How to Leak a 100-Million-Node Social Graph in Just One Week? - A Reflection on OAuth and API Design in Online Social Networks

Pili Hu and Wing Cheong Lau

July 1, 2014

Abstract

In this paper[1], we discuss a design problem of OAuth 2.0 which exposes Online Social Network (OSN) applications to impersonation attacks. When combined with additional OSN provider-specific problems, e.g. design and implementation details of API supported by an OSN, this App impersonation can result in severe privacy/ security vulnerabilites: in one case, massive, rapid leakage of user data involving more than 10^8 accounts becomes possible: the time to complete such an attack would take less than one week with an estimated computational cost as low as US$ 150 based on Amazon Web Service pricing at the experiment time. Although the entire massive data collection campaign would require some engineering efforts, the App impersonation part can be easily accomplished with a few keystrokes and mouse clicks using a standard browser. By reporting our findings, we hope to alert the stake holders in the industry worldwide so that they can launch a thorough review of their current support of OAuth 2.0 implicit flow and bearer token and to provide appropriate opt-out policies as soon as possible. Besides, application protection should also be considered when designing the next version of OAuth.

1 Introduction

Many Online Social Networks (OSN) are using OAuth 2.0 to grant access to API endpoints nowadays. Despite many thorough threat model analyses (e.g. RFC6819), only a few real world attacks have been discovered and demonstrated. To our knowledge, previously discovered loopholes are all based on the misuse of OAuth. It was generally believed that the correct use of OAuth 2.0, e.g. by following RFC6819 [1], is secure enough. We break this belief by demonstrating a massive leakage of user data which roots from the scotoma of OAuth's fundamental design rationale: focus on protecting user, not protecting application.

[1]This work is supported in part by a Hong Kong RGC Direct Grant - project number 4055031

We show that, even if OSN providers and application developers follow best practice, application impersonation is inevitable on many platforms: According to the OAuth 2.0 standard, they support implicit-authorization-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 OSN platforms' support of implicit-authorization-grant flow and bearer-token usage. Since different applications may have different privileges like accessing permissions and rate limits, application impersonation in general enables privilege escalation and the consequence depends on platform-specific details.

As a proof-of-concept experiment, application impersonation has been demonstrated on a large-scale Facebook-like (not Facebook) OSN. Based on this technique, one can use a casual crawler to collect its 100-million-user social graph within just one week and the projected cost based on Amazon Web Service is just $150 USD. Due to its implementation specifics, similar techniques can be applied on this OSN to obtain other private data like all users' status lists and albums. Note that, without privilege escalation, this amount of data (order of 10^8) cannot be obtained in such short time with such little cost even on open graphs like Twitter.

Our discovery shows that it is urgent for industrial practitioners to provide the two aforementioned opt-outs in OAuth and review their API design. This work also highlights that application protection must be considered in the design of the next version of OAuth, and similarly other Single-Sign-On protocols.

2 Background of OAuth 2.0

In the OAuth eco-system, there are three components: *Provider*, *User* and *App*. Users socialize on the OSN platform and create various data objects like statuses and photos. In order for a 3rd-party App to read or write a User's object, User must grant the corresponding permissions to App. The protocol mainly address two problems:

- How does an App obtain an access token? (Authorization flow)
- How should an access token be used? (Token type)

2.1 Authorization Flows

RFC6749 [2] defines four types of authorization flows. Two most commonly used authorization flows are authorization code grant and implicit grant.

The authorization code flow ("server flow" in some literature) plus the invocation of resource API is illustrated in Fig. 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) Provider returns to User an authorization code;

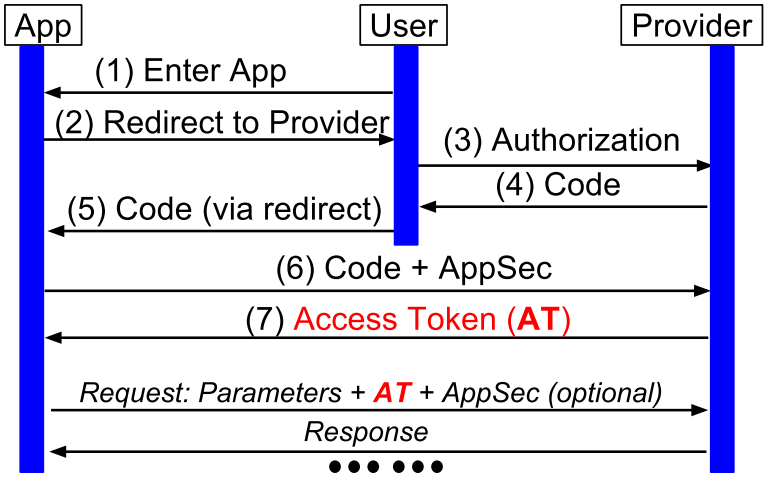

Figure 1: Authorization Code Grant Flow

(5) User is redirected to App with the authorization code; (6) App requests the access token by sending the authorization code and its 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 end-points of resource APIs with this access token. There are two important properties of the authorization code flow: a) Access token is only shared between Provider and the App; b) 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.

The high-level picture of the implicit grant flow ("client flow" in some literature) is shown in Fig. 2. 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. As suggested in RFC6749 [2], implicit grant flow should be avoided whenever authorization code flow is available.

2.2 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 style.

Any party in possession of a bearer token [3] can use it to get access to the 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

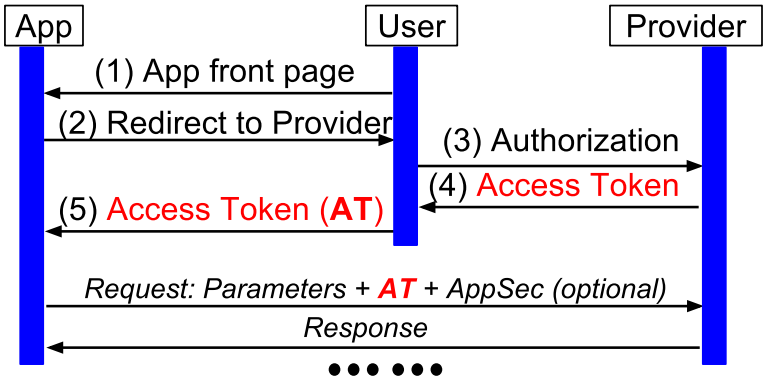

Figure 2: Implicit Grant Flow

other additional parameters are valid, Provider returns the requested resource.

Unlike bearer token, MAC token [4] 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 [4] 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.

3 App Impersonation

Comparing Fig. 1 and Fig. 2, it is easy to see that User can bypass App and use this access token to query Provider's resource APIs. 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. 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.

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 because anyone in possession of the access token can do whatever the original App can, as is defined by RFC6750 [3]. Similar to the forged-implicit-flow attack, forged-bearer-token attack is feasible regardless of whether the developer uses it or not.

When the two attacks are combined, an attacker (a malicious User) can easily impersonate any App. A simple procedure to conduct a proof of concept is as follows:

- Register a valid user account.
- Visit an App and trigger authorization flow.
- The authorize URL has the following pattern:
  (suppose authorization-code-grant flow is used)
  `/authorize?response_type=code&client_id=XXXX&state=XXXX&redirect_uri=XXXX`
- Change browser's address bar to:
  `/authorize?response_type=token&client_id=XXXX&state=XXXX&redirect_uri=XXXX`
- After enter the modified URL, complete the authorization flow by inputing user credentials on the service provider. One can then get the access token directly (in URI fragment after hash tag).
- Use this token in bearer token style and request resource API endpoint. e.g.
  `/api?access_token=XXX&other_parameters`

4 A Case of Massive User Data Leakage Vulnerability

One can see that the App impersonation a generic problem and could potentially affect many OSN providers which have adopted OAuth2.0. The actual extent of damage, however, depends on the API implementation specifics of an individual provider. In this section, we give a case study on Provider X, where App impersonation, when combined with some API design/ App management flaws, results in massive leakage of user data in this OSN possible.

4.1 Wide-Open API Access Rights

Provider X is supposed to be a closed OSN, offering services similar to Facebook (but it is not Facebook). Under this OSN, user data objects like statuses and photos are perceived to be shared among friends only unless their access mode is explicitly set to public. This is obvious if one visits a stranger's webpage under this OSN from a browser. If a user sets the access-mode of his/her homepage under this OSN to be "friend-only", a random visitor to the homepage will be blocked for further access and only very limited amount of profile information will be shown. In other words, detailed privacy-sensitive data objects like user status list and friend list will be "gatekeeped" at the user's homepage. As such, majority of Provider X's users think that their personal data are kept from public eyes as long as they set the access control mode of their homepage. We confirmed this common perception by interviewing 20+ active users of this OSN. The official document of Provider X also states that "One can prevent exposing private data to strangers by setting homepage to be "friend-only" ". Because of this common perception, most

users rely on homepage access control settings and do not separately set individual data objects to be private[2].

Although the gatekeeping effect holds for normal web-based access via http, we found that the API-based access supported by Provider X was wide-open. For example, as long as one valid access token with "read status" permission is granted to an App, it can be used to read *all* user's status on Provider X by filling in the proper parameters of some API call. The same situation applies to other user data objects such as friend-list, albums and shared objects, on the same provider. We had reported our findings on this problematic API design to Provider X but its official response was that such loose access control for their API is a feature, not a bug.

4.2 Undocumented Large API Access Rate

Given the problematic API design and the widespread incorrect perception among Provider X's OSN users, leakage of private user data is inevitable. The only remaining question is on what it takes for a massive privacy leak on this OSN to be materialized. Given the nominal API access quota as documented by Provider X (which is in the order of several hundreds of API calls per hour for an APP), it should take multiple years for a normal App to retrieve the private data objects associated with all user accounts in this OSN. However, during our investigation of different Apps in this OSN, we found that some Apps actually possessed much larger API access quota than the documented one. To further investigate the feasibility of a massive leakage, we have developed a customized crawler for this OSN using the API supported by Provider X. By varying number of crawling worker processes, w, we can observe the achievable crawling rate r. When the aggregated crawling rate becomes large, we observe setup failures for a considerable percentage of TCP connection attempts. The failure rate does not change when the crawlers are launched from different regions and cloud-service providers. We therefore conjecture that performance bottleneck actually lies with Provider X's API server. Assuming API queries are dropped in a uniform random manner upon heavy system loading, we can estimate the processing capacity of Provider X's API server by the following model:

$$r = c\frac{w}{w+b}$$

where r is the aggregated crawling rate, c is the capacity of Provider X, w is number of workers, and b is the background workload in unit of our worker. We observed two points: $w_1 = 50$, $r_1 = 600,000$ and $w_2 = 100$, $r_2 = 960,000$. Solving the simultaneous equations, we can get $c = 2,400,000$ (queries/hour) and $b = 150$. Even for a large network composed of 200 million nodes, an attacker exhausting the full capacity of the API server can enumerate and crawl the entire population in less than 3.5 days. Even if an attacker uses only half of the API server capacity, a complete crawl can be finished in less than one week. Considering the CPU, RAM,

[2]At the time of experiment, there was no itemized option to separately set certain data objects like status list and friend list to be friend-only.

network bandwidth and storage requirement of a full-blown crawler designed for such a task, its overall resource consumption can be well supported by a `m3.2xlarge` Amazon EC2 instance, which only costs US 150$ per week (at the experiment time).

5 Discussion of Other Potential Attacks

We have checked our findings against a dozen major OSN providers worldwide according to the Alexa Top-200 list (many of those with > 100 million registered users). We have found that most of these OSNs are subject to the App impersonation attack discussed in this paper. The actual extent of privacy vulnerability caused by App impersonation is platform-specific. The cases with more severe consequences are often caused by some additional deficiencies in the API design and App management life-cycle of the specific OSN provider, e.g. lack of scope differentiation in the access of information by its Apps and incomplete access-rate monitoring/ throttling. We already notified and shared our full findings with all the providers we studied. Since not every provider has responded and confirmed the problem is insignificant or the problem is fixed, we decide not to disclose further details in the current version of this white paper. Following are some cases of concrete exploits verified on one or more providers:

- Attack App reputation by posting messages using the identity of impersonated App, e.g. "posted via XXX".
- Acquire access privileges (e.g. friend list) that is not available otherwise.
- Send notifications with embedded URLs to all App installers.
- Acquire large API quota to follow/unfollow other users of the OSN to establish social connections (via reciprocal following effect).

Given the generic nature of App impersonation and its simple execution, we expect others may come up with additional exploits and attacks beyond those mentioned above by leveraging such vulnerability.

6 Conclusion

In this paper, we have identified and demonstrated the so-called App impersonation attack via OAuth 2.0 due to its provision of multiple authorization flows and token usage flavours. When implemented without opt-outs, attacker can easily launch forged-implicit-flow attack and forged-bearer-token attack. We have discussed the severe consequences made possible by leveraging App impersonation. 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 their rate control strategy; 3) review excessive power/ privileges they have been granting to some special/ partner Apps.

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

Acknowledgment

This work is supported in part by a Hong Kong RGC Direct Grant - project number 4055031. The authors would like to thank members of MobiTeC OSN Security/Privacy Project including Ronghai Yang, Yue Li and Guanchen Lee among others.

References

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

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

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

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