A survey on security analysis of OAuth 2.0 framework

Shuai LI
Department of Computer Science
University of Calgary, Alberta, Canada
shuai.li1@ucalgary.ca

1 INTRODUCTION

The OAuth 2.0 protocol[7] is one of the most widely deployed authorization protocols. The authorization is the process for granting approval to an entity to access a resource. The authorization task itself can be described as granting access to a requesting client, for a resource hosted on the resource server (RS). This exchange is mediated by the authorization server (AS).

Popular social networks such as Facebook, Google implement OAuth 2.0[5], allowing users to delegate access to specific functions to the third party (client). For example, Google (AS) uses OAuth to allow the email application (client) to add entries into users calendar on her behalf. It also allows a user to log in to a third-party application using her identity managed by an AS. Authorization and SSO solutions have found widespread adoption in the web over last years, with OAuth 2.0 being one of the most popular frameworks.

From the adversary's perspective, if he can launch a successful exploit of an uncovered weakness in the protocol or implementations, the private data from those millions of users [15] could be harvested. Hence, researchers devote intense effort to analyze the OAuth protocol, including formal methods [4][12][1] as well as empirical study of real world implementations[15][9][16][3]. And the conclusion is that the a secure OAuth environment can be built if security recommendations and best practice [7][11] are followed. However, the real world implementations often simplify or even ignore the security recommendations.

The survey on the security analysis of OAuth 2.0 contains the following components:

(1) Introduction of OAuth 2.0 framework in technical details. The security analysis will only focus on OAuth 2.0, OAuth 2.0 is not backward compatible with its predecessor OAuth 1.0 [6]. In OAuth 2.0. the interactions between the client, the AS, the RS can be performed into four different modes: authorization code grant, implicit grant, resource owner password grant and client credentials grant. This survey will not consider resource owner password grant type because the user directly gives passwords to the client in this grant. Such feature only applies to the client with high security standard.
(2) Presentation of significant attacks found in the protocol itself and in the implementation. These attacks can be grouped by where they happens, in communication between client and AS or communication between client and RS. Due to the time limit, I will avoid obvious attacks result from not following those security recommendations which can be easily implemented.
(3) Discussion of the root causes of attacks. The attacks introduced in the survey are largely caused by implementation decisions that trade security for simplicity. There is only one flaw found in the protocol itself. There are few analysis effort that illustrates how simplicity features from implementations lead to weaknesses which is one objective of this project. For example, one simplicity feature could create the possibility to launch several different exploits. The developer should abandon implementation decisions cause severe security threats.
(4) Providing Simple and practical improvement to the implementation of OAuth 2.0. The countermeasure must be simple because that is the main feature to make OAuth 2.0 gain widespread acceptance. This survey provides comprehensives solutions based on different components of OAuth 2.0. Some simple and practical recommendations will also be very helpful to mitigate attacks on extensions of OAuth 2.0. For example, some fixes would also be applicable to improve the security of OAuth based access control in the constrained environment (i.e.IoT).

The rest of the report is organized as follows: Section 2 introduces the OAuth 2.0 protocol. Section 3 presents the attacks. Section 4 discusses the security implications of these insecure decisions which result in the attacks in section 3, then proposes the countermeasures. I outline the contributions and future work of the project in Section 5.

2 OAUTH 2.0

The OAuth 2.0 authorization framework[7] enables a client to obtain scoped access to the resource protected by the RS with the permission of the resource owner. The protocol works generally as follows, authorization information is passed between the entities(client, RS, AS) using access tokens. These tokens are issued by the AS with the approval of the resource owner. Then the client presents the toke to the RS to access the protected resource. To describe the protocol in details, we need to elaborate the roles that are involved in a single protocol run. The roles are:

(1) client: An application that makes resource request to the resource owner. There are two types of clients defined in OAuth 2.0[7]. Public clients are those clients can't securely store the client credentials (issued by the AS during client registration stage). The user-agent based application (a javascript application running on the browser) and the native application (client installed and executed on the device) belong to the public client. This is because the attacker can obtain the credentials on public client (i.e. reverse engineering of the native application to obtain the credentials). The confidential client is the client capable of maintaining the confidentiality of their credentials. The Web server application belongs to the confidential client. For example The client implemented

on a secure server with restricted access to the client credentials.
(2) Resource Server (RS): Hosts the protected request and has the ability to process the access token.
(3) Resource Owner: Grants the permission to others for accessing the resource it owns.
(4) Authorization Server (AS): Issues the token to the client after successfully authenticating the resource owner and obtaining its authorization.

Next, We will discuss three different types of authorization mode which results in three different message interaction sequences.

2.1 Authorization code grant

The client should register the *redirection_uri* and obtain the client credentials at the AS so that the **AS** can identify if a the malicious client sends a malicious *redirection_uri* to the **AS**. The details of the client registration is out of the scope of our discussion. Note the communication between any entities should be protected by TLS against eavesdropping. Below is the message flow:

(1) The **client** *initiates* the flow by directing the **resource owner's** user-agent to the authorization endpoint. The **client** includes *client_id*, *scope*, *state* which is the hash of the current session cookie , and a *redirection_uri* which the **AS** will send the user-agent back once access is granted (or denied). The **AS** will check whether the *redirection_uri* is registered.

(2) Assuming the **AS** verified the **client**'s request successfully. The **AS** authenticates the **resource owner** and asks whether she wants to grant or deny the client's access request.

(3) If the **resource owner** grants the access, the **AS** redirects the user-agent back to the **client** via the *redirection_uri* provided at step (1) earlier. The *redirection_uri* includes an *authorization_code* and *state* provided by the client earlier.

(4) The **client** requests an *accesstoken* from the **AS'**s token endpoint by including the *authorizationcode* received in the previous step, the client credentials and the *redirection_uri*.

(5) The **AS** authenticates the **client**, validates the authorization code, and ensures that the *redirection_uri* received matches the URI used to redirect the client in step (3). If valid, the **AS** responds back with an *access token* and, optionally, a *refresh token*.

(6) Here we use bearer token (only the token string itself). The **client** send a **GET** message including the bearer token to the **RS**. The message payload includes the token, the "iss" claim , "exp" claim, "scope" claim and other useful claims related to the token. Then **RS** validates the token issue by its AS by checking at the "iss" claim in the payload. Ensure that it has not been expired ("exp" claim) and its scope covers the requested resource ("scope" claim).

(7) Assuming that **RS** verified the token successfully, **RS** will return the protected resource back to the requesting **client**.

Optionally, if the **AS** grant the *refresh token* along with a *access token* to the **client**. The **client** can request a new *access token* by only presenting the *accesstoken*. The **AS** must authenticate the **client** again and validate the refresh token before issuing new *access token*. The usage of the *refresh token* has strict requirement, only when the confidential client adopts the authorization code grant.

2.2 Implicit grant

Unlike the authorization code grant, the **client** receives the *access token* directly as the result of the authorization request. Note the communication between any entities should be protected by TLS against eavesdropping [2]. Below is the message flow:

(1) The same as the step (1) in **Authorization code grant**.

(2) The same as step (2) in **Authorization code grant**.

(3) If the **resource owner** grants the access, the **AS** redirects the **user-agent** back to the **client** via the *redirection_uri* provided at step (1) earlier. The *redirection_uri* includes the *access token* and *state* in the URI fragment.

(4) The **user-agent (browser)** retains the fragment information locally and does not include token in the request to the **client**. The user agents do not send the fragment part of URIs to HTTP servers. Thus, an attacker cannot eavesdrop the access token on this communication path, and the token cannot leak through HTTP referrer headers.

(5) The **client** returns a web page containing a script to the **user-agent**. The script extracts token contained in the fragment using JavaScript command such as **document.location.hash.**

(6) The **user-agent** passes the *access token* to the **client**. Then the **client** could use the token to request the resource on the **RS**, Which is the same as step (6) and (7) in section 2.1

2.3 Client Credentials grant

The **client** can request an *access token* using only its client credentials. In this case, the **client** is requesting the resources that have been previously arranged with the **AS** by the **resource owner**. Unlike the previous two grant types, the consent of the **resource owner** is pre-configured as authorization policies at the **AS**. Note the communication between any entities should be protected by TLS against eavesdropping. Below is the message flow:

(1) The **client** authenticates with the **AS** using the client credentials and requests an *access token* from the token endpoint.

(2) The **AS** authenticates the client, and if valid, issues an *access token*.

(3) Then the **client** could use the token to request the resource on the **RS**, Which is the same as step (6) and (7) in section 2.1

2.4 Comparison of three grants

By far, we have discussed two client types (public client and confidential client) and three modes. For the public client, we can further specify the client application into two types, native client application and user-agent based client application. The confidential client is the Web Server client applications. It is necessary to know the requirement and restrictions of each grant type for different client applications. Table 1 illustrates the summary of such restrictions which is generated from the OAuth 2.0 framework documentation. From the table, we can find that 1) OAuth doesn't recommend to allocate the client credentials to public client, only Web Server client application is allowed, due to the credentials on the public client can be obtained by the public client. 2) Only web server client application and native client application should use Authorization code flow. And the client authentication is required for requesting an access token. Native client application uses the *redirection_uri* to prove himself. AS authenticates by ensuring the *redirection_uri*

received matches the URI used to redirect the client in the previous step. 3) Implicit grant flow is optimized for public clients. Particularly for these clients implemented in a browser using a scripting language such as JavaScript. 4) The implicit grant flow does not include client authentication, and relies on the presence of the resource owner and the registration of the *redirection_uri*. 5)Client Credential flow should only be used by the confidential client because this mode totally relies on the authenticity of the client.

Interestingly, it is noticeable that OAuth 2.0 provides each client type with the most suitable flow which could achieve the highest level of usability and the security. And it is clear that for the security: Authorization code > implicit grant > Client Credentials. The reason that the Client Credential ranks in the last one is clear. But why Authorization code grant is more secure than the implicit grant?

Firstly, the authorization code grant supports the 1)client authentication when the client requests the access token. 2) resource owner presence and 3) registration of URI while implicit grant relies only on the latter two to verify the client. Two reasons why client authentication is not included in the implicit grant. One is that public client can't store the client credential securely. Also, it is much simpler to authenticate clients during the direct request between the client and the AS than in the context of the indirect authorization request. The latter would require digital signatures.

Secondly, the implicit grant omits the use of authorization code due to the nature of those clients which are running on the user-agent. In the authorization code grant, the authorization code is encoded into the *redirection_uri*, it may be exposed to the resource owner via browser cache or log file entries. However, the client later exchanges the code for the access token over a more secure direct connection with the AS. Resource owner can not learn anything about the access token. But in the implicit grant, the access token is encoded into the *redirection_uri* when the resource owner authorizes the request. The risk is that the access token can be extracted by the resource owner and other applications residing on the same device.

Table 1: restrictions for different client type.

Mode	Web	CC	Native	CC	User-agent	CC
Authorization	✓	✓	✓			
Implicit			✓		✓	
Client credential	✓	✓				

Web application, Native application, User-gent based application.
CC: client credentials

3 ATTACK

As discussed in the introduction, we present the significant attacks found in the protocol and its implementation. As a result of the study, I find out that it is not scientific to group the attacks based on its roots cause(s) due to the vulnerability interplays. I decide to categorize the attacks found in [4][15][12][16][7][14][11][3]by where they happens, e.g.,in the communication between client and AS or communication between client and RS. Threats in the communication between AS and RS is not covered by this literature survey, which is also excluded from[11]. Please note that I list the attacks mainly caused by commonly adopted simplicity design decisions in real world implementation because the goal is to provide guidance to the OAuth developers. Detailed description of these attacks on applicable grant along with easily implementable fixes are provided. Every countermeasure description refers to a detailed description in Section 4.

The following attacks happen in the communication between client and AS.

3.1 Impersonation attack

The notion of impersonation attack comes up in [15]. In the same paper, they also explain the Access token eavesdropping attack, which can be seen as impersonation attack on the implicit grant. Yang et al. [16] discuss the replay attack module, which is the same attack. Impersonation attack is mentioned in part 4.4.1.6 of the threat model[11].

3.1.1 Authorized code grant.

Assumption: 1)The attacker can obtain a copy of the authorization code, the user agent and the client communication does not use TLS. 2) The authorization code is not limited to one time use and 3) the client does not check whether the response (step (3) in section 2.1) is sent by the same browser from which the authorization request (step (1) in section 2.1) was issued. 4) An client assumes only the user has the knowledge of the identity credentials(authorization code), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack. The attacker may capture an authorization code redirection request (step (3) in section 2.1) in the communication between the User-agent (i.e. browser) and the client application. Then the attacker submits the authorization code to the client. The client will exchange the authorization code for an access token. If OAuth is used in third party log in scenarios, the attacker can use the authorization code to log into the client as the user.

3.1.2 Implicit grant.

Assumption: Since there is no intermediary authorization code in the implicit grant. Assuming 1)The attacker can obtain a copy of the access token, the user agent and the client communication does not use TLS. 2) The access token has not been expired. 3) the client does not check whether the response (step (6) in section 2.2) is sent by the same browser from which the authorization request (step (1) in section 2.2) was issued. 4) An client assumes only the user has the knowledge of the identity credentials(authorization code), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack. The attacker may capture an access token redirection request (step (6) in section 2.2) in the communication between the User-agent (i.e. browser) and the client application. Then the attacker submits the access token to the client. The client uses the token to access protected resources for the benefit of the attacker. If OAuth is used in third party log in scenarios, the attacker can use the access token to log into the client as the user.

3.1.3 Client credentials. Client credentials do not have user-agent involved, However, with out the protection of the communication between the client and the AS. The similar attack could happen.

3.1.4 Fix. The TLS/SSL protection is required by the OAuth for communications between any two entities. However, it imposes performance overhead. Due to the unwanted complications, only 21% [15] of the client websites use SSL to protect the user agent and the client communication.

3.2 Authorization code theft

In [15], they discuss the attack of access token theft due to the automatic authorization granting. **However, I did not find any implementation to date supporting automatic authorization grant of the access token.** While the automatic authorization grant of the authorization code in the authorization code mode is found in the Google implementation. Paper [16] also confirms my discovery. So I assume that this feature is only used for the authorization code in the authorization code grant.

Assumption: 1) The "automatic authorization granting" works as follows: If the resource owner has previously granted permissions to a client application, and the resource owner has already logged into the AS in the same browser session. The AS will not ask the user to grants permissions to the client if it receives the authorization request from the client again. Instead, it will direct the user-agent back to the client with a **new** generated authorization code (step (3) in section 2.1). The attacker does not need to worry about the one-time use of the code because it is new. 2) The attacker can inject a script into any page of a client website. 4) An client assumes only the user has the knowledge of the identity credentials(authorization code), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack: The attacker can inject a script into the client website to initiate a forged authorization request to the AS and then obtain the authorization code in return. The request is transported by a hidden iframe element created by the script. Once the authorization code is obtained, the script will send the code back to the attacker(i.e. attacker.com). With this stolen authorization code, now the attacker can log into the client application as the victim.

Fix: Although this feature offers convenience for the AS, the negative impact of security is considerable. We shouldn't adopt this feature in implementation.

3.3 Code substitution

Code substitution attack is mentioned in part 4.4.1.13, 4.4.2.6 of the threat model[11]. The impersonation attack capture a identity credential(authorization code, access token) in the redirection callback to a target client, using the identity credentials to impersonate a user session. The authorization code theft exploits the automatic authorization granting feature, forging a pre-authorized request to the AS and obtain a new authorization code. Unlike the above two attack, in the code substitution attack, the identity credential is obtained by a malicious application which shares the same AS as the target application.

3.3.1 Authorized code grant.

Assumption: 1) The malicious application is legitimate to the AS of the target application 2) The attacker can obtain the authorization code requested by the malicious application. 3) The authorization code is not limited to one time use and 4) There is no binding between authorization code and client_id, between redirection_uri and authorization code. 5) An client assumes only the user has the knowledge of the identity credentials(authorization code), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack. The attacker tricks the victim into logging into a malicious app using the same AS as the target application. This results in the AS issuing an authorization code for the victim. The malicious application then sends this code to the attacker, The attacker now manipulates the authorization response to the target application and substitutes the code. This code is then exchanged by the client for an access token, which in turn is accepted by the AS. If OAuth is used in third party log in scenarios, the attacker can use the authorization code to log into the client as the user.

3.3.2 Implicit grant.

Assumption: 1) The malicious application is legitimate to the AS of the target application 2) The attacker can obtain the access token requested by the malicious application. 4) There is no binding between access token and client_id, between redirection_uri and access token 5) An client assumes only the user has the knowledge of the identity credentials(access token), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack. The attacker is similar with the attacks in authorized code grant, just change the authorization code to access token. If OAuth is used in third party log in scenarios, the attacker can use the access token to log into the client as the user.

3.3.3 Client credentials. Same assumption and attack as in section 3.3.2.

3.3.4 Fix. When the client sends the token exchange request, the AS must validate whether the particular authorization code is issued to the particular client (binding client id with code), and/or AS must validate whether the particular authorization code is issued and send back via the particular redirection_uri. Refer to the step 4) in section 2.1.

3.4 Session swapping

Session swapping attack is discussed in [4][16][15][11]. In [16],they call it phishing attack, which is exactly the session swapping in the other papers. This attack results from the lack of the state parameters in interactions between the client and the AS (step (1) and step (3) in section 2.1 and section 2.2). There is no state in the client credentials grant type.

3.4.1 Authorized code grant.

Assumption: 1)The client does not provide a state parameter in an authorization request (Step (1) in section 2.1). 2) The attacker can

intercept the authorization code in his own user-agent. 3) The authorization code is not limited to one time use and 4) An client assumes only the user has the knowledge of the identity credentials(access token), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack: The attacker signs into the client using his identities from the AS. Then he intercepts the authorization code on his user-agent (Step (3) in section 2.1). After aborting the redirect flow of the victim back to the client, he sends the authorization code to the client. The client will exchange the authorization code for an access token to access attacker's resource. The user may upload private items to an attacker's resource. If OAuth is used in third party log in scenarios, as the authorization code is bound to the attacker's account on the AS, the attacker can log the victim into his client account to harvest the victim's personal data.

3.4.2 *Implicit grant.*

Assumption: The assumption is the same as in section 3.3.2. 1)The client does not provide a state parameter in an authorization request 2) The attacker can intercept the access token in his own user-agent. 3) An client assumes only the user has the knowledge of the identity credentials(access token), and only by using the correct identity credentials, the client is able to obtain the information about the user.

Attack: The attacker signs into the client using his identities from the AS. Then he intercepts the access token on his user-agent (Step (6) in section 2.2). After aborting the redirect flow of the victim back to the client, he sends the access token to the client. The client will use access token to access attacker's resource. The user may upload private items to an attacker's resource. If OAuth is used in third party log in scenarios, as the access token is bound to the attacker's account on the AS, the attacker can log the victim into his client account to harvest the victim's personal data.

3.4.3 *Fix.*

The use of the state parameter is not optional. Further, a state value should be used only once. If the state does not refresh in every authorized request, once the attacker obtains or guesses the value of the state, he can still mount the session swapping attack.

3.5 AS mix up

This attack is first found in [4], and is reported to the OAuth and OpenID connect working group who confirms the attack. This attack only applies to the authorization code mode.

Assumption: 1) The attacker can manipulate the request in step (1) section 2.1. 2)The client issues the same $redirection\_uri$ to all Authorization servers. 3) The honest AS has not issued a client secret to the client during registration. 4) The attacker can request for the access token code once he obtained the authorization code.

Attack: The attack starts when the Resource Owner selects the AS she wants to log in. The attacker intercepts the request (step (1) in section 2.1) and modifies the request by replacing Honest AS to Malicious AS. Then the attacker redirects to the Honest AS with $clientid$, $state$, $redirection\_uri$. In this attack, we assume that from this point on, the communication between the browser and the client and the AS is protected by using TLS against eavesdropping. The user then authenticates to the Honest AS and if successful, the AS directs the use-agent back to the client with the authorization code. Now, due to the attacker altered the request at the first place, the client thinks that the code was issued by the malicious AS, rather than the honest AS. Then the client now tries to redeem this code for an access token at the malicious AS. This leaks the code to the attacker. If the honest AS has not issued the client a secret, the attacker can redeem code for an access token at the honest AS. The access token allows the attacker to access to the protected resource. In paper[4], they mention that the AS mix up applies to the implicit grant as well. **However, I argue this is not likely to happen**. Because in the implicit grant flow. Even if the attacker can perform the similar attack mentioned above. The Honest AS will finally send the access token to the client, There is no need for the client to contact with the malicious AS anymore. So the access token can not be learned by the attacker.

Fix: Based on the assumption, we could have two easily fixes. 1) AS has to issue the client secret and authenticate the client when the clients request for an access token. 2) A fundamental problem in the OAuth standard is a lack of reliable information in the step (3) in section 2.1. The client does not receive information from where the redirect was initiated. The client can not check whether the information contained in the $redirection\_uri$ comes from the AS that was requested at the first place. The fix is presented in [4], which is to include the identity of the AS in the redirection_uri as a new parameter. Each AS should add such a parameter to the redirect URI. The Client then can check the new parameter to mitigate the AS mix up attack.

3.6 Client Obtains too much access scope

This attack is mentioned in part 4.2.2 and 4.4.3.2 of the threat model[11].

3.6.1 *Authorization code grant.*

Assumption: 1) Unclear/Absent explanation of the scope the user is about to grant.

Attack: When obtaining user authorization, the user may not understand the scope of the access being granted and to whom, or they may end up providing a client with access to resources that should not be permitted.

3.6.2 *implicit grant.*

The same as authorization code grant

3.6.3 *Client Credentials grant.*

Attack: There is no resource owner intervention in this grant type. Thus the client might obtains a token with scope unknown for, or unintended by, the resource owner.

3.6.4 *Fix.*

1) The AS should explain the resources and the permissions in an understandable way. 2) The AS should narrow the scope, based on the client. The AS should consider what scope to grant based on the client type. 3)For the client credentials grant, the AS should force the local authorization policy, denying the request that is not permitted by the policy. 4) The AS should notify the resource owner for unusual request in the client credentials grant.

3.7 Obtaining credentials in the AS database

This attack is mentioned in part 4.3.2, 4.3.4, 4.4.1.2, 4.4.3.2 of the threat model[11]. The credentials in the AS database includes client secret, authorization code, access token, refresh token. Since it is related to insecure storage of the credentials in the AS, this exploit is independent of the grant types.

Assumption: 1) Clear text storage of credentials (storage protection is not employed). 2)SQL injection countermeasures are not enforced.

Attack: An attacker may obtain the credentials from the AS database by gaining access to the database or launching a SQL injection attack.

3.7.1 Fix. 1) Credential storage protection should be employed. 2) Store access token hashes only. 3)Enforce standard SQL injection countermeasures. A detailed description is in Section 4.

The following attacks happen in the communication between client and AS.

3.8 Counterfeit Resource Server

This attack is mentioned in part 4.6.4 of the threat model[11]. It is independent of the grant types.

Assumptions: 1) The attacker can manipulate the resource request 2) Client is allowed to make request with access token to unfamiliar RS. In another word, a client could try to use a token obtained for more than on RS. 3) No client authentication when client makes request to RS.

Attack: An attacker may 1) intercepts the resource request and modifies the request by replacing Honest RS to Malicious RS. 2) makes the client send request with the token to the attacker's RS. After obtaining the token, the malicious RS in turn may use that token to access resource at the honest RS.

Fix: 1) Clients should not make authenticated requests with an access token to unfamiliar resource servers 2) Associate an access token with a signature from the client when the client makes the resource request. The RS needs to verify the signature first then validate the access token. This idea is to bind a token with a cryptographic key, it is not enough to access the resource when someone accidentally obtained the token, he has to know the secret key as well. 3) Restrict the token scope and/or limit the token to a certain resource server.

4 DISCUSSION AND RECOMMENDATION

4.1 Discussion

The aforementioned attacks are largely caused by the implementations which sacrificed the security for simplicity. Only one attack is caused by the fundamental problem in the OAuth standard.The client does not receive information from where the redirect was initiated. However, In OAuth 2.0[6], this problem is fixed. I will discuss the security impact of several implementation decisions. The security evaluation aims to argue why the developer should not follow these decisions.

The client allows user to log in with identities provided by the AS. In the Authorized grant flow, the identities are authorized code and access token. In the implicit grant flow, the identities are the access token. In [15], they showed that an attacker can log into the client by simply using the victim's Facebook account identifier, which is publicly accessible. This allowance enables the attacker to log in the client application associated with the code/token. Using the victim's identity information, the client is vulnerable to the Impersonation attack, authorized toke theft and code substitution. The attacker can also log the victim into his client account to harvest the victim's data, which is the session swapping attack.

The authorization code is not used one-time. This decision violets the requirement in OAuth 2.0. The one-time use of the authorization code alleviates the impersonation attack, code substitution and Session swapping attack on the authorization grant. Even if the attacker obtains the authorization code, the code could probably be used so it is not valid anymore. Turning on the feature of multiple uses will help the attacker successfully launch the three aforementioned attacks.

The state parameter is not used one-time. It is widely known that the state parameter is not optional[7]. The state parameter is typically a value that is bound to the browser session (i.e. a hash of the function) against session swapping. Multiple uses of the state have the risk of session swapping once attacker obtained the value.

The automatic authorization grant feature adopted by Google might be indeed useful, but it can be harmful as well. The attacker can forge an authorization request to the AS if the resource owner has previously granted permission to a client application. Then the AS will directly send a new code to the client. It is very easy to forge such request due to prevalent web application vulnerabilities.

Secure storage protection is not employed. This is the most obvious and severe vulnerability. More detailed protection mechanisms is provided in section 4.2

The user should always be in control of the authorization processes and get the necessary information to make informed decisions. The attack described in section 3.6 results from the unclear explanation of the scope the user is about to grant. Moreover, user involvement is a further security countermeasure. The user can probably recognize certain kinds of attacks better than the authorization server. For example, the AS can notify the resource owner for unusual request to prevent the the unauthorized token grant.

Last but not least, the communication between any two of the entities must be protected by TLS (especially between the browser and the client). If not, the authorization code, the access token (in implicit grant) could be eavesdropped. Transmitting the credentials without encryption could make the client vulnerable to impersonation attack.

4.2 Recommendation

The recommendations contain all the simple yet practical fixes found in the literature work up to now. Some of the recommendations are targeted to mitigate the attacks in section 3. The rest of them aims to fix the other obvious attack. The considerations are grouped by entities in OAuth 2.0, for the AS only, I classified the recommendations further based on functionalities.

4.3 General

Confidentiality of communication: OAuth 2.0 requires the communication between the user-agent, the client, the AS, the RS should be protected by SSL/TLS[7]. Otherwise, the identities credentials are vulnerable to the impersonation attack

Create high entropy tokens. The AS should create tokens with a reasonable level of anonymity in order to mitigate the risk of guessing attacks.

Authenticate resource owner to the client:.* In section 4.1, I have mentioned that if a client allows users to log in with identities provided by the AS (i.e. authorization code and access token). The client is vulnerable to the Impersonation, authorization code theft, code substitution and session swapping attack. This unwanted feature may allow the attacker to perform operations at the legitimate client with the same permissions as the resource owner. However, the OAuth 2.0 does not provide the approach to authenticate resource owners. In the section 10.16 in [7], they claim that authenticating resource owners to clients is out of scope for the specification. Anyway, the developers should not rely on the identities provided by the AS to authenticate the user, they should define authentication schemes independent of the identity credentials.

4.4 Client

*Who should have the client credentials?** . In [12], they find an attack when the public client is issued with client credentials. The client credentials can be obtained by the attack. Which confirms the OAuth 2.0 recommendation. The authorization server must not issue client passwords or other client credentials to public clients for the purpose of client authentication. For the public client, the authorization server should employ other means to validate the client's identity.[7] For example, enforce the client to register its *redirection_uri*. Although it is not sufficient to verify the client's identity, it is useful to prevent the AS delivers the credentials (code, access token) to a counterfeit client after the resource owner's authorization. Another approach to mitigate the authentication gap of the public client is to ask the resource owner. The AS can engage the resource owner to assist identify the client.

4.5 AS

4.5.1 Client authentication and authorization.

Force single-use of authorization code: As discussed in section 4.1. Multiple uses of the authorization code make the client vulnerable to impersonation attack and session swapping attack.

Explicit user consent. As discussed in section 4.1. The automatic authorization grant feature makes the client vulnerable to authorization code theft. The user should consent for every authorization request even if the client asks for the same permission on the same resource. Two same requests do not often appear one after another in a short time. This is because 1) the access token has a period of lifetime. 2) The client could use the refresh token to request a new access token when the old one expires. If the developer decides to turn on the automatic authorization request eventually, a user consent is required for every authorization request originated from the client that asks for extended permissions[15].

Avoidance of iframes. In section 3.2, a hidden iframe element in the script can be used to transport a forged request to the authorization endpoint. In oAuth 2.0 standard[7]and [13], they discuss the clickjacking attack. An attacker registers a legitimate client and then constructs a malicious site which loads the AS Web page in a transparent iframe overlaid on top of a set of invisible buttons. When the user clicks a not important button, the user is actually clicking an invisible button (i.e. authorization button). If the AS use the "x-frame-options" header, any framing will be blocked or framing by sites with a different origin.

Referrer policies on the single-used state: Using referrer policies, a web server can instruct a web browser to suppress the Referer header when browser follows links in[4]. This is used to mitigate the state leak problem. If the attacker constructs a link in the response of the the *redirection_uri* which contains the state and the code. When the user clicks the link, the user's browser will send a request to the attacker's website. The Referer header of this request contains the *redirection_uri*, and the *redirection_uri* contains the state and the code.

Register full URI:. If the AS allows the client to register only part of the redirection URI, for example, domain-based URI, this will significantly reduce the security. Now the AS will deliver the credential to a counterfeit client using it's *redirection_uri* which has the same domain as the trusted *redirection_uri*.

Include the identity of the AS in the redirection_uri: The identity of the AS should be included as a parameter in the *redirection_uri* in step (3) in section 2.1. The client can then check that the parameter contains the identity of the AS it expects to receive the response from. Otherwise, the client is vulnerable to the AS mix up attack in section 3.4.

redirection_uri check.* The authorization server ensures that the *redirection_uri* used to obtain the authorization code is identical to the *redirection_uri* provided when redeeming the authorization code for an access token. Such easy fix could thwart the attacker to redeem the authorization code for an access token in the Honest server (AS mix up attack). However, in order to mitigate the code substitution attack, further action is needed. An invalid redirect URI indicates an invalid client, whereas a valid redirect URI does not necessarily indicate a valid client. So the bind of *redirection_uri* and the authorization code is necessary, since an attacker cannot use another *redirection_uri* to exchange an authorization code into a token.

client_id check.* The authorization server should bind every authorization code to the id of the respective client that initially authorized by the user. This is a countermeasure against code substitution, since an attacker cannot use another client_id to exchange an authorization code into a token. This binding should be protected from unauthorized modifications.

4.5.2 Refresh new token.

Refresh token represents a long-lasting authorization. Client uses this kind of token to obtain new access tokens used for resource server invocations. This design feature offers an advantage for access revocation [10]. As the refresh token is always exchanged at the authorization server. The authorization

server can revoke the refresh token at any time, causing the granted access to be revoked once the current access token expires[11].

Restricted Issuance of Refresh Tokens. Since refresh tokens are long-term credentials, they may be subject to theft. The AS should not issue refresh token to public client.

client_id check. Similar to clientid check in section 4.5.1, the AS should check that the same "client_id" is present for every request to refresh the access token. This is a countermeasure against refresh token theft or leakage.

Refresh Token Rotation. This fix aims to solve the problem when a stolen token is subsequently used by both the attacker and the legitimate client. The basic idea is to change the refresh token value with every refresh request in order to detect attempts to obtain access tokens using old refresh tokens. Once old refresh token usage is detected, AS will revoke the associated access.

4.5.3 Securely store the credentials.

Protection Mechanism. Mechanisms of protecting the storage of the credentials may include but not limited to the following countermeasures. A server system may be locked down so that no attacker gets access to the databases. In order to mitigate injection attack, AS should avoid dynamic SQL using concatenated input. Bind arguments should be used for parameterize queries. The input should be sanitized as well. The authorization server should store credentials hashes or encrypt credentials instead of storing in clear text.

5 RS

RS indicate its AS.* Another approach to fix the AS mix up attack is: The client should contact the RS first, then the RS will send the address of its AS back to the client. Now, even if an attacker intercepts the initial request by replacing the honest AS to the malicious AS. The client will not redeem the authorization code with the malicious AS with the knowledge of the honest AS address. However, in this case, the client needs a whitelist of the AS to prevent the compromised RS sends the address of a malicious AS.

Authenticate resource request. The resource server should be able to validate whether the authorization server issued the token to that client, otherwise, the RS is vulnerable to the counterfeit resource server attack. There are two approach to implement this feature. 1)The client uses his private key to sign the request to the resource server, the public key is either contained in the token or sent along with the request. 2) Alternatively, the authorization server may issue a token-bound key, the client can generate a MAC tag using the secret and send the MAC along with the access token [8]. RS will validate the MAC using the secret obtained from the authorization server before. Or the secret is contained in an encrypted section of the token.

6 CONCLUSION AND FUTURE WORK

This survey contributes in four aspects. 1) The summary of the conditions and restrictions of each grant type for different client applications. Argument why authorization code grant is more secure than the implicit mode. Analysis the benefit of authorization code feature in authorization code grant. Those statements shows the comprehensive understanding of OAuth 2.0 standard. 2) Eight most significant attacks under each OAuth 2.0 modes are considered, including the preconditions of each attack and description of each attack in detail. Eight attacks are group into two categories by where they happen. This survey provides security study on the client credential mode and the possible exploits during resource request (between client and the RS), which is an empty area in the research literature. Rather than believing what the paper states, I look at their work critically. For example, in [15], there is no automatically authorization grant for an access token in the implementation, however, Google implements the automatically grant for the authorized code. Also in [4], I questioned the possibility of performing AS mix up attack in the implicit grant flow in section 3.4. The critique shows the attempt to verify the attack on each mode based on the technical design details in OAuth 2.0 standard. 3) Discussion of the security impact of implementation decisions mentioned in the attack section to cover the gap in current research work. 4) Synthesis the simple and practical improvements to the implementation of OAuth 2.0. The improvements cover all of the entities in OAuth 2.0. For the AS, the survey investigates the security consideration further based on functionalities provided by the AS (e.g., refresh access token by refresh token). I bring up some original fixes as well, *RS indicates its AS, redirection_uri check, authenticate resource owner to the client, validation the identity of the public client.* The list of easy fixes can guide the developer to implement OAuth 2.0 more securely in different scenarios.(i.e. Web single sign-on system, OAuth based authorization in IoT).

The security analysis on the communication between AS and RS, token format, resource owner passwords credentials grant type are not considered in this survey. Future work may focus on these uncovered areas. The implementations of the actual exploits and result evaluations will be another direction for future investigation.

REFERENCES

[1] Suresh Chari, Charanjit S Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2. 0. *IACR Cryptology EPrint Archive* 2011 (2011), 526.

[2] Tim Dierks. Aug 2008. The transport layer security (TLS) protocol version 1.2„IETF, RFC 5246. *available at https://www.ietf.org/rfc/rfc5246.txt* (Aug 2008).

[3] Eugene Ferry, John O Raw, and Kevin Curran. 2015. Security evaluation of the OAuth 2.0 framework. *Information & Computer Security* 23, 1 (2015), 73–101.

[4] Daniel Fett, Ralf Küsters, and Guido Schmitz. Oct 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'2016).* Hofburg Palace, Vienna, Austria, 1204–1215.

[5] Paul Fremantle, Benjamin Aziz, Jacek Kopeckỳ, and Philip Scott. Sep 2014. Federated Identity and Access Management for the Internet of Things. In *Proceedings of the 2014 International Workshop on Secure Internet of Things - ESORICS Workshop.* IEEE, Wroclaw, Poland, 10–17.

[6] Eran Hammer-Lahav. April 2010. The OAuth 1.0 Protocol, IETF, RFC 5849. *available at https://tools.ietf.org/html/rfc5849* (April 2010).

[7] Dick Hardt. Oct 2012. The OAuth 2.0 Authorization Framework, IETF, RFC 6749. *available at https://tools.ietf.org/html/rfc6749* (Oct 2012).

[8] Phil Hunt, William Mills, Hannes Tschofenig, and Justin Richer. Nov 2014. OAuth 2.0 Message Authentication Code (MAC) Tokens. *ACE Working Group Internet-Draft. available at https://tools.ietf.org/id/draft-ietf-oauth-v2-http-mac-02.html* (Nov 2014).

[9] Wanpeng Li and Chris J Mitchell. Dec 2014. Security issues in OAuth 2.0 SSO implementations. In *Proceedings of the 2014 International Conference on Information Security (ICICS'2014).* Springer, Hong Kong, 529–541.

[10] T Lodderstedt and Stefanie Dronia. Aug 2013. OAuth 2.0 Token Revocation,IETF, RFC 7009. *available at https://tools.ietf.org/html/rfc7009* (Aug 2013).

[11] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. Jan 2013. OAuth 2.0 threat model and security considerations, IETF, RFC 6819. *available at*

https://tools.ietf.org/html/rfc6819 (Jan 2013).

[12] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Jun 2011. Formal Verification of OAuth 2.0 using Alloy Framework. In *Proceeding of the IEEE 2011 International Conference on Communication Systems and Network Technologies (CSNT'2011)*. Katra, Jammu, India, 655–659.

[13] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. May 2010. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web. available at https://crypto.stanford.edu/ dabo/pubs/papers/framebust.pdf* 2, 6 (May 2010).

[14] Ludwig Seitz, Goeran Selander, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. Aug 2016. Authentication and authorization for constrained environments (ace). *ACE Working Group Internet-Draft. available at https://tools.ietf.org/html/draft-ietf-ace-oauth-authz-10* (Aug 2016).

[15] San-Tsai Sun and Konstantin Beznosov. Oct 2012. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and communications security (CCS'2012)*. Raleigh, NC, USA, 378–390.

[16] Feng Yang and Sathiamoorthy Manoharan. Aug 2013. A security analysis of the OAuth protocol. In *Proceeding of the 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim'2013)*. Victoria, B.C., Canada, 271–276.