Inside Job: Understanding and Mitigating the Threat of External Device Mis-Bonding on Android

Muhammad Naveed[1], Xiaoyong Zhou[2], Soteris Demetriou[1], XiaoFeng Wang[2], Carl A Gunter[1]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
[2]School of Informatics and Computing, Indiana University at Bloomington
{naveed2, sdemetr2, cgunter}@illinois.edu, {zhou, xw7}@indiana.edu

***Abstract*—Today's smartphones can be armed with many types of external devices, such as medical devices and credit card readers, that enrich their functionality and enable them to be used in application domains such as healthcare and retail. This new development comes with new security and privacy challenges. Existing phone-based operating systems, Android in particular, are not ready for protecting authorized use of these external devices: indeed, *any* app on an Android phone that acquires permission to utilize communication channels like Bluetooth and Near Field Communications is automatically given the access to devices communicating with the phone on these channels.**

In this paper, we present the first study on this new security issue, which we call *external Device Mis-Bonding* or DMB, under the context of Bluetooth-enabled Android devices. Our research shows that this problem is both realistic and serious: oftentimes an unauthorized app can download sensitive user data from an Android device and also help the adversary to deploy a spoofed device that injects fake data into the original device's official app on the phone. Specifically, we performed an in-depth analysis on four popular health/medical devices that collect sensitive user information and successfully built end-to-end attacks that stealthily gathered sensitive user data and fed arbitrary information into the user's health/medical account, using nothing but Bluetooth permissions and public information disclosed by the phone. Our further study of 68 relevant device-using apps from Google Play confirms that the vast majority of the devices on the market are vulnerable to this new threat. To defend against it, we developed the first OS-level protection, called *Dabinder*. Our approach automatically generates secure bonding policies between a device and its official app, and enforces them when an app attempts to establish Bluetooth connections with a device and unpair the phone from the device (for resetting the Bluetooth link key). Our evaluation shows that this new technique effectively thwarts the DMB attacks and incurs only a negligible impact on the phone's normal operations.

I. Introduction

With the rapid progress in smartphone technologies and their unprecedented popularity come increasingly diverse and innovative uses of these technologies. Today's smartphones have become a platform not only for calls, entertainment, and navigation but also for such critical activities as personal financial management and healthcare. These new applications often rely on the hardware not already built into the smartphone and therefore need an external device to work together with the phone through Bluetooth, Near-Field Communication (NFC) and other channels. A prominent example is smartphone-enabled healthcare devices such as blood-glucose meters [10], [11] and Electrocardiography (ECG) sensors [1]. Such devices are typically sensors or other data-gathering mechanisms, which leverage the smartphone to preprocess the information collected from a patient before delivering it to more capable systems such as web services. This utilization of the smartphone, a general mobile computing system, helps reduce the cost of those activities, but also brings in new security and privacy challenges, as the phone's security mechanism is not designed for protecting these external devices.

External device protection on Android. More specifically, let us take a closer look at Android's security design. Its permission and sandbox security model mainly aims at protecting the phone's local resources, such as GPS, SD-card, etc. Each of these resources is guarded by one or more permissions, and can only be used by the Android applications (*apps* for short) granted the appropriate permissions by the user. By comparison, no permissions are assigned to an external device. All Android can control here is the channel that links the phone to the device, such as Bluetooth, NFC, Audio port, etc. The problem is that many devices could all share the same channel and many apps could all claim the permission to use that channel for different purposes. As a result, access to an external device often becomes hard to control in the presence of those "insiders" (unauthorized apps with the permission on the device's communication channel).

As an example, consider a blood-glucose meter that measures a patient's blood-sugar concentration and reports the result to an app running on her smartphone through a Bluetooth channel. Only that app should be allowed to communicate with the device. But, this access control cannot be enforced on today's Android, which lets *any* app with the Bluetooth permission talk to the meter. Note that this problem cannot be addressed by the standard Bluetooth security mechanism [44], as it only ensures that the blood-glucose meter is paired with an authorized smartphone, not an authorized app running on that phone.

Our work. In this paper, we report our first-step study on this emerging security challenge, which we call *external Device Mis-Bonding* (DMB), particularly its threat to Bluetooth devices.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23097

Our research shows that this DMB threat is indeed realistic and serious. Specifically, we thoroughly analyzed four high-profile smartphone-enabled healthcare devices, including a Bodymedia Link Armband [3] (monitoring a user's daily activity pattern), an iThermometer [13], a Nonin Onyx II Pulse Oximeter [15] (monitoring a patient's pulse and Oxygen saturation) and an Entra Health System Blood-Glucose Meter [11]. These devices are very popular, and all of them except iThermometer are FDA approved Class II medical devices. However, we found that the lack of a secure bonding between them and their corresponding apps leaves these devices vulnerable to different types of DMB attacks. In particular, in a *data-stealing* attack, an unauthorized app on the authorized phone can stealthily download the data these devices gather from a patient, using nothing but its Bluetooth permission and side-channel information for determining the right moment for the download. This may happen when such a device is present and the official app, that is, the one the user expects to connect to the device, is, in fact, not connected to it. In a *data-injection* attack, the malicious app, again with a Bluetooth permission only, even breaks the device-level Bluetooth security: the app stealthily unpairs the phone from an authorized external device and pairs it with a malicious device to feed falsified medical data into the original device's official app. Demos of the attacks are posted online [6].

As the current design of Android does not provide any means to bond an app to an external device, individual device manufacturers are left with no option but to develop their own approaches to authenticate their official apps to their devices. This could be challenging, given that these devices are often simple sensors and may not have much resources to perform authentication operations, such as running cryptographic functions. To understand how pervasive the problem is, we further conducted a measurement study in which we analyzed 68 official apps responsible for a wide spectrum of Android external devices sampled from Google Play and confirmed that *none of them* are protected by any app-device authentication mechanism.

Given the grave consequences of a DMB attack (compromises of the confidentiality and integrity of critical user data) and the credible threat it poses (the pervasiveness of vulnerable devices), serious effort needs to be made to address this issue. To this end, we present the first OS-level solution, called *Dabinder*. Dabinder works as an Android service, keeping track of the identities of individual external devices and the apps that initiate the first Bluetooth connections with them right after they are paired with the phone. Based on this observation, a security policy is generated to tie the app to the device, which forms a bond that cannot be broken without the user's consent: a different app is not allowed to connect to the device, nor can it unpair the phone from the device, unless this access-control policy is overruled by the user. We implemented Dabinder and evaluated our prototype against the attacks on these popular healthcare devices, which demonstrates its effectiveness in mitigating the threat and negligible impacts on the phone's normal operations. We released the code of Dabinder as an Android 4.2.2_r1 patch, which can be downloaded here [5].

Contributions. We summarize the contributions of the paper as follows:

- *New understanding*. We made the first attempt to understand the threat of Android device mis-bonding, in the context of Bluetooth bonds. Our study reveals the gravity of the threat that could lead to leaks of sensitive user data or compromise of its integrity, as well as the prevalence of the issue among popular smartphone-enabled devices.
- *New techniques*. We developed the first techniques to mitigate the threat. Our approach automatically generates security policies for protecting the bonding between an external device and its authorized app, and effectively enforces the policies without impeding normal operations on the phone.
- *Implementation and evaluation*. We implemented our design and evaluated its effectiveness and performance.

Roadmap. The rest of the paper is organized as follows: Section II introduces the background information of our study, particularly the generality of the device mis-bonding problem; Section III elaborates our security analysis on smartphone-enabled Bluetooth devices; Section IV presents the design, implementation and evaluation of our protection mechanism; Section V discusses the limitations of our study and follow-up research; Section VI compares our work with related prior research and Section VII concludes the paper.

II. External Device Mis-Binding

In this section, we first explicate the threat of Android external device mis-bonding at a high level and then clarify the scope of our research and its underlying adversary model.

A. Background

The fundamental cause of the DMB problem is the inadequacy of the Android security model in protecting external devices. Here we provide background information about these devices and the way the Android security mechanism works.

External devices. Increasingly smartphones are used to enhance healthcare, financial management, and services in other critical domains. For this purpose, they often need to work together with specialized external devices, helping preprocess the data gathered by the devices and relaying the outcome to various service providers. In particular, smartphones today are extensively used with different medical devices for patient care and monitoring, fitness, health, education and research [43]. Examples of related health devices include WiFi or Bluetooth based blood pressure monitor [20], ECG device [1], muscle fatigue monitor [19], pulse oximeter [15], heart rate monitor [21], thermometer [13], lifestyle monitor [3], [8], [9] and many others. These devices utilize smartphones to analyze and display data, and to communicate with their corresponding web services. The data here is often considered sensitive, as it describes an individual's health status and activities. In addition to healthcare data, smartphone-enabled devices are also utilized to work on other private user information. For example, the Square Reader collects a user's credit-card data through a phone's audio port; and NFC devices are widely used in Europe to accept payments through phones.

As a leading mobile operating system (with 75% of market share [2]), Android is the main platform that supports these external devices. Its security design, therefore, becomes crucial for protecting sensitive user data involved in the operations of these devices.

Android security model. At the center of Android security is its application sandboxing and permission model. Each Android app is confined within its own sandbox and needs permissions to access sensitive resources, including camera, audio, location, network, etc. In particular, some of these resources (e.g., audio, network, Bluetooth, NFC and others) can serve as channels to connect a phone to external devices. To acquire the permissions to use the resources, an app needs to explicitly request (using AndroidManifest.xml) from the phone's user during its installation.

This security model is actually built upon Linux's kernel level protection (process separation and file system access control). Each Android app runs as a Linux user with a unique user ID, which naturally separates its operations and data from those related to other users under the Linux user-based protection. Sensitive system resources are usually mapped to special Linux groups such as inet, gps, etc. An app granted with the permission to use a resource is assigned to that resource's Linux group. Every member within that group (with that permission) is equally entitled to operate on that resource.

B. Mis-bonding Threat

The problem. As we can see here, the Android security model only controls the access rights on the channel used for communicating with an external device (such as Bluetooth, NFC, audio port) not the device itself. As long as an app acquires a permission for this channel, it automatically gains access to any device that is connected through the channel. This is because all apps with the same permission are affiliated with the same Linux group, so they have the same privilege on the resources shared within the group, including the channel (e.g., Bluetooth). Android does not care how this channel is used and which party an app talks to. Also, all information related to an external device (such as Bluetooth pairing data) is also shared within the group members, which could be exploited by a malicious member to compromise the integrity of the data received by the official app of the device (Section III-B).

As an example, consider a medical device that communicates with its Android app using Bluetooth. To make this happen, the smartphone hosting the app first needs to *pair* with the device, which forms a *bond* between the phone and the device. This pairing process yields a set of bonding information, which allows these two devices to connect to each other automatically in the future. The bonding information includes the external device's MAC address and its Universal Unique Identifier (UUID), together with a secret link key for authentication and encrypted communication (when the devices decide to do so). Note that such a bond relation is only established on the device level; there is nothing to prevent an unauthorized app (with Bluetooth permissions) on an authorized phone from connecting to the device. This permission also makes the app a member in the `net_bt_admin` group. As a result, the unauthorized app is given the privilege to break the bonding with an authorized medical device and pair the phone with a malicious one configured with the former's bonding information so as to feed fake medical data into a patient's medical record (Section III-B).

Given the limitations of the Android security model, device manufacturers are on their own to address this security risk. One thing they can do is to design a way to secure the communication between the device and its official app. An instance we are aware about is the Square creditcard reader [18], which connects to a smartphone through its audio port. Its early version is vulnerable because every app with audio permission can read from it. The later one comes with an encryption capability: the reader encrypts the data (using AES) collected from a credit card using a hard-coded key and transmits the ciphertext through the phone to the web. Most other devices, however, do not provide any app-device level protection, as confirmed in our measurement study (Section III-C), possibly due to the fact that most of them are simple sensors, without sufficient computing resources to support cryptographic operations. These devices can upload the data to the online service through the smartphone, which also provides an interface for the user to see and analyze their data. Encrypting this data in the device and just using the phone as a communication relay would severely affect the usability of the device, as the user would not be able to use her phone to see her data. All the devices we analyzed have apps that display the user data on the smartphone. Hence, the treatment adopted by Square does not seem to be suitable for these devices.

Scope and adversary model. In our research, we conducted a preliminary study on this under-researched yet critical security problem. As the first step, our study focuses on Bluetooth healthcare devices, which are becoming increasingly popular in recent years. The security risks we discovered and the new technique we built are expected to be extended to other types of external devices, though further studies are certainly needed here to better understand their related security issues.

We assume that a malicious app is present on the victim's Android phone with both the Bluetooth and Bluetooth Adminstration permissions. These two permissions are claimed by almost all the Bluetooth-capable apps. For a data-injection attack, in which a malicious party clones the target device, we also assume that the fake device can be placed close to the victim's phone (within 100 meters).

III. Attacks and Measurement

In this section, we report on our study that aims at better understanding the magnitude of the device mis-bonding threat. To this end, we built end-to-end attacks on popular smartphone-enabled medical devices and further measured the pervasiveness of the security risk discovered, as elaborated below.

Healthcare devices. As mentioned before, our study focuses on Bluetooth devices. Specifically, we analyze four popular healthcare devices. All of them except the iThermometer are FDA approved Class II medical devices [16], in the category of X-ray machines, infusion pumps, etc., which are used to deal with real patient care and life critical information. The first three devices either have their online services available or are capable of synchronizing the information they collect with other cloud based health-services. Here is more detailed information about these devices:

- *Bodymedia Wireless LINK Armband* [3] is one of the most popular activity monitoring systems, which has been used in over 120 clinical studies [23]. It utilizes four different sensors to collect data about the user's motion, temperature, perspiration, etc., for accurate calculation of calories burned and monitoring of sleep patterns. The output of the device can be displayed by a mobile app running on Android or iOS, and further synchronized to an activity manager website. Disclosure of the data can leak out the user's health status and daily activities.
- *Nonin Onyx II 9560 Pulse Oximeter* [15] is one of the best wireless finger pulse oximeters. Along with a smartphone app, it enables clinicians to remotely monitor blood-oxygen saturation levels and pulse rates of the patients with chronic diseases such as Chronic Obstructive Pulmonary Disease (COPD) or asthma [16]. The device uses Bluetooth to connect to the smartphone, which can deliver the data to the health provider, online health services or stored locally for later analysis. The data collected here is also critical for understanding the patient's status and choosing an effective treatment. This device is Microsoft HealthVault[1] certified [16].
- *Entra Health System MyGlucoHealth Blood-Glucose Meter* [11] is one of the most popular glucose monitoring devices. It comes with a complete diabetes management system (including testing at home) uploading data to the online account through its Android app, which helps a patient manage her disease and share this data with her health provider. Glucose levels determine the amount of insulin to be injected into the patient's body, which is private and also life-critical: a wrong amount of injection can have severe implications, including death [31]. Along with FDA, this device is also approved by CE[2] and is fully HL7[3] compliant [7].
- *iThemometer* [13] is an electronic thermometer that works with Android through Bluetooth for personal health or long-distance monitoring of elderly persons or babies. The body temperature is an indicator for life-threatening conditions like infection.

All these devices involve the user's critical data, whose confidentiality and integrity is important to her health and well-being. In the presence of the malicious insider app, however, we show that such data becomes extremely vulnerable to the DMB threat.

A. Data-Stealing Attacks

In our research, we investigate the feasibility of data-stealing attacks on Bluetooth devices, in which a malicious app running on the victim's phone attempts to steal sensitive data collected by the target device. The attack turns out to be more complicated than it appears to be: particularly, depending on the nature of a device, the malicious app needs to capture a small time window during which the device is on and in proximity, under the competition of the official app that also wants to make a connection to the device. Here we describe how we addressed such technical challenges and designed end-to-end attacks on real devices.

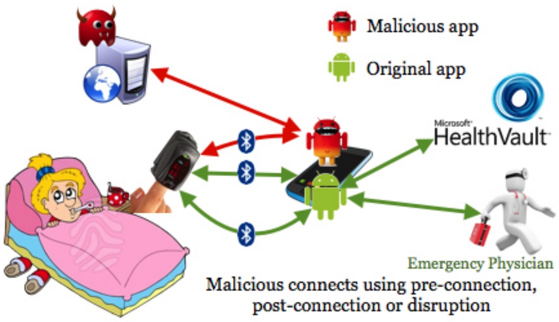
Malicious app
Original app
Microsoft HealthVault
Emergency Physician
Malicious connects using pre-connection, post-connection or disruption

Fig. 1: Data-stealing Attack

Attack strategies. Given the `BLUETOOTH` and `BLUETOOTH _ADMIN` permissions, a malicious app appears to have all it needs to steal data from these healthcare devices, and merely because Android does not mediate which app is supposed to connect to the devices. In practice, however, the situation is much more subtle than it appears to be at a first look: a malicious app must not be oblivious to the fact that the target device could or could not be in proximity and even when they are, for some of them one needs to push a button or take some actions to activate their Bluetooth services. Specifically, the Bodymedia armband is activated a few seconds after it is put on one's arm; the iThemometer has such a button on it; the Nonin pulse oximeter turns on when one inserts her finger into the device and turns off once she takes out her finger; and the MyGlucoHealth meter has a button for activating the Bluetooth and the meter turns off automatically after sending data to the phone. Also complicating the attack is the presence of the official app. Once the official app establishes a socket connection with the target device, the malicious app cannot directly talk to the device before this connection is torn down and vice versa.

A straightforward solution is an opportunistic strategy in which the malicious app either periodically invokes the service discovery protocol to find out whether the target device is in its vicinity or blindly makes repeated connection probes, hoping to get to the device as soon as it shows up. However, neither of these approaches works well in practice due to increased power usage of Bluetooth radio, a power-consuming practice that is usually suggested against [53]. For instance, a user may keep the Bluetooth communication off to save power. Then, when she wants to use it, she runs a Bluetooth-capable app that automatically turns on Bluetooth. A malicious app using this strategy must repeatedly enable Bluetooth to discover the target device; this consumes more battery power than expected and could also be noticed by the user, given the presence of the Bluetooth sign on the phone.

In our research, we adopt a lightweight and stealthy strategy to perform the surveillance. Simply put, the execution of the device's official app is a strong indication that the device is in action and also within the connection range of the target device. Based on this observation, the malicious app can keep checking when any of the target apps launches, an event that can be used

[1] Microsoft HealthVault is a free online service for personal health information management.

[2] CE Mark is medical device approval mechanism in Europe.

[3] HL7 – Health Level Seven International – is a globally interoperable standard for health information exchange.

to trigger an attempt to catch the window of opportunity. Specifically, our app, which works as a service in the background, periodically runs the Android API `getRunningTasks()` to get the app running in the foreground in constant time $O(1)$. This needs an additional permission `GET_TASKS`. Alternatively, we can use `getRunningAppProcesses()`, which does not need any permission, but returns a list of running processes in an unspecified order that the malicious app needs to traverse in search for the target app, which takes $O(n)$ running time, where n is the number of concurrently-running processes on the phone. The same result can be achieved by executing the Linux command `ps`. After the malicious app determines that one of the target apps is in the foreground, it attempts to establish a Bluetooth connection with its respective device.

A catch here is that, when the official app is in communication with the target device, the malicious app cannot connect to it. To get the data, the malicious app needs to connect to the device right before this legitimate connection is established, right after it completes, or during some disruption of the connection. Below we summarize these options:

- *Pre-connection*. The official apps of these devices, once executed, often need the user's intervention to start the communication with their devices. For example, all the apps for the MyGlucoHealth, iThermometer and the Bodymedia armband have a soft button that needs to be pushed to initiate the connection. These apps can also be configured to attempt automatic connections to their respective devices as soon as they are launched. Therefore, in order to capture data from the target device, the malicious app should be in position to exploit the time gap between the moment it discovers that the target app is running and the moment when the legitimate connection is established (after the soft button is pushed or the automatic connection goes through). The likelihood of this succeeding is contingent on how frequently the malicious app checks currently-running processes, i.e., its *sampling rate* for monitoring the official app.
- *Post-connection*. After discovering the running official app, the malicious app can simply wait until its connection ends and then immediately connect to the device. This strategy avoids aggressive monitoring of the official app: the malicious app can keep a slow sampling rate, as long as it can still detect the target during its execution. There is a risk, however, that the user turns off the target device *before* exiting its app. When this happens, the adversary loses the chance to get data at that specific point.
- *Disruption*. The malicious app can disrupt the legitimate app's communication by deactivating Bluetooth on the phone. It can then reactivate the channel and immediately make a connection to the target device. During this attack, the user might observe the disruption and have to manually click the button on the app again to resume data collection. The approach makes the attack less stealthy but more reliable in getting the data from the target device.

Here we elaborate how we utilize these techniques to launch data-stealing attacks on the healthcare devices.

The attacks. In our study, we execute the data-stealing attacks on all four healthcare devices. To prepare for the attacks, we analyzed the code of these devices' official apps and their Bluetooth traffic captured using `hcidump` [12] to facilitate our understanding of their protocols (for talking to the devices), and further built these protocols into the malicious app. During its operation, the malicious app calls the `getBondedDevices()` API to get a list of external devices already paired with the phone and their bonding information, including the name, the MAC address and the UUID of the device of interest. Using such information, the malicious app makes RFCOMM connections to the device to download sensitive user data.

The attack strategy we implement includes a surveillance component that periodically calls the API `getRunningTasks()` to monitor the execution of the device's official app twice per second. With this implementation, our app can keep a low profile incurring, on average, around only 3mW of extra power consumption. In the meantime, given that human interventions (clicking on a button after the app is activated) can take seconds, our app stands a good chance of capturing the time window before the official app establishes a connection to its device. In case, automatic connection is configured on the target apps, there is a race condition on the socket establishment. To make sure that we do not miss the opportunity to capture data when a target app is launched, our design incorporates both the pre-connection and the post-connection strategies: as soon as the malicious app finds that the target app is running, it first makes a connection attempt; if not successful, the app listens for the asynchronous `ACTION_ACL_DISCONNECTED` event broadcasted by the OS, which notifies the app once a low level –(Asynchronous Connection-Less (ACL)– connection with a remote device ends, and then tries to connect to the target device again if the device is the one disconnected and the disconnection is not caused by the malicious app itself. If either the pre-connection or post-connection attempt succeeds, the malicious app requests and captures the data from the appropriate external Bluetooth device, sends them to the adversary and closes the connection, to make it available to the legitimate app.

It is particularly tricky when the official app is configured to automatic connections: once the pre-connection attack succeeds, the malicious app rapidly finishes its operations and releases the socket that is almost instantly captured by the legitimate app. This causes the OS to miss reporting the DISCONNECT event and the consecutive CONNECT. Hence, when the legitimate app releases the socket, the malicious app believes that the disconnection is initiated by itself. As a result, it skips the post-connection opportunity and thus misses the new data the device collects during the period of the legitimate app's connection. To address this issue, we designed the malicious app to check whether enough time elapses from the moment it sends out a disconnection request to when it receives a disconnection event from the OS; if so, the app believes that the event it gets is about another app and then goes ahead to make another connection attempt.

We run the malicious app on a Nexus 4 development phone running JellyBean (4.2), together with all target devices' apps. We evaluated the effectiveness of the data-stealing attack by observing the success rate when the apps were configured to initiate automatic connections to their respective devices once launched. This is the worst case scenario as this operation is much faster than its alternative where the user must click a button to initiate such connections, hence the window of opportunity is smaller for the pre-connection attempt. Our study shows that the malicious app is often successful in capturing this window. The experimental results are presented in Table I.

For the Bodymedia armband device, we found that in 100 pre-connection trials, the malicious app managed to connect 99 times to the device, get the sensitive data and send them to a remote server. The case that the connection failed was attributed to a device de-synchronization issue that rendered even the official app unable to connect to it. We achieve this high success rate because the Bodymedia Link Armband mobile app does some pre-processing operations before attempting to connect to its device, which gives enough time for the malicious app to perform its operations and release the socket. The success rates were also high for other apps, except for iThermometer (e.g., 42 out of 100 trials) due to its app's prompt response in establishing Bluetooth socket connections. When the malicious app won the race, the authorized app failed to connect but it automatically retried after 10 seconds, and succeeded as this interval was often enough for the malicious app to finish its task and release the socket. The post-connection attacks succeeded most of the time except for the glucose meter, MyGlucoHealth, as long as the devices were switched off after the official apps stopped. MyGluooHealth automatically turns itself off after sending data to its app to save its battery power, so none of the post-connection attacks on it succeeded. We also tried the disruption strategy, which also worked, allowing our app to discontinue the official app's connection and get the health data. A problem with this attack strategy, as discussed before, is that the legitimate connection needs to be interrupted, which could be noticed by the user.

Power consumption evaluation. A rough estimation of the power consumption of different surveillance strategies is important for understanding the stealthiness of the malicious app, because this activity dominates all of its operations in terms of the time interval that it has to run. We tested the different options we had. We evaluated `getRunningTasks` (the strategy we implemented in the malicious app) and its alternatives including calling `getRunningAppProcesses()` and making repeated attempts to connect to or check the existence of the target Bluetooth device. We ran the app using each strategy independently for 10 minutes. The average power consumption of the strategy under scrutiny is illustrated in Table II. As depicted, the one we decided to adopt (`getRunningTasks`) turned out to be both much more efficient and stealthier (as the Bluetooth sign appears on the screen only when it is supposed to be, i.e., when the the official app is running). We further compared the power consumption of this strategy with that of two popular apps, as described in Table III. As we can see here, our surveillance strategy has a comparable power-consumption level (3mW) as those apps (1 to 18mW). Accurate power consumption measurement is not required to do this evaluation, we only need rough *relative* power measurement. The software we used for the power consumption evaluation provides accurate measurement for a very limited number of phones and rough measurements for all Android phones. This rough measurement suffices for this evaluation.

Target Device	Pre-connection	Post-connection
Bodymedia LINK Armband	99/100	100/100
iThermometer	42/100	100/100
Nonin Pulseoximeter	99/100	92/100
myGlucoHealth	100/100	0/100[*]

[*] the device turns off few seconds after sending data to the phone.

TABLE I: Success rate of data-stealing attack. This table depicts the successful connections made by the malicious app on 100 trials.

Technique	Avg Power Consumption	Sampling Rate
getRunningAppProcesses()	8mW	2 samples/s
getRunningTasks()	3mW	2 samples/s
connect()	17mW	0.18 samples/s
startDiscovery()	15mW	0.054 samples/s

TABLE II: Average power consumption over 10 minutes per surveillance technique using PowerTutor[53].

B. Data-Injection Attacks

In addition to the threat of the data-stealing attacks, we found that the presence of the insider, the malicious app, on the phone also makes it possible to inject fake data into the device's official app and its online account for the user. More specifically, this attack works as follows:

1) The malicious app first uses its Bluetooth permissions to collect part of the bonding information for the target device and delivers the information to the adversary.
2) The adversary clones the device using the bonding information (MAC address, UUID and device name) and places the clone in the neighborhood of the original device or other places where the user may come close. With a standard Class 1 Bluetooth device, the clone can stay as far as 100m from the phone.
3) When the user gets into the clone's Bluetooth transmission range, the malicious app resets the link key (in the presence of secure communication) by unpairing the phone from the original device and pairing it with the clone, then it invokes the target app (if it is not already running). The clone then talks to the official app and transmits falsified data to the app.
4) To make the attack stealthy, the malicious app can choose to pair the phone back with the original device after the attack, thus the phone user will not notice that her device has been unpaired. This succeeds most of the time, as the PIN for most Bluetooth devices is either "0000" or "1234" [51]. It should be noted that the original Bluetooth device does not need to be discoverable for pairing, as long as its MAC address is known. Moreover, most of the new devices with Bluetooth 2.1+ use Secure Simple Pairing (SSP) [28], which does not need any user intervention for pairing.

Technique	Avg Power Consumption
Facebook	18mW
getRunningTasks()	3mW
Gmail	1mW

TABLE III: Average power consumption over an hour. Comparison between our surveillance technique and 2 popular applications using PowerTutor[53].

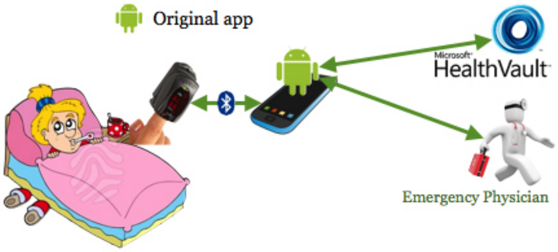

Fig. 2: Normal Scenario

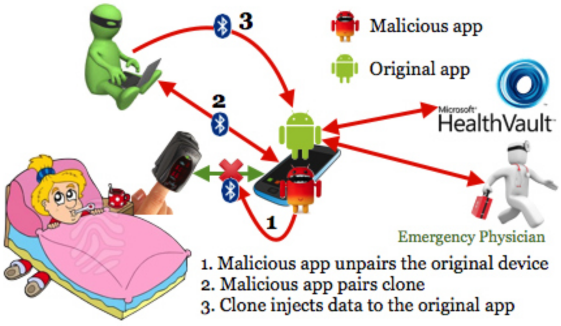
3
2
1
Malicious app
Original app
Microsoft HealthVault
Emergency Physician
1. Malicious app unpairs the original device
2. Malicious app pairs clone
3. Clone injects data to the original app

Fig. 3: Adversary injecting fake data

To make this attack happen, we need to address a few technical challenges, particularly, how to clone the original device, how to stealthily reset the link key when secure connections are in use, and how to connect to the spoofed device in the presence of the original one. Here we elaborate these problems and our solutions.

Device spoofing. To clone a Bluetooth device, all an attacker needs is the target device's name, MAC address and UUID. As discussed before, such information can be easily obtained by calling Android's `getBondedDevices()` method of `BluetoothAdapter` class [24], which gives a list of devices paired with the phone and their bonding data. In our experiment, we ran SpoofTooph [25] on a Linux laptop that masqueraded the original device using its name, MAC and class (i.e., UUID). The spoofed device can be placed wherever the user may come close if the official app of the original device does not have a soft button for activating its Bluetooth connections. An example here is the Bodymedia armband in automatic connection mode. Otherwise, the adversary needs to set it up in the vicinity of the original device. The presence of two devices with the same name, MAC and UUID is hard to detect, as Bluetooth scanners show only one of them. Also, the spoofed device can be much further away from the user's phone than the original device (even outside the door) and still ensure the success of the attack, which we elaborate later in this section.

Link key reset. What gives trouble to this data-injection attack is the link key stored in the operating system that the malicious app cannot get. The link key is used to encrypt Bluetooth packets when the device's official app invokes `createsecureRfcommSocket`. Without knowing this piece of information, the clone cannot talk properly with the app. This is not always a problem: an app can connect to its device without encryption protection (through `createInsecureRfcommSocket`) and even some devices that offer encryption may not provide adequate security. Consider the Nonin pulse oximeter as an example. Its app first tries secure connections, but, if this attempt fails, the app automatically switches to the insecure channel to ensure that the communication can still go through. However, for the device that always sticks to the secure channel, an attacker needs a way to circumvent the defense provided by the link key.

Here is our strategy: if we cannot get the link key, we can simply replace it, setting it to one known to our spoofed device. Given the `Bluetooth` and `Bluetooth_ADMIN` permissions, the malicious app can easily unpair the phone from any Bluetooth device by calling the API `IBluetooth`, which removes the current link key shared with the device. To enable the official app to talk to the clone, however, we need to pair it with the spoofed device. This operation sets a new link key for the clone, which requires the user to enter a PIN to authenticate her phone to the device. (Note that the PIN here is for device-device authentication, not app-device authentication). The PIN itself is not a big issue, as the clone will accept whatever it receives.[4] The problem is that there is no public Android API for programmatically entering the PIN. The phone user's intervention seems inevitable.

A close look at Android source code, however, shows that the OS has a hidden interface that allows programmatic entering of a PIN. All we need to do here is to find a way to use it. The APIs provided by Android communicate with different system services through IPC (Inter-Process Communication) calls, based on those services' interfaces specified by Android Interface Definition Language (AIDL). Actually, a method `setPin()` under Android `IBluetooth` API (Android's private API for Bluetooth services) can be used to programmatically input the PIN. The problem is that this interface is not open to ordinary apps. In our research, we managed to get this interface by using a technique [22] that retrieves the AIDL description of the method from the Android source code and then compiles it together with the code of the malicious app. As a result, the interface becomes visible to the app. Through the interface, a PIN can then be automatically entered for a Bluetooth pairing. To avoid showing any user interfaces that might arouse suspicion from the phone user, the app performs this pairing operation when the screen is off, which can be determined without requesting any permission [54]. We provide a demonstration of this attack on a web page [6].

Connection race. Another technical challenge comes from some apps' soft buttons, which needs to be clicked by the phone user to initiate their Bluetooth communication with the devices. All four device apps we studied can be set in this "button" mode. In this case, automatic triggering of those apps' Bluetooth connections is difficult. Therefore we have to place the clones somehow close to their original devices (within tens of meters), in hopes that when the user starts running the

[4] After the attack, if the malicious app wants to restore the pairing with the original device, oftentimes a default PIN (0000 or 1234) works just fine [51].

official apps, they will mistakenly talk to the clones. This turns out to be pretty realistic: we found that we can set the spoofed device in a way that it almost always wins this connection race, as elaborated below.

A Bluetooth device typically waits for a smartphone to initiate a connection. During this waiting process, it continues to switch between two modes, *page sleep* where it sleeps to save power [35] and *page scan* where it wakes up to look for connection requests. This process is called paging. Let T_{sleep} be the time interval between two scans and T_{scan} the duration of a scan. Obviously, a larger T_{scan} and a smaller T_{sleep} lead to more power consumption, but are more likely to timely respond to the phone's connection requests. According to Bluetooth specifications [35], [26], these parameters are supposed to be set such that $11.25ms \leq T_{sleep} \leq 2.56s$ and $10.625ms \leq T_{scan} \leq T_{sleep}$. For most Android external devices, including all those used in our research, their parameters are chosen for power saving and cannot be modified by the user. The adversary, however, can be more aggressive. In our research, we used the Linux configuration tool `hciconfig` to set these parameters for the Bluetooth dongle on our attack laptop to $T_{scan} = 11.25ms$ and $T_{sleep} = 1.28s$, and ran this spoofed device against all four healthcare devices. We found that the clone almost always won such connection races. This happened even when the original device was more powerful than the clone in terms of radio signal strengths. For example, Nonin pulse oximeter [15] is a Class I Bluetooth device, with a $100mW$ radio and a range up to $100m$, whereas clone was Class II with a $2.5mW$ radio and a range up to $10m$; even under such a disparity, the clone always managed to first connect to the phone even behind a wall and $7m$ away.

The attacks. In our research, we implemented the attack and evaluated them on our NEXUS 4 phone (Android 4.2, 1.5GHz quad-core CPU and 2GB memory) and the medical devices. The experiments were conducted in the presence of both the original devices and their clones (a VM with Intel Core i7 CPU (shared), 1GB memory, a Bluetooth 2.0 dongle), though this is unnecessary when these devices' apps are in the automatic connection mode that enables the malicious app to trigger them to automatically connect to the clones whenever the phone user comes close to the spoofed devices. To make our attacks realistic, we deliberately placed the original devices closer to the phone (0 feet) than the spoofed ones (20 feet away, with a wall in-between). Among all 100 executions of the official apps, the phone was always first connected to the clones, despite their larger distance from the phone. The results are presented in Table IV. Note that in the presence of secure connections, the phone cannot talk to the original device after the reset of the link key. When this happens, however, the phone will get a notification of connection failure if the response from the original device comes first. In our experiment, we never observed such a notification: each time, the phone always smoothly established a connection with the clone.

In all the tests, our app first unpaired the phone from the original device and then paired it with the clone, using a random PIN as an input. All these unpairing and pairing attempts succeeded. Once a connection was established, we observed that the spoofed device easily dumped fake data into the official app, this data was displayed on the phone or the web page for the user's account. A demo is posted online [6].

Distance of cloned device	1 ft	20 ft*
Number of observations	100	100
Distance of original device	0 ft	0 ft
No. of times original device responded	0	0
No. of times cloned device responded	100	100

* with a wall in between

TABLE IV: Data-injection attack launched 1ft and 20ft away from the victim's phone, with the original device touching the phone. In both cases, the experiments were repeated 100 times.

Total apps	90
Apps not using Bluetooth (eliminated)	2
Device apps with sensitive information	68
Device apps with insensitive information	20

TABLE V: Sampled apps

C. Measurement of the DMB Threats

As discussed before, the DMB problem comes from the lack of a bonding between an Android external device and its official app. In the absence of OS-level protection, this threat can only be addressed by the app-device authentication developed by individual device manufacturers. The design and implementation of such an authentication mechanism, however, can be nontrivial, which could raise the cost of the devices. To find out whether such a security measure has already been taken in practice, we performed a measurement study that analyzed a relevant set of apps from Google Play. Our study, for which we give details below, reveals that *all* of the selected apps are actually vulnerable, indicating that the DMB problem is indeed realistic and serious. Given the pervasiveness of vulnerable devices and challenges in fixing them (which could require modifying their hardware), an OS-level solution becomes inevitable (Section IV).

App collection. To collect relevant official apps for different Bluetooth devices, we searched Google Play for those compatible with Google NEXUS 4, using the following terms: "Bluetooth Door Lock", "Bluetooth Health", "Bluetooth Medical Devices" and "Bluetooth Meter". All together, these queries gave us 90 apps. For each of these apps, we manually inspected its descriptions to determine whether it received sensitive user data from its device. Among these 90 apps, 68 involved some private user information, such as the heart rate, blood pressure, body temperature, glucose level, daily activities, and so on as summarized in Figure 4.

Methodology and analysis. To avoid purchasing all those 68 devices, which is too expensive, we analyzed their code to find out whether they included any app-device authentication. This analysis was done both automatically and manually, as follows.

We first decompiled all the 68 apps and searched for authentication-related programming structures. Authentication should be based upon a secret, which was not hard-coded into any of those apps, given the fact that from two independent downloads of the same app, we always got the same code and data. Therefore, such a secret should either come from some external inputs of the app, particularly its user interfaces, web communication or internal memory files, or is generated by cryptographic operations. In our study, we

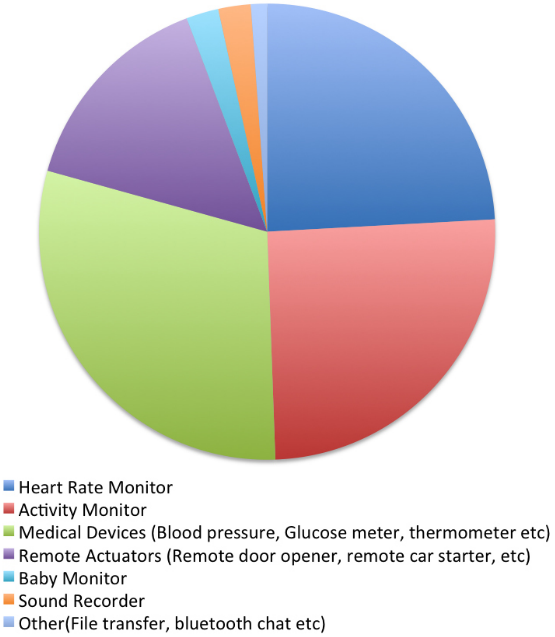

Fig. 4: Classifications of the sampled apps. Some of them collect information in multiple categories.

inspected all such potential sources of authentication secrets (Table VI) to determine whether their outputs affected the inputs of the app's Bluetooth communication, particularly that of BluetoothSocket.write, which transmits data to the device through a Bluetooth socket connection.

We ran a script that used grep to locate the APIs related to those sources and identified the apps where such APIs only appeared within public libraries. For example, we found that, for most apps, their cryptographic APIs (provided by Java JCE, Bouncy castle and spongycastle [14], [4], [17]) were all included in shared libraries such as Google ads, Twitter authentication, OAuth, etc. Those libraries are used for specific purposes, getting ads or performing web-based authentication, for example. It is unlikely that they be used for authenticating the app to its Bluetooth device. Therefore, we removed all the apps that did not have any of those APIs outside the public libraries. There were 48 such apps among all we collected.

For the remaining 20 apps, we manually inspected all the occurrences of these "suspicious" APIs (Table VI) in their code. We looked at the functions where the calls to the APIs were made. It turned out that they were all used for the purposes having nothing to do with app-device authentication. For example, most reads from memory files appeared in the crash-handling mechanisms and most cryptographic operations were performed on the SQL queries on web databases. We also found that HttpClient was used in the functions for sharing tweets or getting the user's workout data from the web. None of these API outputs were propagated to the inputs of the app's Bluetooth communication.

We further installed all the 68 apps and manually inspected their user interfaces. None of them asked for passwords, PINs, etc. for authenticating themselves to their corresponding devices.

Authentication Methods	**Libraries/Functions used**	**Total**	**Apps with app-device authentication**
Crypto	e.g., javax.crypto, bouncycastle	9	0
Internal storage	e.g., openFileInput()	15	0
Web communication	e.g., HttpClient	5	0
UI for app-device authentication	Manual	0	0

TABLE VI: Manual analysis on 20 apps. The other 48 apps were automatically filtered out by the locations of their suspicious APIs.

Summary of the findings. As discussed above, we found no evidence that any of these 68 apps, which were relevant apps in Google Play, performed any app-device authentication. Table VI summarizes our findings. The 48 apps we removed automatically either did not have any suspicious APIs or had such APIs in their shared libraries, including those for advertising, web authentication, crash analysis, etc. For the 20 apps we manually analyzed, 9 called cryptographic APIs in their own code, 5 invoked web APIs and 15 read from memory files. Also, for all the 68 apps we studied, none had user inputs for app-device authentication. Again, none of these apps generated any data flow that affected the inputs of Bluetooth communication functions.

Our study also shows that most of these apps supported secure Bluetooth communication: 42 apps utilized secure socket only; 12 worked under both secure and insecure communications and the rest utilized insecure communication only. This indicates that most of the devices processing sensitive user data do take privacy protection seriously. However, the presence of malicious apps with the Bluetooth permissions on Android renders such device-device authentication insufficient for protecting private user information.

IV. Android Device Binding Control

Our security analysis of existing Android Bluetooth devices shows that most of them are completely unprotected from the mis-bonding attacks. Although theoretically each device manufacturer can provide its own app-device authentication to address the problem, this requires upgrading not only its software (the app) and also the hardware (the external device), thus making the device more expensive. Also, this case-by-case fix renders the quality of security protection for different external devices hard to control. Therefore, we believe that a better solution is to enhance Android to provide an OS-level access control that bonds each device to authorized app(s). In this section, we elaborate our design and implementation of such a technique, called *Dabinder*, and evaluation of its efficacy. *Dabinder* assumes that underlying Android OS is not compromised. The protection mechanism (namely *Dabinder* is developed on framework layer of Android OS.

A. *Overview*

We built Dabinder to effectively control app-device bonding in both pairing and communication stages and to minimize

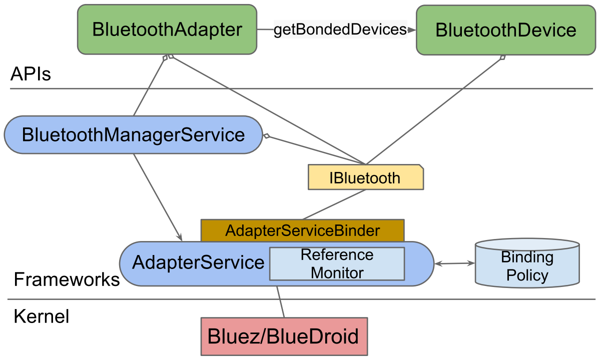

Fig. 5: Bluetooth Subsystem and our defense mechanism: *DaBinder* is built into AdapterService and checks the interaction between apps and Bluetooth devices. It only allows authorized app to access Bluetooth device and keeps bonding policy in a secure storage. *Reference Monitor* and *Binding Policy* blocks (both shown in light-blue) constitute Dabinder.

user involvements in setting access-control policies. Here we describe a high-level design that achieves these two goals.

Architecture. Figure 5 illustrates the architecture for Bluetooth socket communication on Android 4.2, which includes Dabinder components (Reference Monitor and Binding Policy database). To pair a device programmatically, the system calls `setPairingConfirmation` and `setPin` or `setPassKey` of `BluetoothDevice`. To unpair a device, the app uses the API `removeBond`. Alternatively, it can invoke the settings program to control the Bluetooth adapter. In both cases, an IPC request needs to be sent to `AdapterService` to control the Bluetooth device. Once a bond (pairing) is established, the app can make a socket connection to access the device. To this end, again it first needs to talk to the `BluetoothAdapater`, to get a list of paired devices. From this list, the app identifies the target device (MAC) and further requests a socket through the object `BluetoothDevice`. This request is also delivered using an IPC, through the `IBluetooth` interface, to `AdapterService`, which creates the socket for the connection.

In our design, the whole security mechanism is built into `AdapterService`, including a component that controls socket establishment (within an authorized app-device pair[5]) and one that manages the unpairing operation (which can only be performed by an authorized app[6]). Such access controls are based on a set of security policies that unambiguously bonds a device to its authorized app, which are generated automatically by the system from what is observed from the phone's Bluetooth operations.

How it works. Here we explain how Dabinder works. Once the device is activated, it is paired with its authorized app by the phone user. This pairing operation is observed by Dabinder, which then generates a *bonding policy* that associates each device (name, MAC and UUID) to its official app (that is, its Linux user ID or UID). Whenever Android receives a Bluetooth socket-connection request from an app, the policy enforcement mechanism checks whether the app is associated in the bonding policy to the device it is trying to talk to: if the app is not on the device's bonding policy, the request is denied; otherwise, it is allowed to proceed. In this way, our mechanism defeats the data-stealing attacks. Also, Dabinder runs an unpairing controller to manage the operations to dissolve a pairing relation between the phone and a device: in the absence of the bonding policy between the device and the app that requests such an operation, the app is considered to be unauthorized and its unpairing attempt is stopped. As the data-injection attack is contingent on resetting the link key for the phone-device communication, it cannot work without unpairing the phone from the original device. Therefore, such an attack cannot go through.

B. Design and Implementation

Here we present the detailed design of Dabinder, which we implemented in our research on a Galaxy NEXUS 4 phone with Android 4.2. As described before, our design includes mechanisms for generating and maintaining security policies, and for enforcing these policies during phone-device pairing and app-device connection establishment. All these mechanisms were implemented within `AdapterService`.

Policies identification. Critical to Dabinder's mission is the security policies on the legitimate bond between an app and an external device. Such a policy can certainly be manually specified, but it is highly desired that it can also be automatically generated, without the user's intervention if she prefers to do so. In our design, this policy-identification operation is performed within `AdapterService`, when our policy enforcement mechanism inspects a pairing request and its follow-up connection request: if an app is the first one to make a socket connection to the device after the device is paired with the phone, our mechanism automatically adds this app-device relation to a policy database as a new bonding policy.

To securely and persistently maintain these policies in the system, `AdapterService` keeps a Bluetooth MAC Address and `UID` mapping in the `Settings.Secure` key-value storage, which is persistent and read-only to the apps, and can only be modified by the phone user or a system program. The user can manage these policies from `BluetoothManagerService` through a user interface built upon two functions exposed by `AdapterService`, `addDevApp` and `removeAppDev`. In particular, she can explicitly declare an exemption policy for a device, thus allowing it to be accessed by any app.

Connection control. To communicate with an external device, an app needs to establish a connection with it through a Bluetooth socket. Such a socket is created by the system, through a call to the `BluetoothDevice` APIs: `createRfcommSocket`, `createRfcommSocketToServiceRecord`, `createInsecureRfcommSocket` or `createInsecureRfcommSocketToServiceRecord`. A straightforward solution here is to instrument these APIs in order to control the creation of Bluetooth sockets. The problem is that such mediation actually happens in the user land, inside

[5] App-device pair is our terminology for creating a bond between an app and Bluetooth device as opposed to conventional Bluetooth pairing which creates a bond between the phone and the device.

[6] Authorized app: The app that has already established bond to the device.

individual apps' address space. As a result, there is no guarantee that it cannot be circumvented. Also, such a policy compliance checking needs an additional IPC to AdapterService, to get the policies from the system. Instead, in our research, we modified AdapterService.connectSocket, a system function all these APIs have to invoke, for policy compliance checking and enforcement. Whenever the function is called, it first searches the policy database for the device according to its MAC address. If the device is found, our enforcement mechanism continues to look for its related bonding policies. In case, the app does not appear on any of them (i.e. the device has been connected before and is not exempted from the bonding protection), we consider that a policy violation is detected and the request is denied. Otherwise, connectSocket returns a socket and allocates the corresponding resources, such as file descriptors for the connection.

This policy enforcement is implemented on the Android framework layer. Apparently, the app including native code such as createBondNative and removeBondNative may still touch the Bluetooth device on the Linux layer, as illustrated in Figure 5. In our research, we inspected the Bluetooth interface on Linux and found that it is actually included in the Linux group bluetooth. For the app with BLUETOOTH_ADMIN and BLUETOOTH permissions, it can get into the groups net_bt_admin and net_bt, but not bluetooth. As a result, it will not be able to directly access the Linux Bluetooth resources, even through its native code, due to the Linux access control. Under all circumstances, the app needs IPC calls to transfer the execution to a system process in order to use kernel resources. Therefore, we conclude that our protection mechanism cannot be circumvented even by the native code.

Unpairing control. In the presence of secure Bluetooth communication, a malicious app needs to first unpair the phone from the original device before pairing it with the clone to reset the link key. To prevent this unauthorized unpairing, Dabinder interposes on the function removeBond within AdapterService. Whenever an unpairing request is received from the IBluetooth interface, our mechanism checks it against the bonding policy retrieved from the policy dataset: if the app that sends the request is not the authorized one on the policy, this request is denied. Alternatively, we can pop up an interface on the phone to alert the user to the unpairing request and allow the operation to proceed with her permission.

A problem here is that some devices do not use secure Bluetooth communication, which enables the spoofed device to talk to the official app even without knowing the link key. Fortunately, our measurement study shows that most of devices collecting sensitive user data do support encrypted communication (Section III-C), though some of them can also automatically switch to the insecure one when the secure connection fails. To address this issue, Dabinder provides an optional policy through which the phone user can require that any communication with a certain device must be encrypted. In case, the policy is violated (that is, a device is switching to the insecure communication), we can choose to stop the communication and alert the user for further instructions. This happens in AdapterService, within the method connectSocket which checks whether SEC_FLAG_AUTH and SEC_FLAG_ENCRYPT on flag are set.

Functions	Original	Dabinder	Delays
BluetoothSocket	0.0317/0.0059 ms	0.0353/0.0153 ms	0.0036 ms
connectSocket	63.1670/14.7098 ms	86.5152/14.2201 ms	23.3482 ms
removeBond	0.5319/0.1863 ms	0.5493/0.1822 ms	0.017ms

TABLE VII: Dabinder performance evaluation. (mean/sd)

C. Evaluation

We evaluated our implementation to understand its effectiveness in protecting the communication with external devices and its performance impacts on the phone's normal operations. All the experiments were conducted on the Galaxy Nexus phone with a Dual-core 1.2 GHz Cortex-A9 and 1GB memory, Android 4.2 operating system and BlueDroid stack.

Effectiveness. To understand the effectiveness of our approach, we ran it against all the data-injection and data-stealing attacks discussed in Section III. All these attack attempts were thwarted. Specifically, for all the data-stealing attacks, Dabinder stopped the malicious app from making socket connections to the target device, as these connections violated the policy it automatically detected during the pairing stage of the phone. When it comes to the data-injection attacks, our implementation blocked all the attempts to unpair the phone from the devices and therefore defeated the attacks when the secure communication was in use. Also, our approach denied the establishment of a socket for insecure connection required by the Pulse Oximeter app.

Performance. We further evaluated the performance of Dabinder, comparing the execution times for establishing sockets and unpairing a device with and without its policy inspection and enforcement. Specifically, we measured the performance of a set of functions using the code instrumented before and after their executions. The results are illustrated in Table VII.

Here, BluetoothSocket creates a Bluetooth socket, connectSocket builds a socket connection and removeBond unpairs the phone from a device. As we can see from the table, for all these functions, the delay is mainly caused by connectSocket, about 23ms on average. It should be noted that only 8 devices can have Bluetooth connection simultaneously to a smartphone. Moreover, a phone cannot have more than 30 RFCOMM sockets active at the same time, as RFCOMM have only 30 channels [27]. For external devices, they typically can accommodate only one or two connections. Actually, all data from the device is downloaded through a single Bluetooth connection. Therefore, this 23ms delay will not bring in any noticeable inconvenience to the phone user.

V. Discussion

External device mis-bonding is an emerging security challenge for mobile operating systems. Our study reveals its serious consequences and makes a first step toward mitigating this threat. However, what we have done has just scratched the surface of this problem domain. Further efforts are needed to better understand the problem and come up with effective solutions. Following we discuss two examples of possible future research on this issue.

Mis-bonding problems on iOS and other channels. The fundamental cause for the DMB problem is that Android does not differentiate devices attached to the same channel and therefore cannot bind an app to a device. Other mobile OSes may have the same issue. In particular, iOS does not seem to provide any means for tying an app and a device together, nor does it control the access to different devices through other channels. Further research on this OS platform could lead to interesting findings. Also, we have not explored other channels on the phone. For example, NFC is increasingly used for connecting financial apps to external devices for the operations such as payment. Apparently, the same mis-bonding problem also exists on this channel: it is possible that an app only needs to get the NFC permission to access an NFC-enabled device, even if the app is not authorized to do so. If this is true, those NFC devices can also be subject to the data-stealing attacks described in Section III-A. More investigations are needed to understand this threat.

Bonding control. In our research, we developed a technique that controls the binding between an app and a Bluetooth device. The same idea could be applied to other channels and OSes. For example, if device mis-bonding indeed becomes a problem for NFC, we might be able to establish a relation between an NFC device and its app, and ensure that such a relation will not be broken without the phone user's consent. Also, in case, iOS is found to be vulnerable to the DMB threat, a new bonding control mechanism becomes imperative for protecting the external device attached to iPhone, iPad and iPod. Follow-up work on this direction is certainly important.

VI. Related work

Health Device Security. The prior works on the security issues of health devices are the most closely related to our work. Rahman et al. [46] identified several vulnerabilities on Fitbit, a wireless wearable fitness device, which can be leveraged to inject data into the device and launch a denial of service attack against it. Li et al. [38] look into the security weaknesses of glucose monitoring and insulin delivery systems and proposed the technologies for protecting those devices' operations using rolling-code and body-coupled communication. Also, Marti et al. [39] lay out a few necessary requirements for building a secure mobile health care system. All such prior work focuses on the security problems of a specific health device or the communication protocol it uses, whereas our research aims at understanding the security implications of the way Android handles its external devices, in the presence of malicious apps running on the phone.

Android Permission. Android permission system has been under scrutiny for years [42], [47], [41], [33], [30], [45], [41], [29], [40]. Much has been proposed to extend this security model, allowing the phone user to selectively grant permissions to apps [40], deny those with dangerous permission combinations [34], utilize app-defined fine grained access control [42] or leverage IPC provenances for security protection [32]. However, all these prior approaches are designed to guard a phone's local resources. In contrast little has been done on mobile OSes to protect the external devices that connect to smartphones. In particular, on Android, an app that acquires the permissions to use a channel (e.g., Bluetooth, NFC, etc.) is automatically granted the access to any device attached to this channel. There is nothing to bind a device to its authorized app. This problem could be mitigated by SE-Android [49], through labeling the app and the device, and design of a security policy to link them. However, this all depends on the knowledge of the device and app in advance, and a careful setting of the right policy, which oftentimes needs the help from the expert. Also, SE-Android has not been extensively used. Its utility in the presence of diverse applications of Android phones is still not clear.

Bluetooth Security. Bluetooth is a short-range wireless communication technology designed to replace the serial cable links. With its increasing utilization by different devices – including health devices, remote car control, activity monitoring, etc. – security and privacy concerns are running high. Since Bluetooth 2.1, link-layer encryption is required and cannot be turned off, but encryption and authentication alone cannot address all issues with Bluetooth. Prior research [50] summarizes several traditional Bluetooth attacks, e.g., Bluesnarfing, Bluet buggin, Bluejacking, etc. Further security analyses have yielded discoveries of many weaknesses in Bluetooth systems or protocols, which enable the adversary to crack keys, PINs [36], [48], perform Man-in-the-Middle attacks [37] and propagate malware [52], etc. Different from all such prior research, our study focuses on the way mobile OSes handle Bluetooth devices, instead of the Bluetooth system and the protocol themselves. This has never been done before, to the best of our knowledge.

VII. Conclusion

Smartphones are becoming increasingly sophisticated, with their functionalities enriched by assorted external devices such as Bluetooth headsets, medical devices, creditcard readers, etc. This new development, however, also brings in new security and privacy challenges. The OSes these phones use, particularly Android, are not designed to protect the secure interaction between an external device and its authorized app. As a result, oftentimes, *any* app with the permissions to use the communication channels utilized by such a device, such as Bluetooth, NFC, Audio port, etc., automatically gain the access to the device, even if it is not authorized to do so.

In this paper, we reported the first study on this external-device mis-bonding threat, which we found to be both realistic and serious. Particularly, our in-depth analysis on four popular health devices, including a Bodymedia Armband, a thermometer, a Pulse Oximeter and a Glucose meter, shows that all of them are vulnerable to different types of DMB attacks. In a data-stealing attack, the malicious app with the Bluetooth permissions can download sensitive user data from the devices without being noticed, using the side-channel information revealed by Android to determine the right moment to attack. In a data-injection attack, the app can collect the pairing information of the target device and reset the link key, which enables an adversary to place a spoofed device that masquerades the medical device and injects the fake data into the phone user's medical accounts. All these attacks can succeed even in the presence of Bluetooth secure communication, which is designed for protecting device-device communication, not the interaction between a device and an app. Our measurement study, which analyzed 68 device apps randomly sampled from Google Play, further demonstrates that most existing devices are vulnerable to this DMB threat.

To mitigate the threat, we developed a new OS-level protection mechanism, called Dabinder. Our approach automatically identifies the binding relations between an app and its external device from the phone's activities, and then uses such relations as security policies. These binding policies are enforced during the phone's runtime. Whenever an unauthorized app attempts to connect to an external device or unpair the phone from the device, Dabinder detects and blocks the attempts. Our study shows that this new approach works effectively against Bluetooth DMB attacks and incurs a negligible performance impact.

Our study on the DMB threat here focuses on Bluetooth, and therefore is only a first step toward understanding and mitigating this new security issue. Follow-up investigations on the problem in a broader context are important and expected to happen in the near future.

ACKNOWLEDGMENT

We thank our shepherd, William Enck, and the anonymous reviewers for their comments. This work was supported in part by NSF CNS 09-64392 (NSF EBAM) and HHS 90TR0003-01 (SHARPS). The authors with Indiana University were supported in part by the NSF CNS-1017782, CNS-1117106, CNS-1223477 and CNS-1223495. The views expressed are those of the authors only. We also thank Holly B. Cogliati (EPFL) for proofreading.

REFERENCES

[1] Alivecor ecg. [http://www.alivecor.com/].

[2] Android accounts for 75 percent market share. [http://www.zdnet.com/android-accounts-for-75-percent-market-share-windows-phone-leapfrogs-blackberry-7000015496/].

[3] Bodymedia link armband. [http://www.bodymedia.com/].

[4] Bouncycastle library. [http://www.bouncycastle.org/].

[5] Dabinder android open source. https://github.com/DabinderAndroid/extDroid.git.

[6] Demos for our attacks. [https://sites.google.com/site/edmbdroid/].

[7] Entra health systems, myglucohealth. [http://health2con.com/source/gbmf_frontend/company/show/59].

[8] Fitbit one. [http://www.digifit.com/fitbit/index.asp].

[9] Fitbit zip. [http://www.digifit.com/fitbit/zip/].

[10] Foracare testngo. [http://www.foracare.com/glucometer-Testngo.html].

[11] Foracare testngo. [http://www.myglucohealth.net/].

[12] hcidump. http://www.linuxcommand.org/man_pages/hcidump8.html.

[13] ithermometer. [http://www.ithermometer.info/].

[14] Java cryptography extension. [http://www.oracle.com/technetwork/java/javase/documentation/index.html].

[15] Nonin onyx ii pulseoximeter. [http://www.nonin.com/PulseOximetry/Finger/Onyx9560].

[16] Nonin onyx ii pulseoximeter specs. [http://www.nonin.com/products.asp?ID=39&sec=2&sub=9].

[17] Spongycastle library. [http://rtyley.github.io/spongycastle/].

[18] Square up. [https://squareup.com/].

[19] Tmg muscle fatigue monitor. [http://www.tmg.si/].

[20] Withings blood pressure monitor. [http://www.digifit.com/withings-blood-pressure-monitor/index.asp].

[21] Zephyr heart rate monitor. [http://www.zephyr-technology.com/products/bioharness-3/].

[22] Getting ibluetooth instance. http://snipplr.com/view/49526/, 2011.

[23] Bodymedia puts a spin on the ordinary testing procedure. http://blog.bodymedia.com/page/6/, 2012.

[24] Android bluetoothadapter class. http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html, 2013.

[25] Spooftooph. http://www.hackfromacave.com/projects/spooftooph.html, 2013.

[26] S. Bluetooth. Specification of the bluetooth systemversion 2.0, 4. november 2004.

[27] S. Bluetooth. Rfcomm with ts 07.10. *Bluetooth SIG*, 2003.

[28] S. Bluetooth. Bluetooth core specification version 2.1+ edr. *Specification of the Bluetooth System*, 2007.

[29] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[30] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] C. Daniels. Why is too much insulin bad? [http://www.livestrong.com/article/423665-why-is-too-much-insulin-bad/].

[32] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[33] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[34] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[35] M. Handy and D. Timmermann. Time-slot-based analysis of bluetooth energy consumption for page and inquiry states.

[36] M. Jakobsson and S. Wetzel. Security weaknesses in bluetooth. In *Topics in Cryptology CT-RSA 2001*, pages 176–191. Springer, 2001.

[37] D. Kügler. man in the middle attacks on bluetooth. In *Financial Cryptography*, pages 149–161. Springer, 2003.

[38] C. Li, A. Raghunathan, and N. K. Jha. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *e-Health Networking Applications and Services (Healthcom)*, pages 150 – 156, 2011.

[39] R. Marti, J. Delgado, and X. Perramons. Security specification and implementation for mobile e-health services. *eee*, 00:241–248, 2004.

[40] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[41] M. Ongtang, K. Butler, and P. McDaniel. Porscha: policy oriented secure content handling in android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 221–230, New York, NY, USA, 2010. ACM.

[42] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[43] E. Ozdalga, A. Ozdalga, and N. Ahuja. The smartphone in medicine: a review of current and potential use among physicians and students. *Journal of medical Internet research*, 14(5), 2012.

[44] J. Padgette, K. Scarfone, and L. Chen. Guide to bluetooth security. *NIST Special Publication*, 800:121, 2012.

[45] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the*

26th Annual Computer Security Applications Conference, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[46] M. Rahman, B. Carbunar, and M. Banik. Fit and vulnerable: Attacks and defenses for a health monitoring device. *CoRR*, abs/1304.5672, 2013.

[47] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, 8(3):36–44, 2010.

[48] Y. Shaked and A. Wool. Cracking the bluetooth pin. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 39–50, New York, NY, USA, 2005. ACM.

[49] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Preceding of Network and Distributed System Security Symposium*, 2013.

[50] C. A. Soto. A menu of bluetooth attacks. [http://gcn.com/articles/2005/07/20/a-menu-of-bluetooth-attacks.aspx], 2005.

[51] D. Spill and A. Bittau. Bluesniff: Eve meets alice and bluetooth. In *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[52] S. Töyssy and M. Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2):109–119, 2006.

[53] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114, 2010.

[54] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, pages 1017–1028, New York, NY, USA, 2013. ACM.