AppCracker: Widespread Vulnerabilities in User and Session Authentication in Mobile Apps

Fangda Cai, Hao Chen
ShanghaiTech University
{caifd, chenhao}@shanghaitech.edu.cn

Yuanyi Wu
Zhejiang Sci-Tech University
wu_yuanyi@icloud.com

Yuan Zhang
Fudan University
yuanxzhang@fudan.edu.cn

***Abstract*—A fundamental security principle in developing networked applications is end-to-end security, where the confidentiality and integrity of the data transmitted over the network do not rely on the security of the network. In response to the ever increasing traffic from mobile apps, WiFi networks are spreading fast and widely. Since WiFi networks are unregulated, a passive attacker may eavesdrop on the traffic on open WiFi networks, while an active attacker may set up his own WiFi network to modify its traffic at will. In theory, end-to-end security should protect mobile apps from both these attacks; in practice, however, the situation is far less rosy.**

We examine how the popular, important mobile apps on Chinese Android markets defend themselves against untrusted networks. We select top apps from major categories, such as online shopping, banking, social networks, travel services, and apps from companies with huge market capitalization. We analyze both their code and their network traffic to identify vulnerabilities. We design a mini-language for describing the vulnerabilities and develop a tool, AppCracker, that launches both passive and active attacks on these apps to verify their vulnerabilities. AppCracker has confirmed that 100 apps from 69 companies are vulnerable during their user or session authentication. These vulnerabilities allow an adversary to capture the victim user's login credentials or to hijack the victim's session. We describe these diverse types of vulnerabilities, many of which are caused by the misuse of cryptography in their home-grown cryptographic protocols. Finally, we discuss the lessons learned during our investigation to help app developers avoid similar pitfalls. We hope that our findings will raise awareness of this problem among both the research community and app developers, and will encourage research in automated tools for detecting these vulnerabilities.

I. Introduction

We rely on mobile apps in many aspects of our life, such as banking, online shopping, social networking, services, health, and entertainment. We store our private information in these apps and conduct financial transactions on these apps. The importance of protecting apps from adversaries cannot be overstated.

Mobile apps communicate with their servers via wireless networks. Wireless networks have expanded rapidly in the recent years in response to the explosive growth of mobile app traffic. Notably, WiFi networks have sprouted widely because, compared to cellular networks, they are less expensive, easier to deploy, and unregulated. Anyone, including adversaries, can set up WiFi networks legally.

To protect applications from untrusted networks, developers must follow the fundamental principle of end-to-end security, which enables applications to communicate with their servers securely even in the presence of adversarial networks. In theory, this principle should protect mobile apps from insecure or malicious networks, but in practice, do app developers follow this principle?

Category	Number of Vulnerable Apps
Banking	2
Online shopping	17
Health	5
Social networking	9
Entertainment	22
Other services	45
Total	100

TABLE I: Number of Vulnerable Apps in Each Category

We take the first step to study how mobile apps meet end-to-end security. We focus on the Chinese Android markets because China has the largest mobile user population and Android is the dominant mobile platform in China. As Google Play is unavailable in China, Chinese users download most of their apps from Chinese markets. Therefore, vulnerabilities in critical Chinese apps would have high impact. We identify major app categories (Table I) and pick top apps from each category. We also select apps from companies that have high market capitalization.

Our investigation consists of two stages. In the first stage, we reverse-engineer and analyze app code. In the second stage, we set up a WiFi access point and capture and analyze traffic from mobile apps. We design a mini-language for describing vulnerabilities in app traffic. To validate these vulnerabilities, we develop an attack tool, AppCracker. It runs in two modes: in the passive mode, it can eavesdrop on the app's traffic; in the active mode, it can insert, delete, or modify the app's traffic. The passive mode represents the scenario when the adversary and victim are on the same open WiFi network, e.g., at coffee shops, airport, and malls. The active mode represents when the adversary creates his own malicious WiFi network to attract unwary victims.

AppCracker has confirmed that 100 apps are vulnerable, shown in the full version of this paper [4]. This finding is significant in several ways. First, these apps are among the top apps in all the categories that we have examined. Table I shows the number of vulnerable apps in each category. Second, 44 of

these apps are from companies whose market capitalization is over one billion US dollars [4]. Third, AppCracker confirms that by exploiting these vulnerabilities, an adversary can steal the victim's login credentials or hijack the victim's session. Finally, the vulnerabilities are diverse, as described below.

In the passive attack, AppCracker eavesdrops on the traffic between the app and its server. We found several types of vulnerabilities during user authentication. The simplest one is that many apps transmit username and password in plaintext or encoded form (e.g., Base64). Some apps do use cryptography to protect their data on the network; however, instead of using standard secure channels such as SSL, they use home-grown protocols and fall victim to various cryptographic pitfalls, such as insecure selection of symmetric and public keys, insecure distribution of keys, insecure modes of operation, insecure message authentication code, or security by obscurity. Besides user authentication, session authentication is also vulnerable in some apps, as they fail to protect their session IDs in the HTTP request body or cookie.

In the active attack, AppCracker can eavesdrop and modify the traffic between the app and its server. We found that many apps that use SSL to establish secure channels fail to validate server certificates. Thus, AppCracker can successfully launch the Man-In-The-Middle (MITM) attack by providing a self-signed certificate to the app. Instead of using server certificates, some apps request public keys directly from the server but fail to validate them, which also facilitates the MITM attack.

The primary lesson learned from our investigation is the importance of end-to-end security in mobile apps. We also show that developers should avoid home-grown cryptographic protocols whenever possible, hide no secret in apps, avoid security by obscurity, and build security into apps from the very beginning.

We make the following contributions:

- We are the first to study the user and session authentication vulnerabilities in top mobile apps in major categories. We found that 100 popular apps are vulnerable. Note that this is a conservative number, as we report an app as vulnerable only after we successfully attack it using AppCracker.
- We describe the diverse vulnerabilities found in these apps.
- We design a mini-language for describing these vulnerabilities and develop a tool, AppCracker, for performing passive and active attacks to validate these vulnerabilities.
- We discuss lessons learned during our investigation, which would help app developers avoid similar vulnerabilities.
- We hope that our findings will raise awareness of these vulnerabilities among both the research community and app developers, and will encourage research in automated tools for detecting these vulnerabilities.

The rest of this paper is organized as follows. Section II defines the threat model and describes related background. Section III describes the method, the mini-language for describing vulnerabilities, and the tool for validating them. Section IV and Section V describe the vulnerabilities and their exploits. Section VI discusses the lessons learned from our investigation. Section VII describes the related work. Section VIII concludes the paper.

II. Threat Model and Background

A. Threat Model

We consider the following threat model in studying user and session authentication.

a) Apps: We assume that apps are benign but potentially vulnerable, but app servers are secure and impenetrable by attackers.

b) Users: We assume that users are benign and well behaved in that they would not perform actions that are known to be dangerous, such as installing untrusted apps on their devices, visiting untrusted domains in their browsers, or clicking untrusted links. However, they often connect to WiFi networks, including unencrypted or untrusted ones.

c) Adversaries: We assume that adversaries cannot access the user's device and do not have code running on the user's device. They cannot trick the user into taking insecure actions. They cannot access the app's server beyond the API that the server provides to its app. However, they can access the WiFi network that the app is running on in two scenarios:

- *Eavesdroppers* can capture all the traffic on an open WiFi but cannot modify or delete any traffic.
- *MITM attackers* can not only capture but also modify traffic. This is the case when the attacker owns or breaks into a WiFi access point.

B. Reverse-engineering Tool Chain

We use a series of open source tools to obtain Java source code and resources from Android application files (APK files).

- *android-apktool* [1] unpackages an .apk file and dissembles Dalvik bytecode into smali code.
- *dex2jar* [6] converts Dalvik bytecode into Java bytecode.
- *jd-gui* [12] provides a GUI to view decompiled Java source code from Java bytecode.
- *procyon* [17] is another tool to decompile Java bytecode into source code.

III. Methodology

We combine offline analysis of Android app code with online analysis of app traffic.

A. Offline Analysis

We download an app's APK file, disassemble it with *apktool* [1], convert its Dalvik bytecode in `classes.dex` to Java bytecode using *dex2jar* [6], and decompile the Java bytecode to get Java source files. We use two tools for decompiling Java bytecode: *jd-gui* [12] and *procyon* [17], because they occasionally fail on some class files but do not usually fail on the same class file. We examine the decompiled Java source files and the APK's string resource files to try to identify cryptographic methods and keys.

B. Online analysis

We use a Linux machine running *ap-hotspot* as the WiFi access point and connect our Android phone to it. Then, we capture packets from apps running on the phone. We use different methods for capturing HTTP and HTTPS traffic.

- **HTTP**. We capture HTTP traffic from the app on a laptop running Linux (other than the machine serving as the WiFi access point). We configure the wireless card in the monitor mode, which allows the wireless driver to eavesdrop on *all* packets.
- **HTTPS**. Since HTTPS traffic is encrypted, we capture and decrypt it using MITM (Man In The Middle) attack on the WiFi AP. On this AP, we run a web server and configure *iptables* to redirect all the incoming packets destined for port 80 and 443 on other hosts to this web server's HTTP and HTTPS ports, respectively. This web server provides our self-signed certificate during SSL handshake and forwards each request to AppCracker to be described in Section III-C. Note that this attack succeeds only on apps that fail to validate certificates.

Once we capture packets, we examine the HTTP payload in the packets to identify useful data — such as plaintext or encrypted username, password, cookies, session IDs, signatures, etc — and modify them as needed, described in Section III-C.

C. AppCracker

We have developed a tool, AppCracker, to describe the vulnerabilities in apps and to validate our attacks. The tool runs in two modes:

- Passive mode (as an eavesdropper): AppCracker takes as input packets from the app (e.g., the output of *tshark*, a command line tool for dumping network traffic) and extracts useful data from the packets.
- Active mode (as a MITM attacker): Figure 1 shows AppCracker in the active attack mode.

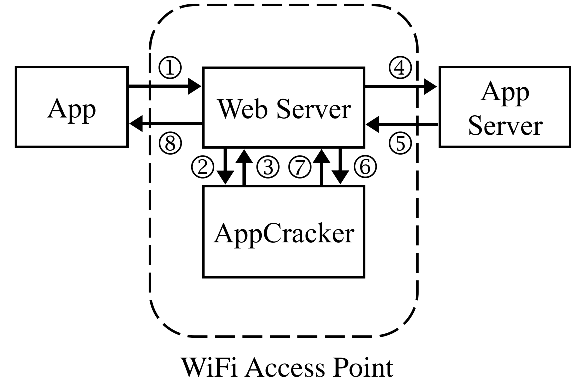

Fig. 1: AppCracker running in the active attack mode. AppCracker interacts with an unmodified Apache web server running on the same WiFi AP.

1) The app sends an HTTP(S) request to the WiFi AP. The *iptables* running on the AP redirects the request to the web server running on the same AP.
2) The web server forwards the request to AppCracker.
3) AppCracker modifies the request and sends it back to the web server.
4) The web server forwards the request to the app server.
5) The web server receives the response from the app server.
6) The web server forwards the response to AppCracker.
7) AppCracker modifies the response and sends it back to the web server.
8) The web server forwards the response back to the app.

Initially, we wrote individual code for analyzing each app in AppCracker. But as the number of apps grows, the size of the code quickly increases because there is much repetition within the code. To mitigate this code complexity, we design a mini-language to describe the vulnerabilities in each app.

The mini-language is in the YAML [23] format. The root document is a dictionary where a key is the name of an app and its value describes the fields that are interesting to the adversary in the app's payload. AppCracker extracts these fields and records them in a database. Instead of providing a formal specification, which might be difficult to understand, we will demonstrate the simplicity and power of this mini-language by examples. In these examples, text in **bold** font represents keywords in the mini-language.

Figure 2a describes the app *hello*. It is a dictionary consisting of the key **hosts** and **rules**. The value of **hosts** is a list of host names as they appear in the **Host** header in HTTP. Based on these host names, AppCracker classifies each HTTP payload. The value of **rules** consists of a list of rules. Each rule consists of the key **predicate**, **request**, **response**, and **cookie**. The value of **predicate** consists of the key **path**, **args**, and **form**. Together they specify a filter to determine which payload this rule applies to. **path** matches the path component of the URL, and **args** and **form** match the keywords in the URL parameters and in the form data from POST request, respectively. **request** describes the action on the HTTP request payload. It is a dictionary whose keys are **args** and **form**, which describe the keys to examine in the URL parameters and in the form data from POST request, respectively. When AppCracker runs in the passive mode, it extracts the values of these keys and stores them in a database. When it runs in the active mode, it replaces the values of these keys with those previously extracted and stored in the database.

On some apps, AppCracker needs to analyze the HTTP payload in more complex ways than merely extracting values of certain keys or replacing certain values with constants. In these cases, we specify a user-defined function written in Python for processing the payload. Figure 2b instructs AppCracker to invoke the function `hello_request()` on the HTTP request and the function `hello_response()` on the HTTP response. The user may also instruct AppCracker to extract or replace the HTTP cookie, shown in Figure 2c.

```
hello:
  hosts:
    - m.hello.xyz
    - api.hello.xyz
  rules:
    - predicate:
        path: login
      request:
        args:
          - username
          - password
    - predicate:
        args:
          - function: order
      request:
        form:
          - sessionid
```

(a) Simple extraction or replacement of values of certain keys can be specified entirely in the vulnerability specification.

```
hello:
  rules:
    - request: hello_request()
    - response: hello_response()
  ...
```

(b) More complex operations on payload may be delegated to user-defined python functions

```
hello:
  rules:
    - cookie:
      - JSESSIONID
  ...
```

(c) Extracting and replacing HTTP cookie

Fig. 2: Examples of vulnerability specification for AppCracker

D. Dataset

Since our investigation requires intensive manual effort, we ought to prioritize on the importance of apps. We select important apps by the following criteria:

- Top apps by number of downloads on the Baidu App Market [3] in each of the major categories in Table I.
- Apps from companies with huge market capitalization.

From the above apps, we exclude apps that use SSL *and* validate certificates, because neither the passive attack nor the active attack of AppCracker would be effective. After this stage, we were left with 113 apps. Among these apps, AppCracker successfully launched attacks on 100 of them.

We report the results of our investigation in Sections IV and V. The results are based on the versions of the apps as of August and September of 2014. Some apps, such as *YiHaoDian*, have since fixed their vulnerabilities, but we leave them in this paper for reference.

More information about the apps whose vulnerabilities are discussed below can be found in the full version of this paper [4]. Section IX explains our rationale and policy for disclosing vulnerability details.

IV. Passive Attacks

In passive attacks, the adversary can only eavesdrop on the traffic between the app and its server but not insert, modify, or delete traffic. This is the case when the adversary is in the same unencrypted WiFi network — e.g., at cafes, airports, and malls — as the app.

A. Plaintext or Encoded Data

Some apps do not encrypt their traffic. *DianPing* is the most popular app for reviewing businesses and purchasing discount gift certificates. It does not encrypt its traffic. Once an attacker penetrates the victim user's account, e.g., by eavesdropping on the password or hijacking a session, he can steal the secret IDs of the gift certificates in the account. Some apps encrypt their traffic during only certain, perhaps more security-critical, phases. For example, *YiHaoDian* is a major online retailer in China. It encrypts its traffic during login, but not during all the other actions.

Eavesdropping on plaintext is trivial. For example, during login, *DangDang* (another top online retailer) sends the username and password in plaintext in the POST request. Some apps try to achieve security through obscurity by encoding confidential data. For example, *Vancl*, another online retailer, encodes the username and password in Base64. Moreover, in the pursuit of readability, it names the keys of the username and password in its POST body as `username` and `password`, which inadvertently helps the eavesdropper.

B. Encryption

Some apps do realize the need to protect confidential data. However, instead of using common high-level protocols, such as SSL, they home-brew their own protocols from cryptographic primitives, such as symmetric key, public key, and secure hash algorithms. In the process, they fall victim to misusing cryptography.

1) Symmetric Key Encryption: Some apps use symmetric key encryption. However, they fail to select secure keys, to distribute keys securely, or to use secure modes of operation.

Insecure key selection Some apps fail to create random keys. For example, *IqiyiVideo*, the most popular video service, uses AES to encrypt the username and password during login. Its key is `iqiyi123)(*\x00\x00\x00\x00\x00`, which is hardly random as it starts with the domain name of the company (`iqiyi`), followed by `123)(*`, and padded with zeros to 128 bits.

Even when an app chooses a random symmetric key, if the app always uses the same key, it is still vulnerable as the adversary can use the app as an encryption oracle even if he does not know the key. For example, *JuMeiYouPin*, an online retailer of discount luxury goods, uses DES to encrypt the username and password during login. Its key is the constant 8e2bf219.

Insecure key distribution A difficulty with symmetric key cryptography is how to distribute keys confidentially.

```
public class Des
{
    public static String getDefaultKey() {
        return "8e2bf219";
    }
}
```

Fig. 3: *JuMeiYouPin* embeds its DES key in class `com.jm.android.jumei.tools.Des`

Since adversaries have full access to apps' bytecode and data, it would be insecure to embed symmetric keys in apps. Unfortunately, many apps commit this mistake.

JuMeiYouPin embeds its DES key in the class `com.jm.android.jumei.tools.Des`, shown in Figure 3.

IqiyiVideo embeds its AES key in the class `org.qiyi.android.corejar.k.a.at`. Instead of specifying the key as a string, it specifies the key as a byte array, which translates to the string `iqiyi123)(*\x00\x00\x00\x00\x00` (Figure 4).

During login, *Taobao*, the largest online retailer, computes a message authentication code over some of the fields in its payload to protect message integrity. Besides these fields, the signature algorithm also takes as input a value `appsecret`, which does not appear in the payload. Purportedly, this makes it difficult for an eavesdropper who can access only the payload to fabricate the signature. *Taobao* computes `appsecret` based on the data in the app and takes several steps to obfuscate the process. First, it provides the string `appsecret` in `res/values/strings.xml` shown in Figure 5a. Next, the class `com.taobao.tao.util.Constants` reads the string into the variable `Constants.appsecretSigned` during static initialization. Finally, the method `getAppsecret` in the class `com.taobao.tao.util.Constants` computes `appsecret` from the variable `Constants.appsecretsigned` by subtracting `i%5` from each character where `i` is the index of the character in the string, shown in Figure 5b.

Insecure modes of operation ECB (Electronic Code Book) mode of operation of block ciphers is insecure, because it encrypts identical blocks of plaintext into identical ciphertext blocks. However, many apps use ECB mode of operation. *IqiyiVideo* uses AES in ECB mode (`AES/ECB/PKCS7Padding`). *JuMeiYouPin* uses DES in ECB mode. We are not surprised at this vulnerability, as prior work found similar problems in English apps [7].

2) Public Key Encryption: We describe traps in using public key cryptography in this section.

Insecure key selection *TMall*, *JuHuaSuan*, and *1688* are e-commerce apps from Alibaba and share the same home-grown cryptographic libraries. They use RSA to encrypt user password during log in. First, the app sends a key request to the server, and the server replies with a public key. Then, the app uses the key to encrypt the user password. The encryption exponent (e) in the public key returned by the server is always 3. This is a common choice because it allows the client to compute encryption very efficiently. However, for this choice of encryption exponent to be secure, the app must pad the plaintext; otherwise, the adversary can decrypt the ciphertext without knowing the private key if the plaintext is small.

In the case of *TMall*, *JuHuaSuan*, and *1688*, the public key provided by the server has 1024 bits. When the plaintext has fewer than 1024/3 bits, the modulus n in the public key has no effect on the ciphertext. In this case, to recover the plaintext, the eavesdropper simply computes the cubic root of the ciphetext (since the encryption exponent is 3). These apps encode each character in the password as a byte, so as long as the password has fewer than $1024/3/8 \approx 42$ characters, which is almost always the case, the eavesdropper can recover the password without knowing the private key.

Taobao is another app from Alibaba. Since it uses 65537 as the public exponent, it does not suffer from this vulnerability. However, Section V-B will describe an active attack on *Taobao*'s public key.

C. Message Integrity

When apps do not use SSL, besides home-growing their encryption mechanisms to protect message confidentiality, they sometimes home-grow their own mechanisms for protecting message integrity too. Again, this attempt is often fraught with traps.

Message authentication code *TMall*, *JuHuaSuan*, and *1688* use the same message authentication code (MAC) to protect the integrity of the payload. They compute the MAC as the MD5 over `appkey`, `appsecret`, `api`, `v`, `imei`, `imsi`, `data`, and `t`. All of these values but `appsecret` are available in the traffic, so apparently `appsecret` is intended to serve as the secret key to this MAC. Section IV-B1 describes how we discover `appsecret` in the app code.

Instead of applying MD5 directly, these apps use a convoluted algorithm that applies MD5 multiple times. Figure 6 shows this method `getSign` in the class `android.taobao.util .TaoApiSign`. We conjecture that the purpose of this complexity is to obfuscate the algorithm, which might deter the adversaries who have no access to this code. But once we located this code, it took us 10 minutes to reimplement it in Python (because AppCracker was developed in Python) to start to forge MAC freely.

Obfuscated message authentication code During login, *JingDong*, the second largest online retailer, sends the username in plaintext. However, instead of sending the password in plaintext, it sends the MD5 of the password, which makes it difficult to recover the password barring the dictionary attack and attacks on MD5. To prevent replay attacks, the payload also includes a MAC. The MAC is taken over the username, MD5 of the password, `functionId`, `uuid`, and current time, all of which are available in the payload.

To deter reverse engineering, *JingDong* implements the MAC algorithm in the native code whose source code is

```
public class at extends con
{
  private static byte[] a = { 105, 113, 105, 121, 105, 49, 50, 51,
                              41, 40, 42, 0, 0, 0, 0, 0 };
}
```

Fig. 4: *IqiyiVideo* embeds its AES key in class `org.qiyi.android.corejar.k.a.at`.

```
<string name="appsecret">756h;d8g:429d;57cf&lt;j8g5f:f3:d&lt;d4</string>
```

(a) The original string for `appsecret` in `res/values/strings.xml`.

```
public class Constants
{
    public static String getAppsecret() {
        if (Constants.appsecret == null || "".equals(Constants.appsecret)) {
            String string = new String();
            for (int i = 0; i < Constants.appsecretSigned.length(); ++i) {
                string += (char)(Constants.appsecretSigned.charAt(i)-i%5);
            }
            Constants.appsecret = string;
        }
        TaoLog.Logd("appkey", Constants.appsecret);
        return Constants.appsecret;
    }
}
```

(b) *Taobao* computes the final `appsecret` based on the original string in Figure 5a, apparently to obfuscate this value.

Fig. 5: *Taobao* embeds and obfuscates *appsecret*.

unavailable. The class `com.jingdong.app.Sign` defines the native MAC method as:

```
public static native Map getSignMap(
        Map paramMap, List paramList);
```

The MAC method, `getSignMap`, takes a `Maps` and a `List` as parameters. Based on how *JingDong*'s Java code invokes `getSignMap`, it appears that the MAC algorithm iterates over the keys in `List`, retrieves their corresponding values in `Map`, and uses these values in the input. The keys are always `functionId`, `body`, and `uuid`. In `Map`, `functionId` is the name of the API method on the server that this request invokes, e.g., `login`. `uuid` is a constant. `body` is the JSON representation of the MD5 of the password and the plaintext of the username, e.g.,

```
{"loginpwd":"0123456789ABCDEF...",
 "loginname":"root@abc.xyz"}
```

To further deter the adversary from forging the MAC, the current time is not an input parameter to `getSignMap` but rather an output from `getSignMap`. `getSignMap` returns a `Map` consists of three keys: `sign`, `sv`, and `st`. `sign` contains the computed MAC, e.g., `WHSyyVjZOe3mr7-07Qe0Ig`, `sv` is always 1, and `st` contains the current timestamp. *JingDong* then appends these keys and their values to the URL parameters.

`getSignMap` does not seem to use any common method for computing hash values, as the `sign` value returned by `getSignMap` does not meet any common encoding standard based. `getSignMap` is provided in the native code library *libjdmobilesecurity.so*, whose size is 17592 bytes. Since reverse-engineering this library would be time consuming, we use the native method `getSignMap` as an oracle during our attack. We wrote a small Java app, *jdtricker*, that takes as input `functionId`, `body`, and `uuid`, creates a `Map` and `List`, calls `getSignMap`, and then extracts `sign`, `sv`, and `st` from the returned `Map`.

Our replay attack proceeds as follows. First, during the eavesdropping stage, AppCracker records `functionId`, `body`, and `uuid` in the login traffic from the victim user. Then, during the replay stage, AppCracker runs our *jdtricker* in an Android emulator where we have installed *libjdmobilesecurity.so*. AppCracker sends the recorded victim's `functionId`, `body`, and `uuid` to *jdtricker*, which then calls `getSignMap` to get `sign`, `sv`, and `st`, and returns them to AppCracker. Finally, AppCracker replays the recorded login traffic but replaces `sign`, `sv`, and `st` with the fresh values returned from `getSignMap`.

D. Sessions

Since HTTP requests are stateless, web applications use sessions to carry states between subsequent requests. Session

```
public static final String getSign(Map<String, String> paramMap) {
  while (true) {
    String str7;
    try {
      String str1 = (String)paramMap.get("appKey");
      String str2 = (String)paramMap.get("appSecret");
      String str3 = (String)paramMap.get("api");
      str4 = (String)paramMap.get("v");
      String str5 = (String)paramMap.get("imei");
      String str6 = (String)paramMap.get("imsi");
      str7 = (String)paramMap.get("data");
      String str8 = (String)paramMap.get("t");
      String str9 = (String)paramMap.get("ecode");
      if (str4 != null) {
        if (!"".equals(str4)) break label381;
        String str10 = DigestUtils.md5ToHex(new ByteArrayInputStream(
                       str1.getBytes("UTF-8")));
        StringBuffer localStringBuffer = new StringBuffer();
        if ((str9 != null) || ("".equals(str9))) {
          localStringBuffer.append(str9);
          localStringBuffer.append("&");
        }
        localStringBuffer.append(str2);
        localStringBuffer.append("&");
        localStringBuffer.append(str10);
        localStringBuffer.append("&");
        localStringBuffer.append(str3);
        localStringBuffer.append("&");
        localStringBuffer.append(str4);
        localStringBuffer.append("&");
        localStringBuffer.append(str5);
        localStringBuffer.append("&");
        localStringBuffer.append(str6);
        localStringBuffer.append("&");
        localStringBuffer.append(DigestUtils.md5ToHex(
                         new ByteArrayInputStream(str7.getBytes("UTF-8"))));
        localStringBuffer.append("&");
        localStringBuffer.append(str8);
        String str11 = DigestUtils.md5ToHex(new ByteArrayInputStream(
                       localStringBuffer.toString().getBytes("UTF-8")));
        return str11;
      }
    }
}
```

Fig. 6: *TMall*, *JuHuaSuan*, and *1688*'s function for computing MAC

IDs are the data that web servers use to link requests to sessions. Session IDs may be stored in URL parameters, the request body, and the cookie. Since session IDs serve as the authentication token, they must be protected.

Session ID in request body During user login, *YiHaoDian* sends `userToken` in the URL parameters. After the authentication succeeds, *YiHaoDian* includes `userToken` in all the messages in the same session. Although *YiHaoDian* sends login requests via HTTPS, it sends all subsequent messages via HTTP. From these messages, AppCracker eavesdrops on `userToken` of the victim user, and then replays it to hijack the victim user's session.

Session ID in cookie *HZBank* uses `JSESSIONID` in the HTTP cookie as its session ID. *PingAnBank* uses the combination of `JSESSIONID` and BIGipServerIBANK-IBP_little_core_test_Pool in the HTTP cookie as its session ID. By eavesdropping and then replaying these IDs, AppCracker successfully hijacks the victim user's session.

V. Active Attacks

In active attacks, the attacker may not only eavesdrop on the traffic but also insert, modify, and delete traffic. The attacker can accomplish this by penetrating a trusted WiFi router or providing his own rogue router. This attack is referred to as *Man-In-The-Middle (MITM)* attack.

A. Forged Certificate

A common mechanism for establishing a secure channel is SSL. However, to defend against the MITM attacks, the client must validate the server's certificate. Unfortunately, most apps

that use SSL that we examined fail to validate certificates, shown in the full version of this paper [4].

AppCracker creates its self-signed certificate whose common name is our domain, but all the above apps accept this certificate. This indicates that they fail to verify at least two items:

- that the certificate authority is trusted.
- that the common name in the certificate matches the expected domain name.

B. Public key substitution

Taobao, *TMall*, *JuHuaSuan*, and *1688* are all apps from Alibaba. During login, the apps first sends a request to the server and the server returns a RSA public key. Then, the app sends the username in plaintext but encrypts the password using the public key.

Section IV-B2 describes the vulnerability in the public keys sent to *TMall*, *JuHuaSuan*, and *1688*: because they choose the public exponent e to be 3 without padding the plaintext, it becomes trivial for the attacker to decrypt the ciphertext without knowing the private key. However, the exponent e in the public key sent to *Taobao* is 65537, so the above passive attack does not work.

Fortunately for the attacker, *Taobao* fails to validate the public key returned by the server. AppCracker takes advantage of this vulnerability to attack *Taobao*. It blocks the response that contains the public key from *Taobao*'s server, saves the public key in its database, replaces it with its own public key in the response, and then forwards the modified response to *Taobao*. Subsequently, when the app sends the next request containing the password, which has been encrypted with AppCracker's public key, AppCracker blocks the request, decrypts the ciphertext of the password, reencrypts the password with the saved original public key, and then forwards the request to the server. During our attack, neither the app nor the server seems to notice the attack because the app continues to run properly.

VI. Discussions

A. Lessons Learned

The primary lesson learned during our evaluation is that end-to-end security matters. Since it would be infeasible to always expect a secure network, apps must protect themselves by establishing secure channels. We describe other lessons below.

1) Use Standard Cryptograpic Protocols: Many apps do not use SSL to establish a secure channel. Instead, they home-brew their own cryptographic protocols. As shown in Sections IV and V, these protocols are often prone to cryptographic pitfalls.

We interviewed a few vendors about why they did not use SSL. A common concern is its performance penalty. Since the SSL protocol requires several round trips during the handshake, it causes significant delays on wireless networks with large latency, such as 2G (300-1000 ms) and some 3G networks (100-500 ms) popular in small towns and villages. At one point, a vendor switched their app from HTTP to HTTPS, but only to reverse course after receiving an avalanche of user complaints in just two days. While we sympathize with the dilemma, we believe that developing ad hoc cryptographic protocols is the wrong approach fraught with pitfalls. We suggest that research be conducted on secure channels that are less sensitive to network latency.

2) No Place to Hide in the App: Since the app is available to any user, including adversaries, there is no place to hide secrets in the app. Any component in the app, such as resources, bytecode, and even native code, may be reverse-engineered and analyzed. This observation implies that embedding symmetric keys in the app is insecure.

3) Security through Obscurity doesn't Work,: Several apps try to prevent reverse-engineering by obfuscation in a number of ways:

- Obfuscate Dalvik code. For example, ProGuard *shrinks, optimizes, and obfuscates your code by removing unused code and renaming classes, fields, and methods with semantically obscure names.* [18] However, this won't deter the determined reverse-engineers – it merely makes their jobs longer. We note that in such obfuscated Dalvik code, although user-defined names have been obfuscated, API method names (e.g., the Java Crypto Library) have not been, which helps reverse-engineers.
- Use native code. Native code is much less readable than Dalvik bytecode, which makes it more difficult for the reverse engineer to understand its functions and algorithms. However, as we showed in Section IV-C, we were able to use *JingDong*'s native code library as a signing oracle to forge messages with the correct MAC without needing to understand the native code.

4) Build Security into Apps from the Beginning: Several vendors told us that they did not believe that security was critical in the beginning of their apps. For example, *DianPing* started as an app for users to review businesses. Since it supported no financial transactions, the developers believed that they could not justify the overhead of using SSL to protect network traffic. However, a few years later, the app began to allow users to buy discount gift certificates and stores each certificate's unique ID in the user's account. This new function makes a user's account much more valuable to adversaries, because to redeem a certificate, one needs to know only its ID. Since *DianPing* did not build security into its framework in the beginning, it now finds it difficult to retrofit security in.

B. Limitations

Our work has a number of limitations. First, we conservatively consider an app vulnerable only after we have successfully attacked it. There are many more apps that are potentially vulnerable, but before we attack them successfully, we do not report them as vulnerable. For example, some apps send MD5 hash of the password in plaintext but also include a MAC that we have not reverse-engineered or found an oracle for. Even though an adversary may launch a dictionary attack

on the MD5 of the password, we do not report these apps as vulnerable yet.

C. Future Work

Currently AppCracker relies on manual analysis of apps to discover their vulnerabilities. We are working on automating this process. As a first step, we try to automatically detect apps that fail to encrypt secret credentials in their network traffic.

VII. Related Work

A. App Vulnerabilities

Misuse of security libraries is a common cause of vulnerabilities in apps. Fahl et al. [10] revealed six types of flaws in the use of SSL/TLS by apps, such as missing validation of sever certificates and allowing all host names. They designed a tool to detect potential SSL vulnerabilities and found 8.0% of the apps potentially vulnerable to Man-In-The-Middle attack. SMV-Hunter [20] combined static analysis and dynamic validation to improve the precision of detecting unsafe SSL use. Georgiev et al. [11] revealed that many security-critical apps and libraries failed to validate SSL certificate correctly, mainly due to the badly designed APIs of SSL implementations. While we also consider SSL vulnerabilities, we examine a much wider range of vulnerabilities in user and session authentication in apps. Particularly, we examine how an eavesdropper, who is unable to launch a MITM attack on SSL certificates, can compromise apps' confidentiality.

Egele et al. [7] studied the misuse of cryptographic API in Android apps and summarized six basic rules in using cryptographic libraries. They designed a program analysis tool, CryptoLint, to detect rule violations in 11,748 applications and found 88% of the apps violating at least one rule. Similar to CryptoLint, we also study the misuse of cryptography. But different from CryptoLint, we discover concrete vulnerabilities and demonstrate live attacks resulting from the misuse. We also discover problems beyond the misuse of crytographic API, such as weak home-grown message authentication code.

Schrittwieser et al. [19] evaluated the authentication mechanisms of nine popular mobile messaging and VoIP applications. They found that no additional authentication mechanisms other than the phone number were used by these applications in authentication. Specifically, most of these apps used SMS or phone calls to transmit authentication code. They found that six apps were vulnerable to Account Hijack Vulnerability due to the insecure use of phone number to authenticate users. By contrast, we study the misuse of cryptography during user and session authentication.

In mobile apps, third-party SDKs are frequently used to access online services, such as cloud storage, social networking. These SDKs are intended to help developers integrate their services; however integrating security-critical third-party services securely is difficult. Wang et al. [21] identified serious authentication and authorization flaws in applications that integrate Single-Sign-On SDKs, even when developers strictly follow the SDK documentation.

Dynamic class loading allows applications to load additional code from external resources at runtime. Poeplau et al. [16] showed that developers of benign applications could inadvertently introduce vulnerabilities when they dynamically loaded code from untrusted resources. The insecure use of dynamic class loading poses a great threat to the integrity and confidentiality of application code and data. They found that 9.25% of 1,632 popular Android applications loaded external code insecurely.

B. App Security Analysis

Enck et al. [8] evaluated the security of 1,100 popular apps by examining the decompiled source code with the ded [13] decompiler. They uncovered pervasive use/misuse of personal/phone identifiers, and deep penetration of advertising and analytics networks in Android apps. The dare decompiler [14] extends the ded decompiler to support further analysis on the decompiled class files. TaintDroid [9] is a dynamic flow tracking tool for detecting information leak in Android apps. Crussell et al. [5] designed a hybrid analysis tool, MAdFraud, to investigate ad fraud in Android apps. MAdFraud could automatically identify fraudulent apps that request ads while the app is in the background or click on Ads without user interaction. FlowDroid [2] is a highly-precise taint analysis framework for Android apps. Epicc [15] supports precise inter-component analysis for large-scale Android apps. Amandroid [22] is a precise and general inter-component data flow analysis framework for security vetting of Android apps.

VIII. Conclusion

We took the first step to study user and session authentication in the most popular mobile apps on Chinese Android markets and confirmed that 100 apps from 69 companies contain vulnerabilities. We designed a mini-language for describing these vulnerabilities and developed a tool, AppCracker, to launch passive and active attacks to verify these vulnerabilities. Many of these vulnerabilities are caused by the misuse of cryptography in the apps' home-grown cryptographic protocols, and violations of principles in security engineering. We hope that our findings will raise awareness of this problem among both the research community and app developers, and will encourage research in automated tools for detecting these vulnerabilities.

IX. Disclosure

We endeavor to balance the protection of apps and vendors and the dissemination of knowledge. Therefore, we notified all the vendors in early October of 2014.

X. Acknowledgment

We thank HTC Corporation for providing smartphones for the experiments.

References

[1] *Android Apktool*. URL: https://code.google.com/p/android-apktool.

[2] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom, 2014, pp. 259–269. ISBN: 978-1-4503-2784-8.

[3] *Baidu App Market*. URL: http://shouji.baidu.com.

[4] Fangda Cai, Hao Chen, Yuanyi Wu, and Yuan Zhang. *AppCracker: Widespread Vulnerabilities in User and Session Authentication in Mobile Apps*. (Full version of this paper). URL: http://shtech.org/research/paper/appcracker.pdf.

[5] Jonathan Crussell, Ryan Stevens, and Hao Chen. “MAdFraud: Investigating Ad Fraud in Android Applications”. In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’14. Bretton Woods, New Hampshire, USA, 2014, pp. 123–134.

[6] *dex2jar*. URL: https://code.google.com/p/dex2jar.

[7] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. “An Empirical Study of Cryptographic Misuse in Android Applications”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. Berlin, Germany, 2013, pp. 73–84. ISBN: 978-1-4503-2477-9.

[8] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. “A Study of Android Application Security”. In: *Proceedings of the 20th USENIX Security Symposium*. USENIX Security’11. San Francisco, CA, Aug. 2011.

[9] William Enck et al. “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”. In: *Proceedings of OSDI’10*. 2010.

[10] Sascha Fahl et al. “Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA, 2012, pp. 50–61.

[11] Martin Georgiev et al. “The most dangerous code in the world: validating SSL certificates in non-browser software”. In: *ACM Conference on Computer and Communications Security*. 2012, pp. 38–49.

[12] *JD-GUI*. URL: http://jd.benow.ca.

[13] Damien Octeau, William Enck, and Patrick McDaniel. *The ded Decompiler*. Tech. rep. NAS-TR-0140-2010. Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA: Network and Security Research Center, Sept. 2010. URL: http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf.

[14] Damien Octeau, Somesh Jha, and Patrick McDaniel. “Retargeting Android Applications to Java Bytecode”. In: *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina, 2012, 6:1–6:11. ISBN: 978-1-4503-1614-9.

[15] Damien Octeau et al. “Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis”. In: *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 543–558. ISBN: 978-1-931971-03-4. URL: http://siis.cse.psu.edu/epicc/papers/octeau-sec13.pdf.

[16] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications”. In: *Proceedings of NDSS’14*.

[17] *Procyon*. URL: https://bitbucket.org/mstrobel/procyon.

[18] *ProGuard*. URL: http://developer.android.com/tools/help/proguard.html.

[19] Sebastian Schrittwieser et al. “Guess Who Is Texting You? Evaluating the Security of Smartphone Messaging Applications”. In: *Proceedings of NDSS’12*. Feb. 2012.

[20] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. “SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps”. In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS’14)*. San Diego, CA, Feb. 2014.

[21] Rui Wang et al. “Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization”. In: *Proceedings of USENIX Security ’13*. Washington, DC.

[22] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. “CHEX: statically vetting Android apps for component hijacking vulnerabilities”. In: *Proceedings of CCS ’14*. Scottsdale, AZ, Nov. 2014.

[23] *YAML*. URL: http://www.yaml.org/.