COMPUTERS & SECURITY 82 (2019) 80–98

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

(In)Secure Android Debugging: Security analysis and lessons learned

Krzysztof Opasiak[a,b], *Wojciech Mazurczyk*[a,*]

[a] *Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warszawa, Poland*
[b] *Samsung R&D Institute Poland, plac Europejski 1, 00-844 Warszawa, Poland*

ARTICLE INFO

Article history:
Received 13 September 2018
Revised 4 December 2018
Accepted 20 December 2018
Available online 25 December 2018

Keywords:
Mobile security
Android
USB
ADB
MITM

ABSTRACT

Universal Serial Bus (USB) is currently one of the most popular standards that controls communication between personal computers (PCs) and their peripheral devices. Thus, it is important to establish whether such connections are properly secured especially when USB is used to connect devices like smartphones, tablets, etc. where sensitive user data can be potentially stored. For this reason, this paper evaluates security of the recent Android versions with respect to the USB-related attacks. In particular, we present a novel approach to compromise Android-based devices by exploiting Android Debug Bridge (ADB) protocol using Man in the Middle (MitM) attacks. Comprehensive analysis of those types of attacks have revealed five novel security vulnerabilities in the Android OS. Security gaps found in this paper cannot only be used to bypass the lock screen security and to gain unauthorized access to the user's private data but also to enable future ADB attacks by incorporating a backdoor to bypass phone security at any time. We also developed a tool which exploits all discovered vulnerabilities and can serve as a security mean to assess current ADB implementations as well as future protocol improvements. By disclosing new security weaknesses we want to raise security awareness of the users, researches, security professionals, and developers related to the USB-related attacks and to the threat they pose not only to PCs but also to the USB devices.

© 2018 Elsevier Ltd. All rights reserved.

1. Introduction

Using USB (Universal Serial Bus) to bypass a modern personal computer (PC) security mechanism is an interesting but not a novel idea. First malware that used this kind of attack vector was Conficker (Hypponen, 2009) discovered in 2008. Its direct successor – Stuxnet (Nicolas Falliere and Chien, 2011) – managed to infect more than 60,000 computers in Iran only. Back in 2014 USB users were thrilled once again due to BadUSB (Karsten Nohl and Lell, 2014) disclosure. It became very popular and many people started developing their own versions of BadUSB attack (Kamkar, 2014; 2016; Kierznowski, 2016; Todd-Simpson, 2016). At the same time people put some effort in developing tools (Kopecek, 2016) to protect against evil USB devices.

The USB security is of a particular concern when abundance of devices utilizing this standard is considered. There is currently billions of devices which employ USB, some which are more sensitive than others – smartphones, tablets and all other mobile devices. They are equipped with high performance CPUs and have a lot of storage which is often filled up with users' private data. USB in these devices is widely used to easily access photos, videos, documents, etc. from the users' desktops without uploading private data to the third-party cloud.

* Corresponding author.
E-mail addresses: k.opasiak@tele.pw.edu.pl (K. Opasiak), wmazurczyk@tele.pw.edu.pl (W. Mazurczyk).
https://doi.org/10.1016/j.cose.2018.12.010
0167-4048/© 2018 Elsevier Ltd. All rights reserved.

Android, with its 88% market share (Sui, 2016) is unquestionably the most common mobile operating system, widely used in all kinds of mobile devices. Apart from being ergonomic, flexible and feature-rich the real strength of any successful operating system lays also in the number of available applications. Android with its 2.8 millions of applications easily fulfills the number of application criterion (number, 2017). Such a huge number of available programs is a result of the existing, vibrant Android Application Developers Community. It is estimated that there is more than 12 millions mobile apps developers worldwide and more than half of them focuses their attention on Android (app, 2016).

Mobile app development process differs from a typical software creation because the code is usually written on a PC machine, often initially tested using mobile OS emulator and then uploaded to the target mobile device for final testing and debugging. Thus, a dedicated communication mechanism between the developer's machine and the target device is necessary. Bootloader and kernel developers use dedicated low level interfaces like JTAG which allows them to stop the CPU and access the RAM memory. JTAG is also used by security researchers to perform forensic analysis of the device using for example JoKER (Guri et al., 2015). Unfortunately it requires expensive equipment and direct access to device's PCB (Printed Circuit Board). That is why, for app development, more convenient and high level tool is required. In Android, this requirement is fulfilled by the custom USB protocol called Android Debug Bridge (ADB). ADB provides many functionalities which are useful for mobile apps developers, including:

- file transfer,
- shell access,
- application installation and debugging,
- port forwarding,
- generating input and sensors events.

Although, as mentioned above, ADB functionality is extremely useful for developers and advanced users, in its current form, it should be considered as a serious threat for Android-based devices' security. Security threats can arise as ADB connection has a direct access to all user's private data without the need to unlock the device. Even though it is sometimes useful to be able to retrieve such data, for example, in case when the mobile device display gets broken, when used with malicious intentions it may easily lead to the private data leakage. Additionally, there are dedicated tools (e.g. within the Metasploit package) which allow for an arbitrary code execution via ADB (met, 2016). In the past there were also several exploits which allowed to gain root access to the device using ADB connection (dir, 2016; Vidas et al., 2011).

To mitigate the ADB-related security risks manufacturers incorporated two main security improvements. Firstly, ADB is now disabled by default. In order enable it user has to explicitly activate the option hidden under developer options submenu. Secondly, from the Android version 4.2.2 a secure USB debugging feature has been introduced. This mechanism is aimed at preventing unauthorized ADB access to mobile devices when the ADB is turned on.

As the Android Developers and Enthusiasts community is growing, more and more people start to utilize the ADB. Many of them activate it once and then forget to disable it later. In this case, their only protection is a Secure USB Debugging feature. To the authors' best knowledge no security analysis of this mechanism is currently available. Therefore, the main contribution of this paper is to fill this gap by evaluating security of the Secure USB Debugging from the USB connectivity perspective. This is achieved by:

- Introducing a classification of the known USB-related security threats.
- Providing up-to-date documentation of the ADB protocol and its current security mechanisms.
- Performing security evaluation and discussing vulnerabilities found in the ADB protocol itself and its implementations in Samsung Galaxy S7 and Google Nexus 9 mobile devices.
- Introducing a new tool called *adb_mitm* which exploits discovered vulnerabilities.
- Discussing lessons learned and proposing potential countermeasures.
- Analyzing performance impact of the proposed ADB protocol modification and suggesting a method to minimize the overhead.

The rest of this paper is structured as follows. Section 2 describes the necessary basics of the USB standard. Then, in Section 3 an up-to-date overview of the ADB protocol is provided. Section 4 discusses related work on the ADB and USB security. Next, in Section 5 experimental test-bed used for our research is presented. Section 6 reveals new security vulnerabilities and introduces *adb_mitm* tool which exploits them. Then, Section 7 proposes modification to the ADB protocol in order to prevent against discussed attacks. Finally, Section 8 concludes the paper and suggests potential future work in this area.

2. USB fundamentals

The USB standard (usb, 2000) is one of the most popular external interfaces. Most of the consumer home electronics is able to communicate over USB. Also communication between the mobile device and developer's machine is usually realized over the USB bus. In order to be able to perform its security analysis it is important to understand how USB protocol works.

The first version of USB standard has been published in 1996. It offered Low Speed (1.5 Mb/s) and Full Speed (12 Mb/s) signaling rates. Then in 2000, USB 2.0 standard has been released. It introduced significantly faster signaling speed named High Speed (480 Mb/s). Its later revisions introduced a lot of features important for USB adoption in mobile devices including minimizing the connector size, improved battery charging etc. Next generation of USB standards started by publishing USB 3.0 specification in 2008. This standard is a first one to introduce full duplex communication and increased signaling speed to 5 Gb/s (SuperSpeed). It also increased the current limit for the USB device to 900 mA. Further minor revisions of this standard (3.1 and 3.2) increased the speed even more up to 20 Gb/s in USB 3.2 SuperSpeed+ mode.

It is worth mentioning that not only USB standard itself evolved but also USB connectors standard. In the beginning it defined only full size Type A and Type B connectors. Later revisions of that standard introduced minimized form factors of that connectors. The biggest revolution was USB Type C connector introduced in 2014, as it is symmetrical for both sides of communication. As most of mobile devices on the market still utilizes USB 2.0 specification in this chapter we describe USB stack based on that standard but most of the information is valid also for the newer ones as well.

A general, high-level concept of the USB is to extend machine's functionality (equipped with such a port) with some additional features offered by the peripherals. In the USB terminology a machine which is extendable via USB is called *USB host*. Usually it is a piece of electronics which is directly used by a human user like a computer, a smart TV or a media player. In contrast, a peripheral which provides some additional functionality via USB is called a *USB device*. Previously a USB device was typically a much smaller and resource-limited device e.g. a pendrive or a web camera. Nowadays USB device mode is widely used in mobile devices which may have significantly larger computing power.

USB standard defines four layers of the USB stack:

physical layer: which is responsible for the electric adjustment and low level signaling.
link layer: which is responsible for the basic packet transfer between two USB ports.
protocol layer: which ensures end-to-end messages reliability and manages bandwidth.
functional layer: which allows developers to implement their own functionality on the top of the USB protocol.

First three layers are generic and independent from the functionality which is being provided over the USB bus. Functional layer is used to define a function-specific protocol which can be used to provide even very specific functionality over the USB bus.

USB standard allows to connect up to 127 USB devices to a single USB host controller. Obviously, mounting 127 USB ports in each USB host is infeasible. So a typical USB host is shipped with a few of physical USB connectors. To increase the number of available USB ports there is a dedicated class of the USB devices called *USB hubs*. They are equipped with a single upstream port which should be connected to the USB host and multiple downstream ports which can be used to connect more USB devices. So a typical physical USB topology is a tree topology (Fig. 1) with a USB host being a root of that tree, USB hubs being nodes and USB devices being the leaves.

From a logical point of view USB hubs are transparent and used only as signal retransmitters. Thus the logical USB topology forms a star topology (Fig. 2) with the USB host being a central node and USB devices as leaves. On a functional level communication is always realized between the host and one of the connected USB devices. There is no logical connection between the USB devices.

Typical USB device consists of the three basic elements:

USB Device Controller (UDC): This is a piece of hardware which allows to communicate in the USB device mode. It may be seen as an equivalent of a Network Interface Card (NIC) in the networking environment.
Executive element: This may be a hardware component (like a flash memory in case of a pendrive) which is required to provide the desired functionality or just a software which provides required resources or an infrastructure.
MCU[1] *and firmware/software:* This is the main controller which communicates with the UDC and the Executive element to implement device logic.

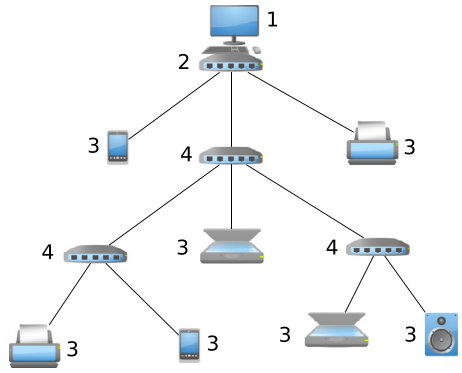

Fig. 1 – Physical USB topology (1 – USB host, 2 – root hub, 3 – USB device, 4 – USB hub).

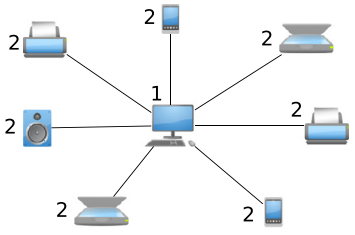

Fig. 2 – Logical USB topology (1 – USB host, 2 – USB device).

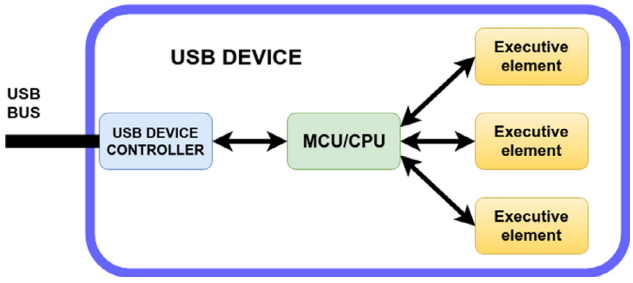

Fig. 3 – Typical USB device block scheme.

The typical USB device block scheme is presented in Fig. 3. USB-enabled mobile devices usually do not use physical executive element but utilize their operating system infrastructure to emulate one. Also the device logic is usually fully implemented in the software. Linux kernel provides even a dedicated subsystem for the USB device logic implementation.

[1] MicroController Unit.

This allows implementing abstract functions relatively easily, when a hardware implementation would not be possible.

In order to put multiple executive elements into a single device, the USB standard allows a single USB device to provide multiple unrelated functionalities. To make them work in an independent manner a single USB device may provide up to 31 independent data pipes. The ends of data pipes on the device's side are called endpoints. They may be understood as an equivalent of ports from the Internet world. The endpoints are identified using a 4 bit number (0–15) and one direction bit. All endpoints apart from the endpoint 0 are unidirectional. Direction of the endpoint is always set from the USB host perspective which effectively means that IN endpoint may transfer the data from the device to the host.

Different functionalities may have different requirements with respect to delays, reliability of delivery, etc. that is why the USB standard defines four different endpoint types based on the typical use cases:

Control: This is the only mandatory endpoint type and the only one which allows bidirectional communication using the same endpoint. This type is reserved for the endpoint 0, so each device may have only one such endpoint. After establishing the connection it is used to discover the USB device capabilities but it must be noted that it can be also used by the application. Due to poor throughput this endpoint type is usually used to send small portions of out-of-bound signaling data.

Bulk: This endpoint type is used to transfer large amounts of the delay-insensitive data. The reliable data delivery is provided but the delays for the data transfer may be significant. This type is widely used when communicating with pendrives and mobile devices.

Interrupt: This type is utilized to transfer small amounts (ca. 10 kb/s) of the time sensitive data. Both the error-free delivery and the maximum delay for the data transfer are guaranteed. This endpoint type is typically used for the Human Interface Devices like a keyboard or a mouse.

Isochronous: This type of endpoint is utilized to transfer large amounts of the delay-sensitive data. Only the maximum transfer delay is guaranteed. In the case of data error there is no retransmissions so the data may be dropped by the host without any notifications. Isochronous type is widely used whenever late data is as good as no data, e.g., for the video streaming.

It is also worth mentioning that the USB standard does not provide any type of endpoint or feature to ensure data integrity or confidentiality. All data is send through the USB in plaintext. Thus it is up to the functional layer to ensure all required security mechanisms.

It must be also noted that the USB standard defines a generic logical structure of the USB device (Fig. 4). All endpoints apart from endpoint 0 are grouped into interfaces. Interface is a group of endpoints which are used to implement some well-defined functionality. All interfaces are grouped into configurations and a configuration is a group of interfaces which can be utilized at the same time. The USB device may have multiple configurations but only one can be active at the given time instant. The USB host may communicate only with

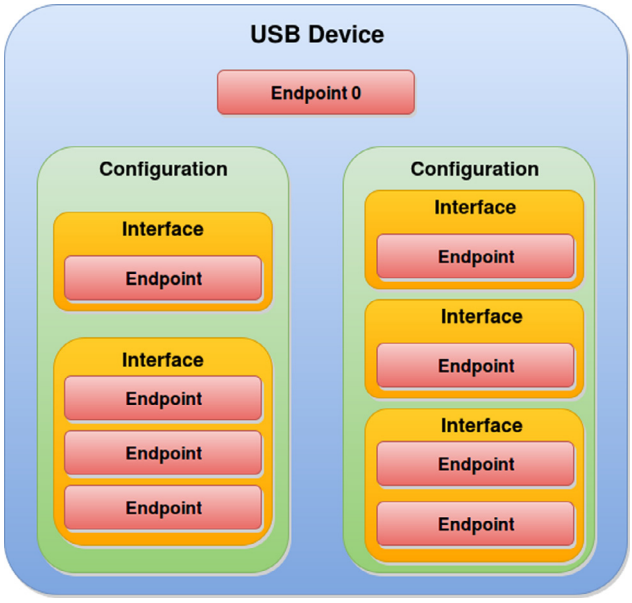

Fig. 4 – Logical USB device structure.

endpoints which belong to the active configuration. As endpoint 0 is not grouped into neither interface nor configuration it is always available for communication.

One of the most famous and appreciated USB features by users is the Plug & Play philosophy implementation. This requires fully automated process of discovery, configuration and drivers probing for each new USB device. To make this possible, the host has to detect capabilities of each new device. In the USB terminology this process is called *enumeration*. Its core part is related to getting information about logical device layout. Each USB entity (device, configuration, interface, endpoint) is described using data structure called descriptor and contains the most important information about the given entity.[2] This information is utilized by the host to choose one of the available configurations and then to select a suitable driver which will make device functionality available for the user. Although Plug & Play feature is very convenient for users, if misused, it may also become a serious security threat (Karsten Nohl and Lell, 2014).

The driver which makes the given USB device functionality available to the userspace is usually just a simple translation layer. Its main task is to transform (for a given OS) generic user calls related to the given functionality to be built on the top of the USB protocol which is understandable by the device. As some functionalities are hard to map to system entities like block devices or network interfaces it is not always convenient to have in-kernel driver. That is why *libusb* (lib, 2017) has been created. Libusb is a library which allows userspace programs to communicate with the device at a USB protocol level.

On the device side Linux kernel also provides infrastructure to implement parts of the USB device logic and the executive element in the userspace. Thus it is possible to create two services: one for the device side and the second for the

[2] The exact structure of each USB descriptor can be found in usb (2000).

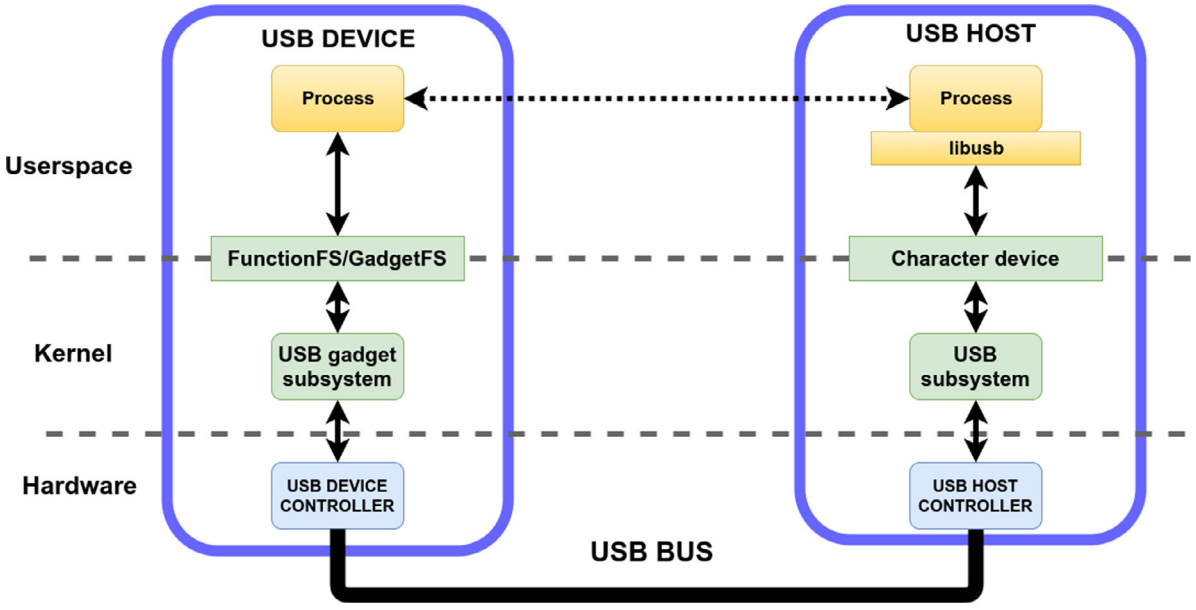

Fig. 5 – Interprocess communication over the USB bus using Linux system infrastructure.

host side. Both will be able to communicate with each other using the USB bus (see Fig. 5).

3. ADB fundamentals

ADB (Android Debug Bridge) is an application layer protocol which is used to enable Android Apps developers to access mobile device resources. Current implementation may use USB bulk endpoints or TCP sockets as a transport layer. In this paper we focus on the USB backend. It must be also noted that most of the presented information is also valid for other transport layers. Detailed description of the ADB communication protocol for all transport layers may be found in adb (2016).

The main goal of the ADB protocol is to simulate multiple data streams using a single bidirectional communication channel. From the USB transport perspective, this means that the ADB protocol uses only one bulk IN and one bulk OUT endpoints which are grouped into a single interface. Each stream is identified by a pair of 32-bits unsigned integers. One identifies the stream from the host side and the other from the device side. Therefore, each stream provides theoretically independent communication channel between the service on a host and the other one on the device side (Fig. 6).

The high-level design of the ADB ecosystem consists of:

adb daemon (adbd): This is a daemon which runs on an Android device. It listens to the requests incoming from the host and executes them.

adb server: This is a daemon which runs on a developer's machine. It uses *libusb*[3] to communicate with an Android device connected via USB. Its main task is to provide synchronization between many independent executions of the adb tool. Adb server listens for requests which should be sent to the device on the localhost TCP port 5037. This daemon has to be run with rights to open the USB device node in the RW (read-write) mode.

adb: This is a command line tool which is directly used by the developer. Its main task is to parse command line call and then to form a request which is then sent to the adb daemon via a TCP socket.

The main reason for splitting *adb* command line tool and the *adb server* is to allow multiple calls of this tool at the same time. It also allows external tools like SDK to reuse existing implementations of the USB protocol. The communication between the *adbd* and the *adb server* is realized using ADB protocol as described later in this section. It is also worth noting that the communication between the *adb* and the *adb server* is performed using different protocol which is out of scope of this paper.

ADB is a message-based protocol and each ADB message is sent in two USB transfers. The first transfer has a static size defined in the protocol and contains a generic header (Listing 1). Then during the second transfer the command-dependent, variable length payload is sent (this part may be omitted as it is optional).

Each field in the header is in a form of 32-bit unsigned integer in the little endian byte order. The field *command* identifies the message type. As every message carries a command these terms can be used interchangeably. The next two fields (*arg0* and *arg1*) contain data specific to a given message type. Then, the *data_length* field contains the length of the data which are going to be sent during the second phase. If it is 0 then the second stage is omitted and the message consists only of a header. The next field is *data_check* which contains a simple checksum of the payload. The last field i.e. *magic* includes the same data as *command* but XORed with 0xffffffff fixed value.

[3] libusb support has been added in v1.0.39. Older version directly utilizes character device provided by the Linux kernel.

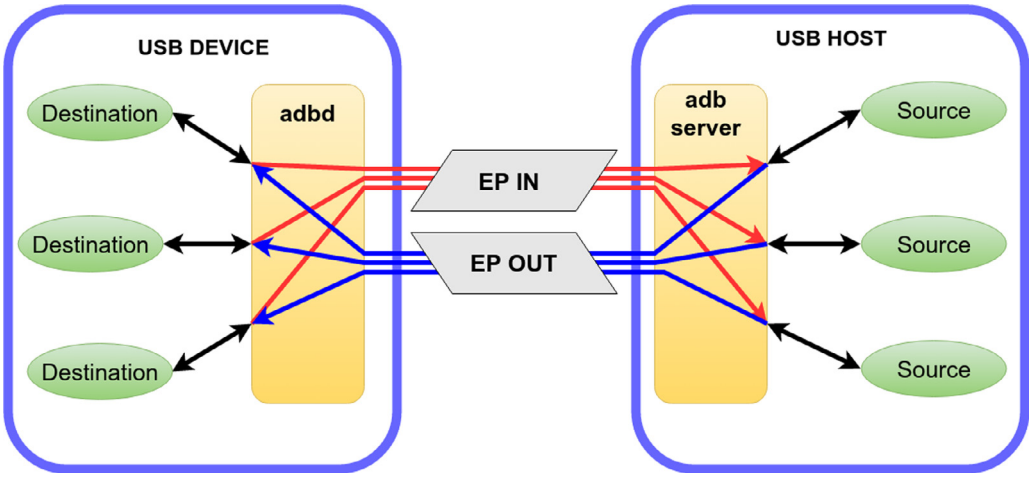

Fig. 6 – ADB streams proxy concept.

Table 1 – Messages defined in the ADB protocol.

Message	Command	arg0	arg1	Payload
CONNECT	A_CNXN	Protocol version	Max payload size	System ID string
AUTH	A_AUTH	TOKEN	0	20 random bytes
		SIGNATURE	0	RSA signature of token
		RSA PUBLIC KEY	0	Host pub. key
OPEN	A_OPEN	Sender side data stream ID	0	Destination[a]
READY	A_OKAY	Sender side data stream ID	Recipient side data stream ID	–
WRITE	A_WRTE	Sender side data stream ID	Recipient side data stream ID	Data to be written to the given stream
CLOSE	A_CLSE	Sender side data stream ID	Recipient side data stream ID	–

[a] Typical destinations are: TCP/UDP/local port, shell, and file.

```
1 struct amessage {
2     uint32_t command;
3     uint32_t arg0;
4     uint32_t arg1;
5     uint32_t data_length;
6     uint32_t data_check;
7     uint32_t magic;
8 };
```

Listing 1 – ADB message header structure.

Message types defined by the ADB protocol are presented in Table 1.

The communication is always initiated by the *adb server*. It waits for the new ADB-capable device to appear on the system and then issues *CONNECT* command. When the device receives a new *CONNECT* message it tries to authenticate a new host which is trying to access it.[4]

The authentication process is done using the classical challenge-response scheme (Fig. 7). The device generates randomly 20 bytes and then send them to the host using *AUTH TOKEN* command. The host digitally signs this received payload using RSA 2048 private key and sends the resulting

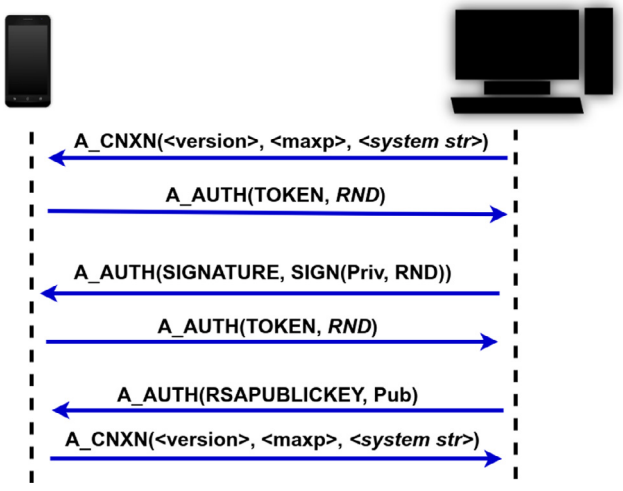

Fig. 7 – Typical ADB challenge-response communication during the process of new host authentication.

signature back to the device using *AUTH SIGNATURE* message. The device checks if a public key which has been used to generate the signature is on a list of trusted keys. If yes, then the host is authenticated and the device responds with a *CONNECT* message. Otherwise the device transmits a new

[4] Android OS versions older than 4.2.2 do not support Secure USB Debugging so they will not perform host authentication – they will just reply with a suitable *CONNECT* command.

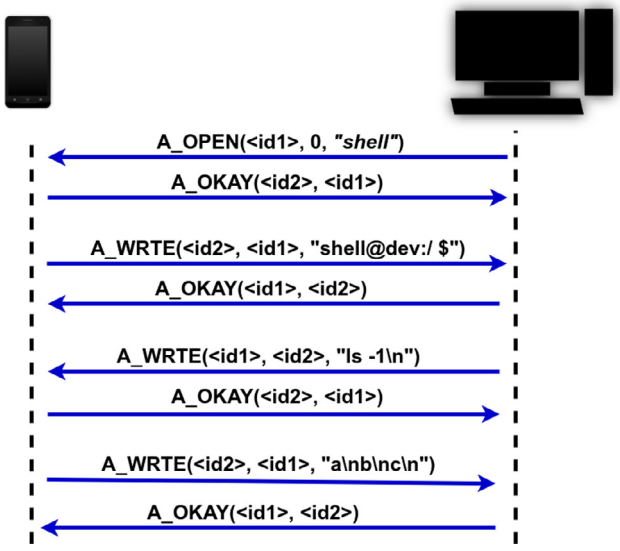

Fig. 8 – The typical message flow during the shell access via ADB.

challenge to the host to give it another chance to use different signing key. When the host runs out of its signing keys then it responds to the *AUTH TOKEN* with *AUTH RSAPUBLICKEY* which contains the public key associated with one of its private keys. After receiving this message, the device calculates MD5 fingerprint of the public key and generates a pop-up for a user to decide whether a host which owns such a key should be allowed to access device resources. If the access is granted, the device responds with the *CONNECT* message, otherwise the connection is closed.

When both sides of communication agree to create a session then the host obtains a full user mode access to the device's resources. It means that it is able to run all applications and access user files or even send input events. All these actions can be achieved using streams abstraction defined by the ADB protocol.

To obtain the access to some device's resources or services the host has to create a new data stream. This starts by choosing (on the host side) some unused identifier and determining the destination. Both local id and the destination are sent to the device in the *OPEN* message. If a given destination is available at the device side, it associates some unused device side identifier and replies to the host with the *READY* command.

To send the data using a stream, one of the communication sides issues the *WRITE* command with the suitable stream identifier and the content which should be written to that stream. The recipient of such a message passes the data to the suitable dispatcher and then replies with the *READY* message. In order to close the data stream one of the communication sides has to issue the *CLOSE* message. An example of the typical message flow during the shell access has been illustrated in Fig. 8. Thanks to such data streams abstraction, the ADB protocol is very flexible and allows to implement even quite complex mechanisms like application debugging or sensors emulation.

4. Related work

As previously mentioned USB is currently the most commonly used external interface. Even machines which, for security reasons, are disconnected from the Internet, usually offer USB host functionality. Popularity and blind trust in the USB security seem to be one of the major reasons why malware started spreading using this attack vector.

From this perspective existing attacks exploiting USB can be categorized in many ways. In Tian et al. (2018) categorized host-related attacks based on the layer which they exploit. As in this article we consider broader spectrum of the USB-related attacks we can divide them into one of three groups:

- USB host-focused attacks,
- USB traffic analysis, modification and injection attacks,
- USB device-focused attacks.

Below we review related work using above mentioned ad-hoc classification.

4.1. *USB host-focused attacks*

This type of attacks aims at taking over the control of the USB host machine. We can distinguish three subgroups of such attacks.

The first group utilizes vulnerabilities in the high level system infrastructure related to support of the USB mass storage devices. The archetypal example of such a threat is Conficker (Hypponen, 2009), which was targeting mostly Windows machines. To infect the victim, Conficker used autorun files which had been automatically executed by Windows whenever new removable medium was mounted. Another and probably the most recognizable representative of this group is Stuxnet (Nicolas Falliere and Chien, 2011). Its targeted victims were PLC controllers manufactured by Siemens, however, as most of them were typically isolated from the global network, Stuxnet needed a way to get into private networks. Thus it infected Windows machines using custom-crafted content of the USB flash drive. The exact infection method was similar as in Conficker but included also new vulnerabilities like CPLINK (cpl, 2010). A lot of vulnerabilities of this type have been collected and described by Larimer (2011).

Both Stuxnet and Conficker used USB flash drives only as just a kind of transport medium for the exploit itself. They did not interfere with the USB traffic directly but only placed malformed files on the removable media. That is why, now they are likely to be detected by most of the anti-virus software as they are not that much different than other malware which spreads over the Internet.

The second group of the USB host-focused attacks uses vulnerabilities in the USB stack implementation. A good example of such attacks has been presented by Barral and Dewey (2005) during BlackHat Conference in 2005. They found a buffer overflow vulnerability in Windows 2000 and Windows XP USB stack that allowed to take control over a locked machine. As this is a kernel-level vulnerability it can be used to install malicious code bypassing SELinux and all other types of kernel-side protection mechanisms. Currently USB stacks are

more resistant to such attacks. There are even fuzzers (uma, 2013; fac, 2017) available which help to find vulnerabilities in the inspected USB implementation.

Finally, the third group of attacks abuses Plug & Play philosophy which is one of the USB principles. This group of attacks became known thanks to the presentation by Karsten Nohl and Lell (2014) during BlackHat Conference in 2014. These attacks are based on a fact that much of the USB devices' logic is implemented in the firmware which can be replaced. It turned out that many USB devices are being shipped to the market with unlocked and unverified firmware upgrades. This, in turn, allows malware to replace firmware in, e.g., a pendrive (Karsten Nohl and Lell, 2014) or a mouse (Maskiewicz et al., 2014) and makes them behave like some other, malicious device, e.g., a keyboard or a network card. Unfortunately, users do not realize that there is no relationship between physical appearance of the device and provided functionality and they often plug in devices even if found on a street (Tischer et al., 2016). Most of attacks in this group utilize only USB HID protocol to execute commands as currently logged in user. But as Kamkar (2016) proved by publishing PoisonTap tool it is also possible to exfiltrate hashes containing user password from the Windows-based machine and crack them to bypass the lockscreen. As Wang and Stavrou (2010) showed, mobile devices can be also used to prepare this kind of attack. There is also a Kali Linux NetHunter toolset (kal, 2017) which is prepared for the Nexus and OnePlus devices which also enables them to be used for the BadUSB-like attacks. It is also worth mentioning that not only smartphones or off-the-shelf USB devices can be used for such attacks. As Mengs proved by publishing P4wnP1 toolset (Mengs, 2017) also a very cheap single-board computer dedicated for IoT (Raspberry Pi Zero in this case) can be extremely handy for the attacker purposes. Such a modified pendrive or a mobile device stays undetected by the anti-virus software and may covertly insert keyboard keystrokes to execute custom commands on a victim machine. To the authors' best knowledge, currently there is no perfect method of protection against such attacks. The most popular approach implemented, for example, in usbguard (Kopecek, 2016) is to ask user for the explicit confirmation to use each new USB device.

It is worth emphasizing that USB can be used not only to take over the control over the host machine but also to exfiltrate data. Many machines which store valuable data (for example BitCoins) are disconnected from the Internet but still they provide USB host capabilities. Obviously in this case the attacker can just use a simple pendrive to store the data but this may be monitored or event restricted by OS policy. That is why there is a need to create a secret communication channel which allows stealthier data exfiltration (Guri, 2018; Guri et al., 2016).

4.2. *USB traffic analysis, modification and injection attacks*

This type of attacks aims at discovering user password or at modification of the USB traffic to abuse functionality expected by the user.

The first subgroup of these attacks includes passive keyboard keystrokes listeners. Such attacks typically consist of a relatively simple and small hardware which is only capable of passively recording Human Interface protocol which is sent over the USB. These hardware keyloggers are intentionally placed in the public places like it was done in a library in Manchester (man, 2011) or at school in California (Cal, 2004). This type of device is usually utilized to obtain users' credentials which are being entered from the USB keyboard.

The second subgroup of such attacks is able to perform an active Man-in-the-Middle (MitM) attacks. Thus they are not only listening to the USB communication but are also able to modify it before reaching the host. To the authors' best knowledge there is no evidence that currently malware is spreading using this attack vector but due to the lack of any security mechanism it may be utilized by military and/or government agencies (usb, 2008). This kind of attack may include a simple modification of the data being sent between the host and the device. A good example of such an attack would be to replace the last digit in One Time Password (OTP). This means that from a user perspective an authentication fails but at the same time the attacker can reuse the original OTP to authenticate his own session. In result, an injection of some USB messages is possible. An attacker is not only able to modify messages but he is also capable to inject ones, too. An example of this attack would be to inject some custom keystrokes after detecting that the user has logged in as a root. Finally, such attacks can involve additional USB device emulation. Thus the attacker may not only be influencing the USB-related traffic but he is also able to inject some additional USB devices to the system. A good example of this scenario would be to hide some USB device and a radio transceiver within the USB hub to exfiltrate the confidential data in a wireless manner.

Currently there are also a couple of tools available which can be used to prepare such attacks. TURNIPSCHOOL (tur, 2015; Dominic Spill and Boone, 2015) project aims to recreate NSA device which description was published by WikiLeaks. It is a very small board which easily fits into the standard USB connector. It is equipped with a USB hub and a chip which allows to emulate some custom USB device while the user device is still visible to the system. Moreover, it has also built-in radio transceiver to enable wireless communication.

Another example is BadUSB 2.0 introduced by Kierznowski (2016). It uses *Facedancer* MitM design proposed in van Tonder and Engelbrecht (2014). It allows to modify HID traffic and use it to exfiltrate the data from the USB host. This enables to not only input some custom commands like in the BadUSB attack but also to receive their results. Unfortunately, BadUSB 2.0 supports only full speed devices which limits its practical usage only to HIDs as they transfer relatively small amounts of data.

There are also projects like USBiquitous (Camredon, 2016) and USBProxy (USB, 2014; Spill, 2014) which aim at proving USB MitM framework for the standard Linux Single Board Computer (SBC). Even though their overall goal is the same they represent two different approaches. USBiquitous provides its own set of custom kernel drivers and communication mechanism based on which the userspace API for the MitM attacks is built. In contrast, the USBProxy aims to use standard Linux infrastructure for both: USB device communication and emulation. This makes USBProxy only a userspace library which utilizes libusb for communication with the attached USB

devices and GadgetFS to communicate with the USB host. The main limitation of both projects is that they allow to interact only with a single USB device. This is because of a hardware limitation of UDC which can be found in single board computers.

Finally, there is also a Daisho project (dai, 2013; Dominic Spill and Kershaw, 2013). It is an FPGA platform capable of Super Speed USB communication in the USB device mode. Its main design goal was to provide a platform for intercepting the most popular wired protocols. It is extendable with hats which are designed separately for each protocol like Ethernet, HDMI or USB. Unfortunately, this project seems to be still in its infancy and not much development has been observed during last year.

4.3. *USB device-focused attacks*

Nowadays, mobile device security frameworks are much more sophisticated than the ones on PCs. Application isolation, dedicated application permissions – all these concepts show that there is a huge need to protect user private data from the unauthorized access.

One of the typical ways to access user private data is to use USB connectivity in the device mode. By default all Android devices provide access to the internal memory using Media Transfer Protocol (MTP) (spe, 2011). However, it is not possible to access the device memory without unlocking the screen and accepting the MTP connection. In addition, since Android version 5.0 it is possible to put a mobile device in the charging only mode which prevents access to the device resources even after screen unlocking.

It is also worth noting that the MTP is not the only way of accessing mobile device. There is also Android Debug Bridge which provides many more features than just multimedia file transfer. Fortunately, it is disabled by default and has to be explicitly enabled in the Developer Options. To prevent unauthorized access via ADB protocol, when it is enabled, Secure USB Debugging has been introduced from Android version 4.2.2 (Elenkov, 2013).

Unfortunately, to authors' best knowledge there is no comprehensive evaluation of the introduced security model and the ADB protocol itself. There has been also not much research related to the assessment and potential improvements of the ADB security.

However, it must be emphasized that there are some reported vulnerabilities which allow to get unrestricted root mode access to some OnePlus devices (adb, 2017). These security holes (and also those identified in Hoggard, 2014) are related rather to the mobile phone misconfiguration than the ADB protocol vulnerabilities.

Also Xu et al. (2015) aimed at improving ADB security. Their attack vector is related to the malicious applications on a host side which could communicate with the adb server and spread malware to the mobile device. The main risk identified by them is utilizing an ADB protocol to root the mobile device (dir, 2016; Vidas et al., 2011). Unfortunately, their solution requires a lot of user interaction and has never been accepted into the adb mainline.

4.4. *Paper's contribution*

Most of the USB-related attacks focus on obtaining access to the USB host or user passwords. As already mentioned there are not many considerations on the USB devices security even though mobile device often contains sensitive and private data. This paper aims at filling this gap by evaluating ADB security model from the USB device security perspective. To the authors' best knowledge this is the first paper that tries to assess ADB security model in terms of passive and active MitM attacks.

The threat model considered here is focused on using malicious USB hubs that can manipulate data which is being exchanged between the host and the device. Modern computers are shipped with a very small number of available USB ports. In contrast to that Android developers often require to connect multiple USB devices so obviously they need to use USB hubs.

Our work resulted in the identification of five, previously unpublished, security vulnerabilities. They have been reported to both Samsung Mobile Security team (Samsung, 2017) and to the Android Security Team (Android, 2017). Unfortunately, Android Security Team decided that the reported vulnerabilities are not an issue because they are exploitable only in developer's mode and Android Team never claimed that this mode is safe. This means that Android apps developers security is effectively on their own. Therefore, by publishing discovered vulnerabilities we would like to rise awareness across ADB community and increase mobile devices' security.

The paper also discusses the root causes of these bugs which can help developers to avoid the same mistakes in their protocols. Finally, this paper introduces a novel tool *adb-mitm* which exploits identified vulnerabilities and which can be used to assess ADB security improvements. To the authors' best knowledge this is the first tool that uses USBProxy framework to find and exploit security bugs in the USB devices.

5. Experimental test-bed

Exploiting USB smartphone connectivity may be achieved in various ways. Nevertheless, we decided to focus our research on the MitM attacks because this allows to find bugs which are hardware independent and present in most Android devices on the market.

For our experiments as the target devices we chose Samsung Galaxy S7 (SM-G930F) and HTC Nexus 9 (OP82100).

The former is a very popular Samsung flagship model which is have been sold in very large quantities. Using a flagship model guarantees that its vendor invested a lot of efforts in its development and that (most probably) it has been well tested. Thus finding a bug in such device is a challenging task but on the other hand it, additionally, confirms that our experimental approach is novel. During our tests the device was running Android 8.0 with the build number R16NW.G930FXXS2ERG7.

The latter is a HTC device officially supported by Google and it runs pure Android OS. Using a pure Android device enables to confirm that the bug has not been introduced due to manufacturer's modification of the ADB source code but is

also present in the Android mainline. For our tests we used Android ROM marked as LRX22C.

Apart from the target devices it is also necessary to utilize a suitable hardware to interfere with the USB communication. As described in Section 4.2 there were several options to consider and we chose the one which fits our needs best.

Both Android devices used in our test-bed support USB communication in the high speed mode. That is why, we decided not to follow *Facedancer* MitM design as it is limited only to the full speed mode. A perfect solution would be to use a hardware which can be placed in any part of the USB tree and is able to play with multiple USB devices at the same time. From this perspective it seems that *Daisho* board with a suitable HAT would be the best choice. Unfortunately, as already mentioned earlier it seems that this solution is still not mature enough to be easily applied to our scenario.

We did not want to build our own hardware from the scratch so after resigning from the *Facedancer* and *Daisho* approaches we decided to use one of the available SBCs. In terms of software, in order to start our work, we could utilize USBProxy or USBiquitous. After inspecting both solutions we decided to select USBProxy because it uses standard Linux interfaces instead of the some custom drivers. This makes it significantly more portable and easier to perform debugging.

USBProxy requirements related to the SBC are as follows. First of all, the board has to run reasonable, non-archaic version of Linux and has to be equipped with both Host and Device controller. We decided to use the same board as USBProxy author – BeagleBone Black (BBB) (bbb, 2017) as it is quite inexpensive and popular.

In our setup BBB is running microSD card image prepared by Dominic Spill for 2014-03-R1 USBProxy release (USB, 2014). After some updates it was running Debian GNU/Linux 7.3 (wheezy) with 3.12 Linux kernel.

In our experimental test-bed we also utilized two PC machines. One of them was used as a victim PC and the second one as an attacker machine. Victim machine was running both Windows 7 and Ubuntu Linux 14.04 to confirm that the discovered issues are host OS-independent. ADB version on a host was initially 1.0.31 and then also the most recent (in the time of writing this paper) version 1.0.39 during the final stage of our research in order to confirm that all vulnerabilities are still present. The attacker machine was running Ubuntu Linux 14.04 with ADB 1.0.31.

The utilized wiring scheme has been illustrated in Fig. 9. The mobile device has been connected to the BBB USB host port. BBB device port has been connected to the victim's PC host port. This allowed to create the main communication channel on which we were listening and modifying the passing communication. The communication between the attacker machine and the BBB was realized over the Ethernet network.

To intercept USB traffic we utilized the most recent version of the USBProxy. As the latest release was issued quite a long time ago we decided to use current HEAD of master branch which at the time of writing was: 4fc16ed86a1c60a391a183e9b3f14251e8c1118c.

To intercept ADB traffic we developed our own tool called *adb-mitm* on a top of the USBProxy. It uses plug-in API to provide both PacketFilter and Injector functionalities. Note that PacketFilter functionality allows to monitor and to modify ADB traffic while Injector enables injection of custom messages. The block diagram of the created tool is presented in Fig. 10.

6. Security analysis, discovered vulnerabilities, and lessons learned

During our experiments we have discovered five novel vulnerabilities in the ADB protocol. All vulnerabilities have been confirmed on both 1.0.31 and 1.0.39 ADB versions on a host side and on both 6.0.1 and 7.1.1 versions of Android OS. They turned out to be independent also from the host side OS (Windows and Linux).

From the security perspective, discovered vulnerabilities may be divided into three categories based on the attack type that they enable:

- Session hijacking,
- Data modification,
- Data injection.

Below we describe identified vulnerabilities using the above mentioned ad-hoc classification. It is worth noting that all described attacks can be easily reproduced using the introduced in this paper *adb-mitm* tool.

6.1. *Session hijacking*

Secure USB Debugging introduced in Android 4.2.2 added a requirement that each ADB session has to complete the authentication phase. As previously described it is typically achieved using a challenge-response scheme. This type of attack aims at hijacking an ADB session of the *previously* authenticated host.

It is worth recalling that the user may be authenticated in one of two ways depending on the scenario.

Firstly, the user can be authenticated when the mobile device is connected to a given host for the first time or the user does not allow to trust this host every time the device is connected to it. This process starts when the device receives A_AUTH RSAPUBLICKEY message from the host. Then the user authentication is performed by unlocking the device and accepting a pop-up with the MD5 checksum of the received public key. It is obvious that to use this method an attacker would need to unlock the device which may reveal his presence.

Secondly, the authentication scheme is utilized when the mobile device is connected to the host which has been previously trusted by the user. It means that the device should verify that the host which would like to start the ADB session owns a private key associated with one of the public keys marked as always trusted. This process is initiated when the A_AUTH TOKEN message is sent and stops when the host responds with an RSA signature (A_AUTH SIGNATURE) that can be verified with one of the trusted public keys. This authentication process does not require any user interaction and may be executed whenever adbd is running on the device. It

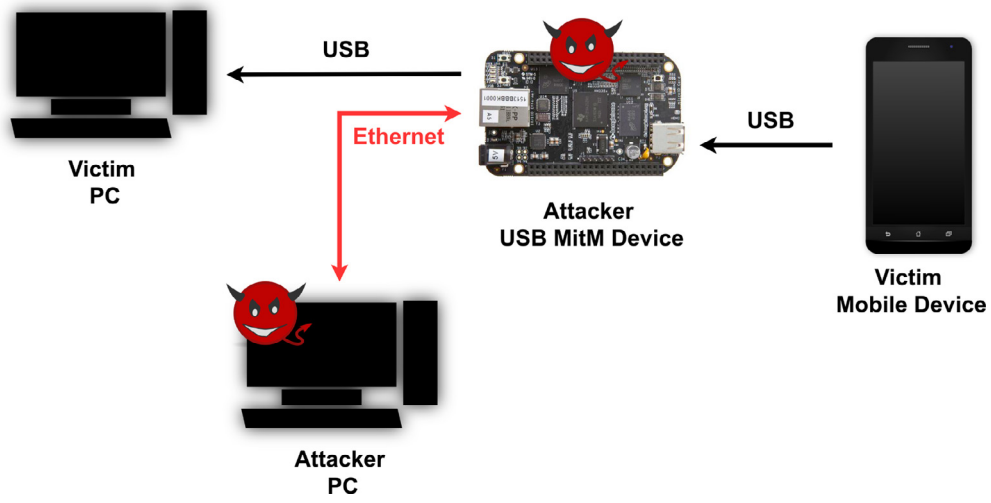

Fig. 9 – Experimental test-bed wiring scheme. In case of the USB connection an arrow points always to the USB host side.

is worth noting that when the USB debugging option is enabled adbd is running automatically, each time the user plugs in USB cable. There is no pop-up displayed like in the case of the MTP where it happens each time the user connects the device. Therefore it is potentially a perfect attack vector to bypass the lock screen and get full access to the device.

First, we tried to perform the replay attack. Fortunately, thanks to randomness of 20 bytes in the A_AUTH TOKEN challenge message this attack cannot be successful. However, the ADB protocol traffic analysis revealed that the A_AUTH SIGNATURE is the only message which has to be generated using suitable private key. There is no data integrity mechanism in any other step of the ADB communication. This effectively means that after receiving the A_AUTH SIGNATURE message with a valid signature mobile device will grant access to any sender of the message in this communication channel even if it does not own the corresponding private key. Therefore, the only thing that is needed to bypass the screen lock is to acquire a valid signature from the trusted host machine.

This in turn can be easily achieved using our experimental test-bed. The only thing that needs to be achieved is to use USBProxy and allow the whole authentication process to complete successfully. Then everything is performed automatically and when the host is authenticated we may just disconnect the BBB from the host. As we disconnect only BBB from the host but not the device from the BBB, the mobile device assumes that it is still communicating with the same machine that provided a correct signature during the initial authentication process. Thus BBB is now able to send any custom command which effectively gives access to all user's private data without unlocking the device screen. The communication flow for this attack has been presented in Fig. 11.

Another a bit more sophisticated version of this attack but still derived from the previous one would be to replace MitM device with just a device emulator.

Instead of getting in between the mobile device and the trusted host, it is possible to use BadUSB-like attack to create a device which will act as an Android-based phone and will send authentication requests to the legitimate host. The scenario of this attack would be to connect the victim device to the attacker's computer. When the A_AUTH TOKEN is issued, it is only necessary to forward this message using different communication channel, e.g. WiFi, to the emulated device. Then this device just issues the same message to the trusted host and capture a valid signature for this challenge. This signature can be send back to the attackers PC and then to the victim's device. As a result the device will authenticate attacker's PC as a trusted host and will grant access to all device resources. The communication flow for this attack has been illustrated in Fig. 12.

Obviously this attack has several limitations. The first is that it requires USB debugging to be activated on the mobile device. Secondly, it involves physical access to the USB host which has been previously trusted by the victim. Lastly, it entails adb server to be active or auto starting when the suitable USB device arrives on a legitimate USB host machine. As neither adb server nor adb daemon check if the machine is locked or not during the authentication phase, both the victim's mobile device and the host PC may be locked.

Lesson learned #1: the root cause of this vulnerability is due to the lack of session integrity mechanism in the ADB protocol. It means that the ADB protocol is simply unable to guarantee that the message sender is the same entity that owns the trusted private key.

6.2. *Data modification*

This type of attack aims at (mostly) unnoticeable message modification, usually to perform some malicious activities e.g. to infect a phone or a host.

The basic idea of this attack is quite straightforward and simple. Neither USB protocol provides any integrity mechanisms nor ADB protocol does. This allows MitM device to modify the USB traffic between the host and the device.

The modification of the USB traffic may occur in one of two phases: during the session creation (host authentication) or after it is completed.

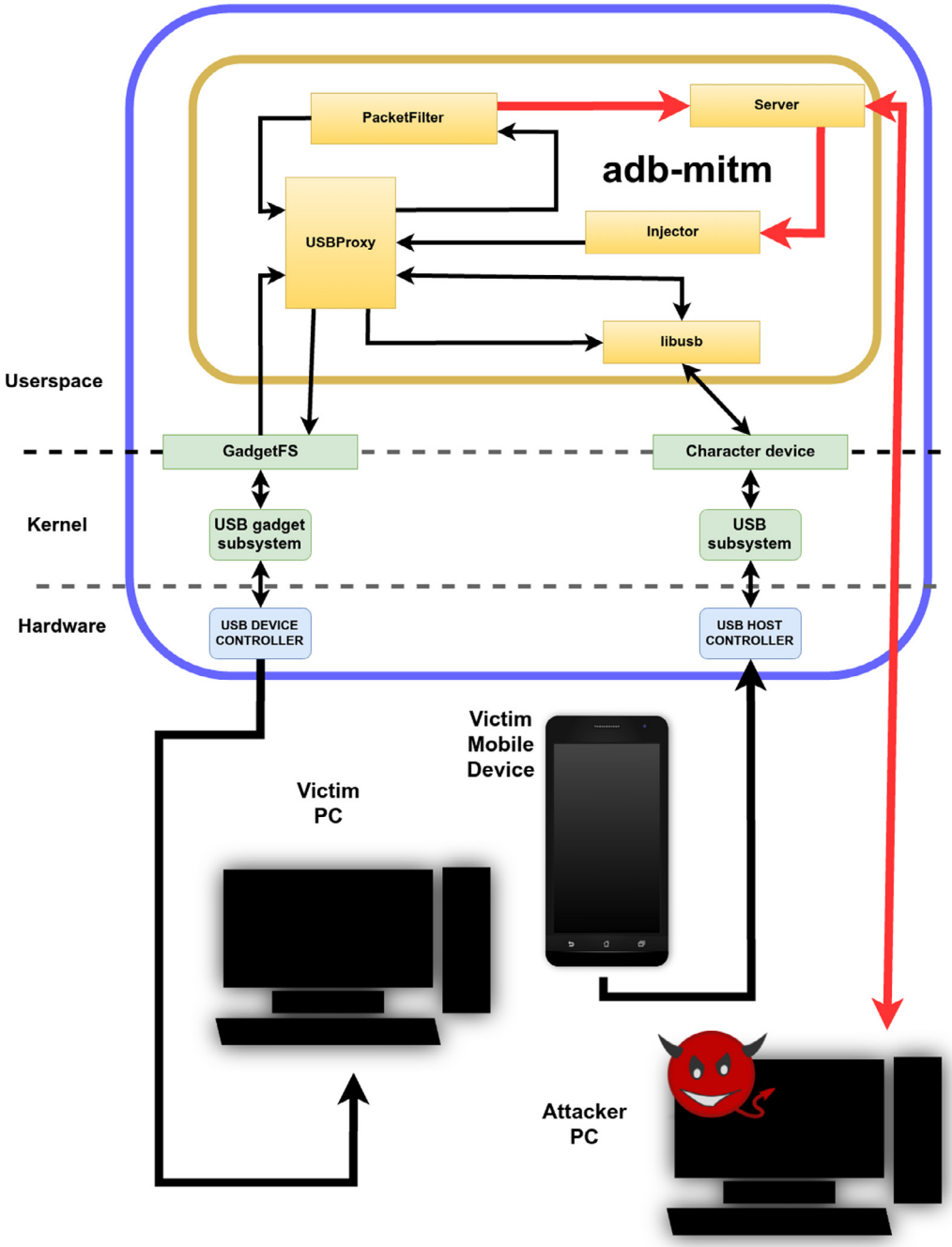

Fig. 10 – Adb-mitm tool block diagram.

Modifications during session initialization may be used, for example, to trick the user to accept some external RSA key as trusted. This may be achieved by replacing RSA public key sent by the host in the A_AUTH RSAPUBLICKEY message with some key provided by the attacker. Obviously, the user should check MD5 checksum and verify whether it matches MD5 checksum of the ADB public key on a host and decline to accept it. Unfortunately, many developers do not pay enough attention to this step and may accept the fake key. As a consequence an attacker can get full ADB access to this device whenever he manages to connect the USB cable to it and ADB is enabled. Thus such an injected public key effectively becomes a kind of the backdoor which opens the device for future attacks using, for example, the fake charging stations. It is also worth noting that after accepting the public key by the user no message is being sent to verify that the host owns the associated private key. This allows attacker to safely spread his public key across many devices without the risk of his private key being compromised.

Lesson learned #2: this "vulnerability" is possible by exploiting human factor. Most of the inexperienced users do not verify their host key fingerprint before accepting it. They find it

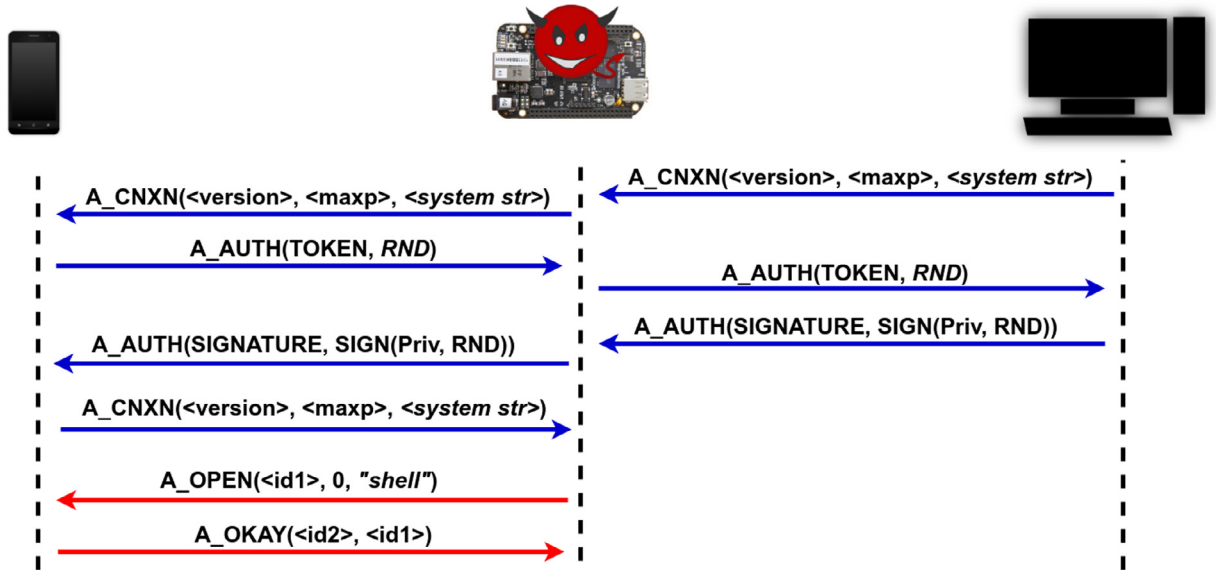

Fig. 11 – Message flow during MitM ADB session hijacking attack.

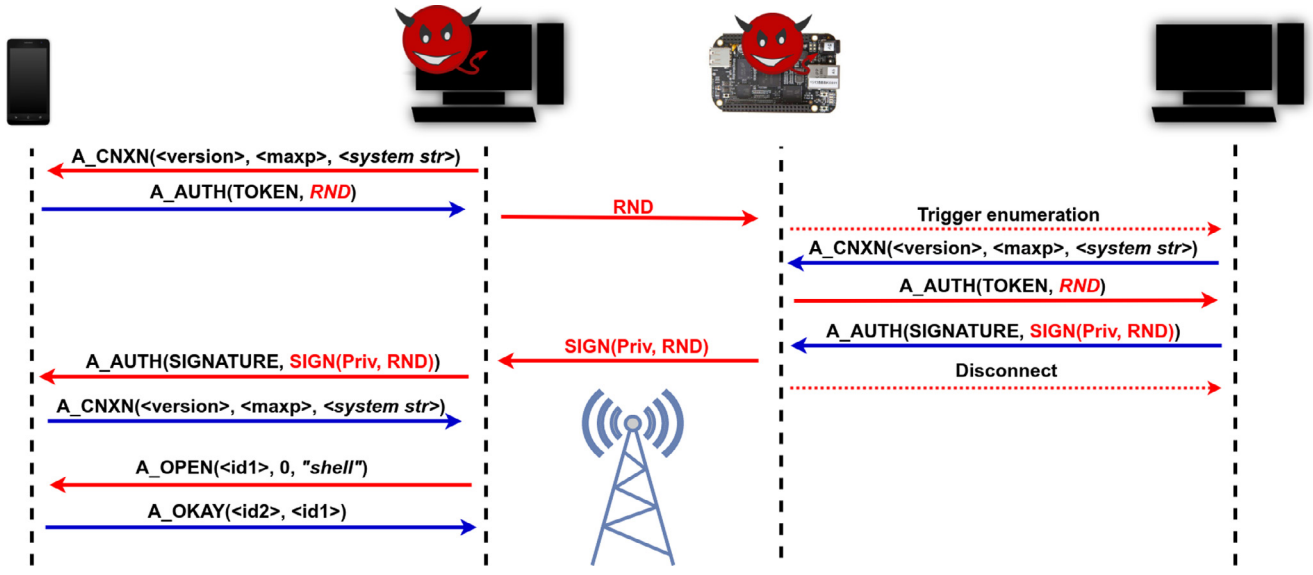

Fig. 12 – Message flow during remote the ADB session hijacking attack.

complicated, as it is not possible to get this fingerprint easily using adb command line tool. If this would be as easy as executing adb fingerprint command then most probably more developers would use it. Even better results could be achieved by replacing user unfriendly MD5 checksums with the verification of the random art of the host key or QR code usage. This could improve users' experience and encourage more of them to verify if what they are accepting is really correct.

Modifications of the data being sent during the ADB session may be used in many ways. The first example of such an attack may be injection of some malicious payload into files or shell commands sent over the USB bus. This involves mostly modification of the files transferred over the bus to infect them with some malware that is able to exploit other bugs in the Android or PC OSes. A good example would be to inject exploits into pictures, pdfs or modify binaries to put there some custom code which could cause a security breach (Cabaj et al., 2018). Finally, it may even modify shell commands to exploit some vulnerabilities in the Android shell (which can be considered as a kind of Shellshock attack variant, Response, 2014).

The second example is based on the ransomware-like (ran, 2016) idea which is (unfortunately) currently gaining more traction among cyber-criminals. In the beginning the attacker may just want to analyze the traffic to identify resources which are important for the victim. Then, when the user tries to access these critical resources once again (for example the list photos), his request may be replaced with the one that encrypts this content. A simplified scenario of such an attack which removes all photos instead of listing them has been

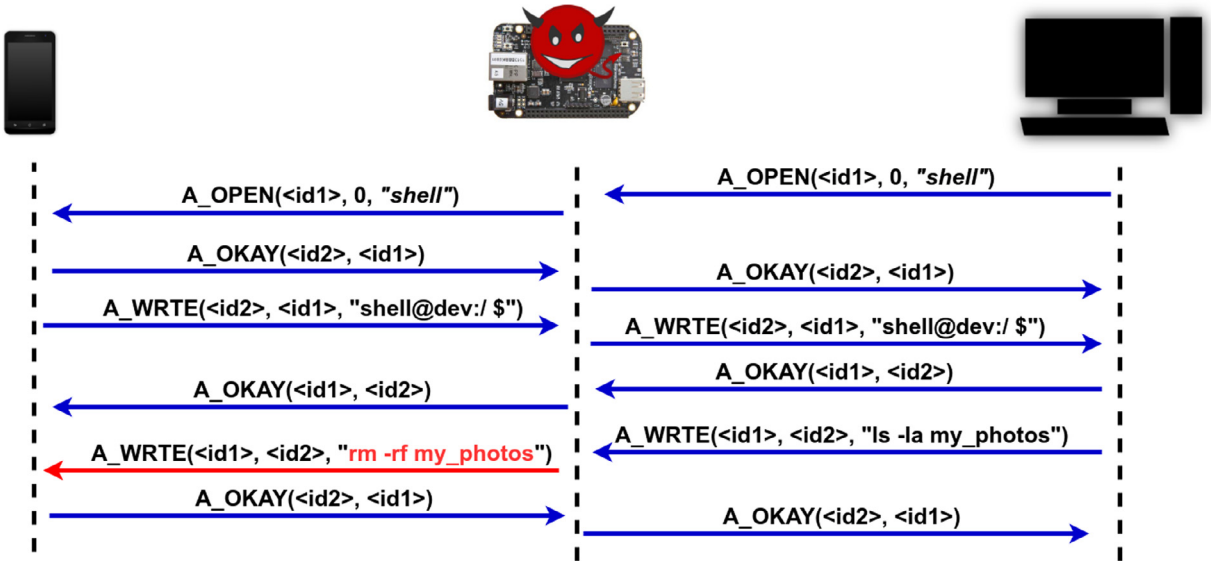

Fig. 13 – Message flow during the ADB stream data modification attack.

presented in Fig. 13. To make the adb-mitm tool more flexible it does not provide any of these scenarios but instead it just provides a plug-in framework which allows everyone to implement their own data change algorithm.

Lesson learned #3: the root cause of this vulnerability is once again the lack of session integrity mechanisms in the ADB protocol.

6.3. *Data injection*

This type of attack aims at unnoticeable injection of the custom messages into authenticated ADB sessions.

The basic concept of this attack is to allow the authentication phase to complete successfully just like in the case of session hijacking attack. However, after this phase, the mobile device is not being disconnected from the victim PC but it is still available and usable. This means that the user may utilize the device via ADB and our MitM device can inject some messages to provide parallel access to the device to the user and to the attacker.

The key ADB functionalities such as shell access, remote debugging, apps installing is provided using the base streams abstraction via the ADB protocol. Data is being sent across the stream by issuing the A_WRTE command by the host or the device. Therefore, the first idea to abuse ADB protocol and inject some custom data is to inject some A_WRTE command to, for example, opened shell stream. This allows to inject arbitrary shell commands to the active shell section without host even noticing this.

As injecting content to the already opened user streams has limited functionality and requires streams to be opened by the user, this attack can be extended by opening new streams, not requested by the host (Fig. 14). This enables attacker to get access to the similar functionality as the user has without host PC even noticing this. The additional requirement to make it works is to provide some kind of the ADB streams proxy on the BBB to ensure stream-id uniqueness between the victim's host and the attacker and to provide some routing policy for the device responses.

Lesson learned #4: Once again the root cause of this vulnerability is the lack of integrity mechanisms in the ADB protocol itself. It must be emphasized that currently no mechanism exists which could enforce opened streams integrity.

The final extension of this attack aims at not only accessing the device now but also at installing custom RSA keys as a backdoor to access the device in the future. As ADB allows injecting commands to manage data streams it should be determined whether it would be possible to inject also A_AUTH commands to the already authenticated session. Our experiments revealed that the adbd does not check the protocol state and executes the same set of actions on the A_AUTH command recipient even if the session is already authenticated. This allows to inject the A_AUTH RSAPUBLICKEY command and cause a pop-up appearance on the device screen. Obviously, under normal circumstances accepting such a key would require user interaction but as the ADB session has been already authenticated it is possible to simply send suitable input events to approve this key without user interaction. The only limitation of this attack is that the user has to unlock the screen for the time of key approval. To force him to do so, an attacker may simply wait for the legitimate A_AUTH RSAPUBLICKEY message transmission from the host and inject his own key shortly after the genuine host key has been approved by the user. Everything may happen so quickly that the user may even not notice that something appeared on the screen.

Lesson learned #5: this vulnerability seems to have two main root causes. The first is the lack of integrity mechanism in the ADB protocol. The second is the absence of protocol state checking at the A_AUTH messages recipient. If the session has been already authenticated then adbd should not take actions on any A_AUTH command or at least invalidate session authentication when such a message arrives. The lack of such sanity check leads to a situation where a single

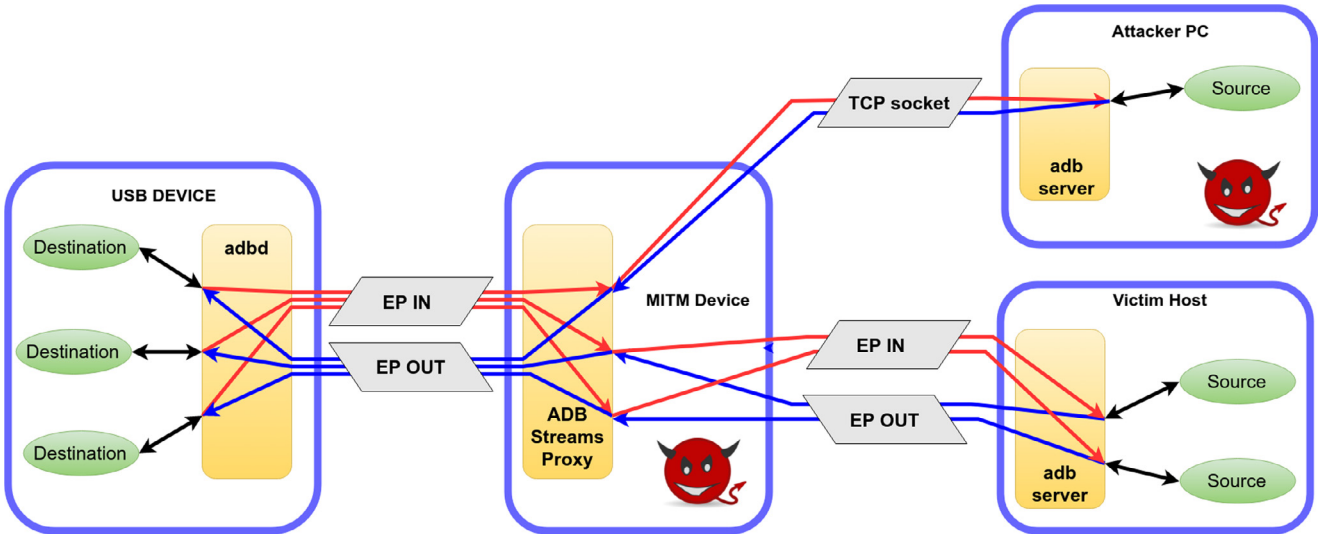

Fig. 14 – ADB streams proxy.

entity is able to do both: trigger the authentication pop-up and then approve it.

7. Countermeasures

In the previous sections we evaluated the ADB security model used in the recent Android versions and discovered vulnerabilities related not only to implementation but also to serious issues in the ADB protocol design. In-depth analysis of the vulnerabilities allowed us to identify two root causes.

First of them is the absence of mechanism which would provide integrity in the ADB protocol. As stated earlier USB does not provide any integrity solutions which means that USB traffic can be easily modified without raising suspicions. The ADB protocol is also not equipped with integrity mechanisms which effectively allowed us to execute the attacks described in this paper.

Second root cause is an automated session authorization and the lack of the private key protection. ADB server owns an RSA private key which is used to authorize host to the device. Unfortunately this key is automatically used to authorize host to any new ADB-capable device event when the machine is locked.

Below we propose improvements to the ADB protocol which can be incorporated to protect against discovered security flaws.

7.1. *ADB protocol extension*

To protect against attacks presented in this paper applying modifications to the ADB protocol is required. Thankfully, the CONNECT message which initiates the communication contains protocol version which can be increased to notify communication peer about the changes. The main goal of the introduced modifications is to provide state-of-the-art integrity mechanism to prevent MitM attacks. One of the major

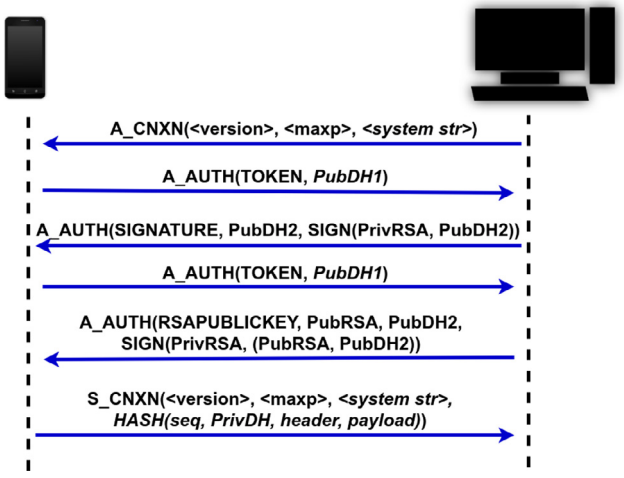

Fig. 15 – The modified challenge response scheme.

requirements is to ensure possibly the best data transfer and low computation cost to avoid ADB performance degradation.

After in-depth analysis of the above mentioned requirements we concluded that the secrecy of the transmission is not a primary goal but first of all integrity must be provided. Thus we decided to use HMAC (keyed-Hash Message Authentication Code) (Krawczyk et al., 1997) due to its simplicity and low computation cost. To be able to use HMAC as an integrity mechanism, shared secret key is required. To ensure forward secrecy this key should be generated on per session manner. For this reason, we decided to use D–H (Diffie–Hellman) key exchange algorithm to generate the secret key.

To incorporate D–H algorithm several changes in the ADB protocol must be incorporated (Fig. 15). First of all, when the device receives the CONNECT command it should generate first D–H public-private key pair and it should send the public key to the host instead of random 20 bytes in the AUTH TOKEN message. When the host receives this message it should

```
1 struct samessage {
2     struct amessage amsg;
3     uint8_t mac[SHA1_HASH_SIZE];
4 };
```

Listing 2 – Extended ADB header structure.

generate the second key pair and reply with the AUTH SIGNATURE message which contains the second public D–H key and its RSA signature derived using host's private RSA key. After obtaining this message the device tries to verify the signature using one of the trusted public keys. If it is not successful then it sends the AUTH TOKEN message once again to give host the chance to use another RSA key. If the host runs out of the public keys it should reply with the AUTH RSAPUBLICKEY message which contains both: the second D–H public key and the RSA public key, signed using the corresponding RSA private key. When the device receives such a message it generates a pop-up window which presents MD5 sum of the host public key to be verified by the user. If the user accepts the public key or if it has been previously authenticated then the D–H key negotiation is completed. Both sides now share the same secret and the device knows that the second part of the key has been generated by the trusted host. As the HMAC algorithm does not prevent from replay-attacks (sending the same message once again) messages have to be numbered. To achieve this, at this point we initialize the message sequence number on both sides to 0 and increases it each time the message is sent.

Any further communication is performed using extended message header defined in Listing 2. The first field is the original ADB header. The second one is the SHA1 hash calculated from the: (i) message sequence number, (ii) secret DH key, (iii) amsg, (iv) payload.

To avoid confusion OPEN, READY, WRITE, CLOSE commands have been replaced with their secure equivalents which use *struct sammesage* as a header. Also a new SCONNECT command has been added as a reply sent from the device to the host when it accepts host signature. This command contains exactly the same information as the standard CONNECT command but it uses the extended header as defined in Listing 2.

Further communication is done in the same way as in the original ADB protocol apart from the HMAC calculation for each message and increasing the sequence number.

7.2. *Evaluation*

7.2.1. *Security*

The modification of the ADB protocol described in the previous subsection ensures message integrity. By incorporating one side authentication phase known from the SSH protocol (RSA signature of one public DH key) it prevents also MitM attacks. Just like the original ADB protocol it ensures that the host is trusted by presenting MD5 of the host public key. In contrast to the unmodified protocol it also ensures that the host actually owns the private key corresponding to the public key sent in the RSAPUBLICKEY. Unfortunately the modified protocol does not ensure device side authentication (nor the original protocol does). Therefore, the user is responsible for verification that he or she connected to the desired device.

The evaluation of the modified protocol proved that it is resistant to all attacks described in this paper apart from the host key replacement attack. The main vulnerability exploited by this attack is the lack of public key fingerprint verification by most users. Thus this depends mostly on a human factor which cannot be completely eliminated. Additionally, it must be also noted that, in general, we can distinguish three groups of users. The first group is fully aware of security risk and has enough competence and determination to verify the fingerprint even in the current design. The second group consists of users who are aware of security risk but do not have enough knowledge or skills to verify the correctness of the fingerprint when this procedure is user unfriendly. Finally, the last group consists of users who are unaware of security risk and just always accepts the fingerprint. In order to mitigate the risk and help users from the second group become more secure a suitable command should be added to the adb command line tool. Future improvement may incorporate some user friendly verification methods like QR code presentation and automatic verification using camera from the mobile device. This could increase the overall level of security even for the users who are unaware but are forced to perform fingerprint validation through the user interface.

The proposed ADB protocol modification not only ensures data integrity but also solves the problem of the automatic private RSA key usage. Thanks to the D–H key exchange and its utilization in HMAC the attacker cannot effectively reuse once generated response. It is also worth mentioning that in our evaluation we assume that both the device and the host side are trustworthy and that there is no kernel-level malicious code involved as this kind of malware would be able to read or modify ADB process memory and still perform MitM attacks in the kernel driver. One of possible mitigation for that could be implementing all the security-related parts of ADB in the Trusted Execution Environment like ARM TrustZone.

7.2.2. *Overhead*

Typically providing improved security features comes at a cost. The proposed modifications not only increase the length of the ADB message header but also add additional operation (HMAC computation) for both sides of the communication.

To estimate the imposed overhead we decided to focus on the file copy operations (push and pull). To this aim we performed experiments and measured the time required to copy files of a different size. We started with a very small file of only two bytes and increased the size by multiplying it by two up to 1 GB. All the measurements have been done using Google Nexus 9 device and Lenovo T470 running Ubuntu 16.04 with 4.15 kernel. The obtained experimental results are presented in Figs. 16 and 17 (in the logarithmic scale). As it can be observed with an increase in the file size the overhead related to the introduced modifications to the ADB protocol increases linearly (red line) and the trend is similar as for the original ADB realization (blue line). Additionally, the increase in delay is stable and it does not depend on the file size.

Moreover, in order to clearly illustrate the "cost" of the improved security features we decided to present the resulting overhead by calculating it as a percentage of the time

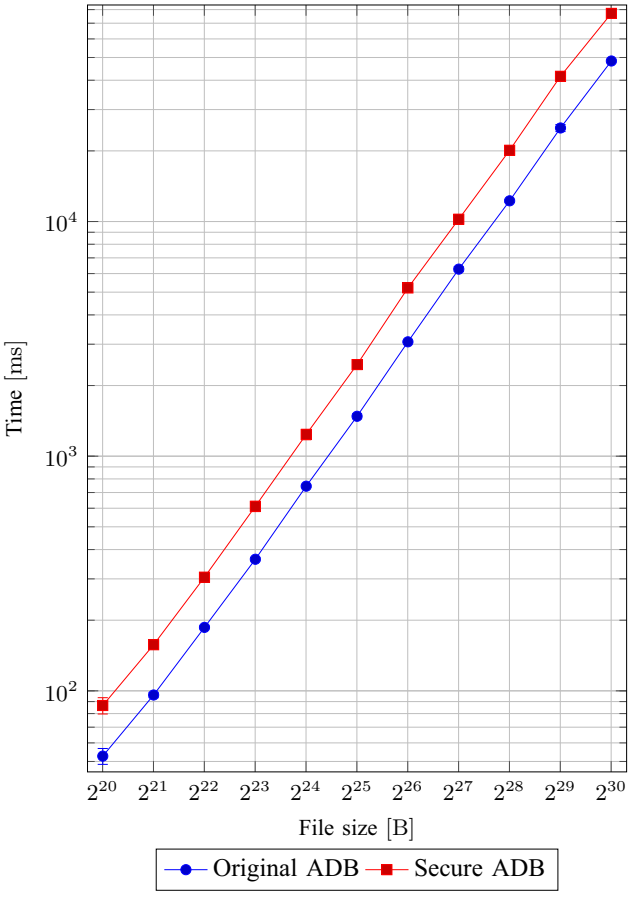

Fig. 16 – The average time required to push a file of a certain size (reduced to files larger than 1 MB).

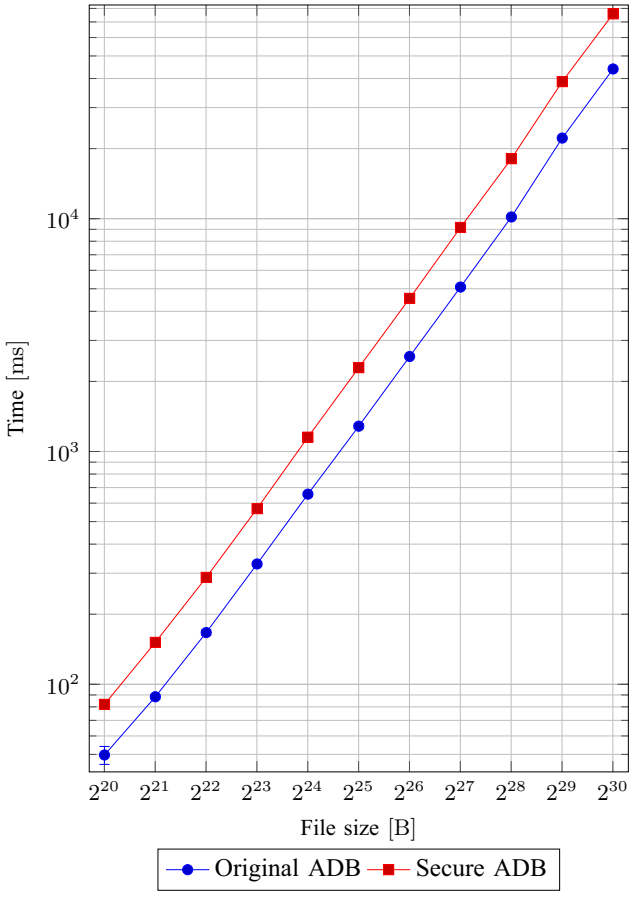

Fig. 17 – The average time required to pull a file of a certain size (reduced to files larger than 1 MB).

difference between the modified and the original ADB implementations. The obtained results are presented in Fig. 18. It is worth noting that we decided to limit the result only to the file sizes larger than 1MB because the time differences for the smaller files are highly variable.

The obtained results are in line with our intuition. We expected that our modification should have a linear performance penalty due to the hash calculation and this is confirmed by our measurements. The measured average overhead is around 65% for the push and around 75% for the pull operation. As percentage results may be misleading it is also important to consider the absolute values as presented in Fig. 19. This plot clearly confirms that the introduced overhead (measured in ms) is practically equal for both: push and pull operations. This almost 10% difference is related to the difference of disk write performance on the host and on the device sides. The host in our case is equipped with the SSD drive which is faster than Nexus 9 flash memory. This means that the pull operation is generally faster than push for both original and modified ADB versions. So if we added the same overhead (in ms) to both operations it has more significant impact (when expressed in percents) for the faster one.

8. Conclusion and future work

In this paper we evaluated ADB protocol security which is utilized in the recent Android versions. In order to achieve this we first classified USB-related attacks to show gaps of knowledge in the USB device related security research. We also presented a novel approach to compromise Android devices by exploiting the ADB protocol in a Man in the Middle attacks.

The conducted research allowed to discover five novel security flaws in the Android OS. Those vulnerabilities can be used not only to bypass the lock screen security and to get unauthorized access to the user's private data, but also to enable future ADB attacks by putting a backdoor (trusted RSA key) to bypass phone security at any time. Our research proved that ADB depends on a protocol state shared between the host and the device but does not utilize any integrity mechanisms which makes it vulnerable to attacks which modifies that state. All discovered security flaws have been reported to both Samsung and Google.

Moreover, we also managed to create a tool which exploits all the found bugs and can be used for security assessment of the current ADB implementations. Our in-depth research resulted in developing ADB protocol extension which protects against all discovered security threats which are not related to the human factor. Even though no malicious software exploit-

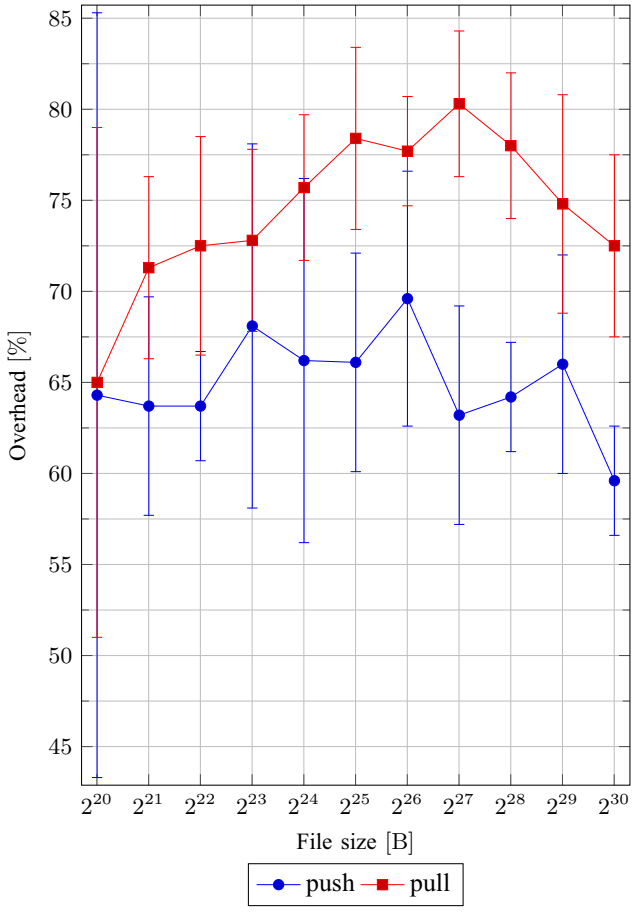

Fig. 18 – The overhead of the push and pull operations in percentages.

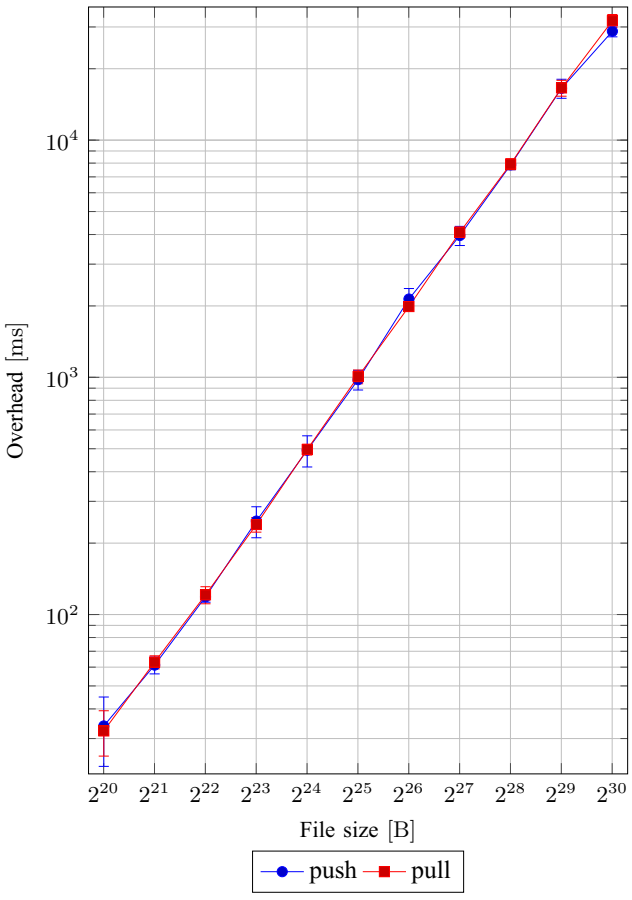

Fig. 19 – The overhead of the push and pull operations in milliseconds.

ing identified vulnerabilities has been found in the wild yet, authors are convinced that presented techniques may be used in the targeted attacks performed e.g. by the state-sponsored malware.

The main goal of this paper is to raise security awareness of the users, researches, and security professionals associated with the USB-related attacks and the threat they pose not only for the PCs but also for the USB devices. Simplicity of the utilized attack techniques shows that the myth that there is no need to implement security mechanisms for the USB communication is invalid. This paper also aims at raising awareness of the USB developers to emphasize that the USB protocols, especially those used to provide such a critical functionality like ADB, should be verified in terms of security just like network protocols are.

Our current research had been mostly focused on Android as it is the most popular mobile platform. It is worth noting that there are also other platforms especially iOS and Windows Phone which also use USB connectivity in the USB device mode for the development purposes. Thus the future work should definitely include security assessment of the USB communication protocol on these platforms.

Our software stack – USBProxy particularly – seems to be also very inefficient for the high bandwidth use-cases. Current implementation utilizes synchronous API of both libusb and GadgetFS. According to Alan's research (Ott, 2014) there may be a significant bandwidth difference between synchronous and asynchronous API if the small transfer sizes are used. As USBProxy typically utilizes 512 bytes as a bulk endpoint packet size switching it to the asynchronous API usage can even double the bandwidth and in addition eliminate most of the I/O threads.

The proposed experimental test-bed has also a notable limitation i.e. only one USB device can be connected to its USB subtree. This makes it impossible to put it before a hub or inside hub case. *Daisho* project seems to solve this issue as it operates on a lower layer. Therefore, future work should include more investigations and the potential development of such a board to overcome above mentioned restrictions.

REFERENCES

usb. Universal serial bus revision 2.0 specification. 2000.
cal. US school expels pupils for using hardware keyloggers to change grades. 2004. [Online]. Available: http://www.techworld.com/news/security/us-school-expelspupils-for-using-hardware-keyloggers-change-grades-3500558/.
usb. ANT CATALOG: COTTONMOUTH-I. 2008.
cpl. CPLINK and Stuxnet – there is a silver lining. 2010.
man. Hardware keyloggers discovered at public libraries. 2011.
spe. Media transfer protocol specification. 2011.

dai. Daisho project. 2013.
uma. umap: the USB host security assessment tool. 2013.
USB. USBProxy project. 2014.
tur. NSA PLAYSET: TURNIPSCHOOL. 2015.
adb. Adb protocol documentation. 2016.
met. Metasploit: Android ADB debug server remote payload execution. 2016.
ran. Ransomware definition from trend micro. 2016.
dir. "Root" via dirtyc0w privilege escalation exploit (automation script)/android (32 bit). 2016.
app. There are 12 million mobile developers worldwide, and nearly half develop for android first. 2016.
Android. Android security updates and resources. 2017.
Beaglebone. Beaglebone black. 2017.
Cve-2017-5554. 2017.
Facedancer21. (USB emulator/USB fuzzer). 2017.
kal. Kali linux nethunter for nexus and oneplus. 2017.
libusb. A cross-platform user library to access USB devices. 2017.
number. Number of apps available in leading app stores as of march 2017. 2017.
Samsung. Samsung mobile security blog. 2017. Link used to report vulnerabilities: http://security.samsungmobile.com/smrreport.html.
Barral D, Dewey D. "Plug and root," the USB key to the kingdom. Proceedings of the 2005 black hat. Las Vegas, NV, USA, 2005.
Cabaj K, Caviglione L, Mazurczyk W, Wendzel S, Woodward A, Zander S. The new threats of information hiding: the road ahead. IT Profr 2018;20(3):31–9.
Camredon B. Usbiquitous: USB intrusion toolkit. Proceedings of the 2016 symposium sur la sécurité des technologies de l'information et des communications. Rennes, 2016.
Dominic Spill MO, Boone J. NSA playset: USB tools. Proceedings of the eleventh ShmooCon information security conference, 2015.
Dominic Spill MO, Kershaw M. What's on the wire? Physical layer tapping with project daisho. Proceedings of the 2013 black hat. Las Vegas, NV, USA, 2013.
Elenkov N. Secure USB debugging in android 4.2.2. 2013.
Guri M. Beatcoin: laking private keys from air-gapped cryptocurrency wallets. CoRR 2018. abs/1804.08714
Guri M, Monitz M, Elovici Y. USBee: air-gap covert-channel via electromagnetic emission from USB. In: Proceedings of the fourteenth annual conference on privacy, security and trust (PST); 2016. p. 264–8.
Guri M, Poliak Y, Shapira B, Elovici Y. JoKER: trusted detection of kernel rootkits in android devices via JTAG interface, 1; 2015. p. 65–73.
Hoggard H. Android 4.4.2 secure USB debugging bypass. 2014.
Hypponen M. The conficker mystery. Proceedings of the 2009 black hat. Las Vegas, NV, USA, 2009.
Kamkar S. USBdriveby. 2014.
Kamkar S. Poisontap. 2016.
Karsten Nohl SK, Lell J. BadUSB – on accessories that turn evil. Proceedings of the 2014 black hat. Las Vegas, NV, USA, 2014.
Kierznowski D. In: Technical report. BadUSB 2.0: USB man in the middle attacks. Royal Holloway University of London; 2016.
Kopecek D. USBGuard: take control over your USB devices. Proceedings of the free and open source software developers' European meeting. Brussels, Belgium, 2016.
Krawczyk H, Canetti R, Bellare M. HMAC: keyed-hashing for message authentication1997.
Larimer J. Beyond autorun: exploiting vulnerabilities with removable storage. Proceedings of the 2011 black hat. Las Vegas, NV, USA, 2011.
Maskiewicz J, Ellis B, Mouradian J, Shacham H. Mouse trap: exploiting firmware updates in USB peripherals. Proceedings of the eighth USENIX workshop on offensive technologies (WOOT 14). San Diego, CA: USENIX Association, 2014.
Mengs M. P4wnp1. 2017.
Nicolas Falliere LOM, Chien E. W32.stuxnet dossier. 2011.
Ott A. USB and the real world. Proceedings of the 2014 embedded Linux conference. San Jose, CA, USA, 2014.
Response SS. Shellshock: all you need to know about the bash bug vulnerability. 2014.
Spill D. USBProxy: an open and affordable USB man in the middle device. Proceedings of the tenth ShmooCon information security conference, 2014.
Sui L. Strategy analytics: android captures record 88 percent share of global smartphone shipments in Q3 2016. 2016.
Tian J, Scaife N, Kumar D, Bailey M, Bates A, Butler K. SoK: "plug & pray" today – understanding USB insecurity in versions 1 through C. In: Proceedings of the 2018 IEEE symposium on security and privacy (SP); 2018. p. 1032–47.
Tischer M, Durumeric Z, Foster S, Duan S, Mori A, Bursztein E, Bailey M. Users really do plug in USB drives they find. In: Proceedings of the 2016 IEEE symposium on security and privacy (SP); 2016. p. 306–19.
Todd-Simpson J. Malduino: the open source badUSB, built on the arduino platform.2016.
van Tonder R, Engelbrecht H. Lowering the USB fuzzing barrier by transparent two-way emulation. Proceedings of the eighth USENIX workshop on offensive technologies (WOOT 14). San Diego, CA: USENIX Association, 2014.
Vidas T, Votipka D, Christin N. All your droid are belong to US: A survey of current android attacks. Proceedings of the fifth USENIX conference on offensive technologies, WOOT'11. Berkeley, CA, USA: USENIX Association, 2011. 10–10
Wang Z, Stavrou A. Exploiting smart-phone USB connectivity for fun and profit. Proceedings of the twenty-sixth annual computer security applications conference, ACSAC '10. New York, NY, USA: ACM, 2010.
Xu M, Sun W, Alam M. Security enhancement of secure USB debugging in android system. In: Proceedings of the twelfth annual IEEE consumer communications and networking conference (CCNC); 2015. p. 134–9.

Krzysztof Opasiak is a Ph.D. student at the Institute of Telecommunications at Warsaw University of Technology, Poland. His research interests includes mobile devices security and external interfaces security. He also works as Senior Software Engineer in Samsung R&D Institute Poland, where he is dedicated to work on open source software.

Wojciech Mazurczyk is an associate professor at the Institute of Telecommunications at Warsaw University of Technology (WUT). His research interests include network security, information hiding, and network forensics. Mazurczyk received Ph.D. and D.Sc. degrees in telecommunications from WUT. He is also an Associate Editor of the IEEE Transactions on Information Forensics and Security and Mobile Communications and Networks Series Editor for the IEEE Communications Magazine.