ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/326904828

OAuth-SSO: A Framework to Secure the OAuth-based SSO Service for Packaged Web Applications

Conference Paper · August 2018

DOI: 10.1109/TrustCom/BigDataSE.2018.00227

CITATIONS
13

READS
1,135

5 authors, including:

Nazmul Hossain
Jessore University of Science and Technology
26 PUBLICATIONS **79** CITATIONS
SEE PROFILE

Md. Alam Hossain
Jessore University of Science and Technology
51 PUBLICATIONS **288** CITATIONS
SEE PROFILE

Md. Zobayer Hossain
Jessore University of Science and Technology
4 PUBLICATIONS **17** CITATIONS
SEE PROFILE

Md. Hasan Imam Sohag
Jessore University of Science and Technology
2 PUBLICATIONS **13** CITATIONS
SEE PROFILE

All content following this page was uploaded by Shawon S. M. Rahman on 27 August 2018.

The user has requested enhancement of the downloaded file.

OAuth-SSO: A Framework to Secure the OAuth-based SSO Service for Packaged Web Applications

Nazmul Hossain
Assistant Professor, Department of Computer Science and engineering
Jessore University of Science & Technology
Jessore-7408, Bangladesh
nazmul.justcse@gmail.com

Md. Alam Hossain
Assistant Professor, Department of Computer Science and engineering
Jessore University of Science & Technology
Jessore-7408, Bangladesh
alamcse_iu@yahoo.com

Md. Zobayer Hossain
Department of Computer Science and engineering
Jessore University of Science & Technology
Jessore-7408, Bangladesh
zobayer130127@gmail.com

Md. Hasan Imam Sohag
Department of Computer Science and engineering
Jessore University of Science & Technology
Jessore-7408, Bangladesh
hasan.i.sohag@gmail.com

* Shawon Rahman, Ph.D.
Associate Professor, Dept. of Computer Science, University of Hawaii-Hilo
Hilo, Hawaii 96720, USA
sRahman@Hawaii.edu
* Corresponding Author

***Abstract*— The OAuth 2.0 is an authorization protocol gives authorization on the Web. Popular social networks, such as Facebook, Google and Twitter make their APIs based on the OAuth protocol to increase user experience of Single Sign-On (SSO) and social sharing. It is an open standard for authorization and gives a process for third-party applications to obtain users' resources on the resource servers without sharing their login credentials. SSO is an identification method that makes allowance for websites to use other, rely on sites to confirm users. This liberates businesses from the need to retain passwords in their databases, minimize on login troubleshooting, and reduces the damage a hacking can cause. OAuth 2.0 is broadly used in SSO service because of its simple implementation and coherence with a diversity of the third-party applications. It has been proved secure in different formal methods, but some vulnerabilities are revealed in practice. In this paper, we mention a general approach to improve the security of OAuth based SSO service for packaged web applications. We have modified method to execute OAuth flow from such applications with the help of SSO manages the life cycle of these applications.**

Keywords—Access Token, SSO Service, OAuth 2.0 Security; SSO; Web Applications Security, Packaged Web Apps; Social Networks; Authentication; Authorization

I. INTRODUCTION

Nowadays the development of the Internet, particularly the wide use of web 2.0, web applications (such as online shopping, instant messaging, wiki, etc.) have to be an important part of our regular lives. Application (hence forth referred to as "app") developers, especially client web app developers, want to utilize this medium to access various user resources, such as photos or videos maintained in some external resource server, in order to provide a richer and connected experience through their apps. To improve the situation, Single Sign-On (SSO) service comes out. In SSO, users are authorized to access various Relying Parties (RP for short) while logging into Identity Provider (IP for short) only at one time, which is openly convenient for users.

OAuth 2.0 [1] is broadly used in Single Sign-On service for its easy implementation and balance with a variety of third-party apps compared with other protocols (e.g. OpenID [2] and SAML [3]). As an open standard for authorization, OAuth gives an approach for third-party apps to access users' resources on the resource servers except sharing their user credentials. Though OAuth has been verified secure in several legitimate methods, the formal experiment was driven in abstract samples but not in the implementations. Several experimental studies have revealed some vulnerabilities and the implementation details, but these experimental studies concentrated on the case studies that cannot veil all the implementations. Therefore, it is still in necessary need of a general formula to dig out the security of SSO services.

In this paper, we propose a general approach to improve the security of OAuth-based SSO service. At first we discuss the parameters and the types of flows defined in the protocol, and design 5 attacks (A1-A5) in some prolepsis. After then the approach containing 9 detection terms (I1-I9) and an app id is provided to check the prolepsis. In last section, we arrange an experiment on some typical IPs and RPs. The results show that the approach is simple and easy-to-use.

II. FANDAMENTALS OF SSO BASED OAUTH

At first, the OAuth protocol is schematic specifically to impede access tokens from revealing in the network, and still we found that abundant access tokens gained on the browser side that are dispatched in unshielded form the RP server side for the intention of authentication state synchronization. In some RPs, access tokens are connected as query parameters to the RP's sign-in endpoint, which published the tokens in the browser's history and server logs. In addition, to simplify accessibility, IPs' JavaScript SDKs or RPs themselves store access tokens into HTTP cookies, and hence opens the tokens to a wide range of attacks (e.g., network eavesdropping, XSS cookie theft). Amazingly, our evaluation shows that only 20% of RPs service SSL to defend SSO sessions, even if about half of tested RPs have preserved their conventional login forms with SSL.

Second, and more amazingly, access tokens can be robbed on most (91%) of the evaluated RPs, if a rival could utilize an XSS vulnerability on many page of the RP website. Apparently, an XSS vulnerability found on the login page of an RP for that access tokens are gained on the browser-side (i.e., client-flow) could grant an opposition to steal access tokens by the SSO process. Although, our test utilizes even succeeded on RPs that gain access tokens only by a direct communication with the IP (i.e., server-flow, not via browser), neglectful of whether the user has herein before logged into the RP website, and when the redirect URL is SSL-secured. XSS vulnerabilities are dominant [4, 5] and their complete subsidence is shown to be hard [6].

Third, the RP website is free from XSS vulnerabilities, cross-site access token theft should have carried out by leveraging certain vulnerabilities that is found in browsers. We improved and tested two such utilize scenarios in which the vulnerable browsers are yet used by about 10% of web users [7]. The first utilize executes the token stealing script attached in an image by leveraging the browser's content-sniffing algorithm [8]. The second one snitch an access token by transmission a sham authorization request by a script component and then it extracting the token though on error event handler that holds cross-origin vulnerability [9].

On the other hand, to access tokens, our appraisal results exhibit that an attacker might obtain whole control of the victim's account on abundant RPs (64%) though sending a phony SSO credential to the RP's sign-in endpoint by a user-agent controlled by the attacker. amazingly, few RPs gain the user's IP account profile on the client-side, so at that point pass it as an SSO credential to the sign-in endpoint on the server side to identify the user. Though, this approves an attacker to reincarnate the victim user by easily using the victim's publicly available Facebook account identifier.

Different CSRF activities may be leveraged to compromise users' data situated on RPs and aid XSS token stealing attacks. When the authenticity of SSO credentials like access token, authorization code, or user identifier is not confirmed though the receiving RP website, this vulnerability could be utilized to advancement a session swapping attack [10], What forces victim user to sign into the RP so the attacker in order to hype the victim's private information (e.g., tricks the victim into linking his credit card to the attacker's account), or improvement an XSS attack as we invented. Furthermore, due to inadequate CSRF protection for RPs, many tested RPs are vulnerable to force-login attack [11] which permits a web attacker to unknowingly force victim user to sign into the RP. After an effectual force-login attack, our appraisement found that an opponent might use CSRF attacks to deflect the users' profile information on 21% of the assess RPs. More amazingly, we found that a session swapping or force-login vulnerability may be leveraged to (1) surpass an attack obedience in that an authenticated session with the RP is prerequisite for a effectual XSS utilize, and (2) bootstrap a token stealing attack by tempting a victim user to view a maliciously crafted page someplace on the web, when a user's RP account information is not sanitized for XSS.

Unlike logic flaws, the primary reasons of the open vulnerabilities cannot easily be withdrawn with a software patch. Our investigation reveals that those uncovered investigations are caused though a mixer of implementation plainness features offered through the sketch of OAuth 2.0 and IP implementations, for example the dismissal of the digital signature from the protocol specification, the clinch of client-flow, and a self-acting authorization granting" characteristic. While these plainness characteristic might be uncertain for security, they are what permit OAuth SSO to acquire rapid and comprehensive adoption.

We purposed to sketch practical mollification mechanisms which might prevent or reduce abate the open threats without immolating plainness. To be practical, our proposed improvements do not need change from the OAuth protocol or browsers and may be adopted though IPs and RPs slowly and differently. In addition, the suggested solicitation do not necessary cryptographic operations from RPs. This is caused by understanding niceties of signature algorithms. Then how to build and sign their foundational string is the common problems for many SSO RP developers [12].

As like OAuth SSO systems are being employed to guard billions of Clint accounts on IPs and RPs, the insights from our tasks are practically significant and urgent and might not be gained without an in-depth analysis and evaluation. To condense, this task makes the subsequent contributions: (1) the 1st experimental inquiry of the security of a delegate specimen of highest-visited OAuth SSO implementations, and an invention of different critical vulnerabilities, (2) an evaluation of the invented vulnerabilities and a gauging of their outbreak across RP implementations, and (3) an improvement of practical recommendations for IPs and RPs to secure their implementations.

III. OAuth 2.0 Authentication Flows

OAuth-based SSO systems are based on browser redirection that an RP redirects the Clint browser to an IP which collaborate with the Clint before redirecting the Clint back to the RP website. The IP authenticates the Clint, recognizes the RP to the Clint, and asking for consent to offer the RP access to resources and services on the side of Clint. Once the implored consents are granted, the Clint is redirected back to the RP with an access token which illustrates the granted permissions. With the approved access token, the RP then invokes web APIs revealed by the IP to access the clients profile attributes.

The OAuth 2.0 specification which defines two flows for RPs to gain access tokens: server-flow (known as the Authorization Code Grant" in the specification), intended for web applications that receive access tokens from their server-side program logic; and client-flow (known as the \Implicit Grant") for JavaScript applications running in web browser [13]. Figure 1 illustrates the following steps, which exhibit how server-flow works:

1. User X clicks on the social login button, and the browser Y sends this login HTTP request to RP.

2. RP sends response type=code, client ID *ci* (a random unique RP identifier assigned during registration with the IP), requested permission scope p, and a redirect URL *ru* to IP via Y to obtain an authorization response. The redirect URL *ru* is where IP should return the response back to RP (via Y). RP could also include an

optional state parameter a, which will be appended to *ru* by IP when redirecting X back to RP, to maintain the state between the request and response. All information in the authorization request is publicly known by an adversary.

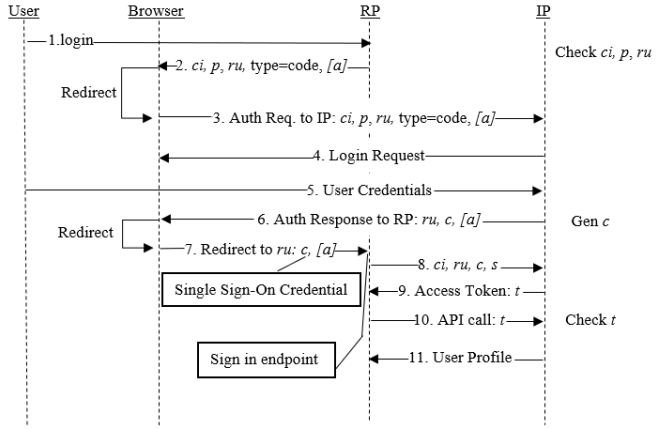

Fig. 1. The Server Flow in OAuth 2.0

3. Y sends response_type=code, *ci*, p, *ru* and optional a to IP. IP checks *ci*, p and *ru* against its own local storage.
4. IP presents a login form to authenticate the user. This step could be omitted if X has already authenticated in the same browser session.
5. X provides her credentials to authenticate with IP, and then consents to the release of her profile information. The consent step could be omitted if p has been granted by X before.
6. IP generates an authorization code c, and then redirects Y to *ru* with c and a (if presented) appended as parameters.
7. Y sends c and a to *ru* on RP.
8. RP sends *ci*, *ru*, c and a client secret s (established during registration with the IP) to IP's token exchange endpoint through a direct communication (i.e., not via Y).
9. IP checks *ci*, *ru*, c and s, and returns an access token t to RP.
10. RP makes a web API call to IP with *t*.
11. IP validates t and returns X's profile attributes for RP to create an authenticated session.

The client-flow is designed for applications that cannot embed a secret key, such as JavaScript clients. The access token is returned directly in the redirect URI, and its security is handled in two ways: (1) The IP validates whether the redirect URI matches a pre-registered URL to ensure the access token is not sent to unauthorized parties; (2) the token itself is appended as an URI fragment (#) of the redirect URI so that the browser will never send it to the server, and hence preventing the token from being exposed in the network. Figure 2 illustrates how client-flow works:

1. User X initiates an SSO process by clicking on the social login button rendered by RP.

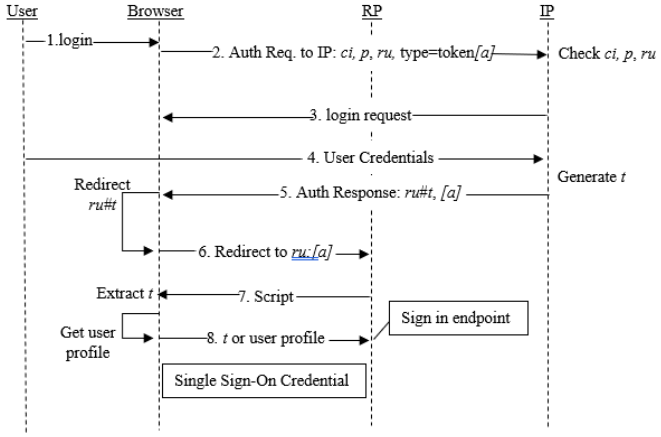

Fig. 2. The Clint Flow in OAuth 2.0

2. B sends response_type=token, client ID *ci*, permission scope p, redirect URL *ru* and an optional state parameter a to IP.
3. Same as sever-flow step 4 (i.e., authentication).
4. Same as sever-flow step 5 (i.e., authorization).
5. IP returns an access token t appended as an URI fragment of ru to RP via Y. State parameter a is appended as a query parameter if presented.
6. Y sends a to ru on RP. Note that Y retains the URI fragment locally, and does not include t in the request to RP.
7. RP returns a web page containing a script to Y. The script extracts t contained in the fragment using JavaScript command such as document.location.hash.
8. With t, the script could call IP's web API to retrieve X's profile on the client-side, and then send X's profile to RP's sign-in endpoint; or the script may send t to RP directly, and then retrieve X's profile from RP's server-side.

IV. Releted Work

In OAuth-based SSO service, a resource sever is discussed as an IP to handle users' identities and identifies the user. Since a third-party application is discussed as a RP to set users and authenticate the user with user's profile gotten from IP. In short, the user's profile hosted on IP is authorized and shared for RP is to identify the user. Note that IP and RP are correlative concepts, as, one could be either IP or RP in different scenarios.

In theory, several formal approaches have been used to examine the OAuth, such as Alloy framework used by Pai et al. [14], universally compostable security framework used by Chari et al. [15] and Murphy used by Slack et al. [16], and all the results were included in the official OAuth security guide [17]. Thus, the implementation following the guide is secure in theory. On empirical analysis, Wang et al. [18] focused on the original web

traffic going by the browser and discovered several logic flaws in some SSO services (e.g. Google ID, Facebook) [19], which are used by attackers to tamper with the authentication messages. Sun et al. [11] examined the implementation details of three major OAuth-based IPs including Facebook, Microsoft and Google, and uncovered several vulnerabilities that gives attackers to obtain access to the user's profile and to reincarnate the user on RP.

Actually, all the existing approaches are deficient. The formal analyses were conducted in the abstract models, and some implementation details could be left out, which has been verified by the empirical studies; while the existing empirical studies exposed the vulnerabilities in the case analyses, which could omit some proofs.

In OAuth 2.0, the cryptography is taken out, and its transport security depends on the protocol SSL/TLS or HTTPS. However, HTTPS protection has no effect on the messages hosted on the end-points (issuer, browser and receiver). Meanwhile, session management is a vital task during the authentication, especially the undetectable session S (cf. step C3 or T3). In this paper, we focus on the overall security of OAuth including transport security, endpoint security and session security, and propose a general approach to detect the security of SSO. On protocol analysis, we examine the flows and five variable parameters, and summarize their usages, requirements and potential risks. On empirical study, 5 attacks based on protocol analysis are proposed, and 10 IPs (including qq.com, weibo.com, renren.com and baidu.com) and 136 login flows using by 50 RPs (such as dianping.com, jumei.com etc.) are checked to validate the availability of the approach.

V. Proposed Framework

Our analytic framework is to provide a general approach to secure the OAuth-based SSO service. In the stage of protocol analysis, we review the protocol carefully to reveal the usages, requirements and potential risks of the five variable parameters and session S (S and X1-X5 in Fig. 1 and Fig. 2). Five attacks (identified by A1 to A5) based on the former are provided in the latter stage, all of which have been proved available in the following experiment. And the analysis on the presuppositions of the five attacks reveals that we need only to execute an detection on item I1 to I9 as follows to evaluate the security of an SSO service:

I1: Whether HTTPS protection is deployed by RP.

I2: Whether an unpredictable state parameter is used by RP.

I3: Whether RP is prevented against CSRF attack.

I4: Whether RP stores the access token in the cookie or URI.

I5: Whether the code is cross in use.

I6: Whether IP supports the two response types simultaneously and indiscriminately.

I7: Whether any redirection URI in the realm of RP could pass IP's checking.

I8: Whether the token is a bearer token.

I9: Whether a mechanism to end the session S is provided.

Based on the analytic framework, the proposed approach can be described with the pseudo-code as follows:

```
AttackList Detection() {

    // Parameters X[1-5] and S are defined in Section III;

    // Attacks A[1-5] are defined in Section III;

    // The symbol * means the attack has greater damage.

    AttackList attacks = NULL; // all the potential attacks.

  if(CHECK(X4.I1: HTTPS is employed by RP) = TRUE) {


    if(CHECK(X4.I5: The code is cross in use) = TRUE)
        attacks.Add(A1*);
    else
        attacks.Add(A1);
}

 if(CHECK(X1.I6: Two response types are supported by IP)  =
                          TRUE AND
    CHECK(X5.I8: The token is a bearer token) = TRUE){
   if(CHECK(X2.I7: The realm URIs are checked out by IP)
        = TRUE)
      attacks.Add(A2*);
  else
      attacks.Add(A2);
}

if(CHECK(X5.I4: The access token is stored incorrectly
by RP) = TRUE AND
    CHECK(X5.I8: The token is a bearer token) = TRUE){
  if (CHECK(X5.I1: HTTPS is employed by RP) = FALSE)
        attacks.Add(A3*);
  else
        attacks.Add(A3);
}

if(CHECK(X3.I3:Protection against CSRF is employed by RP)
        = FALSE){
  attacks.Add(A4);
  // check Item X3.I2': Whether state is invalid.
  if(CHECK(X3.I2: The state parameter is used by RP)
        = TRUE)
     Warning(The state parameter(X3) is INVALID);
}

if(CHECK(S.I9:Ending session S is provided by RP)= FALSE)
        attacks.Add(A5);
  return attacks;
}
```

VI. Analysis On Proposed Framework

A. Protocol Analysis

In the authentication flows, five variable parameters are defined, and two authentication sessions are created. In the rest part of this section, we discuss the usages, requirements and potential risks of the parameters and the session S. [20]

X1. Response Type (response_type)

Usage: It is set by RP and used to select which flow to be used in the following sequence, that is, to decide the way for RP to gain an access token.

Risk: As mentioned before, the both flows are similar in user's view, so attackers could set response_type = token, and gain the access token from the browser through a script without user's awareness.

X2. Redirection URI (redirect_uri)

Usage: It is set by RP and used to decide the destination of request C6 or T6, which receives the code or access token.

Requirement: IP should check it strictly to avoid sending the code or access token to other receivers except RP.

Risk: The receiver, besides an attacker, of the code or access token could log into RP as the victim, and gain the victim's profile.

X3. State Parameter (state)

Usage: It is generated by RP for tracing the session between browser and RP to prevent against CSRF attack.

Requirement: It should be an unpredictable value bound to the session with RP.

Risk: It is an optional parameter. If it is lost, another countermeasure against CSRF attack MUST be taken specified in the protocol. In practice, CSRF attack is often neglected by developers.

X4. Authorization Code (code)

Usage: It is sent to RP in the server flow and used for RP to request an access token from IP.

Requirement: It MUST be an expiring and one-time token and be transported in a channel with HTTPS protection specified in the protocol.

Risk: If gaining it and using it before the victim, the attacker could impersonate the victim and log into RP, which has a greater risk if the code could be used for multiple RPs, e.g. the code sent to RP1 could be used to log into RP2.

X5. Access Token (access_token)

Usage: It is generated by IP and used for RP to make web API calls to gain or handle user's profile. RP use the received profile to authenticate the user.

Requirement: It should be an unpredictable value and not be sent to the others except RP.

Risk: Due to the decryptographic design of OAuth, the access token is often implemented as a bearer token. That is to say, any bearer of the token, e.g. an attacker, could access or handle user's profile.

S. Session with IP

Usage: It is created by IP in step C3 or T3 and used to authenticate the user for the latter login.

Requirement: It is often maintained through the cookie technology.

Risk: After created, the session keeps alive and is only bound to cookies invisible to the user, so other mechanisms should be employed to clear the session.

In conclusion, the misuse of each of X1 to X5 and S could result in unsecure factors, thus understanding the strict requirements are a precondition for implementing an invulnerable OAuth-based SSO service.

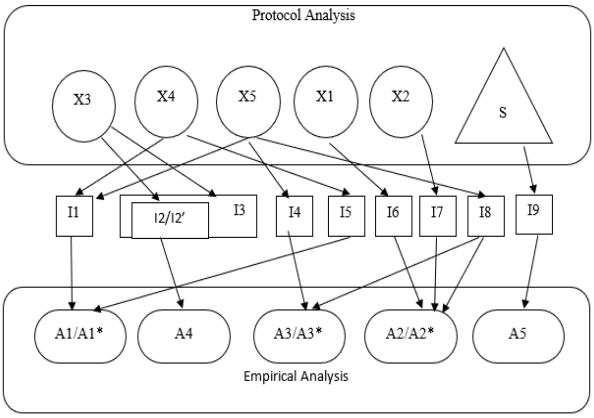

Fig. 3. Analytic Framework of Detecting the Security on Parameters and Session S [25].

B. Empirical Analysis

The preceding outcome express that the total security requires suitable developers to complete the SSO service. In practice, simplicity of OAuth misguides the developers. Most of the time they cannot generate the implementations that meet the necessities. So, in this section, accordingly all the necessities, 10 IPs and 136 authentications used by 50 RPs are experimented, and the outcome denote that risks lay in the cautiousness although the OAuth-based SSO service is broadly used. [20]

A1. Theft & Embezzling Authorization Code

In server flow, HTTPS protection is used for all the contacts with IP, but not for one or more submitting

the authorization code to 90% of RPs. So, an attacker can steal the authorization code & influences the victim fail to submit it. It permits the attacker to log into RP as the victim with the code.

Cross use of the code is not Identified in our study, but it is mentioned that the authorization code created by weibo.com for RP1 can be used to log into RP2 [21]. In this situation, the risk is of larger damage.

Strikingly, 30% of RPs defend their own login page with HTTPS, but only 9.56% of RPs provide HTTPS protection for the authorization code.

A2. Stealing Access Token via XSS

All the detected IPs sustain the two streams and try not to limit's their RPs to choose that flow to utilize. Thusly, XSS vulnerability in redirect_uri page on RPs utilizing the user flow, as Sun et al. [11] put it, permits attackers to dispatch the client-flow sequence & to correct the access token with the script.

In the interim, for RP with XSS vulnerability, the attacker might built a link with response_type = token & trick user to click on it, and the corresponding way might be used to obtain access tokens after user enter his/her passwords.

What is more awful, most of IPs only check whether the redirection URI is in the reign of RP, what causes that this attack might be propelled in the event that any page on RP has XSS vulnerability the attack surface is broadened.

A3. Eavesdropping or Stealing Access Token

The bearer token is deployed in all the detected IPs. These is the only identity for RP to access clint's profile, In other words, the one who occupies the token might be observed as the right RP to access clint's profile. So RP will undoubtedly ensure its confidentiality.

As a matter of fact, a few RPs keep the access token in the cookie or URI except any protections. Therefore, it is likely to be theft from the browser, For RPs except HTTPS protection, even an attacker might eavesdrop on the open traffic to receive the access tokens.

A4. CSRF Attack

It is indicated in OAuth which is CSRF attack, ranking 8 in Top-10 chart issued by OWASP [4], must be dealt with. Also, the state parameter is recommended to deal with it. Except a protection, an attacker might launch the attack as follows:

(a) The attacker beginning a server-flow order with his/her own account but stops earlier & saves the request URI named csrf_uri.

(b) csrf_uri is issued in the Internet with the blog or microblog, and a victim click on the csrf_uri will sign into RP as the attacker.

(c) Because of the victim sign in as the attacker, the attacker can record the victim's information.

At first glance, the authorization code, all things considered, is for one-time utilize, although CSRF attack is wasteful & immaterial. However, binding with a local RP account is needed in some RPs, so the victim may bind one's own RP account to the IP account, as such, the attacker can control victim's RP account with the IP account.

The results explicit that 44.12% of RPs take CSRF attack into consideration, and 39.71% adjust the suggestion utilizing the state parameter. It is important that a little part of RPs set the state parameter as a permanent value, and that about 25% of RPs try not to check the parameter exactly & accept the request except the parameter, all of which fall flat in opposition to CSRF attack.

A5. Risk of the Implicit Session with IP

As the session lays in the browser implicitly, the user can't end it by closing some page after login, which may initiate the following risk: The user logs into RP with the account on IP on a common computer, then logs out of RP but not IP when leaving, that is, the session with IP is alive. The latter user on the same computer can log in to a RP, even other RPs, e.g. jumei.com, as the identity of the former user by just clicking a button.

For this reason, logging out of the IP is a recommendation for a complete solution to refrain from the aforementioned risk. Nevertheless, none of the detected RPs maintains a mechanism to end session S though parts of IPs offer an interface to log out.

VII. RESULTS

To begin an assessment process, the evaluator signs into the RP in question using both traditional and SSO options through a Firefox browser. The browser is augmented with an add-on we designed that records and analyzes the HTTP requests and responses passing through the browser. To resemble a real-world attack scenario, we implemented a website, denoted as attacker.com, which retrieves the analysis results from the trace logs, and feeds them into each assessment module described below. Table 1 shows the summary of our evaluation results. We found 42% of RPs use server flow, and 58% support client-flow; but all client-flow RPs use Facebook SDK instead of handling the OAuth protocol themselves.

Table 1: The Percentage of RPs that is vulnerable to each exploit [22]

RPs			SSL (%)		Vulnerabilities (%)				
Flow	**N**	**%**	**T**	**S**	**A1**	**A2**	**A3**	**A4**	**A5**
Client	56	58	20	6	25	55	43	16	18
Server	40	42	29	15	7	36	21	18	20
Total	96	100	49	21	32	91	64	34	38

VIII. CONCLUTIONS

OAuth 2.0 is attractive to RPs and easy for RP developers to implement, but our investigation suggests that I is too simple to be secure completely. Unlike conventional security protocols [23], OAuth 2.0 is designed without sound cryptographic protection [24], such as encryption, digital signature, and random nonce [25]. The lack of encryption in the protocol requires RPs to employ SSL, but many evaluated websites do not follow this practice. Additionally, the authenticity of both an authorization request and response cannot be guaranteed without a signature. Moreover, an attack that replays a compromised SSO credential is difficult to detect, if the request is not accompanied by a nonce and timestamp [26]. Furthermore, the support of client-flow opens the protocol to a wide range of attack vectors because access tokens are passed through the browser and transmitted to the RP server. Compared to server flow, client-flow is inherently insecure for SSO. Based on these insights, we believe that OAuth 2.0 at the hand of most developers without a deep understanding of web security [28][29] is likely to produce insecure implementations.

To protect web users in the present form of OAuth SSO systems, we suggest simple and practical mitigation mechanisms. It is important for current IPs and RPs to adopt those protection mechanisms in order to prevent large-scale security breaches that could compromise millions of web users' accounts on their websites. In particular, the design of server flow makes it more secure than client-flow, and should be adopted as a preferable option, and IPs should offer explicit flow registration and enforce single-use of authorization code. Furthermore, JavaScript SDKs play a crucial role in the security of OAuth SSO systems; a thorough and rigorous security examination of those libraries is an important topic for future research.

REFERENCES

[1] The OAuth 2.0 authorization framework, IETF Std. RFC6749, 2012

[2] F. Brad, R. David, H. Dick, and H. Josh. (2013) OpenID authentication 2.0-final. [Online]. Available: http://openid.net/specs/openid-authentication-2_0.html

[3] OASIS. (2013) SAML specifications. [Online]. Available: https://wiki.oasis-open.org/security/FrontPage

[4] OWASP. Open web application security project top ten project. http://www.owasp.org/ , 2017.

[5] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In Proceedings of IEEE Symposium on Security and Privacy, 2010.

[6] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In Proceedings of the 20th USENIX Conference on Security, Berkeley, CA, USA, 2011.

[7] W3CSchool. Browser statistics. https://www.w3schools.com/browsers/default.asp , 2012. [Online; accessed April 4, 2018].

[8] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In Proceedings of the 30th IEEE Symposium on Security and Privacy, SP '09, pages 360{371, Washington, DC, USA, 2009.

[9] OSVDB. window.onerror error handling URL destination information disclosure. http://osvdb.org/68855 (and 65042) [Online; accessed April 4, 2018].

[10] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08), pages 75{88, New York, NY, USA, 2008. ACM

[11] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. Computers & Security, 2012.

[12] L. Shepard. Under the covers of OAuth 2.0 at Facebook. http://www.sociallipstick.com/?p=239 , 2011. [Online; accessed 11-February-2018].

[13] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In Proceedings of the 18th Conference on USENIX Security Symposium, 2009.

[14] S. Pai, Y. Sharma, S. Kumar, R.M. Pai, and S. Singh, "Formal verication of OAuth 2.0 using Alloy framework," in Proc. CSNT'11, 2011, p. 655-659.

[15] S. Chari, C. Jutla, and A. Roy. (2011) Universally composable security analysis of OAuth v2.0. [Online]. Available: http://eprint.iacr.org/2011/526.pdf [Online; accessed April 4, 2018].

[16] Q. Slack, and R. Frostig. (2011) OAuth 2.0 implicit grant flow analysis using Murphi. [Online]. Available: http://www.stanford.edu/class/cs259/WWW11/ [accessed April 4, 2018].

[17] OAuth 2.0 Threat Model and Security Considerations, IETF Std. RFC6819, 2013.

[18] 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 Proc. SP'12, 2012, p. 365-379.

[19] OWASP. Open web application security project top ten project. http://www.owasp.org/, 2017 [accessed April 4, 2018].

[20] R Zhu, J Xiang, D Zha. Research on the Security of OAuth-Based Single Sign-On Service. The Steering Committee of the World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.

[21] lhshaoren. (2010) A vulnerability on *weibo.com* (*sina.com*). [Online]. http://www.wooyun.org/bugs/wooyun-2010-039727. [accessed April 4, 2018].

[22] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security, Pages 378-390, Raleigh, North Carolina, USA — October 16 - 18, 2012.

[23] V Beltran. Characterization of web single sign-on protocols. IEEE Communications Magazine, 15 July 2016.

[24] Daniel Fett, Ralf Küsters, Guido Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security Pages 1204-1215, Vienna, Austria — October 24 - 28, 2016

[25] V Sucasas, G Mantas, A Radwan, J Rodriguez. An OAuth2-based protocol with strong user privacy preservation for smart city mobile e-Health apps. Communications (ICC), 2016 IEEE International Conference on Communications (ICC), 22-27 May 2016.

[26] Caimei Wang; Yan Xiong; Wenchao Huang; Huihua Xia; Jianmeng Huang; Cheng Su. A Verified Secure Protocol Model of OAuth Dynamic Client Registration. 2017 3rd International Conference on Big Data Computing and Communications (BIGCOM), 2017.

[27] Loukaka, Alain and Rahman, Shawon; "Discovering New Cyber Protection Approaches From a Security Professional Prospective";

International Journal of Computer Networks & Communications (IJCNC) Vol.9, No.4, July 2017

[28] Al-Mamun, Abdullah, Rahman, Shawon and et al;“ Security Analysis of AES and Enhancing its Security by Modifying S-Box with an Additional Byte ”; International Journal of Computer Networks & Communications (IJCNC), Vol.9, No.2, March 2017

[29] Opala, Omondi John; Rahman, Shawon; and Alelaiwi, Abdulhameed; “The Influence of Information Security on the Adoption of Cloud computing: An Exploratory Analysis”, International Journal of Computer Networks & Communications (IJCNC), Vol.7, No.4, July 2015

View publication stats