Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services

Tongxin Li[1], Xiaoyong Zhou[2,4], Luyi Xing[2], Yeonjoon Lee[2], Muhammad Naveed[3], XiaoFeng Wang[2] and Xinhui Han[1]

[1]Peking University
[2]Indiana University Bloomington
[3]University of Illinois at Urbana-Champaign
[4]Samsung Research America

ABSTRACT

Push messaging is among the most important mobile-cloud services, offering critical supports to a wide spectrum of mobile apps. This service needs to coordinate complicated interactions between developer servers and their apps in a large scale, making it error prone. With its importance, little has been done, however, to understand the security risks of the service. In this paper, we report the first security analysis on those push-messaging services, which reveals the pervasiveness of subtle yet significant security flaws in them, affecting billions of mobile users. Through even the most reputable services like Google Cloud Messaging (GCM) and Amazon Device Messaging (ADM), the adversary running carefully-crafted exploits can steal sensitive messages from a target device, stealthily install or uninstall any apps on it, remotely lock out its legitimate user or even completely wipe out her data. This is made possible by the vulnerabilities in those services' protection of device-to-cloud interactions and the communication between their clients and subscriber apps on the same devices. Our study further brings to light questionable practices in those services, including weak cloud-side access control and extensive use of `PendingIntent`, as well as the impacts of the problems, which cause popular apps or system services like Android Device Manager, Facebook, Google+, Skype, PayPal etc. to leak out sensitive user data or unwittingly act on the adversary's command. To mitigate this threat, we developed a technique that helps the app developers establish end-to-end protection of the communication with their apps, over the vulnerable messaging services they use.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Invasive software*

General Terms

Design, Experimentation, Security

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660302.

Keywords

mobile push-messaging services; Android security; mobile cloud security; security analysis; end-to-end protection

1. INTRODUCTION

Push messaging (aka. push notifications, cloud to device messaging, etc.) is a type of cloud services that help developers of mobile applications (*apps* for short) deliver data to their apps running on their customers' mobile devices. Such services are offered by almost all major cloud providers. Prominent examples include Google Cloud Messaging (GCM), Apple Push Notification Service and Amazon Device Messaging (ADM). Subscribers of the services are popular apps (Android Device Manager, Facebook, Skype, Netflix, Oracle, ABC news, etc.) on billions of mobile devices. Through those services, the developer can conveniently push her notifications, messages and even commands to her apps, avoiding continuous probes initiated from the app side, which are more resource-consuming. With the pervasiveness of the services and the increasingly important information exchanged through them, which includes not only private notifications (e.g., Facebook messages) but also security-critical commands (e.g., wiping out user data on a stolen phone [2]), little has been done, however, to understand their security and privacy implications.

Security risks in push messaging. Recently it has been reported that the GCM cloud was abused by cybercriminals to coordinate and control their malware [5]. What is less clear is whether such push services themselves are vulnerable to infiltration attempts. Those attempts, once succeeded, can have devastating consequences, potentially allowing an unauthorized party to steal sensitive user data or even completely control the apps subscribing those services. More specifically, a push messaging service is typically provided through a *connection* server in the cloud, which delivers data from a 3rd-party app server to its apps running on a large number of mobile devices. This delivery process often goes through a service client (e.g., the GCM client) and the software development kits (SDKs) integrated into the apps. If the interactions that happen between either the client and the server or the SDK and the client have not been well thought-out, a local (on the same device) or remote adversary could intercept the messages pushed through the service, or even impersonate the app server to command the apps to operate on his behalf. To find out whether such security risks are indeed present in today's mobile clouds, which are very complicated systems, an in-depth analysis is needed to understand their operations across the cloud and mobile devices, and the security implications of those operations. With the importance of the problem, such a security study has never been done before.

Our findings. In our research, we conducted the first security analysis on the mobile-cloud services for Android devices. Our findings are astonishing: almost all popular services contain security-critical weaknesses never known before. Even though some of them are subtle, hiding deeply inside the systems, they can all be exploited in practice through carefully-crafted attacks, which often cause serious consequences. Particularly, we found that an unauthorized party can circumvent security checks of GCM to preempt the registration ID of the victim's app. As a result, the victim's sensitive information such as her Facebook messages will all be pushed to the adversary's device. Such information leaks also happen to the most popular push-messaging cloud in China with over 600 million subscribers, though the service works in a way different from GCM. Throughout the paper, we use a pseudonym *mpCloud* to refer to this push-messaging service, on the request for anonymity from the company.

Also serious are the problems with the client-side components of those push-messaging services, including service clients (e.g., the GCM client) and SDKs. Our research reveals critical security weaknesses in such components within almost *all* popular services for Android devices, which are mainly caused by the exposure of the `PendingIntent` object through intent broadcast or service invocation with regard to a designated *action*, which an unauthorized party can claim to be able to handle. With this exposure, the adversary who knows how to use the `PendingIntent` can launch a series of carefully-crafted attacks to push fake messages to subscriber apps and control their communication with the clouds, a problem present in all leading services such as GCM, ADM and UrbanAirship.

The consequences of those attacks are dire, through which not only can the adversary gain access to such critical user data as bank account balance, family members' home addresses, etc., but he can even command Android Device Manager to lock out the device user and wipe out all her data, and the Google Play service to install and uninstall any app without the user's consent. In our research, we analyzed 63 popular apps utilizing messaging services and discovered the pervasiveness of those security-critical problems. Examples of those apps include Facebook, Google Plus, Chase bank and PayPal. We reported all the flaws discovered in our research to Google, Amazon, and other service providers, which all acknowledged the importance of our findings. Particularly, one of them (UrbanAirship [16]) informed us its plan to formally cite us in its release notes and another organization (Section 3.2) has expressed gratitude for our willingness to keep them anonymous. Also, Android security team has formally acknowledged us on their official website [3]. Video demos of the exploits are posted online [15] and most of the flaws have already been fixed based on our reports.

End-to-end protection. Our security analysis reveals the pervasiveness of subtle yet serious security weaknesses across major push-messaging services, which calls into question the reliability of those services. Based upon such understanding, new techniques need to be developed to make these systems also secure to use, even in the presence of a variety of unexpected security flaws an app developer has no control of. As a first step, we designed an end-to-end protection mechanism through which the developer and her apps can establish a secure channel on top of those unreliable services to authenticate each other and safeguard the confidentiality and integrity of the information they exchange. This mechanism was further implemented into a pair of SDKs, one for the app server and the other wrapping the SDKs for popular messaging services. Using the mechanism, the app developer can conveniently build up this end-to-end protection, with only minor adjustments of her code. Our evaluation study further demonstrates that this mechanism incurs only a small overhead and effectively addresses all the problems we discovered.

Contributions. The scientific contributions of the paper are:

• *Security analysis of push messaging services*. Understanding security weaknesses inside practical systems is the starting point for any system security research. For an emerging system as important and complicated as mobile cloud, the problems there can be new, subtle and deep, requiring a well-conceived study to identify and determine their security implications. Our research makes a first step towards this end, which reveals serious vulnerabilities within push-messaging systems, one of the most important mobile cloud services. We performed a security analysis on such services offered by leading cloud providers, uncovered new types of security problems within its client-cloud communication and client-side components, and further investigated the impacts of our findings, which are shown to be significant. This work helps better understand what can go wrong inside mobile clouds and what can be done to make it right, raising the awareness of this new security challenge and laying the foundation for the follow-up research on this direction.

• *New protection technique*. We made the first attempt to secure end-to-end push messaging communication in the presence of weaknesses within the underlying mobile-cloud services. Our approach prevents an unauthorized party from impersonating the app server to push messages to apps or intercepting messages. It can be conveniently integrated into existing apps by their developers.

Roadmap. The rest of the paper is organized as follows: Section 2 gives the background information of our study; Section 3 and Section 4 elaborates the security-critical flaws discovered in both client-cloud communications and client-side components of push-messaging services; Section 5 reports a measurement study on the problems we found; Section 6 presents our end-to-end protection; Section 7 discusses the limitations of our study; Section 8 surveys related prior research and Section 9 concludes the paper.

2. BACKGROUND

2.1 Mobile Cloud Messaging

As discussed before, cloud-based push messaging services have been extensively deployed, supporting almost all the mobile devices in the market. Those services are offered by mobile-platform providers (Google, Apple, etc.), cloud service providers (e.g., Amazon) or messaging service providers (e.g., mpCloud in Section 4). With this diversity, those services share a similar architecture and also closely resemble each other in their operations, which we describe below.

Service infrastructure. Figure 1 illustrates the infrastructure of a typical push-messaging service. At the center of this infrastructure are the connection servers running on the cloud and the client-side components installed on individual users' mobile devices. The connection server receives messages from the app server and pushes them to the service client on each device. It also receives upstream messages from the client. In our research, we call such interactions *cloud-device link*. The client is a persistent process, which monitors messages from the cloud and passes them to subscriber apps, and wakes up the apps when necessary. This client program can be either a standalone service app, e.g., the GCM service app, the ADM Client and the iOS Notification Center, or part of the SDKs from the service provider. Its interactions with the app are called *on-device link* in our study.

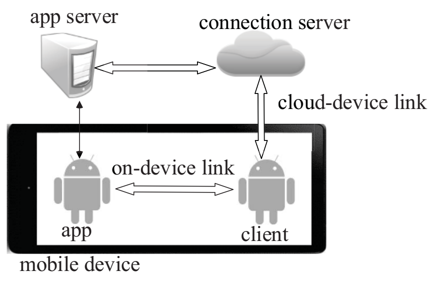

Figure 1: Push Messaging Infrastructure

To utilize a push-messaging service, an app needs to incorporate its SDK to talk to the service client. On the cloud side, the developer of the app needs to run an *app server* to communicate with the connection server, requesting its service to push notifications, commands and other messages to the apps on her customers' devices. In addition to this mobile-cloud communication channel, the app server can also get in contact with its apps out-of-band, receiving messages directly from those apps.

Service operations. The operations of a push-messaging service are performed at two stages, *registration* and *message delivery*. An app is supposed to register with the connection server before it can receive messages from its developer server. This happens when the app sends the client (an SDK or a standalone service) a registration request, which the client forwards to the cloud. Such a request includes the app's identification information like its application ID (generated from the app's package name), an Android ID (for uniquely identifying the host device) and a sender ID (a unique number assigned to the app server). Using such information, the connection server creates a random registration ID for identifying the app. This ID is critical for the operations of the whole push-messaging service, which is used to locate the right app recipient of the messages pushed through the cloud. It is given to the registering app and sent to the app server directly by the app out-of-band, using a secure channel (e.g., SSL).

To push a message to its apps, the app server works with a connection server, providing it registration IDs of the apps and the message. The connection server then stores and enqueues all the messages it receives, and pushes each of them to the corresponding target device as soon as it gets online. On the device, the message is first delivered to the service client, which then hands it over to the subscriber app.

On-device communication. When the service client is a standalone program, such as the GCM app, its interactions with the subscriber app need to go through Inter-Process Communication (IPC) on the mobile device. On Android, serving this purpose is the *intent* mechanism. An intent is a message that describes the operation to be performed by its recipient. It can be used to launch a user interface component (e.g., through the API `startActivity`), communicate with a background service (e.g., through `startService`) or broadcast to a group of registered receivers (e.g., through `send Broadcast`). To use this mechanism, the sender can either explicitly specify the package name of the recipient or just identifies an *action* so that any app registering with the action (through its intent filter) can get it. In the latter case, the intent broadcasted is passed to the recipients in the order of the priorities they set. These recipients can also make their broadcast receivers private or protect them with a permission to ensure that only authorized senders can access them.

A problem for this intent mechanism is that the recipient is not informed of the sender's identity, which needs to be authenticated when an app sends a request to the service client. This problem is resolved by push-messaging services using `PendingIntent`. `PendingIntent` is a token that allows the recipient to perform a set of pre-defined operations on an intent constructed by its sender, *using the identity and permissions of the sender*. More specifically, the sender app can create a `PendingIntent` object and pass it to another app. The app then acquires the capability to perform the operation (including broadcast, `startActivity` or `startService`) as if it was the sender. Particularly, to find out where the message comes from, the recipient can call the object's method `getTargetPackage` or `getCreatorPackage` to get the sender's package name. Note that this information is provided by the operating system and therefore cannot be forged.

2.2 Security Hazards

The security guarantee of a push-messaging service depends on that of the individual links on its complicated communication chain, which include the interactions between client-side components, the device and the cloud, the cloud and the app server, and the server and the app. A crack discovered on any of those links can have a serious consequence, leaking sensitive user data or exposing critical system capabilities. Here we discuss such security hazards.

What can go wrong. The most security-critical information within mobile-cloud services is each app's registration ID. Once the ID has been exposed and also tied to a wrong party, the consequences can be devastating: more specifically, when the ID is bound to an attack device, whatever is supposed to be sent to the app will be delivered to that device, disclosing all sensitive user data pushed through this channel; when the ID is bound to an attack server, the adversary gains the privilege to push commands to the app as its developer server. Therefore, the registration ID needs to be protected on all the links across the whole communication chain, to prevent it from being exposed and manipulated when the app talks to the service client, the client to the connection server, the connection server to the app server and the app server to the app. Also, the connection servers need to check each registration request to ensure that the registration ID of an app is built on the right parameters from right parties. The problem is the complexity of the communication involved in the service, which makes such protection challenging.

The content of messages can also be exposed in other ways, particularly during the on-device communication. Although those messages are often encrypted when they are exchanged between the cloud and the device, they are transmitted in plaintext on the device, between the service client and the subscriber app. Such on-device communication, once going through insecure channels such as broadcasting or service invocation using an action, can be intercepted by an unauthorized app. Even when the communication does not directly carry any sensitive data, it often involves the token (i.e., `PendingIntent`) for identifying a message sender, which could be exploited by a knowledgeable adversary to impersonate the sender or acquire the capability to access its protected resources.

Our study. To understand whether those security hazards indeed pose a credible threat to push-messaging services, we analyzed those services' security protection, focusing on the cloud-device link (Section 3) and the on-device link (Section 4) as the first step. Specifically, for the device-cloud link, we ran Dex2jar [6], JD-GUI [9] and Baksmali [4] to study the code of different messaging services' clients and their SDKs, in an attempt to evaluate their program logic related to the app registration. For this purpose, we also need to look into the logic on the cloud side, particularly the security checks it performs. Without its code, all we can do is a black-box analysis, in which we adjusted the parameters of registration requests to study responses from the server.

Unlike the cloud-service link, the on-device link only involves the service components (the service client and the SDK) on the same mobile devices, whose code and communication are more accessible. What has been inspected in our research is the security implications of the resources (`PendingIntent` in particular) exposed through their interactions. We further analyzed 63 popular apps to understand the damages to their users when their push-messaging services are exploited. The results of this study are elaborated in Section 3, 4 and 5.

Adversary model. We assume the presence of a malicious app on the victim's mobile device. The app does not have system privilege and therefore needs to ask for a set of permissions from the users such as `READ_GSERVICES`, `GET_ACCOUNTS` and `INTERNET`. Although all such permissions are at the dangerous level, they are also extensively requested by popular apps such as Ebay, Expedia, Facebook and Whatsapp. Therefore, we believe that claiming them by the malicious app will not arouse much suspicion. Further, we consider an adversary who has the resources to set up his own app servers with the cloud and run his devices to collect the victim's information.

3. EXPLOITING MESSAGING SERVICES

In this section, we report our security analysis of the cloud-device links on major push-messaging service providers.

3.1 Google Cloud Messaging

Google Cloud Messaging is one of the most popular messaging services, which has been subscribed by billions of Android devices. According to Google, the GCM service pushes about 17 billion messages every day [8]. GCM works in a way as described in Section 2.1, which requires an app to first register with the service before receiving messages. In our study, we analyzed both the registration and the message-delivery stages on an HTC One X phone (serving as the attack device) and a Nexus 7 tablet (as the victim's target device), using mitmproxy [12] and mallory [11] to proxy the SSL communication between the attack device and the GCM connection server.

Flaws. To register with GCM, an app uses its app server's sender ID and a `PendingIntent` (see Section 4.1) to invoke the `register` method, one of `GoogleCloudMessaging` APIs under the Google Play service `com.google.android.gms` on the device where the app is running. The service then makes an HTTPS request to a connection server to obtain a registration ID for the app. This request contains the device's Android ID and its device-token for the authentication purpose, and the package name for identifying the app, as illustrated in Figure 2. Also, the Android ID shows up in three fields `Authorization`, `X-GOOG.USER_AID` and `device`. In our research, we performed a black-box analysis on the connection server by adjusting the content of those parameters through our proxy. What we found is that the content of `device` needs to match the Android ID within the `Authorization` field to let the request go through. However, `X-GOOG.USER_AID` does not need to be consistent with the other two fields, which turns out to be a security-critical logic flaw that can be exploited by an unauthorized party to collect sensitive user messages.

Specifically, what the adversary wants to do is to generate a registration request on behalf of the app on the victim's device (the target) using his own attack device, a critical step for intercepting user messages, which we elaborate later. The catch here is the device-token (within the `Authorization` field in Figure 2), a secret that is bound to the device's Android ID and automatically appended to the request by the Google service during the communication with GCM. This prevents a malicious app on the target device from getting it. Without knowing the token, the adversary can only use his own Android ID and device-token to fill `Authorization`. However, due to the missing consistency check between this field and `X-GOOG.USER_AID` on the connection server, the adversary can still set the latter to the Android ID of the target device (which can be obtained by the malicious app on the target device with the permission `READ_GSERVICES`). Most importantly, we found that the connection server solely relies on `X-GOOG.USER_AID` (not the Android IDs in the other two fields) to create a new registration ID or retrieve the record of an existing one.

Authorization: AidLogin **android-id:device-token**
app: package-name
User-Agent: Android-GCM/1.3
content-length: 171
content-type: application/x-www-form-urlencoded
Host: android.clients.google.com
Connection: Keep-Alive
Accept-Encoding: gzip
URLEncoded form
X-GOOG.USER_AID: android-id
app: package-name
sender: sender-ID
cert: application's certificate
device: android-id
app_ver: application's version

Figure 2: Registration Request

Exploits and consequences. In our study, we implemented an exploit, which includes an app with the permission `READ_GSERVICES` for collecting the Android ID from the target device. Using this information, our approach automatically generates a registration request from our attack device by replacing the content of `X-GOOG.USER_AID` with the target's Android ID. When this registration request is made *before* the target app does, we found that the adversary can hijack this registration ID, binding it to his attack device. Specifically, when an app registers with GCM the first time, the connection server establishes a binding relation between its registration ID and its current device (identified through its Android ID in the field of `device` or `Authorization`), but apparently indexes the registration with the content of `X-GOOG.USER_AID`. If the same app later registers again, the server retrieves the app's registration ID using its Android ID and returns it to the app. Since the content of `X-GOOG.USER_AID` can be different from that in `device` and `Authorization`, the adversary who registers using the target's Android ID in `X-GOOG.USER_AID` (which indexes the registration with the target device) and the attack device's ID in the other two fields (which binds the registration to the attack device) will cause the target app to receive a registration ID tied to the attack device's Android ID when the app is trying to register with GCM. This will have all the messages for the target app pushed to the attack device.

We built an end-to-end attack in which the attack device preempted the Facebook app on the target device in registering with the Facebook server. As a result, the Facebook app, which registered later, got a registration ID tied to the attack device and unwittingly sent this ID to its app server (see Section 2.1). After that, all

Facebook messages and notifications, including those with sensitive content, went to the attack device.

When the adversary makes a registration request *after* the target app registers (on the victim's device), the GCM connection server responds by retrieving the app's registration ID using the `X-GOOG.USER_AID` (filled with the victim's Android ID) on the request and sends it to the attack device. In this way, the adversary can steal the target app's registration ID. Typically, this ID needs to be used by the app server with an authorized sender ID to push messages to the app. However, we found that for GCM, this policy was not in place until very recently, which enabled us to inject messages to the app using its registration ID (see [15]). Although this problem is no longer there, later we show that the target's registration can be bound to an attack server through exploiting a weakness in the on-device link (Section 4.2).

The video demos for the above attacks (binding a registration ID to a wrong device and injecting messages through the ID) are posted on a private website [15]. We reported all the findings to the Android security team and the GCM team. So far, all these problems have been fixed based on our report.

3.2 mpCloud Messaging Services

mpCloud (*the pseudonym we use on the company's request*) is the largest push-messaging service provider in China, which serves 600 million users, including Chinese Internet giants like Sina [13]. Particularly it offers a critical support for Sina Weibo [14], an extremely popular Chinese version of Twitter. The designers of the service clearly took security seriously: for example, on its cloud, registration IDs (called *client ID*) can only be used by authorized app servers to communicate with their corresponding apps.

Flaws. However, our analysis of this push-messaging service shows that it also has serious security flaws, which can lead to exposure of sensitive user messages. Specifically, when reinstalling a service-subscribing app after uninstalling it on the same device, we found that the app always gets the same client ID. A close inspection of the code of the mpCloud SDK reveals that such a client ID is actually generated deterministically on the device, based upon a data file (`/sdcard/libs/package_name.db`) the service deposits on the device's SD card, and its International Mobile Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI) and MAC address. Such information is all accessible to a malicious app running on the victim's device with proper permissions (`READ_EXTERNAL_STORAGE` for reading from the SD card, `READ_PHONE_STATE` for collecting IMEI/IMSI and `ACCESS_WIFI_STATE` for getting the MAC address). With the information exposed, the adversary can come up with the right client ID on his own device.

When it comes to message delivery, what happens within mpCloud is that the device makes a socket connection to the cloud server, providing it the device's identity information to get messages, as other messaging services do. However, different from GCM, in which the server will check both the registration ID of an app and the Android ID/device-token of a device before pushing messages to the app, mpCloud solely relies on the client ID to identify a device and the app running on it. As a result, the exposure of the client ID renders both the victim's target device and the attack device equally entitled to receive push messages from the cloud server. In our research, we found that such messages will be randomly pushed to one of these two devices.

Exploits and consequences. We built an end-to-end attack on Sina Weibo, one of the most important customers of mpCloud. Sina actually modified the mpCloud SDK, changing the directory path of the data file (for generating the client ID) to `/sdcard/sina/weibo/libs_backup/com.sina.weibo.db`. In our research, we ran an attack app to gather the data from this file and other information (IMEI/IMSI/ MAC) to derive the client ID. Using the client ID, our attack device successfully received the push messages (i.e., tweets) from Sina Weibo, which were supposed to be delivered to the target device. This vulnerability has been reported to mpCloud.

4. VULNERABLE ON-DEVICE COMMUNICATIONS

In addition to the client-cloud link, we further analyzed the on-device link, which involves the communication between a push-messaging client and an app subscribing the service on the same device. For this purpose, we studied the client-side code of the most popular push-messaging services for Android, including GCM, Amazon Device Messaging (ADM), UrbanAirship [16] and mpCloud (pseudonym for the largest Chinese push-cloud service provider). Those services support a vast majority of Android devices, delivering messages to billions of users. However, all of their client components were found to include serious security flaws. The problems here start with the use of intent broadcast or `startService` with regard to an action. The most intriguing part, however, is how to use the content exposed through those channels, which itself may not be sensitive (e.g., a message for starting the process of requesting a registration ID) but comes with the capability allowing a knowledgeable adversary to wreak havoc through complicated exploits. Following we describe the problems found in individual services.

4.1 Exploiting Upstream Messaging

GCM client components. GCM client components include the Android service apps `com.google.android.gfs`, `com.google.android.gms` and the SDK `google-play-service.jar` integrated into the service-subscribing app. The communication between the SDK and those services all goes through intent. Such communication includes the request for getting registration ID and the messages pushed to the app through the service. The intent serving those purposes carries the recipient's package name, which ensures that only the right party can get it. However, an exception is made when it comes to *upstream messaging* [7], a new feature that allows an app to push messages to its app server. The intent created for this purpose is delivered through the broadcast channel without specifying the target package, which distributes the message to whoever declares a right *receiver*. This treatment is meant for flexibility, enabling the app to use any service app capable of handling its message. Also, an upstream message often does not contain sensitive data: for example, it is used to notify the Google server when a new account is opened on a device. However, such an exposure gives away the `PendingIntent` object embedded in the intent, which has serious consequences.

More specifically, the `PendingIntent` objects are embedded in the intents the GCM SDK (within the subscriber app) delivers to the Google service apps to inform the latter of the sender's identity, which is not given when a normal intent is passed from one app to another [32]. The problem is that the `PendingIntent` object here can do much more than just the sender identification, and therefore can be abused once it is exposed to an unauthorized party.

Exploits. In our research, we implemented an attack app that declared a broadcast receiver for the action `com.google.android.gcm.intent.SEND` (specified within its manifest file), with a higher priority (1000) in receiving messages than the legitimate Google service app. This attack app was able to intercept any upstream message a GCM-subscribing app sent to its app server, and also prevented the Google services from getting them. This not only violated the confidentiality of the communication, but more importantly, enabled the attack app to obtain the `PendingIntent` embedded in the upstream messages. With this token in hand, we were able to perform the following message-injection attack:

Each app subscribing the GCM service needs to follow Google's instructions [7] to declare an intent receiver to get push messages. Such a receiver is protected by the signature permission `com.google.android.c2dm.permission.SEND` and thus only accessible to the Google services and the app itself, in accordance with the action defined for the receiver (e.g., `com.google.android.c2dm.intent.RECEIVE`). For example, the Facebook app claims a receiver `com.facebook.push.c2dm.C2DMBroadcastReceiver`. The trouble here is that once the `PendingIntent` is exposed, the attack app becomes able to execute the operation specified by the object with the sender app's permissions. Specifically, for GCM, the `PendingIntent` its SDK builds always includes a blank intent with a broadcast operation. Therefore, the attack app that intercepts this object can fill the content of the intent with the target app's package name and action, and then ask the OS to broadcast this intent. Given this new intent is viewed as coming from the target app itself, it is allowed to be delivered to the target's receiver. As a result, the attack app can now push arbitrary messages to the target. In our study, we successfully launched this attack against Android Device Manager and other apps.

Also interestingly, we found that the exposed `PendingIntent` can be used to impersonate the target app to send requests to Android services. For example, an attack app can generate a registration request in the name of the target app to register it again with GCM using its `PendingIntent` object. The `gms` service receiving this intent checks the sender's package name using the `PendingIntent`, and will be convinced that the intent indeed comes from the target app. As discussed in Section 3.1, when a registered app attempts to register again, the connection server just returns to it the existing registration ID. The attack app can then include in the forged registration intent an instance of `android.os.Messenger`, through which it receives the registration ID returned from the cloud. Although the registration ID alone may not be enough for injecting messages to the app remotely, the attack app's capability to impersonate the target needs serious attention.

Consequences. We found that the `gms` itself uses upstream messaging to notify Google whenever a Google account is added or removed on a mobile device. This happens through broadcasting an intent to the `SEND` action of `gms` itself. In our research, our attack app intercepted this intent and the `PendingIntent` object it carried, through which the attack app was able to directly push messages to `gms`. Since Android Device Manager is also running in the same process, our app commanded it to wreak havoc. Particularly, we show that Android's anti-theft protection can be used against the phone user: the Device Manager under our control ringed the phone, locked the legitimate user out of the system and even erased the user's data on the phone. Furthermore, since `gms` bears the same signature as other system services, the Google Play service in particular, our attack app was able to execute the `PendingIntent` from `gms` to talk to the service's GCM receiver protected by the signature permission. Through the service, the attack app silently installed new apps and uninstalled existing ones without the user's consent. The video demos of the attacks are on a private website [15]. Again, we reported the problems to Google, which has fixed the issues.

4.2 Capability Exposure in Registration

Apps using C2DM/GCM template code. Starting from Android Cloud to Device Messaging (C2DM), Google has provided template code to help the app developers conveniently integrate its push-messaging service into their apps. The code segment for initiating the registration process is presented in Figure 3. This code is further recommended to the GCM users after C2DM was deprecated. As a result, many popular apps do not integrate the GCM SDK `google-play-service.jar`, and instead directly use this set of code to dispatch an intent to `gms` for invoking (`startService`) its registration service. Prominent examples include Facebook and UrbanAirship [16], a popular intermediary service that enables developers to send messages to different mobile devices (Android, iOS, Windows, etc.). Interestingly, different from what happens through the SDK, which sets the recipient of the intent to the package name of `gms`, the code here only specifies the action `com.google.android.c2dm.intent.REGISTER`, so whoever defines this action is entitled to receive the intent message. Apparently, this treatment can be useful for the transition from C2DM to GCM: the new GCM service app only needs to declare the action to work with the apps designed for C2DM. However, just like the broadcast channel, this type of service invocations can be easily abused: an attack app only needs to specify the registration action and a high priority (above that of `gms`) to get the intent and also prevent `gms` from receiving it.

```
Intent registrationIntent = new Intent("com.
      google.android.c2dm.intent.REGISTER");
registrationIntent.putExtra("app",
      PendingIntent.getBroadcast(this, 0, new Intent(), 0));
registrationIntent.putExtra("sender", senderID);
startService(registrationIntent);
```

Figure 3: Template Code

Again, though the registration intent itself does not carry any confidential information, its exposure leaks out the `PendingIntent` object. Since this happens during the registration stage, the adversary getting the token can cause an even bigger trouble. Specifically, the target app's registration ID can also be stolen by the attack app through sending to `gms` a new registration intent with the target's `PendingIntent`. In this way, our app essentially acts as a *man-in-the-middle* (MitM) that receives the registration ID from `gms` and hands it over to the target app through the broadcast operation included in its `PendingIntent`, as described before. More seriously, our MitM can fabricate the registration ID given to the target app, binding it to an attack device or an attack server. Following we elaborate these two attacks, using the Facebook app as an example:

• *Device misbinding*. In this attack, the adversary requests from GCM a registration ID for the Facebook app running on the attack device, without sending the ID to the Facebook server. Instead, this ID is transmitted to the attack app on the victim's device (the target). During the registration of the Facebook app on the target device, the attack app intercepts its request and runs the stolen `PendingIntent` to inject the adversary's registration ID to the victim's app in the same way as the aforementioned attack (Section 4.1), except that the attack app's intent is aimed at the action `com.google.android.c2dm.intent.REGISTER` this time. Upon receiving the intent, the victim's Facebook app considers the registration ID received as a legitimate one from the connection server, and therefore unwittingly uploads it to the Facebook server to link it to the victim's account. As a result, all the victim's Facebook messages will go to the attack device. We implemented this attack and successfully executed it on our Nexus 7 tablet.

• *Server misbinding*. As discussed before, the GCM and other clouds (e.g., ADM, mpCloud, etc.) only allow the app server with an authorized sender ID to push messages to the apps with related registration IDs. Here we show that even this protection can be completely circumvented once the `PendingIntent` object is exposed during the target app's registration. The trick here is to generate a registration ID bound to the attack server. Specifically, the attack app first intercepts the registration request from the victim's Facebook app and serves as an MitM. It can then set the sender ID within the registration request it generates to that of the attack server. Based on this request, the registration ID the GCM cloud generates becomes linked to the attack server. After injecting this registration ID to the victim's Facebook app, the adversary can push messages to the app remotely, as the app's registration ID is tied to the attack server. This attack was implemented and evaluated in our research, which was found to work effectively on popular apps, like Facebook messenger.

This problem affects many popular apps according to our study (Section 5). We reported our findings to Google, Facebook and UrbanAirship. Google further notified other parties. In recognition of the importance of the findings, UrbanAirship planned to formally acknowledge us in their release notes and Facebook awarded us $2000.

Amazon Device Messaging. Amazon Device Messaging (ADM) is a push-messaging service Amazon uses to support its popular Kindle Fire device, which accounts for about one third of the Android tablet market according to a recent report [10]. ADM has a registration process similar to that of GCM: a service-subscribing app sends a registration request to the Amazon service app (`com.amazon.device.messaging`) on the same device, which contacts the ADM cloud to get a registration ID for the app. Like GCM, this ID is tied to a specific app server: only this server is allowed to push messages to the app with the ID.

In our research, we analyzed the code of the Amazon SDK `com.amazon.device.messaging.ADM`. It turns out that the SDK behaves just like the Facebook app during the app registration phase: it issues a start-service intent to any recipient that declares an action `com.amazon.device.messaging.intent.REGISTER`; also the intent contains a `PendingIntent` object for the ADM service app to identify the sender of the intent. Therefore, an attack app with the action and a higher priority in receiving messages can intercept the intent and steal the `PendingIntent`. Consequently, the adversary can launch the aforementioned device-misbinding attack to link the target app's registration ID to an attack device, causing all messages for the the app to go to the attack device.

The server-misbinding attack, however, does not work on ADM, because the ADM service app directly gets the sender ID (of the app server) from the target app, not from the registration intent as `gms` does. This thwarts the attempts to inject messages to the target app remotely, given that a registration ID inconsistent with a server's sender ID cannot be used to push messages to the app from the server through the Amazon cloud. On the other hand, the local-injection attack is still effective: our attack app was able to deliver messages to the target app through executing the broadcast operation on its `PendingIntent` (Section 4.1). We reported the flaws to Amazon and are helping them to fix those issues.

4.3 Other Exploits

In addition to the problems with `PendingIntent`, our research on the on-device link also reveals other weaknesses. Specifically, we analyzed the code of the mpCloud SDK and found that the process it runs to receive messages from the cloud actually delivers them to the target app through an intent broadcast targeting at an action. This allows an attack app to easily intercept those messages. Also, instead of declaring the receiver of the intent statically within the app's manifest file, the SDK actually dynamically defines the receiver during the app's runtime, which makes the receiver public. And the app does not take extra measures to guard it either. As a result, the receiver has been made public and any app can send messages to it. We implemented an end-to-end attack that successfully exploited those flaws. Our findings were reported to mpCloud.

Type	Categories	# of tested apps	# of leak
GCM	SOCIAL	17/25 (68%)	13/17 (76%)
GCM	COMMUNICATION	9/13(69%)	3/9 (33%)
GCM	FINANCE	4/11 (36%)	3/4 (75%)
GCM	SHOPPING	5/9 (56%)	2/5 (40%)
GCM	PRODUCTIVITY	3/5 (60%)	1/3 (33%)
GCM	HEALTH & FITNESS	3/3 (100%)	1/3 (33%)
GCM	ENTERTAINMENT	2/17 (12%)	1/2 (50%)
GCM	BUSINESS	1/3 (33%)	1/1 (100%)
GCM	OTHER	9/30 (30%)	0/9 (0%)
mpCloud	SOCIAL	3/3 (100%)	3/3 (100%)
UA	SHOPPING	1/1 (100%)	1/1 (100%)
UA	LIFESTYLE	2/2 (100%)	0/2 (0%)
ADM	SOCIAL	2/2 (100%)	0/2 (0%)
ADM	EDUCATION	1/1 (100%)	1/1 (100%)
ADM	COMMUNICATION	1/1 (100%)	1/1 (100%)
TOTAL		63/126(50%)	28/63(44%)

Table 1: Summary of Measurement Study

5. MEASUREMENT STUDY ON VULNERABLE APPS

To understand the impacts of the problems discovered in our research, we analyzed popular Android apps to study their individual vulnerabilities and the consequences once those flaws are exploited. Here we report what we found.

App collection. We downloaded 599 top free apps from the Google Play store. From their manifest files, 255 were found to use GCM. Also we collected from the Google Play store 3 apps subscribing mpCloud and 4 apps subscribing ADM. For the 255 GCM-subscribing apps, we picked out those among the top 125 and also within the categories of SOCIAL, COMMUNICATION, SHOPPING, FINANCE and HEALTH, and further manually added a few well-known apps with a large number of downloads (more than 10 million) but did not make to the top 125 list, such as Google plus, YouTube, Dropbox, and 3 apps subscribing UrbanAirship (UA). All together, 63 apps, including those using GCM, ADM and mpCloud, were inspected in our study.

Table 1 summarizes the number of apps studied and their results in each category, and Table 3 provides examples of vulnerable apps, including their ranking/download information. Note that for the apps in some categories, particularly FINANCE, we need an account with related organizations to study their functionalities. Under this constraint, we had to only work on those accessible to us. For example, in the FINANCE category, we checked the apps from Chase bank, Bank of America, PayPal and Google Wallet.

Vulnerabilities. For each of those 63 apps, we installed it on our devices and monitored their operations using the ADB LogCat tool [1], which recorded all the messages the app got from its cloud. This logging was done on rooted phones, through setting the "log.tag.GTalkService" property of its Google Play service (which makes the service log all the GCM messages it gets) and modifying related APIs for the ADM service. Note that all such messages

Category	App name	Rank	# of download	leaked contents
COMMUNICATION	Facebook Messenger	3	100,000,000+	messages
COMMUNICATION	Glide - Video Texting	55	5,000,000+	chat messages
SOCIAL	Instagram	4	100,000,000+	comment
SOCIAL	Twitter	15	100,000,000+	direct messages
SOCIAL	Vine	40	10,000,000+	follow, comment
SOCIAL	textPlus Free Text + Calls	98	10,000,000+	chat messages
SOCIAL	LinkedIn	125	10,000,000+	invitation, messages
SOCIAL	Google+	331	500,000,000+	comment, plus - encoded by base64
FINANCE	Chase Mobile	78	10,000,000+	alert messages : minimum balance, incoming/outgoing wire transfer, payment, debit card transaction, ATM withdrawal, external transfer, direct/online/ATM deposit, notify hold, overdraft protection, etcetera.
FINANCE	Bank of America	82	10,000,000+	alert messages : available balance, debit card/ATM deduction, low balance threshold, money transfer, online bill payment, personal information update, irregular debit card activity, etcetera.
FINANCE	PayPal	141	10,000,000+	send money, buy something, receive money, get a request for money
HEALTH & FITNESS	Calorie Counter - MyFitnessPal	112	10,000,000+	comments, like
SHOPPING	eBay	18	50,000,000+	watched items end alert(item name, price, bidding information), winning bid, shipment, messages
ENTERTAINMENT	Find My Phone	535	5,000,000+	messages from family, check-in info, help request, invitation, home address, location of family members

Table 3: Examples of Vulnerable Apps

Category	num	Vulnerability(Section)	vulnerable apps
GCM/UA	56	GCM cloud-device link(3.1)	56(100%)
GCM/UA	56	GCM C2DM/GCM code template(4.2)	10(18%)
mpCloud	3	mpCloud cloud-device link(3.2)	3(100)%
mpCloud	3	mpCloud on-device link(4.3)	3(100)%
ADM	4	ADM on-device link(4.3)	4(100)%

Table 2: Summary of Security Weakness

were confirmed to be exposed to the unauthorized app through the vulnerabilities we found. This step just serves the purpose of extracting their content from the app. We also ran our attack app during its registration, trying to find out whether it includes the vulnerable C2DM/GCM code template. Among them, 10 apps turned out to use the template, whose registration intents were intercepted by our apps. The rest are all vulnerable to the exploit on their cloud-device link (Section 3.1) or other threats. The security weaknesses of those apps are summarized in Table 2.

Consequences. We further manually analyzed the logs of those apps to understand the type of the information the adversary can learn (through binding the victim's registration ID to the attack device) or fake (through pushing falsified messages). It turned out that among those apps, only Sina Weibo encrypts messages with a symmetric key hard-coded in its app, which is very unsafe given that all its apps share the same key, and others simply use plaintext (sometimes, with the Base64 encoding). From the logs, we were able to monitor the chats from Facebook Messenger, posts from Google plus, etc. Also, apps in the FINANCIAL category leaks alert messages including sensitive personal financial data such as the user's minimum/current balance, debit card usage, payment, wire transfer activity etc. All alert messages are in plaintext with the exact amount, receiver, last four digits of account and credit card number and others. Moreover, Find My Phone, an app for locating and tracking phones of family members, exposes the exact home address of every member in the victim's family. Also leaked out from the app are help requests and other private messages. All together, 28 extremely popular apps were confirmed to expose sensitive user data and most of them also allow the adversary to inject security-critical information. More details of some examples are provided in Table 3. The situations with other apps are less clear, due to the challenges in fully understanding their semantics and triggering all their operations related to push messaging. Nevertheless, the preliminary findings already point to the seriousness of the problems we found. In the absence of proper protection, those push-messaging services, including the most leading ones, are not up to the task of safeguarding app users' sensitive information.

6. PROTECTING CLOUD MESSAGING

Our findings point to the pervasiveness of security-critical flaws in push-messaging services, which are very likely to be just a tip of the iceberg. Given the complexity of these services and their diversity, app developers often end up with little confidence in the safety of their communication with their customers across different cloud platforms (GCM, ADM, etc.). To improve this situation, we developed Secomp (secure cloud-based message pushing), a simple mechanism that establishes end-to-end protection across different push-messaging channels. In this section, we describe our design and implementation of the technique, and also our evaluation study.

6.1 End-to-End Protection

The design of Secomp is based upon the observation that apps involving security-critical operations or sensitive data (e.g., the Facebook app, Chase Mobile app, etc.) almost always need to authenticate themselves or their users to their servers, which typically goes through HTTPS, allowing the user to log into her account using password or other credentials. So, the idea here is to leverage this existing secure channel (HTTPS) and authentication mechanism (e.g., password-based login) to establish a secret key between the app and its server, which is later used to protect the confidentiality and integrity of push-messaging communication with an authenticated encryption scheme.

Figure 4 elaborates how Secomp works, where DIRECTCHANNEL can be an HTTPS connection an app uses to log its user into the app server and get the secret key K, $\mathsf{K}_{\text{VERIFY}}$ can be the developer's public verification key embedded within the app, and the $\text{ENC}^{\mathsf{Auth}}$ scheme used for "in-band" communication (through the push cloud) can be AES in Galois/Counter Mode. This simple protection mechanism was implemented in our research into a pair of SDKs that the developer can incorporate into her server-side code and app, respectively. Whenever the app user is authenticated (using her password or a single-sign-on scheme) to the server through an out-of-band HTTPS connection (a direct connection between the app and its server, which is already there for login), the server immediately generates K and sends it to the user through the connection. For all the messages exchanged through the push cloud, K is always used together with $\text{ENC}^{\mathsf{Auth}}$ to let the app and the server

1. Preliminaries:
   - Let ($\textsc{Keygen}$, $\textsc{Enc}^{\mathsf{Auth}}$, $\textsc{Dec}^{\mathsf{Auth}}$) be an **authenticated encryption scheme**, such that $\textsc{Enc}^{\mathsf{Auth}}$ and $\textsc{Dec}^{\mathsf{Auth}}$ guarantees both confidentiality and integrity.
   - Let ($\textsc{Keygen}'$, $\textsc{Sign}$, $\textsc{Verify}$) be an unforgeable digital signature scheme.
   - The push message channel $\textsc{PushChannel}$ between the app server and the app through push messaging service is unprotected as shown in this paper.
   - The direct communication channel $\textsc{DirectChannel}$ between the app and the app-server is authenticated and encrypted.
2. $\textsc{Secomp.Keygen}$:
   - App server generates a key K using $\textsc{Keygen}$ algorithm.
   - App server generates a signing key $\mathsf{K}_{\textsc{Sign}}$ and the corresponding verification key $\mathsf{K}_{\textsc{Verify}}$ using $\textsc{Keygen}'$. (This operation is done once for all users).
3. $\textsc{Secomp.ShareKey}$:
   - App server sends the app specific key K and a global signature verification key $\mathsf{K}_{\textsc{Verify}}$ to the app using $\textsc{DirectChannel}$.
4. $\textsc{Secomp.RegisterApp}$:
   - App registers itself with a push-messaging service such as GCM and receivers a registration ID RegID.
   - App sends this registration ID RegID to the app server using $\textsc{DirectChannel}$.
   - App server records the tuple (app's identity, RegID, K).
5. $\textsc{Secomp.PushMessage}$ ($\mathsf{message}$), where $\mathsf{message}$ is the push message that needs to be send to the app:
   - App server encrypts the $\mathsf{message}$ with key K using $\textsc{Enc}^{\mathsf{Auth}}$ and obtains $\textsc{Enc}^{\mathsf{Auth}}_{\mathsf{K}}(\mathsf{message})$.
   - App server sends $\textsc{Enc}^{\mathsf{Auth}}_{\mathsf{K}}(\mathsf{message})$ to the app using $\textsc{PushChannel}$.
6. $\mathsf{message} \leftarrow \textsc{Secomp.Receive}$:
   - App receives the ciphertext $\textsc{Enc}^{\mathsf{Auth}}_{\mathsf{K}}(\mathsf{message})$.
   - App decrypts the message using K to obtain $\mathsf{message} = \textsc{Dec}^{\mathsf{Auth}}_{\mathsf{K}}(\textsc{Enc}^{\mathsf{Auth}}_{\mathsf{K}}(\mathsf{message}))$.
7. $\textsc{Secomp.Broadcast}$ ($\mathsf{message}$):
   - App-server signs the message $\mathsf{message}$ using key $\mathsf{K}_{\textsc{Sign}}$ to get the signature $\textsc{Sign}_{\mathsf{K}_{\textsc{Sign}}}(\mathsf{message})$.
   - App-server now broadcasts the message as before but includes the signature $\textsc{Sign}_{\mathsf{K}_{\textsc{Sign}}}(\mathsf{message})$ with the message.
8. $\textsc{Secomp.BroadcastReceive}$:
   - App receives these message $\mathsf{message}$ and a signature $\textsc{Sign}_{\mathsf{K}_{\textsc{Sign}}}(\mathsf{message})$.
   - App verifies the signature with verification key $\mathsf{K}_{\textsc{Verify}}$ using $\textsc{Verify}$ algorithm $\textsc{Verify}_{\mathsf{K}_{\textsc{Verify}}}(\textsc{Sign}_{\mathsf{K}_{\textsc{Sign}}}(\mathsf{message}))$.

Figure 4: SECOMP Operations

authenticate each other and protect the content of their messages from other parties. Here are more details about the approach.

Secure channel establishment. Step 3 and 4 in Figure 4 show how to set up an end-to-end secure push-messaging channel. The user runs the app, which builds an HTTPS connection to let the user log into the app server with her password or a third party single-sign-on (SSO) scheme (e.g., a token from Facebook to get the user's identity). The server then sends back a secret key (embedded in the cookie it set to the app) through the connection for follow-up authentication and data encryption. After that, the app registers itself with a push-messaging service (e.g., GCM) and delivers the registration ID to the server using an authenticated encryption channel.

In our research, we implemented a set of APIs within the SDKs that wrap the GCM SDKs (Google Play Service) to support a convenient integration of those check-in operations into both an app and its server-side code. Particularly for the existing apps using GCM, their developers only need to use the wrapped SDKs and slightly adjust the way they invoke the GCM APIs to activate the Secomp protection. Later we discuss other design options to help integrate our SDKs (Section 7). Specifically, in our implementation, once an app completes its login, the server runs the API `checkin` to generate the key, log it together with the app's identity, store the key within the HTTPS cookie and send it to the app by setting the cookie through HTTPS. The app keeps the key in its local storage (with the user's session ID), which is utilized by other APIs for encrypting or decrypting messages and checking their integrity. After the app delivers to the server its registration ID, the server records the ID together with the app's identity and key.

Secure communication. Using the secret key, the app and its server encrypts their messages to protect their communication. In our implementation, we adopted AES in Galois/Counter Mode, a known secure and efficient authenticated encryption scheme, for this purpose. Whenever the app or the server receives a message from the other party, it decrypts the message and verifies its integrity. Step 5 and 6 in Figure 4 describe the operations. Within our SDKs, such a message is created by `secureMessage` and parsed/decrypted/verified by `onReceiveMessage`.

Sometimes, the app server needs to broadcast messages to all apps. Such messages are typically public but their integrity and authenticity still need to be protected. Using the secret key here is no longer efficient, due to the need of generating a large number of authentication tags, one for each app. What we did in our research is simply turning to a public-key scheme, as described in Step 7 and 8 in Figure 4. The server signs the message using a secret signing key and each app checks the message with a public verification key, which either comes from the certificate embedded within the app or from the server during the establishment of the secret key.

Security analysis. The security of Secomp can be directly established upon its underlying security primitives. Specifically, the key K is shared between the app server and the app through an authenticated and encrypted channel $\textsc{DirectChannel}$ that is established for login authentication purposes. All unicast messages sent to the app through $\textsc{PushChannel}$ are encrypted with K using an authenticated encryption scheme ($\textsc{Keygen}$, $\textsc{Enc}^{\mathsf{Auth}}$, $\textsc{Dec}^{\mathsf{Auth}}$). Now, the messages sent by the adversary knowing the registration ID RegID would be detected and discarded, as he does not have the key K. We do not guarantee privacy for broadcast messages, so we only make sure that the adversary is not able to compromise the integrity of broadcast messages protected by the unforgeable digital signature scheme. Under this scheme, the attacker may still reorder messages or send invalid messages to squander the recipient's resources. These threats, however, can be easily addressed.

Specifically, the app server can add a sequence number to specify an order for messages. Also, once invalid messages are found, the app can talk to the server through DIRECTCHANNEL, which can then contact the cloud provider to investigate the problem. In Section 6.3, we further show that this simple scheme defeats all the attacks we discovered.

6.2 Misbinding Detection

Although the secure channel established between the app and its app server protects the confidentiality and integrity of their communication, its availability can be hard to guarantee. The problem here is the threat of the misbinding attack, which can be caused by exploiting the vulnerable on-device link or cloud-device link. Such an attack could block the right app from getting its messages (Section 3.1 and Section 4.2). To detect it, we built into Secomp a probing mechanism, which works as follows:

• First, the app server sets a new challenge field on a message, generates a random number N, encrypts it together with the message using the app's secret key and pushes this message to the app.

• The app receiving the message is supposed to decrypt it and sends back to the server an encrypted version of $N + 1$.

• Then, the app server verifies the response and reports to the push-messaging service if the result is incorrect.

The whole idea of this probing mechanism is to help the server find out whether the app can still get the message pushed to it. A problem is that the adversary can act as a MitM (Section 4.2), passing the challenges received by his device to the malicious app on the victim's target device, which in term injects the message to the target app for generating the correct response. The catch here is that the adversary needs to know exactly when to forward the message. Otherwise, he either has to do this all the time, sending every message to the target device, or gets caught when he stops doing that. Therefore, our strategy is to generate a challenge at a random moment: each time when a message to be pushed to an app, its server flips a random coin, with a certain probability (which can be tuned by the developer) to set the challenge field, asking the app to come up with a response. Since this field is within the encrypted content, the adversary cannot determine when to forward the message. As a result, he cannot effectively prevent the legitimate user from getting her messages without being detected.

6.3 Evaluation

To understand how Secomp performs in practice, we built a GCM-subscribing app and its app server to integrate the Secomp SDKs. This app's login operations has been taken care of by the Google SSO. Incorporating our SDKs turned out to be rather straightforward: all we did is just invoking `onReceiveMessage` when the app receives the message from GCM, and `secureMessage` to prepare and send out an upstream message. On the server side, the API `checkin` was used to generate the secret key and send it to the app, `store_regid` for handling the registration ID from the app and `secure_message` for preparing a message before handing it over to the connection server. Here we report an evaluation of its effectiveness in defending against the attacks through GCM and its performance impact.

Effectiveness. We ran the app against all the GCM-related attacks in Section 3 and Section 4. Even though the adversary was still able to hijack the app's registration ID to bind it to the attack device or use it to inject messages to the apps, due to the underlying GCM vulnerabilities, our app was found to effectively fend off all those attacks. Specifically, all the messages that came from the adversary was easily identified through the authenticated encryption scheme and then dropped by `onReceiveMessage`. Also, through the random probing, the app server immediately identified the attack based on exploiting the GCM authentication problem (Section 3.1), since the adversary in this case could not come up with the right response by himself and was not able to play the MitM to let our app do that. More complicated here is the misbinding attack using `PendingIntent` (Section 4.2), in which the adversary had the capability to talk to the target app and could therefore relay the challenge from the server to the target. However, the adversary had to keep on doing that to avoid getting caught: once he stopped, our app server quickly found out the problem after pushing a few messages, as observed in our experiment.

	send message mean/sd(ms)	receive message delay mean/sd(ms)
baseline	2.88 / 10.33	0 / 0
Secomp	6.63 / 11.05	3.63 / 4.87
delay	3.75 / 4.28	3.63 / 4.87

Table 4: Performance of Secomp (test 200 times). Note that in the case of the baseline, the receiving method delivers a message instantly; for Secomp, a very small overhead is introduced for decrypting the message, checking its integrity and restore the data

Performance. We further measured the performance of the app by comparing it with a baseline, a version using unprotected GCM SDKs. Our evaluation focused on the delay caused by receiving and sending messages: for the registration process, it is identical to what happens to the baseline, since our current design combines key exchange with cookie setting and therefore does not incur any extra cost. As we can see from Figure 4, the overheads caused by sending and receiving messages are low (within 10 ms per message), which was completely caused by AES (in the Galois/Counter Mode) encryption and decryption (with a message size of 256 bytes). In Appendix A, we present our measurement of this cost over messages of different sizes.

7. DISCUSSION

We report our security analysis on push-messaging services in the paper, which reveals critical security weaknesses inside the most popular services (e.g., GCM, ADM, etc.), enabling an unauthorized party to lock out the legitimate user of a device, wipe out her data, silently install/uninstall any apps and steal her sensitive messages. Given the complexity of such services, we believe that what we found is nothing more than a tip of the iceberg. Specifically, we only inspected the cloud-device link and on-device link, and have not yet looked into the interactions between connection servers and the developer's app server, and the way that the app server directly talks to the developers' apps. Even for the "links" we studied, our research is still incomprehensive, missing some important services such as Apple Push Notification Service. More effort is certainly needed to dig deeper on this subject to better understand the security risks in push messaging services and mobile clouds in general.

On the defense side, our current design and implementation of Secomp is still preliminary. Particularly, we built our SDKs as a wrapper of the GCM SDKs, making it convenient for the developers to retrofit them into their apps: all they need to do is just a small adjustment of the APIs the apps call to activate the new protection. On the other hand, this treatment makes our tool kit less general, requiring a new implementation for a different service. We are debating on other options, including a design that allows the developer to build into her app no matter what kind of push-message service it subscribes.

8. RELATED WORK

Mobile cloud security. Cloud computing has been used to protect mobile devices, including malware scanning [28] and dynamic analysis of apps [29]. On the other hand, the platform is also abused by the adversary, who uses push-messaging services as a command and control channel for botnets [5, 35]. However, little has been done to understand the weaknesses in protecting the services provided by existing mobile clouds and the consequences once they have been exploited, not to mention any concrete effort to enhance the protection of mobile cloud services, push messaging in particular, toward which we made the first step in our research.

Security implications of Android IPC. Extensive studies have been done on the security of Android Inter-Process Communication (IPC), including the intent broadcasting and service invocation with regard to an action. Examples include the prior work that identifies the security risks in the IPC channel [21, 24, 32, 23], the permission re-delegation problems [25], and the data leak and pollution issues in content providers [36, 33]. Although the security flaws we discovered on the on-device link are often related to the known IPC vulnerabilities (e.g, intent broadcast), which enables our attack app to intercept messages of a cloud-mobile service, oftentimes, such an exposure itself does not directly reveal sensitive user information. Instead, we studied how to use the capability it discloses, the `PendingIntent` object, to collect confidential user data and inject security-critical commands to the victim's apps. Up to our knowledge, this is the first attempt to utilize the object for attacking real systems. Also importantly, our work demonstrates the serious security risk that comes with the common practice of using the `PendingIntent` to provide the origin of an IPC request, which can easily lead to other vulnerabilities.

Also, techniques have been developed to mitigate the security problems in Android IPC, finding the vulnerabilities through a static analysis [21, 27, 26] or a dynamic analysis [22, 20]. These techniques can help detect some vulnerable IPC usages, which we show in our study still widely exist in popular apps and mobile-cloud services. However, we are not aware of any prior effort to secure the end-to-end communication between an app and its server across the underlying mobile cloud service, which can have those known vulnerabilities and the new ones found in our research, and is beyond the control of app developers who need to use it.

Authentication in web applications. The problems in web applications' authentication mechanisms have been extensively studied recently [19, 30, 17]. For example, prior research reveals serious logic flaws in popular single-sign-on systems [31]. New techniques for mitigating the security threats related to those flaws and other authentication problems have also be developed [18, 34]. Our research made the first step in understanding the authentication issues in mobile clouds, particularly the push-messaging services they provide, which has not been done before. A unique feature of those services is that they authenticate their apps without the user's intervention (e.g., entering her credentials), which makes the authentication process more difficult to analyze.

9. CONCLUSION

In this paper, we present the first security analysis on popular push-messaging services. Our research shows that these services are highly error-prone, allowing unauthorized parties to bind a target app's registration to an attack device or inject arbitrary messages to the app, both locally and remotely. As a result, the adversary can intercept sensitive user messages (Facebook posts, Skype messages, bank account balance, etc.) or even command Android service apps to stealthily install/uninstall any apps on the target device, lock out its legitimate user or wipe out her data. The problems were found to affect many popular apps, such as Facebook, Google Plus, Skype and PayPal/Chase apps, bringing in serious security threats to billions of Android users. Fundamentally, they come from questionable practices in developing those services, including weak server-side authentication and access control, and the insecure use of the IPC channels and `PendingIntent`. To mitigate those threats and help app developers protect their communication over those services, we designed and implemented a new technique that establishes an end-to-end secure channel on top of existing push-messaging services.

Given the critical role played by push-messaging services in the mobile ecosystem and their complexity, we expect that more effort will be made to further the understanding of their security implications, and improve our technique to safeguard such channels.

10. ACKNOWLEDGMENTS

We thank Manoj Prabhakaran for useful discussion about the cryptographic solution. The project is supported in part by National Science Foundation CNS-1017782, 1117106, 1223477 and 1223495. Authors from Peking University are supported in part by National Development and Reform Commission (NDRC) under Project "Guidelines for protecting personal information".

11. REFERENCES

[1] adb logcat. `http://developer.android.com/tools/help/logcat.html`.
[2] Android Device Manager. `https://www.google.com/android/devicemanager`.
[3] Android Security Acknowledgements. `https://source.android.com/devices/tech/security/acknowledgements.html`.
[4] Baksmali. `https://code.google.com/p/smali/`.
[5] Cybercriminals use Google Cloud Messaging to control malware on Android devices. `http://www.pcworld.com/article/2046642/cybercriminals-use-google-cloud-messaging-service-to-control-malware-on-android-devices.html`.
[6] Dex2jar. `http://code.google.com/p/dex2jar/`.
[7] Google Cloud Messaging for Android. `http://developer.android.com/google/gcm/index.html`.
[8] Google I/O 2013. `http://www.zdnet.com/io-2013-more-than-half-of-apps-in-google-play-now-use-cloud-messaging-7000015511/`.
[9] JD-GUI. `http://jd.benow.ca/`.
[10] Kindle Fire's market share. `http://www.geekwire.com/2013/kindle-fire-scorching-android-tablet-market-33-share/`.
[11] Mallory. `https://intrepidusgroup.com/insight/mallory/`.
[12] mitmproxy. `http://mitmproxy.org/`.
[13] Sina. `http://www.sina.com.cn/`.
[14] Sina Weibo. `https://play.google.com/store/apps/details?id=com.sina.weibo`.

[15] Supporting materials. `https://sites.google.com/site/cloudmsging/`.

[16] UrbanAirship. `http://urbanairship.com/`.

[17] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An authentication flaw in browser-based single sign-on protocols: Impact and remediations. *Computers & Security*, 33:41–58, 2013.

[18] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *Network and Distributed System Security Symposium*, 2013.

[19] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 247–262. IEEE, 2012.

[20] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, volume 17, pages 18–25, 2012.

[21] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[22] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium*, 2011.

[24] W. Enck, M. Ongtang, P. D. McDaniel, et al. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[25] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[26] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[27] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[28] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, pages 31–35. ACM, 2008.

[29] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[30] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390. ACM, 2012.

[31] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 365–379. IEEE, 2012.

[32] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646. ACM, 2013.

[33] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[34] L. Xing, Y. Chen, X. Wang, and S. Chen. Integuard: Toward automatic protection of third-party web service integrations. In *20th Annual Network and Distributed System Security Symposium, NDSS*, pages 24–27, 2013.

[35] S. Zhao, P. P. Lee, J. Lui, X. Guan, X. Ma, and J. Tao. Cloud-based push-styled mobile botnets: a case study of exploiting the cloud to device messaging service. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 119–128. ACM, 2012.

[36] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.

APPENDIX

A. COST OF PERFORMING ENCRYPTION ON MOBILE DEVICE

Because mobile device has limited battery and computation power, we further measure the cost of doing encryption and decryption on a mobile device. Figure 5 shows the cost of performing encryption and decryption on a mobile device. For each message length, we test 1024 times and calculate the mean value. As we can see, for a message of 4096 bytes (maximum message size allowed by GCM), it takes less than 10ms using `AES in Galois/Counter Mode` on Nexus 7 with Quad-core 1.5 GHz Krait CPU.

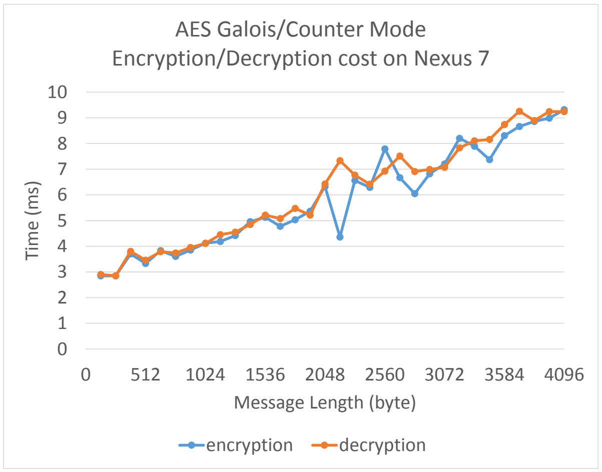

Figure 5: Average cost of performing encryption and decryption on mobile device.