Toward Discovering and Exploiting Private Server-side Web APIs

Jia Chen[†], Xingmin Cui[‡], Ziming Zhao[§], Jie Liang[¶] and Shanqing Guo*[†‖]
[†]School of Computer Science and Technology, Shandong University
[‡]The University of Hong Kong
[§]Arizona State University
[¶]China Information Technology Security Evaluation Center
[‖]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
Email: chenjia@mail.sdu.edu.cn, xmcui@cs.hku.hk, zmzhao@asu.edu, liangj228@gmail.com, guoshanqing@sdu.edu.cn

***Abstract*—Many service providers including large enterprises have released their own applications (apps) that incorporate HTTP clients to facilitate the communications with their servers. The workflows of and APIs used by a web app and its corresponding mobile app are not always the same. We call the APIs found in apps *private web APIs* in that they are only supposed to be invoked by apps that developed by the service providers themselves. However, checking the origin of an HTTP request is very difficult, and private web APIs can be easily invoked by other entities. Hence, it is imperative to study if private web APIs provide the same level of security checks and validations as their public counterparts. To automatically discover the undocumented private APIs in Android apps, we design a system that uses static analysis to find the activities that invoke web APIs. Our system then runs the discovered activities on a customized Android system to monitor its HTTP requests and responses. We evaluated our system on 76 popular apps on the Google Play market. Our system successfully run 48 apps and discovered many private server-side APIs from more than 30 apps. Further manual investigation discovered that 9 of the apps have vulnerabilities that would enable API misuse and session hijacking.**

***Index Terms*—Web APIs, Android Apps, Static Analysis, Dynamic Analysis**

I. Introduction

In the era of personal computers, users use one web browser, such as Internet Explorer, Chrome, or Firefox, to visit different websites. Web browsers usually provide an address bar where users can use a keyboard to input a uniform resource locator (URL) to specify their destinations. However, using web browsers on small-sized and keyboard-free mobile devices, such as smartphone and tablets, is much more unwieldy. As mobile devices gain more popularity and the Internet usage from them has surpassed the usage from PC [4], more and more service providers release an alternative app for users to access their services in hopes of providing better user experiences.

Apps usually use a web client, such as `WebView` or `DefaultHttpClient` on the Android platform, to invoke web APIs provided by the servers. Since these apps are developed by the service providers themselves, they are usually mistakenly regarded as a part of the trusted computing base. The web APIs invoked in these apps are usually designed as private and only supposed to be called by the right apps. However, it is difficult for the server-side to verify if the requests sent to these private web APIs are actually from legit apps. We argue that this is a bad practice of achieving security through obscurity, and attackers can either discover these APIs and craft malicious requests or modify the original apps to send crafted requests. Since app usage (80% of time) dominates browsers (20% of time) in mobile usage [4], it is imperative to study if the private web APIs can provide the same level of security checks and validations as regular web interfaces that are designed to be accessed by general browsers.

To answer this question, the first step is to discover the URLs and parameters of such private APIs. Since these APIs are not documented, we propose an approach to automatically discover them from Android apps by hooking the relevant methods in the Android system to monitor HTTP requests and responses. To this end, we design a novel system to first use static analysis to find the Android activities that invoke web APIs. Then, our system triggers these activities and monitors them on our customized Android system. The customized Android system is responsible for logging information of HTTP requests and responses, such as URLs, parameters and HTTP headers. By analyzing the HTTP responses, we can check whether they are vulnerable to typical web-based attacks, such as session hijacking, web API misuse, etc. Even though we use Android apps to demonstrate our approaches, our approaches are not limited on the Android platforms and are applicable to iOS and other mobile systems as well. The discovered private API vulnerabilities are agnostic to other mobile platforms too.

To evaluate our approach, we use 76 out of the 120 most popular apps on the Google Play market as our test dataset. Our system successfully run 48 apps and discovered many private server-side APIs from more than 30 apps. We manually analyze the discovered APIs and find that 9 of the apps are vulnerable to attacks such as session hijacking and API misuse.

Our contributions are the following:

- We designed and implemented a system that can automate

*Corresponding Author

978-1-5090-2675-3/16 $31.00 © 2016 IEEE
DOI 10.1109/ICWS.2016.61

the process of discovering private web APIs in Android apps.
- We tested our system on 76 of the most popular apps on Google Play. Further investigation demonstrated that the vulnerabilities exposed in the discovered private web APIs are exploitable.

The rest of this paper is organized as follows. Section II introduces some background information as well as a motivation example. In Section III we present the details of our approach. Section IV shows the experimental results and findings with several case studies. Section V discusses the limitations of our approach. Section VI overviews the related work, and Section VII concludes the paper.

II. Background and A Motivating Example

In this section we first discuss some background knowledge of Web APIs and Android apps. Then we present a motivating example, which shows the differences in a web page version and an Android app of the same online video sharing service.

A. Background

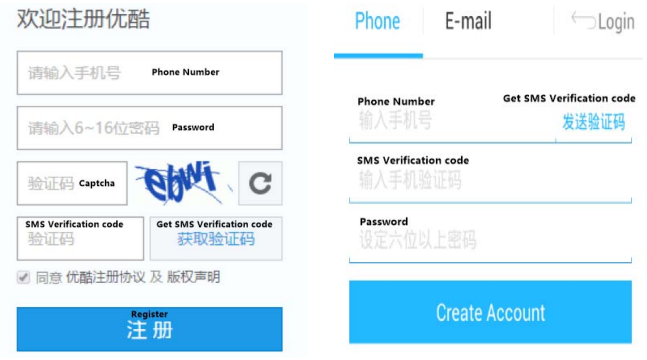
1) Web API: A server-side web API is a programmatic interface consisting of one or more publicly accessible endpoints exposed via the web. A web API is addressed by a Uniform Resource Identifier (URI). It can be called with a standard HTTP method, such as GET, PUT, or POST. It usually passes the return values back in JSON or XML format.

2) HTTP Clients in the Android Framework: An Android app needs to ask for the network permission `android.permission.INTERNET` to perform network operations. Android provides two basic HTTP clients for application developers to use, namely Apache HTTP client and `HttpURLConnection`. `DefaultHttpClient` and `AndroidHttpClient` are extended from the Apache HTTP client. Compared with Apache HTTP client, `HttpURLConnection` is more lightweight and efficient due to its use of transparent compression and response caching. Simple APIs and smaller size make it more popular in apps that target at Android 2.3 and higher versions.

3) AsyncTask: Network operations usually involve unpredictable delays, which will hang the residing thread. To avoid hanging the user interface (UI) thread and achieve a smooth user experience, the Android framework provides app developers with a dedicated class `AsycTask` to implement network operations. `AsycTask` encapsulates another thread to keep the UI thread separated. The virtual functions `onPreExecute`, `doInBackground` and `onPostExecute` define the operations that should be performed at different stages of the new thread. If the task is cancelled by invoking `cancel()`, `onCancelled` will be invoked instead of `onPostExecute` after `doInBackground`.

B. A Motivating Example

Youku is the most popular online video sharing service provider in China. Tapped as the Chinese Youtube, Youku is the 110th most visited website in the world according to the Alexa 2016 report[1]. The Youku Android app has also been downloaded more than 5 million of times according to Google Play[2].



(a) Web Page (b) Mobile App

Fig. 1: Screenshots of Youku Web Page and Mobile App. The web page version has a Captcha, whereas the mobile app version does not.

As shown in Figure 1, to register as a Youku member one is required to answer a Captcha in the web page version. However, this step is omitted if a user registers through the corresponding mobile app. By extracting and invoking the private web APIs in the mobile app, an attacker can easily bypass Captcha. Note that even though both the web and app versions require an SMS verification, this step, however, cannot tell if the registree is human or machine. Attackers can use mobile verification code platform APIs to automate the process of obtaining phone numbers and SMS verification code for registering zombie accounts.

Moreover, this private registration API can be used by attackers to send SMS spam. The APIs used in web pages to send SMS usually have IP and time limit to prevent users from continuously sending SMS. And the time interval between two consecutive requests increases very fast after multiple requests. However, the APIs used by mobile app are not properly implemented, which makes using them for spamming possible.

III. Our Approach

In this section we present our approach to automatically discover private web APIs via a hybrid analysis of the mobile apps. The discovered APIs would help security experts understand the functionality and working mechanism of an app and facilitate security auditing.

A. Challenges and System Overview

To recover the server-side web APIs, we need to discover the URLs and parameters of these APIs. Both static and

[1]http://www.alexa.com/siteinfo/youku.com
[2]https://play.google.com/store/apps/details?id=com.youku.phone

dynamic approaches are widely used in the analysis of Android apps [19], [15], [12]. Static analysis is usually faster than dynamic analysis. However, since API parameters and URLs are usually concatenated by several substrings, statically simulating the construction process is neither efficient nor accurate. Moreover, developers seldomly hard-code all the APIs in the mobile app, URL/parameters of some APIs are constructed during the execution of the app. Static analysis cannot deal with these cases. The limitation of static analysis drives us to resort to dynamic analysis.

However, using dynamic analysis we still face two challenges: 1) how to simulate the UI interaction with the mobile app; 2) how to capture the parameters of the web APIs during the interaction. Some researchers propose to intercept the network traffic to identify the broken SSL certificate verification using proxy [13], [19], [5]. However, if the network traffic is encrypted, the proxy cannot capture the original plaintext information.

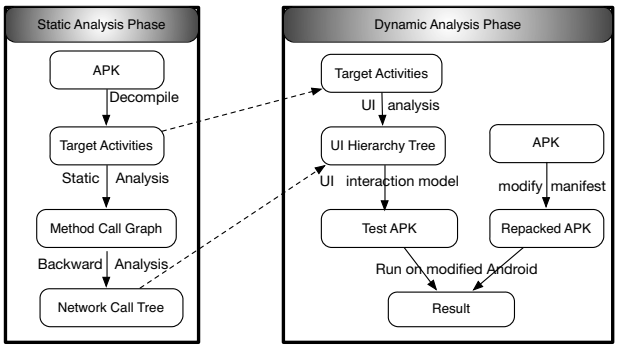


Fig. 2: System Overview

As shown in Figure 2, our system uses a combination of static and dynamic analysis. In the static analysis phase, our system has three steps: 1) identifying target activities that invoke web APIs; 2) building call graphs for target activities; 3) constructing network call tree. Then, our system repackages the APKs to make it possible for our system to trigger any target activity. In the dynamic analysis phase, our system generates test cases to trigger these activities dynamically and log the information of private web APIs. To solve the first challenge in dynamic analysis, we represent the UI state with the status of each widget and model the state transition. To cope with the second challenge, we choose to intercept URLs before they are sent out.

B. Identifying Target Activities

Our system first identifies activities that invoke web APIs. As discussed in section II, Android provides three types of HTTP clients, two of which are derived from Apache HTTP client and the other is `HttpURLConnection`. To discover the usage of these clients, our system decompiles an APK file into smali code and identifies the invocation of these HTTP client instances. Figure 3 shows the corresponding smali code snippets to make HTTP requests using the clients.

```
1  // AndroidHttpClient
2  Lorg/apache/http/client/HttpClient; ->execute(Lorg/apache/http/client/
       methods/HttpUriRequest;)Lorg/apache/http/HttpResponse
3
4  // DefaultHttpClient
5  Lorg/apache/http/impl/client/DefaultHttpClient; ->execute(Lorg/apache/http
       /client/methods/HttpUriRequest;)Lorg/apache/http/HttpResponse
6
7  // HttpURLConnection
8  Ljava/net/URL;->openConnection()Ljava/net/URLConnection
```

Fig. 3: HTTP Request Code Snippets for the Three Different Clients in Smali

Among these three clients, `HttpClient` is the built-in library in the Android system before Android 5.0 and `HttpURLConnection` is the recommended way of network connection by Google. They are widely used in Android apps. Even if some third-party network libraries encapsulate these clients (For example, Volley [10], an open-source network framework provided by Google, uses HttpClient in versions before Android 2.3 and uses HttpURLConnection in Android 2.3 and higher versions), our approach can still discover the usage of these clients.

C. Building Call Graphs for Target Activities

Then, our sytem builds a method call graph of each target activity. Our system looks for the `invoke` instructions in the smali code to find method invocations. If a method `A` explicitly invokes a method `B`, our system adds an edge from method `A` to `B` in the call graph. However, the Android framework uses callback mechanism to invoke app code that is not explicitly invoked by an app itself. Only dealing with explicit method invocations would fail to consider many user-defined event handlers, which are usually implemented in event-listener classes, and network operations, which usually are defined in an `AsyncTask` class. To cope with this challenge, our system also checks two types of implicit calls: 1) the ones that invoke methods in an Android event listener class; 2) the ones that use the `AsyncTask` class.

1) Event listener class: In Android apps, user-defined event handlers are registered by implementing the event listener class in an anonymous class or inner class and overriding the corresponding methods (e.g. `onClick`). In smali code, anonymous classes are also interpreted as inner classes. If the annotation of a class includes `.annotation system Ldalvik/annotation/InnerClass`, then this class is an inner class. The value follows `.annotation system Ldalvik/annotation/EnclosingMethod` is the method name declared in this inner class. Since inner classes usually implement Android EventListener interfaces, they override methods such as `onClick` and `onTouch` in these interfaces. For these cases, we add edges from the event listener method to the event handler method.

A line ending in the format `value=[class name]` indicates the usage of multi-threads. In this case, we will create a code block of a virtual method to link the `start()` and `run()` methods. After adding this block, multi-thread calls can be correctly identified and called.

```
1  MainActivity$1:
2  ...
3  # annotations
4  .annotation system Ldalvik/annotation/EnclosingMethod;
5  value = Lcom/example/urldemorunable/MainActivity;->onCreate(Landroid/os/
       Bundle;)V
6  .end annotation
7  ...
8  # virtual methods
9  .method public onClick(Landroid/view/View;)V
10 ...
11 invoke-direct {v1, p0}, Lcom/example/urldemorunable/MainActivity$1$1;-><init
       >(Lcom/example/urldemorunable/MainActivity$1;)V
12 ...
13
14 MainActivity$1$1:
15 ...
16 # annotations
17 .annotation system Ldalvik/annotation/EnclosingMethod;
18 value = Lcom/example/urldemorunable/MainActivity$1;->onClick(Landroid/view/
       View;)V
19 .end annotation
20 ...
21 # virtual methods
22 .method public run()V
23 ...
24 # invokes: Lcom/example/urldemorunable/MainActivity;->get(Ljava/lang/String
       ;)Ljava/lang/String;
25 invoke-static {v1}, Lcom/example/urldemorunable/MainActivity;->access$0(
       Ljava/lang/String;)Ljava/lang/String;
26 ...
27
28 MainActivity:
29 .method static synthetic access$0(Ljava/lang/String;)Ljava/lang/String;
30 ...
31 invoke-static {p0}, Lcom/example/urldemorunable/MainActivity;->get(Ljava/
       lang/String;)Ljava/lang/String;
32 ...
33 .end method
34
35 .method private static get(Ljava/lang/String;)Ljava/lang/String;
36 ...
37 invoke-interface {v1, v6}, Lorg/apache/http/client/HttpClient;->execute(Lorg
       /apache/http/client/methods/HttpUriRequest;)Lorg/apache/http/
       HttpResponse;
38 ...
```

Fig. 4: Code Snippets in Smali

2) AsyncTask:: As mentioned in section II, developers often use `AsyncTask` to perform network operations in the background. In smali code, the UI thread will call `execute(Ljava/lang/Object;)` to start a new background task. In this case, we add an edge to the `doInBackground` method of the `AsyncTask` subclass.

D. Constructing the Network Call Tree

Our system uses a *network call tree* to represent the invocation sequence of methods starting from the entry point until the triggering of HTTP requests. Once our system builds the call graph for each target activity, it constructs network call trees via backward reachability analysis on the call graph, starting from the methods shown in Figure 3. The backward analysis continues until the entry point of the activity (e.g. `onCreate`) is reached. The constructed network call tree consists of all the intermediate methods that would be executed from the entry point to fire an HTTP request.

Our system denotes the widgets that have registered event listeners as *executable widgets* and keeps a record list of them. If our system encounters an event listener method during the traversal, it records the corresponding executable widget in the network call tree. Our system leverages this information to verify whether the exploration paths have clicked all the executable widgets that would lead to HTTP requests during UI interaction emulation. Figure 5 shows an example of a network call tree that is generated from the code snippets shown in Figure 4.

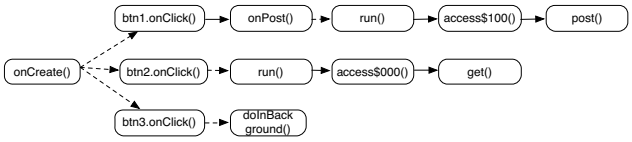


Fig. 5: The Constructed Network Call Tree from Smali code shown in Figure 4

E. Repackaging the APK

In the dynamic analysis stage, our system runs the targeted activities and simulates user-interactions with it. To make sure that all HTTP requests are triggered, our system modifies the manifest file of the original APK and adds `android.intent.action.MAIN` to the attribute of each target activity so that it can be triggered externally. Our system also sets `debuggable=true` to enable debugging. Then, our system repacks the APK using the modified manifest file.

F. UI Interaction Model

Android UI interaction has been widely explored to enable automatic testing [23], [1], [14]. Testing tools can generate events by considering Android apps as either a *black box* or *white box* [7]. To combine the advantages of black and white box approaches, we adopt a *grey box* approach as in [23]. We perform a user interface analysis to extract the UI widgets of each target activity, after which our system uses these properties to simulate UI interactions.

1) User Widgets Extraction: Android applications are event-driven and many important events occur through UI interactions. Therefore for each target activity, our system first analyzes its UI to retrieve its UI elements. To capture both statically and dynamcially generated UI elements, our system analyzes each activity at runtime. We use UIAutomator to dump the UI hierarchy. The dumped information is stored in an XML file of the same name with the activity class. The given activity's UI elements are included in the XML file hierarchically, usually stemming from container views such as LinearLayout. The widges of our interest are those defined by developers, i.e., whose resource-id starts with the activity name (eg. `resource-id="com.example.click:id/button1"`) instead of `android`. We further find the mapping of the widgets in the XML file with that in the static analysis to get the list of *executable widgets* and those would lead to HTTP requests.

2) UI Interaction Simulation: Our approach involves the following steps:

- *Inferring Input Type:* Our system generates test data based on the data type of the attributes of a widget. If widget `hint` is present, we also use hint to predicate the data type. `EditText` usually follows a `TextView` which tells the user what to input.
- *UI State Representation and Transition:* We represent each UI state using the status of each of the *executable*

Algorithm 1 Automated UI Interaction

Require:
Converted JSON file
The list of executable widgets L
Current Activity name N
Ensure: Tranverse all UI status to trigger HTTP requests
function UI INTERACTION
for all $w \in L$ **do**
$w.status = false$
end for
Fill EditText with inferred data
while do $\exists\ w\ st.\ w.status = false$
Click w to trigger its event listener method
$w.status = true$
Get the current activity name N'
if $N! = N'$ **then**
Trigger a back button event
end if
end while
end function

widgets. Each widget that is executable has two status: *true* for clicked or *false* for not clicked. Initially each widget is not clicked. Since we have inferred the data type to put in each first tier widget, we fill it in each time before we begin the emulation. Then we click each *executable widget* to trigger its event listener method. After that we change the status of the widget to be *true* and retrieve the current activity name to check whether it has been transited to another activity. If the control has flows to another activity, a back button event is triggered to go back to the tested activity. The simulation continues until the status of all executable widgets has been set to *true*. The algorithm used to emulate UI interaction is given in algorithm 1. Following this model, we get the test application and pack it to an APK file.

G. Execution on Our Customized Android

The last step is to execute the repackaged APK on our customized Android system. We modified Android to enable automatic logging of web-connection information. This is achieved by hooking the methods that are triggered during an HTTP request. We hooked these methods by adding Loggers to log information such as URI, request headers and body, response headers and body. Compared with other approaches such as using proxies to intercept the network traffic, our approach has several advantages. Firstly, we capture the HTTP request data before the request is sent out. At this time the parameters has been completely-constructed but has not been disguised. Other tools such as Burp Suite [21] intercepts network traffic after the HTTP request is made. At this time the request data may have been split or encrypted (e.g. in the case of VPN). Secondly, we do not need to forge fake certificates to launch MITM SSL attacks in order to get the network traffic. Thirdly, some apps (e.g. com.creditkarma.mobile) may stop working in detection of the proxy.

Since we focus on HTTP requests, we hooked the methods related to `HttpClient` and `HttpURLConnection`. For HttpClient, we mainly trace the methods *httpclient.execute()* and *defaulthttpclient.execute()*. For HttpURLConnection, we trace the method *URL.openConnection()* and its parameters. We log the traced methods and parameters sequentially following the UI interaction order.

IV. EXPERIMENTS AND FINDINGS

We selected 76 out of the 120 most popular apps on the Google Play market as our test set. We did not deal with apps in the Games category since they usually use a different framework for network connection. Our automated system successfully run 48 apps and recovered many APIs from more than 30 apps. Other apps fail due to decompilation or repack errors since they have imported self-defined frameworks. We further investigated the recovered APIs manually and found that 9 apps are possessed of vulnerabilities, such as API misuse and session hijacking. Next we will present some examples of the vulnerabilities we have found. For better illustration, we also compare the web APIs of mobile apps with that of the browser-based applications.

A. API misuse

Nowadays, with the huge impact of security issues, many applications already take some measures to protect their APIs. For example, they usually use HTTPS instead of HTTP and use signatures to ensure the integrity of the parameters. From our experiment results, we found that apps such as *TextNow, Flipagram, Smule* and *lionmobi:battery* use signatures to protect their parameters. Some of them also use HTTPS in all APIs. *Dubsmash* even uses HTTPS+MAC to protect its information.

However, we also found some apps fail to protect their APIs, which would lead to API misuse. Next we summarize the problems we found and classify them by their functionality.

Registration API: Zombie account is always a threat for network companies. So they usually use verification to prevent machine registration. But some enterprises removed verification measures such as captcha and authentication code for ease of use in their mobile apps. Besides, although some mobile apps require the email address for verification, they do not wait for the user to click on the received verification link before proceeding with other user requests. We found two apps, *Wallapop* and *Flipagram*, have weakness in protecting their sign up APIs. Although both of them use HTTPS to protect the network data in transmission, they cannot sign the parameters, which make replay attacks possible. Besides, neither of them verify the escheatage of the email address. In the API of *Wallapop*, we just changed the email key of the post parameter to successfully create a zombie account. For *Flipagram*, we need to change the email and username keys of the parameter.

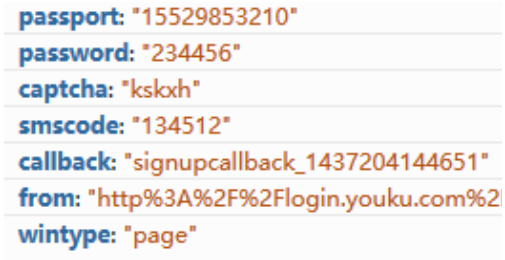


Fig. 6: Registration parameter of the web application

Since the mobile app and browser app have different interfaces, mismatch may occur when updating the app. We mentioned in section II-B that the registration API of the mobile app Youku does not provide captcha verification. We further found that while the browser web page forbids users to register a new account with email address (Figure 1a) to prevent zombie accounts, the mobile app still allows user registration with email address (Figure 1b).

The registration API's URL and parameters are constructed in the *getRegistURL* method in the file *com.youku.phone/smali/com/youku/http/URLContainer.smali* in the Activity *com.youku.ui.activity.HomePageActivity*. By comparing it with the parameters of browser-based app (Figure 6), we found that captcha has been removed in the mobile API. Leverage the API information, we wrote a Python script and successfuly registrated a bunch of accounts by only providing fake email addresses and passwords.

Login API: For the login API, a secure application should limit the times to input the wrong password to avoid brute force attack on the password. But some mobile apps fail to do so. From the retrieved login API of Youku (Figure 7), we found that no captcha is presented during the login process. We further looked into the network call trees of the login API and investigated its smali code. In combination with the parameters of the registration API, we can easily infer the meaning of each parameter. After our test we also found that the mobile login API does not have restriction on the access frequency from the same source IP address. In this case we can do brute force attack by trying each of the <username, password> pair in a given database (e.g. leaked from other websites).

API to Send SMS Verification Code: In our experiment, we found that many mobile applications send SMS verification code to verify a user's identity. But they fail to protect the APIs to send the SMS messages, which enables attackers to abuse these APIs and send messges to users. To give an example, we wrote a python script to send messages making use of the exposed API in *JuanPi* (an online shopping app). Their API forbids the same phone number from receiving messages within 60s. We set a mobile number pool that contains one hundred phone numbers from 182****2200 to 182****2300. The script will automatically select a number from the pool in order, fill it as the mobile parameter of the API and send it to the server. The successfully sent messages are shown in Figure 8.

POST request to /user/passport/login

Type	Name	
URL	pid	0865e0628a79dfbb
URL	guid	d9d425ab4f240e3a60e39a769107a818
URL	mac	08:00:27:3f:cf:3d
URL	imei	000000000000000
URL	ver	4.6
URL	_t_	1437491615
URL	e	md5
URL	_s_	7d9cd6becb1f72871636fb726478b695
URL	operator	Android_310260
URL	network	WIFI
URL	uname	abc@xyz.com
URL	pwd	39d8f4781b6146372038cb956e2360b8

Fig. 7: Youku login API

This kind of attack not only disturbs users (phone numbers from 182****2200 to 182****2300 will constantly receive SMS), but also leads to exta cost on the SMS sending platform which is paid by the enterprise.

API to check account existence: When a new account is created, the mobile app usually needs to check whether an account with the same email address already exists. When a user forgets his password and uses the email address to find it, the app also needs to check whether the given email corresponds to an existing account. Our investigation shows that many apps fail to protect the API to check the existence of an account. For example, the "forget password" API of the *Weather Channel* app and the "check email availability" API of the *Flipagram* app can be used to test whether a given account exists for the app. Advertisers can use these APIs to test whether an email address is active and send advertisements to them specifically.

API to fetch content: Many applications provide useful information to gain a large number of users so that they can benefit from the pay of users or the ad fee. The *Weather Channel* app provides the weather information to attract users so that they can benefit from Google ads. But from our analysis, we found its core APIs to fetch the weather info is open to public. We can fetch the weather information everywhere around the world by just changing the "geocode", a parameter defined by Google Map, without opening the app! As another example, *Zedge* is used to download wallpaper for different mobile phones. But from its private APIs, we can easily get the detailed download addresses of the wallpaper and download it in batch automatically. Similarly, the "list GIF picture" API of the *Cute Emoji Keyboard* app is ready to be invoked by external apps to fetch the GIF picture info in JSON format.

B. Session hijacking

To better understand the behavior of mobile apps after user login, we manually login in some apps by filling in correct username and password before running them on our automatic analysis system. From our experiment, we found that mobile

Fig. 8: Successfully sent message by SMS verification API

apps either store their session info in the cookie or in the request parameters.

When we analyze the *Youku* app, we found many APIs with a special cookies parameter named "yktk". So we checked the browser application of Youku and confirmed that it is a special string which is used to keep sessions. We further found from the logged web APIs that the "yktk" value is saved in a special class. With this information we designed two kinds of session hijacking attacks. The first one is to copy the value of the "yktk" parameter to the browser app's cookie and get logged in successfully. The second one is to insert this value to the sharedpreferences files (located at /data/data/-com.youku.phone/shared_prefs/com.youku.phone_preferences .xml). When we restart the application we found that we successfully logged in the same account. The same attack also applies to the "BDUSS" cookie keyword of the *Baidu* app.

As another way of protection, many applicaitons keep session in the request parameters. Some apps use signatures to protect the session info securely, eg, *TextNow* use HTTPS and parameter signatures to protect the session info. However some apps, for example *Smule:Sing!*, use HTTPs for sign up and login but use HTTP for other services. In this case, the session parameters can be hijacked by storing the value of */data/-data/com.smule.singandroid/shared_prefs/network.xml* on the attacked phone. Another app, *Flipagram*, stores the session info in the HTTP request headers, which is not signed and enables session hijacking.

V. Limitations

Our approach adopts a hybrid strategy that combines static and dynamic analysis. In the static analysis part, we filter target activities by looking for smali code in figure 3. Therefore we cannot deal with self-defined network frameworks that are not based on HttpClient or HttpURLConnection. Besides, not all network connection uses HTTP protocol. For example, push services usually use TCP to keep connect with the server to receive messages immediately. Some game or chat services also use the TCP protocol instead of HTTP protocol to decrease delay. Our work only focuses on HTTP requests and cannot deal with these cases.

In the dynamic analysis part, to make sure that all target activities can be launched, we modified the manifest file to make these activities, exported and as the launcher activity. However, during runtime some activities may be launched by other activities with specified intents. Besides, some activities may not be launched unless the user provides correct information such as user name and password. We cannot simulate such cases.

VI. Related Work

A. Mobile App Vulnerability Discovery

In the past several years, a considerate amount of efforts has focussed on discovering various vulnerabilities in mobile apps. For instance, TaintDroid [12] detects privacy leakage vulnerability by tracking information flows. PiOS [11] uses static analysis to detect such leaks in iOS apps. CHEX [15] detects component hajacking vulnerabilities in Android apps by using a data-flow based static analysis approach. FlowDroid [3] proposes a highly precise and Android lifecycle-aware taint analysis for Android applications. Similar works include [22], [8], [9]. [16] discussed a number of attacks on WebView, including JavaScript injection and event sniffing and hijacking. They also showed the wide usage of webview hooks and the `addJavascriptInterface` API, which indicates the damage of possbile attacks on WebView. SMV-Hunter [19] detects man-in-the-middle SSL/TLS vulnerabilities with a hybrid static and dynamic analysis. However, few efforts have focused on identifying the vulnerabilities in the apps server side. AUTOSIGN [6] made such a step in this direction and demonstrated that there are also serious security vulnerabilities such as password brute forcing if app server developers do not perform the necessary security checks.

B. Automated UI Interaction

To enable automatic testing of Android apps, automatic UI interaction is a necessity. [7] gives an overview of the automated test input generation approaches and performs a thorough comparison of them in four dimensions including their code coverage, fault detection, ability to work on multiple platforms and ease of use.

Events (including UI events and system events) are generated following either a random or systematic exploration strategy. Random test input generators [17], [20] can efficiently generate events, therefore they are suitable for stress testing. But they usually generate redundant events since they are not aware of how much behavior has been covered. Among the works that adopt a systematic strategy, some follow a model of finite state machine with activities as states and events as transitions [1], [23]. Model-based exploration strategy leads

to more effective results. But these tools may miss the events that would not result in GUI changes as they usually represent new states when there are UI changes. Other tools use more sophisticate techniques such as symbolic execution and evolutionary algorithms to guide the exploration for a better coverage [18], [2].

VII. Conclusion

Given the popularity of mobile apps, it is imperative to check if the server-side APIs they use provide the same level of security checks as their public counterparts. The first step to achieve this is to automatically discover these undocumented web APIs. In this paper, we designed and implemented a system that can automate the process of discovering private web APIs. We tested our system on 76 out of the 120 most popular apps on Google Play and successfully recovered many APIs. Our experiments have demonstrated that the vulnerabilities disclosed by the private server-side APIs can be exploited.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their comments on previous drafts of this paper. This work is partially supported by National Natural Science Foundation of China (91546203, 61173068, 61572295, 61573212) , Program for New Century Excellent Talents in University of the Ministry of Education, the Key Science Technology Project of Shandong Province (2014GGD01063, 2015GGE27033), the Independent Innovation Foundation of Shandong Province (2014CGZH1106) and the Shandong Provincial Natural Science Foundation (ZR2014FM020).

References

[1] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (2012), ACM, pp. 258–261.

[2] Anand, S., Naik, M., Harrold, M. J., and Yang, H. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 59.

[3] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 259–269.

[4] Bosomworth, D. Mobile Marketing Statistics compilation. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/, 2015.

[5] Cai, F., Chen, H., Wu, Y., and Zhang, Y. Appcracker: Widespread vulnerabilities in user and session authentication in mobile apps. In *Proceedings of Mobile Security Technologies Workshop (MoST)* (2015), IEEE.

[6] Chaoshun, Z., Wubing, W., Rui, W., and Zhiqiang, L. Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services. In *Proceedings of the 23rd ISOC Network and Distributed System Security Symposium (NDSS)* (2016).

[7] Choudhary, S. R., Gorla, A., and Orso, A. Automated Test Input Generation for Android: Are We There Yet? *arXiv preprint arXiv:1503.07217* (2015).

[8] Cui, X., Wang, J., Hui, L. C., Xie, Z., Zeng, T., and Yiu, S. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2015), ACM, p. 25.

[9] Cui, X., Yu, D., Chan, P., Hui, L. C., Yiu, S.-M., and Qing, S. Cochecker: Detecting capability and sensitive data leaks from component chains in android. In *Information Security and Privacy* (2014), Springer, pp. 446–453.

[10] developers, A. Transmitting Network Data Using Volley. http://developer.android.com/training/volley/index.html.

[11] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS), Internet Society. USA* (2011).

[12] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), pp. 1–6.

[13] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. 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* (2012), ACM, pp. 50–61.

[14] Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services* (2014), ACM, pp. 204–217.

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

[16] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 343–352.

[17] Machiry, A., Tahiliani, R., and Naik, M. Dynodroid: An input generation system for Android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering* (2013), ACM, pp. 224–234.

[18] Mahmood, R., Mirzaei, N., and Malek, S. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 599–609.

[19] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. 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)* (2014), Citeseer.

[20] testing tool, T. M. U. A. . http://developer.android.com/tools/help/monkey.html.

[21] web security, P. Burp Suite. https://portswigger.net/burp/.

[22] Wei, F., Roy, S., Ou, X., et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1329–1341.

[23] Yang, W., Prasad, M. R., and Xie, T. A grey-box approach for automated GUI-model generation of mobile applications. In *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.