Security in The Representational State Transfer API

Using OAUTH2 for Authorization

Abhibandu Kafle
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL

Alicia Lindsey
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL

***Abstract*—In this paper we discuss security and the REST API. Specifically discussed is a security implementation using OAUTH as part of the security framework to protect access to resources (data and services). Security of resources can be implemented either by basic authentication or by methods which include both authentication and authorization. Traditionally authentication only has been used to protect resources however it is difficult using authentication alone to provide varying levels of access to resources, basically using authentication only the user will receive all access or no access. The more developed the web becomes the more important it is to provide a finer level of control to resources, more than the 'all or nothing' approach authentication only provides. Combining authentication with authorization can provide fine control of access to resources as defined by rules put in place by the system administrator. OAUTH provides a framework for authorization and when combined with an effective authentication method can provide the flexibility in access control required by modern web services.**

The OAuth 2 spec itself leaves numerous choices over to the practitioner. Rather than portraying every single conceivable choice that should be made to effectively actualize OAuth 2, this paper settles on choices that are fitting for most usage. This paper is an attempt to portray OAuth 2 in a disentangled configuration to help designers and administration suppliers actualize the security aspect of the convention.

Keywords—OAuth2.0, Authorization, Security

1. INTRODUCTION

As the internet has become more developed there exists a need for computer applications and computer resources to share information without human intervention. Sharing data from web servers to applications is generally referred to as web services. In order to allow developers to more easily provide applications and services standardized methods of interfacing applications to services have been developed. One of those methods is Representational State Transfer (REST). The REST architecture provides a standardized method of accessing web based resources. The REST API has become popular due to the fact that it uses existing HTTP technology to access and transfer resources between applications and web based resources thereby reducing server and client complexity allowing for quicker application and server development.

In order for a web service to be considered RESTful it must meet the following constraints:

1. Have a uniform interface
2. Be stateless
3. Be cacheable
4. Be a client-server system
5. Be a layered system
6. Support code on demand (optional)

The REST architecture is both lightweight and scale-able using HTTP to connect computers for exchanging information. The HTTP operations implemented by RESTful services are:

1. GET: retrieve resources
2. POST: create resources
3. PUT: update resources
4. DELETE: delete resources

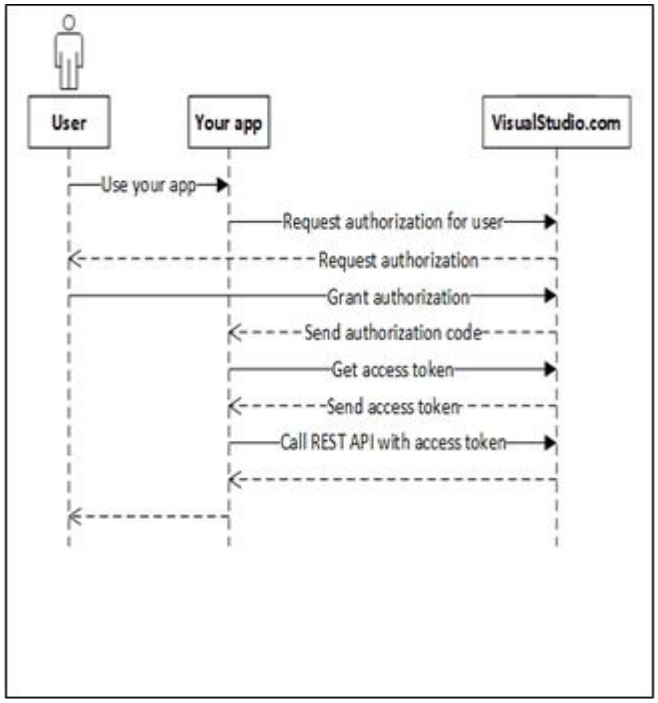
REST SECURITY REST applications are subject to many of the same vulnerabilities that web servers in general are exposed to. Security must be implemented in REST services wherever the resources controlled by the service require protection. Resources require protection in every phase of the transfer process:

1. Access protection on the server – Authentication and authorization
2. Protection from eavesdropping during transport to the application – HTTPS/SSL
3. Protection once delivered to the application – Depends on application requirements

Protection from eavesdropping during transport (can be provided by HTTPS/SSL) and protection once the application receives the data is beyond the scope of this paper. This paper is focused on using OAUTH as a method of providing access authorization in order to protect data on the server.

REST AND OAUTH2 REST and OAUTH work together to allow users to authorize an application to access resources on behalf of the user. The REST services provide access to the resources only after obtaining authorization via the OAUTH protocol. The OAUTH protocol in turn uses a separate method to provide authentication of a user prior to granting authorization to the API as approved by the user. Authorization to all or part of a resource can be granted via the OATH protocol, the ability to provide access to only part(s) of a resource as approved by the resource owner is part of what differentiates the OAUTH protocol from basic authentication where 'all or none' of the resource can be authorized. OAUTH also provides a standardized API for applications to access the resources via REST using the granted authorization. In Section 4 "Authorization Using OAuth2", we will provide the details of OAuth2.

In Figure 1, the diagram shows the process of an application requesting authorization to access the user's data. When the user runs an application that needs data from another application, the authorization mechanisms are carried out. Then, once authorization has been granted, the client's application issues a resource request that includes the authorization token using a REST request message.



2. BACKGROUND

As web services on the internet have evolved the need for higher levels of security has arisen, initially only requiring limited protection. As more and more data has been made available on the web the need for improved security to access the data has caused a need for more secure authentication mechanisms. As even more data is made available the need for not only secure authentication but also finer control of authorization to portions of a resource are required. There are several methods that are used to provide authorization to access resources.

A. HTTP Basic Authentication

HTTP basic authentication requires an "authorization request" containing the user name and password to be sent to the server with each request in turn requiring the application to store a copy of the user's name and password in order to send it to the server. Storing the user's name and password on the device poses a risk to the user in cases where the device is lost or hacked. Each application which requires access to resources on the user behalf will be required to store a copy of the name and password which further increases the risk. Early implementations sent the user's name in clear text along with a minimal encryption of the user's password to the server possibly allowing eavesdropping. Later implementations use HTTPS/SSL to encrypt the transport which improves security during the transport phase. If the user's name and password are compromised the only way to prevent inappropriate access to the protected resource is to change the user's password, if for instance the user's device is stolen all the user's passwords on all services the user accesses via the device would have to be changed. This places a high burden on the user in order to prevent inappropriate access of the user resources. Additionally only very course control of the resource can be achieved, only 'all or nothing' access can be granted or revoked by changing the password.

B. OAUTH1.0a

OAuth was developed to allow third-party applications to gain access to an owner's resources without the owner having to reveal his user name and password to the server that possesses the resource or to the application which needs access to the user's resource. The first widely deployed version of OAuth was OAuth1.0 which was found to have a security flaw and was replaced by OAuth1.0a which had several issues including:

1. The signature scheme used is complex to implement
2. The tokens used to control access do not expire

OAuth 1.0 used notion of signing the requests using client ID and secret. Oauth 2.0 replaces signatures simply uses https for communication between all parties involved. OAuth 1.0 wasn't really scalable because it required temporary credentials, and it was difficult to synchronize the data across different data centers.

C. *OAUTH2.0*

OAuth2.0 was developed to solve the issues of OAuth1.0a as well as the following:

1. More support for non-web applications (desktop applications, mobile devices)
2. More scale-able for larger projects.

OAuth2.0 is more similar to a framework than simply a protocol leaving parts of the implementation to be defined by the user which can cause implementations not to be inter-operable. OAuth2.0 is not directly compatible with Oauth1.0a. OAuth 2 also provides password as a grant type option. This means instead of exchanging tokens, applications can use username and password to authenticate a user to a system. However, this is usually done only within an application. For an instance Facebook messanger app could use OAuth 2 to get login information from its native mobile application.

3. RELATED WORKS

In this paper we have taken the approach of using OAuth2 as the method for providing authorization to access resources. When OAuth2 was released in October of 2012, OAuth1.0a was obsoleted. However, OAuth2 has been plagued by many problems, even before it was released.

When the OAuth2 project was in a mature stage, one of the main contributors left the project. He was very unhappy with how the specification had progressed. He did not approve of the final product so much that he did not want his name associated with the product. He concluded that OAuth2 is a bad product. He believed this occurred due to compromising all along the way. He stated that the result was that two main goals were not going to be provided by OAuth2. These are security and interoperability.

Some companies are not transferring to OAuth2, but many other big companies have adopted OAuth2.

However, work continues on OAuth2 to analyze and discover its performance. In one study of social media web applications that use OAuth2, the authors[] claim there are many malicious websites and vulnerabilities, such as cross-site request forgery (CSRF) and open redirectors.

Another study concluded that OAuth2 is vulnerable to application impersonation attacks due to having several flows and token types. They also concluded that this vulnerability can lead to large-scale exploits and privacy leaks[].

In another study of OAuth2, the results are that a request is not guaranteed to be confidential. A brute force attack against the server can lead to a loss of confidentiality. Also, there is a lack of server trust. OAuth2 is concerned with authenticating the client, but not the server[].

There are many ongoing studies of OAuth2. One software architect, Sergey Beryozkin, believes that OAuth2 will continue to be deployed and that it will become a “big thing”, because it is well-suited to the Cloud and to Big Data.

4. AUTHORIZATION USING OAUTH2

OAuth2 is a framework for delegating access authorization.

A. *Roles*

There are four roles in OAuth2. They are listed below.

- Resource Owner: The Resource owner.
- Client: The Client.
- Resource Server: The Resource Server.
- Authorization Server: The Authorization Server.

B. *Tokens*

There are two kinds of tokens used in OAuth2. They are the authorization token and the access token.

C. *Flows*

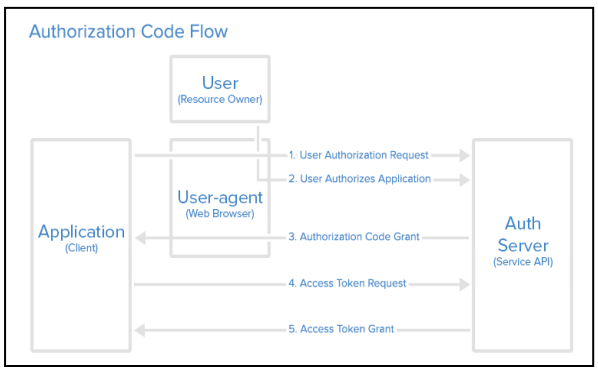
Flows are how the different roles interact in requesting authorization. Sometimes flows are referred to by the type of authorization granted. OAuth2.0 offers four flows: Authorization Code, Implicit, Resource Owner Password Credentials and Client Credentials. The flow used depends on the authorization grant type and by the types of grants that the API supports. The flows are described below.

- Authorization Code: This flow is used with server-side applications.
- Implicit: This flow is used with mobile applications or with web applications.
- Resource Owner Password Credentials: This flow is used by trusted applications. A trusted application may be owned by the service.
- Client Credentials: This flow is used application APIs.

The flows are depicted below.

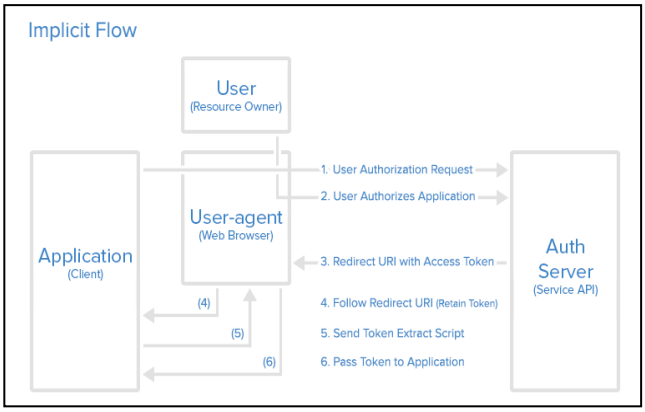
Authorization Code

This is the most commonly used flow because it works with the source code on the server side. Also, client confidentiality can be maintained. This flow uses redirection, so the application must interact with the user agent.



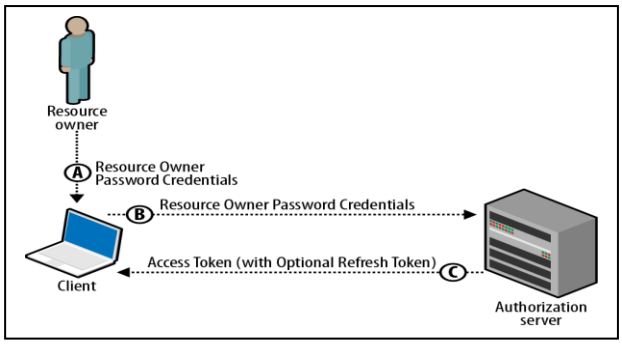
Implicit

This flow is used with mobile applications and with applications that run in a web browser. Client confidentiality is guaranteed. This flow also uses redirection. In this flow, the token is given to the user-agent to forward to the application. This flow does not authenticate the application. The redirect URI must perform this action.



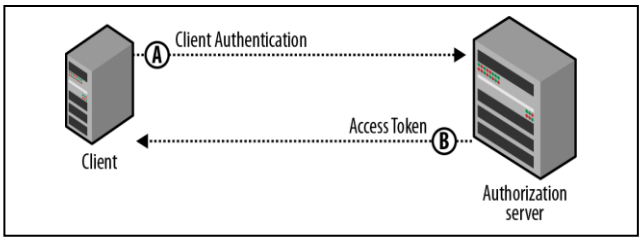
Resource Owner Password Credentials

In this flow, the user provides his username and password to the application. The application uses this information to get an access token from the service. This flow should only be used if no other flows are offered by the API and if the user trusts the application.



Client Credentials

This flow allows an application to access its own service.



5. OAUTH2 SECURITY PROBLEMS

One important thing to keep in mind is Oauth inherently doesn't guarantee privacy/security of data. The security bugs generally originate because of implementation issues. Hence most security risk aren't in the protocol, but in the implementation[9].

5.1) Client side attacks:

Most of the client side attacks are targeted on stealing access token from the end user.

Here, we assume that attackers control the user's browser or has the capability to force (or lure) user to visit a URL.

1) Phishing: If the attacker has the knowledge of client id (which is very easy to obtain) and redirect URI, resource owner can be redirected to the attacker's application instead of legitimate application [6]

2) Client impersonation: Attacker grabs the session (or authorization code) using callback endpoint (combining other flaws like** URL redirection, click jacking, exploiting cross domain referrer leakage[12] etc.) and then drops the original request to prevent authorization code from being stale.* He then establishes session with client using the victims' authorization code.

**applications strictly following RFC 6749 were found to be vulnerable to URL redirection thus possibly leading to OAUTH token leak.[11]

5.1.1) Case Study-client side attacks:

1) Hotmail CSRF vulnerability: In September 2015, Researchers from Synack found a CSRF vulnerability[15] in OAuth 2.0 implementation. In CSRF vulnerability, the attacker has the user's browser and he performs action on behalf of the user.

The problem was that Hotmail's OAuth implementation didn't validate CSRF tokens in the server side. This let anyone send request to read/write in the resources of hotmail on behalf of victim user. Although, in the client side a nonce token was being sent to the server for every new request, server wasn't

validating them and removing the entire nonce value would still perform the intended action, which could easily have been turned into a worm to steal tokens from hotmail users.

* Most authorization servers donot simply expire access token in their implementation, which means even if attacker didn't drop the packet, he would still be able to use it.

5.2) Server side flaws and attacks:

In this section we will discuss the problems in this protocol that are related to the server side implementation of OAuth 2.0.

First, there exists no inherent signing process to ensure security of client application before allowing them to ask for authorization token.

Second type of problem is caused by implementation of OAuth 2.0 where an attacker could escalate privilege to perform actions that are outside of the scope of authorization token

5.2.1) Case Study- server side attacks:

1)Missing authorization checks in/Facebook: After Facebook released its' graph API, there have been more than 100 security related bugs that were originated due to missing authorization checks [14]. The attacker in this scenario was resource owner instead of an attacker unrelated to a system.(like in client side attacks). For an instance, if a user only allowed the resource owner to read his name from authorization server, he could read other personal details like his email address, physical location etc.

6. RECOMMENDATIONS

We will discuss ways of defending attacks on OAuth 2.0 implementations in both client and server side.

6.1)Client side attacks mitigation techniques:

Solution for phishing related issues: Whitelist a set of protected URI's that can be used as field in redirect URI parameter.

Different measures have to be implemented to prevent client impersonation, some of which include strict implementation of SSL/TLS in pages that exchanges authorization token. Ensuring CSRF tokens issued to the client are properly validated plays and important role towards protecting users from impersonation attacks.

The X-frame-options header should always be set to 'sameorigin' to prevent probability of victim being tricked into performing actions that the user didn't intend.

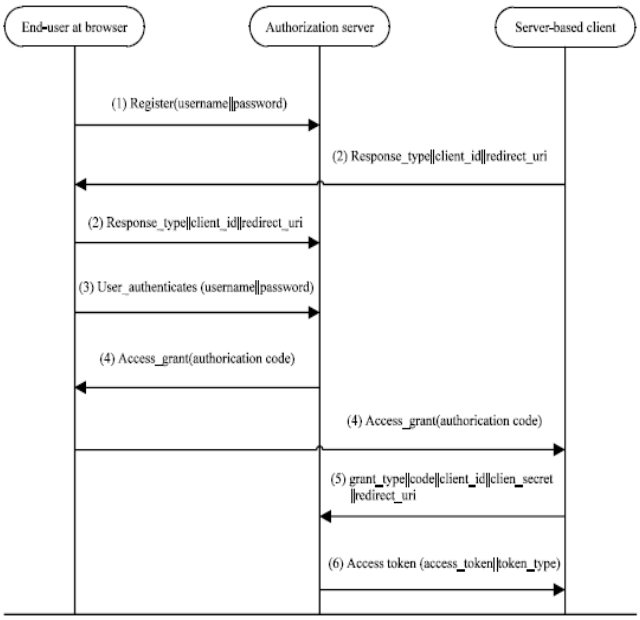


fig: secure implementation of OAuth 2.0 for authorization with.client_secret

2)Server Side attacks mitigation technique:

Since implementations can have different approaches, server side authorization flaws don't have any known solutions except exhaustively testing each authorization scopes.

Other than that, now we have access tokens that don't expire. This is still secure, provided that along with access token, appsecret proof is also sent from the resource server to authorization server. [8] This will ensure that even if attacker steals the access token, he doesn't have the appsecret token, which is held by resource server. Even if attacker gets hold of resource server, app secret token can be changed according to necessity of resource server.

7. SUMMARY

Properly implemented web services using the REST API along with OAuth2.0 can provide secure access to user resources. Services can be designed and implemented which provide only the level of access a user explicitly allows while protecting the remainder of the user's resources from access.

8. CONCLUSION

In this paper, we discuss the necessity of web services in web application and authorization procedures currently being implemented. OAuth 2.0 is the most common way of authorizing a user in REST API implementation. This allowed us to login to several applications without having to remember password for them. However, it also introduced risk of authorization token being stolen from client/server side. We present details and case study of current threats in OAuth 2.0 implementation and ways to mitigate those threats. We divide the attacks into client side and server side based on who was being targeted. Our survey shows that most of attacks are possible due to improper implementation of RFC 6749. We present several issues that are being found in popular websites so that they could be eliminated before deploying OAuth for authorization in REST API.

REFERENCES

[1] Mou'ath Hourani, Qusai Shambour, Ahmad Al-Zubidy and Ali Al-Smadi, "Proposed design and implementation for restful web server", Journal of Software, vol 9, No 5, May 2014.

[2] Mohamed Ibrahim B, Mohamed Shanavas A R, "Applying security for restful web services – limitations and delimitations", IJETAE, ISSN 2250-2459, ISO 9001:2008 Certified Journal, vol 4, Issue 9, Sept 2014.

[3] Muhammad Imran Hussain and Naveed Dilber, "Restful web services security by using asp.net web api mvc based", Journal of Independent Studies and Research – Computing, vol 12, Issue 1, Jan 2014, pp 4-10.

[4] Khash Kiani, "Four attacks on oauth – how to secure you oauth implementation," Sans, vol. 3.1, 2015.

[5] Chetan Bansal, Karthikeyan Bhargavan, Sergio Maffeis, "Discovering concrete attacks on website authorization by formal analysis," IEEE, pp. 247–262, 2012 [25th Computer Security Symposium].

[6] Pili Hu, Ronghai Yang, Yue Li, and Wing Cheong Lau, "Application impersonation: problems of oauth and api design in online social neworks", Association for Computing Machinery, 2014.

[7] D. Hardt, Ed., Microsoft, The OAuth 2.0 Authorization Framework http://tools.ietf.org/pdf/rfc6749.pdf Oct 2012.

[8] T. Lodderstedt, Ed. Deutsche Telekom AG, M. McGloin, IBM, P. Hunt, Oracle Corporation https://tools.ietf.org/pdf/rfc6819.pdf Jan 2013.

[9] Four Attacks on OAuth - How to Secure Your OAuth Implementation, K. Khash
https://www.sans.org/reading-room/whitepapers/application/attacks-oauth-secure-oauth-implementation-33644

[10] Yang, F., & Manoharan, S. (2013). A security analysis of the OAuth protocol. Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference, 271-276.

[11]https://hackerone.com/reports/26962

[12]http://oauthsecurity.com

[13] Chari, S., C. Jutla and A. Roy, 2011. Universally composable security Analysis of OAuth v2.0. http: //eprint. iacr. org/2011/526. Pdf

[14]http://philippeharewood.com/

[15] https://www.synack.com/2015/10/08/how-i-hacked-hotmail/