OAUTHLINT: An Empirical Study on OAuth Bugs in Android Applications

Tamjid Al Rahat
Dept. of Computer Science
University of Virginia
U.S.A
tr9wr@virginia.edu

Yu Feng
Dept. of Computer Science
University of California, Santa Barbara
U.S.A
yufeng@cs.ucsb.edu

Yuan Tian
Dept. of Computer Science
University of Virginia
U.S.A
yuant@virginia.edu

***Abstract*—Mobile developers use OAuth APIs to implement Single-Sign-On services. However, the OAuth protocol was originally designed for the authorization for third-party websites not to authenticate users in third-party mobile apps. As a result, it is challenging for developers to correctly implement mobile OAuth securely. These vulnerabilities due to the misunderstanding of OAuth and inexperience of developers could lead to data leakage and account breach. In this paper, we perform an empirical study on the usage of OAuth APIs in Android applications and their security implications. In particular, we develop OAUTHLINT, that incorporates a query-driven static analysis to automatically check programs on the Google Play marketplace. OAUTHLINT takes as input an *anti-protocol* that encodes a vulnerable pattern extracted from the OAuth specifications and a program P. Our tool then generates a counter-example if the anti-protocol can match a trace of Ps possible executions. To evaluate the effectiveness of our approach, we perform a systematic study on 600+ popular apps which have more than 10 millions of downloads. The evaluation shows that 101 (32%) out of 316 applications that use OAuth APIs make at least one security mistake.**

***Index Terms*—Security, OAuth, Android, Static Analysis, Bug Finding**

I. INTRODUCTION

The OAuth protocol has been widely used for mobile developers to implement Single-Sign-On services. For example, many mobile game developers use OAuth to implement authentication with user's major accounts in Google, Twitter, or Facebook. However, the OAuth protocol was originally designed for the authorization for third-party websites in 2009, not to authenticate users in third-party mobile apps. Since the security implications for authentication and authorization are fundamentally different, and mobile platforms have different security schemes comparing to the browsers, it is challenging for developers to implement the OAuth protocols in mobile platforms securely.

Recent years, many widespread attacks for the OAuth implementation in popular mobile applications have been reported, causing users to lose their accounts or information. For example, Facebook has a bug for the insecure storage of OAuth tokens, which exposes a large number of user accounts to attackers [1]. Studying such a critical and popular multi-party protocol will also provide insight for building other multi-party protocols securely.

Due to the complexity of the OAuth protocol, it is challenging to implement it in a secure way. In particular, there are three parties in OAuth: the user who owns protected resources, the service provider that hosts the user's services, and the relying party that uses the service provider to get access to the user's resources or authenticate the user. During authorization, the three parties need to verify each other's identity and synchronize their secure states. However, despite wide deployment, the OAuth protocol is still too complicated for most developers to follow. For example, as we mentioned earlier, OAuth was first designed for *authorization*, but not for *authentication*. Therefore, the use case for authentication is not clearly defined in the original specification. As a result, developers may misunderstand the threat model and security implications in OAuth, which leads to vulnerabilities that compromise user privacy. In the meantime, the original OAuth 1.0 protocol [2] was first proposed for websites, and then got widely used in mobile applications. However, many security schemes cannot be directly mapped from the web to mobile platforms. Therefore, mobile application developers make many mistakes [3] for using the OAuth protocol.

Although an OAuth protocol typically involves multi-parties (i.e., user, relying party, and service provider), in this paper, we mainly focus on studying bugs in the relying parties and developing techniques for automatically finding those bugs. The ultimate goal is to provide a tool for helping developers to write secure authentication using the OAuth protocol. We focus on the relying parties for several reasons. First, comparing to the very few identity providers such as Google or Facebook, there are tens of thousands of relying parties in Android platforms and the developers of the relying parties contribute a majority of the common mistakes in using the OAuth protocol [3]. As a result, it is important to develop tools for helping numerous relying parties to check the security of their OAuth implementations. Second, mobile applications of most of the relying parties are publicly available through Google Play store. In contrast, the implementations of most of the identity providers are not available to us and we can only approximate their behaviors through black-box testing [4].

Previous studies [3], [5], [6] have identified many security issues of the mobile OAuth implementations by manual or semi-automatic analyses. However, the entire process is very

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00036

time-consuming and error-prone as it requires security experts with domain knowledge in OAuth to inspect the implementations manually. On the other hand, there are several challenges for automatically finding OAuth vulnerabilities in Android platforms. First, verifying OAuth implementation requires precisely modeling the data- and control-properties of mobile applications which are highly asynchronous and interactive. Second, to precisely reason about the OAuth vulnerabilities in mobile applications, our system has to consider the interactions among multiple parties (i.e., Service Provider, Relying Party, SDK, etc) and the majority of their implementations are not available. Finally, a system would need a specification for finding the OAuth vulnerabilities, however, the specification of the latest OAuth 2.0 protocol [7] has over 75 pages which are extremely difficult to digest by developers.

To understand the OAuth vulnerabilities in relying parties and help mobile developers to write secure code using OAuth, we perform the first empirical and systematic study on 600+ popular Android applications from Google Play. In particular, we propose OAUTHLINT, the first static analyzer for checking anti-protocols (i.e., vulnerabilities) in OAuth implementation. First, we present a simple but effective query language for describing common anti-protocols in OAuth implementations. Here, each anti-protocol is a class of vulnerabilities in OAuth implementation. Second, we perform a thorough study on specifications of the existing OAuth protocols and summarize five common *anti-protocols*. Third, given an anti-protocol expressed in our query language, we leverage a fully-automatic and demand-driven static analysis to identify the anti-protocols that appear in the Android applications from the relying parties. Formally speaking, an anti-protocol π for program P is true iff π matches a trace of P's possible executions. Our system will return a counter-example if the specified anti-protocol configuration is feasible in the application.

Most of the anti-protocols that we check are logical flaws due to the misunderstanding of the OAuth protocol. Because of the misunderstanding of the security implications of the OAuth protocol, the developers can make logical errors that compromise user privacy. For example, in the OAuth 2.0 implicit flow, the *bearer token* represents that a user grants a set of permissions to an application. By its design, the *bearer token* is bound to the user and the set of permissions, but not to the application. Therefore, if developers just verify the *bearer token*, it's not secure to authenticate the user because a malicious application might get the bearer token and reuse it in the OAuth process of another application to gain access to the user's account.

To evaluate the effectiveness of our approach, we perform a systematic study on 600+ popular applications which have more than 10 millions of downloads. Our evaluation shows that for those popular apps that use OAuth API, more than 32% of them contain at least one anti-protocol. For those anti-protocols identified by OAUTHLINT, we also reported them to the developers of corresponding mobile applications and get acknowledged by companies such as WordPress and GoFundMe.

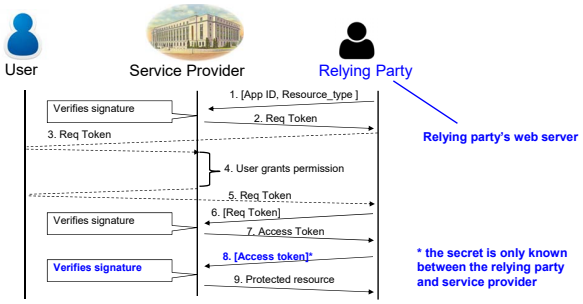

Fig. 1: Overview of OAuth 1.0.

Contributions. In summary, this paper makes the following key contributions:

- We devise a simple but effective query language for expressing common anti-protocols that violate the OAuth specification.
- We propose five anti-protocols that widely appear in the implementations of relying parties.
- Given an anti-protocol expressed in our query language, we design a demand- and query-driven static analysis for checking violations in the mobile applications from the relying parties.
- We implement our proposed ideas in OAUTHLINT, and evaluate it on over 600 popular applications from Google Play. Our result shows that more than 32% of the applications contain at least one anti-protocol.

II. BACKGROUND

The concept of OAuth protocol was proposed in 2007 and was designed as an authorization protocol which contains two major OAuth versions that are currently deployed: OAuth 1.0 and 2.0. The first version of the OAuth protocol (OAuth 1.0) was published in April 2010 [8]. Since then, the protocol has gone through a few revisions. The most notable changes to the protocol were released as the OAuth 2.0 framework in October 2012 [9].

A. OAuth 1.0

OAuth 1.0 was released for around 10 years, however, some service providers such as Twitter are still using OAuth 1.0 [10].

We illustrate the OAuth 1.0 protocol flow in Fig. 1. The dashed lines in our figures for OAuth represent redirection and solid lines represent direct server-to-server API calls (e.g., a REST API call). Also, parameters inside square brackets are signed using shared secrets.

We summarize the OAuth 1.0 flow in the following:

- **Initialization:** Relying party developers will need to register with the service provider and obtain shared secrets (consumer secret and consumer key). The shared secrets will be used in the following steps to sign packets and verify the signatures.
- **Unauthorized request token (Step 1,2):** First, the relying party gets a request token from the service provider using a direct server-to-server call.

- **Authorized request token (Step 3-5):** Then, the relying party redirects the user to the service provider (in mobile devices, it is done by inter-process communication such as Intent in Android) with the request token as a URI parameter. Then, the user grants the relying party access to their protected resource and is redirected back to the relying party.
- **Access token (Step 6,7):** With the request token authorized, the relying party can exchange the request token for an access token using another direct server-to-server call with the service provider. Note that these two steps are very critical for the security of OAuth 1.0. The relying party needs to sign the packet with the shared secret (consumer secret/key) and the service provider needs to verify the signature to check the identity of the relying party.
- **Protected resource (Step 8,9):** Finally, the relying party can use the access token to obtain the users protected resource.

B. OAuth 2.0

Instead of building on top of the existing OAuth 1.0 protocol, the working group changed the specification completely to create a different protocol, known as OAuth 2.0.

Compared to OAuth 1.0, OAuth 2.0 removed obtaining the shared secrets and providing signature as mandatory processes. Instead, OAuth 2.0 introduced the concept of *bearer token* [11]. For the bearer token, a users access token was no longer bound to a relying party; any party with this token could access the users protected resource.

In addition, OAuth 2.0 also offers four different flows, these methods are referred to as grants and they can be viewed as different "versions of OAuth 2.0. Out of the four OAuth grants, the most popular one is the implicit grant.

In the following, we will use Fig 2 to explain the most popular implicit grant of OAuth 2.0 [9].

The implicit grant is the shortest of all OAuth 2.0 flows. It consists of two steps. First, the user is redirected to the service provider to grant the relying party access to his/her protected resource. After the user grants the permission, the service provider redirects the user back to the relying party along with an access token as a parameter in the URI. The relying party can then use this access token to exchange for the users resource. Besides the implicit grant, we find that the authorization code grant is also used in mobile applications we studied. Comparing to the implicit grant, the authorization code grant has additional steps for the relying party to obtain an authorization code and then use the authorization code to exchange for the token.

There are two important differences in the implicit grant comparing to other OAuth flows. First, with exception to the final protected resource request, every message in the protocol is exchanged through the user agent (e.g., using browser redirection or Android Intent). Second, the implicit grant does not require the relying party to present a shared secret to the service provider. This is ideal for the mobile environment, where the relying party resides on an untrusted device.

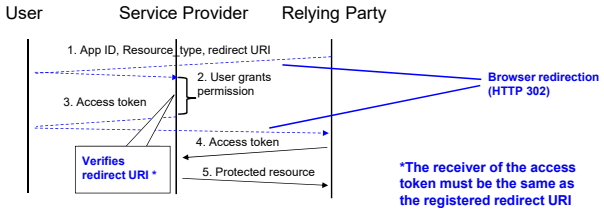

Fig. 2: Overview of OAuth 2.0.

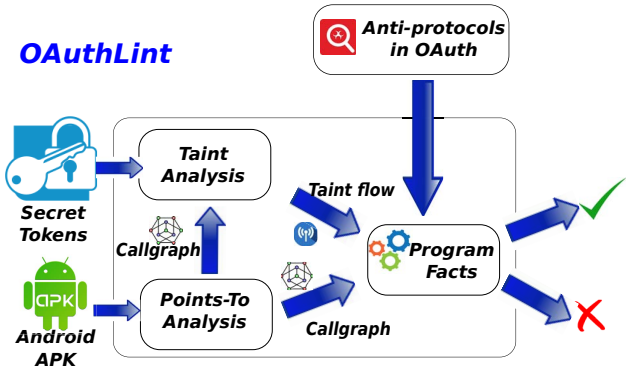

Fig. 3: Overview of OAUTHLINT.

C. Using OAuth for authentication

Although OAuth is originally designed for authorization, developers re-purposed it for authentication. Therefore, the way to do authentication using OAuth is never documented in the OAuth standard protocol and developers have to figure out their own ways to run the authentication using OAuth. Typically, they just changed the last step of the OAuth protocol, using the user's resources to identify the user. Some service providers such as Google and Facebook recognized the limitation of using OAuth for authentication and propose additional steps of verification to improve the security, such as the appsecret_proof [12].

III. OVERVIEW

In this section, we give an overview of our approach with the aid of a motivating example, and then summarize the threat model in our system.

A. System overview

Fig. 3 gives a high-level overview of OAUTHLINT's architecture. In particular, OAUTHLINT incorporates standard pointer and taint analyses to explicitly track sensitive information (e.g., secret keys, access tokens, etc.) in OAuth protocols.

*a) **Pointer analysis**:* Given the source code or bytecode of an Android application, OAUTHLINT leverages FLOWDROID [13] to perform (field- and object-sensitive) pointer analysis to build a precise call graph and identify all variables that may be an alias to each other. The call graph and alias

information are further used by the subsequent taint analysis for generating additional program facts that are relevant to answering the *anti-protocols* in OAuth.

b) ***Taint analysis****:* The taint analyzer leverages the annotations in OAUTHLINT's configuration file to determine taint sources (i.e., secret keys and access tokens) and propagates them using a field- and object-sensitive analysis. Intuitively, a taint flow encodes that a sensitive field (e.g., secret keys) may flow to an untrusted resource (e.g., local storage, WebView, etc.).

c) ***Anti-protocols for OAuth****:* To find logical flaws due to the misunderstanding of the OAuth protocol, OAUTHLINT first proposes a query language that enables developers to specify *anti-protocols*. These anti-protocols represent the OAuth-implementation mistakes by developers that may result in severe vulnerabilities such as impersonation attack and users' privacy violation. Here, each anti-protocol is expressed as a logical formula which encodes a class of vulnerabilities that compromise user's security and privacy. Section V includes detailed explanation and security impacts of five anti-protocols that widely appear in existing Android apps in Google Play Store. While we propose five anti-protocols using the query language in OAUTHLINT, a security expert can come up with more anti-protocols based on her insight on the standard OAuth specifications. After that, using the program facts generated by our previous analyses, OAUTHLINT leverages a fully-automatic and demand-driven static analysis for checking whether there exists an execution trace such that the anti-protocol holds. If so, a violation will be reported to the developer.

For instance, *TikTok*, a very popular app for creating and sharing short videos, has over 500 million installs by the time of our submission. To integrate the user accounts from service providers such as Facebook, Instagram, and Google, *TikTok* implements the standard authentication flow using the OAuth 2.0 protocol [9]. However, after running OAUTHLINT on *TikTok*, we found the application contains multiple logical flaws in their implementation for the OAuth protocol.

Firstly, *TikTok* bundles its consumer key and secret within the application code. The following code snippet from *TikTok* represents one of the most common ways that developers use to bundle their secrets:

```
1 final String CONSUMER_KEY =
2    "YYWjeT***...";
3 final String CONSUMER_SERECT =
4    "w981H5bEd***...";
5 ...
```

According to the specification in OAuth 1.0 [8], the consumer secrets or keys should never be bundled in the clients. The reasons are as follows: first, the consumer secrets are highly sensitive information shared between service providers and relying parties, and service providers will use consumer keys to verify identities of the relying parties. Second, all mobiles devices are technically untrusted, which means that an attacker could extract the consumer secrets through reverse engineering and impersonate a benign replying parties to get access to users' information.

Moreover, *TikTok* embeds a WebView to retrieve the access token attached with the redirect URL from Twitter and Instagram.

```
1  String url = "provider.com/..?
2     client_id=".."
3     &redirect_url=".."
4     &response=token";
5  ...
6  webView.loadUrl(url);
7
8  void onPageStarted(String url){
9    String token = parseToken(url);
10 }
```

When an application hosts service provider's website inside a Webview, it gets full access to the information such as users' cookies. Therefore, using WebView for OAuth enables a malicious relying party app to log into a victim user's account with the service provider. This is due to the fundamental design of isolation in WebView and there is no way for the service provider to protect herself when it was loaded in WebView. Often service providers use long term cookies, which makes such vulnerability persistent.

Furthermore, OAUTHLINT also detects that *TikTok* exchanges its access token with service providers to fetch the corresponding userId for login purpose.

```
1 String user_id = fetchUserInfo(token
2                  , consumer_secret);
3 storeInSharedPreferences(user_id
4                  , token);
5 ...
6 authorizeUserLogin(user_id);//login
```

This approach is also vulnerable because any requests made from a client could be potentially tampered by an attacker. Hence, for authentication using OAuth, client devices should not be trusted. According to OAuth 1.0 and 2.0, a secure authentication should be made through server-to-server calls.

Finally, after obtaining the userId, *TikTok* stores both userId and access token as plain text in the SharedPreferences. Here, there are multiple security issues. First, information stored in SharedPreferences is insecure, since they can easily be accessed by another malicious application in a rooted device or emulator. Hence, any sensitive information should not be stored in SharedPreferences. Second, storing sensitive information such as access token as plain text is insecure. According to the suggestion from OAuth 1.0 and 2.0, the access tokens should be encrypted and stored on the server side of the relying party.

We have reported all the above-mentioned flaws to *TikTok*'s security team and they are still working on those issues by the time of this submission.

B. Threat Model

In this paper, we focus on the vulnerabilities in the design and implementation of the relying parties for using OAuth protocols. We assume the service providers are trusted and correctly implement the security schemes for OAuth. For the attacks to authentication, we consider the case where an attacker has the capability of accessing the mobile apps on behalf of the victims. For the attacks to authorization, we consider the case where malicious relying parties seek to access user's data without consensus.

IV. Analysis

This section describes OAuthLint's static analyses for computing an over-approximation of some built-in predicates that encode the data- and control-dependency of a program. In what follows, we first give some background on Datalog, and then describe the syntax and semantics of OAuthLint's built-in predicates.

A. Datalog Preliminaries

A Datalog program consists of a set of *rules* and a set of *facts*. Facts simply declare predicates that evaluate to true. For example, `parent("Bill", "Mary")` states that Bill is a parent of Mary. Each Datalog rule is a Horn clause defining a predicate as a conjunction of other predicates. For example, the rule:

```
ancestor(x, y) :- parent(x, z), ancestor(z, y).
```

says that `ancestor(x, y)` is true if both `parent(x, z)` and `ancestor(z, y)` are true. In addition to variables, predicates can also contain constants, which are surrounded by double quotes, or "don't cares", denoted by underscores.

Datalog predicates naturally represent relations. Specifically, if tuple (x, y, z) is in relation A, this means the predicate $A(x, y, z)$ is true. In what follows, we write the type of a relation $R \subseteq X \times Y \times \ldots$ as $(s_1 : X, s_2 : Y, \ldots)$, where $s1$, $s2$, ... are descriptive texts for the corresponding domains.

Base Facts. The base facts of our inference engine describe the instructions in the application's control-flow graph (CFG). The base facts take the form of $\mathrm{A}(y, x_1, ..., x_n)$, where `A` is the instruction name, `y` is the variable storing the instruction result (if any), and $x_1, ..., x_n$ are variables given to the instruction as arguments (if any). For example, the instruction $r_1 = 0$ is encoded to $\mathsf{assign}(r_1, 0)$. Further, the special local store instruction $\mathsf{lstore}(d, v)$ denotes that the value of v is stored in location d.

Flow-to Predicates. The `flowTo` predicate is defined over pairs of variables and is inferred from the application's CFG. The intuitive meaning (also summarized in Fig. 4) is: $\mathsf{flowTo}(x_1, x_2)$ holds for x_1 and x_2 if the value of variable x_2 depends on the value of x_1.

OAuth-specific Predicates. In addition to base facts from programs, OAuthLint also defines a list of predicates that are specific to the OAuth domain. As shown in Fig. 5, `isToken(x)` denotes that x may be an access token.

$$
\begin{array}{lcl}
\mathsf{FlowTo}(x, y) & :- & \mathsf{alloc}(y, x) \\
\mathsf{FlowTo}(x, y) & :- & \mathsf{assign}(y, x) \\
\mathsf{FlowTo}(x, z) & :- & \mathsf{assign}(y, x), \mathsf{FlowTo}(y, z) \\
\mathsf{FlowTo}(x, z) & :- & \mathsf{alias}(y, z), \mathsf{FlowTo}(x, y)
\end{array}
$$

Fig. 4: Rules for computing the **Flow-to** predicate. Here, the alias predicate **alias** is directly obtained from FlowDroid.

Since it is difficult to precisely pinpoint strings that corresponds to access tokens, we use both pattern matching (i.e., regular expressions) and domain-specific knowledge (i.e., API that may return an access token) to over-approximate the domain of access tokens. Similarly, client secret is encoded as `isSecret(x)`. Furthermore, statement `y = new SecretKeySpec(x)`, which is used to construct a client secret, is represented as `SecretKeySpec(y, x)`. Finally, `isLocalStore(x)` represents a location for local store, which can be Android `SharePreferences` or file systems.

$$
\begin{array}{rcl}
\mathsf{isToken}(x) & :- & x \text{ may be an access token} \\
\mathsf{isSecret}(x) & :- & x \text{ may be a secret} \\
\mathsf{webView}(x) & :- & x \text{ is the URL of a WebView} \\
\mathsf{secretKey}(y, x) & :- & \mathsf{SecretKeySpec}(y, x) \\
\mathsf{Http}(y, x, \_) & :- & y \text{ is an HTTP object of which} \\
 & & \text{arguments contain } x \\
\mathsf{lstore}(x, z) & :- & \mathsf{isLocalStore}(x), \mathsf{FlowTo}(z, x) \\
\mathsf{login}(x, y) & :- & \text{Login with user x and password y}
\end{array}
$$

Fig. 5: Rules for computing the OAuth related predicates.

V. Anti-protocols in OAuth

This section describes OAuthLint's anti-protocol language, which is a Datalog program augmented with builtin predicates. We propose five anti-protocols that widely appear in existing mobile apps and formalize them using our language. To identify the pattern of these anti-protocols, we manually reverse engineered 43 Android apps from Google Play Store. The anti-protocols are designed with Android and Java APIs, and therefore are not biased with any particular application. For each anti-protocol, the user defines a unique predicate that serves as the signature for this anti-protocol. In what follows, we first describe the syntax of OAuthLint's built-in predicates, and then discuss five common anti-protocols in OAuth.

A. An anti-protocol language for OAuth

We first introduce our anti-protocol language for OAuth, shown in Fig. 6. Here, `arg`, `reg`, and `mem` are variables from function arguments, registers, and memory, respectively. The predicate `flowTo` determines the data dependency between two variables, as specified in Section IV. Finally, we can express more complex queries by composing simple expressions with logical operators (i.e., $\neg, \wedge, \vee$, etc.).

$$
\begin{array}{rcl}
v_i & ::= & \mathsf{arg} \mid \mathsf{reg} \mid \mathsf{mem} \mid ... \\
\phi & ::= & \mathsf{flowTo}(v_1, v_2) \mid \mathsf{isToken}(v_1) \mid \mathsf{isSecret}(v_1) \\
& \mid & \mathsf{secretKey}(v_1, v_2) \mid \mathsf{Http}(v_1, v_2, v_3) \mid \mathsf{lstore}(v_1, v_2) \\
& \mid & \mathsf{webView}(v_1) \mid \mathsf{login}(v_1, v_2) \\
& \mid & \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi
\end{array}
$$

Fig. 6: The Anti-protocol language in OAUTHLINT.

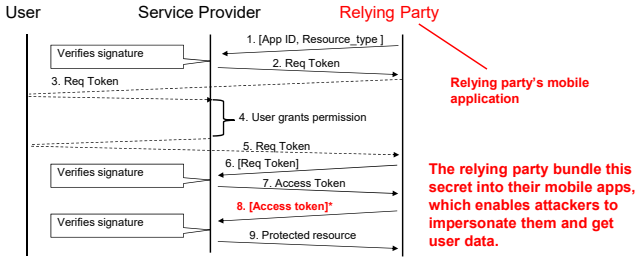

Fig. 7: Vulnerability of locally bundled *client secret* OAuth 1.0 flow. Parameters inside the square brackets are typically signed by client secret.

B. Common Anti-protocols in OAuth

1) Locally Bundled Client Secrets: Relying party secret, which is often referred to as the *client secret* by OAuth 2.0 and *consumer secret* by OAuth 1.0, is used by service providers to authenticate the relying party. Developers can obtain the relying party secret from the developer's console of the service providers when they register their application. Many developers misunderstand the purpose of the relying party secrets and store them locally on client-side applications. If a developer bundles the relying party secret with her mobile application, an attacker can easily retrieve it through reverse engineering, and use this secret to get her own application to be authenticated by the service providers as a benign application. Fig. 7 use OAuth 1.0 as an example to illustrate the vulnerability of locally bundled client secret. Relying party gets the client secret from the service provider when it registers for the OAuth service, and the client secret is then used to generate the signature from the token in step 8. Since the service provider verifies the signature to check the identity of the relying party, if the relying party bundles the secret in their mobile app, the attacker can do simple reverse engineering to extract the client secret and impersonate the relying party (i.e., victim) to access the user's protected resources.

In practice, OAUTHLINT identifies that many developers bundle their relying party secrets as field variables, resource files, or constants in their application code. For instance, here is an example from the GoFundMe application [14] which hard-codes its secret for communicating with Twitter:

```
String consumer_secret =
    "QfMu9***...";
mac = Mac.getInstance("HmacSHA1");
key = encodeParam(consumer_secret);
mac.init(new SecretKeySpec
(key.getBytes(), "HmacSHA1"));
HttpURLConnection c =
new URL(url).openConnection();
params = new ArrayList();
params.add("signature="
    +getSignature(mac,token,..));
...
r = c.getInputStream();
```

The locally store client secrets can be encoded as the following anti-protocol in OAUTHLINT:

$$\mathsf{isSecret}(x) \wedge \mathsf{secretKey}(y, z) \wedge \mathsf{flowTo}(x, z) \\ \wedge \mathsf{Http}(\_, u, \_) \wedge \mathsf{flowTo}(y, u)$$

In addition to the above scenario, developers also store relying party secrets in Android *SharedPreferences*, which is also insecure:

```
String appSecret = getAppSecret();
SharedPreferences sf = getActivity()
    .getPreferences(mode);
SharedPreferences.Editor editor =
    sf.edit();
editor.putString("app_secret",
    appSecret);
editor.commit();
```

Since the data in *SharedPreferences* are stored in the file system, an attacker can access the secret by other malicious applications if the device is rooted. OAUTHLINT uses the following anti-protocol for checking this variant:

$$\mathsf{isSecret}(x) \wedge \mathsf{lstore}(\_, z) \wedge \mathsf{flowTo}(x, z)$$

where `lstore` represents all untrusted locations.

2) Using WebView for OAuth Transactions: Using WebView for OAuth transactions is insecure because WebView breaks the isolation between the service provider and the relying party. When a malicious relying party hosts the service provider's website in the WebView of their mobile applications, the malicious relying party can get the user's cookies to log in on behalf of the user. Fig. 8 illustrates an example of using Webview for Facebook login. Webview enables the hosting app (malicious relying party) to access the cookies of the service provider such as Facebook. With the cookies, the malicious relying party can log in on behalf of the user. This attack can be persistent if the service providers such as Google and Facebook use long term cookie for better user experiences, which is quite common.

In practice, we found many developers use embedded WebView or mobile browsers for user redirection in OAuth, which endangers user privacy. For example, here is a code snippet from the Waplog application [15]:

```
WebView webview;
String url = "provider.com/..?
    client_id=".."
    &redirect_url=".."
```

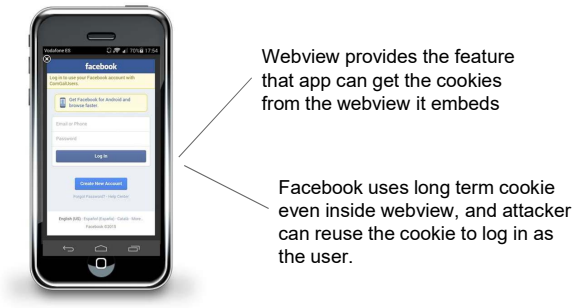

Fig. 8: Vulnerability of using Webview for OAuth transactions.

```
    &response=code";
...
public void onCreate() {
    webview.loadUrl(url);
}
```

Using WebView for OAuth transactions can be encoded as the following anti-protocol:

$$\mathsf{isSecret}(x) \wedge \mathsf{isToken}(y) \wedge \mathsf{webView}(z) \\ \wedge \mathsf{flowTo}(x, z) \wedge \mathsf{flowTo}(y, z)$$

3) Client-side API Call: Client devices should not be trusted during API calls that are involved in the flows for OAuth authentication. In other words, relying parties should always assume that any requests made from client devices could be tampered by the attackers. Unfortunately, in reality, developers often misunderstand the security implication and assume that the access token granted by the service providers are only bounded to the relying party. As a result, an attacker could leverage the access token granted for some malicious applications and login as a user for other benign applications to access other sensitive information. For example, the Topface application [16] exchanges access token for user id by doing API call from the application to authenticate users.

```
String url = "api.provider.com/..";
HttpURLConnection c =
new URL(url).openConnection();
c.setRequestMethod("POST");
...
params = new ArrayList();
params.add("oauth_consumer_key="+
    client_id);
params.add("oauth_token="+
    access_token);
params.add("oauth_signature="+
    getSignature(client_secret));
...
c.setRequestProperty("Authorization
    ",createHeaders(params));
String user_id =
parseJSON(c.getInputStream(),"id");
newUserLogin(user_id);
```

We express the Client-side-API-Call vulnerability using the following anti-protocol:

$$\mathsf{isToken}(x) \wedge \mathsf{isSecret}(y) \wedge \mathsf{Http}(r, u, \_) \wedge \mathsf{login}(v, \_) \\ \wedge \mathsf{flowTo}(x, u) \wedge \mathsf{flowTo}(y, u) \wedge \mathsf{flowTo}(r, v)$$

4) Storing Access Tokens on Client Devices: By the end of a typical OAuth transaction, the relying party receives an access token, which is a raw string that can be used to make API calls to retrieve protected resources from the service provider. Stealing access token provides an ideal vector through which an attacker can compromise user accounts and harvest confidential data such as email and contacts. Some service providers (e.g., Google) also allow access to user's files stored in the cloud via access tokens [17]. More importantly, an access token does not require user's password and is capable of bypassing any two-factor authentication. To make things worse, the only way to revoke an attacker's access is to explicitly revoke access to the malicious application that uses the access token to launch attacks. Thus, having access token obtained by attackers could have an adverse impact on users. During U.S. Presidential Election in 2016, one of the tactics attackers used was collecting OAuth access tokens, as reported [18] by security experts from FireEye.

Our analysis finds that many developers often do not encrypt their raw access tokens before storing them to client devices, using *SharePreferences* or files in the external storage. This is insecure, as data in *Sharedpreferences* or filesystem and can easily be accessed from any rooted device or an emulator. For example, the Chatous application [19] stores the access token received from Instagram as follows:

```
String aToken = getAccessToken();
SharedPreferences sf = getActivity()
    .getPreferences(mode);
SharedPreferences.Editor editor =
    sf.edit();
editor.putString("access_token"
    ,aToken);
editor.commit();
```

Storing access token on client devices can be encoded as the following anti-protocol:

$$\mathsf{isToken}(x) \wedge \mathsf{Istore}(\_, u) \wedge \mathsf{flowTo}(x, u)$$

5) Sending Raw Access Token to Server: To make server-to-server API calls during OAuth transactions, developers send the access token to the relying party server. However, if a raw (i.e., unsigned) access token is sent to the backend server, a modified client application can send arbitrary access token and initiate an impersonation attack.

A client device is assumed to be untrusted when OAuth is used for authentication. Thus, instead of making API call directly, a mobile application should communicate with its own backend server and pass the access token to the server. The server would then use the access token to make API calls to communicate with the resource server of the corresponding service provider. However, it is important for developers to

understand that access tokens are portable. Once an access token is received, it can be used from both applications and server to fetch users' resources. Thus, sending an unsigned access token to the server can lead to *Token Hijacking*. An example from the Wish application [20] that sends the raw access token to the backend server is given below:

```
1  String aToken = getAccessToken();
2  HttpClient httpClient =
3     new DefaultHttpClient();
4  HttpPost httpPost =
5     new HttpPost("/backend.com/
          tokensignin");
6  params = new ArrayList(1);
7  params.add(new BasicNameValuePair
8     ("access_token", aToken));
9  httpPost.setEntity(new
10    UrlEncodedFormEntity(params));
11
12 httpClient.execute(httpPost);
```

Sending raw access token to server can be encoded as the following anti-protocol:

$$\mathsf{isToken}(x) \wedge \mathsf{Http}(\_, u, \_) \wedge \mathsf{flowTo}(y, u)$$

Google recently added a security notice to address this vulnerability in their official documentation [21]. To authenticate users on Google's backend servers, they recommend developers to send ID-token ((returned by *GoogleSignInAccount.getIDToken()*)) which is signed by Google's public keys. If backend server receives sensitive information such as access token in plain text, a modified application can send an arbitrary token to the server and thereby, initiate an impersonation attack.

VI. Experimental Evaluation

To evaluate the effectiveness of OAuthLint, we conduct a series of experiments to answer the following questions:

- How effective is OAuthLint at identifying real-world vulnerabilities in OAuth implementations?
- How prevalent are those anti-protocols discussed in section V?
- How do the real-world OAuth vulnerabilities look like?

A. Data Collection

To find the current scenario of vulnerable implementation of OAuth, we have analyzed 600 latest Android applications (shown in Table I) collected from Google Play Store. Our unbiased selection of applications includes top 300 free applications from all categories, top 200 free social applications, and top 100 free communication applications. The reason why we include more applications in the social and communication category is that these applications are usually more likely to use OAuth. Most applications use OAuth SDKs from service providers such as Google and Twitter. Table III shows the detail of the statistics. All applications were collected in April 2019.

Our analysis found that 316 out of the 600 applications use OAuth with at least one Service Provider. We built a Soot-based sanitizer that automatically analyzes the Dalvik bytecodes to filter out applications not using any relevant classes or APIs provided by the Service Providers. We included 20 most popular Service Providers for this sanitization step.

Category	#apps	#apps (OAuth)	#apps (Vulnerable)
Top Free	300	178	39
Social	200	109	48
Communication	100	29	14
Total	600	316	101

TABLE I: Top Android applications using OAuth

B. Results

OAuthLint successfully analyzed, in total 316 applications that use at least one OAuth service provider. Among the remaining 284 applications, 273 applications did not use any OAuth implementation and 11 applications ran out of memory during the analysis. In average, total runtime for each application was 282.6 seconds and maximum memory consumption was 1931.06 MB during the analysis. All results mentioned here are in reference to the 316 successfully analyzed applications.

Table II lists the number of vulnerable applications for different OAuth vulnerabilities. Total vulnerable applications with distinct number of vulnerabilities are illustrated in Fig. 9. We discuss the results for each vulnerability below:

Locally Bundled Client Secrets. OAuthLint successfully identified 29 applications that bundle the consumer key/secret of at least one service provider within the application code. We found 18 applications bundled Twitter consumer key and secret, 7 applications bundled the client id and client secret from Instagram, 3 applications bundled the facebook app secret, and 12 applications bundled the consumer keys/secrets from other service providers in the source code.

Using WebView for OAuth Transactions. This vulnerability is a common scenario for the application that implements OAuth in an embedded WebView for the web-based service providers. By hosting the service provider's website in the WebView of the relying party's mobile applications, the relying party can interact with the service provider easily. In addition, a malicious relying party can access the user's cookies in the service provider website to log into the user's account. OAuthLint successfully identified 24 applications who use WebView for the authorization and authentication transactions of OAuth. 9 applications choose to implement the web-based OAuth version instead of app-based version, even though corresponding service providers provide official SDK particularly for app-based OAuth implementation.

Client-side API Call. When using OAuth for authentication, developers should pass the encrypted access token received from the relying party application to the backend server of the same relying party and then, should verify the access token with the service provider by doing server-to-server API call. OAuthLint identified 69 applications that violated this

secure approach and performed the authentication by making client-to-server API call. For authentication, client device should not be trusted as an API call made from the client side could be tampered by a malicious user.

Storing Access Token on Client Device. OAUTHLINT identified 21 applications that stores plain (i.e., unencrypted) access token in Android *SharedPreferences*. Access tokens are portable, which means an access token obtained from a relying party application can also be used from any client, server or otherwise. Hence, if malicious attackers retrieve the access token from a client device, they can use it from a different machine to launch an impersonation attack. Storing sensitive access token in insecure *SharedPreferences* gives partial security, as data stored in *SharedPreferences* can be easily accessed by any applications or user from a rooted device or emulator. Android provides a *KeyStore* system to enhance the security for storing such sensitive information such as access token.

Sending Raw Access Token to Server. Right after obtaining the access token at the authorization endpoint, relying party application should send the access token to their backend server in order to exchange it for user's resources (i.e., user id). However, to this end, relying party server should not trust data received from the application. A malicious attacker could modify the application to tamper the access token and hence, initiate an impersonation attack to the relying party server. OAUTHLINT identified 17 applications that send the plain access token to the relying party server.

Finally, we note that the analysis in OAUTHLINT is precise. As shown in Table II, the average false positive rate is 10%. We manually inspected the false alarms and confirmed that most of them are caused by the imprecision of the pointer analysis and call-graph construction.

Vulnerability	**#apps**	**#FP**
Locally Bundled Client Secrets	29	2
Using WebView for OAuth Transactions	24	3
Client-side API Call	69	7
Storing Access Token on Client Device	21	3
Sending Raw Access Token to Server	17	1

TABLE II: OAuth vulnerabilities in top android applications

OAuth API	Total Apps	Vulnerable Apps	Oauth1.0 or 2.0
Facebook	301	52	2.0
Google	295	11	2.0
Twitter	34	23	1.0a, 2.0

TABLE III: Statistic of top OAuth APIs in our evaluation

C. Case Studies

CBS Sports. To estimate the impact of vulnerable implementations of OAuth, we manually inspected the applications that were reported as vulnerable by OAUTHLINT. *CBS Sports* [22] is one of the most popular applications in U.S. for top sports news, scores, and videos. According to Google Play, this application was installed more than 10 millions times. This application got flagged by OAUTHLINT

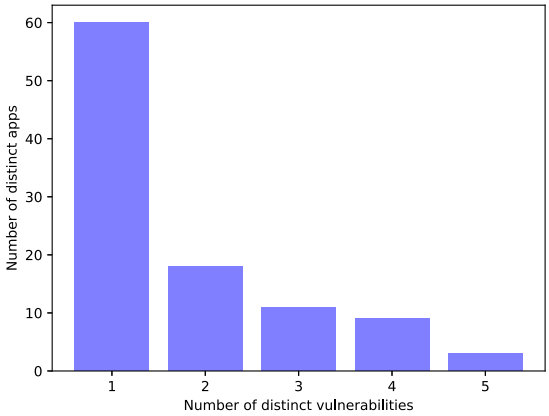

Fig. 9: Number of applications with 1, 2, ...5 vulnerabilities.

for three reasons. First, it bundles the twitter consumer key and consumer secret within the application code, which makes it vulnerable for impersonation attack with the authorization transactions of OAuth. Secondly, it uses an embedded WebView for OAuth transactions between relying party application and service provider. Currently, there exists no secure way for the web-based service provider to deliver the sensitive OAuth data to the honest relying party application. Hence, using WebView for OAuth transactions makes it very difficult for the service provider to determine the identity of the *CBS Sports* application. Thirdly, the application uses client-side API calls to authenticate new users. Client device must not be trusted during the OAuth transactions for authentication since corresponding API calls could be tampered by malicious users.

Topface. We also performed a thorough inspection with *Topface* [16], one of the most popular social application in Google Play. This application allows users to meet new people online and was installed more than 10 million times. However, *Topface* provides authentication using OAuth for various service providers including Google, Facebook, Instagram, and Vkontakte. The application was flagged by OAUTHLINT for several reasons. Even though it provides secure intent-based OAuth transactions for Google and Facebook, it uses WebView for both authorization and authentication transactions for Instagram and Vkontakte. More importantly, it performs all API call for OAuth transactions from client-to-server instead of server-to-server call, which allows a malicious user to tamper sensitive information such as *access token* and *user id*. The application also stores plain access token, user id and email address in the *SharedPreferences* without doing any encryption. Storing these sensitive information in *SharedPreferences* gives partial security since any data stored in *SharedPreferences* can be easily accessed from any rooted devices and emulators.

VII. Related work

Since OAuth is a critical protocol for authentication and authorization, many researchers have studied the implementation of OAuth and discovered many prominent attacks for web applications [23]–[27]. In recent years, mobile developers also use OAuth to build authentication or authorization schemes, and they have many misunderstandings of the security implications of OAuth. Researchers also did field studies to identify the vulnerabilities due to the implementation errors of OAuth in mobile apps. Chen et al. [3] present a comprehensive study on implementation errors and misunderstandings regarding OAuth protocols in mobile applications. Their study also shows that over 60% of mobile OAuth implementations have at least one vulnerability. Also, Shehab et al. [6] analyze source codes of Android applications and demonstrate possible attacks in OAuth implementation. However, these studies depend on dedicated manual analysis by security experts such as inspecting the network traffic and inferring the protocol flows and cannot be scaled to large-scale studies. Furthermore, researchers also studied user's privacy risks during OAuth transactions [28] Comparing to these works, our system is designed to automatically identify the vulnerabilities of mobile OAuth implementations efficiently and effectively.

Realizing that OAuth is a critical problem, researchers also propose solutions to improve the security of OAuth. For example, Yang et al.'s work [4] is the most relevant one, they build an automatic testing tool using symbolic execution to check the correctness of 10 popular OAuth SDKs and identified 7 vulnerabilities. In comparison, our study focuses on the implementation errors of the relying party because the developers of the relying parties are more easily to misunderstand OAuth and make mistakes comparing to the developers of popular SDKs. Researchers also propose to run automatic traffic analysis to identify OAuth implementation errors [29], however, this kind of approach will fail to point out the details of the implementation errors. Compared to their work, OAuthLint can identify the details of the implementation errors, which is very helpful for the developers to fix the security issues. Wang et al. propose a tool that combines static analysis and network analysis to identify OAuth bugs, however, their tool is semi-automatic and requires manual work to identify the vulnerabilities [30]. Applying formal analysis to the OAuth protocols is also another way to improve the security of the protocols [31]–[33]. However, though these papers elegantly model the OAuth protocol, the level of abstraction in these papers make it difficult to detect implementation errors.

Detecting Implementation Mistakes In recent years, researchers have been studying the problem of implementation errors in many security applications such as SSL, and crypto libraries. For example, CryptoLint [34] did a systemic analysis of the incorrect usage of crypto API, and SSLLint [35] did program analysis to identify the errors of implementing SSL. Moreover, researchers also utilized demand-driven static analysis for other domains. Arzt et al. present Flowdroid [36] - a static taint-analysis tool designed to identify data leaks in Android applications. Feng et al. [37] implement a tool to answer a class of interprocedural control flow queries about Java programs. Martin et al. [38] present a query language (PQL) that uses context-sensitive pointer alias analysis to mine information (e.g., bugs) from a program. Sridharan et al. [39] introduce a demand-driven points-to analysis algorithm that outperforms the previous techniques. Feng et al.'s [40] work implements a semantic-based approach that identifies an Android malware that leaks the user's private information. Compared to these papers, OAuthLint needs to overcome more challenges because OAuth is a multi-party protocol and we don't have access to the data from all parties.

VIII. Best Security Practice for OAuth Relying Party

According to our analysis, 101 of the OAuth relying party implementation suffers from at least one vulnerability. We hope this study can help to provide more guidelines for the mobile OAuth developers, especially for the relying party developers.

For securing the OAuth protocol, there are two general major points : (1) be aware that the security of OAuth partially lies in its access token delivery methodology; (2) never trust the mobile client because it might belong to a malicious user who can access the secret data and temper the verification results or data. For example, if developers are developing OAuth 2.0 in Android, instead of using the default Intent scheme to deliver the access token, they should use the developer key hash in order to check the identity of the party that receives the token. If developers are developing OAuth 1.0, they also need to make sure they can verify the token receiver's identity correctly. The relying party should never bundle the developer's consumer secret and consumer key into its mobile app because a malicious user can just extract the secret and key from the app and pretend to be the relying party to access user data.

For using OAuth to do authentication, the relying party needs to be more careful about dealing with the users. First, the relying party must not bundle any security related protocol logic (e.g., security checks) or any sensitive information (e.g., the token) into its own mobile application. Second, the relying party must assume that the attacker could tamper with any data sent from the users device. Because of this, the relying party must check that the relying party receiving the users ID in the last step of the protocol is the same relying party that the user intends to authenticate to. For example, instead of using the default implicit flow of OAuth 2.0 for authentication, the relying party should do an additional verification step such as described in the *appsecret_proof* flow of Facebook OAuth guidelines [12].

IX. Discussion

Like any other program analysis tool, OAuthLint has a number of limitations:

OAuthLint proposes a query language that helps developers to define OAuth-based anti-protocols. In this paper, we

explore five anti-protocols, each of which represents a class of vulnerabilities in OAuth implementation that widely appears in Android apps. However, defining a new anti-protocol that covers a wide range of vulnerable implementation scenarios may seem tedious for a security-expert. Also, applications having complex logical implementation flaws may still be undetected by OAUTHLINT. In addition to that, service providers typically allow developers to make certain changes in security settings of the developer's panel. Cross-checking between program facts in relying party apps and security settings in developers panel is beyond the limit of OAUTHLINT.

X. CONCLUSIONS AND FUTURE WORK

In this paper, we propose OAUTHLINT, the first static analyzer for checking anti-protocols (i.e., vulnerabilities) in mobile OAuth implementation. We propose five anti-protocols that widely appear in mobile apps and formalize them using our query language. Furthermore, we leverage a fully-automatic and demand-driven static analysis to identify anti-protocols that appear in the Android apps from the relying parties. To evaluate the effectiveness of our approach, we perform a systematic study on 600+ popular apps which have 10 millions of downloads. Our evaluation shows that for those popular apps that use OAuth API, more than 32% of them contain at least one anti-protocol. For those anti-protocols identified by OAUTHLINT, we also reported them to the developers of corresponding mobile apps.

There are several future directions that we plan to explore. First, we will develop techniques to automatically repair Android apps that have vulnerabilities in their OAuth implementations. Second, we are also very interested in applying program synthesis to perform a correct-by-construction paradigm for OAuth implementations. In that case, developers only need to specify the OAuth specifications using a high-level domain specific language and let the synthesizer generate the implementations.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their insightful comments. This work is supported in part by National Science Foundation under the agreement number of 1920462, 1850479, and 1908494.

REFERENCES

[1] Facebook: Security update. https://newsroom.fb.com/news/2018/09/security-update. Accessed: 05-11-2019.
[2] Oauth 1.0. https://oauth.net/1/.
[3] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 892–903, 2014.
[4] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. Vetting single sign-on SDK implementations via symbolic reasoning. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1459–1474, 2018.
[5] Wanpeng Li and Chris J. Mitchell. Security issues in oauth 2.0 SSO implementations. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, pages 529–541, 2014.
[6] Mohamed Shehab and Fadi Mohsen. Towards enhancing the security of oauth implementations in smart phones. In *IEEE Third International Conference on Mobile Services, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 39–46, 2014.
[7] Oauth 2.0. https://oauth.net/2/.
[8] Eran Hammer-Lahav. The oauth 1.0 protocol. *RFC*, 5849:1–38, 2010.
[9] Dick Hardt. The oauth 2.0 authorization framework. *RFC*, 6749:1–76, 2012.
[10] Oauth with the twitter apis. https://developer.twitter.com/en/docs/basics/authentication/overview/oauth.html. Accessed: 05-11-2019.
[11] Michael B. Jones and Dick Hardt. The oauth 2.0 authorization framework: Bearer token usage. *RFC*, 6750:1–18, 2012.
[12] Facebook: Securing graph api requests. https://developers.facebook.com/docs/graph-api/securing-requests. Accessed: 05-09-2019.
[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269. ACM, 2014.
[14] Gofundme android application. https://play.google.com/store/apps/details?id=com.GoFundMe.GoFundMe.
[15] Waplog android application. https://play.google.com/store/apps/details?id=com.waplog.social.
[16] Topface android application. https://play.google.com/store/apps/details?id=com.topface.topface.
[17] Authenticate your users. https://developers.google.com/drive/api/v3/about-auth. Accessed: 05-07-2019.
[18] M-trends 2017 by fireeye. https://content.fireeye.com/m-trends/rpt-m-trends-2017. Accessed: 05-07-2019.
[19] Chatous android application. https://play.google.com/store/apps/details?id=com.chatous.chatous.
[20] Wish android application. https://play.google.com/store/apps/details?id=com.contextlogic.wish.
[21] Authenticate with a backend server. https://developers.google.com/identity/sign-in/android/backend-auth. Accessed: 05-07-2019.
[22] Cbs sports android application. https://play.google.com/store/apps/details?id=com.handmark.sportcaster.
[23] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth SSO systems. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 378–390. ACM, 2012.
[24] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 651–662. ACM, 2016.
[25] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin R. B. Butler. More guidelines than rules: CSRF vulnerabilities from noncompliant oauth 2.0 implementations. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *Lecture Notes in Computer Science*, pages 239–260. Springer, 2015.
[26] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. Wpse: Fortifying web protocols via browser-side security monitoring. 2018.
[27] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O single sign-off, where art thou? an empirical analysis of single sign-on account hijacking and session management on the web. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1475–1492. USENIX Association, 2018.
[28] Victor Sucasas, Georgios Mantas, Saud Althunibat, Leonardo Oliveira, Angelos Antonopoulos, Ifiok Otung, and Jonathan Rodriguez. A privacy-enhanced oauth 2.0 based protocol for smart city mobile applications. *Computers & Security*, 74:258–274, 2018.
[29] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Acm Sigsac Conference*, 2017.
[30] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of oauth implementations

in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 61–70. ACM, 2015.

[31] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1204–1215. ACM, 2016.

[32] Suhas Pai, Yash Sharma, Sunil Kumar, and Sanjay Singh. Formal verification of oauth 2.0 using alloy framework. In *International Conference on Communication Systems & Network Technologies*, 2011.

[33] Quanqi Ye, Guangdong Bai, Kailong Wang, and Jin Song Dong. Formal analysis of a single sign-on protocol implementation for android. In *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015*, pages 90–99. IEEE Computer Society, 2015.

[34] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 73–84, 2013.

[35] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V. N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 519–534, 2015.

[36] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.

[37] Yu Feng, Xinyu Wang, Isil Dillig, and Calvin Lin. EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 520–534, 2015.

[38] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 365–383, 2005.

[39] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 59–76, 2005.

[40] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 576–587, 2014.