
aracne

Classificazione Decimale Dewey:

005.8 (23.) SICUREZZA DEI DATI

GERARDO **IOVANE**, GERMANO **INGENITO**

RED TEAM CYBERSECURITY LECTURES

ANDROID OFFENSIVE OPERATIONS: THEORY, ENGINEERING AND RED PREREQUISITES

aracne

ISBN
979-12-218-2569-5

PRIMA EDIZIONE
ROMA 9 MARZO 2026

Table of Contents

Part 1

1 Introduction

The emergence of mobile devices has revolutionized digital interaction, with Android capturing over 70% of the mobile OS market share in 2023. Consequently, Android has become a popular target for threat actors, facilitated by its open-source nature and vast ecosystem. Offensive attacks against Android systems are diverse and include malware, spyware, zero-click exploitation, and multi-phase attack chains that abuse Android's modularity and permission management. "Android Offensive Operations: Theory, Engineering and Red Team Practice" aims to provide a formal investigation into the methodologies and techniques applied in offensive operations for Android systems. The primary question to be addressed is: How can offensive operations targeting Android systems be systematically modeled, engineered, and simulated within a red team framework, and what methodologies, tools, and payload structures are most effective for achieving persistence, stealth, and data exfiltration in real-world Android environments? The discipline of offensive cybersecurity targeting Android systems has become a focus area in computer science, particularly due to the burgeoning of mobile-based malware and advanced anti-forensic tactics. There have been substantial publications and reports documenting the increasing volume and sophistication of Android-based malware samples, including novel repackaging techniques, obfuscation schemes, and stealth surveillance capabilities. There is a growing necessity to simulate offensive operations to thoroughly understand attack surfaces. Red team testing, payload engineering, and attack simulation can provide a pathway to better understand the limitations of Android and enable more robust security implementations. This paper aims to thoroughly explore the principles and organization of offensive cybersecurity on Android systems, providing a detailed reference that combines a comprehensive treatment of foundational concepts and advanced technical modules. The goal of this document is to (1) introduce the method of red teaming on Android devices, including reconnaissance, payload building, and software architecture, (2) present a framework to execute multi-phase attack chains through the implementation of custom Android payloads, and (3) introduce advanced methods for zero-click payload delivery, rootless persistence, anti-forensics, covert surveillance, and dynamic control. The combination of conceptual discussion, examples, and simulations serves to fulfill the analytical and operational needs of both researchers and practitioners alike. The methodology of this document combines literature review, historical analysis, comparative analysis, and source criticism to explore different attack

methods, techniques, and available tools. Real attacks are used to simulate different Android payloads, allowing for dynamic analysis and concrete, scenario-based examples.

While Android offensive cybersecurity has made progress in simulating adversarial tactics, (i.e., MITRE ATT&CK) and detection/prevention approaches, knowledge gaps still exist within red teaming, offensive simulation, and multi-phase attack chains. Several related works either introduce single techniques or exclusively address detection. No publications discuss offensive simulations and how they can be used to model more realistic adversarial behavior, particularly in a red team setting.

This work provides a combination of technical depth and simulation-based demonstrations to bridge the knowledge gap. The introductory part introduces the motivation behind the project and describes the central research question and goals. The following chapters explain the process of reconnaissance and intelligence gathering, focusing on OSINT and metadata profiling. Then, it delves into advanced topics, such as payload engineering and obfuscation, which is followed by file-based delivery, proximity-based delivery, and C2-initiated delivery. Afterward, exploitation and execution are described, including trigger mechanisms, API abuse, background execution, and hiding the payload. Next, a chapter is focused on Android payload persistence and anti-forensics. Finally, the last chapter delves into the communication, data exfiltration, and control aspects that are important to Android red team operations.

1.1 Android OS Security Architecture

Android's permission-based security model is a basic principle of its architecture. It is meant to enforce fine-grained access control over system resources and user data. Each application must explicitly declare the permissions that it requires in the XML manifest file. Permissions are granted to limit access to resources such as the camera, location, or contacts. This limitation is system-wide enforced. However, user behavior plays a big role in defeating this security model, as users approve permission grants without careful assessment. According to Ahmed and Sallow (2017, p. 3) and Enck et al. (2009, p. 5), this behavioral vulnerability exposes Android devices to information leakage and malicious code exploitation. The permission-based security model is most favorable to red teams and attackers because of this vulnerability.

In Android, each application is assigned to a unique UID to isolate applications, so direct inter-application access is not allowed. Isolation is enforced by the operating system's sandboxing. Application isolation

can be bypassed using Android permissions and/or inter-process communication. Only applications that share the same UID can communicate with each other directly or access another's data. The theoretical application isolation proposed by Enck et al. (2009, p. 1) is compromised through enforcement flaws, incorrect permissions, and/or attack strategy. Application-level privilege escalation can be obtained by repackaging or through IPC and application-level vulnerabilities.

Originally, Android's permission control system was designed with defense-in-depth in mind. All system resources are controlled through permissions granted in the XML manifest file. Permissions control acts as a binding contract that regulates application-OS interaction. Security inconsistencies between the granular application controls have led to possible permission-related attack scenarios. As per Ahmed and Sallow (2017, p. 3) and Enck et al. (2009, p. 5), Android developers can improperly configure permissions. If these loopholes are found by a red team, an attacker may utilize permission flaws to propagate attacks.

The Dalvik Virtual Machine (DVM) is a component of the software stack which operates as a just-in-time interpreter that is designed and optimized for mobile devices. According to Ahmed and Sallow (2017, p. 2), applications are isolated in the DVM using sandboxing. As a result, only the compromised application's environment is at risk of malicious activity. However, in practice, applications make many calls to system APIs for the most basic functionality. Enck et al. (2009, p. 1) suggests that system API calls can be abused for covert activities. For example, activities in one Android application can start another, but only if certain IPC criteria are met. Intents, content providers, and broadcast receivers are the main IPC mechanisms. Broadcast intents have several purposes. However, they have security implications because they can be sent by and delivered to any applications. As a result, malicious applications may intercept sensitive system information.

The system APIs and middleware introduce many risk points to Android because the orchestration of inter-component communication is dependent on them. As reported by Ahmed and Sallow (2017, p. 2), it is possible to modify API implementation for permission control. An adversary may make modifications that allows them to retrieve restricted data or escalate privileges. An API is a dynamic attack surface because attackers may discover novel payload injections at various times, as reported by Ahmed and Sallow (2017, p. 2). Theoretical intrusion prevention tools fall behind against sophisticated attack techniques that an adversary can use during payload delivery. Due to this, it is essential that system designers test defense strategies against potential threats.

Attackers can bypass the operating system-level sandboxing in ways that theoretical intrusion prevention cannot account for. Malicious code can be hidden and added at runtime, allowing it to bypass strict checks.

An attacker may be able to use dynamic code loading, dynamic obfuscation, and native code in APK files to evade code control limitations. According to Ahmed and Sallow (2017, p. 2), an Android device is not as isolated as it can be because isolation levels can be compromised through exploiting shared system resources, improper permissions, and incorrectly configured services. During a red team, if an attacker wants to bypass isolation for reconnaissance and penetration, he/she may use obfuscated code in Android's native library. By doing so, it makes detection, debugging, and inspection difficult for system security measures.

Repackaging Android applications is an all too prevalent attack in this mobile security ecosystem. Attackers can download a legitimate application's APK, decompile it, add malicious payload code, and then repackage it and re-upload it to either the app store or to third party repositories for distribution. According to Ahmed and Sallow (2017, p. 3) and Verma (2024, p. 1), the open source nature of Android applications makes repackaging quite easy and widespread. Vidas et al. (2011, p. 6) report that in 2 hours of observation, 3.5 repackaged versions are available for distribution, each of which has approximately 300 downloads.

The openness of the Android ecosystem is not only a source of rapid application growth but also a primary reason for the prevalence of Android-based malware. According to Verma (2024, p. 1), adversaries can analyze and understand Android source code much faster than a software developer or defender. The source code and internal algorithms allow attackers to design sophisticated payload techniques to either retrieve sensitive data or evade security protocols. Android's rapid ecosystem expansion has resulted in tremendous growth of fragmentation. As a result, red teams need to test across many devices and custom ROMs that can have varying security protocols installed. Many custom ROMs can also include undocumented application or API versions that can weaken security defenses, as noted by Caturano (2021, p. 106). A device that installs a custom ROM can install code or applications that can bypass normal system protections. Custom ROM developers can also install firmware that weakens many traditional countermeasures. Adversaries may exploit these variants in custom ROMs to deliver effective attack models.

Over time, tools have been developed to identify possible malware contained within repackaged applications. Both static and dynamic approaches are utilized in malicious code analysis. Static tools such as MalDozer and RevealDroid use API-based feature vectors, artificial intelligence and machine learning to classify applications as malicious or legitimate with an accuracy rate of up to 97.2%, as reported by Bacci et al. (2018, p. 2). Static analysis can achieve good results when not up against strong obfuscation

strategies. Dynamic analyzers such as Alterdroid simulate Android application execution. Alterdroid's analysis achieves close to a 99% detection rate for Android malware, Bacci et al. (2018, p. 2), but does so with a specific corpus of popular and prevalent malware samples. Many newer and more advanced dynamic application analysis technologies are being developed, and one commonality they all have is that they are no match for more modern and complicated obfuscation strategies.

A broader design problem for the Android ecosystem is balancing usability against security for Android device users. Users can be easily manipulated to bypass security barriers if an application request appears innocent. It is important to have end users in charge of access policies, but as Ahmed and Sallow (2017, p. 3) suggests, they are sometimes irresponsible when asked for sensitive permissions. They may blindly approve resource permission requests to install the most harmless of applications. Due to this behavioral loophole, red teams always take advantage of user error in attack model execution. By combining the weaknesses of architectural and implementation flaws with user manipulation, a realistic and effective attack model can be created for red teaming, confirming the findings reported by Caturano (2021, p. 5).

1.2 Threat Landscape and Actors

The threat landscape for Android devices is significantly affected by its immense market adoption and various types of attackers. Boasting over 70% of the smartphone market, Android devices have become a dominant target for attackers ranging from organized cybercriminals to nation states conducting advanced persistent threats. As Vidas et al. (2011, p. 1) and Ahmed and Sallow (2017, p. 1) argue, Android's dominance presents an ideal opportunity for attackers to exploit a largely homogenous landscape, allowing them to create sophisticated, highly scalable attacks with minimal investment. With over four billion downloads, Android's central marketplace creates a platform for both efficiency and scale, thus increasing its attackability (Vidas et al. 2011, p. 1). The ecosystem’s scalability is therefore an advantage for attackers who wish to execute attacks across a vast landscape of similar device models.

The ease with which a malicious app can gain legitimacy contributes to opportunistic attacks against Android devices. In the app ecosystem, attackers publish seemingly benign applications on the marketplaces to attain a high rate of downloads from users. For instance, an example app of Vidas et al. (2011, p. 6) accumulated over 200 downloads in just 24 hours after publishing. The effort that social engineering plays as a means for distribution is paramount in opportunistic attacks, as well as the

exploitation of the implicit trust users hold within the application ecosystem. With a more open app ecosystem and lower development overhead than other platforms, both novice and veteran attackers are able to freely exchange attack tactics and techniques, making opportunistic attacks more profitable and widespread on Android, as pointed out by Ahmed and Sallow (2017, p. 1).

The permission system on Android safeguards resources from misuse by apps installed by the user. The permission system relies on the user to appropriately accept these requests for apps to gain the ability to access resources within the Android phone. Ahmed and Sallow (2017, p. 3) argue that the lack of OS verification for the user-consented permissions allows attackers to easily engineer the consent from users. Social engineering is paramount in the Android attack surface. Similar to the traditional web environment, users fall prey to cleverly crafted advertisements and links, which manipulate their willingness to share permissions. Exploiting user confusion and inherent trust, adversaries can effectively distribute malicious software to devices, a feat that can be easily replicated thanks to Android's popularity. Users are prone to granting more permissions than an application needs (overprivileged), and many third-party applications ask for excessive access to resources they do not need to operate (Ahmed and Sallow 2017, p. 3).

In repackaging attacks, attackers decompile a legitimate application, insert malware, and publish it to official marketplaces or other third-party application stores. The open-source nature of the Android operating system allows attackers to modify the APK, embed malware, and repackage legitimate applications (Ahmed and Sallow 2017, p. 3). Further, the openness of many application stores gives way for attackers to push these malicious applications without raising red flags to most users (Vidas et al. 2011, p. 5). This allows the adversary to benefit from users' implicit trust of familiar applications in the official Android app stores. These repackaged applications can have widespread effect as seen in past malware such as the Geinimi and DroidDream campaigns, which infected dozens of legitimate Android applications. Repackaging is popular with malware authors due to the increased stealth. Many malware detection schemes are unable to detect these attacks, as static analysis of applications fails to realize that malicious code was injected. Although static analysis of such attacks may be successful when the applications are running dynamically, Vidas et al. (2011, p. 6) explain that common antivirus techniques utilizing signature-based dynamic analysis are also unable to capture the malware. Red teams, in this respect, are crucial to simulating attackers using this technique and, by extension, assessing defensive strategies as proposed by Ahmed and Sallow (2017, p. 3).

Another type of attack, privilege escalation, gives adversaries root access to devices in order to circumvent security barriers. Bugiel et al. (2011, p. 2) described such an attack called Soundcomber that leveraged