Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information

Antonio Bianchi
University of California, Santa Barbara
antoniob@cs.ucsb.edu

Eric Gustafson
University of California, Santa Barbara
edg@cs.ucsb.edu

Yanick Fratantonio
University of California, Santa Barbara
EURECOM
yanick.fratantonio@eurecom.fr

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Today's mobile applications increasingly rely on communication with a remote backend service to perform many critical functions, including handling user-specific information. This implies that some form of authentication should be used to associate a user with their actions and data. Since schemes involving tedious account creation procedures can represent "friction" for users, many applications are moving toward alternative solutions, some of which, while increasing usability, sacrifice security.

This paper focuses on a new trend of authentication schemes based on what we call "device-public" information, which consists of properties and data that any application running on a device can obtain. While these schemes are convenient to users, since they require little to no interaction, they are vulnerable *by design*, since all the needed information to authenticate a user is available to *any* app installed on the device. An attacker with a malicious app on a user's device could easily hijack the user's account, steal private information, send (and receive) messages on behalf of the user, or steal valuable virtual goods.

To demonstrate how easily these vulnerabilities can be weaponized, we developed a generic exploitation technique that first mines all relevant data from a victim's phone, and then transfers and injects them into an attacker's phone to fool apps into granting access to the victim's account. Moreover, we developed a dynamic analysis detection system to automatically highlight problematic apps.

Using our tool, we analyzed 1,000 popular applications and found that 41 of them, including the popular messaging apps WhatsApp and Viber, were vulnerable. Finally, our work proposes solutions to this issue, based on modifications to the Android API.

CCS CONCEPTS

• **Security and privacy** → *Mobile platform security*; *Software security engineering*;

KEYWORDS

mobile-security, vulnerability, authentication

ACM Reference format:
Antonio Bianchi, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2017. Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information. In *Proceedings of ACSAC 2017, San Juan, PR, USA, December 4–8, 2017,* 12 pages.
https://doi.org/10.1145/3134600.3134615

1 INTRODUCTION

Mobile applications ("apps") have evolved from being simple conveniences, into complex systems, aimed at powering the latest generation of Internet-connected, distributed, massively multi-user services. This implies that these apps depend to some extent on backend services to function. For example, many apps function as frontends for existing online services, where their entire behavior is tightly coupled to the remote service. To handle multiple users securely on these backends, some sort of authentication needs to occur.

Traditionally, this procedure relies on a combination of "user-private" credentials, such as username and password. However, given the incredibly crowded market in which these apps compete and the fickle nature of users, there is a significant pressure to lower the "friction" new users encounter when using an app. For this reason, applications are moving away from authentication schemes based on user-private credentials, toward those schemes that are more automatic. An existing solution that is often used to accomplish this is OAuth, an authorization mechanism that can enable users to leverage accounts on identity services, such as Google and Facebook, without creating new, ad-hoc, ones. Nonetheless, developers constantly strive to create novel, custom authentication mechanisms to increase the ease-of-use of their applications.

In this paper, we study and characterize a new broad class of vulnerable authentication schemes, which fully rely on what we call *device-public* information. With this term, we refer to all information, properties, and data that can be accessed by any application (with proper permissions, as explained in Section 3.2) installed on the same device.

As an example, consider a messaging app that, after users identify themselves, stores a token in the device's external storage, which any app can access. The app then sends this token to the app's backend server each time it is used, as a form of authentication. This technique has the advantage that if the user uninstalls the app and later wants to use it again, the token will persist on the external storage, and no re-authentication will be required. Unfortunately, this versatility comes at a price: a malicious app running on the same device can obtain and leak this token to a remote attacker, who can now easily hijack the user's account. Even if an app is leveraging a technology such as OAuth, poor handling of the resulting tokens could render them device-public as well. This is just one possible scenario for the mis-use of device-public information; apps can and do use such schemes as the *only* form of authentication, without requiring private data from the user such as a password, rendering their associated accounts wide-open to malicious apps on the device.

In this paper, we perform the first comprehensive analysis and characterization of vulnerable authentication schemes based on device-public information. We start by describing the *identity-transfer* attack, a generic exploitation technique, composed by two steps. First, a malicious app, termed the "ID Leaker," steals all device-public information from a victim's device without any user interaction. Then, this app transfers this data to "ID Injector," an app installed on the attacker's device that collects the received information and injects them into the device. Once this step is completed, the attacker can simply install the vulnerable target app, which will automatically log the attacker in the victim's account.

We also take the first step toward understanding how widespread this class of vulnerabilities is, by developing a dynamic analysis system that aims at uncovering potentially-vulnerable apps among a much larger set. While "authentication" is a difficult behavior to characterize, we can leverage interesting behavioral patterns to locate authentication with enough accuracy to help a human analyst determine if a vulnerability is present. In particular, the system we developed records the app's user interface behavior during its first execution on a device, when authentication and registration is likely to appear. Then, as a second step, the system wipes the app's private data (by uninstalling and then re-installing the app), and it runs the app once again. The key intuition is that if the behavior of an app changes after the re-installation, it means the app somehow relies on device-public information for authentication, and is very likely to be vulnerable to our attack. As a final step, our system attempts to confirm the vulnerability by using the generic exploitation technique described above to transfer the identity used in the previous steps to an entirely new device.

Although some of the ideas and intuitions behind this work can be applied to any mobile operating system (and the corresponding apps), in this paper we focus on Android. This choice has been motivated by two main reasons: the fact that Android is currently the most widespread mobile operating system [36] and the ease of performing automatic analyses on Android apps.

We used this analysis system to vet 1,000 of the most popular applications from the Google Play Store, and 41 of them were correctly identified as vulnerable. Two of these vulnerable apps were WhatsApp and Viber, two of the most popular messaging apps, which are used by hundreds of millions of users. For both these apps, we discovered that it was sufficient to *steal* the content of a single file (and spoof the value of some device's identifiers) to fully hijack a user account. We reported our findings to the respective security teams, which quickly acknowledged the vulnerabilities. Among the apps flagged as vulnerable, our system also identified several popular games that allow a user to purchase virtual objects or currency: our automatically generated exploit was able to hijack these accounts as well. We conclude this work by proposing and implementing solutions for the identified class of vulnerabilities.

In summary, the contributions of this paper are as follows:

- We identify and study a new class of insecure authentication schemes that rely on device-public information.
- We demonstrate how it is possible to automatically exploit these vulnerable schemes by developing a generic "identity-transfer" attack, which is capable of stealing and replaying device-public information to hijack accounts.
- We explore the scope of the vulnerability in 1,000 popular apps from the Google Play Store using an automated dynamic analysis system, and identified 41 vulnerable apps, including Viber and WhatsApp.
- We propose and implement solutions to the identified problems.

2 AUTHENTICATION SCHEMES

Authentication in mobile applications can take on a variety of distinct forms, with differing security properties. The first, and most obvious, authentication scheme is the traditional username and password, in which the user is asked directly by the authenticating app for credentials. The app then sends these credentials to its backend server, which verifies their correctness. After this step, the server sends to the app a *token*, which is a shared secret string that can be used for authenticating all following interactions between the client and the server.

Another way to authenticate is to use third-party authentication services. This method removes the need to handle tedious per-app registrations. In Android, the AccountManager [14] offers a generic API that can be used to obtain an OAuth-like authentication token from third-party identity providers, such as Google or Facebook. The obtained token is presented with the app's requests to its backend, and can then be used by the backend to ask the third-party service for more information about the user.

Another popular scheme uses text messages (SMS) and the user's phone number as a form of authentication. In this scenario, the user would need to prove that they own a given phone number. As a part of the verification process, the user would typically enter the phone number manually. A code is then sent via SMS to the user, and is typically parsed automatically from the user's SMS inbox and verified. After this step, the phone number is used as the user's primary identity.

Lastly, some Android apps employ schemes in which distinguishing information about the device itself is used to bind a device to an account. This works under the implicit assumption that these identifiers are static and unique per device. To authenticate, the required identifiers are sent to the app's backend server, an authentication token is obtained, and such token is then sent along with future requests.

To reach the widest possible audience, many apps offer multiple authentication schemes, such as Facebook, Google, or regular user name and password authentication. While some of these methods may be securely implemented, the app may still be vulnerable if it allows users to use unsafe login-less methods that rely only on device-public information.

At their core, all these authentication schemes aim to obtain some sort of *token* that can be used to authenticate a user to the app's remote backend. However, if the authentication token can be obtained using information that another application on the same device can obtain, the authorization scheme is not safe. We also note that, even when apps employ schemes that are thought to be secure, they can still be vulnerable to account hijacking if they store authentication tokens in publicly accessible locations.

3 IDENTITY-TRANSFER ATTACK

Our key observation is that if an app only relies on device-public information to authenticate the user to its backend, it is possible for a malicious app to mine and leak all relevant information. If such a scheme is in use, an attacker can perform an identity-transfer attack, transferring information from the victim's device to the attacker's, so that the user's identity associated to a given app is effectively transferred.

3.1 Threat Model

In this paper, we assume that an attacker is able to lure the user into installing an attacker-controlled malicious application. This application requests all the needed permissions to acquire the device-public information being stolen, as outlined in Section 3.2. Moreover, we assume the operating system of the device to be uncompromised, and it thus constitutes a trusted computing base. Furthermore, we assume that the victim's device is *not rooted* (if it is, our attack does not take advantage of it), which means that an attacker cannot get *root* privileges. Therefore, the malicious app does not have access to app-private data, as the separation of the apps' private storage is strictly enforced by the OS.

3.2 *Device-Public* Information Sources

In this paper, we refer to *device-public* information as information that can be accessed from any app on the device that requests the permissions needed to obtain it. We will focus primarily on Android versions ranging from Android 4.4 to Android 7. When necessary, we will describe differences among different versions.

Here we will discuss the different sources of device-public information we have considered in our study, which are used by apps to identify users (also summarized in Table 1). Some of these identifiers are related to a specific hardware device, and cannot be changed by the user, whereas others can be changed after a "factory reset" of the device, or are linked to a Google Account. Google has recently attempted to hide some identifiers from apps to thwart tracking. That said, as of Android 7.1.1, we found that we are still able to access every identifier mentioned here, save for the Bluetooth MAC address.

Apps may need specific permissions to access some of these sources of device-public information, therefore a careful user may be able to notice that a malicious application is accessing some device-public information. However, while Android 6 introduced a feature alerting the user at the time some permissions are used, a malicious app can bypass this alert by lowering its own "Target SDK Version." By doing this, the old permission model, in which the user is not informed the moment an app uses a permission, is used.

ANDROID_ID. This is a device's unique ID number, set by Android upon a device's first boot or factory reset.

IMEI. The IMEI is a hardware identifier given to each piece of cellular equipment, including the baseband radios of mobile phones.

WiFi MAC address. Similar to the IMEI, MAC addresses are uniquely assigned to most conventional network hardware. The WiFi MAC address can be obtained by any app requesting the `ACCESS_WIFI_STATE`, using the API `WifiInfo.getMacAddress()`. In Android 6 (and later versions), the behavior of this API has been changed, so that it always returns the value `02:00:00:00:00:00`. However, we found that it is still possible to access this identifier using the `NetworkInterface.getHardwareAddress()` API.

Bluetooth MAC Address. The device's Bluetooth MAC address is a persistent hardware identifier that can be queried by using the API `BluetoothAdapter.getAddress()`. This API requires the `BLUETOOTH` permission. Starting from Android 6, the behavior of this API has been changed, so that it always returns the value `02:00:00:00:00:00`.

ADB serial number. The device's ADB Serial Number, which is used to identify devices on the Android Debug Bridge, is an identifier that persists across factory resets. It can be accessed by querying the `android.os.SystemProperties` object using the key `ro.serialno`.

Google account email. Many Android devices use Google account emails as a form of Single Sign-On, and the email address used can be easily obtained using the AccountManager API.

Google Service Framework ID. This ID is used to identify a user when accessing Google Service Framework applications.

Google Advertising ID. In an attempt to allow users to opt-out of mobile ad tracking campaigns, Google created a specific persistent identifier [15] to be used with advertising. It can be queried by any app (through the `AdvertisingIdClient` class in the Google Play Services), but, unlike the other identifiers, also freely reset by the user. Google's policy [18] states that all advertising must use exclusively this identifier for tracking ("in place of any other device identifiers for any advertising purposes"), although in practice it is often not used [35].

Phone number. We consider the phone number associated with the SIM card inserted in a device as device-public information. A specific API (*getLine1Number*, requiring the *READ_PHONE_STATE* permission) exists to retrieve this value, however the returned value is not always reliable, depending on the SIM Card manufacturer. Various workarounds do exist, including reading the call log, which requires the `READ_CALL_LOG` permission.

Received SMS Messages. Any app (with proper permissions) can request to be notified of the origin and content of new SMS messages.

Incoming Phone Calls. Apps can request to be notified about the basic data of incoming calls, including the caller's number. Additionally, Apps can also read the call history. Interestingly, phone calls can be used for authentication, by using part of the sender's phone number (which remote services can control) as a verification code.

SIM Card Serial Number. In devices where a SIM Card is present, apps can access this identifier, which is tied to the used SIM Card, by using the `getSimSerialNumber` API.

External Storage. Many Android devices today come with, or have the ability to add, some form of external storage, usually in the form of a larger Flash-based storage device or SD card. The precise behavior of external storage differs among Android versions and devices,

Table 1: Considered sources of *device-public* information.

Source	Required Permission	Survives factory reset	Linked to a Google Account	Linked to a SIM Card
ANDROID_ID	–	–	–	–
IMEI	READ_PHONE_STATE	✓	–	–
WiFi MAC address	ACCESS_WIFI_STATE	✓	–	–
Bluetooth MAC address	BLUETOOTH	✓	–	–
Google account email	GET_ACCOUNTS	–	✓	–
Google Service Framework ID	READ_GSERVICES	–	✓	–
Google Advertising ID	–	–	✓	–
Phone Number	READ_PHONE_STATE or READ_CALL_LOG	–	–	✓
Incoming Phone Calls	READ_CALL_LOG	–	–	✓
SIM Card Serial Number	READ_PHONE_STATE	–	–	✓
Received SMS Messages	RECEIVE_SMS or READ_SMS	–	–	✓
External Storage	READ_EXTERNAL_STORAGE	–	–	–

but, typically, any app can request the READ_EXTERNAL_STORAGE permission to access its contents. This gives the app access to the public areas of the external storage, shared by all apps.

Files stored in here are publicly accessible and some of them are not deleted upon app's uninstallation [23]. Therefore, as a usability feature for the users, some apps store authentication cookies in this location, so that the credentials *survive app's re-installation*. Unfortunately, while this may sound a reasonable practice, it is not secure. In fact, in this scenario, an attacker would be able to easily hijack the user's account by reading the files containing these authentication cookies and using their content to authenticate with the victim apps' remote backends.

3.3 Proof-of-Concept Attack Implementation

In simple terms, the attack consists of an app on the victim's device, which steals a set of device- and user-specific information, and exfiltrates it to the attacker. The attacker can then inject this information into their own device, so that apps behave seamlessly as if they are still on the victim's device. In particular, the attacker can use vulnerable apps as if authenticated as the victim on the victim's phone.

We implemented the "identity transfer" attack in two different components: the "ID Leaker," and the "ID Injector." The "ID Leaker" app, which could be thought of as a prototypical third-party malicious application, requires the Android permissions to access the SMS, device call notifications, external storage contents, and static device identifiers (refer to Table 1). The app then uses the well-documented Android APIs to access and leak the device-public data that constitutes the *user's identity*, and it sends it to the attacker's device. We note that the app's functionality could be easily hidden inside a seemingly legitimate app, and that it can run on completely unmodified devices without requiring any admin privileges (therefore on un-*rooted* devices). We also note that if an attacker aims at hijacking the account of a specific victim app, the "ID Leaker" only requires the permissions needed to access the specific device-public information used by the victim app for authentication purposes.

For the attacker's device, we created the "ID Injector," which takes data from the "ID Leaker" and injects them into an attacker-controlled device. We use the Xposed framework [1] (a tool for performing run-time patching of the Android framework) to easily hook the Android API methods used to query device-public information, and spoof their results to return the data leaked from the victim. The external storage's content is also transferred from the victim and copied into place. Without external information, there is no way for the app under analysis to tell that the data has been spoofed. Because of our usage of the Xposed framework, the attacker-controlled device (but not the victim's one) must be rooted to properly spoof the received identity.

4 VULNERABILITY DETECTION

In order to understand how widespread device-public authentication schemes are on Android, we created an automated system to locate vulnerable apps in the wild. This system could also be used by security researchers, software developers, and app market operators to automatically spot weakness in the authentication mechanisms used by the analyzed apps.

While the attack described in Section 3.3 is very effective against vulnerable apps, we cannot simply use it against all apps to build a detection system, for two main reasons. First, it is difficult to differentiate success of the attack from other application behaviors, as we have no baseline of the app's normal behavior to compare it to, and cannot link changes in this behavior to device-public information. Second, as we discuss in Section 2, "authentication" can be implemented in a variety of ways, making it difficult, if not impossible, to concretely define and locate authentication behaviors in a generalized way. We therefore cannot rely on any direct knowledge of the authentication itself to help understand when our exploit is having an effect.

To address these challenges, we developed an approach that aims at identifying authentication behaviors *indirectly*. We build our approach on the observation that an app behaves differently depending on whether or not it has already authenticated its user to a previously created account, and that this difference will be reflected in the app's user interface.

Thus, as a first step, the system executes the app, provides any requested device-public information to the app, and records the app's behaviors. These behaviors are in the form of a trace of different UI *states* (as detailed in Section 4.4). The aim of this initial execution is both to trigger the app's authentication or registration mechanism, as well as to get the server's backend to store some sort of state for the user, which can be observed in future traces.

Next, all app-private information for the app is deleted. We achieve this by uninstalling and re-installing the app. This operation deletes all app's files in private locations.

At this point, the app is executed again and, if the behavior is different from the one observed during the first run, it is possible that the app may be using device-public information (which could be both device's identifiers or publicly accessible files in the external storage) to authenticate the user. Typically, a difference may be observable because of the absence of a "login" screen due to already being authenticated, or the absence of an introductory "welcome" screen due to restoring the previously-saved user state, but more subtle UI modifications are possible.

As a last step, the system confirms the vulnerability, by transferring the device-public information to a different device, executing the app again, and comparing these behaviors with the previous ones. This transfer operation encompasses copying both publicly accessible files and device's identifiers.

In the remainder of this section, we will first discuss in detail the three steps of our analysis, as shown in Figure 1. We will then provide several technical details about the underlying dynamic analysis and the comparison of states and traces.

4.1 Step 1: Capturing Initial Behavior

First, we need to characterize the behavior of a given application when installed for the first time on an Android device. Our system functions primarily by collecting and comparing *traces*, consisting of an ordered list of UI *states* encountered during a given execution of the app. Details of how states and traces are collected and compared can be found in Section 4.4.

However, our system truly needs to characterize the "normal" behavior, not just merely record one execution. This is far from trivial, mainly due to the fact that dynamic analysis is hindered by non-deterministic behaviors present in apps, the OS, and network communications. To address this challenge, we first execute the analyzed application on multiple devices, collecting multiple traces. Recent work has shown that running the same app multiple times is, in Android, effective in reducing the effects of non-deterministic behaviors during dynamic analysis [9]. The collected app behaviors are then used to compute a so-called *Invariant*, representing the most common set of behaviors. Specifically, the Invariant set is computed as the set of all states that appear in *all* the collected traces.

4.2 Step 2: Vulnerability Detection

In this step, we delete all app-private data from the devices used in Step 1, collect new traces, and compare them with the Invariant. We accomplish clearing the app-private data by re-installing the app, which is known to remove all app-private information, including authentication tokens, cookies, databases and other private files.

After re-installation, the app is dynamically stimulated, and traces are collected in the same way as in Step 1. Then, we compare the new traces against the Invariant, looking for behavioral differences in the traces. These discrepancies are typically due to setup, registration, or login interfaces. Therefore, they are a strong signal the app was able to authenticate with the remote backend, only using information that *survived* the app's re-installation, which must therefore be device-public.

More precisely, if we determine that, during the execution of the analyzed app in this step, at least *one* state present in the Invariant has been skipped in *all* the collected traces, the app is flagged as potentially vulnerable.

4.3 Step 3: Exploit Verification

In Step 3, we verify if an app uses an insecure authentication scheme by actually attempting an identity-transfer attack against it.

To perform the attack, we transfer the device-public information stored in the devices used during Step 1 and Step 2 to new devices (which have not been used in the analysis of this app before), as explained in Section 3.3. Then, the same procedure used in Step 1 is used to obtain execution traces from the previously-unused devices.

These traces are then compared against the Invariant, as in Step 2. If we detect that at least *one* of the states skipped during Step 2 is also always skipped during Step 3, we conclude that the attack succeeded, and we flag the app as vulnerable.

4.4 Dynamic Analysis

In order to accomplish the above steps, we need to deterministically execute an application to trigger the authentication behavior, while minimizing behavioral divergences due to non-deterministic operating system or network behaviors. To this end, our system stimulates apps through their UIs, including buttons, text fields, and other interactive elements, as well as taking note of any incoming SMS and phone calls the used device may receive.

We rely on *uiautomator*[21], both to control the device and to obtain state information about the device itself. We control uiautomator from a normal PC by connecting it to the device using the Android Debug Bridge (ADB) and the uiautomator Python wrapper [8].

Possible actions are derived from the UI's content (button labels, text field descriptions, ...), and inserted into a priority queue. The priorities are arranged such that the most specific actions are performed first. The developed system also keeps track of previously touched UI elements, removing them from the priority list, so that every element is touched at most once. This is done to prevent the stimulation from entering an infinite-loop by continuously interacting with the same element.

The following is a list of the actions that our detection system can perform, in order of priority:

Fill text fields. Our system automatically fills some text fields. In particular, it first determines the type of information a text field is suppose to contain by (similar to [33]) checking labels and IDs associated to each text field against a pre-determined list of strings. Then, if a text field is determined as asking for a phone number, our system fills it with the device's phone number. Likewise, if a text field is determined as asking for a username, our system inserts a randomly generated one. It is important to note that no user-private

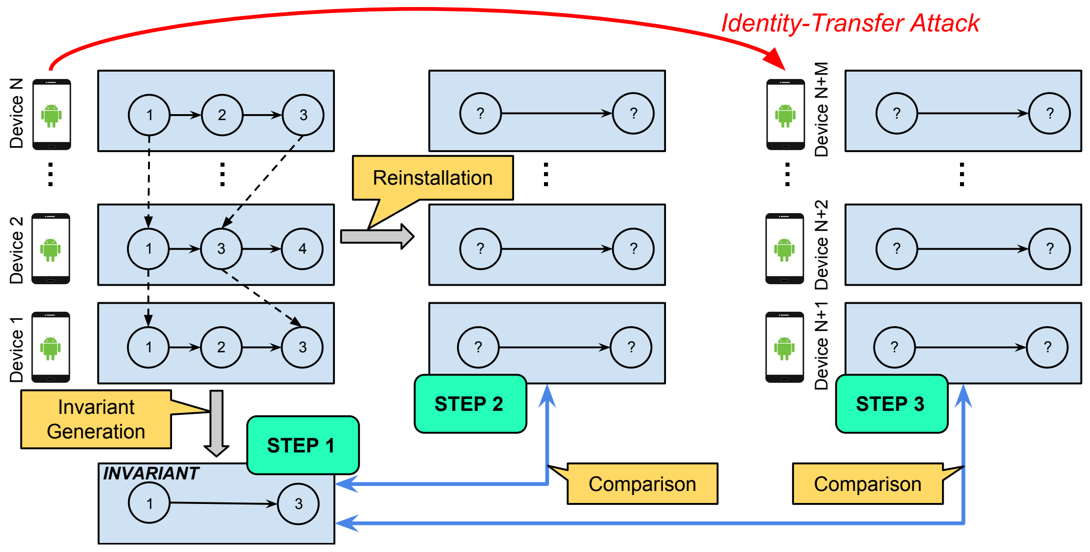


Figure 1: Overview of the developed dynamic analysis system.

information (e.g., a password) is inserted during this (or any other) step of the dynamic stimulation of an app.

Touch button. Our system interacts with UI elements that are "clickable." All clickable objects found are prioritized based on their type (e.g., buttons have higher priority than text fields) and their content; this allows us to, for example, touch an "OK" button before a "Cancel" button.

Pseudo-random touch events. If none of the previously mentioned actions can be performed on a state, our system will try to explore the app's behavior by simply randomly clicking on its UI. This situation usually happens, when the application uses custom UI elements, which do not export standard layout information to the OS.

In addition, if the analyzed app loses its focus (e.g., a window is opened in the system browser), we perform appropriate actions to make the analyzed app regain focus.

4.5 App States Extraction and Comparison

In order to make meaningful comparisons of different executions, we need a way to collect the current *state* of an app (e.g., which content it is showing to the user) at different times during our analysis and compare those states. The way in which states are encoded and compared needs to be sufficiently informative to capture significant behavioral changes, but also flexible enough to help ignore minor changes unrelated to the app's functionality. Specifically, the behavior of an app is encoded as a trace of states, which are then compared, looking for evidence of vulnerable authentication schemes.

State Extraction. Every five seconds, the system checks if the current device's UI is in a *steady state*. By this, we mean a situation in which the UI is likely not to change if no action is performed. If so, we record the current app's state (as better defined below) and we perform an action. Otherwise, the system waits up to a maximum threshold of 30 seconds. We employ this approach to perform actions and capturing states only when the effects of previous actions on the app's UI are completed. This also allows the sample rate of our system to be dynamic, and it helps to ensure that the captured states make the most sense when compared later. We use information provided by the Android video and input subsystems to know when an animation is being rendered (and therefore the current state is not steady). However, if we are unable to reach a steady state (e.g., the app uses OpenGL, or is otherwise constantly animating), we resort to an image-comparison approach.

Once the UI is steady, the system records a state, consisting of the following:

- The activity name (in Android, an Activity is a specific UI window)
- A hash of the simplified UI layout data
- A perceptual hash of the device's screen-shot

Hash of simplified UI layout data. To hash the information about the UI elements, we make important simplifications to the UI data, so that it is more easily comparable. In particular, from the layout tree describing the UI state, we remove the information about the location of the different layout's components and the text shown. These positioning or text differences are oftentimes due to intrinsically non-deterministic or rapidly changing UI elements, which are not relevant to our analysis.

Additionally, we take steps to avoid comparing deliberately dynamic content, especially advertising and web content. Advertising on Android is difficult to locate through explicit UI information. However, most mobile advertising is standardized by the International Advertising Bureau [6], which dictates specific pixel dimensions for ads, therefore we filter out elements from the simplified layout that have these sizes. Furthermore, we also filter out all WebView objects, since dynamic web content is typically a significant source of non-determinism. Lastly, we use the MD5 algorithm to condense the state information.

Hash of device's screen-shot. To hash the image acquired during the screen-shot, we use the algorithm called *average hash* provided by the *ImageHash* Python library [5]. This algorithm was chosen to provide meaningful fuzziness for images, abstracting away small, unimportant differences, such as constantly-animating UI elements. Specifically, this algorithm compresses every image in a 64-bit locality-sensitive fuzzy hash. The algorithm is designed so that images "appearing" as similar for humans are hashed to the same value, regardless of small graphical differences they may have.

State Comparison. We consider two states as equal if all the 3 components described above are equal. Moreover, when comparing states in traces collected during Step 2 against the Invariant, we also consider two states as equal if their image hashes only differ slightly (less than 10% of the bits composing the image hash). This threshold was determined empirically, by taking a subset of the apps from our dataset, and manually determining the optimal value.

Additionally, if during the dynamic stimulation of an app the device receives an SMS or a phone call, we add special states to the trace.

5 EXPERIMENTAL RESULTS

5.1 Datasets

We used the vulnerability detection system to probe apps from two different datasets:

***"Top Free"* dataset.** A dataset of 606 apps containing all the most popular available free Android apps. To generate this dataset, we first downloaded all the 539 available apps listed in the "Top Free" category on the Google Play market. Then we supplemented this set with other 67 applications starting from the ones that have the highest cumulative number of installations.

***"Top Grossing"* dataset.** A dataset containing the 394 most popular free apps in the "Top Grossing" category on the Google Play market (excluding the ones already present in the "Top Free" dataset). We chose this specific category because experimental results on the "Top Free" dataset and previous executions of our experiment revealed that apps from this category often allow users to authenticate using non-secure methods to ease their adoption.

Apps from both datasets were downloaded in January 2016. These datasets constitute a heterogeneous corpus of very popular applications both in terms of installations and developers' revenue. In total, we analyzed 1,000 distinct apps.

5.2 Experimental Setup

Our system is implemented using a series of Nexus 5 handsets tethered to a controlling PC. Specifically, we used 3 phones during the Invariant Generation and Vulnerability Detection phases (Step 1 and Step 2) and 3 additional phones during Step 3. All handsets run Google's official Android 4.4.4 images (the most adopted Android version at the time the apps were downloaded [20]).

During the collection of every trace of our analysis, we dynamically stimulated an app for two minutes. To ease the deployment of our infrastructure, devices' identifiers and phone numbers were modified during different runs of the experiment, effectively simulating the usage of a new device every time the experiment was run.

Averagely, the experiment needed 458 seconds per app to run Step 1 and Step 2 (including time necessary to reboot a device and install an application). For apps flagged as potentially vulnerable after these two steps, the analysis required, in average, additional 223 seconds per app to run Step 3 (including the time necessary to transfer the device's identity).

5.3 Results

Our system flagged 50 apps as vulnerable in our corpus of 1,000 distinct apps. Using manual analysis, we verified that 41 out of the 50 detected apps were actually vulnerable to the identity-transfer attack. Among these, two apps are Viber and WhatsApp, two very popular messaging apps with hundreds of millions of installations. We postpone the discussion of the vulnerabilities identified in these two apps to Section 6.1. Another group of 38 apps is composed by popular games, in which an attacker can perform an identity-transfer attack to steal the victim's virtual currency or objects. We will provide more details about them in Section 6.2.

Another detected app authenticates users by using standard SMS authentication. Specifically, this app identifies users with their phone number, by sending an authentication code to their phone number using an SMS, which is then automatically read by the app. If this code is stolen, an attacker can login and control all aspects of the user's account.

This security issue is different from the one found in the messaging apps described in Section 6.1, as it needs the attacker to steal the content of an SMS received by the victim. However, it still falls into our threat model, since the SMS content is device-public information.

In other 7 detected apps, we were able to transfer an identity with the exploit, but the identity was not protecting anything sensitive. Our system cannot, of course, detect which content is truly sensitive (e.g., related to a user's account) to a particular app, but the differences in the UI were present.

For example, in one app, the device-public information was used to track whether a user had accepted the application's End-User License Agreement (EULA), and, in another one, whether the user configured application preferences. In the other 4 apps, the backend uses this information to track whether a user has viewed certain full-screen "special offers" or advertisement from the app's developer.

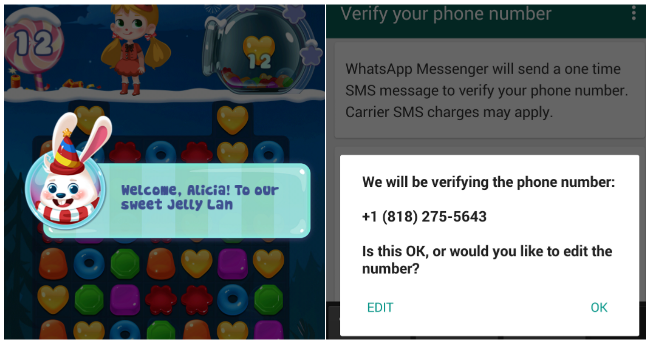
In addition, we found an app that, on first usage, shows a sign-in interface, since it assumes that the user does not already have an account. However, on subsequent re-installations, this app shows a username and password login interface, because it infers, using device-public information, that a returning user would already have an account. While not allowing any sort of account compromise, this information can still be leveraged by any other app to infer valuable information about the user, as explored in [10].

Finally, 2 additional apps were detected because of problems of our testing infrastructure, such as connectivity issues of the apps that caused the appearance of different graphical elements between the first installation and the subsequent ones. Subsequent runs of our experiment on these samples confirmed that these apps were detected for temporary problems. We consider these two apps as false positives.

6 CASE STUDIES

6.1 Messaging Applications

Two of the detected applications are the very popular messaging apps, WhatsApp and Viber, which allow users to send and receive



Verify your phone number

WhatsApp Messenger will send a one time SMS message to verify your phone number. Carrier SMS charges may apply.

We will be verifying the phone number:

+1 (818) 275-5643

Is this OK, or would you like to edit the number?

Figure 2: Examples of states recorded during the Invariant Generation phase (Step 1). Since these states were not present during Step 2 and Step 3 of our analysis, our system correctly classified these apps as vulnerable. In the left example, the skipped state shows to the user an introductory tutorial of the game. In the right one, the skipped state asks the user to confirm the entered phone number before validating it.

text messages, VOIP calls, and media. While the statistics on the Google Play market are not precise, WhatsApp is estimated to have more than 1 billion installations and Viber has more than 500 million.

Our vulnerability detection system flagged WhatsApp as vulnerable, since it detected, in the Vulnerability Detection and Exploit Verification phases, the absence of the "confirm your phone number" interface (shown in Figure 2), and the missing reception of an incoming SMS, used for authentication. Similarly, while analyzing Viber, the system detected this app as vulnerable because of the missing "Enter Your Name" dialog (shown only to new users) and the missing reception of a phone call whose part of the caller number is used as an authentication token.

Initially, we speculated that those apps were detected as vulnerable because they use the user's phone number, verified using received SMS or incoming phone calls, as their authentication method. This authentication method is common among popular messaging apps, and we consider it as vulnerable in our threat model. In fact, an attacker with a malicious application installed on the victim's device can pretend to own the victim's phone number and verify it by sending the authentication SMS received (or the caller phone number) from the victim's device to an attacker-controlled device.

However, further analysis surprisingly revealed that an even simpler attack is possible against these apps. The identity-transfer attack was successfully performed for both apps even though their backend did not send any SMS or phone call. This was possible because these apps used the content of a hidden file stored in the external storage to authenticate users upon re-installation.

Therefore, in the version of the apps we have analyzed, an attacker controlling an app on the victim's device could authenticate to the remote backend on behalf of the victim by:

(1) Copying the content of a specific file (stored in the external storage of the victim's device) to an attacker-controlled device.
(2) In case of WhatsApp, spoof the value of the Google email account. To achieve this, an attacker can use a malicious controlled app on the victim's device to query the AccountManager API and exfiltrate its value to an attacker-controlled device. Then, on the controlled device, the attacker can spoof it by using, for instance, the Xposed framework (as we implemented in the "ID Injector," explained in Section 3.3).
(3) Open the app.
(4) When asked to insert a phone number, specify the victim's one.

After these operations, the vulnerable app running on the attacker's device is automatically logged in as the victim, without even having the victim's device receiving an authentication text message. This exploit gives attackers full use of the victim's account, allowing them to send messages on the victim's behalf, and to receive all future messages sent to the victim.

Vendor Reaction. Upon discovery of these vulnerabilities, we contacted both vendors in August 2015. Both vulnerabilities were quickly acknowledged and, after working with the vendors, mitigations were deployed.

Specifically, both WhatsApp and Viber removed reliance on static identifiers and publicly-accessible files. However, they still rely on the content of a received text message (or, in case of Viber, the caller's number of an authentication phone call) for their primary means of authentication.

Interestingly, after our first notification, Viber was initially changed in a way that was ineffective against our attack. In particular, the file in the external storage remained, but just spoofing it was not enough. We discovered that Viber was changed to also check that other device's identifiers matched the ones used during a previous registration. However, as we discussed through the paper, an attacker could just query the values of the different device identifiers using an attacker-controlled app on the victim's device (see Section 3.2) and then spoof them on an attacker-controlled device, as implemented in the "ID Injector" (see Section 3.3). We note that this attack was working until Viber issued another update, in September 2017, removing reliance on the content of publicly-accessible files to perform authentication.

In addition, after our notification to the vendors (but likely independently from our disclosure), in 2016 both apps implemented new cryptographic measures limiting an attacker's ability to impersonate a user when an account is stolen (via ours or other attacks). In particular, both apps implemented an end-to-end encryption mechanism, based on the usage of a per-user key pair. This functionality allows users to authenticate and encrypt exchanged messages. For instance, suppose that two users A and B communicate together. This system encrypts the communication channel between A and B using their keys. Moreover, a user, for instance A, can check the value of B's public key and, in case B's public key changes, A would be notified (however, A and B could still communicate together). The same notification would be shown after the aforementioned attack is performed because the per-user key is stored in an *app-private* location, so it cannot be stolen and transferred to the attacker-controlled device.

6.2 Free-to-play Games

The other 38 identified vulnerable applications are games, in which an attacker can perform an identity-transfer attack to, steal the victim's virtual currency or in-game objects. In these apps, the system detected, for instance, the fact that graphical interfaces used to enter the user's name, or to show game tutorials and welcome messages were skipped during the Vulnerability Detection and Exploit Verification phases (see Figure 2). This indicated that the app authenticated with the remote backend and was able to obtain the user's state.

After an identity-transfer attack was performed, we noticed three different kinds of behaviors on the victim's device, when attacked while the victim is using them. Some of the apps show to the user a generic error message after the attack, such as "Connection Timeout." Others show a message informing the users that "another device" accessed their account. Finally, some of them do not show any information to the victim.

All these games offer in-app purchases, and the virtual currency used is derived from real money, using the Google's In-App billing API (IAB). For this reason, these vulnerabilities are particularly worrisome, since they can represent actual financial loss to the victims and the apps' developers. One surprising result was that the account transferred during the Exploit Verification phase can include virtual currency purchased through Google's In-App Billing API [19].

Notably, this is not an explicit attack against the In-App Billing API, as explored in previous work [30], but rather that its use in conjunction with the discovered vulnerabilities makes this data vulnerable as well. This is due to how the IAB API is implemented and how it is typically used by developers. In particular, even though the IAB mechanism offers to store information about a user's purchases (in a way which is secure under our threat model), it cannot be easily used as a store of the user's current account balance, since precise accounting is not possible.

Therefore, developers need to store the virtual currency balance differently, in the (potentially unsafe) app's backend. In case of applications vulnerable to identity-transfer attacks, this means the user's paid-for currency is as easy to steal as any other information in the user's account. This is particularly of concern, given the already-established trend in malware on other platforms targeting online game accounts [12].

7 PROPOSED DEFENSES

We propose two defenses against the attack studied in this paper: one aimed at creating secure device identifiers, and another aimed at safeguarding SMS-based authentication. A fully working prototype implementation of our defenses is publicly available [4], as an Xposed framework's module.

7.1 Securing the SMS Channel

Design. All installed apps on a device (that request the proper permission) can request to be notified of the content of incoming SMS messages, even when these messages are only intended for use by a particular app. As we shown, this behavior is particularly problematic when received SMS messages contain authentication codes destined for only a particular app.

Our proposed solution, similar to one discussed in [29], works by delivering authentication SMS only to the apps intended to receive them. Specifically, we propose a convention that authentication-related SMS should be pre-pended with the string `AUTHCODE: app_cert_fingerprint`, where `app_cert_fingerprint` is the fingerprint of the certificate used to sign the destination app. The OS would then route the message only to the main SMS reader app, and the app bearing the included fingerprint. This improves on Mulliner et al.'s previous solution by not requiring the OS to be notified ahead of time about how incoming messages should be routed.

For example, consider an app named "Foo Messaging" signed with a certificate whose fingerprint is `0d5af23c`. In this case, users enter their phone number into the app, which is sent to the app's backend. As a response, the app's backend sends an SMS message with the content `AUTHCODE: 0d5af238c Verification Code: 34782`. When received, the OS would then only notify the user's default SMS reader and "Foo Messaging" about the new message.

To improve usability, we propose that the default messaging app, by default, hides this routing information. Alternatively, the default messaging app could replace it with an indication of the app the message was delivered to. This functionality can be implemented without any modification to existing apps. However, it would require a small modification to the app's backends to prepend the app's fingerprint to the outgoing SMS.

We note that the recently-released Android version 8 introduces a new API called *createAppSpecificSmsToken*. This API "creates a single use app specific incoming SMS request for the calling package" [16]. When using this API, an app would first get a secret token from the operating system. Then the app's backend would send an SMS containing that specific token to the user's device and the SMS will be subsequently automatically routed by the OS to the correct app (through the token) and it will not be made readable by any other apps (or visible by the user).

While it may seem that this new feature mitigates the weakness of the usage of SMS to authenticate user, we argue that, on the contrary, it eases the attack we have described in this paper. In fact, while the usage of this API would stop an attacker from stealing and replaying authentication codes when the user attempts to authenticate, the attacker can just attempt their own authentication (simulating a user re-installing the app on the same device), at which point the SMS will be sent and routed to the attacker's app and it can thus be easily stolen. From the conceptual point of view, the attack works because the app's backend does not have enough information to determine whether the app receiving the SMS is the legitimate app or the attacker's app.

There are two additional aspects that make apps using this new API more vulnerable to the attacks presented in this paper. First, neither the user nor the legitimate app will notice the incoming SMS message (triggered by the attacker), since it will be routed only to the attacker's app. Second, the attacker's app does *not* need to require the *READ_SMS* permission when receiving messages using this API, thus making this malicious app stealthier.

Implementation. There are two ways an app can access SMS in Android: an app can ask to the operating system to be notified when a new message is received, or an app can access the list of received messages. Thus, to implement our defense, we modified both the Android *InboundSmsHandler* component, responsible to notify apps of incoming messages, and the *SmsProvider* component, mediating apps' accesses to received SMS. Globally, our modifications consist of approximately 100 LOC added to the original Android code. The added code introduces an average slowdown of 5.381ms every time an SMS is received and a slowdown of 2.064ms every time an app queries the operating system for received SMS. We consider both slowdowns as negligible, given the fact that, receiving a text message it is not a frequent event.

7.2 Secure Device IDs

Design. The most common and easily obtained device-public identifier is the ANDROID_ID, which is intended to be used to allow apps and their backends to differentiate Android devices.

In our defense we modify the API used to access the ANDROID_ID, so that it returns a Private Device ID (*PDID*) different for every app (more precisely, different for every app's signing certificate), instead of the original device-wide value. Specifically, the first time a device boots (or after a factory reset) a random Secret ID (*SID*) is generated. The Private Device ID is then derived from the Secret ID using the signing certificate included with each app, which uniquely identifies its developer. In this way, the semantics of the ANDROID_ID are preserved, apps from different developers cannot steal each other's identifiers, but no convenience is lost for a developer with multiple apps on the same device. Moreover, the *PDID* does not change after app's re-installation.

Specifically, the *PDID* is computed as follows: `HMAC(SID, caller_app_cert_fingerprint)` where: `caller_app_cert_fingerprint` is the certificate fingerprint of the app calling the API and `HMAC` is a cryptographically secure keyed-hash message authentication code (e.g., HMAC-SHA256) in which `SID` is used as "key" and `caller_app_cert_fingerprint` as "message."

The security of this method is bolstered by the fact that

1. Upon installation, Android verifies that an app has been correctly signed.
2. The operating system can securely identify the caller of a framework API [17].
3. No API is provided to get the value of *SID*.

For these reasons, as long as the developer's private key remains uncompromised, the privacy of the PDID to an app is maintained.

We implemented this modification as complete transparent replacement of the current API used to get the ANDROID_ID. In this way this defense could be deployed without requiring code changes to existing apps. This will necessarily interfere with advertising libraries, which seek to use the ANDROID_ID to track the usage of multiple apps on the same device. However, as explained in Section 3.2, the only identifier that advertisement libraries are supposed to use to track users is the Google Advertisement ID. A possible alternative implementation would be providing the PDID to apps trough a separate API.

It is interesting to note that, concurrently (and independently) to the development of this work, Google changed the behavior of the ANDROID_ID to follow our proposed modification. Although the implementation details differ, the functionality achieved by this change is the same. This modification is available starting from Android version 8 [22].

Implementation. We implemented this defense by modifying the Android *SettingsProvider*, the operating system component responsible to deliver the ANDROID_ID value to the running apps. Our modifications consist of approximately 70 LOC added to the original Android code. The added code introduces an average slowdown of 1.497ms when the API to get the ANDRODID_ID is called. The standard Android API caches this value after the first time an app access it, thus we consider this slowdown as negligible.

8 LIMITATIONS AND FUTURE WORK

While we were able to find a surprising number of vulnerable apps, our system is far from perfect. There are a few conceptual ways in which our system might miss vulnerable apps. The most important one is the inability to influence a change in the user's state stored by the app's backend server. For instance, in some games, the dynamic analysis system would need to effectively play the game and, for example, score points or spend virtual currency.

An important source of error in dynamic analysis is the non-determinism inherent in today's operating systems and apps. Some apps explicitly perform random behaviors, which our Invariant Generation step attempts to remove, but it is by no means perfect. For example, if the non-deterministic behaviors are time-dependent or influenced by network delays, they may produce the same result during the Invariant generation, but not during the other phases. Some previous work has been done to try to have fully deterministic replay of actions (see Section 9.2), but the current state-of-the-art does not handle all the source of indeterminism that our system has to deal with.

One other source of future improvement is in the number of identifiers spoofed and transferred by our system. We used a large set of known identifiers for which we could locate Android APIs, but apps could conceivably invent their own identifiers based on collections of obscure system properties, or implement other means of fingerprinting devices. Finding all possible ways this can happen is an open problem.

We would also like to explore the use of network traffic as part of the Invariant generation, to attempt to more precisely determine when a backend is saving and retrieving user state. In particular, a way to assist with the network traffic analysis, as well as other data sources and sinks, is to use a taint-tracking-based analysis system, such as TaintDroid [11]. Unfortunately, we have noticed that many identifiers are sent to the app's backend, even if they are not directly used for authentication purposes, which represents a significant source of noise for this kind of analysis. This is potentially done to aid in gathering metrics about apps, or to aid in advertising.

Finally, an interesting future work would be to study if the authentication problems we have identified in this paper also affect applications running in other mobile operating systems.

9 RELATED WORK

9.1 Authentication-Related Vulnerabilities in Android

Zhang et al. explore the issue of *uninstallation residue*, where uninstalling an app does not properly clean up all data and references in the system, creating an opportunity for an attacker to elevate their privileges and steal sensitive information [41]. While the detection system we developed uses uninstallation of apps to trigger a removal of app-private data, the vulnerabilities targeted in this work and ours are very different. In fact our system detects authentication vulnerabilities that involve the usage of both device identifiers and files in public locations.

The paper entitled Mayhem in the Push Clouds [26] explores the related issue of push messaging platforms, which are commonly used by apps to communicate asynchronously with their backends. The paper found that authentication tokens for these services are often handled insecurely, especially when sent using Android's Intents. Our work focuses on what can happen when these tokens are created using device-public information, or are in turn stored as device-public information.

Chen et al., in OAuth Demystified for Mobile Application Developers [7], explore the usage of OAuth-like mechanisms for authentication. Their work included a manual study of 149 applications using OAuth, and found that 89 used it incorrectly. Moreover, Wang et al. investigate OAuth misuse from a different angle in their paper Explicating SDKs [39], which examines the way applications use authentication and authorization SDKs from companies such as Facebook and Microsoft. A related, but distinct, vulnerability can occur in improperly implemented services using OAuth, which implicitly trust responses from identity providers without verification [40]. In contrast to web platforms, in mobile apps, these responses originate from the user being authenticated, meaning they can be tampered with, allowing an attacker to authenticate as the victim without their private credentials.

As we do in our work, Liu et al. [27] studies how apps unsafely use public storage. However, their work focuses on how the public storage is used to store sensitive information (such as the user's contact list), whereas we focus on how the public storage is used to store information that, together with device's identifiers, is used to authenticate with remote backends. Similarly, a work by Bai et al. [2] studies how a specific class of apps (backup tools) leaks information in publicly accessible files in the external storage. However, the apps studied by this work require either *root* or *shell* privileges, not obtainable by normal apps under the threat model we considered (non-compromised OS).

Zuo et al. [42] developed a system, named AutoForge, to automatically find authentication vulnerabilities revolving around user-private information. Specifically, they focused on detecting apps' backends vulnerable to password brute-forcing, leaked username and password probing, and Facebook access-token hijacking. We consider this work as complementary to ours. In fact, their work studies how apps' backend behaves when probed with supposedly-secret data, such as usernames, passwords, and Facebook authentication tokens. Conversely, our work focuses on an entire class of authentication schemes that do not rely on this supposedly-secret data.

A paper by Mulliner et al. [29] looks directly at the issue of SMS-based one-time passwords. They explore various layers of the problem, including issues of wireless interception, and smartphone Trojans, similar to our "ID Leaker." While their work was primarily motivated by the use of mobile Transaction Authorization Numbers in the banking industry, this same idea has also spread to most areas of the mobile world that require verification of a user's phone number, as we explore in our study. SMS authentication is further investigated by Schrittwieser et al. [34]. In this work, authors manually analyze a selection of messaging apps, verifying their security properties and finding different vulnerabilities in them.

The intrinsic weakness of SMS-based authentication has been recently pointed out. For instance, security researchers have shown that, by exploiting vulnerabilities of the SS7 network used by telecom company to route phone calls and SMS, it is possible for an attacker to intercept SMS and steal authentication codes [38]. Moreover, state-sponsored attackers could easily interfere with local telecom companies to intercept these authentication messages [37]. For this reason, the latest security guidelines advise against the use of SMS as a two-factor authentication method [31]. It is important to notice that the vulnerabilities we found in popular messaging apps (see Section 6.1) were *not* due to the usage of SMS content for authentication, but a consequence of the usage of public accessible files and device's identifiers to authenticate their users.

9.2 Android Dynamic Analysis

Rastogi et al. proposed AppsPlayground [33], a dynamic analysis framework aimed at maximizing code coverage of dynamic analysis. Other works with similar goals are Brahmastra by Bhoraskar et al. [3] and DynoDroid by Machiry et al. [28]. Our vulnerability detection system utilizes similar techniques to interact with apps, however, our goal is different, since we do not aim to maximize code coverage but to trigger the authentication mechanisms in a deterministic manner.

Different tools have been proposed to deterministically record and playback input events on Android: RERAN [13], MOSAIC [24], MobiPlay [32], and VALERA [25]. The usage of these tools as a part of our dynamic-analysis based vulnerability detection system constitutes an interesting future direction, since they could remove non-deterministic behaviors which currently hinder our analysis. However, in their current state, these tools do not completely solve the problem. For instance, RERAN, MOSAIC, and MobiPlay do not deterministically replay network traffic, whereas in our experiments we determined that most of the non-deterministic behaviors are due to discrepancies or delays in the network traffic between an app and its backend. The approach of VALERA is able to deal with network traffic, however, it cannot replay user's interaction in case of applications using customized rendering, like many of the ones we detected as vulnerable (see Section 6.2). Unfortunately, most of the apps we correctly detected as vulnerable actually use customized interfaces and heavily interact with online backends.

10 CONCLUSION

In this paper, we explored the real-world vulnerabilities of apps that authenticate their users using device-public information. Some app authors appear to make the assumption that this information is somehow hard to obtain or spoof.

To disprove this, first we developed an "identity-transfer" attack that can be automatically applied to any apps relying on device-public information to authenticate its users. Then, we developed a system, based on dynamic analysis, that infers information about the apps' backend states to locate insecure authentication mechanisms, and perform our attack against them. After analyzing 1,000 popular apps from the Google Play market, we found 41 that were vulnerable to our generic identity-transfer attack, including two major messaging apps used with hundreds of millions of installations.

Finally we proposed and implemented solutions to the identified problems, requiring minimal modifications to the Android operating system and no modifications to the existing apps.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Elias Athanasopoulos, for his help and comments.

This material is based upon work supported by NSF under Award No. CNS-1408632 and by Google's "Security, Privacy and Anti-Abuse Award." Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the NSF or Google. This material is also based on research sponsored by DARPA under agreement number FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] Xposed Installer (framework). http://repo.xposed.info. (2015).

[2] Guangdong Bai, Jun Sun, Jianliang Wu, Quanqi Ye, Li Li, Jin Song Dong, and Shanqing Guo. 2015. All Your Sessions Are Belong to Us: Investigating Authenticator Leakage through Backup Channels on Android. In *Proceedings of the 20th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS).*

[3] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security).*

[4] Antonio Bianchi. Implementation of the proposed defense mechanisms. https://github.com/ucsb-seclab/android_device_public. (2017).

[5] Johannes Buchner. Image Hash library. https://github.com/JohannesBuchner/imagehash. (2015).

[6] International Advertising Bureau. Ad Unit Guidelines. http://www.iab.net/guidelines/508676/508767/ad_unit. (2015).

[7] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. OAuth Demystified for Mobile Application Developers. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[8] Xiao Cong. uiautomator. https://github.com/xiaocong/uiautomator. (2015).

[9] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proceedings of the 24th Network & Distributed System Security Symposium (NDSS).*

[10] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *Proceedings of the 23rd Network & Distributed System Security Symposium (NDSS).*

[11] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. 2010. 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).*

[12] Chun Feng. 2008. Playing with shadows – exposing the black market for online game password theft. In *Virus Bulletin Conference.*

[13] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proceedings of the 35th International Conference on Software Engineering (ICSE).*

[14] Google. AccountManager. https://developer.android.com/reference/android/accounts/AccountManager.html. (2016).

[15] Google. Advertising ID. https://support.google.com/googleplay/android-developer/answer/6048248?hl=en. (2016).

[16] Google. Android Documentation: SmsManager. https://developer.android.com/reference/android/telephony/SmsManager.html. (2016).

[17] Google. Binder. https://developer.android.com/reference/android/os/Binder.html#getCallingUid(). (2016).

[18] Google. Google Play Developer Program Policies. https://play.google.com/about/developer-content-policy.html. (2016).

[19] Google. Implementing In-app Billing. https://developer.android.com/google/play/billing/billing_integrate.html. (2016).

[20] Google. Platform Versions. https://web.archive.org/web/20160131030000/https://developer.android.com/about/dashboards/index.html. (2016).

[21] Google. Testing Support Library. https://developer.android.com/tools/help/uiautomator/. (2016).

[22] Google. Android O Behavior Changes. https://developer.android.com/preview/behavior-changes.html#privacy-all. (2017).

[23] Google. Using the External Storage. https://developer.android.com/guide/topics/data/data-storage.html#filesExternal. (2017).

[24] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).*

[25] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet Lightweight Record-and-Replay for Android. In *ACM SIGPLAN Notices*, Vol. 50.

[26] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. 2014. Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[27] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. 2015. An Empirical Study on Android for Saving Non-shared Data on Public Storage. In *Proceedings of the IFIP International Information Security Conference.*

[28] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE).*

[29] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. 2013. SMS-Based One-Time Passwords: Attacks and Defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).*

[30] Collin Mulliner, William Robertson, and Engin Kirda. 2014. VirtualSwindle: An Automated Attack Against In-App Billing on Android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (Asia CCS).*

[31] NIST. Digital Authentication Guideline. https://pages.nist.gov/800-63-3/sp800-63b.html. (2016).

[32] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: a Remote Execution based Record-and-Replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE).*

[33] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY).*

[34] Sebastian Schrittwieser, Peter Frühwirt, Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Markus Huber, and Edgar R Weippl. 2012. Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *Proceedings of the 19th Network & Distributed System Security Symposium (NDSS).*

[35] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What Mobile Ads Know About Mobile Users. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS).*

[36] StatCounter. Operating System Market Share Worldwide – May 2017. http://gs.statcounter.com/os-market-share#monthly-201705-201705-bar. (2017).

[37] Telegram. Keep Calm and Send Telegrams! https://telegram.org/blog/15million-reuters. (2016).

[38] Thomas Fox-Brewster. Watch As Hackers Hijack WhatsApp Accounts Via Critical Telecoms Flaws. http://www.forbes.com/sites/thomasbrewster/2016/06/01/whatsapp-telegram-ss7-hacks/#43e6fc1c745e. (2016).

[39] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. 2013. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization.. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security).*

[40] Ronghai Yang, Wing Cheong Lau, and Tianyu Liu. Signing into One Billion Mobile App Accounts Effortlessly with OAuth2.0. BlackHat Europe. (2016).

[41] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. 2016. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *Proceedings of the 23rd Network & Distributed System Security Symposium (NDSS).*

[42] Chaoshun Zuo, Wubing Wang, Rui Wang, and Zhiqiang Lin. 2016. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS).*