An Overview of Android Operating System Security

Kyler Dickey
College of Engineering and Computer Science
Arkansas State University
Jonesboro, Arkansas, USA
kyler.dickey@smail.astate.edu

***Abstract*—Android security is an imperative topic of discussion for researchers and developers due to its sheer scale on the market. The platform is an appealing target for attackers because of the open nature of the platform's development, as well as the nature of the platform itself which attracts casual users that make use of the applications and services offered by the device to store sensitive and private information. Android's security issues are made worse by inappropriate development practices, a lack of tools for those developers to adequately test and validate the security of their software, and the fragmentation of the platform. A large portion of the vulnerabilities that exist in the Android platform are derived from low-level software components of the stack due to the pitfalls of low-level programming. Significant research and attention have been given to the platform in the past decade, which is helping shore up the defenses of the platform to make the security components in the platform more robust and comprehensive.**

***Index Terms*—Android Malware, Android Vulnerabilities, Mobile Platform Security, Systems Security, Privacy**

I. Introduction

The Android operating system (OS) is a mobile computing platform designed for personal use. The OS platform is built on the Linux kernel and has additional components that facilitate application development and hardware interfacing [1]. The platform is an open source project licensed under the Apache License [2].

The platform consists of a number of layers referred to as the "software stack". The lowest layer is the Linux kernel, on which the rest of the operating system components are built. The next lowest layer is the hardware abstraction layer (HAL), which exists as an interface for hardware vendors to implement to ensure their hardware is compatible with the higher level components of the stack. The next layer is the native libraries installed on the platform, which consist of widely-used low-level implementations of common services used by applications (encryption, rendering, etc.) provided for developer convenience as well as OS functionality. The framework layer exists in the abstraction level above the native libraries and contains wrapper APIs for direct interfacing with a number of them. The applications for the platform are built using the tools listed prior. The Android runtime is also worth mentioning (the runtime exists at the same level of abstraction as the native libraries) because it ensures that applications run in their own virtual environment.

The Android platform has been maturing for well over a decade now, and has been reported to have a significant number of vulnerabilities in that time [3], [4]. Vulnerabilities seem to exist mostly in the Linux kernel layer and withing the supported native libraries. The reason for this is likely due to the difficult nature of programming using low-level languages.

Android's worldwide market share is immense [5], which is likely what paints the platform as a lucrative target for attackers. DroidKungFu, a particularly sophisticated malware family that targeted Android, exposed the platform at the time of its discovery outed the platform as woefully unprepared to deal with the threat of zero-day malware attacks, even from variants of known malware families [6]. Since then, a significant amount of research attention has been given to the platform, especially within the domain of malware detection and prevention [7], [8].

Android applications also suffer from a number of security issues. Frequently, applications are implemented with incorrect protocols or applications contain misused components. These cases of misuse introduce vulnerabilities into the applications that uses said protocols and components [9]–[11]. Misuse of these components seems to stem partially from misunderstandings on the part of the developer.

Tools to help developers avoid programmatic pitfalls are constantly being developed. Such tools would help programmers avoid introducing security issues into the applications made available to their user base and to the Android platform as a whole. Examples of new tools that developers could use to secure the system include newer, safer programming languages and software designed to detect and mitigate vulnerabilities and malware that were developed through research.

This survey is partitioned into three sections. Section I is a brief overview of the survey. Section II is further divided into subsections. Each subsection describes a part of the Android software stack and describes key vulnerabilities that exist at that layer. Section III describes various malware-related vulnerabilities that exist or existed in the Android platform. A discussion about the efficacy of malware detection on the platform is also specified. Section IV describes other vulnerabilities in the Android platform that don't fit neatly anywhere else in the survey. Section V concludes the survey and reaffirms courses of action that may be taken to increase the overall security of the platform.

II. Android Architecture Security

This section details the overall architecture of the Android OS and various security concerns at each level. The operating system is composed of five major layers that facilitate access

to the hardware components and software services provided by the OS. Figure 1 shows Android's architecture model by layer and is pulled directly from the Android developer documentation [1].

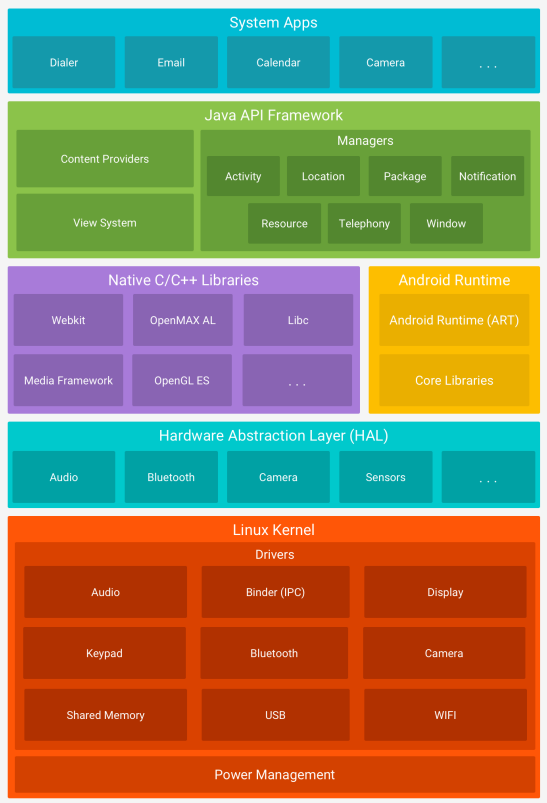

Fig. 1. Android architecture model.

Linares-Vasquez *et al.* breaks downs the vulnerabilities affecting the Android system [3]. A majority of vulnerabilities occur at the lowest level of abstraction for the Android architecture model: the Linux kernel layer. The next highest share of vulnerabilities exist within the native libraries supported by the Android platform. These two portions of the Android architecture account for roughly 73% of the sampled vulnerabilities studied. This study provides a good overview of the distribution of vulnerabilities that exist in the Android platform.

A. The Linux Kernel Layer

The lowest part of the architecture stack is the Linux kernel that Android derived from. The primary benefit of building an OS based on a Linux kernel is the maintenance performed on the Linux kernel by Linux developers. In the case of Android, the overwhelming majority of bug fixes (95%) are the result of work done by the Linux kernel development team [12].

Another benefit of basing an OS on the Linux kernel is the wide range of functional OS support provided by the kernel already. The adapted version of the Linux kernel used as the foundation to build the Android OS employs preexisting kernel features. Most changes made to the Linux kernel adapted for Android extend current kernel functions or employ new features that make the OS more suitable for use on mobile devices, such as a more aggressive memory manager used to preserve the limited memory resources on the device and a power manager [1], [12].

Linux kernel development, at the time of version 2.6, was carried out by a large group of developers from a multitude of different organizations and from developers acting independently [13]. A significant portion of the kernel was dedicated to driver and architecture support, which is code that interfaces and provides hardware support. This large pool of developers doesn't explicitly increase the safety of the software, since there is no substantial evidence to support the claim that open source software is more secure than a proprietary counterpart [14], [15]. Instead, the open source nature of the Linux kernel facilitates three key points:

- Organizations may develop drivers for their own hardware and maintain a measure of control over their own hardware security as a result. More generally, users have direct control when improving security of open source systems.
- Organizations and their developers may trust open source software more than a proprietary alternative.
- Organizations and their developers gain access to new features of the kernel approved by the maintainers of the source code.

The last point is particularly pertinent. The earliest versions of Android were based on the Linux kernel 2.6 versions [2]. These versions of the Linux kernel introduced the Linux Security Modules (LSM), which is a framework for general purpose access control [16]. The LSM explicitly supports the implementation of different security models that already existed at the time the feature was introduced, without favoring one over the other. A notable example would be SELinux, which is an extensive, non-discretionary access control implementation contracted by the National Security Agency (NSA) that was adapted to use the LSM [17]. Access to these types of security feature updates are important for implementing OS functionality such as the mandatory access control that exists in more recent versions of the Android OS [18].

Third-party hardware drivers are the largest source of vulnerabilities within the kernel. This means that hardware vendors introduce the highest amount of vulnerable code into the project [3]. A possible reason for this is a vendor's profit incentive. So long as the majority the consumers of the vendor's products believe that the amount of risk associated with owning a device with insecure drivers is acceptable, those consumers will continue to pay for the device. A lack of interest or tangible change from the perspective of consumers that comes from security-related development gives a vendor no reason to deviate from the business model of developing the new features instead of securing existing systems [15].

B. Hardware Abstraction Layer

The HAL exists as an interface because the higher level application programmable interfaces (API) needs to remain

the same for platform application developers [1]. In order to decouple the kernel drivers from the higher layers, the concrete implementation of the HAL is left to the hardware vendors. Only the vendor would have knowledge of the hardware used to assemble the device and the drivers used to control them.

Since the layer is an abstraction expected to be fulfilled by the aforementioned vendors, Android cannot explicitly trust the code inside concrete implementations as authentic. However, the HAL is among the least affected layers when it comes to security-related vulnerabilities, according to the study done by Linares-Vasquez *et al*. [3], meaning the vulnerabilities are less likely to exist in the HAL layer. The largest sources of vulnerabilities within the layer in the sample used in the study are the media component interfaces.

C. Android Runtime

The Android Runtime, like the HAL, suffers very little from vulnerabilities in comparison to the other layers. This layer consists of the Dalvik Virtual Machine (VM) or the Android runtime (ART) and the core runtime libraries that provide functionalities within the scope of the Java programming language [19]. The Dalvik VM and the ART are the environments that the system applications and some system services run on [1].

Most of the vulnerable code in this layer (of which there is comparatively low amount of compared to other parts of the Android software stack) exists in the core libraries. Core libraries are derived from the implementations of the Java programming language. Android has since moved to using OpenJDK [20] over other implementations, mainly due to legal issues between Google LLC and Oracle America Inc. over the latter's claim copyright infringement [21], [22].

Bugs may exist in the implementations that provide attackers with information about a system. These forms of attacks are called side-channel attacks [23]. The behavior of an implementation, such as the time intervals of system calls and power consumption, can be observed by an attacker to glean information about the aforementioned implementation. The attacker may then combine that information with knowledge about algorithms or standards used by the system. For example, an attacker's knowledge of encryption algorithms may be combined with information extracted from a system implementation may allow the attacker to determine the value of a key.

Tizpaz-Niari *et al*. demonstrated a tool, FUCHSIA, developed to detect the presence of side channels [24]. In the study, the researchers found a zero-day side-channel vulnerability in OpenJDK implementation. Early returns in a method used in the cryptography library resulted in code that could have been exploited by timing side-channel attacks. This issue has since been resolved by the OpenJDK development team, but is a good example of the types of implementation vulnerabilities that have the potential to affect the Android platform core libraries.

D. Native Libraries

Native libraries in the Android platform consist of native compiled code that is specific to the system hardware architecture [1]. These native libraries are typically implemented in C [25] and/or C++ [26]. The implementation of these libraries in low-level languages like C and C++ is important because low-level programming languages tend to offer better performance than other languages (namely Java, the primary language used for Android application development) in terms of computation time and memory usage [27]. Memory usage is particularly important for mobile embedded devices, since such devices tend to have limited capabilities for scaling the amount of physical memory that can be fitted onto the device.

Native libraries are utilized in the Android platform in two ways. First, the native libraries are used by the Android system itself to implement low-level components such as those in the HAL. Second, Android provides wrapper APIs written in Java for native libraries. These wrapper APIs allow programmers to access some native libraries made available by the Android system without re-implementing them in Java (which wouldn't be an optimal use of the system's limited resources anyway). This has the added benefit of reducing the amount of development maintenance needed to keep the hypothetical re-implementations patched.

A study done by Daoyuan Wu *et al*. further corroborates the findings in the study conducted by Linares-Vasquez *et al*. in regard to breaking down the ratio of vulnerabilities per layer of the Android platform [3], [4]. Both studies report that the native libraries own the second-highest number of vulnerabilities, with the Linux kernel having the absolute highest number of vulnerabilities in both cases as well. Both studies also report that a large portion of the vulnerabilities that exist in the native library layer is due to poor handling of memory allocated by the libraries, especially by media and communication components.

The most likely reason for the sheer quantity of vulnerabilities in both the Linux kernel layer and the native library layer lies in the code. Both layers are largely implemented in low level programming languages. It is far more difficult to write secure code in C and C++ especially because of the lack of any safety features that exist in languages like Java, which has robust bounds checking on data structures [19]. These safety features come at the cost of reduced performance, but greatly reduce the chance of a program being exploitable since a program written in a language with bounds checking will raise an exception or error instead of allowing the program to continue unchecked.

Poor memory handling has the potential to be catastrophic for any computerized system. Vulnerabilities like the buffer overflow open the floodgates for a multitude of different attacks, let alone accidental memory corruption. There are numerous examples of the different attacks that may be conducted from buffer overflow vulnerabilities [28], [29].

Figure 2 is a high-level abstraction of a buffer overflow in memory. It should be noted that the buffer overflow can

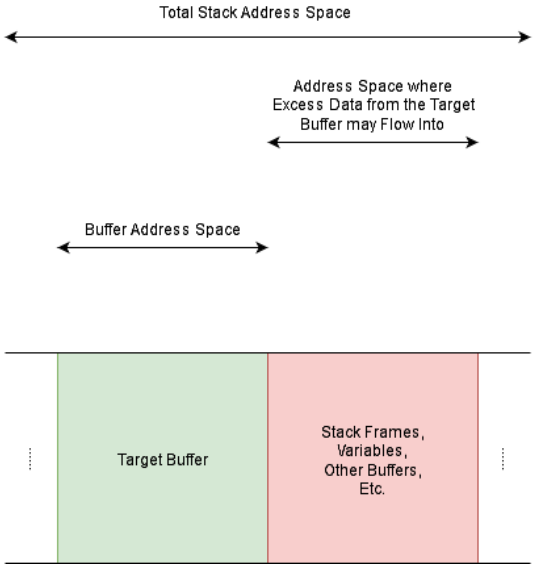

Fig. 2. High-level diagram of a buffer overflow.

overwrite parts of a stack frame. Attackers can intentionally overwrite the return address of the stack frame to redirect a program to malicious code.

One such example of an attack conducted via buffer overflow is an attack on Android's KeyStore service [30]. The KeyStore service allows the user to securely store keys for cryptographic operations [31]. Keys are stored on disk in files containing key as ciphertext (keys are stored in encrypted form). The keys are identified by filename. The lack of bounds checking on buffers present in the code (explicitly left out by the KeyStore authors for performance reasons) allows a programmer to inject arbitrary data into the stack memory through the Java API. By using exploiting this vulnerability, an attacker could inject malicious code that could leak the unencrypted keys. This attack is particularly damaging to a user, since the keys stored in the KeyStore could be used by applications installed on the device to access sensitive information.

It is difficult to overstate the importance of detecting and patching memory-related vulnerabilities. Since both the Linux kernel and the native library layers of the Android platform make use of low level languages that tend to allow these vulnerabilities, attacks may be conducted from a multitude of places within the software stack. Savvy attackers with a robust knowledge of a system's inner workings may find and exploit the vulnerabilities found from reviewing source code. At the same time, white hat hackers and researchers may also review the code and find vulnerabilities in the code and notify the maintainers so that the code may be patched. This is the double-edged sword of open source software.

In the meantime, researchers are developing tools to assist in the detection and prevention of memory corruption. A long-term solution might include the development of more modern, safer programming languages that avoid the classic pitfalls that come with programming applications and libraries in C and C++. One such example is the Rust programming language, which prevents unsafe memory operations at compile time unless explicitly told to ignore them using the "unsafe" keyword built into the language [32]. The "unsafe" keyword has the added benefit of making the code searchable if the aforementioned unsafe operations ultimately do result in exploitable or buggy code.

Other solutions might be the complete isolation of native libraries into a non-privileged environment, as proposed by Sun and Tan [33]. Their model proposes the loading of native code into a separate application all on its own that has only the permissions it needs to perform its functions (principle of least privilege). This limits the scope of the memory the native code has access to, decreasing the threat of memory corruption. This is an improvement, since normally native code has full access to a device's memory. It is clear, though, that there is no catch-all solution to fully patch and prevent these exploits in the native libraries (and in the Linux kernel).

E. Application Framework Layer

The application framework layer consists of a Java API that allows a programmer to access the services provided by the Android platform [1]. Both Linares-Vasquez *et al*. and Wu *et al*. report several vulnerabilities in the Activity Manager that exists at this layer [3], [4]. The activity manager is a service that oversees the application that are actively running on the system.

Armando *et al*. describe a Denial-of-Service (DoS) attack carried out by Android's Activity Manager [34]. The DoS attack effectively creates a fork bomb, filling the device memory space with identical processes until an automatic device reboot is triggered (which would cause an infinite boot loop in the case of the program being run from the bootloader). The attack exploits the mechanisms through which Android creates new processes via the Activity Manager, which involves interoperating with the Linux kernel layer to create new processes.

Attacks on the Activity manager are a good example of the types of vulnerabilities that may exist in the application framework layer. Such attacks highlight a particularly unsafe part of the framework, since vulnerabilities here could potentially cause a loss of availability to one or many applications or services. Other areas of interest might be the telephony API, the built-in web browser, or any other component that facilitates communications and connections (which tend to be vulnerable to man-in-the-middle attacks among others).

F. Application Layer

The application layer is where the programs accessible by the end user of the device exist. These applications range from short messaging service (SMS) applications, to web browsers, to calculators. Applications are allowed to use each other's services in order to reduce the burden on the programmer to provide code for features users would want from an application (such as two-factor authentication, autofill from a password

manager, etc.) [1]. Application security in the Android environment is discussed at length in Section IV.

III. Android Malware

Android malware poses a significant threat to the user base of the platform. Smartphones are a vital tool for many and functions as a platform for work and play. Android, being a leader in the smartphone OS market worldwide, is used by a significant number of people [5]. The safety of this platform is of critical importance to ensure that the users of the OS do not incur personal losses and to ensure that the platform may not be a medium for larger scale attacks on networks.

Recalling earlier discussion about the application framework layer in the Android platform [1], it is clear that a large collection of powerful tools are available to the programmer. Furthermore, it is possible to download and install applications not listed on any app stores as long as those applications fulfill the application requirements and follow the APK format [35] These conditions (powerful, easy to use, well documented platform software tools and an attack surface with little to no moderation) make Android a ripe target for Trojans, adware, and otherwise potentially unwanted applications (PUAs).

In a study published in 2012 about the characterization of malware targeting the Android platform, Zhou and Jiang reported that 86.0% of the malware sample they characterized were repackaged applications containing malicious code that was added by the attackers [6]. Furthermore, the study highlights an outbreak of a notorious Android malware named DroidKungFu. The outbreak of DroidKungFu included the initial version of the malware, as well as several version iterations and variants.

As DroidKungFu evolved through variants and versions, it became more sophisticated, employing self-installation of a functional copy of the malware payload in the repackaged application. This ensured that the malicious payload would remain even if a user uninstalled the original infected program from the device. This payload was encrypted in later versions, which is a sophisticated obfuscation technique used by attackers to hide their malicious programs from detection.

In fact, in the same study that publicized the existence of DroidKungFu, Zhou and Jiang found that mobile antivirus services were inadequate solutions to malware detection [6]. The software used to detect malware on the platform seemed to struggle to detect newer versions of the same malware family, due to the latter's rapid evolution and the former's use of signature-based detection.

Signature-based malware detection is considered to be largely ineffective when defending against so called "zero-day" malware, which is a fancy term for unmitigated operational malware that exists on live system, sometimes without the user's knowledge. Signature-based malware detection schemes rely on pre-existing knowledge of a malware's existence, since a signature needs to be generated against the malware. This gives zero-day malware time to propagate, damage, leak data, and otherwise cause harm to the platform ecosystem [36].

The fact that malware like DroidKungFu circumvented the techniques used by antivirus services through versioning and branching variants is a topic of concern. It is apparent that malware authors only need to expend minimal effort to obfuscate their malware. In the case of DroidKungFu, the release of newer versions might have only taken a matter of months.

Faruki *et al*. give a good overview of the types of malware that threaten the Android platform, as well as other vulnerabilities and issues with the system (which is discussed in Section IV) [7]. This survey also highlights a number of efforts that have been made to secure the platform against threats and study them. The study also provides a robust and clear breakdown of a number of malware families and the timeline of their emergence.

IV. Other Android Security Challenges

One of the principal reasons that the Android platform suffers from a plethora of security issues is fragmentation [2], [7], [37]. The fragmentation of the Android platform is derived from the various customizations new devices undergo during development. Customizations may include hardware drivers and pre-installed applications. As stated prior, the hardware drivers that exist in the Android platform are a significant source of vulnerabilities.

Since the implications and mechanics of the vulnerabilities at the low-level parts of the stack have been discussed at length in Section II, the main focus of this section will be on the applications. A study done by Zhou *et al*. discusses the security implications of Android's fragmentation issue [37]. In the study, the researchers found that the topic was not well researched and that different images of the Android OS customized by hardware vendors contained security vulnerabilities. Particularly, a number of apps that came pre-installed on some images were given excessive privileges, among other issues.

In regard to responsible application development, Android applications are not always up to an acceptable standard. Web applications are a large source of vulnerabilities, which opens them up to network-based attacks. Third party libraries (code libraries written to assist in application development with the goal of reducing the burden of implementing complicated or repetitive aspects of other programs) are also a common source of vulnerabilities. The reason for these vulnerabilities seems to be, in large part, by developers who are either unaware of the potential vulnerabilities of their system, a lack of awareness or domain knowledge of the components used in their system, or a lack of understanding for how to use a particular technology [9], [10].

One domain of misuse includes OAuth implementations. OAuth is an open standard for access control mechanisms built into Internet services. A common use case is to use the standard in an application to access the information of a service from another service or application if user permission is given. A study conducted by Wang *et al*. found that applications vendored by both American (specifically the Google

Play Store) and Chinese Android app markets demonstrated misuse of the OAuth protocol [38]. Part of the reason for this seemed to be a lack of understanding of the specifications by the developers. Figure 3, sourced from Oracle's documentation [39], demonstrates a typical control flow for the OAuth 2.0 protocol.

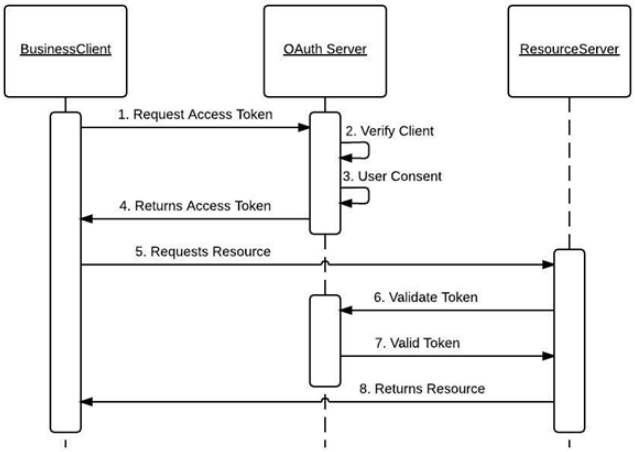

Fig. 3. OAuth 2.0 protocol control flow.

Another misuse case may be derived from a native library supported by the Android platform: SQLite [40]. SQLite is a widely used embeddable SQL relational database engine. SQLite, like other SQL databases, is susceptible to query injection [11].

Expanding on SQLite some more, a peculiar quirk that sets it apart from other SQL databases is access control. SQLite effectively have no access control mechanisms because the file is stored on disk (access to that file may be controlled like any other, but it does not extend to the tables, columns, rows, etc. stored in the file). Mutti *et al*. propose an SELinux-like model to the SQLite database included in the Android platform to improve access control to databases stored on-device [41]. Since applications on Android are sandboxed and are assigned their own user identifier, permissions could be given to certain applications to access only the subset of the database it needs, allowing applications to exchange data securely.

Application development on the Android platform could be significantly improved with the invention of tools that assist in the testing and verification of code safety. Developers could find themselves under significant time pressure, might be using unfamiliar components or standards, or may engage in happy-path testing. Programmers could produce poor quality code for any number of reasons. Tools and automation are consistent when developing sensitive systems and may help reduce the human aspect of flawed software.

V. Conclusion

A multitude of vulnerabilities have existed in every part of the Android platform. In particular, the platform suffers from common pitfalls encountered when a system is implemented using low-level programming languages like C and C++. These low level components exist in the Linux kernel layer and the native library layer in the platform's software stack. In the kernel layer, most of the vulnerabilities tend to come from hardware vendors implementing drivers for their products. In the native library layer, most vulnerabilities are derived from media-related libraries.

Malware is a pervasive threat to the Android platform, DroidKungFu being a particularly sophisticated example. Malware, left untreated and undetected, may do significant harm to users of the platform. Fortunately, new developments and research projects are being conducted on an ever-growing scale to combat the issue.

Other security concerns mainly pertain to inappropriate programming practices and lack of developer tools to ensure the deployment of robust code. Examples of this concern include SQL injections on the native SQLite library included in the Android platform as well as misused authentication protocols. Table I lists the various vulnerabilities that affect each layer of the Android software stack.

Steps may be taken to reduce the effect these vulnerabilities, such as updating the libraries to use modern languages with safeguards against exploitable memory errors and using compiler tools to detect faults before runtime. Work could be done to provide developers with tools to bolster the security of their systems by that could be used to detect, correct, and code to decrease number of vulnerabilities in the low-level systems. Adoption of better development practices and tools would drastically improve Android's current security situation going forward. However, fixing issues on existing systems would prove exceptionally challenging. The intense fragmentation of the platform exacerbates the issue of security by scattering the number of Android versions available on the market, making it difficult to manage and preventing some users from receiving important updates.

The Android platform is used on a large scale worldwide, and is seen as a lucrative target by attackers because of its sheer scale. As a result, the security of the system should be ensured by the developers and maintainers of the software used to build the platform, from the Linux kernel to the application framework. Furthermore, maintainers of reputable application sources should constantly take steps to ensure their reputability by keeping their platform free of malicious applications that could cause damage to users.

References

[1] Platform architecture. Google LLC. [Online]. Available: https://developer.android.com/guide/platform

[2] P. Gilski and J. Stefanski, "Android os: a review," *Tem Journal*, vol. 4, no. 1, p. 116, 2015.

[3] M. Linares-Vasquez, G. Bavota, and C. Escobar-Velasquez, "An empirical study on android-related vulnerabilities," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017.

[4] D. Wu, D. Gao, E. K. T. Cheng, Y. Cao, J. Jiang, and R. H. Deng, "Towards understanding android system vulnerabilities," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. ACM, jul 2019.

TABLE I
ANDROID PLATFORM VULNERABILITIES

Software Platform Layer	Vulnerability Examples Directly Affecting the Platform Layer
Linux Kernel	Corrupted Bootloader, Buffer Overflow, Data Leaks, Insecure Driver Code, Malicious Code Injection, Rootkits
Hardware Abstraction Layer	Untrustworthy Code Written by Third Party Vendors
Native Libraries	Buffer Overflow, Malicious Code Injection
Android Runtime	Side-Channel Attacks on Core Libraries
API Framework	Built-In Web Browser Abuse, DoS Attacks via the Activity Manager, Telephony API Misuse
Applications	Absence of Database Security, Adware, Authentication Protocol Misuse, Inappropriate Permissions, Trojans

[5] Mobile operating system market share worldwide. StatCounter. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/2021

[6] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, may 2012.

[7] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[8] P. Bhat and K. Dutta, "A survey on various threats and current state of security in android platform," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–35, jan 2020.

[9] J. Qin, H. Zhang, J. Guo, S. Wang, Q. Wen, and Y. Shi, "Vulnerability detection on android apps–inspired by case study on vulnerability related with web functions," *IEEE Access*, vol. 8, pp. 106 437–106 451, 2020.

[10] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "ATVHunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, may 2021.

[11] A. Maalouf and L. Lu, "Precise command injection analysis in android applications," in *2021 the 5the International Conference on Management Engineering, Software Engineering and Service Sciences*. ACM, jan 2021.

[12] F. Khomh, H. Yuan, and Y. Zou, "Adapting linux for mobile platforms: An empirical study of android," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012.

[13] G. KroahHartman *et al.*, "Linux kernel development," in *Linux Symposium*. Citeseer, 2007, pp. 239–244.

[14] G. Schryen and R. Kadura, "Open source vs. closed source software," in *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*. ACM Press, 2009.

[15] B. Witten, C. Landwehr, and M. Caloyannides, "Does open source improve system security?" *IEEE Software*, vol. 18, no. 5, pp. 57–61, 2001.

[16] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *11th USENIX Security Symposium (USENIX Security 02)*, 2002.

[17] S. Smalley, C. Vance, and W. Salamon, "Implementing selinux as a linux security module," *NAI Labs Report*, vol. 1, no. 43, p. 139, 2001.

[18] Selinux concepts. Google LLC. [Online]. Available: https://source.android.com/security/selinux/concepts

[19] Java programming language. Oracle Corporation. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html

[20] Openjdk. Oracle Corporation. [Online]. Available: https://openjdk.java.net/

[21] Google llc v. oracle america inc. Oyez. [Online]. Available: https://argument2.oyez.org/2020/google-llc-v-oracle-america-inc/

[22] R. Amadeo. (2016, Jan.) Android n switches to openjdk, google tells oracle it is protected by the gpl. [Online]. Available: https://arstechnica.com/tech-policy/2016/01/android-n-switches-to-openjdk-google-tells-oracle-it-is-protected-by-the-gpl/

[23] K. Tiri, "Side-channel attack pitfalls," in *Proceedings of the 44th annual conference on Design automation - DAC '07*. ACM Press, 2007.

[24] S. Tizpaz-Niari, P. Cerny, and A. Trivedi, "Data-driven debugging for functional side channels," 2018.

[25] B. Kernighan, *The C programming language*. Englewood Cliffs, N.J: Prentice Hall, 1988.

[26] B. Stroustrup, *The C++ programming language*. Upper Saddle River, NJ: Addison-Wesley, 2013.

[27] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[28] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: Practice and Experience*, vol. 33, no. 5, pp. 423–460, 2003.

[29] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote, "Detection and prevention of stack buffer overflow attacks," *Communications of the ACM*, vol. 48, no. 11, pp. 50–56, nov 2005.

[30] R. Hay and A. Dayan, "Android keystore stack buffer overflow," 2014.

[31] Android keystore system. Google LLC. [Online]. Available: https://developer.android.com/training/articles/keystore

[32] Rust. The Rust Foundation. [Online]. Available: https://www.rust-lang.org/

[33] M. Sun and G. Tan, "NativeGuard," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks - WiSec '14*. ACM Press, 2014.

[34] A. Armando, A. Merlo, M. Migliardi, and L. Verderame, "Would you mind forking this process? a denial of service attack on android (and some countermeasures)," in *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2012, pp. 13–24.

[35] Android fundamentals. Google LLC. [Online]. Available: https://developer.android.com/guide/components/fundamentals

[36] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. IEEE, nov 2010.

[37] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, may 2014.

[38] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability assessment of OAuth implementations in android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference on - ACSAC 2015*. ACM Press, 2015.

[39] Oauth 2.0. Oracle Corporation. [Online]. Available: https://docs.oracle.com/cd/E82085_01/160027/JOS%20Implementation%20Guide/Output/oauth.htm

[40] Sqlite. [Online]. Available: https://www.sqlite.org/index.html

[41] S. Mutti, E. Bacis, and S. Paraboschi, "SeSQLite: Security enhanced SQLite," in *Proceedings of the 31st Annual Computer Security Applications Conference on - ACSAC 2015*. ACM Press, 2015.