Security enhancement of secure USB debugging in Android system

Mingzhe Xu
University of Toledo

Follow this and additional works at: http://utdr.utoledo.edu/theses-dissertations

Recommended Citation
http://utdr.utoledo.edu/theses-dissertations/1801

This Thesis is brought to you for free and open access by The University of Toledo Digital Repository. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of The University of Toledo Digital Repository. For more information, please see the repository's About page.
A Thesis

entitled

Security Enhancement of Secure USB Debugging in Android System

by

Mingzhe Xu

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the

Master of Science Degree in

Engineering with Concentration on Computer Science

_________________________________________

Dr. Weiqing Sun, Committee Chair

_________________________________________

Dr. Mansoor Alam, Committee Co-Chair

_________________________________________

Dr. Henry Ledgard, Committee Member

_________________________________________

Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo

December 2014
An Abstract of

Security Enhancement of Secure USB Debugging in Android System

by

Mingzhe Xu

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Master of Science Degree in Engineering with Concentration on Computer Science

The University of Toledo

December 2014

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 new security feature secure USB debugging was implemented so that only trusted hosts can use ADB. This research studies the features of internal communications of ADB. Then it 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. The implementation of this design and its evaluation are also provided to demonstrate its effectiveness.
# Table of Contents

Abstract .............................................................................................................................. iii

Table of Contents ............................................................................................................... iv

List of Tables ..................................................................................................................... vi

List of Figures ................................................................................................................... vii

List of Abbreviations ....................................................................................................... viii

1 Introduction ..................................................................................................................... 1

2 Literature Review ........................................................................................................... 5

  2.1 Android and Mobile Device Security ........................................................................... 5

  2.2 Android Debug Bridge Introduction and Security ...................................................... 8

3 Background .................................................................................................................... 13

  3.1 Privileges of USB Debugging Mode .......................................................................... 13

  3.2 Sensitive Commands of ADB .................................................................................... 14

  3.3 ADB Security Issues .................................................................................................. 15

4 Understanding ADB Internals ...................................................................................... 18

  4.1 Communication Protocol ......................................................................................... 18

  4.2 Services ..................................................................................................................... 22

  4.3 Logging System ........................................................................................................ 25

5 Secure USB Debugging ............................................................................................... 29

  5.1 The Implementation of Secure USB Debugging ....................................................... 30
5.2 Potential Problems ........................................................................................................31
6 Demonstration Attack ........................................................................................................34
7 Security Enhancement .........................................................................................................39
  7.1 ADB Action Monitor .......................................................................................................39
  7.2 Evaluation ......................................................................................................................43
8 Conclusion and Future Work ..............................................................................................47
  8.1 Conclusion .....................................................................................................................47
  8.2 Future Work ....................................................................................................................47
References ................................................................................................................................49
A List of ADB Services ...........................................................................................................53
List of Tables

4.1 Summary of message types.................................................................20
4.2 Summary of ADB trace mask............................................................28
7.1 Monitored ADB operations..............................................................40
7.2 Shell commands on the white list....................................................41
7.3 Evaluation of ADB Action Monitor (debugging)...............................45
7.4 Evaluation of ADB Action Monitor (attack).......................................45
List of Figures

1-1 Structure of Android Debug Bridge ................................................................. 2
4-1 ADB internal communications flow ................................................................. 23
4-2 Listing device command .................................................................................... 24
4-3 ADB daemon logs ............................................................................................. 26
5-1 Secure USB Debugging confirmation dialog .................................................... 31
5-2 Secure USB Debugging protection .................................................................... 32
6-1 Demonstrates the process of the attack .............................................................. 34
6-2 Fetched public & private key pair ....................................................................... 37
6-3 Establishing the ADB connection via TCP/IP .................................................... 37
7-1 ADB Action Monitor ......................................................................................... 42
List of Abbreviations

ADB ....................... Android Debug Bridge
ADBD ..................... Android Debug Bridge Daemon
API ....................... Application Programming Interface
APP ........................ Application (especially for mobile system)

CPU ........................ Central Processing Unit
IDE ....................... Integrated Development Environment
I/O .......................... Input/Output

OS .......................... Operating System

SDK .......................... Software Development Kit
Chapter 1

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. The software is totally transparent to the debugger when it is under debugging. Though 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 for their 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 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. By default, it is disabled. 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.

![Figure 1-1: Structure of Android Debug Bridge [12].](image)

Android designers provided ADB to help developers easily connect to the Android device from their development machines to debug APPs. 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 input 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 will 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 solutions to improve the ADB security and fix the problem.

The structure of the remainder of this thesis is as follows. In chapter 2, some researches and articles related to our study are introduced. Chapter 3 describes the background of our research, including the overview of Android debugging function, and its security issues. Chapter 4 expresses the internal features of ADB, which include its structure, communication, and logging function. Chapter 5 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 new feature and discuss its security effects. The security hole we have found is also described in chapter 5. In
Chapter 6, we use sample intrusions to demonstrate the attacks which exploited this security hole. Then, in chapter 7, the security enhancement is described along with the evaluation. Chapter 8 concludes our research and describes the future work.
Chapter 2

Literature Review

2.1 Android and Mobile Device Security

The research [14] analyzes permissions requested of 940 Android applications and the permissions they actually used. Data from this research shows that one-third of them are over-privileged, even most of applications developers try to follow the least-privilege rule. Those researchers developed an analyzing tool called Stowaway to check the permissions really used by Android application. The Stowaway collects all API calls from application package, and then it can map those calls to the corresponding permissions of Android permission system. By using their research results, researchers can prove that some over-privileged applications are caused by insufficient API documentation.

The article [15] was published in IEEE Security & Privacy by three researchers from Pennsylvania State University. It first introduced Android system, and then described the structure of its application in detail. In the article, an example application is used to explain the different type of components of Android application and the interactions among those components. After this comprehensive explanation, security enforcement of Android was also introduced and discussed. It primarily includes two
features: 1, every application is executed by a unique user of Android’s Linux kernel, this design could achieve system-level application isolation. 2, all application’s inter-component communications are managed by a reference monitor owned by the Android middleware between Linux kernel and Android applications. This helps the system designers to easily build security policies about component accessibility.

The research [16] proposes a new Android security framework which can meet the security requirements considered in the thesis. The Secure Application INTeraction (saint) infrastructure is able to manage the install-time permission assignment as well as the run-time permission usage. In normal systems, those permission managements are only controlled by the application developers, users just can choose to accept or deny. Researchers provide some semantics of application policy for this framework. Next, they describe the Saint infrastructure in detail, followed by explorations of its extension, optimization, and improvement. Finally, examples are shown to demonstrate how the Saint framework can achieve its designated security goals.

The paper [17] offers a very interesting research which is an empirical analysis of the Android application’s permission security system. It uses the Self-Organizing Map (SOM) algorithm of Kohonen (2001) to perform this analysis on 1100 applications. The researchers use a visual way to express their research findings. With some discussions about those results, they address some potential improvement points of the current permission-based security model. They can increase the expressiveness of the permission set of Android while the permission security system will not become more complex.

The paper [18] points out some security defects of the current third-party Android application permission control system. The permissions of an Android application define
what type of resources this application can access. Those permissions are granted at the
time of installation and could not be changed any more once installed. Based on those
characteristics, authors point out that permissions requested by applications cannot be
partially granted. And also, the system does not have ability to apply constraints about
how to use those permissions at run-time. Thus, the authors presented their work, a policy
enforcement framework for Android called Apex, which overcomes the shortages
mentioned before. An improved package installer cooperating with this framework is also
presented.

The paper [19] provides a security improvement for Android using the model
analysis method to current Android permission scheme. Within this model, system
resources are categorized by entities and relationships, and the permission-related
application states, which consist of the behavior specification of permission
authorizations and the interactions between application components, could be defined. In
such a modeled Android permission system, users are able to logically check its current
security state to see if it meets specified security requirements. How to accomplish that
security verification are also presented by the authors. One more contribution of this
paper is, during the specification checking process for Android security state, a security
weakness was found by authors.

The paper [20] was published at the time when Android source code was just
released. The security design was still new to all researchers. The authors give a
comprehensive introduction about the permission security model of Android system. The
major contribution of this paper is the analysis of Android security using formal methods.
Authors use the state and transition approach to modelize the Android system. Then they did some security analyses based this formal model.

Nowadays, as portable smart devices become more and more powerful, they are gradually taking over the roles of desktop and laptop. Security of mobile devices is becoming most important than ever. Papers [24] and [25] are focused on mobile device security. They both focus on the different security concerns between the mobile platforms and the traditional computing environments. They also introduce the possible security threats of mobile devices from various aspects, including hardware, software, user, etc. Those discussions make contributions for the future researches in the mobile device security field.

The three papers [26], [27], and [28] discussed smart phone security. Comparing to the two papers introduced above, these three papers are more specific. [26], [27] discuss the security issues discovered by the authors, and [28] describes a new design of malware detection system for Android. As the new versions of mobile Operating Systems are released continuously, lots of problems found by [26] and [27] have been fixed. But they have helped to improve the security of mobile systems. And the methods about how to address those problems are still very helpful to other researchers. The idea provided by [28] is also very useful. It shows how to implement a customized system to Android environment.

2.2 Android Debug Bridge

The paper [20] provides a new method for Android system forensics process. The advantages of this method are a design of multi-purposed data collection and supporting of continuous forensics. The basic idea is to rebuild an Android device, make it being
designated for collection. This rebuilding process begins with the creation of an Android system recovery image file. Then it modifies the image file to let the changed system open for further customization. Finally, this image is used to “restore” a device to accomplish the flash process, and then needed components and tools are added to the new system to make it workable for collection. During the modification process, the Android Debug Bridge is granted root permission to take full control of the system. And most of the rebuilding tasks, such as adding new utilities, starting daemon processes, transferring binaries, are accomplished through ADB. And also, as this paper states, the ADB is a good interface for collecting system data.

The research in [21] focuses on the anti-forensics techniques. It studies several anti-forensics methods on mobile devices, and then demonstrates their automatically realized instances for Android. The authors also challenge those methods with some acquisition tools for Android system and check their effectiveness. As an important utility of Android which can be used as a forensics tool, Android Debug Bridge is indicated by this paper. Some of the ADB commands allow the extraction of internal memory data. And this research of anti-forensics techniques shows that some of those methods could bypass the ability of forensics brought by ADB.

The paper [22] is about the debugging and testing of Android applications. The authors show the implementation and automation of the model-based graphical user interface testing on Android platform. The major demonstration is doing model-based test on the BBC New Widget APP (Android Application). During the process, authors presented the application modeling process, the test design and execution, and certainly the problems found. Then they collect the data from the test and make a comparison with
the traditional graphical user interface testing, and discuss whether the model-based testing could bring new advantages for the Android applications debugging. The original purpose of developing Android Debug Bridge is the debugging usage. In the paper, the ADB utility is used to connect the Window service to a socket through its port forwarding function.

A mobile device remote control system on Android system is designed and implemented in [10]. This basic idea of this system is building a server providing some services for controlling mobile devices. The server communicates both with clients and mobile devices which are remotely controlled, act like a middleware between them to establish the remote control. In the implementation, a number of remote control functions are realized by using some ADB features such as installing and un installing applications, downloading and uploading files, opening a shell console, and starting applications. The socket connectivity of ADB utility is used in this implementation as well.

The paper [11] presents a very interesting framework for Android applications privilege escalation exploiting (called “rooting”). Differentiating from other temporary or permanent privilege escalation methods, this framework could allow the exploit process to help only selected applications to grant “root” permissions. The authors enumerate several exploits which allow the device to achieve root capabilities. Three of those example exploits, overflowing limit of available processes, remapping shared memory, and restricting access to this Android shared memory, make full use of the ADB utility to accomplish the tasks. Even though some of them are outdated, they still need to be considered when concerning the security issue of USB Debugging Mode and Android Debug Bridge.
The paper [9] does a comprehensive discussion on the possible ways of attack toward Android system. Through the discussion, the authors point out some security weaknesses of Android. And then they demonstrate some real attack examples to support their opinions. The privilege and access of ADB utility are also discussed by this paper. The researchers proposed several attacking scenarios, all of which gained privileged access to Android. Two of those scenarios are highly related with ADB. The assumption of one of the scenarios is “physical access with ADB enabled”, and the other one’s assumption is “physical access without ADB enabled”. In the first scenarios, access to the ADB allows the attacker to exploit the target device easily. Attackers do not have access to ADB in the second scenario, but with the security hole of the recovery mode of Android, they still can enable the ADB and gain the access to it. So the security problem brought by ADB is directly mentioned in this paper.

While the USB connectivity of smart phone devices increases the flexibility of customizing the system, the functional capabilities of the USB physical link are not properly protected. The paper [13] introduces and discusses some attacking approaches through the USB cable of current popular smart phone systems. Those privilege exploiting methods are primarily targeting at Android systems. Authors defined the threat into three scenarios, phone to computer attacks, computer to phone attacks, and phone to phone attacks. Attacks described in both computer-to-phone and phone-to-phone scenarios have mentioned the uses of ADB utility. By flashing the boot-load memory through the USB connection, attackers using a computer connected with smart phone can grant ADB utility the root permission. In phone to phone attack, one Android device will be controlled by another USB-connected Android system if the ADB daemon of the
controller is in “host” mode. That mode is turned off by default, but it can be turned on manually.

These three online articles [1], [2], and [3] introduce the ADB utility and its security design. When the first version of Android Jelly Bean (v4.1.1) build was released, Android designers hid the developer options from users and required special action to enable it (7 clicks on build number tab). Then in the v4.2.2 release, the secure USB debugging security feature was added.

This speech [12] in the 2012 Android Builders Summit did a comprehensive introduction about Android Debug Bridge. It talked about the structure, internals, and commands of ADB, as well as some technical tips and details that people rarely know about. It was the first time that many obscure but important features of ADB including the logging function were presented. This is very helpful for researchers who want to study the ADB utility.
Chapter 3

Background

3.1 Privileges of USB Debugging Mode

In USB Debugging Mode, users are granted special permissions to facilitate the actions required during the debugging process. Due to the reason that 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 which are applied to the normal Android installation process. Users are allowed to trace the APP’s actions such as APIs called, memory allocated, and system settings influenced. The activities of an APP are able to be tuned manually in the USB Debugging Mode. Also, there are some 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.
and is sent 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.

3.2 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 [1]. 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. All of the “am” and “pm” commands start with the “adb shell” command. 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.

3.3 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 some ADB features such as installing and uninstalling applications, downloading and uploading files, opening a shell console, and starting applications. Attackers can also take advantage of this work to launch attacks. 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 by taking actions 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 user 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, then used its ADB utility to connect to and control other Android targets.
Chapter 4

Understanding ADB Internals

As mentioned in chapter 1, ADB works in a client-server model. Clients could request the services provided by ADB daemon (adb). All the major functional tasks of ADB are completed by the ADB daemon. The ADB utility has a logging ability which is disabled by default. This logging function allows people to trace the actions of ADB. In a client-server model, a communication protocol is needed to deal with the service requesting and result handling processes. Certainly, ADB does have this type of protocol. In the followed paragraphs, we will introduce ADB internal structure from three aspects, protocol, services, and logging.

4.1 The Client-Server Communicating Protocol of ADB

To ensure the cooperation between client and daemon works properly, the communicating protocol for ADB is developed. This protocol is simple and easy to implement. It is designed to be capable for future function extending. This straightforward design can also fit the requirement for multiple environments implementation. All the content transporting tasks are accomplished by the message type defined in protocol. The length of the message header is 24 bytes. The header is responsible for connection stream type defining and state acknowledgement. It is
consisted of 6 24-bits components. The structure definition of message header is given below [30].

```c
struct message {
    unsigned command;  /* command identifier constant */
    unsigned arg0;     /* first argument */
    unsigned arg1;     /* second argument */
    unsigned data_length;  /* length of payload (0 is allowed) */
    unsigned data_crc32; /* crc32 of data payload */
    unsigned magic;     /* command ^ 0xffffffff */
};
```

First three components (command, arg0, and arg1) define the type of message and pass the parameters needed by receiver. The length and crc32 of payload are used for data receiving and consistency check for received data. The last one is the mask of command.

With the fixed header structure, the overall message is in flexible length. There is no checksum for the header, thus, no recovery method for receiving message containing invalid header can be provided. In this simple design of protocol, the only result of receiving failed messages is the closing of remote connection. The total number of message types defined by the header command constants is 7. Six of them are used for network communicating activities, and the one left is internally used by the stream input/output pump module. Table 4.1 concludes the parameter format and command constant definition of each type of message [30]. All data and parameters are carried by the two header arguments words (arg0, arg1) and the flexible payload. The local-
id/remote-id arguments always identify the I/O stream relating to the sender side. Thus, in the receiver’s view, remote-id represents the open stream owned by the receiver itself.

Table 4.1: Summary of parameter format and command constant definition of each type of message.

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter and Data Format</th>
<th>Command Constant Name</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT</td>
<td>version maxdata &quot;system-identity-string&quot; &quot;data&quot;</td>
<td>A_CNXN</td>
<td>0x4e584e43</td>
</tr>
<tr>
<td>AUTH</td>
<td>type 0 &quot;data&quot;</td>
<td>A_AUTH</td>
<td>0x48545541</td>
</tr>
<tr>
<td>OPEN</td>
<td>Local-id 0 &quot;destination&quot;</td>
<td>A_OPEN</td>
<td>0x4e45504f</td>
</tr>
<tr>
<td>READY</td>
<td>local-id remote-id &quot;&quot;</td>
<td>A_OKAY</td>
<td>0x59414b4f</td>
</tr>
<tr>
<td>WRITE</td>
<td>0 remote-id &quot;data&quot;</td>
<td>A_WRTE</td>
<td>0x45545257</td>
</tr>
<tr>
<td>CLOSE</td>
<td>local-id remote-id &quot;&quot;</td>
<td>A_CLSE</td>
<td>0x45534c43</td>
</tr>
<tr>
<td>SYNC</td>
<td>online sequence &quot;&quot;</td>
<td>A_SYNC</td>
<td>0x434e5953</td>
</tr>
</tbody>
</table>

The CONNECT (version, maxdata, "system-identity-string") message is sent at the first time to initiate the talk between both sides. The “version” parameter verifies the adb version, its default value is “0x1000000”. Despite new Android builds are released frequently, this default value keeps unchanged. The “maxdata” is the maximum size of a message body. Similarly, this value remains at 4096 as Android being updated. The “system identity string” sends the identity of connected system in the format "<systemtype>:<serialno>:<banner>", where systemtype is "bootloader", "device", or "host". Serialno is some kind of unique ID (or empty), and banner is a human-readable
version or identifier string. The banner is used to transmit useful properties. AUTH (type, 0, "data") message handles the authentication process. If type is TOKEN(1), data is a random token that the recipient can sign with a private key. The recipient replies with an AUTH packet where the type is SIGNATURE(2) and data is the signature. If the signature can be verified using one of pre-stored public keys, the connection and following communications are allowed. The recipient can keep trying different keys. A new token will be send when verification fails. If all keys are not useful, recipient can reply with an AUTH packet where the type is RSAPUBLICKEY(3) and data is the public key. Then user can determine whether to accept this new public key (connection). Once the OPEN (local-id, 0, "destination") message was sent, it means the sender has a stream identified by local-id that it wishes to connect to the specific destination. Below list summaries the common destination naming conventions [30]. For the services can be reached, we will introduce them in detail later.

- "tcp:<host>::<port>"
- "udp:<host>::<port>"
- "local-dgram:<identifier>"
- "local-stream:<identifier>"
- "shell"
- "upload"
- "fs-bridge"

The READY (local-id, remote-id, "") message informs the recipient that the sender's stream identified by local-id is ready to write messages and that it is connected to the recipient's stream identified by remote-id. The WRITE (0, remote-id, "data") message
sends data to the recipient's stream identified by remote-id. The payload length cannot exceed the number defined by maxdata. A WRITE message may not be sent until a READY message is received. The CLOSE (local-id, remote-id, "") message closes the connection between the sender's stream (local-id) and the recipient's stream (remote-id). The local-id argument will be zero if the message responds to a failure OPEN. The SYNC message is used by the I/O pump to make sure that stale outbound messages are discarded when the connection to the remote side is broken. It is only used internally to the bridge and never valid to send across the wire.

4.2 Services

This section introduces the services provided by ADB. All services are come from two components of ADB utility. The first one is ADB daemon, and the second one is the host server. ADB daemon is a daemon process of the Linux kernel of Android system [29]. This daemon receives service request from remote client. Based on the different type of services being requested, daemon will execute certain instruction locally and return the result. So, the services provided by ADB daemon are called “local services”. The host server is a background process runs on the host machine that connects to Android devices. This server process manages all ADB connections between machine and each device. Thus, it also provides services for client for the purposes of maintaining and managing those ADB connections. This type of services is called “host services”.

Like the communication protocol for ADB client-server, there is a simple message format implemented for requesting services [29]. This type of service requesting message is carried by the payload of ADB communicating message packet if the client requested services from ADB daemon. The host server is always combined with the
command-line client utility as a single executable binary on host machine. So, the service request for host services is transferred internally. It does not need to be embedded in the message body of ADB communicating protocol.

The format of service request message is pretty simple. It is only consisted of two parts, one 4-byte hexadecimal string giving the length of the payload and the following payload containing the request message. The payload carries service naming key words and arguments separated by colon and space. The service key words could be one or more reserved words indicating the requested service. If arguments are needed, all the arguments passing to service are directly written behind the key words.

![ADB internal communications flow for “shell:ls” service.](image)

Figure 4-1: ADB internal communications flow for “shell:ls” service. [12]

The host services provide functionalities for maintaining ADB connections with Android devices [31]. For example, the “listing device” service can return client a list of devices currently connecting with the host machine. The actual service requesting message being sent is “000Chost:devices”. The first 4-byte hexadecimal string shows that the following 12 characters indicate the service wanted by client. This 12-byte long string
contains two key words showing the requested service is “listing device” of host service. From another point of view, those ADB services are the implementation of ADB commands. The corresponding command for this service is $adb devices$ (Fig. 4.1). We have introduced several security sensitive commands. Here we talk about some ADB services which should be concerned for security reason. The total number of services is huge, and we cannot discuss all of them in the paper. The list of all services is given in appendix A.

![Microsoft Windows](Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\>adb devices
List of devices attached
1ae52d90 unauthorized
015d25685938140b device
emulator-5554 device
C:\>

Figure 4-2: Listing device command.

All the security sensitive services belong to “local services” [31]. Results of running those services may change the Android device into insecure state. Only services from ADB daemon on the Android kernel can have the abilities to influence security state of this Android system. The “shell command” service executes the command sent from client and returns the result output. The command can include arguments separated by space. This service is the implementation of ADB utility’s $adb shell <command>$. The format of its requesting message is “<host-prefix>:shell:command arg1 arg2 ...”. The
<host-prefix> is only used by the host server to know which device the client is communicating with and will not be passed to device’s ADB daemon via the ADB connection. Alternates for <host-prefix> include “host-serial:<serial-number>”, “host-usb”, “host-local”, and “host”. The <serial-number> indicates the target device that is represented by this number. When “host-usb” was used, it means the client is requesting the service from the only one device connected by USB cable. If the “host-local” is written in the message, the request will be sent to the only Android emulator running on the host machine. The “host” prefix shows the target is the only one device/emulator connected with the ADB server. The “shell” service starts an interactive shell session on device. It will appropriately redirect the standard I/O streams of the kernel on device to client. The message format is “<host-prefix>:shell:”. The “shell” service is the implementation of *adb shell* command. The “remount” and “sync” services deal with file transferring service. Usually the client sends a “remount” request to remount the file system of the device in read-write mode instead of the read-only mode. Then it will ask to start a “sync” service to copy or send a file. These two services are used to implement the *adb push/pull* command. The message formats of them are “<host-prefix>:remount:” and “<host-prefix>:sync:”. The <host-prefix> parts of those requesting messages have exactly the same meaning.

4.3 Logging System

As a very powerful and important module of Android system, ADB utility has its own logging function. Besides the logging functionality module of Linux and the Android’s major logging function called “logcat”, the logging function of ADB is totally independent. It is directly embedded into the utility. This logging system can only log all
the stuffs related to ADB. Those stuffs include the actions taken by ADB and the communications occurred among the different portions of ADB. These communications can be either the talks between command-line client and host server or messages through the connection between host server and daemon.

The ADB utility has two parts which are physically separated. The one located at host machine is composed of the host server and the command-line client. The other portion is the daemon process on Android OS. The logging function also separately comes with the two different parts. Both the host server and device daemon have their own logging modules. Those modules can correspondingly trace actions of server and daemon.

Figure 4-3: ADB daemon logs in category packets and services. (Trace mask value: 0x205)

For host server logging, an environment variable of the host machine called “ADB_TRACE” is used to configure what kinds of actions should be traced. On the device side, an Android system property “persist.adb.trace_mask” controls the category
of ADB activities to be logged. ADB actions are grouped into 9 categories. The logging filter is programmatically controlled by an 11-bit mask called “ADB trace mask”. The first bit of the mask (bit0) switches the tracing on and off. Each of the following bits (bit1 to bit10) indicates the logging of specific type of actions being enabled or not. On the host side, users can specify the categories of actions to be logged by writing string tags to the “ADB_TRACE” environment variable. Those tags contained by the variable can be separated by comma, space, colon, and semicolon. The tag list will be read by ADB host server when the server process is initiating. Then it will automatically configure the trace mask based on the tag list. There are two tags which do not directly associate with certain bit of the trace mask, they are “all” and “1”. If one of the two special tags is written to the “ADB_TRACE”, all kinds of actions will be traced by logging module and all other tags will be ignored. On the device side, the “persist.adb.trace_mask” system property contains the hexadecimal value of the trace mask. This value is directly used by the daemon’s logging function. For example, the hex value for tracing ADB service actions is 201, which is “1 (bit0) + 100000000 (bit9)” in binary number. Tracing logs of host server are sent to the standard I/O streams of the host machine. In the other hand, ADB daemon writes the logs to log files which are stored in directory “/data/adb”. The naming convention of those log file is “adb-%Y-%m-%d-%H-%M-%S.txt”. Table 4.2 summarizes the information about the ADB trace mask.
Table 4.2: Summary of ADB trace mask.

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Constant Name</th>
<th>Environment Variable Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TRACE_ADB</td>
<td>adb</td>
<td>Enable or disable logging feature.</td>
</tr>
<tr>
<td>1</td>
<td>TRACE_SOCKETS</td>
<td>sockets</td>
<td>Logs of socket connections (local &amp; remote).</td>
</tr>
<tr>
<td>2</td>
<td>TRACE_PACKETS</td>
<td>packets</td>
<td>Logs of packet detail.</td>
</tr>
<tr>
<td>3</td>
<td>TRACE_TRANSPORT</td>
<td>transport</td>
<td>Logs of message type (one of the seven types introduced in section A).</td>
</tr>
<tr>
<td>4</td>
<td>TRACE_RWX</td>
<td>rwx</td>
<td>Unknown. (Logs generated are same with transport logs.)</td>
</tr>
<tr>
<td>5</td>
<td>TRACE_USB</td>
<td>usb</td>
<td>Logs of all usb connections</td>
</tr>
<tr>
<td>6</td>
<td>TRACE_SYNC</td>
<td>sync</td>
<td>Logs of synchronizing file-system process.</td>
</tr>
<tr>
<td>7</td>
<td>TRACE_SYSDEPS</td>
<td>sysdeps</td>
<td>Unknown. (Logs generated are same with sync logs.)</td>
</tr>
<tr>
<td>8</td>
<td>TRACE_JDWP</td>
<td>jdwp</td>
<td>Logs of JDWP connection.</td>
</tr>
<tr>
<td>9</td>
<td>TRACE_SERVICES</td>
<td>services</td>
<td>Logs of services being requested.</td>
</tr>
<tr>
<td>10</td>
<td>TRACE_AUTH</td>
<td>auth</td>
<td>Logs of authentication process.</td>
</tr>
</tbody>
</table>
Chapter 5

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 deem it as a security improvement.

Thus, in the 4.2.2 update, designers introduced the new security policy 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 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].

5.1 The Implementation of Secure USB Debugging

In the design of secure USB debugging [2], a 2048-bit RSA encryption method 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 SHA1 with RSA 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, which means the host cannot perform any action on the device. Failure of the verification 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 before it sends back the verification message. 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.
5.2 Potential Problems

This method provides a white list design for ADB connection authorizations for the sake of better security and usability. It does not require lots of 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” files 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.

Figure 5-2: Secure USB Debugging prevents an attack via ADB. [2]
However, sophisticated attacks might still be able to bypass the above protection. We assume that 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 cause damages to 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.

For another similar 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. This situation could be more dangerous. The host is intruded and as the result the hacker could obtain the file storing the private key. Then this private key can be used to bypass secure USB debugging when the attacker uses other hosts to connect to the device. 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.
Chapter 6

Demonstration Attacks

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 6-1: 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 cable on the computer. This software also can run some scripts in the background without users’ knowledge.

We assume that someone just downloaded this “malicious” software from Internet and installed it. 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 following command needs to be executed:

“adb tcpip 5555”
This ADB command will restart the ADB daemon on the device and set it 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 treating it 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, this connection does not require a connected USB port. If the device gains 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. Figure 6-2 shows the stolen RSA key pair and Figure 6-3 shows the process of setting and creating the ADB connection via a TCP/IP network.
Figure 6-2: Fetched public & private key pair.

Figure 6-3: Establishing the ADB connection via TCP/IP.
Once the attacker takes control of ADB, he/she could do more damages to the target. In particular, the attacker can install malicious software with all permissions for Android APP, using it to steal owner’s privacy, 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 information of the targeting device.
Chapter 7

Security Enhancement

The previous chapters show that the new secure USB debugging design lack abilities in defending against attacks from the subverted trusted host. Thus, improvements to address this security issue need to be made. Here we design and implement a new approach to enhance the USB debugging security.

7.1 ADB Action Monitor

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 defeat 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 messages are only applied to several security sensitive
operations, (Table 7.1 summarizes all such operations which should be monitored.) so the number of prompt messages will not be too overwhelming for the users, especially when a normal debugging operation is taking place.

Table 7.1: Monitored ADB operations.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>install &lt;path-to-apk&gt;</td>
<td>Installs an Android application to the device.</td>
</tr>
<tr>
<td>pull &lt;remote&gt; &lt;local&gt;</td>
<td>Gets a file from the device to local host.</td>
</tr>
<tr>
<td>push &lt;local&gt; &lt;remote&gt;</td>
<td>Sends a file to the device.</td>
</tr>
<tr>
<td>shell</td>
<td>Starts a remote shell in the connected device.</td>
</tr>
<tr>
<td>shell [shell command]</td>
<td>Issues the shell command to the connected device.</td>
</tr>
<tr>
<td>shell am [command]</td>
<td>Issues the activity manager command to the device.</td>
</tr>
<tr>
<td>shell pm [command]</td>
<td>Issues the package manager command to the device.</td>
</tr>
</tbody>
</table>

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 7.2 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 7.2: Shell commands on the white list.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shell echo</td>
<td>Shows a string on terminal.</td>
</tr>
<tr>
<td>shell getprop</td>
<td>Gets stored property settings of Android system.</td>
</tr>
<tr>
<td>shell ls</td>
<td>Lists the content of a directory.</td>
</tr>
</tbody>
</table>

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 on device. 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 monitored ADB operations, the ADB Action Monitor process will invoke an alert dialog window to inform users the type and time of the latest security sensitive ADB operation. Figure 7-1 shows a screenshot of the ADB Action Monitor. Potential attacks take actions through the ADB daemon services, so the logs of ADB daemon need to be monitored. The ADB Action Monitor automatically configures system to enable the logging function on ADB daemon. Only services related actions are traced to minimize the size of log file being generated. Old log files will be discarded to prevent over occupying system memory.
This monitor module is designed to be a kernel-level process. We developed it based on Android Open Source Project source code and added its utility binary to the Android kernel. It will be automatically started when the kernel starts. Becoming a kernel-level process has two major benefits. 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
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 very dangerous condition, discussing its security problems is beyond the scope of this paper.

7.2 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.

For the second scenario, we simulated three different attacks.

- Attack 1. This attack needs the target device to be rooted so that it can grant 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. This attack tries to steal commercial related privacy information from the devices. It first checks the list of APPs installed on the target, then focuses on Social Networking Services (SNS), online shopping, and web browsers, trying to get all data which could be used for business needs.
• Attack 3. This attack steals all kinds of personal information, such as photos, contacts, working documents, notes and dairies, which may be stored on mobile devices.

Tables 7.3 and 7.4 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 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 highest number of messages per minute for the two scenarios, as indicated in Tables 7.3 and 7.4.
Table 7.3: Evaluation of ADB Action Monitor (debugging).

### Debugging on Device 1 (Smart Phone)

<table>
<thead>
<tr>
<th>#</th>
<th>Time</th>
<th>Message Types</th>
<th>Total Messages</th>
<th>Average Messages per Minute</th>
<th>Most Messages per Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9m39s</td>
<td>Install, Activity Manager</td>
<td>7</td>
<td>0.73</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>9m3s</td>
<td>Install, Activity Manager</td>
<td>6</td>
<td>0.66</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>7m50s</td>
<td>Install, Activity Manager</td>
<td>5</td>
<td>0.64</td>
<td>2</td>
</tr>
</tbody>
</table>

### Debugging on Device 2 (Tablet)

<table>
<thead>
<tr>
<th>#</th>
<th>Time</th>
<th>Message Types</th>
<th>Total Messages</th>
<th>Average Messages per Minute</th>
<th>Most Messages per Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9m30s</td>
<td>Install, Activity Manager</td>
<td>7</td>
<td>0.74</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>8m59s</td>
<td>Install, Activity Manager</td>
<td>6</td>
<td>0.67</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>7m49s</td>
<td>Install, Activity Manager</td>
<td>5</td>
<td>0.64</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.4: Evaluation of ADB Action Monitor (attack).

### Attacks on Device 1 (Smart Phone)

<table>
<thead>
<tr>
<th>#</th>
<th>Time</th>
<th>Message Types</th>
<th>Total Messages</th>
<th>Average Messages per Minute</th>
<th>Most Messages per Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6m41s</td>
<td>Shell Commands, File Push/Pull, Package Manager</td>
<td>27</td>
<td>4.04</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>10m23s</td>
<td>Shell Commands, File Pull, Package Manager, Activity Manager</td>
<td>35</td>
<td>3.37</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>23m1s</td>
<td>Shell Commands, File Pull, Activity Manager</td>
<td>81</td>
<td>3.52</td>
<td>7</td>
</tr>
</tbody>
</table>
### Attacks on Device 2 (Tablet)

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Message Types</th>
<th>Total Messages</th>
<th>Average Messages per Minute</th>
<th>Most Messages per Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7m9s</td>
<td>Shell Commands, File Pull/Push/Pull, Package Manager</td>
<td>29</td>
<td>4.06</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>10m59s</td>
<td>Shell Commands, File Pull, Package Manager, Activity Manager</td>
<td>33</td>
<td>3.01</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>21m11s</td>
<td>Shell Commands, File Pull, Activity Manager</td>
<td>82</td>
<td>3.87</td>
<td>8</td>
</tr>
</tbody>
</table>

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 or some APPs which require users to enable USB Debugging Mode are running. 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.
Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this research, after a comprehensive introduction about Android Debug Bridge, we turned to focus on the internal structure of ADB. Then we analyzed the protection effectiveness of secure USB debugging design against ADB based attacks. The results show that the new feature has increased ADB’s security but still lack the capability in defending against the intrusions from the subverted trusted host. We implemented such an attack and proposed a solution to enhance the security of USB Debugging Mode. The mean idea of the security enhancement is to add monitor module to the daemon portion of ADB utility to warn user all the security sensitive activities through ADB, which are taking place in the back end. Then the user being notified could take actions accordingly to prohibit all the unauthorized activities. 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.

8.2 Future Work

In our future work, we will improve the ADB Action Monitor from three aspects. First, we plan to increase its functionality. Right now, when the normal debug process is occurring, some action warnings will still be prompted by the monitor. We will work on
reducing the number of alerts during debugging. To accomplish this, the monitor will become more powerful for reading ADB logs. The monitor will be able to read more information from logs. That newly granted information will be used to determine if the current sensitive action caused by debugging processes or attacks. Accordingly, the log analyzing logic will become more sophisticated to be able to use this information.

The second future work is to measure the system resource cost. We did not do any test for system resource taken by ADB Action Monitor. The monitor module is designed to be started automatically and always be running when Android system is not off. So, the resource cost should be in an acceptably low level. We will test the CPU and memory cost as well as the battery cost, which is very important for mobile devices. If those costs are too high to be affordable by mobile systems, we will try to improve the module to lower the costs.

The third improvement is to let the ADB Action Monitor become smart and automatic. Now the ADB Action Monitor is only a monitor. It can only warn users about the attacks to their devices. It still needs users to take actions to prevent those attacks. If the user currently cannot reach the device, the monitor will not be able to do anything that can prohibit the attack. In the future improvement, the monitor can automatically terminate a specific ADB connection once the attack through that connection was found. Thus, no further user action is required to stop the attack. The protection on ADB is automatically achieved.
References


49


Appendix A

List of ADB Services

1. HOST SERVICES [31]:

*host:version*

Ask the ADB server for its internal version number. As a special exception, the server will respond with a 4-byte hex string corresponding to its internal version number, without any OKAY or FAIL.

*host:kill*

Ask the ADB server to quit immediately. This is used when the ADB client detects that an obsolete server is running after an upgrade.

*host:devices/host:devices-l*

Ask to return the list of available Android devices and their state. devices-l includes the device paths in the state. After the OKAY, this is followed by a 4-byte hex len, and a string that will be dumped as-is by the client, then the connection is closed.

*host:track-devices*

This is a variant of host:devices which doesn't close the connection. Instead, a new device list description is sent each time a device is added/removed or the state of a given device
changes (hex4 + content). This allows tools like DDMS to track the state of connected devices in real-time without polling the server repeatedly.

host:emulator:<port>

This is a special query that is sent to the ADB server when a new emulator starts up. <port> is a decimal number corresponding to the emulator's ADB control port, i.e. the TCP port that the emulator will forward automatically to the adbd daemon running in the emulator system. This mechanism allows the ADB server to know when new emulator instances start.

host:transport:<serial-number>

Ask to switch the connection to the device/emulator identified by <serial-number>. After the OKAY response, every client request will be sent directly to the adbd daemon running on the device. (Used to implement the -s option)

host:transport-usb

Ask to switch the connection to one device connected through USB to the host machine. This will fail if there are more than one such devices. (Used to implement the -d convenience option)

host:transport-local

Ask to switch the connection to one emulator connected through TCP. This will fail if there is more than one such emulator instance running. (Used to implement the -e convenience option)

host:transport-any
Another host:transport variant. Ask to switch the connection to either the device or emulator connects to/running on the host. Will fail if there is more than one such device/emulator available. (Used when neither -s, -d or -e are provided)

host-serial:<serial-number>:<request>

This is a special form of query, where the 'host-serial:<serial-number>:prefix can be used to indicate that the client is asking the ADB server for information related to a specific device. <request> can be in one of the format described below.

host-usb:<request>

A variant of host-serial used to target the single USB device connected to the host. This will fail if there is none or more than one.

host-local:<request>

A variant of host-serial used to target the single emulator instance running on the host. This will fail if there is none or more than one.

host:<request>

When asking for information related to a device, 'host:' can also be interpreted as 'any single device or emulator connected to/running on the host'.

<host-prefix>:get-serialno

Returns the serial number of the corresponding device/emulator. Note that emulator serial numbers are of the form "emulator-5554".

<host-prefix>:get-devpath

Returns the device path of the corresponding device/emulator.

<host-prefix>:get-state

Returns the state of a given device as a string.
<host-prefix>:forward:<local>;<remote>

Asks the ADB server to forward local connections from <local> to the <remote> address on a given device. There, <host-prefix> can be one of the host-serial/host-usb/host-local/host prefixes as described previously and indicates which device/emulator to target.

The format of <local> is one of:

- tcp:<port>  TCP connection on localhost:<port>
- local:<path>  Unix local domain socket on <path>

The format of <remote> is one of:

- tcp:<port>  TCP localhost:<port> on device
- local:<path>  Unix local domain socket on device
- jdwp:<pid>  JDWP thread on VM process <pid>

Or even any one of the local services described below.

<host-prefix>:forward:norebind:<local>;<remote>

Same as <host-prefix>:forward:<local>;<remote> except that it will fail it there is already a forward connection from <local>. Used to implement 'adb forward --no-rebind <local> <remote>'.

<host-prefix>:killforward:<local>

Remove any existing forward local connection from <local>. This is used to implement 'adb forward --remove <local>.'.

<host-prefix>:killforward-all

Remove all forward network connections. This is used to implement 'adb forward --remove-all'.
List all existing forward connections from this server. This returns something that looks like the following:

\<hex4\>: The length of the payload, as 4 hexadecimal chars.

\<payload\>: A series of lines of the following format:

\<serial\> " " \<local\> " " \<remote\> "\n"

Where \<serial\> is a device serial number. \<local\> is the host-specific endpoint (e.g. tcp:9000). \<remote\> is the device-specific endpoint. Used to implement 'adb forward --list'.

2. LOCAL SERVICES [31]:

All the queries below assumed that you already switched the transport to a real device, or that you have used a query prefix as described above.

shell: command arg1 arg2 ...

Run 'command arg1 arg2 ...' in a shell on the device, and return its output and error streams. Note that arguments must be separated by spaces. If an argument contains a space, it must be quoted with double-quotes. Arguments cannot contain double quotes or things will go very wrong. Note that this is the non-interactive version of "adb shell".

shell:

Start an interactive shell session on the device. Redirect stdin/stdout/stderr as appropriate.

Note that the ADB server uses this to implement "adb shell", but will also cook the input before sending it to the device (see interactive_shell() in commandline.c).

remount:
Ask adb to remount the device's filesystem in read-write mode, instead of read-only. This is usually necessary before performing an "adb sync" or "adb push" request. This request may not succeed on certain builds which do not allow that.

*dev:* <path>

Opens a device file and connects the client directly to it for read/write purposes. Useful for debugging, but may require special privileges and thus may not run on all devices. <path> is a full path from the root of the filesystem.

*tcp:* <port>

Tries to connect to tcp port <port> on localhost.

*tcp:* <port>:<server-name>

Tries to connect to tcp port <port> on machine <server-name> from the device. This can be useful to debug some networking/proxy issues that can only be revealed on the device itself.

*local:* <path>

Tries to connect to a Unix domain socket <path> on the device.

*localreserved:* <path>

*localabstract:* <path>

*localfilesystem:* <path>

Variants of local:*<path>* that are used to access other Android socket namespaces.

*log:* <name>

Opens one of the system logs (/dev/log/<name>) and allows the client to read them directly. Used to implement 'adb logcat'. The stream will be read-only for the client.

*framebuffer:*
This service is used to send snapshots of the framebuffer to a client. It requires sufficient privileges but works as follow:

After the OKAY, the service sends 16-byte binary structure containing the following fields (little-endian format):

- `depth`: uint32_t: framebuffer depth
- `size`: uint32_t: framebuffer size in bytes
- `width`: uint32_t: framebuffer width in pixels
- `height`: uint32_t: framebuffer height in pixels

With the current implementation, depth is always 16, and size is always width*height*2. Then, each time the client wants a snapshot, it should send one byte through the channel, which will trigger the service to send it 'size' bytes of framebuffer data. If the adbd daemon doesn't have sufficient privileges to open the framebuffer device, the connection is simply closed immediately.

`jdwp:<pid>`

Connects to the JDWP thread running in the VM of process `<pid>`.

`track-jdwp`

This is used to send the list of JDWP pids periodically to the client. The format of the returned data is the following:

- `<hex4>`: the length of all content as a 4-char hexadecimal string
- `<content>`: a series of ASCII lines of the following format: `<pid> "\n"

This service is used by DDMS to know which debuggable processes are running on the device/emulator. Note that there is no single-shot service to retrieve the list only once.

`sync:`
This starts the file synchronisation service, used to implement "adb push" and "adb pull".