Detection and Response to Data Exfiltration from Internet of Things Android Devices

Mariem Graa, Ivan Marco Lobe Kome, Nora Cuppens-Boulahia, Frédéric Cuppens, Vincent Frey

Detection and response to Data Exfiltration from Internet of Things Android Devices

Mariem Graa[1], Ivan Marco Lobe Kome[1,2], Nora Cuppens-Boulahia[1], Frédéric Cuppens[1] and Vincent Frey[2]

[1] IMT Atlantique, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
{mariem.graa,ivan.lobekome,nora.cuppens,frederic.cuppens}@imt-atlantique.fr
[2] Orange Labs, 4 rue du Clos Courtel, 35510, Cesson-Svign, France
{ivan.lobekome,vincent.frey}@orange.com

Abstract. Hackers can exfiltrate sensitive data stored in an IoT device such as Android smartphones. He/She abuses the Android pairing mode and targets a personal computer system previously trusted by the device user. The existing protocols that allow file transfer from Android IoT devices to the computer cannot detect this attack. In this paper, we propose an approach to detect attacks exploiting trusted relationship between a third party system such as personal computer and an Android device to exfiltrate user data from the victim device to an attacker. We implement a protocol to secure communication between IoT Android device and third party system. Our approach has been proved to be effective in detecting these category of attacks with reasonable performance overheads.

1 Introduction

The Internet of things (IoT) is the network of physical devices, sensors, actuators and smart devices that are connected through the Internet to exchange data [13]. The number and variety of devices that are used to collect data have increased at an accelerated rate in recent years. Experts estimate that the IoT will consist of about 30 billion objects by 2020 [24]. It is also estimated that the global market value of IoT will reach 7.1 trillion by 2020 [16]. With the increasing number of IoT devices, the user privacy threat is growing. Hackers aim to exfiltrate personal data stored in the IoT devices such as smartphones through USB port. Do *et al.* [5] propose an adversary model for Android covert data exfiltration, and demonstrate how it can be used to construct a mobile data exfiltration technique to covertly exfiltrate data from Android devices. Dorazio *et al.* [6] investigate how an attacker could exfiltrate data from a paired iOS device by abusing a library and a command line tool distributed with iTunes. Existing security tools [7], [8], [9], [10], [1] in Android systems focus on detection of the sensitive data leakage. While such tools may be effective in protecting against malicious third party applications installed in the Android systems, they are less suitable when the data exfiltration is performed by application installed in the personal computer requesting connection to the Android IoT device. In this

paper, we propose an effective approach that allows detection and response to attacks exploiting trusted relationship between a third party system such as a personal computer and an Android IoT device to exfiltrate user data from the victim device. We implement a protocol that ensures security of IoT Android device and personal computer communication. The rest of this paper is organized as follows: Section 2 discusses the existing protocols that allow files transfer from Android IoT devices to the computer. Section 3 describes the threat model. Section 4 presents the proposed approach. Section 5 provides implementation details. We give a security and performance evaluation of our approach in section 6. We present related work about data exfiltration attacks and countermeasures in section 7. Finally, section 8 concludes with an outline of future work.

2 Background

Older Android devices support USB mass storage for transferring files back and forth with a computer. Modern Android devices use the media transfer protocol (MTP) or picture transfer protocol (PTP). The Open Authorization framework (OAuth) [4] is a new widely implemented protocol for the delegation of authorization. The proof of this authorization relies on a token defined as JSON Web Token [23] standard.

2.1 USB Mass Storage

The USB mass storage device class is a set of computing communications protocols that makes a USB device accessible to a host computing device. In addition, it enables file transfers between the host and the USB device. The USB device acts for a host as an external hard drive; the protocol set interfaces with a number of storage devices. The USB mass storage was the way older versions of Android exposed their storage to a computer. Using USB mass storage, users and applications running in the computer will be able to access to all files (system files, media and picture files...) in the Android devices.

2.2 Picture transfer protocol

Picture Transfer Protocol is a protocol developed by the International Imaging Industry Association. It allows the manipulation and the transfer of photographic images from Android devices to computers without the need of additional device drivers. When Android uses this protocol, it appears to the computer as a digital camera. The protocol has a strong standard basis, in ISO 15740. It is standardized for USB as the still image capture device class.

2.3 Media transfer protocol

The Media Transfer Protocol is an extension to the Picture Transfer Protocol (PTP). Whereas PTP was designed for transferring photos from digital cameras,

Media Transfer Protocol allows the transfer of music files on digital audio players and media files on portable media players from devices. MTP is standardised as a full-fledged Universal Serial Bus (USB) device class in May 2008. A main reason for using MTP rather than the USB mass-storage device class (MSC) is that the latter is designed to give a host computer undifferentiated access to bulk mass storage, rather than to a file system, which might be safely shared with the target device. Therefore, a USB host computer has full control over the connected storage device. When the computer mounts an MSC partition, it may corrupt the file system and makes it unsupported by the USB device. MTP and PTP specifically make the unit of managed storage a local file rather than an entire unit of mass storage at the block level to overcome this issue.

2.4 Open Authorization Framework 2

The Open Authorization framework (OAuth) [4] is built on the top of the HTTP protocol. It enables a third-party application to obtain limited access to a service on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the service. It is implemented by the most used identity providers (Facebook, Twitter, Google). OAuth defines:

- A **resource owner.**
- A **client** as the application requesting authorization from the resource owner.
- An **authorization server** which delivers authorization grants representing the resource owner's authorization.
- A **resource server** hosting owner's resources.
- A **token** as a string designed for the resource owner hosting the authorizations granted by the authorization server.

One of the challenges we are addressing in this paper is to allow an application requesting connection to the Android IoT device installed in the personal computer to show the proof of its honesty. This can be done by presenting to the resource owner a token granted by a trusted third party, also known as *Identity Provider*.

In our proposed protocol, we are using a OAuth2.0 like architecture to authenticate and authorize applications and prove that an application is trustworthy.

2.5 JSON Web Token

JSON Web Token (JWT) is an open standard [23] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This token can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA. It is compact and self-contained allowing to transmit rich information on a lightweight payload. The JWT consists of three parts separated by dots, which are:

– **Header** that gives the type of the information and the algorithm being used.
– **Payload** which contains statements about an entity called claims. There are *Registered*, *Public* and *Private* claims.
– **Signature** is the result of the encoded header and the encoded payload signed using a secret and the algorithm specified in the header.

A JWT is then a Base64URL encoded header, a Base64URL encoded payload and the signature putting all together and separated by dots. In our proposed protocol, we use a JWT to guarantee the integrity of the rights granted by the identity provider to one application installed in the personal computer.

3 Target Threat Model

The picture and media protocols limit the access to Android system files. But, all the existing transfer protocols in Android devices (USB Mass Storage, picture transfer protocol and media transfer protocol) allow an application running in the computer and requesting connection to the Android device to transfer the Android files. Let us consider the data exfiltration model based on a client-server TCP/IP architecture as presented in the figure 1. In this model, the attacker can exfiltrate data from the Android devices connected to personal computers over USB. The client application is installed on victim computers to interact with connected Android devices. It is a malicious application that can be installed from the Internet such as a virus. Let us assume that the anti-virus cannot detect this application. The server application resides on a remote computer controlled by the attacker. A socket for each client requesting connection is created by the server application to facilitate data exfiltration. The client application monitors events occurring on the target computer, such as the connection and disconnection of Android devices to and from USB ports. When the Android device is connected, the client application starts requesting for device media and picture files. Then, it sends files to the server application, which is actively listening for incoming connections. Therefore, the data exfiltration attack is launched and the server application gets data stored in the Android devices.

Fig. 1. Target Threat Model

4 The proposed protocol

The proposed approach allows a secure communication between the Android IOT devices and applications running in the computer that request a connection to the Android device, to get access to user's files. Thus, we define a protocol that controls access to files stored in the Android device. It is designed on top of a USB transfer protocol (MTP and PTP) and a web authorization protocol. It involves three entities: the Android IoT devices, the Computer and the Authentication Server (AS) (see Figure 2). When the application installed in the computer requests a connection to the Android IOT devices, our proposed protocol obliges it to send its *id* and an authorization token to be able to access to media or picture data stored in the device. The token was delivered to the application by the AS after have been authenticated in the registration phase. The id is used for the application authentification and the token is used for the application authorization. The Android device sends this information (*id* and authorization token) to the Authentication Server. The AS responds favorably to the Android device when the application is registered and the authorization token is valid. In this case, the application can access to Android media or picture files. Data exfiltration attack is identified when the application fail to authenticate or when the access token has been modified. In these cases, our protocol blocks application access to Android files and the user is notified that he/she is under data exfiltration attack. In this case, our protocol blocks application access to Android files and we notify the user that he/she is probably under a data exfiltration attack. So, our protocol allows detecting malicious application that exploit trusted relationship between the personal computer and an Android device to get sensitive data.

5 Protocol design and deployment

The following sections explain how to design the proposed protocol that involves the Android IoT devices, the Computer and the Authentication Server entities to detect and react to data exfiltration attacks.

5.1 Android IoT devices

We modify the Android OS code to implement our protocol. We instrument the Java class UsbSettings in the package "com.android.settings.deviceinfo". When the device is connected, Android blocks the access to media and picture files until the user chooses the type of transfer protocol. After having selected the type of transfer protocol (MTP or PTP), we verify the application installed in the personal computer identity by asking the Authentication Server. If the installed in the personal computer is authenticated and has a valid token, we authorize the access to the device data exclusively to the given application. Since we are preventing the mounting process of the file system, awaiting for the application authentication, we create a **RAM disk** with the following characteristics:

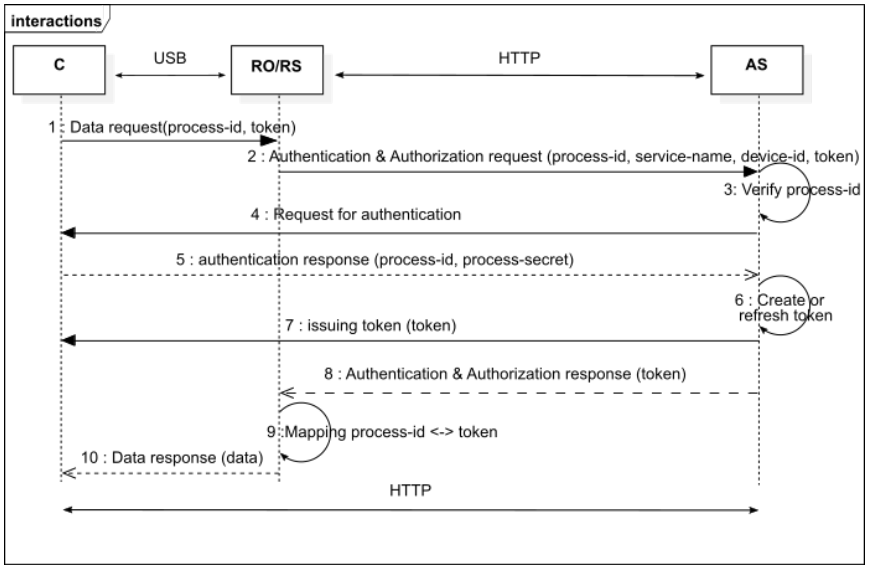


Fig. 2. The proposed protocol

- A **5 MB** of memory, which is big enough to contain a file with an *id* and a *token*.
- A **temporary file storage facility (ramfs)** type of memory. We can also use **tmpfs**, either way this virtual disk has a very short lifetime.
- And a **mount point**. Here the disk is mounted on */mnt/ramdisk*

We consider the application malicious only when the token has been modified. When detected, the user is notified that he/she is probably under a data exfiltration attack. When the application is not known by the authentication server, the response message invites this latter to register itself. In both cases, we block access to media and picture files. The user must update his Android system to integrate our protocol.

5.2 Computer

The device delegates the authentication and authorization to the Authentication Server. The application on the computer will not have access to data unless the AS responds favorably to the Android device authentication and authorization request. To be able to perform the authentication, the application should be registered to the AS and then communicate its *id* to the Android system device. When initiating data request on the device, in order to receive a fresh token, the application will have to send a token *null* with its *id*. The implementation of this step is possible with the MTP library (libmtp) and the PTP library (libptp). It consists of writing on the unique file shown by the device, the application id and token. Before generating a new token, the application will have to authenticate

Table 1. The authentication server responses

	id	*token*	application state	AS responses
1	unregistered	all cases	all cases	error : unregistered
2	registered	null	authenticated	a new token is delivered
3	registered	null	not authenticated	data exfiltration attack
4	registered	is modified	authenticated or not	data exfiltration attack
5	registered	not modified and valid	authenticated or not	access allowed
6	registered	not modified and not valid	authenticated or not	proceed to authentication

itself. The case with a null token corresponds to the first time a registered application is requesting for a device data. If the authorization token is outdated but not modified, a new application will be forged and delivered also after having been authenticated.

5.3 Authentication Server

The authentication server is the trusted third party communicating via HTTPS and in charge of delivering and verifying tokens. In a registration phase, the application is granted an *id* and a *secret*. The editor of the application will have to register to the AS with information of his corporate like: the official website, an email address. The exhaustive list of information required for the registration depends on each AS. In our implementation, the editor is registered with an email address. The token is forged on the demand of the android device when the user authorizes access to its data. The application communicates its *id* to the android device when initiating the data request. In fact, the server needs to make a link between an application, an android device, the type of data requested (MTP or PTP) and a period of validity. This information is represented in a way that it only makes sense for the AS before being issued to the authenticated application. We use JWT [23] to forge the token because this formalism guarantees that the information represented by the token cannot be modified. We use PyJWT [21] to implement the JWT token. There are 6 types of AS responses depending on the state of the application and the two sent parameters: *id* and *token*. Those responses addressed to the Android device are depicted in table 1. If there is no *id* communicated, then we consider that the application is not supporting the protocol. It is therefore, redirected to the registration page. We consider the state *not authenticated* only when the application failed to authenticate itself.

6 Evaluation and Results

In this section, we evaluate the security and the performance of our proposed solution.

6.1 Security evaluation

In this chapter we are evaluating the security of the protocol. To include all scenarios, we model the protocol considering that the token is refreshed on each run. That includes the case when the token is null. As the main goal of this protocol is to secure USB data transmission, we model HTTP communications as secure channels. Those channels are out of the scope of the attacker because they are meant to be secured with the use of protocols like TLS.

Model We state the security goals of the protocol from each of the three entities point of view (application, device, server) and in terms of messages sent and received over the protocol. We model the protocol as depicted in Fig. 3 and verify that the security properties are fulfilled using **ProVerif** tool.

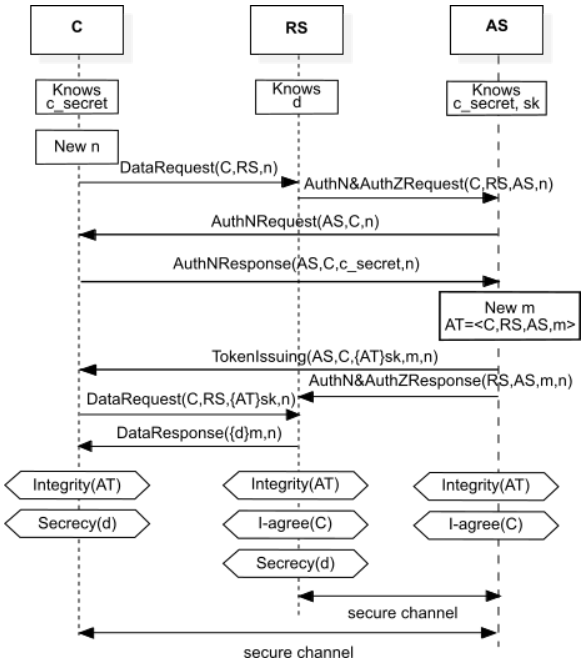


Fig. 3. Security model of the protocol.

ProVerif. ProVerif [19] [2] is an automatic cryptographic protocol verifier, based on the formal model (so called Dolev-Yao model [3]). This protocol verifier is based on a representation of the protocol by Horn clauses. Its main features are:

– It can handle many different cryptographic primitives, including shared- and public-key cryptography (encryption and signatures), hash functions, and Diffie-Hellman key agreements, specified both as rewrite rules or as equations.
– It can handle an unbounded number of sessions of the protocol (even in parallel) and an unbounded message space. This result has been obtained thanks to some well-chosen approximations. This means that the verifier can give false attacks, but if it claims that the protocol satisfies some property, then the property is actually satisfied. The considered resolution algorithm terminates on a large class of protocols (the so-called "tagged" protocols). When the tool cannot prove a property, it tries to reconstruct an attack, that is, an execution trace of the protocol that falsifies the desired property.

Security goals. We assume that the attacker cannot break the cryptographic construction used to make the secure channel. The protocol guarantees the following security properties:

– **Secrecy:** If an application data message m is sent over the channel c, between an honest client C and an honest server S, then this message is kept confidential from an attacker.
– **Integrity:** If an application data message m is sent over the channel c, between a honest client C and an honest server S, then this message can be seen but cannot be modified by an attacker. The security property holds even if the message m was given to an attacker.
– **Authentication** via
  • **injective agreement:** This security property holds if each event is executed in the order defined by the protocol and for all n, each event from run n is different from events from run $n + 1$.
  • **integrity of m:** The authentication property is satisfied if the injective agreement holds and if the message "m" has not been modified.

Those security properties prevent from replay and man-in-the-middle attacks. In listing1.1, we can see a part of the ProVerif model of our protocol. The public and private channels are respectively c and cs. We have 3 roles: application client, Android resource server and Authentication Server. We then declare 3 processes: *ProcessC*, *ProcessRS* and *ProcessAS*. Only *ProcessC* is depicted on that listing. Security properties are declared according to the attacker knowledge and events. The types *bitstring* is a predefined one in ProVerif unlike *skey* which must be declared.

Listing 1.1. Sample of the protocol modeled with ProVerif

```
(* Declaring honest host names C and RS and AS *)
free C, RS, AS: host.
(* Declaring channels *)
free c: channel.
free cs: channel [private]. (* secure channel *)
(* Declaring private names. *)
```

```
free c_secret, d:bitstring [private].
free sk:skey [private].
(* Declaring functions *)
fun enc(bitstring, skey):bitstring (*symmetric encryption*)
fun dec(bitstring, skey):bitstring (*symmetric decryption*)
equation forall x:bitstring, y:skey; dec(enc(x,y),y) = x (*equational
    theory for symmetric key encryption*)
...
(* Declaring events*)
event startC(bitstring,host,host,bitstring);
event endAS(bitstring, bitstring)
(* Declaring security properties*)
not attacker(bitstring d). (* Secrecy assumptions *)
query a:bitstring, b:host, c:host, d:bitstring; inj-event(endAS(a,d))
    ==> inj-event(startC(a,b,c,d)). (* Injective-agreement assumptions
    *)
...
(* Queries : verify security properties*)
query attacker(d); (*Verify the secrecy of 'd'*)
...
(*Application client role*)
let ProcessC(c_secret)=
new nc:bitstring;
out(c, (nc, C, RS, AS));
in(cs, (x:bitstring, y:host, z:host));
out(cs, (nc, AS, C, c_secret));
in(cs, (x1:bitstring, y1:bitstring, z1:bitstring));
out(c, (n, C, RS, z1)
in(c, x2:bitstring)
```

6.2 Results

We tested the cases presented in the table 1. In the first case, the application (process running in the computer) does not give an *id*. Our protocol blocks the access to Android files (see figure 4 (b)). In the cases 2 and 5, the application can transfer Android media and picture files (see figure 4 (a)). The application is registered and authenticated in the case 2 and it is registered and the token is valid and not modified in the case 5(see Listing 1.2).

Listing 1.2. Case 5: id registered and valid token

```
10.0.2.2 - - [02/Feb/2018 13:36:17] "POST /ptp/123/123456789 HTTP/1.1"
200 -
> The process 123456789 wants to get access to 123 in mtp mode
> Verifying process_id : 123456789
process 123456789 registered
```

```
Token decode result : {u'iss': u'123', u'rec': u'123456789',
u'sub': u'mtp', u'exp': u'201802022243'}
This token is still valid. Process 123456789 can have access
```

The cases 3 and 4 present attack scenarios because the application is not authenticated and the token is modified respectively. In these cases, we block access to Android user files (see Listing 1.3 for the case 4).

Listing 1.3. Case 4: id registered and modified token

```
10.0.2.2 - - [02/Feb/2018 14:05:34] "POST /mtp/123/123456789 HTTP/1.1"
200 -
> The process 123456789 wants to get access to 123 in mtp mode
> Verifying process_id : 123456789
process 123456789 registered
This token have been modified ! 123 may be under attack
```

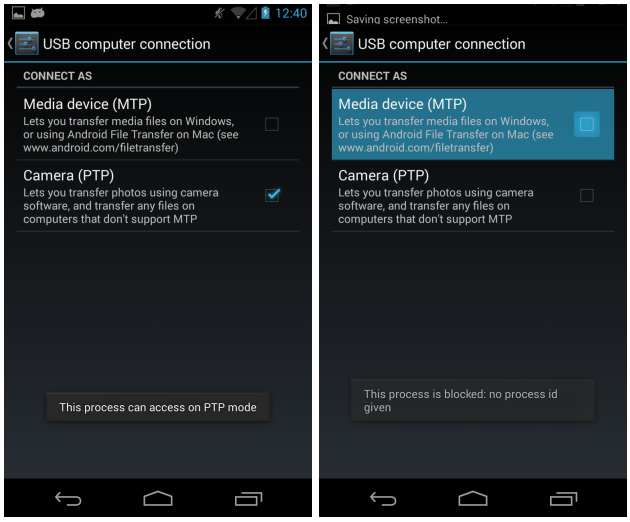
In the case 6, the authentication of the application is required (see Listing 1.4).

Listing 1.4. Case 6: id registered and token is no more valid

```
10.0.2.2 - - [02/Feb/2018 14:05:53] "POST /mtp/123/123456789 HTTP/1.1"
200 -
> The process 123456789 wants to get access to 123 in mtp mode
> Verifying process_id : 123456789
process 123456789 registered
Token decode result : {u'iss': u'123', u'rec': u'123456789', u'sub':
u'mtp', u'exp': u'201802020910'}
This token is no more valid. Starting 123456789 authentication...
```

6.3 Performance evaluation

We evaluated the performance impact of the proposed architecture on the computer and on the Android device. As we can see on the figure6.3, we have queried 50 times the AS in each case depicted in table 1. We have an average of the response time for each type of query. The zeros are not taken into account in the average calculus. They are representing a slow-down in the Android system due to an important use of memory resources by the application in charge of the test. In fact, the test consists of querying the AS every 5s, which requires a lot of the device resources. The experiment shows that this architecture is requiring about 1.5 milliseconds to make a response.



(a) (b)

Fig. 4. Android system device notifications

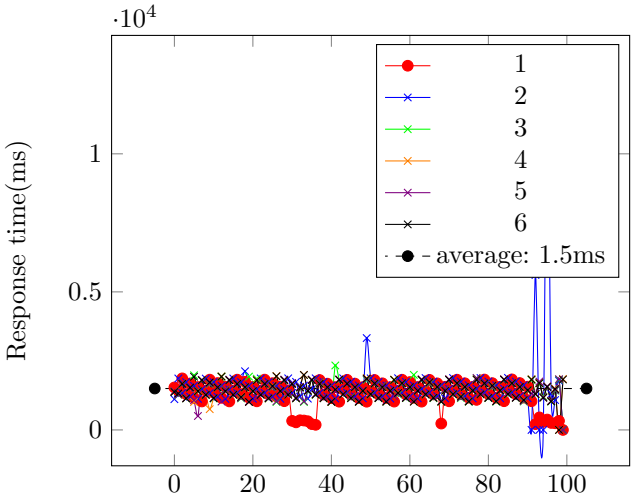


Fig.5 Evaluation of AS response time.Plots 1-6 represent different response types according to table 1.

In addition, we use the CaffeineMark [6] to evaluate the influence of the protocol execution on the system performance. We test the not modified Android overhead when the MTP and PTP protocols are executed. Then, we test our Android modified system overhead when our protocol is executed. We observe that our proposed protocol generates 2.5 percent execution time overhead with respect to the unmodified system. Thus, it does not really affect the performance of the system.

7 Related work

In this section, we present data exfiltration attacks. We also discuss existing countermeasures. Do *et al.* [5] present an adversary model for Android covert data exfiltration using communication mediums (SMS and audio) on mobile devices. Spolaor *et al.* [26] demonstrate how an adversary can exfiltrate data from the smartphone via a USB charging cable using an Android application that encodes sensitive information and transmits it via power bursts back to the public charging station. USBee [14] uses the USB data bus to generate electromagnetic signals, modulate and exfiltrate digital data over these signals. MACTANS [20] is an implementation of a malicious usb charger that injects a Trojan horse app with a payload to compromise an iOS device. All these data exfiltration attacks use malicious application installed in the phone to collect user data. In our approach, the malicious process is running in the personnel computer and exploit the usb connection to obtaint sensitive data stored in the Android phone. Dorasio *et al.* [6] present the same data exfiltration attack from iOS devices and not from Android system. Many works exist in the literature to detect data exfiltration attacks. Sharma *et al.* [25] describe a framework to detect potential exfiltration events caused by infected USB flash drive on a machine. The detection system flags alerts based on temporally-related anomalous behavior detected in multiple monitored modules. GoodUSB [27] enforces permissions of devices by encoding user expectations into USB driver loading. GoodUSB includes a security image component and a honeypot mechanism for observing suspicious USB activities. All solutions cited above assume that maliciousness comes from devices connected to the computer. In our approach, the malicious process is installed in the personnel computer and not in the device. Many dynamic taint analysis approaches defined in smartphones like TaintDroid [7], AppFence [15] and Graa *et al.* [11], [9] allow detection of sensitive data leakage by third party applications running in the Android device using the data tainting. Wang *et al.* [28] enforce security policies on data flows for Android applications to prevent unauthorized usb hardware flows. Hwang *et al.* [18] propose the use of static analyzer to detect a malicious service installed from an infected PC using ADB [22]. However, these solutions cannot detect data exfiltration attack performed by application requesting connection to the Android IoT device installed in the personal computer. They control the behaviours of applications running in the phone and they assume that the relationship between the personal computer and an Android device is trusted.

8 Conclusion

The transfer protocols defined in the Android system such as MTP and PTP can be bypassed by exploiting data exfiltration attacks. We have improved these protocols to protect sensitive user data stored in the Android IoT device. Our approach allows detection and response to data exfiltration attacks. We control identity of applications running in the computer and requesting connection to

the device. The data transfer is allowed only if the application is registered and the authorization token is valid. We block access to sensitive data and we notify the user that he/she is probably under a data exfiltration attack when the application id is not known by the AS or the token is not valid. So, our protocol allows detecting malicious application that exploit trusted relationship between the personal computer and an Android device to get sensitive data. We evaluate the security and the performance of our proposed solution. We prove that our proposed protocol ensures secrecy, integrity and authentication. Those security properties prevent form replay and man-in-the-middle attacks. The experiment shows that the AS is requiring about 1.5 milliseconds to make a response. We observe that our proposed protocol generates 2.5 percent execution time overhead with respect to the unmodified Android system. Thus, it does not really affect the performance of the Android system.

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 49(6), 259–269 (2014)
2. Blanchet, B.: Symbolic verification of tls 1.3 with proverif, https://github.com/inria-prosecco/reftls
3. Cervesato, I.: The Dolev-Yao intruder is the most powerful attacker. In: 16th Annual Symposium on Logic in Computer ScienceLICS. vol. 1. Citeseer (2001)
4. Dick Hardt, E.: The OAuth 2.0 Authorization Framework, https://tools.ietf.org/html/rfc6749
5. Do, Q., Martini, B., Choo, K.K.R.: Exfiltrating data from android devices. Computers & Security 48, 74–91 (2015)
6. DOrazio, C.J., Choo, K.K.R., Yang, L.T.: Data exfiltration from internet of things devices: ios devices as case studies. IEEE Internet of Things Journal 4(2), 524–535 (2017)
7. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 32(2), 5 (2014)
8. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android. Tech. rep. (2009)
9. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Lanet, J.L.: Tracking explicit and control flows in Java and native Android apps code. In: ICISSP 2016 : 2nd International Conference on Information Systems Security and Privacy. vol. Proceedings of the 2nd International Conference on Information Systems Security and Privacy, pp. 307–316 (2016)
10. Graa, M., Boulahia, N.C., Cuppens, F., Cavalliy, A.: Protection against code obfuscation attacks based on control dependencies in android systems. In: Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on. pp. 149–157. IEEE (2014)
11. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Cavalli, A.: Detecting control flow in smarphones: Combining static and dynamic analyses. In: Cyberspace Safety and Security, pp. 33–47. Springer (2012)

12. Grier, J.: Detecting data theft using stochastic forensics. digital investigation 8, S71–S77 (2011)
13. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things (iot): A vision, architectural elements, and future directions. Future generation computer systems 29(7), 1645–1660 (2013)
14. Guri, M., Monitz, M., Elovici, Y.: Usbee: air-gap covert-channel via electromagnetic emission from usb. In: Privacy, Security and Trust (PST), 2016 14th Annual Conference on. pp. 264–268. IEEE (2016)
15. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 639–652. ACM (2011)
16. Hsu, C.L., Lin, J.C.C.: An empirical examination of consumer adoption of internet of things services: Network externalities and concern for information privacy perspectives. Computers in Human Behavior 62, 516–527 (2016)
17. Hu, Y., Frank, C., Walden, J., Crawford, E., Kasturiratna, D.: Profiling file repository access patterns for identifying data exfiltration activities. In: Computational Intelligence in Cyber Security (CICS), 2011 IEEE Symposium on. pp. 122–128. IEEE (2011)
18. Hwang, S., Lee, S., Kim, Y., Ryu, S.: Bittersweet adb: Attacks and defenses. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. pp. 579–584. ACM (2015)
19. Kobeissi, N., Bhargavan, K., Blanchet, B.: Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In: 2nd IEEE European Symposium on Security and Privacy (EuroS&P'17). pp. 435–450. IEEE, Paris, France (Apr 2017)
20. Lau, B., Jang, Y., Song, C., Wang, T., Chung, P.H., Royal, P.: Mactans: Injecting malware into ios devices via malicious chargers. Black Hat USA (2013)
21. Lindsay, J.: Pyjwt, https://github.com/jpadilla/pyjwt
22. Liu, F.: Windows malware attempts to infect android devices (2014)
23. M. Jones, J. Bradley, N.S.: Json web token (jwt), https://tools.ietf.org/html/rfc7519
24. Nordrum, A.: Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. A vailable at https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated (August 2016)
25. Sharma, P., Joshi, A., Finin, T.: Detecting data exfiltration by integrating information across layers. In: Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on. pp. 309–316. IEEE (2013)
26. Spolaor, R., Abudahi, L., Moonsamy, V., Conti, M., Poovendran, R.: No free charge theorem: A covert channel via usb charging cable on mobile devices. In: International Conference on Applied Cryptography and Network Security. pp. 83–102. Springer (2017)
27. Tian, D.J., Bates, A., Butler, K.: Defending against malicious usb firmware with goodusb. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 261–270. ACM (2015)
28. Wang, Z., Johnson, R., Murmuria, R., Stavrou, A.: Exposing security risks for commercial mobile devices. In: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. pp. 3–21. Springer (2012)