SMASheD: Sniffing and Manipulating Android Sensor Data

Manar Mohamed
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, Alabama
manar@uab.edu

Babins Shrestha
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, Alabama
babins@uab.edu

Nitesh Saxena
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, Alabama
saxena@uab.edu

ABSTRACT

The current Android sensor security model either allows only restrictive *read access* to sensitive sensors (e.g., an app can only read its *own* touch data) or requires special install-time permissions (e.g., to read microphone, camera or GPS). Moreover, Android does not allow *write access* to *any* of the sensors. Sensing-based security applications therefore crucially rely upon the sanity of the Android sensor security model.

In this paper, we show that such a model can be effectively circumvented. Specifically, we build *SMASheD*, a legitimate framework under the current Android ecosystem that can be used to *stealthily* sniff as well as manipulate many of the Android's restricted sensors (*even touch input*). *SMASheD* exploits the Android Debug Bridge (ADB) functionality and enables a malicious app with only the INTERNET permission to read, and write to, multiple different sensor data files at will. *SMASheD* is the first framework, to our knowledge, that can sniff and manipulate protected sensors on unrooted Android devices, without user awareness, without constant device-PC connection and without the need to infect the PC.

The primary contributions of this work are two-fold. *First*, we design and develop the *SMASheD* framework. *Second*, as an offensive implication of the *SMASheD* framework, we introduce a wide array of potentially devastating attacks. Our attacks against the touchsensor range from accurately logging the touchscreen input (*TouchLogger*) to injecting touch events for accessing restricted sensors and resources, installing and granting special permissions to other malicious apps, accessing user accounts, and authenticating on behalf of the user — essentially almost doing whatever the device user can do (secretively). Our attacks against various physical sensors (motion, position and environmental) can subvert the functionality provided by numerous existing sensing-based security applications, including those used for *(continuous) authentication*, and *authorization*.

1. INTRODUCTION

Sensing-enabled computing is rapidly becoming ubiquitous. With mobile device manufacturers embedding multiple, low-cost hardware sensors onto the devices and mobile OS providers adding full software support for developing applications using these sensors, there is a transformational growth in the adoption of mobile devices.

CODASPY'16, March 09-11, 2016, New Orleans, LA, USA
© 2016 ACM. ISBN 978-1-4503-3935-3/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2857705.2857749

The most common categories of sensors available on the current breed of mobile devices, smartphones, smartwatches and smartglasses, include: (1) *user input sensor* (touchscreen and hardware buttons), (2) *audio-visual sensors* (microphone and camera), (3) *navigational sensors* (e.g., GPS), (4) *motion sensors* (e.g., accelerometer and gyroscope), (5) *position sensors* (e.g., magnetometer and proximity), and (6) *environmental sensors* (e.g., pressure, and temperature). Mobile device sensors are a cornerstone of a wide range of security and privacy applications, including those geared for authentication and authorization (e.g., [6, 7, 15, 20]).

Since mobile sensors provide potentially sensitive information about the host device, the device's user or the device's surroundings, protecting sensor data from abuse by malicious applications becomes paramount. Consequently, most mobile platforms have established a sensor security access control model. Specifically, Android, one of the most popular mobile OSs and the subject of this paper, follows a model where *read access* to many sensitive sensors is very restrictive (e.g., an app can only read its *own* touch data) or requires special install-time permissions granted by the user (e.g., to access microphone, camera or GPS). The read access to most other sensors, including motion, position and environmental sensors, is not restricted within this model because Android may not consider these sensors as explicitly sensitive. Moreover, Android security model does not allow *write access* to *any* of the sensors. Clearly, the sensing-based applications therefore crucially rely upon the sanity of the Android sensor security model.

In this paper, we demonstrate that the current Android sensor security model can be effectively circumvented to a large extent. Specifically, we build *SMASheD*, a legitimate systems framework under the current Android ecosystem that can be used to *stealthily* sniff (read) as well as manipulate (write to) many of the Android's restricted sensors. To be precise, *SMASheD* can be used to: (1) directly sniff the touchsensor, (2) directly manipulate the touch, motion, position and environmental sensors, and (3) indirectly, using the touch inject capability, sniff the audio-visual and navigational sensors. *SMASheD does not* require the device to be rooted.

SMASheD exploits the Android Debug Bridge (ADB) functionality and enables a malicious app with only the INTERNET permission to read from, and write to, multiple sensor data files at will. ADB is a functionality designed to allow Android app developers with extended permissions to systems resources that are otherwise protected by the Android sensor security model. This workaround is legitimate and has been used by many apps in Google Play Store such as screenshot apps [9], sync and backup apps [3], and touch record/replay apps [22]. All of these apps ask the user to connect her device to a PC via USB, launch ADB and run a native service with ADB privilege. The app then communicates with this service to obtain access to the resources which Android deems as protected.

As part of *SMASheD*, we develop a service that provides read and write sensor events functionality. This functionality can be hidden inside any service that requires the ADB workaround, e.g., a screenshot service. When installing an app, the user is usually made aware of the permissions that she is granting to the app. However, while installing and executing the service through ADB, the user is completely oblivious as to what permissions the service might have. Also, *SMASheD* can be published for debugging or any other benign purposes but can contain malicious code that will utilize the functionality provided by the service for malicious purposes. Moreover, such services can be exploited by malicious apps in a similar way as presented in [11]. Our *SMASheD* platform encompasses a native service and an Android app.

Our Contributions: In this paper, we expose the vulnerability underlying the ADB workaround allowing us to read from and write to many Android sensors currently protected by the Android access control model. Equipped with this powerful capability, we then go on to present the offensive implications in many security contexts. The research contributions of our work are outlined below:

1. **A Framework to Sniff & Manipulate Android Sensors** (Section 3): We design and develop the *SMASheD* framework to sniff and manipulate many restricted Android sensors.
2. **Powerful Adversarial Applications** (Section 4): As a significant offensive implication of the *SMASheD* framework, we introduce a broad array of potentially devastating attacks. Our attacks include the following:
   (a) Logging the touchscreen input, leading to the first full-fledged, highly accurate *TouchLogger*.
   (b) Injecting touch events for accessing restricted sensors and resources (e.g., microphone, camera or GPS), installing and granting special permissions to other malicious apps (translating into many known malware schemes, such as [17, 23, 24], without the need for the user to grant special permissions), accessing user accounts and authenticating on behalf of the user – essentially *almost doing whatever the device user can do* (secretively).
   (c) Manipulating various physical sensors (motion, position and environmental) in order to subvert the functionality provided by many sensing-based security applications, including those used for *(continuous) authentication* (e.g., [6, 7]), and *authorization* (e.g., [10, 20]).

2. BACKGROUND: ANDROID SENSOR SECURITY MODEL

Android's core security principle is to protect user data, system resources and apps from malicious apps. Android utilizes the Linux approach of process isolation to enforce the isolation of apps and operating systems components. This isolation is achieved by assigning each app a unique User Identifier (UID) and Group Identifier (GID) at the app installation time. Therefore, each app is enforced to run in a separate Linux process, called *Application Sandbox*, and the Linux process isolation ensures that an app cannot interfere with other apps or access system resources unless permissions are explicitly granted. In order to allow apps to communicate with each other and access system resources, Android provides a secure Inter-Process Communication (IPC) protocol.

Discretionary Access Control (DAC) is the typical access control employed in Linux. In DAC, the owner/creator of the data sets the access permissions of the data to three types of users: the owner, the users in the same group and all other users. When an app is installed, Android creates a home directory for the app (i.e., /data/data/app-name) and allows only the owner to read from and write to this directory. The apps signed with the same certificate are able to share the data among each other.

File system permissions are also used to restrict the access of system functionality. For example, /dev/cam permission is set to allow only the owner and the users in the camera group to read and write to the camera sensor as shown in Listing 1. When an app requests the CAMERA permission, and if the permission is granted, the app is assigned the camera Linux GID, which would allow it to access /dev/cam. The mapping between the Linux groups and permission labels are set in platform.xml, and ueventd.rc is responsible for setting the owners and groups for various system files.

Some Android resources do not require any permission. In particular, reading motion, position and environmental sensors is globally permitted. Most of the other resources require read-write permissions, and these permissions have four levels:

1. *Normal*: The app needs to request the access, however, the system grants the permission automatically without notifying the user (e.g., vibrate).
2. *Dangerous (protection level 1)*: The system grants the permission to the app only if the user approves granting this permission (e.g., accessing camera, microphone, or GPS).
3. *Signature (protection level 2)*: The system grants the permission to the app only if the requesting app is signed with the same certificate as the app that declared the permission, without notifying the user (e.g., allowing two apps signed by the same developer to access each other components, inject event).
4. *SignatureOrSystem (protection level 3)*: The system grants the permission only to the apps that are in the Android system image or that are signed with the same certificate as the app that declared the permission (e.g., system reboot).

In any Linux system, an executable runs with same permission as the process that has started it. ADB shell is already assigned to several groups (graphics, input, log, adb, sdcard_rw, etc). Therefore, any executable that starts through the ADB shell is granted the same level of access to the resources which belong to any of these groups. As shown in Listing 1, since the directory "/dev/input/*" which contains the sensor files, belongs to "input" group, and the ADB shell has read-write access to all the resources associated with "input" group, any executable that is initiated by ADB shell can read from and write to the "/dev/input/*" resources. This is the key idea upon which our *SMASheD* framework is based, allowing us to sniff and manipulate many of the Android's sensors.

Listing 1: ueventd.rc file

```
...
/dev/input/*    0660    root        input
/dev/eac        0660    root        audio
/dev/cam        0660    root        camera
...
```

3. SMASHED DESIGN, IMPLEMENTATION AND THREAT MODEL

In this section, we explain the design, implementation and threat model of our proposed *SMASheD* framework.

3.1 Design Overview

As mentioned in Section 2, the current Android security model considers many resources as sensitive and thus limits the access of these resources only to the apps that are signed by the system (protection level 3 for the permissions declared by the system and protection level 4). These protected resources include: injecting user events into any window (INJECT_EVENTS), taking screen shots (READ_FRAME_BUFFER), and reading system log files (READ_LOGS). However, Android allows access to these resources through the ADB shell for development purposes, by assigning the ADB shell to the groups that can access these resources. For example, the ADB shell is assigned to the input group which allows any process with the ADB shell privilege to read from and write to any of the files in the /dev/input/ directory. This directory contains the files associated with user input, motion, position and environmental sensors.

Moreover, Android's current directory structure has the /data/local/tmp/ directory which is assigned to shell user and shell group, and gives read, write and execute permission to the shell user and any user in the shell group. This folder allows the user to run executable files on their Android devices through ADB shell.

Many developers have exploited these capabilities given to the ADB shell to grant permissions to their apps that are not otherwise allowed. This ADB workaround is performed by developing a native service, pushing it into the /data/local/tmp/ directory and running the service through the ADB shell. This way the native service grants all the permissions that are granted to the shell. Finally, to allow other apps to communicate with the service, both the app and the service open sockets and communicate through it. This approach has been utilized by many apps that are already published in Google Play Store such as apps that allow the users to take screenshots programmatically [9], sync and backup [3], USB tethering [2], and touch record/replay [22].

The above design allows any app with only the INTERNET permission to communicate with the service. Hence, the app with only the INTERNET permission will obtain access to the resources that the service provides *without the user's knowledge*. This vulnerability has been explored in [11], focusing mainly on screenshot apps published in Google Play Store. The authors developed an app, Screenmilker, which communicates with the native services of many screenshots apps. They showed that Screenmilker is able to collect user's sensitive data, such as user's credentials on many banking apps by sending requests to the screenshot's native service to take screenshots while the user is inputting her credentials. (A detailed comparison of our *SMASheD* framework with related prior work is later provided in Section 6).

In this paper, we are exploring and extending this vulnerability further, and with potentially much broader consequences. We focus on INJECT_EVENTS permission. There are already some apps in Google Play Store, such as *FRep* – Finger Replayer [22], which allow users to record their touch interactions with their devices and replay them later. *FRep* has already been installed by 100,000 to 500,000 users. These apps also utilize the ADB workaround, similar to the screenshot apps, in order to gain access to the read and inject touchscreen data. Also, as the communication between the touch repeater app and its native service is done through a socket, the native service becomes accessible to any app installed on the phone with only the INTERNET permission. Therefore, if the user installs any malicious apps with the INTERNET permission, these apps can also communicate with the service and read/inject touch events maliciously.

We implement the *SMASheD* framework which encompasses three components: *SMASheD* server: a native service that provides the sensor data reading and injection capabilities, scripts: two simple scripts used to copy the *SMASheD* server to the device and to start the server, and *SMASheD* app: an app that runs a status detection module in the background, and depending on the phone's status and the desired functionality, it sends requests to the *SMASheD* server to read or inject sensor events. Figure 1 depicts the overall *SMASheD* architecture.

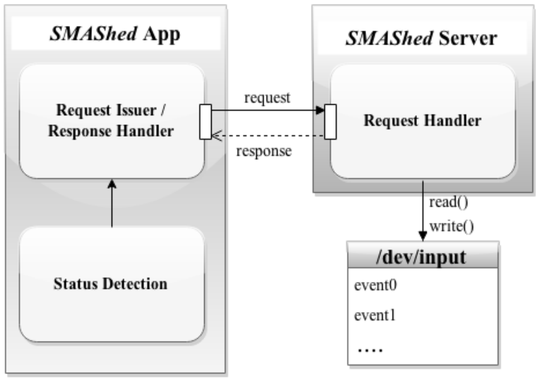

Figure 1: The architecture of *SMASheD*

3.2 SMASheD Server

Our system works with the sensors whose events are made available to apps through low-level event interface and have files under the directory /dev/input/, and not through system services (e.g., camera, microphone, and GPS). This includes user input, motion, position and environmental sensors. For each of these sensors, a corresponding file named event$x$ exists in the directory /dev/input/. Android allows reading and injecting sensor events through ADB commands $getevent$ and $sendevent$, respectively.

Each hardware event generates multiple input events. Each input event encompasses $time$, $type$, $code$ and $value$.

- $time$ represents the time at which the event occurred.
- $value$ represents the value of the event.
- $code$ is the event code and it precisely defines the type of the event. For example, REL_X, REL_Y, REL_Z represent relative changes in X, Y and Z axes, respectively.
- $type$ is the event type, which groups the event's codes under a logical input construct. Each event type has a set of applicable event codes. For example, EV_ABS represents the absolute axis value changes, EV_REL represents the relative axis value changes. A special event type, EV_SYN, is used to separate input events into packets of input data changes occurring at the same moment in time.

For a complete list of the applicable events' types and codes, we refer the reader to linux/input.h[1].

As an example, a simple touchscreen press-release event generates around 19 input events. Listing 2 in Appendix A displays a sample output of executing $getevent$ command, pressing on the screen at point (946,1543), and then releasing the touch. BTN_TOUCH DOWN and BTN_TOUCH UP indicate the beginning and the end of the touch, ABS_MT_POSITION_X and ABS_MT_POSITION_Y represent the touch's x and y positions.

We implemented a native service designed in C with code similar to Android's $getevent$ and $sendevent$ for reading and injecting the sensor events. First, the service scans /dev/input/ directory to find out what sensors are available in the device. Although the file names in the directory are event0, event1, etc, we use *EVIOCGVERSION ioctl* function to retrieve the name of the sensor that corresponds to each file. To read from and write to the sensors' files, we use *read()* and *write()* functions.

[1]https://github.com/torvalds/linux/blob/master/include/uapi/linux/input.h

To allow other apps to communicate with the service, the service creates a socket. The socket keeps on listening to the incoming requests. In the current implementation, the service accepts three kinds of requests: $read$, $stop$ and $inject$.

- $read$: The service reads the input events from all the sensors. We can limit the read to a subset of sensors to improve the efficiency. The service continues reading until it receives a $stop$ request.
- $stop$: The service stops reading the sensor events. Then, it either writes the events to a file, and sends the file name as a response to the request or sends all the read input events.
- $inject$: Inject needs to have a file name or a list of sensors events as an argument. The service injects the sensors events in the incoming list or in the file to their corresponding sensors files.

3.3 Scripts

We wrote two shell scripts to start the service. The first shell script is responsible for pushing the native service and the second script to /data/local/tmp/ folder on the device, and for starting the second script. The second script starts running the service. In this way, the service will run with the same privileges as the shell user. The service will then keep running until the phone is switched off or it gets killed by the user.

3.4 SMASheD App

We implemented an Android app, which only requires the INTERNET permission. The app connects to the *SMASheD* server through socket and sends requests to $read$ and $inject$ events. For example, it may send $read$ touch events when a banking app is open to retrieve the password input by the user.

In order to determine whether a specific app that the attacker might be interested in is running, our app has a service that starts when the app is launched and keeps running in the background. The service runs *ps* command periodically, every 100 ms, until the app that the attacker is interested in is launched (status detector shown in Figure 1). Once the app under attack is running and on the foreground, *SMASheD* app connects to the *SMASheD* server through socket and sends $read$ request with the list of sensors (e.g., touchscreen data only, all sensors, etc). Once the user exits the app or moves out of the app, *SMASheD* app sends $stop$ request to the *SMASheD* server. In case the purpose of reading is to replay the sensor events later in the same device, to reduce the communication between the *SMASheD* server and app, the *SMASheD* server stores the read events in a file and only sends the file name to the *SMASheD* app. Otherwise, the *SMASheD* server sends all the sensor events.

Also, the *SMASheD* app can send $inject$ request along with a list of sensor events to inject or a file name previously acquired from the service, whenever it wants to inject sensor events.

3.5 Threat Model

Our threat model is highly realistic, facilitated under three scenarios:

1. **Already Installed Benign ADB Services**: Apps that read and inject touch events (e.g., FRep [22]) are already available in Google Play Store and installed by many users. Given such an already installed benign app, our attacks that read/inject touch events work under the exact same threat model as [11] by using a malicious app that communicates with the service associated with the already installed app.
2. **Future Benign ADB Services**: Benign developers can publish an app/service that reads/injects sensors events for providing some benign functionality (e.g., debugging or testing). Once such an app is installed, attacker will launch our attacks similar to [11].
3. **Malicious ADB Services**: The attacker can create a benign-looking (malicious) screenshot app, adding read/inject sensor events functionality to its service. The attacker just needs to fool users into installing this app. Note that when user installs a service using ADB, he/she is not notified about the resources the service is accessing. Therefore, the user will not be able to differentiate between services that only take screenshots from services with added malicious functionality. Moreover, if the attacker can gain physical access to an unlocked Android device, the attacker can quickly install the malicious service on the device (e.g., in a lunch-time attack) [16].

The first and second threat model scenarios exploit the vulnerability of the services that expose their ADB functionalities to all the apps installed on the same device with INTERNET permission (same as [11]). The last scenario exploits Android's vulnerability of granting all the shell privileges to any service installed via ADB *without notifying the user*.

SMASheD works on unrooted devices, and does not require an infected PC (unlike [8]) or a constant connection between the device and a PC (e.g., unlike monkeyrunner [2]).

4. ATTACKS USING SMASHED

In this section, we present various attacks that can be performed based on the sensor data reading-writing capability provided by *SMASheD*. The entire spectrum of attacks that *SMASheD* can enable, especially those involving touch injection, is possibly very broad. As such, our exposition is not exhaustive. However, we introduce some of the most interesting and potentially devastating attacks targeting both real-world applications and research systems.

4.1 Sniffing Touchscreen Input (Touchlogger)

We will demonstrate how *SMASheD* can be used as a TouchLogger in order to sniff a user's sensitive information. According to Android security model, an app cannot read touch events performed by the user on other apps [6]. However, we will show how it is possible to infer the keys that the user has pressed, and therefore extract sensitive information efficiently and with 100% accuracy. We note that *SMASheD* is not only able to detect user key presses but it can also log any interaction of the user with the touchscreen.

The *SMASheD* app can send *read* request to the *SMASheD* server to obtain all the events the user performs on the touchscreen. However, getting only the raw touch events is not enough to hamper the user privacy. Moreover, the attacker will be interested only in a small subset of these events. For example, an attacker will be interested in learning the password of the user on banking apps but not the input corresponding to user's interaction with a game.

According to the information the attacker wants to learn about the user, the attacker can modify the service in the *SMASheD* app so it sends *read* request when the app corresponding to the data the attacker wants to collect is launched. Moreover, if the attacker wants only to learn the keys the user presses while the app is running, he can send the *read* request when both the app and the keyboard are on the phone's foreground.

To evaluate the ability of *SMASheD* to extract the username and password from various banking apps, we repurposed the original *SMASheD* app. When the *SMASheD* app is launched, it gets the list of installed apps on the device using *getPackageManager* API. Note that no permission is required to get the list of installed apps. Then, the *SMASheD* app looks for the apps that are already installed and are of interest to the attacker. The *SMASheD* app also finds the soft keyboards installed. The *SMASheD* app starts running its status

[2]http://developer.android.com/tools/help/monkeyrunner_concepts.html

detection service in the background, which regularly executes ps command to find out the list of running applications. Once any of the apps that the attacker wants to collect user data from appears in the output of the execution ps command, the service gets that app process ID, PID_{app}. The service also gets the process ID of the keyboard, PID_{kb} from the execution of the ps command. The service then executes $ps - tPID_{app}$ command, which returns the list of threads of that process; whenever an app is on the foreground, the list of threads of that app has a thread named "GL updater".

If the app is running the "GL updater" thread, the service also checks if the keyboard is running "GL updater" thread. If both the app and the keyboard app have "GL updater" thread running, *SMASheD* detects that both of them are in the foreground and sends $read$ request to the *SMASheD* server. When user exits the app or the keyboard (which can be detected by checking if the app is no longer in the list returned by executing the ps command, or if "GL updater" is not in the list of the thread running for either the app or the keyboard), the *SMASheD* app sends $stop$ request to the *SMASheD* server. As a response to the $stop$ request, the *SMASheD* server sends all the touch events to the *SMASheD* app. The *SMASheD* app parses the events and extracts the events with event type ABS_MT_POSITION_X and ABS_MT_POSITION_Y (Listing 2 in Appendix A). Finally, it maps the x and y coordinates to keys (keyboard layout can be detected by determining which soft keyboard the user is using, the orientation of the device and the screen dimensions) and sends the text typed, for example, to the attacker's web service via HTML request, or via other methods as we will discuss in Section 4.2.3.

4.2 Manipulating Touchscreen Sensor

The ability of injecting touch events could be extremely dangerous. In essence, it will allow the malware to do whatever the user can do with her device. The primary challenge for the attacker is to be stealthy. To do so, the attacker should inject the touch events while the user might not be attending to the phone, such as when the user is asleep or the phone is left inside a pocket or a purse. Such contextual scenarios can be determined by monitoring various motion and environmental sensors on the phone, as shown by prior research [14]. For example, the attacker can monitor the proximity and light sensors to infer when the phone is inside a pocket or placed in dark [25]. Moreover, *SMASheD* can change the phone settings, e.g., decrease screen brightness, mute sound and erase logs/-traces to make the attacks "user-invisible."

Following subsections layout some of the attacks that *SMASheD* can perform given its capability to inject touch events.

4.2.1 Installing Apps Bypassing Permissions

SMASheD can install apps (benign or malicious) with extended permissions, available from Google Play Store, or any other website by injecting touch events on the infected device. To do so, *SMASheD* first sends an intent either to open the URL of the website where the malicious app resides, or to Google Play Store's app page. As the interface of the Google Play Store app is standard, *SMASheD* can inject touch events on the install button and then the accept button to grant the app with the requested permission. The position of the touch events can be calculated based on the screen dimensions. *SMASheD* can then close the Play Store app and clean-up any installation-related notifications. Similarly, *SMASheD* can open the malware-hosting website, download the APK, install the malware by clicking on the downloaded APK, grant the malware the desired permissions, and clean-up the traces. The installed malware apps can then do whatever they are designed to do against the phone or the user.

4.2.2 Permission Escalation

SMASheD can utilize already installed apps to compromise user's privacy. For example, *SMASheD* can open the camera app and collect images of the user's surroundings to learn sensitive information about the user, similar to PlaceRaider [23], but without asking the user to grant the CAMERA permission to the *SMASheD* app. Similarly, *SMASheD* can open an audio recording app and monitor the ambient audio. Also, *SMASheD* can open any installed app having the GPS permission, acquire the location of the user, and take a screenshot of the app displaying the location by pressing and holding down the Power and Volume Down buttons. *SMASheD* then can either send the snapshot to the attacker or perform simple image processing to extract the user location, given that *SMASheD* knows the app's layout and the screen dimensions.

Other possible attacks include: making phone call to premium rate numbers by opening the phone dialer and pressing the calling button, sending SMSs via a messaging app, sending the contact list of the user by opening the contact app, and sharing all the contacts via email or SMS with a remote attacker, changing the phone settings (such as toggling WiFi, GPS, etc) through the Settings app, muting the phone, and so on.

4.2.3 Data Exfiltration

Whenever *SMASheD* needs to send any data to a remote attacker (e.g., previously sniffed passwords, credit card numbers or pictures), it can stealthily transmit this data utilizing other apps, such as email or SMS. As some malware detection mechanisms detect malicious apps based on abnormal data usage, *SMASheD* can remain surreptitious and undetected by such systems. Moreover, *SMASheD* can delete the logs from the email and SMS apps so that users cannot trace back. This simple strategy will prevent *SMASheD* from being detected by either the device user or the anti-virus apps. Such an exfiltration will also avoid the need for doing any data processing on the infected device itself but rather allow the attacker to outsource all processing to a remote machine.

4.2.4 Phone Unlock

In order to allow *SMASheD* to access any of the device resources that require the device to be unlocked, *SMASheD* needs to unlock the device first. To do that, *SMASheD* first utilizes the TouchLogger presented in Section 4.1 to log the user's PIN or pattern unlock while the user unlocks his phone. Then, whenever *SMASheD* wants to unlock the phone, it will simply inject the recorded PIN or pattern unlock onto the touchscreen.

4.2.5 Accessing User Accounts

SMASheD can be used to open different apps that require authentication, and log into user's accounts. To do so, *SMASheD* will first extract the user's credentials for the target account by using the TouchLogger described in Section 4.1. *SMASheD* will then utilize this credential to log into the user account from her device. Accessing the user accounts from the *SMASheD* infected device is important for several reasons. Many web services and banks implement a second factor authentication approach which may only allow the user to login from a registered device. Similarly, many banks require the user to answer security questions when she logs in from a different device, and others send notification to the user specifying the devices that are used to access her account. After having logged into the user accounts, *SMASheD* can, for example, access the account and perform any kind of the allowed banking transactions, read user's emails, send fake emails, forward the emails to a remote attacker, or read user's private data from or post messages on social media sites — the possibilities are endless.

4.2.6 Attacking Biometric Authentication

Recently, a significant amount of research has been done to authenticate a user transparently using biometrics. The touch-based biometrics are applied either as a second factor authentication mechanism during the device unlock or as a continuous authentication mechanism when the user is performing some activity on the device. Among these, some systems analyze the keystrokes of the users to capture the biometrics while others analyze touch gestures provided by the users. We now analyze a variety of these biometrics systems proposed in the literature and provide a systematic methodology to attack them using *SMASheD*.

Keystroke Biometrics: *Maiorana et al.* [13] present an approach to authenticate users based on their typing habits on the smartphones. Their approach relies on the analysis of keystroke dynamics. The system acquires and processes the time stamps generated by the mobile phones related to key press and release. Using these, the system further calculates different features such as Manhattan distance, Euclidean distance and statistical features and generate a template for each user. During the authentication, the system computes the normalized distance and compares that with a threshold.

To authenticate against such system, *SMASheD* needs to learn how the user types. During the learning phase, *SMASheD* can record the user's keystroke behavior and compute the features in a similar way to the authentication system. After learning, it can create the keystrokes such that the time interval *SMASheD* presses and releases the keys closely correlates with that of the user. Note that *SMASheD* can simply record and replay the user's keystroke without computing the features and the system may still fail to detect such malicious input. However, creating new keystrokes after learning the features is more detrimental to the user as the attacker can recreate any events or activities he likes.

Touch Gesture Biometrics: *Frank et al.* [6] present "Touchalytics", a continuous touch-based authentication system which utilizes the strokes performed by the user while using her phone. Touchalytics focuses on single touch gestures such as sliding horizontally and vertically. To authenticate using touch, Touchalytics records the touch coordinates, finger pressures, the screen areas covered by each finger, and times. Touchalytics extracts 30 different features from these raw inputs. Touchalytics uses these features to build a profile of the user and utilizes it later to identify the user.

Since Touchalytics is monitoring and matching the touch with the trained data for horizontal and vertical slides only but not with other actions, *SMASheD* can perform tap/click and pinch without getting detected. However, to navigate up/down or right/left where *SMASheD* has to provide such horizontal/vertical slides, *SMASheD* needs to record the previous authentic slides from the user, and later inject them as desired. While outsider attacks using robots [18] have previously been reported against Touchalytics, the *SMASheD* attack represents the first known insider attack to our knowledge.

Shahzad et al. [19] present "GEAT" for screen unlocking based on simple gestures. Along with the user touch input, GEAT uses other features such as finger velocity, device acceleration, stroke time, inter-stroke time, stroke displacement magnitude, stroke displacement direction, and velocity direction. GEAT segments each stroke into sub-strokes of different time duration where, for each sub-stroke, the user has consistent and distinguishing behavior. GEAT utilizes these features to train and later identify the user.

Since GEAT is only authenticating when user wants to unlock the screen, *SMASheD* can record all the raw touch and device acceleration data during the legitimate authentication by the user. It can later just replay the touch providing the recorded data such that the features would fully match.

Luca et al. [5] present another transparent authentication approach that enhances password patterns with an additional security layer. They study the touch stroke gestures corresponding to the horizontal slide and the pattern unlock. Their approach uses dynamic time warping for the analysis of touch gestures using different features including XY-coordinates, pressure, size, time, and speed of the touch.

SMASheD cannot only thwart the password pattern to unlock the device but also foil the additional security layer provided by this system. As discussed in the Section 4.1, *SMASheD* first simply sniffs the password pattern. In addition, *SMASheD* records the pressure, size, time and speed of the touch when the legitimate user performs the pattern unlock gesture. Now, when the *SMASheD* app needs to unlock the device, it simply injects the previously recorded touch events to circumvent the authentication functionality.

4.2.7 Attacking Touch-based Authorization

Roesner et al. [15] propose the *user-driven access control* system where permission is granted using user actions rather than using manifests or system prompts. It introduces *access control gadgets* (ACGs). Each user-owned resource exposes UI elements, ACGs, which are embedded by the apps. The user's UI interaction with the ACG grants the app permission to access the corresponding resources. The system assumes that the kernel has complete control of the display and the apps cannot draw outside the screen space designated for them. Furthermore, it assumes that the kernel dispatches UI events only to the app with which the user is interacting.

The threat model of the system tries to restrict access such that only one app gets the permission from the user interaction, while other apps do not. It does not assume that the touch can be injected. No app will have permission to use the resource until the user explicitly interacts with the ACGs embedded by the app. To attack this system, *SMASheD* can provide the touch input to any app. For example, if *SMASheD* wants to make a phone call, it needs to interact with and provide touch to the phone calling ACG of the app. Since the system receives the touch, it will permit the app to make the phone call. In summary, *SMASheD* can fully bypass this system by injecting simple touch events.

Chaugule et al. [1] present a defense against unauthorized malicious behavior by utilizing the keypad or touchscreen interrupts. The system differentiates between malware and human activity by analyzing the presence of touch input which generates a hardware interrupt. Their approach especially focuses on preventing unauthorized messaging. The system assumes that the operating system is within the Trusted Computing Base and the hardware is not compromised. It assumes that the kernel memory interfaces are not exported to userspace so that userspace applications are not allowed to write into kernel memory and alter kernel control flow. They claim that there is no direct way in which the touchscreen interrupt handler will be called from userspace code unless the operating system is tampered with.

SMASheD can break this claim by providing the touchscreen interrupt without tampering with the operating system. *SMASheD* can provide touch screen input while sending the text message. When the touch event is injected, it will provide the necessary hardware interrupt that the system is looking for and hence any app will be authorized to send the messages.

4.3 Manipulating Other Sensors

In this section, we first describe the systems which provide different security or non-security functionality based on the motion, position and environmental sensors. Then, we provide an attack scheme against each system using the sensor event injection capability of *SMASheD*.

Attacking these systems may not be straightforward. The best scenario to manipulate the sensor readings is the one where the current sensor readings are not being altered by the natural events. For example, when the phone is in a pocket, the light sensor may not change or report constant values. Since the sensor file will not be altered by the natural environment in this case, the malware can manipulate the sensor data as it likes. Also if the system is implementing a statistical approach (such as based on mean, standard deviation, etc., of the sensor data), the malware may not even need to manipulate the sensors for the whole duration when the system is monitoring the sensors. *SMASheD* can insert some values that significantly changes these statistical features which causes the system to misjudge the sensing context, thereby failing to provide its intended security functionality. For the other systems, which implement specialized algorithms based on continuous sensor data, *SMASheD* needs to inject sensor readings at different timestamps that correlate to the sensor values during benign case.

4.3.1 Attacking Authentication Systems

Conti et al. [4] propose a system that transparently authenticates the user by analyzing her hand movement gesture while she is making or answering a phone call. It uses accelerometer and orientation sensor to detect the proposed gesture. The system uses the dynamic time warping distance (DTW-D) algorithm to verify if the authorized user is making or answering the phone call.

To attack this system, *SMASheD* can record the accelerometer and orientation sensor data when the user is making or receiving a valid call. Later, when *SMASheD* wants to make a call (e.g., to premium rate numbers or to user's contacts), it can replay the previously recorded sensor data.

Gascon et al. [7] present an approach to continuously authenticate users on smartphones by analyzing their typing motion behavior. Along with touch input, it also records the timestamps when the keys are pressed or released. The system uses different motion and position sensors such as accelerometer, gyroscope and orientation sensors to capture behavioral biometrics so as to authenticate the user. It extracts various features leading to a 2376-dimensional vector representing the typing motion behavior of a user in a given time frame. The system is trained with the linear SVM classifier.

To attack this system, *SMASheD* needs to learn how the user presses each character, and then reproduce it later. During the learning phase, *SMASheD* continuously records all the raw sensor data until it gets necessary information used by the system for all the keys during the legitimate key presses. Once the learning phase is complete, *SMASheD* can provide the touch injects with proper timings and the corresponding sensor readings. Since the motion and position sensors are continuously recording the data from the hardware, *SMASheD* may need to wait for a favorable time, e.g., when the phone is static, otherwise the natural readings may interfere with the injected sensor readings possibly leading to rejection by the system.

4.3.2 Attacking Authorization Systems

We now consider various systems that provide the authorization functionality to access mobile device resources/services. The main purpose of these systems is to differentiate a human user from a bot so as to authorize access to the requesting app. To authorize human-vs-bot actions, these systems capture different explicit and implicit gestures provided by the user measured using multiple sensors.

Li et al. [10] present "Tap-Wave-Rub". They propose multiple gestures that can be used for the purpose of authorization. An implicit gesture, such as tapping the phone with another device (*tap*), is used to provide NFC permission to the requesting app. The system uses accelerometer sensor to detect the *tap* gesture. An explicit gesture, such as waving a hand in front of the phone (*wave*) or rubbing a finger near the proximity sensor (*rub*), is used to grant permissions for the services where no implicit gesture can be used. To detect *wave* and *rub* gestures, the system uses proximity sensor. *Shrestha et al.* [21] also present "WaveToAccess", in which another mechanism for *wave* gesture detection is proposed. It utilizes the light sensors to infer the fluctuation in light due to hand waving and the accelerometer sensor to reduce the possibility of detecting other events as hand wave. Both Tap-Wave-Rub and WaveToAccess assume that the kernel is immune and the sensor data cannot be manipulated by the malware.

SMASheD attacks the assumption made by these systems. To generate the *tap*, *wave* or *rub* gesture, the attacker can record his own gesture and later inject the recorded values via *SMASheD*. Alternatively, *SMASheD* can record the gesture provided by the user during the benign case and replay it later. A simpler attack can be performed on *wave* and *rub* gestures in Tap-Wave-Rub, in which *SMASheD* fluctuates the proximity sensor in quick succession so that the system infers the corresponding gesture.

Shrestha et al. [20] later present a similar defense to mobile malware using *transparent human gestures*. The system uses the hand movement gesture to prevent unauthorized access of the services such as phone calling, picture snapping and NFC tapping. It looks for multiple, motion, position and environmental sensor data to detect the calling, snapping and tapping gestures. The assumption that the system makes is the device is already infected with malware. However, the device kernel is healthy and is immune to the malware infection, and also that the malware is not capable of manipulating the sensors.

SMASheD can attack the assumptions made by these systems. The attacker can record the sensor data that is being used by these systems to detect the gesture during call, snap or tap. Now, when malware is trying to make a call, snap photo or tap NFC tag, *SMASheD* can replay all these sensor data fooling the system to believe that the user is performing the activity.

5. SMASHED MITIGATION

To protect against the adversarial applications of *SMASheD* (Section 4), we suggest the following potential mitigation strategies. Although these strategies may not fully prevent the attacks, they may help reduce the impact of the underlying vulnerability.

First, we believe that it is important to raise people's awareness of the possible security risks associated with installing services through the ADB shell. Second, we suggest following the permission models of Android for native services that are executed through the ADB shell. In the current model, any native service that starts through the ADB shell is granted all the permissions that the shell has without notifying the user. These permissions include accessing logs, frame buffer, motion, position, environmental, and user input sensors. An attacker may not reveal all the resources that the service is accessing. For example, the attacker could publish a service as a snapshot service while injecting code that accesses sensor files as well. This may be prevented if the service is only granted permissions after informing the user. Third, we suggest enforcing security policies for the communication between processes running on the device through sockets. We recommend that Android monitors the open sockets on the device and the apps that are accessing those sockets. Whenever an unusual communication is detected, Android should at least inform the user. Whether or not users would pay attention to such notifications is an independent concern. However, we believe that the potential risks should be conveyed to the users.

6. RELATED WORK

Our paper is not the first to study the vulnerability underlying the ADB workaround. Recently, Lin et al. [11] developed Screenmilker, a malicious app that can glean sensitive information from the mobile device's screen (specifically passwords of banking apps) by communicating with a snapshot service installed through ADB. Screenmilker exploits the vulnerability of snapshot services of exposing their ADB capabilities to any app with only INTERNET permission installed on the same device.

Also, Hwang et al. [8] presented "Bittersweet ADB", a set of conceptual attacks using ADB ranging from private data leakage to usage monitoring and behavior interference. The threat model in [8] assumes that the user has enabled USB debugging on her phone and forgot to disable it and later her PC got infected with a malware such as the one explained in [12] which installs an ADB service on the connected device. Then, whenever the user connects her device to her infected PC, the malware on the PC installs a malicious service with ADB capabilities on the user phone. The authors also developed an app that can enable USB debugging without user knowledge. Although Android 4.2.2 and higher display a dialog asking the user to allow debugging via PC when the device is connected to a PC, the authors assume that the users would just accept. The authors proposed the use of static analyzer to detect the proposed attacks. The static analyzer checks if the private information resulted from executing ADB command is sent outside the Android device via socket API.

In our paper, we extensively expanded the scope and the impact of the ADB vulnerability to much more devastating, stealthy and accurate attacks than the one proposed in [11] and with a weaker (more realistic) threat model than [8]. We comprehensively and systematically exposed the vulnerability of sniffing and manipulating many protected Android sensors, and translated it into a wide spectrum of catastrophic attacks against real-world and research systems. Our proposed framework is the first, to our knowledge, that sniffs and manipulates protected sensors on unrooted Android devices, without user awareness, without constant device-PC USB connection (unlike the monkeyrunner tool) and without the need for an infected PC (unlike [8]).

7. CONCLUSION AND FUTURE WORK

In this paper, we called the Android's sensor security model into question. We exploited Android's ADB workaround to develop a framework that can effectively sniff and manipulate many sensors currently protected by Android's access control model. Our framework can be used to: (1) directly sniff the touchscreen sensor data, (2) directly manipulate the touchscreen, motion, position and environmental sensor data, and (3) indirectly, using the touch inject capability, sniff the audio-visual and navigational sensors. Based on this framework, we introduced a wide spectrum of potentially devastating attacks that can compromise user privacy and subvert many security applications that rely upon different sensors.

We believe that our framework can facilitate many other applications beyond the ones we presented, which we also plan to explore in our future work.

References

[1] A. Chaugule, Z. Xu, and S. Zhu. A specification based intrusion detection framework for mobile phones. In *Applied Cryptography and Network Security*, 2011.

[2] ClockworkMod. Clockworkmod tether (no root). https://goo.gl/qg2e80.

[3] ClockworkMod. Helium. https://goo.gl/ceW329, 2013.

[4] M. Conti, I. Zachia-Zlatea, and B. Crispo. Mind how you answer me!: transparently authenticating the user of a smartphone when answering or placing a call. In *ACM Symposium on Information, Computer and Communications Security*, 2011.

[5] A. De Luca, A. Hang, F. Brudy, C. Lindner, and H. Hussmann. Touch me once and i know it's you!: Implicit authentication based on touch screen patterns. In *SIGCHI Conference on Human Factors in Computing Systems*, 2012.

[6] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE Transactions on Information Forensics and Security*, 2013.

[7] H. Gascon, S. Uellenbeck, C. Wolf, and K. Rieck. Continuous authentication on mobile devices by analysis of typing motion behavior. In *Sicherheit*, 2014.

[8] S. Hwang, S. Lee, Y. Kim, and S. Ryu. Bittersweet adb: Attacks and defenses. In *ACM Symposium on Information, Computer and Communications Security*, 2015.

[9] E. Kim. No root screenshot it. https://goo.gl/hksbHY, 2013.

[10] H. Li, D. Ma, N. Saxena, B. Shrestha, and Y. Zhu. Tap-wave-rub: Lightweight malware prevention for smartphones using intuitive human gestures. In *ACM conference on Security and privacy in wireless and mobile networks*, 2013.

[11] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *Network and Distributed System Security Symposium*, 2014.

[12] F. Liu. Windows malware attempts to infect android devices. http://goo.gl/x2Dwc2. Accessed: 2015-08-08.

[13] E. Maiorana, P. Campisi, N. González-Carballo, and A. Neri. Keystroke dynamics authentication for mobile phones. In *ACM Symposium on Applied Computing*, 2011.

[14] J.-K. Min, A. Doryab, J. Wiese, S. Amini, J. Zimmerman, and J. I. Hong. Toss'n'turn: smartphone as sleep and sleep quality detector. In *ACM conference on Human factors in computing systems*, 2014.

[15] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[16] D. Rogers. *Mobile Security: A Guide for Users*. lulu.com, 2013.

[17] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[18] A. Serwadda and V. V. Phoha. When kids' toys breach mobile phone security. In *Conf. on Computer & Communications Security*, 2013.

[19] M. Shahzad, A. X. Liu, and A. Samuel. Secure unlocking of mobile touch screen devices by simple gestures: You can see it but you can not do it. In *Mobile Computing & Networking*, 2013.

[20] B. Shrestha, M. Mohamed, N. Saxena, and S. Tamrakar. Curbing mobile malware based on user-transparent hand movements. In *Pervasive Computing and Communications*, 2015.

[21] B. Shrestha, N. Saxena, and J. Harrison. Wave-to-access: Protecting sensitive mobile device services via a hand waving gesture. In *Cryptology and Network Security*. Springer, 2013.

[22] strAI. Frep - finger replayer. https://goo.gl/2F5k7J, 2015.

[23] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. Placeraider: Virtual theft in physical spaces with smartphones. *Network and Distributed System Security Symposium (NDSS)*, 2013.

[24] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng. Stealthy video capturer: a new video-based spyware in 3g smartphones. In *ACM conference on Wireless network security*, 2009.

[25] J. Yang, E. Munguia-Tapia, and S. Gibbs. Efficient in-pocket detection with mobile phones. In *ACM conference on Pervasive and ubiquitous computing adjunct publication*, 2013.

APPENDIX

A. SAMPLE OUTPUT FROM GETEVENT

Listing 2: Sample output from running *getevent* for a single press release

```
[69934.435503] EV_ABS  ABS_MT_TRACKING_ID    0000038d
[69934.435533] EV_KEY  BTN_TOUCH            DOWN
[69934.435564] EV_ABS  ABS_MT_POSITION_X     000003b2
[69934.435564] EV_ABS  ABS_MT_POSITION_Y     00000607
[69934.435595] EV_ABS  ABS_MT_TOUCH_MAJOR    00000012
[69934.435595] EV_ABS  ABS_MT_TOUCH_MINOR    00000009
[69934.435625] EV_ABS  ABS_MT_WIDTH_MAJOR    00000002
[69934.435625] EV_ABS  003c                  ffffffa6
[69934.435778] EV_SYN  SYN_REPORT            00000000
[69934.452105] EV_ABS  ABS_MT_TOUCH_MAJOR    00000024
[69934.452105] EV_ABS  ABS_MT_TO UCH_MINOR   0000001b
[69934.452135] EV_ABS  ABS_MT_WIDTH_MAJOR    00000008
[69934.452135] EV_ABS  003c                  fffffffd
[69934.452166] EV_SYN  SYN_REPORT            00000000
[69934.462847] EV_ABS  003c                  00000000
[69934.462877] EV_SYN  SYN_REPORT            00000000
[69934.494371] EV_ABS  ABS_MT_TRACKING_ID    ffffffff
[69934.494402] EV_KEY  BTN_TOUCH             UP
[69934.494402] EV_SYN  SYN_REPORT            00000000
```