Open Sesame! Design and Implementation of Backdoor to Secretly Unlock Android Devices

Junsung Cho, Geumhwan Cho, Sangwon Hyun, and Hyoungshick Kim*
Sungkyunkwan University, Republic of Korea
{js.cho,geumhwan,swhyun77,hyoung}@skku.edu

Abstract

This paper presents a practical design of backdoor to permanently bypass the screen lock mechanisms on Android devices. Our design has many advantages such as difficulty in detecting backdoor, fast execution time and low power consumption. The key feature of our backdoor is *remote triggering* that allows the backdoor to be temporarily triggered and executed through push notification services also used by many normal applications. Furthermore, in our proof-of-concept backdoor, about 98% of 4-digit PINs and screen lock patterns were cracked within 5 seconds, and only a small amount of power was consumed. We show the stealthiness of our backdoor to effectively evade the existing malware detection tools (55 anti-virus scanners provided by VirusTotal and SandDroid).

Keywords: Android, Malware, Backdoor, Firebase

1 Introduction

To protect user's mobile devices, Android provides several screen lock mechanisms such as Personal Identification Number (PIN), pattern, password and fingerprint. In particular, PIN and pattern are popularly used (about 42% of mobile users use those mechanisms) [6]. Moreover, although biometric authentication allows to unlock a device, users are still required to set PIN or password on their devices by default because a high entropy secret should not be stored on the device itself [3]. In consequence, the overall security is only as good as the used PIN or password.

The goal of this paper is to design backdoor that provides the attacker with persistent access to the victim's mobile device by compromising the secret for user authentication while effectively hiding its presence from the victim. To achieve this goal, we need to find suitable answers to the following research questions: "How can the attacker economically trigger the backdoor in a stealthy manner?", "How can the backdoor obtain the victim's screen lock password?", and "How can the backdoor deliver the captured password to the attacker?" To answer those questions, we designed and implemented backdoor and examined its effectiveness in terms of execution time and power consumption. In this paper, our contributions are summarized as follows:

1. We designed novel Android backdoor which allows an attacker to permanently unlock a victim's Android device whenever the attacker wants to access it. The designed backdoor cannot be easily detected by existing anti-virus scanners because the backdoor can secretly communicate with an attacker through *push-based* message delivery services that are also popularly used in normal Android applications (read Section 3).

*Corresponding author: Department of Software, Sungkyunkwan University, Suwon, Republic of Korea, Email: hyoung@skku.edu

2. We implemented proof-of-concept backdoor on Android 5.1 and evaluated its feasibility in terms of execution time and power consumption. Our results showed that about 98% of 4-digit PINs and screen lock patterns were cracked within 5 seconds via dictionary attacks while a small fraction (below 0.01%) of the total power consumption was consumed. Also, existing anti-virus scanners failed to detect our proof-of-concept backdoor implementation (read Section 4).

3. We suggest reasonable mitigation techniques against the proposed backdoor on Android by protecting hashed password files with a trusted execution environment and reporting the history of suspicious unlock activities and/or apps using the push notification messages to the smartphone owner (read Section 5).

In the rest of this paper, we will present the above results in detail. First, we will describe the threat model considered to provide a better understanding of our backdoor design and then present the design and implementation of our proof-of-concept backdoor.

2 Threat model

In this section, we describe the threat model considered to provide a better understanding of our backdoor design and then present the design and implementation of our proof-of-concept backdoor.

2.1 Attacker's goal

We assume that the victim uses a properly secure screen lock mechanism (either PIN or pattern) to protect his (or her) smartphone. Moreover, the victim can often update his (or her) unlock secret. Under these conditions, the attacker's goal is to continuously spy on a victim's smartphone without revealing his (or her) spying activities. In practice, many likely attackers are such *insiders* rather than strangers in that the people who most want to intrude on a victim's privacy are likely to be in the victim's circle of acquaintances. Muslukhov et al. [10] found that smartphone users are most concerned with protecting their smartphone data from unauthorized access by insiders (e.g., their family members, school friends, college classmates, and work colleagues).

2.2 Attacker's capabilities

Probably, an *insider* attacker can often have some chances to install his (or her) backdoor on a victim's smartphone in a stealthy manner by either physically accessing the device or performing a sophisticated social engineering method [5] (e.g., sending a gift app). To hide its existence, the backdoor can also be embedded in a legitimate application as a repackaged version of the application [14], which is popularly used in distributing Android malware. We assume that the attacker's backdoor is secretly installed on the victim's smartphone at the initial stage.

We also assume that the victim's smartphone is already *rooted* or will be rooted before performing the attacker's spying on the victim's smarphone because the hashed password files (i.e., "`*.key`" files) can be accessed on rooted devices only. Probably, this is a rather strong assumption but it is one that appears to be commonly made. In practice, many Android users have rooted their Android devices to update their operating system with a new version of Android or uninstall unnecessary built-in apps. According to a survey, about 7% of survey participants rooted their device [7]. Furthermore, the Google's official report in 2016 [9] shows that about 5.6% of all Android devices were rooted by device owners or potentially harmful applications. According to the large-scale survey of Tencent [2], about 80% of Chinese smartphone users had rooted their Android smartphones during 2014. To make matters worse, Android devices can forcibly be rooted with a security bug too [12].

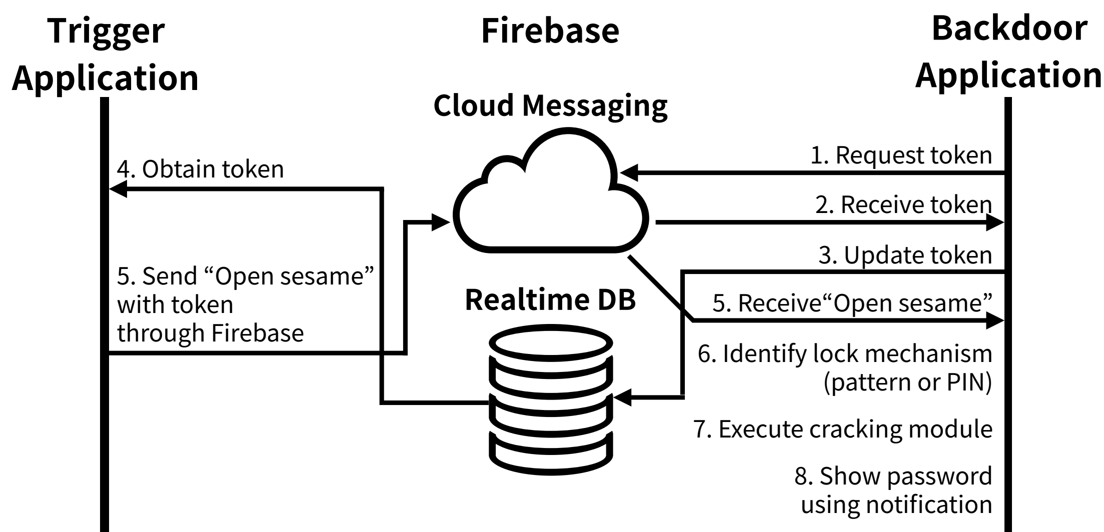

Figure 1: Overview of our backdoor implementation.

We note that the attacker cannot simply change or reset the victim's PIN or pattern even when a victim's smartphone is rooted for the following reasons: (1) the attacker's goal is to persistently spy on the victim's smartphone rather than a one-time observation. If the PIN or pattern is newly updated, the victim can notice that something has happened to his (or her) smartphone. (2) On Android, there is no simple way to change the Android device's unlock secret, which is totally different from conventional Linux systems providing the `passwd` command for the root user.

3 Design and implementation

To validate the feasibility of the proposed attack, we designed and implemented proof-of-concept backdoor on Android devices. Our implementation consists of three components: *trigger application*, *Firebase* (`https://firebase.google.com`) and *backdoor application*. Here, trigger application and Firebase are controlled by an attacker.

The implementation of the remote triggering feature is important for hiding the backdoor from the device owner (i.e., the victim). We achieved this by using Firebase which supports the communication between the attacker and the backdoor through push notification services. The use of push notifications makes it difficult to detect the malicious traffic using Firebase because many normal applications are also using Firebase for push notification services.

To perform experiments, the trigger application and the backdoor application should be installed on the attacker's and the victim's devices, respectively. We used two Android devices; the rooted Nexus 5 with Android 5.1 Lollipop for the victim's device to run the backdoor application, and Nexus 5X with Android 6.0 Marshmallow for the attacker's device to run the trigger application. For other Android versions, our backdoor can also be adapted. We found that our backdoor performs well on under Android 5.1 with just a slight modification of the code. As a result, 84.2% of all Android devices could be vulnerable to our attack in September 2017 (`https://developer.android.com/about/dashboards`).

3.1 Attack procedure

We describe how the attack is processed by remotely triggering the backdoor on the victim's device. Figure 1 illustrates the overall attack procedure.

```
Content-Type:application/json
Authorization:key=AIzaSyCRcGcfM7MsBwxxSyBp12i7g...zsu

{
  data : {
    msg : Open Sesame
  },
  to : bk3RNwTe3H0:CI2k_Hl3pja99tlsaiuuaAHOM1...H2N
}
```

Figure 2: Example of "Open sesame" message.

When the backdoor application is installed on the victim's device, the application requests and receives a unique token from Firebase (Step 1 and 2), and updates the real time database of Firebase with the new token (Step 3). Then the attacker retrieves the token from the database of Firebase (Step 4). Whenever the attacker wants to unlock and access the victim's device, the attacker sends an "Open sesame" message attached with the retrieved victim's token to the backdoor application via Firebase (Step 5). Once receiving the message, the backdoor application is automatically triggered to identify which lock mechanism (either PIN or pattern) is used by the victim (Step 6). Then the backdoor application executes the cracking module matching the identified lock mechanism and eventually finds out the victim's password (Step 7). Finally, the backdoor application informs the attacker of the found password by showing up a notification popup (Step 8).

A video demo of our implementation is available at (`https://youtu.be/iyWhsYyPFnc`).

3.2 Firebase

Firebase is a mobile platform that provides several features such as Firebase Cloud Messaging (FCM) and Firebase Realtime Database. In our implementation, the attacker uses FCM to send an "Open sesame" message to the backdoor installed on the victim's device. In order to send a message to a specific target application (e.g., the backdoor), the token information of the target is needed. The attacker runs an database which maintains a token database to collect the token information of the backdoor.

3.3 Trigger application

Trigger application is installed on an attacker's device and sends an "Open sesame" message to FCM (`https://fcm.googleapis.com/fcm/send`) through HTTP. In fact, trigger application can be implemented with any programming language and run on any platform. Figure 2 shows an example message in JSON format. `Authorization key` is used to indicate an FCM server to which the message is sent. The field of `to` contains the token to identify the target backdoor that should eventually receive the message.

3.4 Backdoor application

Backdoor application is designed to obtain a victim's password stored in the victim's device and expose it to the attacker whenever the attacker wants to access the device. In order to hide the backdoor application, we carefully designed it to provide the following properties in terms of resilience against detection. Firstly, we adopt the remote triggering mechanism, thus our backdoor is executed only when there is

```
hashedPassword ← getHashedPassword();
hashedOldPassword ← getHashedOldPassword();
if hashedPassword = hashedOldPassword then
    oldPassword ← getOldPassword();
    sendNotification(oldPassword);
else
    salt ← getSalt();
    while True do
        candidatePassword ← readFromDictionary();
        saltedPassword ← candidatePassword + salt;
        hash ← calculateHash(saltedPassword);
        if hashedPassword = hash then
            sendNotification(candidatePassword);
            break;
        end
    end
end
```

Algorithm 1: Password cracking algorithm.

a request from the attacker while normally hiding its presence. Secondly, our backdoor only communicates with the push messaging server without directly communicating with the attacker. Therefore, network intrusion detection systems cannot distinguish the network traffic of our backdoor from that of other benign applications that are also using push notification services. Thirdly, our backdoor requires the only permissions that are needed to use Firebase. Hence, it is not easy to recognize the potential risk of our backdoor even for security conscious users because those permissions are also required by normal applications using push notifications. For the same reason, the effectiveness of advanced malware detection techniques using permission behavior (e.g., [15]) might also be degraded.

3.4.1 Identifying the lock mechanism being used

Before cracking the password, the backdoor application first needs to identify which lock mechanism is currently used to protect the victim's device. The lock mechanism in use is specified in `lockscreen.password_type` field of `locksettings.db` which is a SQLite database file. To read this field, the backdoor application copies `locksettings.db` file from `/data/system/` to its internal storage, makes the copy readable, and reads `lockscreen.password_type` field using `SQLiteDatabase` class. The field has an integer value; for instance, either `131072` or `196608` stands for PIN, and `65536` stands for screen lock pattern.

3.4.2 Guessing the victim's password

To securely store a user input password, Android stores the hashed password instead of the password itself. We implemented Algorithm 1 to efficiently guess a victim's password on his or her Android device from the stored password hash.

To crack the victim's password, the backdoor application reads the hash value of the password from `password.key` for the case of PIN (or `gesture.key` for the case of pattern). In particular, the corresponding salt should also be retrieved from `locksettings.db` for the case of PIN.

Table 1: Average execution time to find a victim's password (Dictionary Attack (D) vs. Brute Force Attack (B)).

Time (sec)	**4-digit PIN**		**Lock pattern**	
	D	B	D	B
<**1**	52.68%	10.60%	85.34%	44.10%
<**2**	74.88%	57.88%	93.76%	62.36%
<**3**	85.76%	70.98%	96.22%	68.90%
<**4**	93.34%	83.72%	97.54%	71.00%
<**5**	98.72%	94.28%	98.28%	76.60%
Avg. Time	1.525s	2.316s	1.017s	5.015s

Definitely, it seems significantly effective to use a dictionary attack rather than a brute-force attack. A dictionary can be built with real world PIN (or pattern) datasets. Algorithm 1 summarizes our dictionary attack procedure to crack the victim's password. To avoid unnecessary computational efforts, this algorithm first checks whether the password was updated (see lines 1–6). The PIN and pattern cracking algorithms are similar except for the hash algorithms to use (see lines 10–11). Especially for PIN, the SHA-1 and MD5 hash values of a candidate PIN were calculated, respectively, with the salt and their concatenation is eventually tested while, for the case of pattern, the SHA-1 hash value of a candidate pattern is computed without a random salt.

For patterns, a *rainbow table* could be particularly used for pre-computing the dictionary of all unlock patterns since random salts are not used. However, in this case, the size of a rainbow table is about 5.2 Mbytes for 389,112 patterns. When a legitimate Android application is repackaged with such a large rainbow table, a significant increase in the file size could be a hint for detecting the existence of backdoor. Hence, we would not recommend using a rainbow table.

4 Evaluation

In this section, we show the feasibility of our backdoor by analyzing the stealthiness of the designed backdoor with several malware detection tools and measuring the execution time and the power consumption of our prototype backdoor application.

4.1 Stealthiness of backdoor

When casual users encounter this backdoor installation on their Android devices, they cannot realize that the backdoor is installed and used because their unlock secrets are remained unchanged and all the activities of the backdoor can be *temporarily* performed when a specific push notification is triggered by an attacker.

To validate our backdoor's resistance against malware detection, we tested whether commercial anti-virus scanners could successfully detect our backdoor implementation. We used VirusTotal (`https://www.virustotal.com`) and SandDroid (`http://sanddroid.xjtu.edu.cn`) that are popularly used for analyzing Android APK files.

VirusTotal provides 55 anti-virus scanners in total to detect various types of malicious applications. However, we found that all those scanners failed to detect our backdoor implementation. The detailed results are shown on the web page (`https://goo.gl/5vOeuG`).

SandDroid performs both static and dynamic analysis on a given Android application to measure its potential risk score between 0 and 100. In our experiment, SandDroid reported 24 as the risk score of our

backdoor implementation. The detailed results are shown on the web page (`https://goo.gl/ry3Dm6`). We note that this risk score is close to the mean for normal Android applications. The mean risk score of 30 randomly selected Google applications (e.g., YouTube and Gmail) from the Google Play (`https://goo.gl/k5iyOs`) was 26.47 with a standard deviation of 17.87. We also analyzed 30 malicious applications using SandDroid to compare their resulting scores with those of the normal applications. Unlike the normal applications, SandDroid reported 100 as the risk scores for all the tested malicious applications.

Significant changes on the Android application package (APK) file after being repackaged with our backdoor could be an important clue to detect the backdoor. To analyze this, we repackaged a normal application using Firebase with our backdoor implementation and compared the differences between before and after the repackaging in terms of the following aspects: required permissions, activities, services, receivers, providers, intent filters, contained files and file size. As a result, no difference was observed except the increases in the APK file size and the number of files contained in the APK file; specifically, the number of contained files has increased by 8 and the APK file size has increased by about 1.5 Mbytes due to the backdoor's codes and dictionary files.

4.2 Measuring execution time

To measure the execution time, we conducted both dictionary and brute force attacks for the two lock mechanisms. We used the 4-digit PIN dictionary [8] and the pattern dictionary established by the 3-gram Markov model [11]. To avoid the selection bias, we randomly selected 5,000 password samples from real PIN and pattern datasets, respectively. We measured the execution time taken from our backdoor implementation after receiving an "Open sesame" message (i.e., the time taken from step 6 to 8 in Figure 1).

As shown in Table 1, for 4-digit PIN dictionary attacks, 52.68% was cracked within 1 second while 10.60% for brute force attacks. The average execution time of dictionary attacks is 1.52 times faster than brute force attacks. However, for both types of attacks, over 94% were cracked within 5 seconds. For screen lock pattern dictionary attacks, 85.34% was cracked within 1 second, and the average execution time was 1.017 seconds. On the other hand, for brute force attacks, only 44.10% was cracked within 1 second, and the average execution time was 4.93 times slower than dictionary attacks. Our empirical study shows that dictionary attacks are more efficient than brute force attacks, particularly within a short time interval.

4.3 Measuring power consumption

To see the impact of our backdoor on power usage, we measured the power consumption of our backdoor application using PowerTutor [13]. We configured the victim's device to include only our backdoor, PowerTutor and default system applications. We measured the power consumption of our backdoor in three different circumstances; `Idle` represents the case that our backdoor has not yet been triggered, `Changed` represents the case that the user changes the password every time, and `Not changed` represents the case that the user never changes the password. We note that in `Not changed` case our backdoor just pops up a notification of the password cracked before without the need of executing the password cracking procedures. We used the last password in our dictionaries as the victim's password to estimate the worst case execution time.

In Table 2, `Ratio` represents the influence of our attack on battery. To take this, we measured the total power usage of the device for an hour. The measurements in Idle case indicate that our backdoor application consumes no power before triggered. For PIN, `Not changed` and `Changed` results show that our prototype implementation has only a negligible effect on battery while `Changed` results show a

Table 2: Average power consumption (J) taken to run our backdoor application, and the influence ratio (%) of our attack on battery (μ: average, σ: standard deviation).

	Changed		Not changed	
	Pattern	**PIN**	**Pattern**	**PIN**
μ	74.597J	10.393J	0.043J	0.050J
σ	4.493J	1.944J	0.056J	0.050J
Ratio	10.862%	1.669%	0.007%	0.008%

considerable impact on the battery for pattern. We note that it is not common to frequently change user password on mobile devices.

5 Backdoor prevention

In this section, we discuss two defense strategies to mitigate the backdoor.

5.1 Protecting hashed password files with a trusted execution environment

The first defense line against this type of backdoor is to block it from accessing `password.key` and `gesture.key`. In theory, we can easily achieve this with a trusted execution environment (TEE) that is a "secure" processing environment for applications with high security requirements isolated from the "normal" processing environment (e.g., Android operating system) [4].

ARM TrustZone [1] is a suitable candidate to implement a TEE because ARM TrustZone architecture is widely deployed in a majority of existing mobile devices. The hardware-level access control mechanism provided by TrustZone prevents any process in the normal processing environment from accessing the resources assigned to the secure processing environment. Based on this TrustZone technology, we can prevent the backdoor from accessing `password.key` and `gesture.key` files by putting the files and the manufacturer's screen lock app in the secure processing environment.

5.2 Reviewing the history of suspicious events

One possible mitigation strategy is to increase the chance of detecting the existence of a backdoor by reviewing the history of all suspicious events on an Android device. Whenever the device is unlocked and/or a push notification is received, those events are reported to Google or the device manufacturer's server. The device owner can see the summary information about the data reported from his (or her) device by visiting the server website.

6 Conclusions

We presented a design of backdoor that makes an attacker consistently unlock a victim's Android device in a stealthy manner. Our experimental results showed that our backdoor design provides several benefits such as difficulty in detecting backdoor, fast execution time and low power consumption.

In future work, we plan to generalize our backdoor design to construct other types of malware. In fact, our current proof-of-concept implementation was just focused on 4-digit PINs and screen lock patterns to unlock Android devices, but the proposed architecture might work well for other purposes (e.g., botnet).

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1D1A1B03030627), and the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2012-0-00646) supervised by the IITP (Institute for Information & communications Technology Promotion).

References

[1] T. Alves, D. Felton, et al. TrustZone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, January 2004.

[2] A. Boxall. 80% of android phone owners in china have rooted their device. `https://goo.gl/dH4zQZ`, April 2015.

[3] I. Cherapau, I. Muslukhov, N. Asanka, and K. Beznosov. On the impact of touch ID on iphone passcodes. In *Proc. of the 11st Symposium On Usable Privacy and Security (SOUPS'15), Ottawa, Canada*, pages 257–276. USENIX Association, July 2015.

[4] J.-E. Ekberg, K. Kostiainen, and N. Asokan. Trusted execution environments on mobile devices. In *Proc. of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13), Berlin, Germany*, pages 1497–1498. ACM press, November 2013.

[5] S. Grzonkowski, A. Mosquera, L. Aouad, and D. Morss. Smartphone security: An overview of emerging threats. *IEEE Consumer Electronics Magazine*, 3(4):40–44, October 2014.

[6] M. Harbach, E. von Zezschwitz, A. Fichtner, A. D. Luca, and M. Smith. It's a hard lock life: A field study of smartphone (un)locking behavior and risk perception. In *Proc. of the 10th Symposium On Usable Privacy and Security (SOUPS'14), Menlo Park, CA, USA*, pages 213–230. USENIX Association, July 2014.

[7] F. Howarth. Is rooting your phone safe? the security risks of rooting devices. `https://goo.gl/axbkX9`, October 2015.

[8] H. Kim and J. H. Huh. PIN selection policies: Are they really effective? *Computers & Security*, 31(4):484–496, June 2012.

[9] A. Ludwig and M. Mille. Diverse protections for a diverse ecosystem: Android security 2016 year in review. `https://goo.gl/6o4tBf`, March 2017.

[10] I. Muslukhov, Y. Boshmaf, C. Kuo, J. Lester, and K. Beznosov. Know your enemy: The risk of unauthorized access in smartphones by insiders. In *Proc. of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services (MobileHCI'13), Munich, Germany*, pages 271–280. ACM press, August 2013.

[11] Y. Song, G. Cho, S. Oh, H. Kim, and J. H. Huh. On the effectiveness of pattern lock strength meters: Measuring the strength of real world pattern locks. In *Proc. of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI'15), Seoul, Republic of Korea*, pages 2343–2352. ACM Press, April 2015.

[12] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15), Denver, Colorado, USA*, pages 1093–1104. ACM press, October 2015.

[13] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. of the 16th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'16), Scottsdale, AZ, USA*, pages 105–114. IEEE, Oct 2010.

[14] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12), San Antonio, Texas, USA*, pages 317–326. ACM press, February 2012.