AuthScope: Towards Automatic Discovery of Vulnerable Authorizations in Online Services

Chaoshun Zuo
The University of Texas at Dallas
800 W Campbell Rd
Richardson, Texas 75080
chaoshun.zuo@utdallas.edu

Qingchuan Zhao
The University of Texas at Dallas
800 W Campbell Rd
Richardson, Texas 75080
qingchuan.zhao@utdallas.edu

Zhiqiang Lin
The University of Texas at Dallas
800 W Campbell Rd
Richardson, Texas 75080
zhiqiang.lin@utdallas.edu

ABSTRACT

When accessing online private resources (e.g., user profiles, photos, shopping carts) from a client (e.g., a desktop web-browser or a mobile app), the service providers must implement proper access control, which typically involves both authentication and authorization. However, not all of the service providers follow the best practice, resulting in various access control vulnerabilities. To understand such a threat in a large scale, and identify the vulnerable access control implementations in online services, this paper introduces AuthScope, a tool that is able to automatically execute a mobile app and pinpoint the vulnerable access control implementations, particularly the vulnerable authorizations, in the corresponding online service. The key idea is to use differential traffic analysis to recognize the protocol fields and then automatically substitute the fields and observe the server response. One of the key challenges for a large scale study lies in how to obtain the post-authentication request-and-response messages for a given app. We have thus developed a targeted dynamic activity explorer to perform an in-context analysis and drive the app execution to automatically log in the service. We have tested AuthScope with 4,838 popular mobile apps from Google Play, and identified 597 0-day vulnerable authorizations that map to 306 apps.

CCS CONCEPTS

•**Security and privacy** →**Access control; Authorization;** *Web application security;*

KEYWORDS

Access control; authorization; vulnerability discovery

1 INTRODUCTION

For any multi-user computing systems (e.g., online shopping and social networking), there is a need to regulate who can view or use a resource. A particular security mechanism to achieve this is to use access control, in which a user needs to be first authenticated (i.e., telling the system who the user is) and then the access is granted if the authenticated user has the permission to do so. The use of access control can be dated back to Multics Operating Systems [32], where a user first logs in the system to acquire a user ID (UID) via a password-based authentication, and then the kernel checks the UID when the user requests access to a protected resource based on the corresponding permissions. Nearly all of the later multi-user operating systems (e.g., UNIX/Linux) have followed such an approach when implementing their access control mechanisms.

When moving to the online services, designing and implementing a secure access control mechanism becomes a challenging task for several reasons. First, an online service can have up to hundreds of millions (even billions) of users, and handling such a large scale of users often needs to use efficient database technologies. Second, managing the user credential correctly for authentication is another challenge (e.g., many online service today still mistakenly store plaintext password [3, 12]). Third, the client side (e.g., a browser, a mobile app) can be completely controlled by an attacker and cannot be trusted at all. That is, a request message generated by a client can be untrusted, and the efficient security check is needed at the server side [47].

While the use of single-sign-on (e.g., with Facebook Login) [38] has made the authentication management much easier for online services, it does not solve the authorization problem automatically in that the online service provider (e.g., shopping sites such as Amazon) still has to regulate that the authenticated user only views and updates her own resources (e.g., her user profile or shopping cart). Over the past many years, an efficient approach of using security tokens to handle authorization was developed [20], and popularized especially in web applications. In particular, in traditional desktop web applications, a browser cookie or a session ID (these are often called security tokens) is used for the authorization.

Consequently, the security of the authorization depends on how strong the token is (and also whether the server enforces it). Any disclosure, capture, prediction, brute force, or fixation of the security tokens will lead to severe attacks such as account hijacking, where an attacker is able to fully impersonate a victim to get all of her personal data. Unfortunately, not all of the online service providers follow the best practice when using the security tokens for the authorization. For instance, we have observed weak security tokens (e.g., just a very small integer) passing through mobile apps. Meanwhile, we have also observed that even though a service provider may have used strong security tokens (e.g., a 256-bit cryptographic hash), the server actually does not enforce whether this token belongs to a particular user (i.e., the token) or it is just a token. Given the fact that so many mobile apps used in our daily lives, it is imperative to systematically identify these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
© 2017 ACM. ISBN 978-1-4503-4946-8/17/10...$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134089

vulnerable access control servers, otherwise users' personal data can be thus leaked.

To this end, this paper introduces AuthScope, a tool to automatically identify the vulnerable access control servers, especially the vulnerable authorizations, when given just mobile apps. Since we do not have any source code of the sever side implementations, we can only perform a blackbox analysis of the remote server by generating and analyzing various network request and response messages between the client and the server. To pinpoint the vulnerable access control implementations, our key insight is to use differential traffic analysis, a widely used network protocol analysis technique (e.g., [13, 16, 17, 40, 47]), to recognize the network protocol fields and then automatically substitute the fields of interest and observe the server response to identify the vulnerable services. One of the key challenges for a large scale study lies in how to obtain the post-authentication request-and-response message pairs for a given app. We have thus developed an adaptive dynamic app activity explorer to perform an in-context analysis and drive the app execution to automatically log in the service.

We have implemented AuthScope and tested it with 4,838 very popular mobile apps from Google Play. Note that these apps all contain Facebook login and they belong to the top 10% of the mobile apps in terms of the accumulated downloads in Google Play. To our surprise, AuthScope has identified 597 0-day vulnerable access control implementations in the server side of 306 mobile apps (with an upper bound of total install of 61 million). The root cause of these vulnerabilities comes from the mistaken use of either predictable IDs, or user's email address, or user's Facebook ID for the authorization without (or enforcing) any security tokens. Consequently, these online services can all be completely broken by an adversary, and privacy sensitive or even secret data for up to 61 million mobile users can be leaked due to these vulnerabilities.

In short, we make the following contributions in this paper.

- **Novel System**. We present AuthScope, a novel tool to automatically identify the vulnerable access control on the server side. It does not require any code access of server's implementation, other than just the traffic between an authenticated user and the server.
- **Efficient Techniques**. We apply differential traffic analysis to automatically reverse engineer protocol fields of interest, such as security tokens, and also we develop an adaptive app activity exploration scheme to execute a mobile app in a targeted way and apply it to trigger post-authentication request messages.
- **Practical Results**. We have tested AuthScope with 4,838 popular Android apps. Our tool has identified 597 0-day vulnerable access control implementations among the remote servers of 306 mobile apps. We have made responsible disclosure to all of the vulnerable service providers.

2 BACKGROUND

In this section, we present necessary background in order to understand the common mistakes and root causes of vulnerable access control implementations in online services. We begin with the basic concepts of authentication and authorization in §2.1, and then examine why the authorization in UNIX/Linux is secure in §2.2. Finally, we discuss how a typical secure authorization in an online service is implemented and its practical security issues in §2.3.

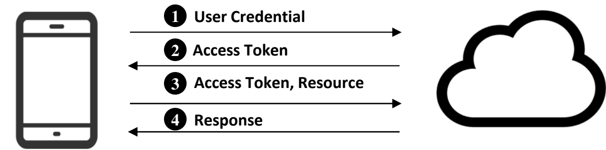

Figure 1: A Simplified Authentication and Authorization Protocol in Online Services.

2.1 Authentication and Authorization

When providing private resources to multiple users, it often requires two security services: authentication and authorization.

- **Authentication**. The process of verifying a user's identity is called authentication. In a multi-user system, it is crucial to accurately identify who makes the request. A widely used approach to perform authentication is to use a password system in which a user's identity is verified by checking with a hashed password typed during the login. Also, authentication typically only needs to be performed once; otherwise it will be annoying to the user.
- **Authorization**. The process of granting the access of specific resources based on user's privileges or permissions is called authorization. Not that authentication provides the proof of identity, but it does not describe the resources that are allowed to be accessed by the authenticated user. For instance, a user is authenticated before accessing a database, but this does not tell the database system which data the user is entitled to access. For this, it requires the authorization service.

2.2 Authorization Security in UNIX/Linux

For a multi-user operating system such as UNIX/Linux, right after an authenticated user logs in, the system will automatically assign a UID (which is just an integer) based on the profile in the system (e.g., /etc/password) and create a shell process to serve the user's request. This shell process will interact with the system on behalf of the user with the assigned UID that is maintained by the process descriptor in the kernel. To change the UID of a process or a user, it must invoke system calls.

More specifically, to ensure the security, any access to a resource needs to invoke system calls, in which access control is enforced based on the UID and permissions. An adversary cannot forge his or her UID to someone else's even though the UID is known, because the kernel remembers the UID and tracks it at the corresponding process descriptor. For an adversary to really change the UID, he or she must exploit software vulnerabilities in the system such as buffer overflows in a daemon process. When the user logs out, the shell process also terminates and the user has to be authenticated again in order to use the system.

2.3 Secure Authorization in Online Service

As stated early, there are many challenges (e.g., a large volume of users, untrusted client, etc.) when implementing a secure authorization in online services. More importantly, online services often require high scalability and availability. Unlike the UNIX/Linux authentication and authorization, which is stateful and kernel remembers who has logged in and out, the majority of today's online service uses HTTP/HTTPS protocol, which is stateless. A stateless protocol can be forced to behave as if it were stateful if the server and the client can send the state along with every request and response message. A typical way of accomplishing this in HTTP/HTTPS is to use security tokens such as cookies or session IDs. As illustrated in Figure 1, at a high level, the authentication and authorization protocols in on-line services can be abstracted using the following four steps:

- **Step ❶**: The client sends a request to the server with user credentials such as a password. The server authenticates the identity of the user via the password, social single sign on (e.g., Facebook Connect), or other means. To prevent any leakage of the credentials, transport layer security (TLS) is often used to ensure the communication security.
- **Step ❷**: The server creates a randomly generated token and binds it with the authenticated user, and the server then transmits the token back to the client.
- **Step ❸**: The client includes the server provided token on subsequent requests to the server as a proof of identity, the server then grants or rejects the user access to protected resources based on her permissions.
- **Step ❹**: The server responds to the user request with the appropriate information.

Sometimes, there are even more simplified implementation of the protocol and the client does not need to complete the first three steps to access a resource stored in the server, if the client knows (e.g., distributed via an email first) the resource ID (RID in short) and this RID is sufficiently random. For instance, when using Overleaf, an online Collaborative Writing and Publishing service, to share a paper repository, the user could just sent the URLs (e.g., https://www.overleaf.com/9357323vdzpzwzmwdmx) generated by Overleaf. Only the recipient who has the URL can access the paper repository because the RID (e.g., 9357323vdzpzwzmwdmx, which is essentially a token) is sufficiently random. Also, in this case, the server does not have to remember who holds the RID: anyone who has it can access the repository. That is, the binding of the token to a user is performed separately (e.g., managed by the user not by the server).

Practical Issues. According to the above discussion, we can notice that proper generation and use of security tokens (or RIDs) is paramount to ensure the authorization security. In theory, because the token (or RID) is generated at the time of login (or creation) and is random and unguessable, its presence sufficiently serves as proof that the request really comes from the authenticated user to whom the token (or RID) was assigned. In reality, however, we believe not all developers would have followed such best practice, and there will be many poorly engineered servers (as those poorly engineered mobile apps suffered from various vulnerabilities identified in the past few years such as component hijacking [24], information leakage [9]), and privilege escalation [44]).

More specifically, there will be a variety of issues when implementing the authorization security on the server side. For instance, is the security token (or RID) sufficiently random? Has the server really enforced the check of the security tokens? Even though the server checked the token, has it really made sure it is the token binded to a particular user or it is just a token (a token vs. the token)? Does the user have the permissions to access the protected resources? With these questions in mind, we would like to perform a large scale, systematic study of how online service providers implement their access control for mobile users, and identify those vulnerable ones if there is any.

3 OVERVIEW

The goal of this work is to understand how online service providers implement their access control of user resources, and identify those servers that are vulnerable to account hijacking and private information leakage, by just analyzing the traffic between the mobile apps and the server. While there are a variety of ways to do so, we seek to design an approach that is scalable, automated, and systematic. In this section, we first use a running example (§3.1) to discuss various challenges (§3.2) we have to solve, and then give an overview of our system (§3.3).

3.1 A Running Example

To illustrate the problem clearly, we use a running example from a popular social app named *W*[1] that manages users' pets (e.g., dogs) and track their activities. *W* app is very interesting in that it actually contains a vulnerable access control implementation even though it uses strong security tokens for user authorization.

In particular, as illustrated in Figure 2, right after a legitimate user logs in the app, the *W* client will automatically send a request to the server to get all the notification messages (which are private resources belonging to this particular user). For each specific notification message, the server will assign an RID (e.g., 433222 and 433227) and send the response message containing the RID to the client, as shown in Figure 2(a) and Figure 2(b), the request and response message pairs for user Alice and Bob we registered with the service, respectively.

We can observe that the server of *W* does use a user specific random string as the security token (i.e., as shown in the in_app_token field). The server also assigns two integers, namely 21690 as Alice's user ID (UID in short) and 21691 as Bob's UID. Unfortunately, if we substitute the UID in the post-authentication request message of Alice with the value from Bob's, e.g., replacing 21690 with 21691, we can successfully read Bob's private notification message by using Alice's token as shown in Figure 3.

Therefore, as can be noticed, the server of *W* has made either of the two following mistakes:

- **No enforcement of the security token whether or not belonging to a particular user**. If the server has checked UID 21690 with Alice's token and 21691 with Bob's token, the substitution attack would not have succeeded.

[1] Note that we do not report the concrete name of this app, since the vulnerability identified in the server of *W* has not been patched yet as the time of this writing.

```
GET /api/v1//users/21690/notifications?in_app_token=e67315b35aa3
8d4ac8cac3cd9c7f88ae7f576d373f HTTP/1.1
Host: api.*****.com
Connection: close
```

```
HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Content-Type: application/json
ETag: W/"5319d96924bb6d0a761b5f13b248919c"
Server: nginx/1.6.2
X-Request-Id: 5775d45e-cc3b-4665-8bc6-c2c7a2c9180d
X-Runtime: 0.027840
Content-Length: 191
Connection: Close

[{"id":433222,"sender":null,"dog":null,"notification_type":15,"n
otification_text":"Welcome to *****.","object_id":21690,"is_seen
":true,"is_read":true,"created_at":"2017-01-28T23:54:59.831Z"}]
```

(a) Alice's first request and response message after login

```
GET /api/v1//users/21691/notifications?in_app_token=fb153b7d8c0a
0c6ac841d7bfbd9446de627c642858 HTTP/1.1
Host: api.*****.com
Connection: close
```

```
HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Content-Type: application/json
ETag: W/"6ee365b32e7f3e145d5c74778ea243cd"
Server: nginx/1.6.2
X-Request-Id: 4970cafb-9438-4a70-96e0-ca2f789f0d5d
X-Runtime: 0.022889
Content-Length: 192
Connection: Close

[{"id":433227,"sender":null,"dog":null,"notification_type":15,"n
otification_text":"Welcome to *****.","object_id":21691,"is_seen
":true,"is_read":false,"created_at":"2017-01-28T23:56:40.533Z"}]
```

(b) Bob's first request and response message after login

Figure 2: Sample Request and Response Messages of our Running Example. The server name has been anonymized with ***.**

```
GET /api/v1//users/21691/notifications?in_app_token=e67315b35aa3
8d4ac8cac3cd9c7f88ae7f576d373f HTTP/1.1
Host: api.*****.com
Connection: close
```

```
HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Content-Type: application/json
ETag: W/"6ee365b32e7f3e145d5c74778ea243cd"
Server: nginx/1.6.2
X-Request-Id: 4970cafb-9438-4a70-96e0-ca2f789f0d5d
X-Runtime: 0.022889
Content-Length: 192
Connection: Close

[{"id":433227,"sender":null,"dog":null,"notification_type":15,"n
otification_text":"Welcome to *****.","object_id":21691,"is_seen
":true,"is_read":false,"created_at":"2017-01-28T23:56:40.533Z"}]
```

Figure 3: Alice Read Bob's Private Message.

- **No randomness of the** UID. If the server does not attempt to enforce the consistency check between the UID and the corresponding user token, it can make the UID sufficiently random and attacker cannot make a predictable guess, thereby defeating the substitution attack.

The goal of our AuthScope is exactly designed to identify these vulnerable servers automatically and in a large scale, by performing the request message field inference and substitution systematically.

3.2 Challenges and Key Insights

From the above running example, we can notice that there will be a number of challenges in order to achieve our goal and these include:

- **How to obtain the post-authentication messages**. Since we focus on the identification of the vulnerable authorization implementations (which occur after the user authentication), we must execute the app to reach the state that generates the post-authentication request messages. In other words, we must have a registered legitimate user of the testing service and obtain a legal post-authentication message. While we can use manual efforts to register a legal user in each of the to-be-tested service, this cannot scale to a large volume of apps. This also contradicts our goal of fully automation. Therefore, we have to design techniques to drive the app execution to trigger the legitimate post-authentication messages (e.g., the ones illustrated in Figure 2).
- **How to recognize the protocol fields of interest**. With the traced legitimate request and response messages, we have to also identify the fields that are of our interest. For instance, as shown in Figure 2, we have to recognize various fields such as in_app_token in which there are field-name associated, and those that do not have any field-name (e.g., 21690 and 21691 in the URL path though we suspect it is a UID field) in the messages. Note that unlike traditional HTTP request message in which we can directly recognize the protocol fields by field names, we have to systematically recognize all of the protocol fields including field-name hidden ones used in URLs such as those using REST APIs.
- **How to identify the vulnerability**. Having obtained the post-authentication messages and recognized the protocol fields, we still need to systematically substitute the protocol fields in the request messages to observe how a server would respond to the substituted request messages. How to decide whether a server is vulnerable based on the response message is another challenge.

Fortunately, all of the challenges listed above can be solved or partially solved with the following key insights.

- **Executing the app with single-sign-on**. It is tedious to manually register a user account one by one for each of the tested mobile app. Interestingly, we notice that many of the mobile apps today support social login such as using Facebook login. With this, we can automatically log in an app to exercise the post-authentication messages if we are able to drive the app to execute the Facebook login. The limitation for this approach is for those that do not use social login we will not be able to test them automatically.
- **Recognizing protocol fields of interest with differential traffic analysis**. With just one request and response

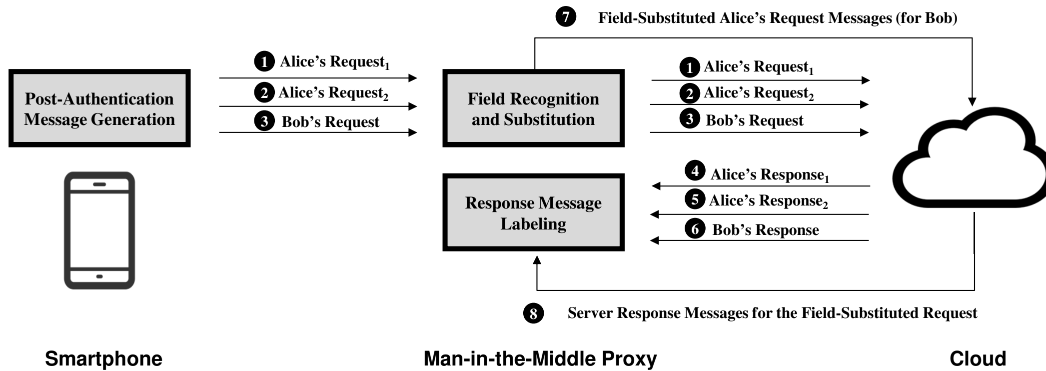

Figure 4: An Overview of AuthScope.

message pair, it will be challenging to recognize the protocol fields of our interest. However, if we have two legitimate users and have two such message pairs, we can easily identify the fields of our interest by aligning the two corresponding messages and looking for the differences, as what we have done in our prior work AutoForge [47]. For instance, if we align the two request messages generated by Alice and Bob, we can easily recognize the UID field and the in_app_token field as shown in Figure 2.

- **Substituting the fields having small Euclidean distance**. We do not have to substitute the cryptographically generated token fields since it will be so random and impossible to guess (e.g., the the in_app_token field in our running example), and instead we should substitute the field whose corresponding diffed value has a short distance (e.g., the UID field with value 21690 and 21691, which has just one Euclidean distance, if we convent these two numbers to integers). This also means we have to convert all the numbers and strings to computable forms such that the Euclidean distance can be measured between the two diffed values. Note that there might be some other distances but euclidean distance can serve our purpose in our problem setting.
- **Labeling server response also with differential traffic analysis**. After we substituting the fields of our interest (e.g., the UID field) as shown in Figure 3, we have to decide whether the substitution indeed proves the existence of the vulnerable authorization in the server side. Fortunately, we notice that when substituting the Alice's UID with Bob's, if the server responses with Bob's private message we observed before, then it is indeed vulnerable. More specifically, as demonstrated in our running example, the response message in Figure 3 is identical to the response message in Figure 2(b), which truly confirms that the server side of W app is vulnerable. However, the response may contain some message specific information such as the time stamp. Fortunately, differential traffic analysis can also identify these fields and filter them out, as demonstrated in AutoForge [47].

3.3 System Overview

An overview of AuthScope is presented in Figure 4. There are three key components: (1) **Post-Authentication Message Generation** that drives the app execution and triggers the legitimate user's post authentication request messages, (2) **Protocol Field Recognition and Substitution** that recognizes the protocol fields of the request messages and mutates the field of our interest, and (3) **Response Message Labeling** that labels the response messages and decides whether the server is vulnerable to access control violation attacks. All of these components run in the client side (without accessing any server code) either in a mobile device, or in a man-in-the-middle network proxy.

Scope and Assumptions. We focus on analyzing the mobile apps that use HTTP/HTTPS protocols, although AuthScope can be extended to analyze non-text protocols. Also, we focus on the apps that use the Facebook login; otherwise we will not be able to automatically trigger the post-authentication messages. Regarding the type of the access control vulnerabilities, we focus on the vulnerable authorization implementations that are caused by (i) no security token, (ii) no randomness of when referring resources at server side when no token, (iii) no access control enforcement when using token. Other vulnerabilities of the server access control such as (1) a user security token is never changed in the life span of the user, (2) how random a token is, (3) the token is transmitted in plaintext, or (4) no/weak authentication, are out-of-scope of this work.

With respect to the HTTPS traffic, since we control the smartphone and also the man-in-the-middle proxy, we install a root certificate in the phone signed by ourselves, and then we can observe the traffic in the proxy in plaintext. Such a method has been widely used in many systems to observe the HTTPS traffic between mobile apps and servers (e.g., [5, 46, 47]).

4 DETAILED DESIGN

In this section, we present the detailed design of the three key components of AuthScope. Based on their execution order, we first describe how to trigger the post-authentication message of a mobile app in §4.1, then explain how to perform the protocol reverse engineering to recognize and substitute the protocol fields of interest in §4.2, and finally present how we label the response

message and detect the vulnerable access control of the remote services in §4.3.

4.1 Post-Authentication Message Generation

Unlike many other mobile app dynamic analyses which only need to randomly trigger some app activities, we need an analysis that can allow the app to enter an important state (i.e., the post authentication state). At a high level, this would mean that we need to first register a legal user in the remote service when given a mobile app, and then execute the app with the registered user and meanwhile successfully log in the server. However, the user registration (i.e., sign up) interface of a mobile app can actually be quite sophisticated. We cannot run a dynamic random testing tool such as Monkey [7] to perform the user sign up because of the various constraints in the interface such as some input may need to follow certain format (e.g., username, passwords, emails, zip code, phone numbers), some input (e.g., passwords, PINs, and emails) may need to enter twice for consistency checks, and some input must satisfy some constraints (e.g., age needs to be greater than 18).

It might appear we need to use symbolic execution to collect the constraints and solve them to finish the server sign up process from a mobile app. However, many of the constraints checking code may just exist in the server side, and symbolic execution of mobile app may not be able to collect these constraints. Meanwhile, many of the registration processes may also need users to click certain links sent via the emails. In addition, there might be CAPTCHAs in the user sign up interface. These all make user registration process non-trivial for a large scale study.

Fortunately, we also notice that many mobile apps today use social login, in which a user just needs to log in the service with her social account and the server will automatically pull the data from the corresponding authentication service providers (e.g., Facebook). With this, we can avoid running the sign up process and instead directly run the app to trigger the social login interface. Also, it will be very rare to have sophisticated constraints in order to trigger the Facebook login, and most of the time the social login interface can be triggered with the first few activities if the app does contain such an interface. We also do not need symbolic execution for our later stage post-authentication analysis, as long as we can have one sample request and response message pair (this is based on the observation that if the server is vulnerable to the authorization, it is very likely that this vulnerability will exist in many of its request messages). Meanwhile, most mobile apps are designed to pull data from servers. An in-depth activity explore shall be able to trigger at least one such message pair. Therefore, we decide to design a targeted app activity explorer (§4.1.1), which will solve both automatic service login via social-based single sign on (§4.1.2), but also the generation of post-authentication messages for our later stage analysis.

4.1.1 Targeted App Activity Explorer

Again, while we could have just run random dynamic testing tool such as Monkey to explore the app activities, such an approach would be very inefficient (cannot meet our large scale study goal) and cannot provide any guarantees of triggering the code we intended. Inspired by prior works such as AppsPlayground [34], SVM-Hunter [35], and Gui Ripping [27, 31] in which UI elements are recognized and used to drive the app execution, we also design an approach that parses the UI elements in a given activity and then leverages a depth-first-search (DFS) algorithm to explore the next-layer app activities and trigger the activities of our interest (such as Facebook Login).

In Android, an activity represents a single screen (can be a window or a floating window embedded in another activity) interface that interacts with users. Every activity defined for the app must be declared in the manifest file. Within each activity, every UI element such as a Button, an ImageButton, a CheckBox, an EditBox, etc., represents a view. All the views are defined in the layout file of an activity or defined by programmers in code. Each view can be binded to a specific action. When a user interacts with an activity (e.g., click a Button), the action binded to the corresponding specific view will be invoked, which might lead to jump to another activity.

Since we would like to explore as many activities (as well as the views inside an activity) as we can, we have to uniquely identify each activity and each view such that we do not have to explore the activity and the view again (e.g., click a Button again) if we have explored it before (otherwise our DFS activity exploration may encounter dead loops). To uniquely identify an activity is trivial, we use the name of each activity as the signature, due to the uniqueness of the activity name. However, there is no such a single obvious attribute to uniquely identify a view. Note that intuitively, the memory address of each view object should be unique, but the memory address of a view can be changed when an activity is refreshed.

View Identification. In Android, all activities for a task are maintained using a stack, and they are arranged in the order according to the time when each activity is opened. For example, the current activity is at the top of the stack; when jumping to another activity, the state of current activity is saved in the top of the stack and then opens the new activity. When the new activity finishes (e.g., the user clicks a back Button), the older activity stored in the top of the stack will be popped up. Such an activity exploration mechanism can make each single view appear on the screen multiple times. But for our analysis, exploring each view only once is enough.

To avoid exploration redundancy and ensure efficiency, we need to uniquely identify each view. In AuthScope, we use a vector with six attributes $<N, C, T, I, A, H>$ to uniquely identify a view, more specifically:

1. N: the *N*ame of the activity, to which the view belongs.
2. C: the *C*lass name of the view (e.g., the class name of Button android.widget.Button).
3. T: the *T*ext or image displayed on the view. Typically, the text or image of a view should be different with other views in the same activity.
4. I: the *I*D of the view. Developers may assign each view with an ID in the layout file. However this value could be NULL because this is not a mandatory rule for developers and not every view has an ID.
5. A: the class name of the *A*ction binded to the view.
6. H: the *H*ierarchy of the view in the layout file. Typically, each activity has its own layout file, which contains the type and location of each view. However, not every activity has a layout file, because Android allows developers to hard-code the arrangement of the layout in the source code.

With $<N, C, T, I, A, H>$, AuthScope can uniquely distinguish each view from others. We have to note that we are not the first to encounter this view identification problem. In fact, AppsPlayground [34] has used $<T, H, L>$ where L represents the location of the view in an activity and T and H are the same as in AuthScope. While $<T, H, L>$ may be sufficient in their application scenario [34], we find we do need more information in our use case. For instance, we observe H can be missing because not all developers use layout files for view arrangement. Meanwhile, L can be changed in some views. For instance, when scrolling up and down, the location of the view in an activity can be changed. In contrast, AuthScope has a much stricter policy in determining the uniqueness of a view, and our N, C, A will never be missing and H can be used to solve most of the scrolling problems.

View Exploration. When an activity is created, AuthScope will automatically create the $<N, C, T, I, A, H>$ vector for each view within the activity. Having uniquely identified each view, we then explore the current activity using a prioritized DFS algorithm. Since we aim to exercise the social login interface before authentication and any explorable interface after authentication (to get sample request and response message pair), we classify all views in the same activity into three categories and then prioritize the DFS traversal in the following order:

- view contains social login (e.g., Facebook login).
- view has a binding action.
- view has no binding action.

To summarize, similar to AppsPlayground [34], when an activity is created, AuthScope recognizes each unique UI element (i.e., view) of the activity, traverses each view using a prioritized DFS algorithm. If the view has been visited before, we will not traverse it again. We finish the exploration of the current activity, when we traverse all of its views.

4.1.2 Automatic Social-based Service Login

Having the capability of exploring the app activities, next we need to drive the app to execute social login interface. We use a similar approach of how a real user recognizes whether a view contains social login. In particular, take Facebook login as an example, real users recognize there is a Facebook login by reading the text over a Button, such as "Sign in with Facebook" or "Facebook Login". By scanning the text of a view in the layout file whether or not containing Facebook sub-string, we prioritize the activity exploration to such a view. If there is no such string, there must be a binding action to invoke the Facebook login, and our DFS traversal will also eventually invoke it. Normally, this login interface exists in the first few activities and it is very unlikely that there will be any constraints involved to invoke the Facebook login.

After AuthScope successfully clicks the Facebook login button, the app will follow the execution logic in the library from the Facebook, which is a very standard logic. We just pre-register two accounts Alice and Bob with the Facebook service (the reason of why we need two users is presented in §4.2), and then automatically log in the corresponding servers using the Facebook account when the login interface pops up. The app execution after this stage will trigger those post-authentication request and response messages when performing our DFS traversal of the activities and views. Our DFS traversal algorithm is deterministic. Therefore, the exercised request and response message sequences are consistent among different users.

4.2 Message Field Recognition and Substitution

With the exercised request and response messages collected by our man-in-the-middle proxy, next we need to infer the message fields and substitute the fields of interest to see whether the server has vulnerable authorization implementations. To perform this automatically, we need to design a principled approach to (1) parse the message fields (§4.2.1), (2) identify the fields of interest (§4.2.2), and (3) substitute the fields that are enumerable (§4.2.3).

4.2.1 Parsing Message Fields

Since AuthScope focuses on HTTP/HTTPS protocol, we just need to parse the post-authentication request and response messages for this well-formed text protocol. According to the HTTP protocol specification [2], each request message consists of (1) a request line (e.g., GET /index.html HTTP/1.1), (2) request-header fields (e.g., Host: www.sigsac.org), (3) an empty line, and (4) optional message body. Similarly, each response message consists of (1) a status line (e.g., HTTP/1.1 200 OK), (2) response-header fields (e.g., Accept-Language: en), (3) an empty line, and (4) an optional message body. Both Figure 2 and Figure 3 contains more concrete examples of request and response messages.

Parsing Request Messages. Each request message needs to be responded by a server API, and this API can be indexed by the value of Host and the resources requested in the request line. To parse each request line, we need to first parse the path segment by scan the reserved path symbol "/" and then retrieve each directory name. If there is any URL encoding in the request line (as in our running example), we also need to parse each request parameter name (e.g., in_app_token) and its value. Note that in URL encoding, the parameter name and its value is connected by symbol "=", each pair is concatenated by "&". It is quite straight forward to index the parameter name and its value, and we store them in a pair $<name, value>$.

Regarding the message body, it can be just empty, data encoded with URLs, JSON (e.g., as shown in our running example), XML, html page, or just some text. We only parse URL, JSON or XML encodings of the message body, and treat the rest just as text. To parse URL encoding, we parse it in the same way as in request line. For JSON and XML, they both have a hierarchy tree structure, which means that each value can be tracked by the path from the root of the tree. Also, note that if the value of a parameter is a JSON array, we will not consider the order of the element in the array. That is the array [a,b] should be treated as the same array as array [b,a] when we build the parameter and value pair (i.e., $<name, value>$) when parsing the message field.

Parsing Response Messages. The response message is sent by the server after it processes the request (essentially the return value of the server API). We will associate the response messages with the corresponding request messages. Similar to how we parse the

request messages, we use the same way to parse the response messages and build <*name*, *value*> pairs if there is any. The response message will be primarily used in §4.3.

Indexing the request and response messages. After parsing each request and corresponding response message pair, we need to index it such that we can easily locate it during our next stage analysis (§4.2.2). Essentially, this can be considered as an instance of a server API execution, and we have collected the server interface (i.e., the URLs that include the Host address and data reference path), the parameters, and return values. Therefore, we index it based on the URLs, the <*name*, *value*> pair we parsed from the request message (which can be considered as parameters) and the response message.

4.2.2 Identifying Fields of Interest

Clearly, not all fields are of our interest. For instance, in our running example shown in Figure 2(a), we are just interested in field with value 21690 and the in_app_token field in Alice's request message. Since there are many non-related fields in a request message, we must automatically select the fields of our interest. The key solution here is to use *message alignment* and *value diffing*, a common approach used in protocol reverse engineering, such as Protocol Informatics [13]. That explains why AuthScope requires at least two registered users (e.g., Alice and Bob) with the service, and also one user needs to login and logout twice to exercise two sets of the same request messages (e.g., Alice's Request_1 and Alice's Request_2 as shown in Figure 4).

Message Alignment and Value Diffing. In general, a request message could contain user-specific fields (e.g,. in_app_token), and non user-specific fields (e.g., the request-header fields, and also timestamp field if there is any in the request message). By using message alignment and value diffing, we can quickly locate user-specific fields, and non user-specific fields.

- **Aligning and Diffing Different Users' Same Request**. By aligning and diffing with the same request messages (recall that we have indexed all of the request messages) of two different users (e.g., Alice's Request_1 and Bob's request), we can quickly identify the user-specific fields by selecting the value diff-ed fields. For instance, by aligning and diffing the two *different users* request messages showing in Figure 2, we can automatically locate field 21690 and 21691, and the fields in_app_token. The rest fields have no differences and are therefore not of our interest.
- **Aligning and Diffing Same Users' Same Request at Different Time**. However, some *message-specific* fields (e.g., timestamp if there is any) can also be value different. Therefore, we will further align and diff the two request messages of the *same user* (e.g., Alice's Request_1 and Alice's Request_2) to remove those message-specific fields.

Selecting the Fields of Interest. The key objective of AuthScope is to discover the vulnerable authorization by performing what an attacker could do — substituting a guessable field and observing whether other user's information can be leaked. Therefore, we should focus on the fields that are guessable or enumerable (can be performed by a brute-force attack). In our running example,

Field-Value of Alice vs. Field Value of Bob	*ED*
fb153b7d8c0a0c6ac841d7bfbd9446de627c642858 e67315b35aa38d4ac8cac3cd9c7f88ae7f576d373f	$+\infty$
21690 21691	1.00

Table 1: The Euclidean distance of the diffed-fields between Alice's and Bob's request messages.

clearly we should select and substitute field 21690 with value 21691 (as what we did in Figure 3), instead of the in_app_token field because a token is in general unguessable and substituting a token does not reveal the vulnerabilities (if a security token is changed, the response should be changed as well). As such, we need an algorithm to select the enumerable fields. Fortunately, we notice that by using the Euclidean distance and predictable values, we can automatically locate such fields.

- **Euclidean Distance**. An Euclidean distance (*ED*) is a metric that measures the ordinary straight-line distance between two points in Euclidean space. The smaller an *ED* of a field, the more likely to be guessed by attackers. For instance, as shown in Table 1, the *ED* of 21690 and 21691 is just one, whereas the *ED* between the two tokens is 14a225ca31667f1fff7713f22114be2fe324f6f119 (we thus consider it a giant astronomical number $+\infty$). Certainly, when having a sample message with 21690, an attacker can quickly probe other user's information in a service by changing to other closer numbers, whereas for token it is hard for attackers to guess other's.

  Then the next question becomes how AuthScope computes *ED* and decides whether a distance is $+\infty$ (unguessable). To achieve this, AuthScope converts all diffed value (including strings and byte sequences) to numbers using their minimal base. For instance, we will convert 21690 to a decimal value (using base-10), and the token using base-36 (alphabetic + number). If a string contains other printable ASCII symbols (recall HTTP is text-based protocol), we will use the worst case base-95 to convert it (there are maximum 95 printable ASCII characters).

  To decide whether an *ED* is $+\infty$, we set a threshold based on the number of downloads of the app. The intuition is if the *ED* is smaller than the total number of downloads showing in the app market, we consider the corresponding field enumerable because any substitution of the value with a nearby one will likely lead to the disclosure other user's information if the server is vulnerable.
- **Predictable Value**. Using *ED* can find most of the guessable fields. However, there are a few special cases that the *ED* might be $+\infty$, but it is guessable. One example is the email address. Very likely, the *ED* of two email addresses can be $+\infty$, but an attacker can easily guess other's email address because of the recent huge data leakage of user accounts in online services, making the email address value predictable. Therefore, we use string matching to handle such fields. More specifically, if any of the request message contains Alice's email address (using email address

pattern matching), this guessable email address field is of our interest.

The other example is the Facebook ID (FID). While it is a giant integer (e.g., 17927643151, which is the ACM's), it can be publicly crawled. Other than this ID, when user using Facebook login to log in to a specific app, Facebook will issue an app-specific ID [1] (e.g., 106611716575863 as shown in the case study in Figure 6) to the user which is unique to each app, and such ID can also be easily crawled (e.g., within the app). Therefore, we also call this app-specific ID FID and consider it public available knowledge. Similar to the email case, if we observe Alice's FID is used in a request message, we will replace it with Bob's and observe how server would respond the request.

4.2.3 Substituting Enumerable Fields

Now we have identified all of the guessable fields, next AuthScope will substitute them to decide whether there is a vulnerable authorization implementation. This step is quite straightforward: for any identified enumerable fields in Alice's request message, our man-in-the-middle proxy will just replace the value of this field with Bob's. If there are multiple fields, we will send multiple request messages. Only one field at a time is substituted in each message, and we will not simultaneously substitute fields at the same time (as it is unlikely that an authorization depends on two fields).

4.3 Response Message Labeling

After we have sent a field substituted request message of Alice with the value of Bob's to the server, we then label the response message to determine whether the server is vulnerable. The key idea to decide this is if the response message returns the identical ones with the same response message requested by Bob, then the server is vulnerable.

More specifically, we label a response message that is returned by a field-substituted Alice's request message is identical to the corresponding Bob's response, if the user-specific data in the response message is the same (byte-by-byte identical). That is, we will remove those non-user specific data (such as message-specific timestamp) using the differential traffic analysis again, i.e., the alignment and differing approach described in §4.2.2 when identifying the non-user specific field in the request messages. Without differential analysis, we will not be able to tell we have successfully retrieved Bob's data by just byte-by-byte comparison of the response messages if there is any message-specific data. After removing these non-user specific data in the response message, AuthScope outputs that the server is vulnerable if we find an identical response for the corresponding request interface. We will keep substituting and labeling, until all the Alice's request messages have substituted. If none of the response messages are identical with initial Bob's, then the server is not vulnerable. A server may have multiple vulnerable interfaces if multiple of its server request interfaces are vulnerable.

Pruning the Vulnerable Interface that Provides Public Resources. Certainly, AuthScope can have false positives if the vulnerable interfaces identified are used to provide the public resources. Since it is a public resource, no matter how we substitute the enumerable fields, the server will always return the same response. For instance, a news app that provides news to subscribed users may be flagged as vulnerable if the news is fetched after authentication and this news can also be accessed without login (a public resource).

To further prune such cases, we then let AuthScope take one more run of the app without loggin the service. That is, when it encounters the Facebook login interface, it directly skips it and continues exploring the app as deeply as it can. We will align these after authentication-skipped request messages with those in Alice's and Bob's. If we observe a previously identified vulnerable interface can actually serve the public resource, we will not flag it vulnerable.

5 EVALUATION

We have implemented AuthScope atop Android 4.4 platform by using the Xposed [6] framework to drive the app execution and perform targeted app exploration, and our man-in-the-middle proxy is implemented with the Burp Suite [5]. In total, AuthScope consists of over 5,000 lines of our own Java code and 300 lines of our own python scripts. In this section, we present our detailed evaluation results.

5.1 Experiment Setup

Dataset Collection. As of today, Google Play has over 2 million mobile apps. To have a reasonable coverage of these apps, we crawled the top 10% of free mobile apps based on the number of installs in March 2017. Recall that AuthScope requires automatic login and currently we only focus on the apps that use Facebook login, and thus we have to select such apps. To this end, we first analyzed the 200,000 apps to filter out those that do not import any Facebook libraries. Note that if an app has not imported Facebook libraries, definitely it does not have Facebook login. After this initial filtering, we have 33,950 remaining apps.

However, even if an app has imported Facebook library, there is no guarantee that it will use Facebook login, we have to perform a further analysis. In particular, we have observed that there are two ways to integrate Facebook login in an app: (1) directly put a Facebook login button (implemented by Facebook library) in one of its activity layout files, or (2) call Facebook login function using program code. Based on these two observations, we design another screening procedure, which first checks whether the Facebook login button exists in one of its activity layout files within an app; if there does not exist such a button, then continues to search code that invoking Facebook login methods from the Facebook library. This code search is implemented by using the Soot framework and checking the function call patterns. If the filter neither finds out the Facebook login button nor the invoking code, then this app will be discarded. With all these filtering analyses, eventually we have 4,838 apps in our dataset.

Testing Environment. All of our apps were tested in a real LG Nexus 4 smartphone with Android 4.4 system. This phone is installed with our app post-authentication message generation component, and is connected with a Ubuntu 14.04 desktop running

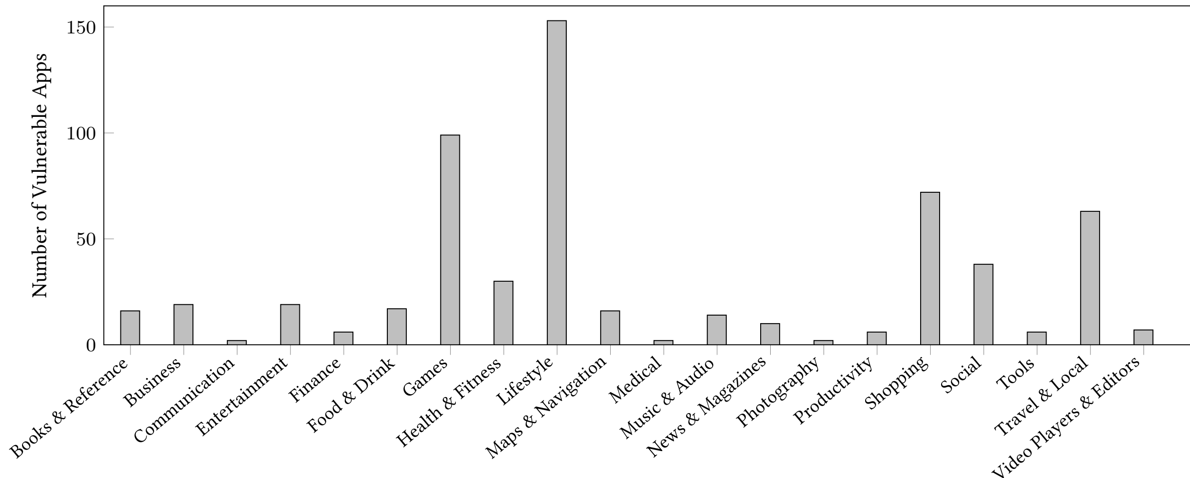

Figure 5: Distribution of the Vulnerable Interfaces Based on the App Category.

Item	Value
Σ # Apps	4, 838
Σ Time to perform the test (hours)	562.4
Σ # Request messages	3, 220, 886
Σ # HTTP Messages	178, 539
Σ Size of the messages (G-bytes)	59.2
Σ Time of activity exploration before authentication (hours)	169.9
Σ # Explored activities before authentication	15, 367
Σ # Identified views before authentication	503, 441
Σ # Explored activities after authentication	20, 704
Σ # Identified views after authentication	1, 181, 442
Σ # Mutated fields	57, 736
Σ # Suspicious interfaces	2, 976
Σ # Public interfaces	2, 379
Σ # Vulnerable interfaces	597

Table 2: Overall Experimental Result.

atop an Intel i7-6700k Skylake 4.00 GHz CPU with 8G memory. This desktop controls the automatic app execution in the smartphone through the ADB interface by python script, and meanwhile intercepts, collects, and mutates the network messages between the apps and remote servers, using our man-in-the-middle proxy. Also, we registered with Facebook two test accounts Alice and Bob with email address alice4testapp@gmail.com and bob4testapp@gmail.com, respectively.

5.2 Evaluation Result

Macro Level. We spent 562.4 hours in total to dynamically analyze these 4, 838 apps, and eventually we discovered 597 vulnerable authorization implementations in the corresponding servers that map to 306 apps. The overall experimental result is presented in Table 2. In total, we generated 3, 220, 886 request messages, and among them, 178, 539 are HTTP protocols (the rest are HTTPS). The total size of these messages is 59.2 G-bytes. To execute the Facebook login, our analysis spent 196.9 hours, during which we explored 15, 367 activities, and 503, 441 views. After we get authenticated, we explored 20, 704 activities, and 1, 181, 442 views. We mutated in total 57, 736 fields, and found 2, 976 suspicious server interfaces that have vulnerable authorization implementation. Among them, our further analysis revealed that 2, 379 are public resource interfaces. Eventually, we found 597 vulnerable server interfaces after pruning those that provide public resources, and they map to 306 mobile apps.

To understand those popular vulnerable services, we present the distributions of these vulnerable app servers based on the corresponding top level app category[2] assigned by Google Play. This result is shown in Figure 5. Interestingly, we found these apps belong to 20 categories. The top three categories include Lifestyle (which has 153 vulnerable interfaces), Game (99), and Shopping (72). One reason of why these categories contain so many vulnerable implementation is that we found the apps in these categories typically highly interactive, the user data is often stored, shared, updated in their servers, which also means there are more resources in those apps and more complicated access control implementation. Also, surprisingly, we found a number of vulnerable implementations in Finance (3) and Business (9) related apps. The data leakage in these servers can cause serious damages to the end users. We will discuss the severity of these leakages in §6.

Micro Level. After we have described the overall result, next we show clearly how AuthScope performs for each app. We selected the top downloaded app in each vulnerable server category presented in Figure 5 and show the detailed result in Table 3. The first two columns are the category name and package name[3], respectively, followed by the numbers of activities that we explored, and the numbers of unique views that we identified during the dynamic analysis on each app. The fifth column is the time that our system spent on finding Facebook login, which is the time from starting the app to successfully login the app. The sixth column is total numbers of request messages that the app has generated, the seventh column is the total number of fields that we substituted

[2]The only exception is we further cluster all top level game sub-category into a game category.

[3]For the apps whose servers have not been patched yet as the time of this writing, we do not reveal their full name and instead anonymize their name with ***.

Category	Package Name	# Activities	# Views	Time to Login (s)	#Request Messages	#Mutated Fields	#Public Interfaces	#Vulnerable Interfaces
Books & Reference	com.***.e***	3	288	45	975	16	5	3
Business	com.***.k***	8	1,224	30	927	12	2	3
Communication	com.***.w***	18	970	41	727	1	0	1
Entertainment	com.***.c***	3	184	32	739	2	0	1
Finance	com.***.m***	8	549	16	790	7	0	2
Food & Drink	com.***.h***	10	924	21	1,032	8	4	1
Games	com.***.c***	7	609	20	1,050	7	3	1
Health & Fitness	com.***.u***	12	788	15	966	10	2	2
Lifestyle	com.m***	17	1,938	25	1,229	29	5	5
Maps & Navigation	com.******.c***	11	667	26	490	12	7	1
Medical	com.***.a***	18	1,616	23	927	9	2	1
Music & Audio	com.b***	2	456	25	933	15	3	1
News & Magazines	com.***.a***	5	462	37	880	9	0	2
Photography	com.***.j***	15	909	26	965	7	0	1
Productivity	com.***.d***	15	1,347	32	882	10	5	1
Shopping	cl.******.i***	8	795	44	961	10	0	5
Social	in.v***	10	645	20	1,068	20	4	5
Tools	com.mediaingea.uptodown.lite	7	1,347	112	1,276	25	6	1
Travel & Local	com.t***	5	321	35	1,024	10	0	2
Video Players & Editors	cz.***.n***	4	218	25	821	5	1	1

Table 3: Detailed Experimental Results for Top Tested App in Each Category.

Category	Detailed Privacy Type
User E-Profile	(01) Email, (02) User ID, (03) Registration Date, (04) IP Address, (05) Last Login Date, (06) Last Update Date
User Physical-Profile	(07) Real Name, (08) Birthday, (09) Geo-location, (10) Home Address, (11) Phone Number, (12) Body Information
User Secrets	(13) Token, (14) Password, (15) Pass Code
App Specific Private Data	(16) In App Messages, (17) Shopping History, (18) Book Shelf, (19) Favorites or Subscription, (20) Account Balance (21) Contacts Information, (22) Payment Information, (23) Private Activity Information

Table 4: User Privacy

APP	Version	Credential Type	User E-Profile	User Physical-Profile	User Secrets	App Specific Private Data
com.***.e***	2.2	N	(01)(02)	(07)(08) (10)(11)		(18)
com.***.k***	2.0.11	E	(01)(02)(03)(04) (06)	(07) (09)(10)(11)	(13)(14)	(17) (19)
com.***.w***	1.0.5	F	(01)(02)	(07)	(13)	(16) (19) (21)
com.g***.c***	2.4.1	E	(01)(02)	(11)	(13)	(17) (20)
com.***.m***	1.6.8	N	(01)(02)	(07) (09)(10)(11)	(15)	(16) (20) (22)
com.***.h***	2.5.6.0	E	(01)(02)		(13)	(16) (19)
com.***.c***	2.6.1	F	(01)	(07) (10)	(13)	(19)
com.***.u***	2.03	N	(01)(02)(03)	(07)(08) (12)		(16) (19)
com.m***	7.3.0	N	(02)	(07)(08) (10)		(16) (19)(20)
com.******.c***	7.5.5v	F	(01)(02)	(07) (09)(10)(11)	(13)	(19) (22)
com.***.a***	3.09	F	(01)(02)	(07)	(13)	(16) (19)
com.b***	2.0.4	N				(19)
com.***.a***	2.3.2	N	(01) (03)(04)(05)	(07)(08)(09)	(13)	(16) (19)
com.***.j***	2.7.4	F	(01)(02)(03)	(07)	(13)	(16) (19)
com.***.d***	2.4.2	E	(02)(03)	(07) (09)	(13)	(23)
cl.******.i***	2.1.0	N	(01)(02)	(07)(08)(09)		(20)
in.v***	4.4.5.2	N		(07)(08)		(16) (19) (21)
com.mediaingea.uptodown.lite	3.18	F	(02)		(13)	(16) (19)
com.t***	1.4.0	F	(01)(02)	(10)(11)	(13)	(16) (23)
cz.***.n***	4.8	F	(02)	(07)	(13)	(23)
Statistics		8 4 8	14 16 05 02 01 01	15 06 06 07 06 01	13 01 01	11 02 01 13 04 02 02 03

Table 5: Vulnerability Details for Top Tested App in Each Category, where N denotes Numeric values, E denotes Emails, and F denotes Facebook IDs.

for the tested app, the eighth column is the number of public interface identified, and the last column is the number of vulnerable interface discovered for the tested app.

We can notice from Table 3 that some apps have many activities, which means it would be really hard to use blind dynamic analysis tools such as Monkey [7] to explore all of these activities. Also, all apps have hundreds of request messages. This is actually because many of the messages are related to Facebook login. In our experiment, we found for each Facebook login, Facebook library will generate hundreds of request messages to **static.xx.fbcdn.net** to retrieve **js** files.

Also, the last column shows that 9 apps have more than one (from 2 to 5) vulnerable authorization interfaces at the server side, and 13 apps also contain several (from 1 to 7) public interfaces. Interestingly, we also find if the attack surface is either email address or FID, then there will be just one vulnerable interface (and this interface is usually the one serves the first request message right after authenticated with Facebook). If the attack surface is a predictable number, then there are likely more than one vulnerable interfaces. This is because likely all other requests also use the predictable number, which makes their corresponding server interfaces all vulnerable.

6 SECURITY ANALYSIS

6.1 Systematized Analysis

Next, we would like to understand what kind of data leakage the vulnerable access control implementation can cause and how severe they are. To this end, we have manually examined the 20 vulnerable app servers for the app presented in Table 3. To systematize the leakage, we first classify the leaked data into four categories as shown in Table 4 (based on our best understanding) including **user e-profile** such as her email address, service registration date, IP address, last login date, last update date; **user physical profile** such as full name, birthday, geo-location, home address, phone number, and body information such as weight and height; **user secrets** such as access token, user password (either plaintext or hashed), app pass code; and **app specific private data** such as shopping history, book shelf, favorites, payment information, account balance, etc. Based on these classification, we looked into each of the vulnerable service interface and examined their data leakage. The detailed result for these 20 vulnerable servers is presented in Table 5.

From the 3rd column of Table 5, we notice that 8 out of 20 vulnerable servers just use predictable numbers to access a user's private information (e.g., for app cl.***.***.i*** and we will just call it I app, our user Alice has a UID 673436 and Bob has 673491 as presented in Figure 6), 4 use email addresses, and 8 use FIDs. Also, we can observe that various user private data can be leaked from the vulnerable servers. The top leaked data includes UID (16), email address (14), and security token (13). Note that these tokens actually belong to Bob, but can be retrieved by Alice. Meanwhile, surprisingly, some of the servers even leak user's password, as shown in Figure 7. This is astonishing, since a user's password should never be leaked to a client regardless of the query.

In short, given such easily predictable numbers and potentially public available email addresses and FIDs without any further authorization checks, it makes an attacker trivially crawl all user's private data from the victim servers. In our experiment settings, an adversary can possibly get up to 61 million mobile users private record according to the total number of downloads for all the vulnerable apps.

```
00 {
01  ...
02  "response":{
03   "user":{
04    "idnum":false,
05    "name":"Bob",
06    "lastname":"Ccs",
07    "birthday":"1990-04-26",
08    "gender":"M",
09    "email":"bob4testapp@gmail.com",
10    "type":"EMAIL",
11    "firstlogin":"1",
12    "country":{
13     "id":"10",
14     "name":"United States",
15     ...
16    },
17    "post_on_activities":"disabled",
18    "bananas_count":0,
19    "id":"673491",
20    "fbid_number":"106611716575863",
21    "current_latitude":"30.9863214",
22    "current_longitude":"-86.7501116",
23    "bananas_history":"https:\/\/profile.*******.com\/bananas\
/store\/673491\/?accesstoken=debda35ccd92f4b8e2e06f0bff3b6e49279
a557d&latitude=30.9863214&longitude=-86.7501116&lang=",
24    ...
25   }
26  }
27 }
```

Figure 6: Alice Read Bob's Account Information in app I.

6.2 Case Studies

As demonstrated in our systematized analysis, vulnerable authorization can easily lead to user private data leakage. To understand this threat more concretely, in the following, we would like to perform further analysis of two mobile apps, namely the I app and com.***.k*** (we just call it K) app from Table 3, to show how they could leak user's privacy sensitive data including user's secrets. These two case studies require detailed knowledge of the mobile apps and were conducted manually.

Sensitive Data Leakage. We use I app as an example to illustrate this attack. I app is a very popular app in Google Play with 100, 000 to 500, 000 downloads. This app can provide discount information for shopping. During our test, AuthScope intercepted the Alice's request which asks for personal information, and replaced Alice's account ID or UID (673436) with Bob's UID (673491) for a new request. Figure 6 shows a portion of the response message. We can see clearly that it leaks a lot of Bob's sensitive information, including his birthday, gender, email, Facebook ID, current location and balance history.

For this app and so many other alike apps, the attacker only needs to get the UID of a user to perform the attack. Moreover, to get other's UID is relatively straightforward. In this app, the UID is generated incrementally, not randomly. Given such a 6-bit UID and its install numbers, statically, an attacker can easily enumerate other's UID. Since this app has close to 500, 000 installs, an adversary can easily retrieve 500, 000 user's private information.

```
00 {
01  "pk_i_id": "163126",
02  "dt_reg_date": "2017-04-30 23:21:59",
03  "dt_mod_date": "2017-04-30 23:36:58",
04  "s_name": "Bob Ccs",
05  "s_username": "163126",
06  "s_password": "7c4a8d09ca3762af61e59520943dc26494f8941b",
07  "s_secret": "6stgMaAb",
08  "s_email": "bob4testapp@gmail.com",
09  "s_website": "bob.ccs\/index.html",
10  "s_phone_mobile": "4695855213",
11  "s_pass_ip": null,
12  "fk_c_country_code": null,
13  "s_country": "Tanzania",
14  "s_address": "15246 Sni Rd. APT 252 Tanzania",
15  "fk_i_region_id": "17",
16  "s_region": "Mara",
17  "d_coord_lat": null,
18  "d_coord_long": null,
19  "b_company": "0",
20  "i_items": "1",
21  "i_comments": "0",
22  "dt_access_date": "2017-04-30 23:46:05",
23  "s_access_ip": "",
24  "b_prefer_phone": "1",
25  "s_dialing_code": "+255",
26  "fk_i_category_id": "22",
27  "s_facebook_page": "http:\/\/",
28  ...
29 }
```

Figure 7: Alice Read Bob's Information in app K.

Secret Data Leakage. Other than user's private data, more sensitive secret data can also be leaked from the vulnerabilities discovered by AuthScope. Considering the K app as an example, it is a second-hand goods trading app on Google Play, which has between 500,000 and 1,000,000 downloads. With this app, any registered user can sell/buy second-hand goods. Unfortunately, we found the authorization vulnerability can lead to user's secret data leakage.

In particular, after authentication, the server will push detailed information of the user based on her email address. After substituting Alice's email with Bob's, AuthScope successfully got Bob's information, part of which is shown in Figure 7. As we can see, there are quite a number of private records in the response message such as the registration date (line 2), modification date (line 3), user name (line 4), phone number (line 10), home address (line 14), Geolocation (line 17, 18 which is null in our test case), last login time (line 22). Among these leaked data, the most dangerous record is Bob's hashed password (which is 7c4a8d09ca3762af61e59520943 dc26494f8941b). Under no circumstance should the app provide user's password to the user. With a further investigation, we found this hash password is generated by SHA-1, which can be easily cracked by many online services (e.g., https://crackstation.net/ which takes less than a second to return the plaintext of this password).

With this authorization vulnerability in K's server, an attack can easily get the hashed password, and further crack a user's password when provided with the victim's email address. Recently, there are huge data breaches and likely the attacker can trivially probe the victim's email in K's server. However, we also found when opening a seller's page, her email address is embedded in the meta data. Therefore, an adversary can also crawl all the products in this service and get all seller's email, and further get their hashed password. Considering that most online users today reuse their password, such an attack can cause serious damages to many online users.

7 DISCUSSIONS

Limitations and Future Works. While AuthScope has made a first step towards automatic discovery of authorization vulnerabilities in online service, it still has a number of limitations. First, clearly AuthScope has false negatives. For instance, we only focused on the apps that use Facebook login (essentially using Facebook login to bypass the authentication step), but not all the apps have been using this social login. In our experiment, we filtered more than 25,000 mobile apps that do not contain Facebook logins. How to handle other social login schemes (e.g., Google login), or in general how to automatically login a remote service is still an unsolved problem. This may require solving the challenges of automated service sign up, more intelligent Android UI recognition and test case generation, etc.

Second, AuthScope only discovers the authorization vulnerability that leads to the information leakage and account hijacking attacks. Basically, these are attacks that lead to *unauthorized read*. However, there are also many other interesting attacks such as the *unauthorized write*. For instance, a user should not modify any items that belong to other users. Currently, AuthScope is not able to infer the *unauthorized write* automatically.

Finally, the vulnerable authorization is a general problem in online services and is not just limited to Android app's server side implementation. Currently, we only developed the prototype that performs dynamic Android app analysis and protocol reverse engineering to infer the vulnerability, and we believe our methodology can also be applied to other platforms such as iOS and Windows. Also, AuthScope currently only handles the network communications with HTTP/HTTPS protocols. We will study how to enable AuthScope to analyze the vulnerabilities for other platforms and other protocols, as well as addressing the first two limitations in our future work.

Practicality of the Attack and Countermeasures. It is absolutely incorrect to use predictable numbers without further authorization checks to allow access of a user's private resource. However, service developers may feel it is secure to just use email address or other sophisticated numbers such as Facebook ID for the authorization. However, we have to note that recently there are massive data leakages and huge volume of Internet user's email addresses have been leaked. We have to consider that email address is a public information now. Also, Facebook ID can be crawled and it can also be considered public. Therefore, the attacks we discovered are quite practical. To really fix these vulnerabilities, we urge service developers to follow the best practices (as we have discussed in §2.3) such as using random token in each session, enforcing the security checks with the token and particular user, and never assuming that a client is always trusted.

Ethics and Responsible Disclosure. When developing AuthScope for vulnerability discovery, we do take ethics in the highest standard. First, we only tested the services with the two legitimate users we registered (namely Alice and Bob), and we never steal any other user's private information. Second, we never sent a large volume of traffic to a remote service (to perform any denial of service attack), and all the traffic is generated at the speed as how

a normal user interacts with the remote system. Finally, we have made responsible disclosures when we discover a vulnerability.

In particular, we have immediately notified the developers based on the corresponding contact information on Google Play. As a result, some app developers contacted us to discuss the details of their server vulnerabilities and we have worked together with them to patch the vulnerabilities. For those apps whose vulnerabilities have not been fixed yet at the time of this writing, we do not reveal their concrete app names and instead just masked their names with symbol '***' as shown in Table 3. We will continue to provide our best efforts to help fix their vulnerabilities.

8 RELATED WORK

Vulnerability Discovery in Online Services. It is challenging to develop vulnerability free software, and many online services contain various vulnerabilities ranging from SQL injection [21], cross-site-scripting [37], cross-site-forgery [11], to broken authentication [19], and even application logic vulnerabilities (e.g., [33, 39, 40, 43]). Correspondingly, significant amount of efforts have been focusing on identifying these vulnerabilities through either white-box analysis with server code, or black-box analysis with just network traffic.

There are also efforts to particularly study the access control issues in the online services. Most of them focused on the authentication related problems, such as security with single-sign on (e.g., [38, 45]), oauth (e.g., [15, 36]), authentication vulnerability scanning (e.g., [10]), and password brute-force attacks with online services (e.g., [47]). Compared to these works, AuthScope is among the first few to look into the post-authentication issues in online services and is able to automatically discover the vulnerable authorizations when given mobile apps enabled with social login.

Dynamic Analysis of Mobile Apps. AuthScope leverages dynamic analysis of Android apps to generate server request messages. In the past several years, there are a large body of research in dynamic analysis of Android apps (e.g,. Monkey [7], Robotium [4], AppsPlayground [34], and DynoDroid [26]). Recently, there are also efforts of using symbolic execution (e.g., [8, 29, 42, 46] for more systematic dynamic analysis of mobile apps.

Compared to these works, AuthScope is partially inspired by AppsPlayground and we have extended it to support more accurate and deeper UI element exploration. While we can also leverage the symbolic execution to have better coverage, we realize that we may not need symbolic execution as identifying vulnerable authorization may not need large volume of request messages. Certainly, symbolic execution will help though.

Protocol Reverse Engineering. AuthScope needs to reverse engineer the application protocol fields of interest and then perform fields substitution to identify security vulnerabilities. Over the past decade, there are significant amount of efforts of analyzing both network messages (e.g., [13, 16, 17, 25]) and instructions traces (e.g., [14, 18, 22, 23, 28, 41]) to discover protocol formats and use them for security applications. AuthScope is particularly inspired by the protocol informatics project [13], and uses a customized Needleman-Wunsch algorithm [30] to align and diff the protocol messages and infer only the fields of our interest.

9 CONCLUSION

We have presented the design, implementation, and evaluation of AuthScope, a tool that is able to automatically execute a mobile app, generate post-authentication messages, and pinpoint the vulnerable access control implementations, particularly the vulnerable authorizations, on the server side. We have tested AuthScope with 4,838 popular mobile apps from Google Play, and identified 597 vulnerable authorization implementations in 306 mobile apps. These are very serious security vulnerabilities, very easy to attack, and can cause severe damages to end users such as personal information leakage and account hijacking. We have made responsible disclosure to all of the vulnerable service providers, and many of them have acknowledged us and patched (or started to patch) their vulnerabilities. Finally, given the capability of such an automated analysis, we would like to raise the awareness of the vulnerable authorization implementation issues in online services and hope the rest vulnerable service providers could patch their services shortly.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their very helpful feedbacks. This research was supported in part by AFOSR under grants FA9550-14-1-0119 and FA9550-14-1-0173, and NSF awards 1453011 and 1516425. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and NSF.

REFERENCES

[1] "Facebook app-specific ids," https://developers.facebook.com/docs/graph-api/reference/user/.
[2] Hypertext transfer protocol. https://www.w3.org/Protocols/rfc2616/rfc2616.html. Last accessed in May 2017.
[3] "Plain text offenders," last accessed in May 2017.
[4] "Robotium," https://code.google.com/p/robotium/, last accessed in May 2017.
[5] "Using burp proxy," https://portswigger.net/burp/help/proxy_using.html, last accessed in May 2017.
[6] "Xposed module repository," http://repo.xposed.info/.
[7] "Ui/application exerciser monkey," https://developer.android.com/tools/help/monkey.html, 2017.
[8] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11.
[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "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*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269.
[10] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations." in *NDSS*, 2013.
[11] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 75–88.
[12] E. Bauman, Y. Lu, and Z. Lin, "Half a century of practice: Who is still storing plaintext passwords?" in *Proceedings of the 11th International Conference on Information Security Practice and Experience*, Beijing, China, May 2015.
[13] M. Beddoe, "The protocol informatics project," 2017, https://github.com/wolever/Protocol-Informatics.
[14] J. Caballero and D. Song, "Polyglot: Automatic extraction of protocol format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, Alexandria, Virginia, USA, 2007, pp. 317–329.
[15] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 892–903.
[16] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through

differential analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017, pp. 1–16.
[17] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proceedings of the 16th USENIX Security Symposium (Security'07)*, Boston, MA, August 2007.
[18] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, Virginia, USA, October 2008, pp. 391–402.
[19] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: Preventing authentication & access control vulnerabilities in web applications." in *USENIX Security Symposium*, 2009, pp. 267–282.
[20] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "Http authentication: Basic and digest access authentication," Tech. Rep., 1999.
[21] W. G. Halfond, J. Viegas, and A. Orso, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1. IEEE, 2006, pp. 13–15.
[22] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.
[23] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, Atlanta, GA, USA, November 2008.
[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.
[25] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker, "Unexpected means of protocol inference," in *Proceedings of the 6th ACM SIGCOMM on Internet measurement (IMC'06)*. Rio de Janeriro, Brazil: ACM Press, 2006, pp. 313–326.
[26] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.
[27] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–.
[28] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," in *IEEE Symposium on Security & Privacy*, Oakland, CA, 2009, pp. 110–125.
[29] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.
[30] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.
[31] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.
[32] E. I. Organick, *The multics system: an examination of its structure*. MIT press, 1972.
[33] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications." in *NDSS*, 2014.
[34] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Third ACM Conference on Data and Application Security and Privacy*, 2013.
[35] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. 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, February 2014.
[36] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of oauth sso systems," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 378–390.
[37] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." in *NDSS*, vol. 2007, 2007, p. 12.
[38] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 365–379.
[39] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online–security analysis of cashier-as-a-service based web stores," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 465–480.
[40] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization." in *USENIX Security*, vol. 13, 2013.
[41] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.
[42] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.
[43] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations." in *NDSS*, 2013.
[44] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 393–408.
[45] Y. Zhou and D. Evans, "Ssoscan: Automated testing of web applications for single sign-on vulnerabilities." in *USENIX Security*, 2014, pp. 495–510.
[46] C. Zuo and Z. Lin, "Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th World Wide Web Conference*, Perth, Australia, April 2017.
[47] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.