Security Enhancement of Secure USB Debugging in Android System

Mingzhe Xu, Weiqing Sun, and Mansoor Alam
The University of Toledo, Toledo, Ohio 43606, USA
mingzhe.xu@rockets.utoledo.edu, {weiqing.sun, mansoor.alam2}@utoledo.edu

***Abstract*— The security of Android Debug Bridge (ADB) has attracted much attention from researchers, because it has a high privilege level and a low level of protection. Many attacks on Android systems have taken advantage of the security holes of ADB. Thus, in the updating patch of Android 4.2.2, a security feature secure USB debugging was implemented so that only trusted hosts can use ADB. Our research analyzes its protection effects on ADB based attacks and found that the new feature cannot provide sufficient protection when the host used to connect with Android devices has been compromised. A demonstration attack following this method is given along with an improvement design of the security mechanism of USB Debugging Mode. An implementation of this design and its evaluation are also provided to demonstrate its effectiveness.**

Keywords—Android Security; Android Debug Bridge; Secure USB Debugging; Smart Phone Security.

I. INTRODUCTION

Debugging is an important procedure in software development workflow. It is the process of detecting and fixing errors in the software under development. During the debugging process, much software information is traced by the debugger, such as memory allocation, API usages and subroutine calling stacks. Although so many details of the software can be fetched, the debugging process itself is generally not deemed as a dangerous process. That is because in general systems, debugging functions always come with a software development kit (SDK) for an integrated development environment (IDE), which is provided by trustworthy third parties or from the system's designers. Thus, only selected systems can run in the debugging mode to let applications be debugged on them. Usually those systems for debugging are restricted by extra security policies, and are not worthwhile to hack into them for the lack of important data.

Android is an operating system (OS) designed for mobile/portable devices [6]. In mobile devices, hardware capability is highly limited, and Android is designed for multitasking based on limited system resources. It is developed based on a Linux kernel with multiple virtual machine processes, called Dalvik, running to support its multitasking feature. Because of this lower performance (compared with non-portable computers), one important characteristic of Android is that, most Android developers usually develop their Applications (APPs) in an IDE running on regular computers, and compile them and then send the APPs to the Android environment for debugging [7].

To run and test the APP, the USB Debugging Mode must be turned on in Android devices, which is disabled by default. Once an Android device is connected to a computer with a USB cable, the USB Debugging Mode can be enabled. This mode can authorize the device to establish a connection between an Android device and a computer using the Android Debug Bridge (ADB) utility [1]. It also allows the computer-end software (primarily SDKs and IDEs) to read the debugging information of the tuned APP.

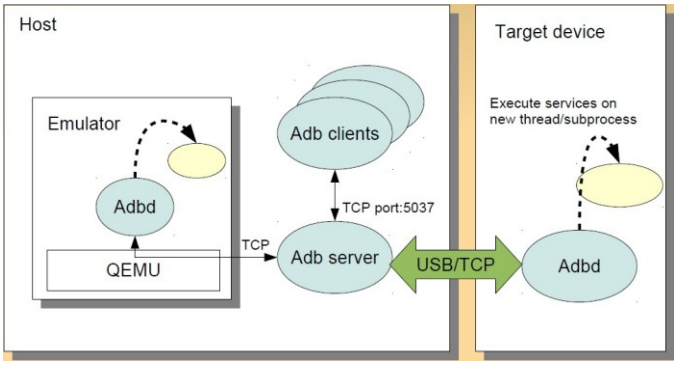

Fig. 1. Structure of Android Debug Bridge [12].

Android designers provided ADB to help developers easily connect to the Android device from their development machines to debug APPs. The structure of ADB is shown in Figure 1. As a default component of Android, the ADB has three parts that separately run on the host (desktop/laptop computer) and the device (Android testbed). The ADB on the device is a daemon process which receives commands from the host, executes them and returns the results. The host part of ADB is composed of the ADB server and a command line client. The major functionality of the ADB server is to monitor the connection between the host and the device. The command line client is an interface for getting user commands and sends them to the device via TCP/IP or USB connection [12]. This paper will discuss the security of the ADB, with the primary focus on the new security feature of ADB, secure USB debugging, which was introduced in Android version 4.2.2 [3], and allows only authorized hosts to use ADB. This feature will undoubtedly enhance the security of ADB and the overall Android system. However, a detailed analysis of the feature is needed in order to understand its functionality, effects and drawbacks. In this paper, we perform a thorough analysis. Through the analysis, we identified a security issue that cannot be fully addressed by this feature. Therefore, it is possible that potential attacks can be launched by exploiting it. We

designed, implemented and evaluated such an attack. We also propose a solution to improve the ADB security and fix the problem.

The structure of the remainder of this paper is as follows. Section II describes the background of our research, including the overview of Android debugging function, and its security issues. Section III explains the major security considerations about Android debugging and introduces the secure USB debugging feature developed in Android version 4.2.2. Then we analyze this feature and discuss its security effects. The security hole we have found is also described in Section III. In Section IV, we use a simple intrusion to demonstrate the attacks which can exploit this security hole. Then, in Section V, the security enhancement is described along with the evaluation. Finally, Section VI concludes our research and presents the future work.

II. BACKGROUND

A. Privileges of USB Debugging Mode

In USB Debugging Mode, users are granted special permissions to facilitate the actions required during the debugging process. As source code must be modified frequently, the APP is sent from computer's IDE and installed on the device. The debugging mode let users install/uninstall APPs freely, without the restrictions applied to the normal Android installations. Users are allowed to trace the APP's actions such as APIs called, memory allocated, and system settings influenced. The activities of an APP can be tuned manually in the USB Debugging Mode. Also, there are special commands which can only be executed in this mode.

Basically, the realization of the above privileges depends on ADB utilities and the communications through the USB cable connection. All the debugging functionalities can only be possible through the connection established between the ADB server and the client. By using ADB, users can directly install APPs to the connected device, and the Android system will automatically give APPs all the permissions they need, without any further security check. The debugging information is collected by the Android system via the ADB connection. The special utilities will be functional when they receive commands issued from ADB. In addition, the ADB can act as a terminal client to the Linux kernel on Android devices.

B. Sensitive Commands of ADB

ADB provides a wide range of functions for the interaction between the host and the device. These functions are usually executed by typing commands in a command-line interface on the host. Some of the commands are security sensitive. For usability reasons, a very high privilege level is given to ADB. Once an Android device is connected with a host through ADB, all commands can be directly executed without any further authorization.

The ADB command "*install*" can enable users to install a new package to the system. That means if an attacker subverted a device through ADB, he/she can silently install malicious APPs to it, and those dangerous APPs will then be granted all the permissions they need from ADB. In addition, the "*push*" and "*pull*" commands can transport files between the host and the device. They can send files to and get files from any directories on the device, either in the system storage or the memory card.

Both Activity Manager (command: "*am*") and Package Manager (command: "*pm*") are parts of the ADB shell functionalities and their commands start with "*adb shell*". By using the activity manager, an ADB user can control all the activities within the system, for example, initialize/stop an activity, specify the activity's action, and start/stop a background process. Activity is the basic component of Android APPs. All system actions, such as open/exit applications and modify system settings, will be under the user's control if he/she is using the Activity Manager. Package Manager manages application packages on the device. It can enable users to perform a series of package management operations, such as querying system information, listing installed packages with filter functions, installing/uninstalling packages, revoking/granting permissions to APPs and creating/removing users.

In order to make the debugging process easier, a shell functionality "*screenrecord*" is provided in the Android 4.4 update. Like Activity Manager and Package Manager, the "*screenrecord*" command also needs to be executed in the ADB shell environment. As the name indicates, it can record a video of the Android device's screen for up to three minutes. The generated video file is in the mp4 format and will be stored in a selected path specified by the user.

In addition, we found some ADB utilities that are not included in the Google's Android documentation website. One critical function of them is the "*tcpip*" command. This utility is used to restart the device's ADB daemon to enable it to listen for the TCP/IP requests on a specified port. By using this function, a host can connect to a device's ADB utility via a TCP/IP network. Compared with the USB connection, it is more flexible with less restriction.

C. ADB Security Issues

By taking advantage of ADB connections, users could install APPs with any permission they want. Those permissions are the basic access control components in Android. APPs with all permissions are granted full access to the whole system. It can read/write any file in the storage, control voice calls and SMS messages, change system settings, and read all the account information on the device. If the ADB feature is used by a person who is not the owner of the device, it may result in severe personal information leakage. Through the ADB connections, users are also able to input commands to the Linux kernel. Several methods are available for obtaining the root user privilege. Users will take full control of the system if the device is "rooted", and some of the rooting methods are completed by interacting with kernel through ADB.

With ADB, high privileged operations can be performed to control the Android device. Therefore, it has become an attractive attack vector. There have been a number of attacks against Android systems by exploring ADB's security hole. In recent years, many security holes and potential problems caused by ADB have been found.

DroidDream [5] first appeared in spring 2011, and the attack can send malicious software to the Android system by installing a rootkit to the device. This rootkit installation is accomplished via a resource exhaustion attack on the ADB [9]. As the result, the DroidDream malware will gain root access to the device and automatically download more malicious software if the rootkit is installed. In [10], a number of remote control functions were realized by using ADB features such as installing and uninstalling applications, downloading and uploading files, opening a shell console, and starting applications. A framework for on-device privilege escalation exploit execution on Android was discussed in [11]. This is an ADB security hole that can let Android APPs escalate their privileges and obtain root privilege of the system (Android rooting). The privilege escalation process is accomplished via ADB connections. This issue was found by two researchers from Upper Austria University of Applied Sciences. Super One-Click [9] is a desktop application running in the Windows environment. It helps users to root their Android devices by only "one click" on the computers. This rooting process also takes advantage of ADB, which requires users to enable the device's USB debugging mode and establish USB connection with the host [8]. Once the device was rooted, all the installed software could gain super user privilege, even the malicious ones. The research in [13] described two possible attack methods using ADB. The first attack established an ADB connection with root privilege by flashing the targeting device's memory to change the ADB daemon's parameters. The second method switched one Android system's ADB daemon into host mode, and then used its ADB utility to connect to and control other Android targets.

III. SECURE USB DEBUGGING

The security on the USB debugging mode has not been improved for years. The only enforcement provided by designers before the new feature came out is in the updating patch 4.2 [4]. It hides the USB mode enabling option from the system setting menu. In order to make the option checkbox visible, a user needs to touch seven times on the "Build Number" section under the "About Phone/Tablet" menu in system settings. The purpose of this design is to prevent users from accidentally turning on the USB debugging mode. However, it is hard to be deemed as a security improvement.

Thus, in the 4.2.2 update, designers introduced the new security feature on ADB called "secure USB debugging". In this solution, only authorized hosts are allowed to use the USB connection with the device. If a device is to be connected to an unauthorized host, the host will not see files on the device and cannot establish an ADB connection to the device. The authorization process is completed by the device administrator when a new host is connected for the first time. After activating the device, a prompt dialog will ask the administrative user to confirm the authorization. Once the confirmation is made, the device will communicate normally with this host. If the administrator chooses to always allow this host, it will be added to the white list of the device and no authorization will be required when subsequent connections occur [3].

A. *The Implementation of Secure USB Debugging*

In the design of secure USB debugging [2], a 2048-bit RSA encryption function is used to secure the authentication process. The host's private key and public key are generated from its desktop ADB server utility. The host's public key is treated as the host identity. When the device is connected to a host, the device sends a 20-byte random message to the host. Then the host encrypts the message's SHA1withRSA signature using its private key and sends this encrypted signature back. The device will decrypt this signature and compare it with the signature calculated from its original message. If the verification fails, the device will stay in the offline status and cannot perform any action on the device. The verification failure may be a result of no corresponding public key on the device or the two signatures do not match. If the device does not contain this host's public key in its storage, the host will send its public key to the device first. After receiving the public key, a confirmation dialog shows on the device with the key's MD5 hash waiting for user actions. If the user selects "OK", then the device will use this key to decrypt the verification message. If the "Always allow from this computer" checkbox is checked, the device will save this key in its storage drive.

B. *Potential Problems*

This method provides a white list design for ADB connection authorizations for the sake of better usability and security. It does not require many user actions to authorize a host; instead, just a simple click would suffice. This security design helps to protect Android systems from malicious attacks originated from ADB connections. All the unknown hosts are not allowed to connect to the protected device. Some attacks begin with establishing the ADB connection with the Android system and then try to obtain administrator privileges by sending and installing "rootkit" to the system via the ADB connection. Now these attacks cannot work anymore because they are not allowed to connect to the target Android devices. The hosts that initiated the resource exhaustion attacks would not be able to establish the ADB connections with target devices. This kind of attacks is thwarted at the beginning. Similarly, secure USB debugging could also provide good protections in both the remote control [10] and privilege escalation [11] attack scenarios.

However, sophisticated attacks might still be able to bypass the above protection. Assume there is one piece of desktop software which can help manage the Android system by providing attractive auxiliary functionalities. This software requires users to connect their devices to the computers using the USB port, and it needs the user to enable the USB debugging mode. But the software contains malicious code that can take advantage of the ADB utility on the hosts and damage the device through the ADB connection. Under this situation, because the user enables USB debugging and authorizes the host to establish the ADB connection, the attack can certainly bypass the secure USB debugging protection. The only thing the software needs to do in order to bypass secure USB debugging is phishing. If a user decides to "trust" the phishing software, the secure USB debugging policy will not be able to provide any protection.

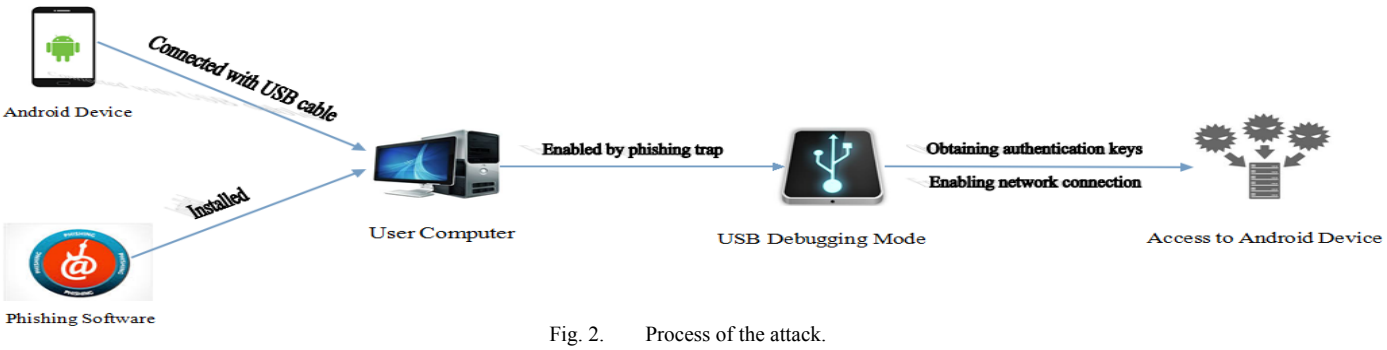

Fig. 2. Process of the attack.

For another similar but more dangerous situation, if a trusted host (in the white list of the device) is later hacked, the secure USB debugging will be unable to protect USB connected devices against the malicious operations from the subverted host. The host is intruded and as the result the hacker could obtain the file storing the private key. Then this private key can be installed in other fake hosts to bypass secure USB debugging. If a fake host wants to connect using keys stolen from a trusted host, secure USB debugging will not block it as it has an authenticated host key.

IV. Demonstration Attack

This Section describes an intrusion into an Android device from a host which is on the device's secure USB debugging white list. We assume that there is a hacker who wants to launch the initial attack to random targets. The goal of this attack is to steal the private key and public key of the host, and then configure the device's ADB daemon to listen to the TCP/IP connection. After that, the intruder will have the ability to establish the ADB connection with the device whenever he/she wants. The connection can be initiated either from the hacked host or from other unknown hosts owned by the intruder. Figure 2 demonstrates the process of the attack.

The attacker first developed a computer application, which can provide some auxiliary functionality to help users to manage their Android devices, such as installing new APPs, transferring files between the SD card and the computer hard drive, and managing multimedia files on the system. To achieve those functionalities, the USB debugging mode must be turned on when a device is plugged in a USB port on the computer. This software also can run some hidden scripts in the background without users' knowledge.

Then some user downloaded and installed this "malicious" software. When this user first runs this software, it will prompt the user to enable its USB debugging mode. It also requests the user to add the connected host to the white list. To take advantage of features provided by the software, the user will likely do what he/she was asked to do. Then the next step is done by executing a short batch script in the background.

We developed the script which can find the files containing the private and public keys, using variables to store them and print them out. This script can be executed in MS-Windows system. A few simple modifications can be done in order to port the script to run under UNIX/Linux or other OS environments. For the folder containing the key files, different configurations may be applied in practice, including $ANDROID_SDK_HOME/.android, $ADB_VENDOR_KEYS, C:\Windows\System32\config\systemprofile\.android, etc. [2] Here we simplified the key files searching process and used the default system setting.

Then the command ("*adb tcpip 5555*") will be executed to restart the ADB daemon on the device and set it to listen to TCP/IP requests on port number 5555, which is the default port for the ADB TCP/IP daemon. The attacker can change it to any port by setting the number in the command. During the restarting process, the USB connection will be reset once. In most cases it would not be noticed or would be ignored by being treated as an unstable connectivity issue.

After the above process, the hacked host is able to connect to the device whenever they are in a same LAN. If the device gets a WAN IP address, in case of connecting to a public WI-FI, the attacker will be able to connect to the device remotely using a fake trusted host with the stolen authenticated keys. One major feature of the intrusion through ADB is its invisibility [9]. Operations from the intrusion will go unnoticed by the users on the target device for a long period of time. Even when the user finally discovers the intrusion on the host and blocks it, the attacker might still be able to intrude the device remotely later.

Once the attacker takes control of ADB, he/she could do more damages to the target. For example, the attacker can install malicious software with all permissions for an Android APP, using it to steal owner's private information, get or remove all files in storage, or delete installed applications. Furthermore, the attacker is able to use ADB to generate a backup file of the whole system, transfer it to local and fetch all the information from the targeting device.

V. Security Enhancement

The previous two Sections show that the secure USB debugging design lack abilities in defending against attacks from subverted trusted hosts. Thus, improvements to address this security issue are needed. Here we design and implement a new approach to enhance the USB debugging security.

A. Design and Implementation

In our approach, the security critical ADB operations will be made visible to the Android system users to enhance the security. As mentioned before, it is hard for the user to notice once an attack through the USB debugging mode occurred. However, this kind of attack is easy to be defeated by simply

disabling the USB debugging mode. Thus, this approach focuses on increasing the transparency of ADB operations, so that users can monitor what is happening with the USB debugging mode and take actions accordingly. Once an ADB connection is established, the Android device will display real-time messages about what the ADB connection is doing. With this, users can see the list of ADB commands being executed on their devices, including those commands that are not issued by them if an attack via the ADB connection is in progress. To minimize the possible user interruptions, the prompt messages are only applied to a list of security sensitive operations as summarized in Table I, so that the number of prompt messages will not be too overwhelming for the users, especially when a normal debugging operation is taking place.

TABLE I. MONITORED ADB OPERATIONS

Command	Description
install <path-to-apk>	Installs an Android application to the device.
pull <remote> <local>	Gets a file from the device to local host.
push <local> <remote>	Sends a file to the device.
shell	Starts a remote shell in the connected device.
shell [shell command]	Issues the shell command to the device.
shell am [command]	Issues the activity manager command to the device.
shell pm [command]	Issues the package manager command to the device.

For "*shell [command]*" operation, many commands can be executed and some of them are frequently used in the normal debugging process. Hence, we use a white-list method to filter the shell commands to be displayed. Commands not on the white list will be shown to the user when they are executed. Table II shows the commands on the white list. These shell commands are often used by Android debugging tools like the Dalvik Debug Monitor Server (DDMS).

TABLE II. SHELL COMMANDS ON THE WHITE LIST

Command	Description
shell echo	Shows a string on terminal.
shell getprop	Gets stored property settings of the system.
shell ls	Lists the content of a directory.

To implement this design, we developed a new component for the Android system called ADB Action Monitor. This monitor module runs as a background process of the Android system. It uses the log function of the ADB utility to trace the ADB operations. The log function generates records about the current ADB operation. This process keeps monitoring the log output file of ADB. Every time the file is appended, ADB Action Monitor will check the newly appended log entries and capture the newly executed ADB operations. If they belong to any of the to-be-monitored ADB operations, the ADB Action Monitor will invoke an alert dialog window to inform users the type and time of the security sensitive ADB operation. Figure 3 shows a screenshot of the ADB Action Monitor, which reported a potentially malicious APP installation.

This monitor module is designed to be a kernel-level process based on Android Open Source Project source code. It will be automatically started when the kernel starts. It has two major advantages as a kernel-level process. First, ADB Action Monitor can read the ADB tracing logs without acquiring permissions like the application-layer programs. The monitor process is started and owned by system user of the kernel, so reading the low-level system log files, which are created by system user as well, does not need any extra permission. Second, different from the general Android APPs, kernel-level processes cannot be controlled by the ADB daemon if root permission is not granted. Most Android devices for normal users are not rooted. Thus, the attackers are not able to disable ADB Action Monitor by taking control of the ADB connection. If the ADB attacker tries to root the device, the monitor can warn the user before it can succeed. If the target device is already rooted, a rooted Android system is under a dangerous condition, discussing its security problems is beyond the scope of this paper.

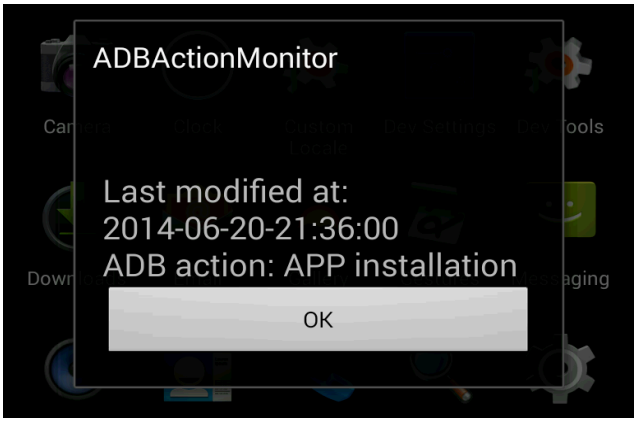

Fig. 3. ADB Action Monitor.

B. Evaluations

To evaluate the effectiveness of our approach, we tested ADB Action Monitor in two different scenarios (debugging scenario and attack scenario) and compared the results. We used an Android simulator called Android Virtual Device (AVD) to emulate two devices, a smart phone and a tablet, which have the same Android build version 4.2.2 with the ADB Action Monitor module activated. For the first scenario, the two devices were used by developers for debugging usage. In particular, we collected the testing data of three activity-debugging processes. The developer first used the APP manually on the device. Then he or she invoked specific activities of the APP through ADB to check the debug data.

For the second scenario, we simulated three different attacks on both devices. Attack 1 needs the target device to be rooted so as to obtain higher privilege to steal sensitive data. The root permission can also let the attacker install more malicious programs to the target. Once the attack is initiated, it will check the target first to see if it is rooted. If not, the attack will try to root the device, and then continue next steps after the rooting process is successful. Attack 2 tries to steal business related privacy information from the devices. It focuses on APPs including Social Networking Services (SNS), online shopping, and web browsers. Attack 3 steals all kinds of personal information, such as photos, contacts, working documents, notes and dairies, which may be stored on mobile devices.

Tables III and IV provide the evaluation results. The data shows some major differences between the debugging and attack scenarios. First, they have different message types. In debugging processes, only "Install" and "Activity Manager" messages are displayed, which indicates that debugging processes primarily uses these two types of ADB operations considered for security monitoring. Attacking processes are more diverse than debugging, so their ADB operations vary and many of them are security sensitive. Second, the average number of messages displayed per minute (Msg/min) is different for the two scenarios. This difference shows that attacks to Android system trigger more warnings (messages) than the debugging during the same period of time. Also, there is an obvious difference in the maximum number of messages per minute for the two scenarios, as indicated in Tables III and IV.

TABLE III. ADB ACTION MONITOR EVALUATION (DEBUGGING)

Debugging on Device 1 (Smart Phone)					
#	***Time***	***Message Types***	***Total Msg***	***Average Msg/min***	***Max Msg/min***
1	9m39s	Install, Activity Manager	7	0.73	2
2	9m3s	Install, Activity Manager	6	0.66	2
3	7m50s	Install, Activity Manager	5	0.64	2
Debugging on Device 2 (Tablet)					
#	***Time***	***Message Types***	***Total Msg***	***Average Msg/min***	***Max Msg/min***
1	9m30s	Install, Activity Manager	7	0.74	2
2	8m59s	Install, Activity Manager	6	0.67	2
3	7m49s	Install, Activity Manager	5	0.64	2

TABLE IV. ADB ACTION MONITOR EVALUATION (ATTACK)

Attacks on Device 1 (Smart Phone)					
#	***Time***	***Message Types***	***Total Msg***	***Average Msg/min***	***Max Msg/min***
1	6m41s	Shell Commands, File Push/Pull, Package Manager	27	4.04	6
2	10m23s	Shell Commands, File Pull, Package Manager, Activity Manager	35	3.37	7
3	23m1s	Shell Commands, File Pull, Activity Manager	81	3.52	7
Attacks on Device 2 (Tablet)					
#	***Time***	***Message Types***	***Total Msg***	***Average Msg/min***	***Max Msg/min***
1	7m9s	Shell Commands, File Push/Pull, Package Manager	29	4.06	7
2	10m59s	Shell Commands, File Pull, Package Manager, Activity Manager	33	3.01	6
3	21m11s	Shell Commands, File Pull, Activity Manager	82	3.87	8

From the comparison, we can see that our ADB Action Monitor is able to notify users ADB based attacks while minimizing the number of warnings for normal debugging processes. In most cases, if a warning message comes up and the user is not using USB Debugging Mode, obviously the user's Android device is under attack. In addition, there is a possibility that the attack happens when functions provided by USB Debugging Mode are running, such as debugging. Under this situation, if the number of warnings is larger than expected or the characteristics of the warnings are not expected, it is very possible that the system is being attacked. By implementing this security enhancement for USB Debugging Mode, Android users can be made aware of ADB attacks and stop them easily to prevent further damages.

VI. CONCLUSION

In this paper, we analyzed the protection effectiveness of secure USB debugging feature against ADB based attacks. We found that the feature has increased ADB's security but still lack the capability in defending against the intrusions from subverted trusted hosts. We implemented such an attack and proposed a solution to enhance the security of USB Debugging Mode. The evaluation of a prototype based on this solution demonstrated that the approach is able to defeat attacks from trusted hosts while not introducing a big burden for normal debugging processes.

For the future work, we plan to perform a comprehensive evaluation of our approach which will involve various Android devices, ADB based attacks and debugging processes. Also, we plan to enhance the functionality of ADB Action Monitor. In particular, we will work on reducing the number of alerts during debugging and automatically terminate a specific ADB connection once the attack through that connection is detected.

REFERENCES

[1] (2013). *Android Debug Bridge | Android Developers* [Online]. Available: http://developer.android.com/tools/help/adb.html

[2] N. Elenkov. (2013). *Secure USB debugging in Android 4.2.2* [Online]. Available: http://nelenkov.blogspot.com/2013/02/secure-usb-debugging-in-android-422.html

[3] F. Chung. (2013). *Security Enhancements in Jelly Bean* [Online]. Available: http://android-developers.blogspot.jp/2013/02/security-enhancements-in-jelly-bean.html

[4] (2013). *Using Hardware Devices | Android Developers* [Online]. Available: http://developer.android.com/tools/device.html

[5] R. Lemos. (2011). *Open source vulnerabilities paint a target on Android* [Online]. Available: http://www.informationweek.in/informationweek/news-analysis/178446/source-vulnerabilities-paint-target-android

[6] (2013). *Android, the world's most popular mobile platform* [Online]. Available: http://developer.android.com/about/index.html

[7] (2013). *Workflow Introduction | Android Developers* [Online]. Available: http://developer.android.com/tools/workflow/index.html

[8] Rooting the Droid without rsd lite up to and including FRG83D, http://androidforums.com/droid-all-thingsroot/171056-rooting-droid-withoutrsd-lite-up-including-frg83d.html (2010)

[9] T. Vidas, D. Votipka, and N. Christin. "All your droid are belong to us: a survey of current android attacks." In Proceedings of the 5th USENIX conference on Offensive technologies (WOOT'11), Berkeley, pp. 81-90, August 2011.

[10] A.G. Villan, J. Jorba. "Remote control of mobile devices in Android platform." arXiv preprint arXiv:1310.5850 (2013).

[11] S. Höbarth, R. Mayrhofer. "A framework for on-device privilege escalation exploit execution on Android." Proceedings of IWSSI/SPMU, San Francisco, 2011.

[12] T. Kobayashi, "ADB (Android Debug Bridge): How it works?" Android Builders Summit, 2012.

[13] Z. Wang, and A. Stavrou. "Exploiting smart-phone usb connectivity for fun and profit." In Proceedings of the Annual Computer Security and Applications Conference (ACSAC), 2010.