Application Impersonation: Problems of OAuth and API Design in Online Social Networks

Pili Hu[1], Ronghai Yang[1], Yue Li[2] and Wing Cheong Lau[1]
[1]Department of Information Engineering, The Chinese University of Hong Kong
[2]Department of Computer Science, College of William and Mary
{hupili,yr013}@ie.cuhk.edu.hk yli@cs.wm.edu wclau@ie.cuhk.edu.hk

ABSTRACT

OAuth 2.0 protocol has enjoyed wide adoption by Online Social Network (OSN) providers since its inception. Although the security guideline of OAuth 2.0 is well discussed in RFC6749 and RFC6819, many real-world attacks due to the implementation specifics of OAuth 2.0 in various OSNs have been discovered. To our knowledge, previously discovered loopholes are all based on the misuse of OAuth and many of them rely on provider side or application side vulnerabilities/ faults beyond the scope of the OAuth protocol. It was generally believed that correct use of OAuth 2.0 is secure. In this paper, we show that OAuth 2.0 is intrinsically vulnerable to App impersonation attack due to its provision of multiple authorization flows and token types. We start by reviewing and analyzing the OAuth 2.0 protocol and some common API design problems found in many 1st-tiered OSNs. We then propose the App impersonation attack and investigate its impact on 12 major OSN providers. We demonstrate that, App impersonation via OAuth 2.0, when combined with additional API design features/ deficiencies, make large-scale exploit and privacy-leak possible. For example, it becomes possible for an attacker to completely crawl a 200-million-user OSN within just one week and harvest data objects like the status list and friend list which are expected, by its users, to be private among only friends. We also propose fixes that can be readily deployed to tackle the OAuth2.0-based App impersonation problem.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection —*Unauthorized access, Authentication*

General Terms

Security, Measurement

* Pili Hu and Ronghai Yang have contributed equally to this work.
+ On May 22, 2014, we have notified all providers, to our knowledge, that are affected by the OAuth App Impersonation Attack.At the same time, we also sent the intial verison of this paper to all of these providers.

COSN'14, October 1–2, 2014, Dublin, Ireland.

http://dx.doi.org/10.1145/2660460.2660463.

Keywords

OAuth 2.0; App Impersonation Attack; API Design in OSN; Social Network Privacy; Single Sign On

1. INTRODUCTION

Many Online Social Networks (OSN) are using OAuth 2.0 to grant access to API endpoints nowadays. Despite of many thorough threat model analyses and security guidelines (e.g. RFC6819), most OSNs are still vulnerable to a variety of attacks. To our knowledge, previously discovered loopholes are all based on the misuse of OAuth 2.0 and some depend on provider-side or application-side faults beyond the scope of OAuth protocol, such as XSS and open redirector. It was generally believed that correct use of OAuth 2.0 (by OSN provider and application developer) is secure enough. However, we clarify this common misunderstanding by demonstrating various leakage of user data which roots from the blind-spot of the OAuth's fundamental design rationale: OAuth focuses on protecting the user, not the application.

We show that, even if OSN providers and application developers follow the current best practice in using OAuth 2.0, application impersonation is still inevitable on many OSN platforms: According to the OAuth 2.0 standard, they support implicit-grant flow and bearer-token usage. Although it has become common knowledge for application developers to use authorization-code-grant flow and use access token in a MAC-token style wherever possible, there is no mechanism for them to opt out from the support of implicit-grant flow and bearer-token usage in an OSN platform. Note that different applications may have different privileges like accessing permissions and rate limits. Towards this end, application impersonation in general enables privilege escalation and the extent of the actual damage would depend on platform-specific details. To summarize, this paper has made the following technical contributions:

- We found and formalized the forged-implicit-flow attack and forged-bearer-token attack, which lead to App impersonation attack, in OAuth 2.0 framework[1].
- We examined 12 major OSN providers and conducted proof-of-concept experiments to show the possible consequences of App impersonation attack. To our knowledge, these are the first demonstrations of massive attacks based on the design problem of OAuth, instead of the mis-use of OAuth.
- We propose immediate and deployable fixes to the App impersonation problem.

[1]For the rest of the paper, we use OAuth to denote OAuth 2.0 if not specified otherwise.

This paper is organized as follows. In Section 2, we survey related work in OAuth security and privacy issues on OSN. In Section 3, we discuss the authorization flows supported by OAuth 2.0, and propose the forged-implicit-flow attack. In Section 4, we discuss and analyze the two commonly used token types, and propose the forged-bearer-token attack. In Section 5, we discuss major issues related to API design. In Section 6, we summarize our findings on 12 major OSN providers and illustrate the feasibility of several large-scale exploits/ privacy-leaks. In Section 7, we conclude the paper and summarize the immediate fixes that OSN providers should adopt and deploy.

2. RELATED WORK

OAuth 1.0 is defined in RFC5849 [1] and obsoleted by OAuth 2.0 in RFC6749 [2]. RFC6749 devotes the whole chapter 10 to security considerations of the OAuth 2.0. The informational RFC6819 [3] further discusses OAuth security using a comprehensive threat model analysis. The access token obtained from the OAuth protocol can be used in two ways: bearer token defined in RFC6750 [4] and MAC token proposed in the draft [5].

Although OAuth itself has a sound security model, and the authorization code grant flow was cryptographically proved secure [6] under the assumption that TLS is used, many real-world attacks were found. This is mainly caused by the fact that many OSN providers implemented OAuth before the eventual standardization, and App developers are also likely to misuse the SDKs [7] due to undocumented assumptions in the SDKs. Motivated by formal protocol checking researches, [8] used automatic tools to discover a previously known security flaw from the specification of OAuth. [9] provided a finer-granulared modeling method of system components and was able to discover more concrete loopholes. Since the devil of OAuth (or more generally authentication/authorization protocols), is in the implementation details, researchers have recently used network-trace based approach [10] [11] [12] to discover security flaws, like fail of complete parameter checking, session swapping and Cross Site Request Forgery (CSRF), just to name a few. Many of the problems are ad-hoc in nature and some even depend on the existence of non-OAuth related fault components e.g. an open redirector on the Application site. Regardless of the details, previous demonstrable attacks are based on the misuse of OAuth and theoretically covered by RFC6819. On the contrary, we show that OAuth is intrinsically vulnerable to App impersonation attack. This was overlooked for a long time because the initial intention of OAuth is to protect users rather than applications.

In the field of user privacy study, there are qualitative or small-scale quantitative studies regarding privacy policy and settings on OSNs. It is shown that most users leave privacy settings as default [13] and those who ever tried to change the settings usually fail to achieve their goal [14] [15]. The consequence is that their social behaviour data is exposed to more audienaces than expected [16] [17]. In recent years, more concerns are raised regarding the potential leakage caused by 3rd-party Apps. [18] examined 150 popular Facebook Apps and found that 90% of them request user private data which were not actually needed. The App impersonation attack makes the problem worse, because un-used permissions might be exploited by attackers.

One of the most valuable assets generated by an OSN is the social-relationship graph. Based on the topology, one can conduct various privacy-breaking graph mining campaign like de-anonymizing a graph [19] and infer user ages [20]. Towards this end, protecting the social graph from massive and systematical leakage is of great importance for providers. We observed that all major providers gradually reinforced the crawling barriers over the years. For example, the million-level large-scale crawling methods used in [21] (1.7M Facebook) [20] (3M Facebook) [22] (40M Renren) [23] (70M Renren) were invalidated after some service upgrades. It is now usually costly (in terms of time and resources) to crawl a substantially large graph. Depending on the specific OSN, it may require a large amount of Sybil accounts, baiting applications, IP pools, etc. With the App impersonation flaw discovered in this paper, it is possible to conduct large-scale crawling in a rapid and cheap manner again.

3. OAUTH 2.0 AUTHORIZATION FLOWS

There are three parties in the OAuth eco-system: *Provider*, *User* and *App*. Users socialize on the OSN platform and create various data objects like statuses, photos and friendship graph. In order to read or write a User's object, the third-party App should get the corresponding authorization from User.

To solve this problem, RFC6749 [2] defines four types of authorization flows. Regardless of the authorization flow, the ultimate goal of OAuth is for App to get an *access token*, which is a proof that App can access to the associated resources (on behalf of User). In this section, we first introduce the two most common types of authorization flow, i.e. authorization code grant and implicit grant, and then we propose the forged-implicit-flow attack.

3.1 Authorization Code Grant Flow

The authorization code flow ("server flow" in some literature) plus the invocation of resource API is illustrated in Fig. 3.1. The steps are as follows:

1. User visits App;
2. App redirects User to Provider for authentication;
3. User reviews the permissions requested by App and present User credential to Provider for confirmation;
4. Once authenticated, Provider returns to User an authorization code;
5. User is redirected to App with the authorization code;
6. App exchanges authorization code for the access token by sending AppSecret to Provider.
7. After checking the validity of the authorization code and the App's identity (via the shared AppSecret), Provider responds to App with the access token.

After obtaining access token, the App can query the HTTP endpoints of resource APIs with this access token. There are two important properties of the authorization code grant flow.

- Access token is only shared between Provider and the App;
- Code alone is of no use to User because Provider requires proof of AppSecret before issuing the actual access token.

Note that the "+" sign in the figures just illustrate that AppSecret and/or access token is used in a request. The concrete process may involve signature using those elements.

3.2 Implicit Grant Flow

The high-level picture of the implicit grant flow ("client flow" in some literature) is shown in Fig. 3.2. The authorization steps are as follows:

1. User Visits App;

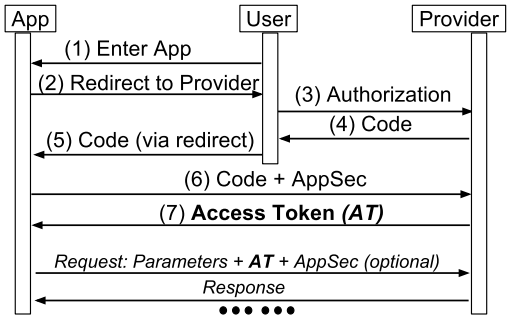

Figure 1: Authorization Code Grant Flow

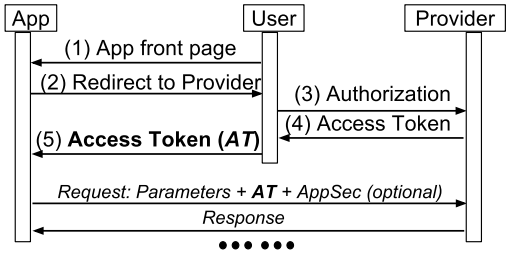

Figure 2: Implicit Grant Flow

2. App redirects User to Provider for authentication and declares to use implicit grant flow;
3. User reviews the permissions requested by App and show the credential to IdP for confirmation.
4. Once authenticated, Provider returns an access token;
5. Access token is redirected to App dirctly.

Unlike the authorization code flow, the access token issued by Provider is relayed through User to App directly. This authorization flow is intended to lower the barrier of App development. It is useful in cases where App can not protect the AppSecret or crypto primitives are too heavy for the execution environment of App. But it makes the authrization process less secure at the same time since IdP simply treats the access token holder as the App without further identity authentication (via AppSecret). Therefore, RFC6749 [2] suggested that implicit grant flow should be avoided whenever authorization code flow is available.

3.3 Forged-implicit-flow Attack

Comparing Fig. 3.1 and Fig. 3.2, it is easy to see that User can bypass App and use this access token to query Provider's resource APIs. This is the first key that enables App impersonation. Note that the problem here is the *availability of platform support* for implicit grant flow, regardless of whether an App would use it or not. Even if App follows the best practice of OAuth and employ the authorization code flow, User can enforce to use implicit flow instead as no mechanisms provided by Provider to opt out this flow. In order to accomplish Steps(1)-(4) of the implicit grant flow, A user only needs to know two public parameters from the App: `client_id` (AppID) and `redirect_uri` (callback URL). Even if App uses authorization code grant flow, User can still harvest these two parameters and forge an implicit grant flow.

For backward compatibility with a large number of existing 3rd-party Apps, OSN providers cannot totally remove the support of implicit grant flow. A practical and immediate fix for forged-implicit-flow attack is to let an App opt-out implicit grant flow if it is capable of performing authorization code flow.

4. OAUTH 2.0 TOKEN TYPES

Upon completion of OAuth protocol, an App obtains a valid access token. Regardless of the actual implementations, most providers use access token in (slight variation of) bearer token style or MAC token styles.

4.1 Bearer Token

Any party in possession of a bearer token [4] can use it to get access associated resource without identity assertion (AppSecret). An App simply puts the access token in the HTTP request as a header field or as part of the query-string. As long as the access token and other additional parameters are valid, Provider returns the requested resource.

4.2 MAC Token

Unlike bearer token, MAC token [5] requires the proof of the identity when making request. There are generally three steps to construct a request:

1. List all request parameters and the access token;
2. Concatenate those data as a single string in a canonical form;
3. Compute an HMAC [5] of the concatenated string using AppSecret as the key.

The Resource API endpoint verifies the authenticity and integrity of this request based on the shared secret AppSecret before returning results.

4.3 Forged-bearer-token Attack

One can see that MAC token is preferable from a security perspective. Since AppSecret is only shared between Provider and App, the HMAC output protects the authenticity and integrity of this request. On the contrary, bearer token is vulnerable to theft because anyone in possession of the access token can do whatever the original App can, as is defined by RFC6750 [4].

Similar to the forged-implicit-flow attack we point out in the previous sections, forged-bearer-token attack is feasible regardless of whether the App uses it or not. When these two attacks are combined, an attacker (a malicious User) can easily impersonate any App.

5. API DESIGN OF OSN

Using the aforementioned two attacks, one can launch App impersonation attack on any platform that supports, without opt-out, implicit-grant flow and bearer token usage. The consequence of App impersonation depends on the platform specifics, in particular how the API is designed. In this section, we discuss three general issues when designing the API.

5.1 Scope Design

The rights to access different resources is controlled by the "scope" parameter of OAuth. Although OAuth is originated and populated by large-scale OSNs, its use is not limited to the OSN scenario. Towards this end, how to design "scope" is not specified as part of the standard and every system should specify according to its own service nature. We have studied 12 major OSN providers (Table 1) and find the best practice is to make a three dimensional scope design, as illustrated in Fig. 3 and Fig. 4:

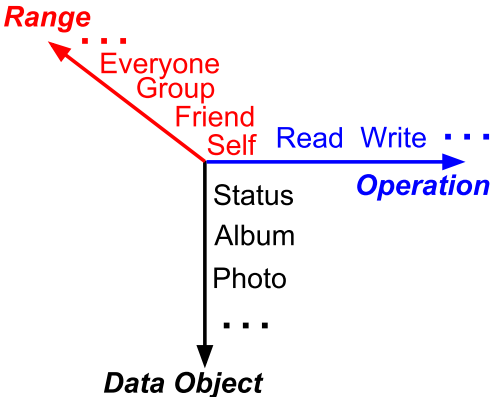

Figure 3: User Permissions

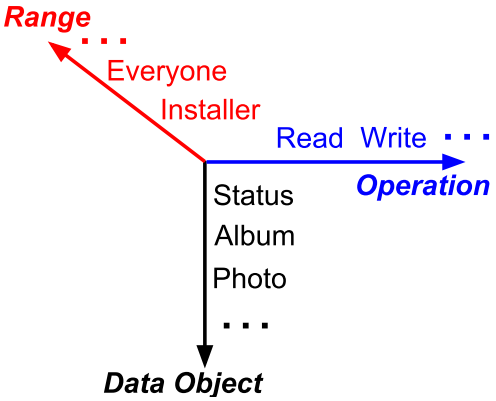

Figure 4: App Permissions

- Data Object – Examples are like status and photo.
- Operation – It can be coarsely classified as "read" or "write". Or, "write" can be further classified into "create", "modify" and "delete".
- Range – In the case of a User originated operation, e.g. "read status", the range can be "Self", "Friend", "Group" and "Everyone". In the case of an App originated operation, e.g. "send notification", the range can be "Installers" and "Everyone".

Note that not all combinations of the three dimensions are valid. For example, "write status of everyone" is a peculiar permission in the context of OSN. We have two remarks regarding scope:

- Permission systems for App originated and User originated operations should be separated. Or, malicious User can operate on behalf of the App as we will show in a proof-of-concept (POC) experiment later.
- Regardless of the dimensions of scope, there should be a way for Provider or App to constrain scope. This can make sure no unwanted permissions are granted and thus protect App from the abuse of access token when App credential is leaked.

5.2 Rate Limit

To avoid Denial of Service (DoS) attack, OSN providers should limit the API access rate. According to our study, the best practice should be to limit rate from four aspects:

Table 1: Statistics of Examined Providers

ID	Name	Registered Users	Alexa Rank*
P1	Provider 1	>300,000,000	≤ 10
P2	Provider 2	>200,000,000	≤ 1000
P3	Provider 3	>300,000,000	≤ 20
P4	Provider 4	>100,000,000	≤ 50
P5	Provider 5	>200,000,000	≤ 10
P6	Provider 6	>300,000,000	≤ 10
P8	Provider 8	Not Found	≤ 20
P7	Provider 7	>5,000,000	≤ 200
P9	Provider 9	>200,000,000	≤ 10
P10	Provider 10	>200,000,000	≤ 10
P11	Provider 11	>200,000,000	≤ 20
P12	Provider 12	>300,000,000	≤ 10

- *: Rank is for the root domain.
- Provider names are masked to avoid the concrete attacks being directly mapped back.

1. rate per IP (/IP);
2. rate per User per App (/User/App);
3. rate per App (/App);
4. rate per User (/User).

Since an access token is bound to a User and an App, those rate limit mechanisms are easy to implement. As we pointed out in the related work section, API rate limit model is of great interest to large-scale crawling campaigns. If not fully controlled from the four aspects, there is room for attackers to amplify crawling rate one way or the other.

5.3 App Differentiation

We find that some of the providers differentiate Apps and give them different access rate limit or access rights. For example, a test stage App may only get a rate limit of 10 queries/hour. After approved by the provider, the App may get 100 queries/hour. For another example, Provider can classify Apps into several categories and each category enjoys different permissions. App differentiation is the final trigger for most concrete attacks we found so far. If Apps are differentiated, App impersonation immediately means privilege escalation.

6. STUDY OF MAJOR PROVIDERS

We have systematically studied 12 major providers, whose statistics are shown in Table 1. In this section, we first summarize our findings of the features/problems of their OAuth/API implementations in Table 2 and then we perform case studies of concrete exploits.

6.1 Summary of Major OSN Providers

The features of 12 providers are summarized in Table 2. In order to review whether a provider is subject to the App impersonation attack, we focus what authorization flow it supports, what token type it supports, and whether it provides opt-outs for implicit flow and bearer token usage. The results are shown in the first 6 rows of Table 2. A simple way to check whether App impersonation attack is available is to examine row {1, 2, 4, 5} and see if the result is {Y,N,Y,N}. We find that 8 out of the 12 providers satisfy this condition.

Table 2: Summary Providers Properties

ID	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Implicit Grant Flow	Y	Y	Y	Y	N	Y	N	Y	Y	Y	Y	N.A.
Implicit Grant Flow Opt-Out	Y[1]	N	N	N	N.A.	N	N.A.	N	N	N	N	N.A.
Authorization Code Grant Flow	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N.A.
Bearer Token	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N.A.
Bearer Token Opt-Out	Y[1]	N	N	N	N	N	N	N	N	N	N	N.A.
MAC Token	Y	Y	N	N	N	N	N	N	N	N	N	Y
Scope Dimensions	3D	2D	3D	2D	3D	3D	2D	3D	0D	3D	2D	2D
Scope Constraint	Y	N	Y	Y	N	N	N	N	N	Y	Y	Y
Rate Control (/IP)	Y[2]	N	Y	N	N	N	N	-	N	N	N	N
Rate Control (/User /App)	Y[2]	Y	Y	Y	N	N	N	-	N	N	Y	Y
Rate Control (/App)	Y[2]	N	N	Y	Y	Y	Y	-	Y	Y	N	N
Rate Control (/User)	N[2]	N	Y	N	Y	Y	N	-	N	N	N	Y
App Differentiation	-	Y	Y	-	-	-	-	-	Y	Y	Y	-

- [1]: Provider 1 did not support the two opt-outs proposed in this paper when we started the investigation. The two features were observed by our team in April 2014.
- [2]: No official rate control documentation is found for Provider 1. The control types are from other developers' tests on the forum.

In order to find more concrete and serious exploits, we investigate the API design issues from three aspects, namely, scope design, rate limit and App differentiation. For scope design, we emphasize two features: the dimensions and the constraints beyond protocol. For the former, we may find a problem once there is a missing dimension. For the latter, we mean whether Providers offer another way to limit the "scope" regardless of the "scope=xxx" parameter an (impersonated) App uses in the protocol sequence. This can be achieved by a request/approval process between App and Provider, or with a control panel for developers to self-limit the permissions. It helps to protect a legitimate App's access token from being abused.

The result of our study is summarized in Table 2. Following are the major observations:

- Out of the 12 major providers, only 3 of them support MAC token. Since there is only one token type in other platforms, Apps can not opt-out bearer token.
- Ideal dimensions of the scope is not a global constant. Instead, it depends on the service nature of one provider. For example, if one OSN is fully public, it does not hurt to remove the Range dimension we stated in Section 5.2.
- We find that no providers has actually implemented all the four types of rate control. Since all the four aspects increase the barrier for large-scale exploits, they should be enforced wherever possible.
- Although we focused systematic and documented difference of Apps in the survey, we did find Apps with undocumented difference. The surprisingly large power possessed by those Apps leads to serious exploits and some cases are shown in the next section.

6.2 Case Studies of Concrete Exploits

App impersonation has a great potential for various attacks especially when combined with additional design and/or implementation flaws of each specific platform. The consequence can be very serious, depending on the api design of a provider. In this section, we discuss some actual exemptified exploits enabled by App impersonation and show the corresponding implication.

6.2.1 App Reputation Attack

Many OSN providers will show the source when User performs a certain action, e.g. "message posted via XXX App" and "follower added via XXX App". This feature is useful for App to build up its brand if properly used. However, the attacker can perform some malicious actions to ruin the reputation of the impersonated App. For example, one can mention ("@") a user on some micro-blogging services. The attacker can post spams while mentioning other users. Form the victim user's point of view, the source (impersonated App) may look spammy. We have verified that this type of exploit is realizable on Provider 1, Provider 2, Provider 3, Provider 6, Provider 9, Provider 10 and Provider 11.

6.2.2 Privilege Escalation and Rate Amplification

Provider 3 supports scope constraint, which is a permission request/ approval process with this provider. Under this setup, the attacker can easily obtain advanced privileges without passing the request/ approval process, which is usually time consuming and may not always succeed. Furthermore, the permission to read the friend list of all users is only granted to a strictly selected set of "privileged" Apps, e.g. official client apps authored by the OSN provider itself or by some of its special business partners. An ordinary App can only get the authorizing User's friend list. It is not very useful because the authorizing User is a Sybil node in our App impersonation attack. Via a proof-of-concept App, we have demonstrated that it is possible to get the "friend list" of an arbitrary user by impersonating a "high-end" App. Worse still, the privileged App (which can get the friend list of any arbitrary user) possesses 40 times more API-based query rate quota than a standard App created by an ordinary user. When combined with App impersonation, the different treatment in access privilege and API query quota among different types of Apps becomes a flaw (or a feature) of severe consequence as it makes massive leakage of user data possible. In fact, we have found that at least Provider 2, Provider 3 and Provider 9 are vulnerable to such rate-amplification attacks as they can be tricked to give 10 to 1000 times more API-based query quota for a privileged impersonated App. Meanwhile, Provider 6, Provider 4,Provider 10 and Provider 11 are also vulnerable to access privilege escalation attack.

6.2.3 *Unauthorized Notification Delivery to Millions of App Users*

As we discussed in Section 5.1, the permission systems for User-originated operation and App-originated operation should be separated. The access token returned by the authorization grant code flow or implicit grant flow is bound to the User and the App. This access token should only be used when an App requests User-originated operation on behalf of the User. However, Provider 2 also uses this access token for a should-be App-originated operation, namely to send app notification to all the users who have installed the App. As a result, an attacker can impersonate an App and send notifications to *all* of its users with only a single Sybil user account. Since the notification message can contain a URL, this loophole can be used for massive spamming or delivery of malicious URLs or contents. To maximize the power of this attack, an attacker can first identify the popular Apps within the platform. The only remaining question is how to harvest the list of users who have installed one of these popular Apps. With some investigation of several popular Apps, we find that many App users also follow the public page of that App. To verify the feasibility of such a massive exploit, we have successfully conducted a proof-of-concept experiment as follows:

1. Send unauthorized notifications carrying arbitrary URLs to a list of our own accounts
2. Collect the IDs of 4 million users who have installed a popular App by examining the public fan list of the App.

6.2.4 *Massive Leakage of 200 Million+ User Data in a Matter of Weeks*

Provider 2 is supposed to be a closed OSN, i.e. user data objects like statuses and photos are shared among friend unless their access mode is set to public. We confirmed this perception by interviewing 20 active users. However, as noted in Table 2, the scope of access control in Provider 2 is only two-dimensional, i.e. it misses the Range dimension. In other words, as long as one valid access token with "read status" permission is granted, one can use it to perform actions like "read my status", "read my friends' statuses" and even "read one stranger's status". The same situation applies to other user data objects, e.g. albums and shared objects, on the same provider. We had reported this flawed API design to Provider 2 back in June 2013 but it claimed that the wide-open access control (collapsed dimensions in our terminology) is a feature rather than a bug. Instead, this provider seems to solely rely on rate throttling to prevent massive leakage, because it would take multiple years for a normal App to retrieve all the private data objects. However, we have verified that Provider 2 differentiates Apps and give them different API access rates. During the investigation of one privileged App, we discovered that the App possessed shockingly large API access quota of at least 1 million queries/hour. It is estimated to be close to 40% of the overall API server capacity of Provider 2[2] Given the size of the user base of Provider 2 (200 million), an attacker exhausting the full capacity can enumerate the whole network within one week. Considering CPU, memory (for ID deduplication in BFS), bandwidth and storage, the overall resource consumption can be well supported by a `m3.2xlarge` Amazon EC2 instance, which only costs US$150 per week.

6.2.5 *Massive Connection Establishment for Sybils*

Advertisers/Spammers on OSNs usually register many Sybil accounts on OSN. The more followers one account has, the more valuable it is. It has been shown in previous studies that reciprocal following is a common phenomenon on directed OSNs. That is, if A follows B, B will follow back as an acknowledgement. This is more likely to happen if B is a low-degree node. It is common knowledge for advertisers/spammers to automatically follow/unfollow other users in order to increase its number of followers. OSN providers already take action to limit the following rate. Prior to our discovery, the per-User per-App access rate limit is effective enough. One way to boost the "following" rate is to register more Apps, but it is more costly to register Sybil developer accounts than Sybil user accounts on many providers. Another way to increase aggregated rate is to register more Sybil user accounts and each account gets a small rate everyday. Neither is this alternative way preferable because of the preferential attachment effect on OSNs, namely, it is better to have one account with 10K followers than ten accounts with 1K followers. With App impersonation, it becomes very affordable (resource-wise) for an attacker to aggregate higher rate on a single user. As long as the provider does not have per-User rate limit, the attacker can easily launch a massive connection establishment campaign by using access token from different Apps and sending friend request to normal users. We have demonstated the feasibility of such an exploit via a POC experiment with our own unprivileged Apps on Provider 9 and Provider 11.

[2]The estimation is derived by observing the random drop rate of probing queries when the API-server is heavily loaded.

7. CONCLUSION

In this paper, we have identified and demonstrated the so-called App impersonation attack by leveraging OAuth 2.0's provisions of multiple authorization flows and token usage flavours. When implemented without opt-outs, attacker can easily launch forged-implicit-flow attacks and forged-bearer-token attacks. We have systematically examined 12 major OSN providers and found that 8 of them exhibited this vulnerability. The consequence is rather serious on some of the providers under study due to various platform-specific deficiencies in API design. Our findings show that it is high time for industrial practitioners to:

1. support the two opt-out policies we proposed for OAuth;
2. review the scope design in their access control architecture;
3. provide scope constraint mechanism beyond protocol;
4. review their rate control strategy;
5. review excessive power/ privileges they have been granting to some special/ partner Apps.

In the long run, this work calls for the re-examination of the need of providing application protection in the design of the next version of OAuth.

Responsible Disclosure

On May 22,2014, we have notified all providers, to our knowledge,that are affected by the OAuth App Impersonation Attack. At the same time, we also sent the intial version of this paper to all of these providers.

Acknowledgments

This work is supported in part by a CUHK-Hong Kong RGC Direct Grant - project number 4055031.

8. REFERENCES

[1] E. Hammer-Lahav, "The oauth 1.0 protocol," April 2010. RFC5849.

[2] D. Hardt, "The oauth 2.0 authorization framework," October 2012. RFC6749.

[3] T. Lodderstedt, M. McGloin, and P. Hunt, "Oauth 2.0 threat model and security considerations," January 2013. RFC6819.

[4] M. Jones and D. Hardt, "The oauth 2.0 authorization framework: Bearer token usage," October 2012. RFC6750.

[5] E. Hammer-Lahav, "HTTP authentication: MAC access authentication," Feb 2012.

[6] S. Chari, C. S. Jutla, and A. Roy, "Universally composable security analysis of oauth v2.0," *IACR Cryptology ePrint Archive*, vol. 2011, p. 526, 2011.

[7] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization," tech. rep., Microsoft Research Technical Report MSR-TR-2013, 2013.

[8] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, "Formal verification of oauth 2.0 using alloy framework," in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pp. 655–659, IEEE, 2011.

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

[10] 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*, pp. 365–379, IEEE, 2012.

[11] 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*, pp. 378–390, ACM, 2012.

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

[13] R. Gross and A. Acquisti, "Information revelation and privacy in online social networks," in *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, 2005.

[14] M. Madejski, M. Johnson, and S. M. Bellovin, "A study of privacy settings errors in an online social network," in *PERCOM Workshops*, 2012.

[15] M. Madejski, M. L. Johnson, and S. M. Bellovin, "The failure of online social network privacy settings," Tech. Rep. CUCS-010-11, Department of Computer Science, Columbia University, 2011.

[16] Y. Liu, K. P. Gummadi, B. Krishnamurthy, and A. Mislove, "Analyzing Facebook privacy settings: user expectations vs. reality," in *IMC*, 2011.

[17] Y. Wang, G. Norcie, S. Komanduri, A. Acquisti, P. G. Leon, and L. F. Cranor, "I regretted the minute i pressed share: A qualitative study of regrets on facebook," in *Proceedings of the Seventh Symposium on Usable Privacy and Security*, p. 10, ACM, 2011.

[18] A. Felt and D. Evans, "Privacy protection for social networking apis," *W2SP*, 2008.

[19] A. Narayanan and V. Shmatikov, "De-anonymizing social networks," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 173–187, IEEE, 2009.

[20] R. Dey, C. Tang, K. Ross, and N. Saxena, "Estimating age privacy leakage in online social networks," in *IEEE INFOCOM*, p. 3118, 2012.

[21] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *EuroSys*, 2009.

[22] J. Jiang, C. Wilson, X. Wang, P. Huang, W. Sha, Y. Dai, and B. Zhao, "Understanding latent interactions in online social networks," in *IMC*, 2010.

[23] Anonymous, "Crawling Renren by ID space enumeration." unpublished (Private Communication), 2010.