Bittersweet ADB: Attacks and Defenses

Sungjae Hwang
KAIST
sjhwang87@kaist.ac.kr

Sungho Lee
KAIST
eshaj@kaist.ac.kr

Yongdae Kim
KAIST
yongdaek@kaist.ac.kr

Sukyoung Ryu
KAIST
sryu.cs@kaist.ac.kr

ABSTRACT

Android devices and applications become prevalent and ask for unanticipated capabilities thanks to the increased interests in smartphones and web applications. As a way to use the capabilities not directly available to ordinary users, applications have used Android Debug Bridge (ADB), a command line tool to communicate with Android devices for debugging purposes. While ADB provides powerful features that require permissions to use critical system resources, it opens a gate to adversaries.

To understand the ADB capabilities and their possible risks, we present various types of attacks that are not easily identifiable using ADB capabilities and device-specific functions. We show that applications using ADB capabilities can modify installed applications, leak private user data, and track phone calls, among other things only with the INTERNET permission on the same device. To protect Android devices from such attacks, we present several mitigation mechanisms including a static analysis tool that analyzes Android applications to detect possible attacks using ADB capabilities. Such a tool can aid application markets such as Google Play to check third-party applications for possible attacks.

Categories and Subject Descriptors

K.6.5 [**Software**]: Security and Protection

General Terms

Security

Keywords

Mobile application; Android; security; ADB

1. INTRODUCTION

The advent of explosive interests in smartphones and web applications [25] have dramatically increased the number of Android applications with unanticipated capabilities [2]. Because Android mobile devices can provide powerful features once provided only by computers, Android developers are extending the application categories to areas limited only by their imagination. At the same time, Android applications access private user data stored in devices to perform security sensitive services such as mobile banking and emails, which makes the security of mobile devices tremendously important. Indeed, attackers become much interested in Android applications leading to a huge number of malicious Android applications [24].

Among various malicious Android applications, a recent Windows malware that attempted to infect Android devices to steal private data uses Android Debug Bridge (ADB) [8] as its main tool [19]. ADB is a command line tool to communicate with Android devices for application developers to debug their programs under development. Even though ADB is originally only for debugging purposes, the powerful features of ADB have attracted application developers to build applications with ADB-level capabilities. ADB provides sweet capabilities to draw users' attention but, at the same time, it opens a gate to adversaries.

To understand possible security risks by exposing ADB capabilities to adversaries, we analyzed ADB capabilities and found that they are very much powerful enough to enable various critical attacks. As a proof of concept, we developed malicious applications that leverage ADB using only the INTERNET permission on the same device. The malicious applications can steal private user data and lauch Dos and overbilling attacks as well. Because the malware requires only the INTERNET permission, it is highly likely that ordinary users cannot identify them as malware.

To defend the Android system from such attacks using ADB capabilities, we present several mitigation mechanisms. We first propose a static analysis tool that analyzes Android applications before executing or even before uploading to detect possible attacks using ADB capabilities. Such a tool can aid application markets such as Google Play to check third-party applications for possible attacks. We also describe ways to control uses of ADB capabilities to only authorized parties or debugging purposes.

This paper makes the following contributions:

- We analyze the ADB capabilities to understand possible security risks of exposing them to adversaries. To our surprise, using ADB completely bypasses Android access control policies with just the INTERNET permission.
- As a proof of concept, we develop malicious applications to concretely show possible attacks on Android applications by leveraging ADB and Android utilities.
- We present mitigation mechanisms to protect the Android system against attacks using ADB capabilities. We develop a static analysis tool that detects such attacks automatically and discuss feasible ways to guide the Android system from such attacks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '15, April 14–17, 2015, Singapore
Copyright © 2015 ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714638.

2. ANDROID DEBUG BRIDGE

ADB is a command-line tool that enables Android application developers to communicate with connected Android devices. The comprehensive description of ADB and its features is available from the Android developers' site [8]. ADB consists of three components: a client running on a development machine, a server running as a background process on the development machine, and a daemon running as a background process on a device. When an ADB client starts, it starts an ADB server process if it does not already exist, and it binds to a local TCP port and listens for commands from ADB clients. Thus, arbitrary third-party Android applications can establish local connections to devices and access ADB capabilities via the ADB server. Indeed, various kinds of Android applications in Google Play utilize the ADB server. Screenshot [16, 26], USB tethering [5], and Remote shell [11] are such applications. Since Android 4.4 that introduced the `screenrecord` command to record the display of devices, more screen recording applications have been developed [14, 12].

Even though ADB is useful for developers to debug their applications under development, it may introduce critical security vulnerabilities. Because the connections between clients and the ADB server are not using any authentication, malware can access the ADB sever like other ordinary applications. Lin *et al.* [18] demonstrated that an Android malware can access ADB utilities by using the local-socket channel. They also showed that a malware can steal users' private data by using the screenshot capability in ADB.

To lessen possible security risks by abusing ADB, Android 4.2.2 or higher protects user devices by displaying a dialog that asks to allow debugging via a computer when a user connects an Android device to the computer. While this security mechanism guarantees that USB debugging and ADB capabilities can execute only when users approve their uses, it is not a firm solution because it delegates security decisions to ordinary users. Moreover, since such a dialog merely asks whether to accept an RSA key without any security warnings, it is very difficult for ordinary users to understand what the RSA key is for and how risky ADB capabilities are.

We have carefully analyzed the capabilities of ADB, and we found that they are surprisingly powerful. While they provide sweet features for Android developers to support advanced functionalities, they make the security of Android applications bitterly vulnerable. Before describing and showing what kinds of attacks are possible using the ADB capabilities and proposing defense mechanisms to such attacks, we briefly explain the ADB capabilities.

The Android system provides various functionalities such as viewing system logs and modifying installed applications to authorized users as binary files stored in the `/system/bin` directory. In this paper, we call such functions *utilities*. While the Linux system also contains utilities in the `/bin` directory, the Android system contains powerful utilities which do not exist in the Ubuntu system. For example, unlike Android, Ubuntu 12.04.01 does not provide any screenrecord utility in the directory. Note that Ubuntu also has a screen recording utility, but it does not provide it by default; a user should explicitly download and install it to the system. Therefore, we believe that Android provides unnecessary powerful utilities that increases security risks.

Because utilities are powerful functions, the system protects them by requiring that only users with the UID or GID of ROOT and SHELL can access them. However, since ADB has the SHELL UID, it can access most of such utilities and by leveraging ADB many resources with permissions which applications do not have are accessible to arbitrary third-party applications. Thus, malicious applications can also access the utilities via the exposed ADB capabilities.

3. ATTACKS USING ADB

We describe malicious applications that can perform various critical attacks using the ADB capabilities.

3.1 Threat Model

We assume that our malware is installed on a victim's mobile device. Because our malware requires only the INTERNET permission, it is difficult for ordinary users to identify it as malware. Indeed, even though Google Play uses the service named Bouncer [20], which analyzes submitted Android applications to detect potentially malicious applications, it could not detect our submitted malware that use ADB capabilities and successfully uploaded it. Note that our fake malware does not send private user data to our server; it merely sends bogus values to our server. Thus, it did not leak any private user data from anyone happened to download it. We also assume that our malware can access the ADB server. While the ADB server is disabled by default, many legitimate Android applications require enabling the ADB server to access unauthorized resources. Various screen capture applications via ADB have been downloaded millions times already [18]. Other applications also use unauthorized resources via the ADB server, which implies that our assumptions are realistic. If any such applications are installed on a mobile device, the ADB server should be already running.

3.2 Enabling the ADB Server

We assume that our malware can access the ADB server, which is reasonable because many legitimate Android applications require enabling the ADB server to access unauthorized resources.

In order to enable the ADB server, an Android device should enable the "USB debugging" option which requires several steps. However, we can simulate these steps to enable the option just with two taps using a UI redressing attack [21]. First, by executing a "Development Setting" activity with an intent set to the option `ACTION_APPLICATION_DEVELOPMENT_SETTINGS`, our malware can navigate to "Developer options". Then, the malware displays a toast [9] in full screen to hide the "Development Setting" activity. Because toasts pass touch events to lower layers unlike other dialogs, we can trick users to touch screen on the toast leading to touch events on the hidden "Development Setting" activity. Our malware asks users to touch two buttons on the toast to enable the "USB debugging" option. To understand whether this attack is effective, we evaluated the UI redressing attacks on several smartphones including Samsung Galaxy S5, LG G2, and LG G Pro, and we found that all the devices are vulnerable to the UI redressing attacks allowing us to enable the "USB debugging" option.

3.3 Representative Attacks Using ADB

In this section, we present three kinds of attacks we created to show the powerful features of the ADB server. The first kind is the *private data leakage* attacks that expose personal data to outside devices, the second kind is the *usage monitoring* attacks that track various usage information, and the third one is the *behavior interference* attacks that prevent intended operations. Because all attacks require only the INTERNET permission to communicate with the ADB server, most of them are background attacks that run silently on devices, and even foreground attacks are not easily discoverable because the attacks can take place very shortly when users do not see the screen, it is difficult for ordinary users to notice them.

3.3.1 Private Data Leakage

Even though the Android system protects private user data in devices by checking application permissions, it is not enough to prohibit private data leakage as we illustrate in this section.

Message Tracking.

While Android requires applications have the RECEIVE_SMS or READ_SMS permissions to access messages like SMS, MMS, and internal messages from the Android system, applications with only the INTERNET permission can access such messages via ADB. Note that most messenger applications, mail clients, and SNS applications like Facebook notify incoming messages to users through *notification bars*. Because the dumpsys utility provides system data including notification information, the malware can read all the messages from notification bars.

Call Tracking.

In the Android system, in order for an application to get notified of phone-call actions, it should register as a broadcast receiver for the actions. For example, if an application registers as a broadcast receiver with the NEW_OUTGOING_CALL action, when an outgoing call event occurs, the Android system broadcasts the event, which notifies the event to the receivers registered with the NEW_OUTGOING_CALL action. However, our malware can catch call actions without registering as a receiver. Using the dumpstate utility via ADB, it can log all the call information by periodically collecting broadcast intents and by choosing only the intents with the call-related actions. Note that because not only making calls but also sending messages, sending emails, pressing the camera button, and more actions can trigger broadcasting intents, similar approaches would be applicable to track such actions.

Private Database Access.

While Android allows applications to use private databases without sharing them with other applications, our malware can access private databases of other applications using ADB and the run-as utility. The run-as utility changes the current UID and GID to those of a specified application. As the UID and GID are changed to the specified application, malware can have the same permissions as the specified application, and thus it is able to access the private database of the specified application. Even though the run-as utility works for debuggable applications only, debuggable applications are not insignificant. We have collected 8,870 applications from Google Play from February 2013 to August 2013, and we found that 269 applications are debuggable. Due to these security issues, Google Play rejects debuggable applications from November 2013, but debuggable applications already uploaded to Google Play are still available to users unless they have been updated. Moreover, other Android markets than Google Play still accept debuggable applications for uploading.

SIM Information Leakage.

Most smartphones require the SIM card information to connect to and communicate with networks, and SIM card contains a variety of private data such as a phone number, IMSI (International Mobile Subscriber Identity), ICCID (Integrated Circuit Card Identifier), IMEI (International Mobile Equipment Identity), and SPN (Service Provider Name). Because adversaries can utilize phone numbers to infer user identities, the Android system requires applications have the android.permission.READ_PHONE_STATE permission to obtain phone numbers. Also, adversaries have shown various attacks that impersonate users by using the IMSI information [17]. However, our malware can obtain the SIM information by using the dumpstate utility via ADB even when it does not have the READ_PHONE_STATE permission. Because the dumpstate utility provides the SIM information without asking for a user's confirmation, adversaries can perform the attacks silently.

3.3.2 Usage Monitoring

The Android system and applications keep several logs for debugging or other purposes. A kernel driver called "logger" [1] in the Android system records the main application logs, system event information, phone-related information, and low-level system messages, which contains most usage information. Even though the logs are inaccessible by user applications in general, our malware can access them via ADB. We show only two attacks that monitor network packets and users' key events in this section but other usage information is also accessible via ADB.

Packet Dump.

Because data exchanges of mobile devices over network contain users' usage information and they may also contain sensitive information, dumping 3G/4G data packets is not possible unless phones are rooted. If adversaries can dump data packets, they can steal credential data and they can also track users' private activities such as webpage visits. To our surprise, we found that some smartphones provide powerful capabilities including network packet dumps in a hidden menu. Using the ADB server with the input utility, our malware can access the hidden menu and leverage hidden functions there even without rooting devices. The "Packet log" function in the hidden menu is disabled by default but once it is enabled, it captures all the exchanging packets. Such captured packets are stored in SD Card and adversaries can retrieve them even without the required READ_EXTERNAL_STORAGE permission.

Keystroke Logging.

When a user touches the screen of a mobile device, a key event occurs including the key action and the location of the touched point. If an application running in background silently monitors all the key events occurring on a device, it can steal credential information without being noticed by users. Zhou *et al.* [29] developed a touch screen keylogger by reading the Linux input driver files such as /dev/input/event3. This attack is no longer possible as this vulnerability has been patched. However, by leveraging the getevent utility via ADB, our malware can read input driver files.

3.3.3 Behavior Interference

We presented attacks that do not interfere with the functionalities of applications so far, and now we describe attacks that abuse devices like modifying applications and locking screens.

Overbilling.

While applications must have the SEND_SMS and CALL_PHONE permissions to make phone calls and SMS, respectively, our malware can perform an overbilling attack by using the am and input utilities via ADB. First, the malware launches a system application to send an SMS by issuing the following command:

```
am start -a android.intent.action.SENDTO
         -d sms:<Phone Number>
         --es sms_body "<SMS Text>"
         --ez exit_on_sent true
```

which starts an SMS activity with a receiver's phone number and an SMS content. Then, it issues the following commands in order:

```
input keyevent 22; input keyevent 66;
input keyevent 3
```

where the first moves the current focus to a Send button, the second presses the button, and the third changes the current screen to the home screen. Similarly for MMS, the malware can also make extra billing by making many phone calls .

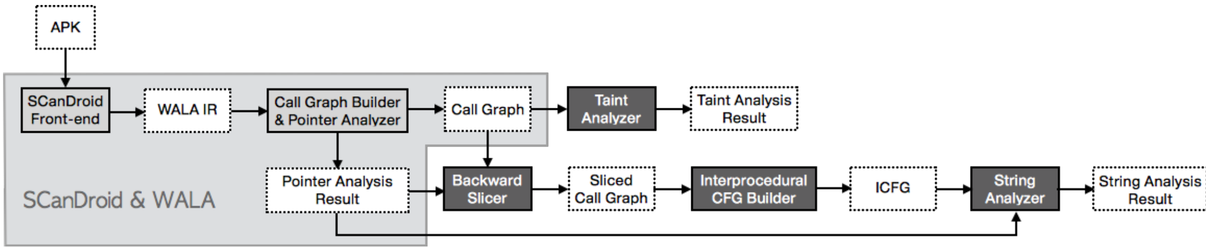

Figure 1: Static analyzer that detects malicious applications using ADB

Application Modification.
The Android system protects applications by requiring developers sign the applications with their private keys. Therefore, only the developers of the applications should be able to modify them for updates, and updates should be the only way to modify installed applications. Of course, the Android system should reject any modification to applications without the corresponding private keys. However, by using the `pm` utility via ADB, our malware can replace installed applications with fake applications without notifying users. For example, the malware can first uninstall the original Facebook application and re-installs a fake Facebook application that contains malicious functionalities. Once the fake Facebook application is installed on a mobile devices, various private data such as a list of friends and messages exchanged via the Facebook messenger can be leaked via the fake application.

DoS Attack.
We can make a variety of DoS attacks on Android using ADB, and here we present two such attacks: target application killing and screen resizing. By using the `wm` utility via ADB, our malware can repeatedly kill target applications. First, the malware checks whether a target application is installed on a mobile device by using the `PackageManager` class. If the target application is installed on the device, it keeps killing the application every 5 seconds to prevent users from using it. Also, our malware can modify the screen size of a device using the `wm` utility via ADB as follows:

```
wm size <Width> X <Height>
```

By setting screen sizes too small or big, the malware can prohibit users from using their devices. Note that no methods can restore screen sizes and even rebooting devices cannot restore the sizes.

4. DEFENSES AGAINST ATTACKS USING ADB

In this section, we present mitigation mechanisms to protect the Android system against attacks using ADB capabilities.

4.1 Defense without Changes to Android

As a mitigation mechanism that does not require any changes to Android, we propose a static analysis to detect possible attacks using ADB capabilities. Based on our analysis of Android applications, we identified representative attack patterns using ADB as illustrated in the attacks described in the previous section. To automatically identify such attacks, we designed and developed a static analyzer that performs a *string analysis* and a *taint analysis* as presented in Figure 1. After describing the high-level architecture of the static analyzer, we present each of its components in detail.

Static Analyzer Overview.
The main purpose of the static analyzer is to extract a set of commands sent to the ADB server by a string analysis, and to check whether the private data resulted from performing such ADB commands are delivered to outside of Android devices via socket APIs. We built the analyzer on top of the state-of-the-art analysis frameworks, WALA [13] and SCanDroid [7]. Our analyzer uses the front-end of SCanDroid to take an APK file consisting of Dalvik bytecode as its input and to translate it to WALA Intermediate Representation (IR). Then, it performs base analyses like a “class hierarchy analysis” and a “pointer analysis” using the WALA analysis capabilities to aid our main malware detection analysis.

Backward Slicing.
Before the main string analysis, our analyzer first performs backward slicing to reduce the size of its analysis target. If we can extract a subset of an input program that contributes to some target variables at a program point under consideration, analyzing the subset would be more efficient and precise than the entire program.

While a traditional backward slicing technique works quite well for tracking primitive values, tracking the values of string objects as in our analysis is not trivial. Because a string object has a character array, backward slicing of string objects requires backward slicing of array objects, which tracks the construction sites of array themselves instead of their elements. To perform a backward slicing of array elements as well, we extended the traditional backward slicing with modeling of methods in `String` and `StringBuilder` classes. When the backward slicing encounters methods that create values for character array elements, it keeps track the flows for array elements by using the modeling behavior.

Furthermore, for Android 3.0 and higher, network operations should execute in different threads from the main thread [10]. While the backward slicing works well for single threaded programs as being extensively used for Java program analyses [27], it may not be suitable for multi-threaded code like Android applications [25].

When a backward slicing for single threads cannot track data flows in a given thread, we extend the backward slicing to consider data flows in other multiple threads. Using the pointer analysis result from WALA, it first collects all the possible objects that target variables may have and collect the instructions that assign values to the target variables in any threads. Performing such an inter-thread slicing repeatedly until it reaches a fixed point achieves a backward slicing considering communications between multiple threads.

String Analysis.
The string analysis takes an ICFG built by the backward slicing as its input and estimates a set of possible string values for the parameters of `write`. To address the asynchronous communications between multiple threads, it performs two string analyses: a flow-insensitive analysis for threads not using network operations and a flow-sensitive analysis for threads using network operations. In order to over-approximate string values on a shared memory between multiple threads, it first performs a flow-insensitive string analysis to estimate possible values of the variables on the shared memory.

Then, using the analysis results as the initial state of the shared memory, it performs a flow-sensitive string analysis to estimate possible string values of the parameters of `write`. Our string analysis provides sound results for asynchronous execution of multiple threads. Because the flow-insensitive string analysis result contains all the possible values of the variables on the shared memory, it soundly estimates concrete values constructed at run time. Also, because the flow-sensitive string analysis builds on top of the flow-insensitive string analysis, and it tracks data flows between multiple threads, the final string analysis results are also sound.

Taint Analysis.

In addition to detecting string values leaked through the `write` function, our tool also tracks value flows from the results of the `read` function to the parameters of the `write` function via taint analysis. Because private user data received from the ADB server via `read` may be leaked through `write`, we perform a taint analysis with the `read` calls as sources and the `write` calls as sinks. Similarly for the string analysis, we extend the traditional taint analysis to track data flows between asynchronous multiple threads. Unlike static analysis which requires precise string values that particular variables may have, taint analysis simply tracks value flows between sources and sinks. Thus, the taint analysis performs only a flow-insensitive analysis which provides sound analysis results. If we want to improve the taint analysis results, we can apply the same approach as the string analysis and use two kinds of taint analyses.

Evaluation.

Because we proposed new attacks in this paper, we evaluated the efficiency of our static analyzer using all 7 applications that leak string values to the `write` function, and we evaluated the taint analysis using 2 malicious applications described in Section 3 that leak information to outside of devices. For the former, we tested such cases where string values flow intra-procedurally, inter-procedurally, via fields of classes, and via static fields of classes. For the latter, we tested malicious applications for application modification, SIM information leakage, and message tracking. Our static analyzer correctly detected command strings that are sent to the ADB server. The analyzer could detect data flows from the ADB server to the outside from the malicious applications.

4.2 Defenses with Changes to Android

We propose mitigation methods that require changes to Android.

Informative Message for Using ADB.

Starting from Android 4.2.2, the Android system protects ADB by asking a user's confirmation to use the USB debugging capability. When trying to enable USB debugging, the Android system show a dialog including a RSA key to a user to approve USB debugging. However, RSA keys may not be useful information to ordinary users who are not familiar with security vulnerabilities. To help ordinary users understand possible security issues resulting from allowing USB debugging, more informative message than RSA keys should be provided.

Automatic Disable of USB Debugging.

While still allowing the ADB capabilities to ordinary applications, the Android system may turn them off automatically. Even though USB debugging is disabled by default, once it is enabled, it remains to be enabled even after the system reboots. Instead, disabling USB debugging by the system periodically is much more secure. The BlackBerry system indeed turns off the USB debugging option after a certain amount of time [3].

Restricted ADB Functionalities.

Because ADB capabilities are vulnerable to many attacks as we showed in the previous section, restricting ADB functionalities may be a plausible option. One way to restrict their uses is to prohibit combined uses of multiple ADB commands. While this restriction does not preclude all the attacks we described in this paper, it can rule out many attacks like contact collection using `screenrecord` with `input` and overbilling using `am` with `input`. More rigid approach is to disallow ADB capabilities from production applications. This restriction goes back to the original intention of ADB, which is only for debugging purposes. We have compared the debugging utilities of the Android system and Ubuntu 12.04.01, and we found that Android provides more debugging utilities by default than Ubuntu. Restricting ADB capabilities in production applications will surely limit the application functionalities but it will guide them more securely.

Secured ADB Channels.

Even though the Android system displays a dialog with an RSA key for allowing USB debugging, the system does not authenticate the server and the client. Once the ADB server is running, any applications can connect to the server via a TCP connection. We propose several approaches to guide the communication with the ADB server securely. The most straightforward approach is to add an authentication mechanism to the ADB server. For example, the ADB server may allow connections only from the shell by checking clients whether their UID are 2000. Another possible solution is to create a new ADB permission and to modify the system so that only the applications with the ADB permission can communicate with the ADB server. Also, if the ADB permission is protected with the level of the signature permission, ordinary applications including malicious ones cannot leverage the ADB server any more.

5. RELATED WORK

Recently, various attacks in Android applications have been reported. These attacks leverage vulnerabilities in intents [4], dynamic code loading [23], content providers [30], permission escalation [6], and advertisement libraries [22]. In this paper, we reported a variety of attacks that leverage one of the Android system component, ADB. Because ADB is originally for debugging purposes, the Android system assigns higher privileges to ADB than to third-party Android applications. Thus, we could present powerful attacks of various kinds by leveraging ADB and its utility functions only with the INTERNET permission.

Even though protecting system components is critical due to their high privileges, only a few work have focused on the Android kernel layer security. Zhou *et al.* [29] pointed out that vulnerable Android device drivers can leak users' private data. Moreover, Jana and Shmatikov [15] demonstrated that examining shared memory can leak private information. Similarly, investigating public information can lead to private data leakage [28]. Compare to these work, our research focused on understanding security risks of ADB and the Android system utilities. The most closest work to ours is Lin *et al.*'s work [18], which studied security risks of the screenshot function using ADB. While Lin *et al.* focused on the screenshot function to build a malware using ADB, we presented how powerful ADB capabilities are and showed their security risks by presenting various kinds of attacks. Given that a malware leveraging ADB to launch attacks has appeared [19], we believe our study on ADB capabilities would be useful to understand their security vulnerabilities.

6. CONCLUSION

In this paper, we demonstrated that by leveraging ADB capabilities, malicious applications can leak private data, monitor device usages, and even interfere with device behavior. For private data leakage, we showed that malicious applications can track messages exchanges, call history, and geographic locations, record screen snapshots, access private database of other applications, and leak SIM information. To demonstrate usage monitoring attacks, we presented packet dump and keystroke logging attacks. Finally, we showed that malware can disturb users by overbilling and modification of applications, and they can even prohibit users from using their devices by DoS attacks and locking device screens. To our surprise, all these attacks are possible only with the INTERNET permission.

To protect Android users from such attacks, we presented multiple mitigation mechanisms. We developed a static analysis tool that detects potential malware leveraging the ADB server. Instead of simply displaying a dialog with an RSA key to protect the ADB server as the current Android system does, a more informative message would be helpful for ordinary users to be aware of security vulnerabilities. We strongly believe that ADB capabilities should be restricted or they should use secure channels; if possible, ADB should be used only for debugging purposes.

Acknowledgment

This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2014R1A2A2A01003235 and NRF-2008-0062609).

7. REFERENCES

[1] Android logging system. http://elinux.org/Android_Logging_System, 2012.

[2] AppTornado GmbH. AppBrain: Number of Android applications. http://www.appbrain.com/stats/number-of-android-apps, 2014.

[3] BlackBerry. Blackberry developer. http://developer.blackberry.com.

[4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011.

[5] ClockworkMod. ClockworkMod tether (no root). https://play.google.com/store/apps/details?id=com.koushikdutta.tether, 2013.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, 2010.

[7] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, 2009.

[8] Google. Android debug bridge. http://developer.android.com/tools/help/adb.html.

[9] Google. Toasts. http://developer.android.com/guide/topics/ui/notifiers/toasts.html.

[10] Google. NetworkOnMainThreadException. http://developer.android.com/reference/android/os/NetworkOnMainThreadException.html, 2014.

[11] C. Gutman. Remote ADB shell. https://play.google.com/store/apps/details?id=com.cgutman.androidremotedebugger&hl=en, 2013.

[12] Hiandroidstudio. No root screen recorder-trial. https://play.google.com/store/apps/details?id=com.screenrecnoroot&hl=en, 2014.

[13] IBM. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[14] Invisibility. Free screen recorder no root. https://play.google.com/store/apps/details?id=uk.org.invisibility.recordablefree&hl=en, 2014.

[15] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[16] E. Kim. No root screenshot it. https://play.google.com/store/apps/details?id=com.edwardkim.android.screenshotitfullnoroot, 2013.

[17] D. F. Kune, J. Koelndorfer, N. Hopper, and Y. Kim. Location leaks on the GSM air interface. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.

[18] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your Android screen for secrets. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.

[19] F. Liu. Windows malware attempts to infect Android devices. http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices, 2014.

[20] H. Lockheimer. Android and security. http://googlemobile.blogspot.kr/2012/02/android-and-security.html, 2012.

[21] M. Niemietz and J. Schwenk. UI redressing attacks on Android devices. In *Black Hat Abu Dhabi*, 2012.

[22] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[23] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.

[24] E. Protalinski. Android malware numbers exploded to 25,000 in June 2012. http://www.zdnet.com/android-malware-numbers-explode-to-25000-in-june-2012-7000001046, 2012.

[25] J. Rivera and R. van der Meulen. Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. http://www.gartner.com/newsroom/id/2665715, 2014.

[26] SmartUX. Screenshot UX. https://play.google.com/store/apps/details?id=com.liveov.shotux, 2012.

[27] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27:631–661, 2005.

[28] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from Android public resources. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[29] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in Android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[30] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in Android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.