[22] A. Ali and S. Al-Perumal, "Source code analysis for mobile applications for privacy leaks," in *2021 IEEE Madras Section Conference (MASCON)*. IEEE, 2021, pp. 1–6.

[23] M. Molski, "Theoretical study on the radical scavenging activity of gallic acid," *Heliyon*, p. e12806, 2023.

[24] I. Shammugam, G. N. Sam, P. Magalingam, N. Maarop, S. Perumal, and B. Shanmugam, "Information security threats encountered by malaysian public sector data centers," *Indonesian Journal of Electrical Engineering and Computer Science*, 2021.

[25] N. Kaur, S. Azam, K. Kannoorpatti, K. C. Yeo, and B. Shanmugam, "Browser fingerprinting as user tracking technology," 2017.