Analyzing of secure authorization technique of Webservice using Authorization tool : OAuth 2.0

SAHIL[1], AKASH SHARMA[2]
[1] UG Scholar, Chandigarh College of Engineering and Technology, Chandigarh
[2]UG Scholar, Chandigarh College of Engineering and Technology, Chandigarh

ABSTRACT

A website or application can access resources maintained by other web apps on behalf of a user thanks to the OAuth 2.0 standard. Open Authorization 2.0 is referred to as OAuth 2.0. The most well-known and secure authorization technique currently in use. This article focus on the work through all the difficulties experienced when building an oauth2.0 system, and explain and comprehend the complicated workings of each phase. For an completely secure system, discuss the various exploitation techniques and its key features.

KEYWORDS OAuth 2.0; Authorization; HTTP request

I. INTRODUCTION

It is a popular authorization technique that many online apps employ. It may sound complicated, but it allows a first-party website to access a restricted and chosen quantity of user data through a second-party website without divulging its password to the first-party website. It can be used for IoT and its uses in security surveillance [1]. However, let's move on to an example. Assume a person wants to register on the website (http://www.1stparty.com), which requires them to enter some basic information like their first and last names, email addresses, and so on [2][3]. But instead of doing all that work, this first-party website can retrieve the user's information that they have previously given to websites like Facebook and Google. The primary and most crucial step in all of these procedures is for this first-party website to receives access to only fundamental knowledge not any sensitive data, such as passwords and other

During the Process there Are Number of Interactions Between three different points named

a. Client application: -The website which wants the user data
b. OAuth 2.0 service provider: - The website which access to the data of users
c. Resource owner [4]: -The user whose data is data is going to be shared. With his/her permission it would never be possible

OAuth can now be applied in a variety of ways (via different grant types). However, "authorization codes" and "implicit" grant types are the only two methods we typically employ. Although there are some differences, most of the stages in these grant kinds are the same [5].

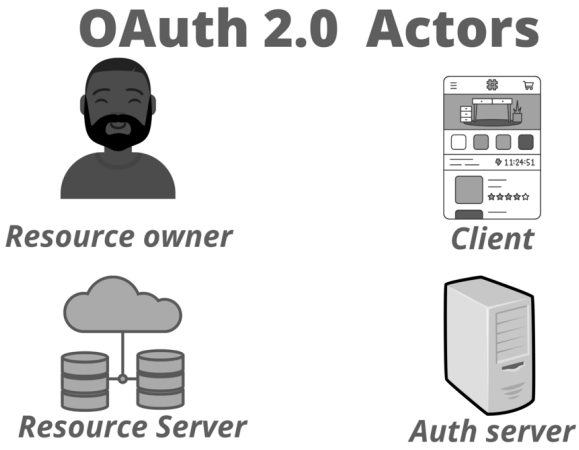

FIGURE 1: Roles and Actors in OAuth 2.O

II. PREVIOUS WORK

In order to encourage the reuse of user identities for various, unique services without the necessity for password exchanges, OAuth was first introduced in December 2006. AOL launched the OpenAuth framework in April 2007, which was then adopted by Google's OpenAuth group and updated in October 2012 to address newly discovered security flaws. It takes into account a client-server authentication approach in which the client asks the server for access to limited resources after logging in with her credentials on the client

side. The OAuth framework makes use of an authorization server to distribute securitytokens to various users that ask to access resources that are protected. Specific OAuth features were extracted and analysed using exploratory data analytics (EDA) techniques so that each output class could be assessed to determine the impact of the identified OAuth features.

OAuth 2.0 is also used to offer authentication when users log themselves into a web service using their identity handled by a third-party service. This is known as a single sign on (SSO) framework, in which the user needs to authorise websites to act on her behalf. Mobile applications frequently employ the authentication process to connect to the application server's back end. The resource owner (also known as the relying party), who is the end user or a host acting on her behalf and has the ability to request access to protected resources, is represented by the client, which is the application used by the user when requesting access to the protected data or service. We investigate the bit-level parallelism's side-channel security in the GPU-based bitsliced AES implementation. To locate special leakage, a non-profiled leakage detection approach is used. The side-channel security of the implementation is examined using patterns on multiple bits, followed by the multi-bits feature-level fusion attack (MBFFA) and multi-bits decision-level fusion attack (MBDFA).

The resource owner is also represented by the authorization server, which is the issuer of access tokens and the one that ensures the authenticity of the owner. The Resource Owner (RO) is the principal actor who grants access to the IdP resources from the client, and the Resource Server and Authorization Server are typically the same host and are also known as identity provider (IdP) servers.Any client application must register with the identity provider in order to communicate with them, and during the authentication process, credentials (a public client ID and a client secret) are generated. shows a high-level schematic of the authentication schema, which can be summed up as follows:

1. The client asks for access to the resource owner's data.
2. In response, the resource owner sends the authentication grant. The reply includes the selected grant type, the preferred authentication server to use, and which server hosts the resources, among other things.
3. The client requests an access token from the authentication server by sending the authentication grant to the server.
4. After the permission has been verified, the authentication server sends back an access token that identifies the resource owner and notifies the user that the authorization has been granted.
5. The client submits the access token to the server hosting the data and requests permission to access the protected resources.
6. After validating the token and returning the requested data, the resource server receives it.

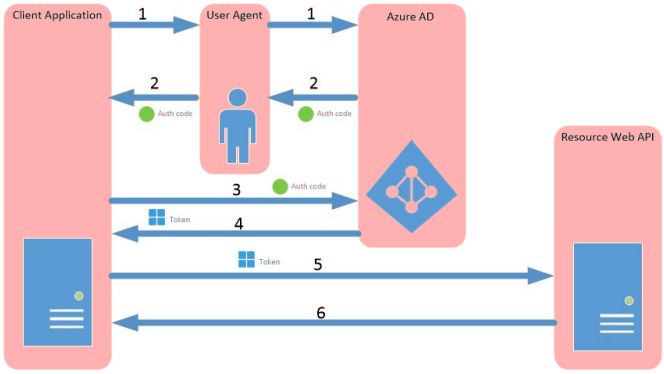

FIGURE 2: Control Flow of OAuth 2.O

III. DIFFERENT GRANT TYPES

Let's first look at the different grant categories. The grant type can be compared to the OAuth Blueprint. It controls when and how the OAuth procedure will proceed, which is why it is also known as "OAuth flows." Let's examine various grant types right now.

I. Authorization code grant type: -

a. Authorization request: The client application first asks the OAuth server for data related to a specific user [6]. The HTTP request for the next action is formatted as follows:

```
GET    /callback?code=a1b2c3d4e5f6g7h8&state=ae13d489bd00e3c24
HTTP/1.1
Host: client-app.com
```

b. User Login and consent: - The user will see a window in their browser asking them to confirm their intention to log in to the website using OAuth when the authorization server sends the first request...

c. Authorization code grant: - The browser will automatically reroute to the authorization request's redirect uri as soon as the user grants permission. And the authorisation code will be included in the ensuing get request.

d. Access token request: - The client application will exchange the authorization code it has received for an access token via the OAuth server's /token endpoint. The OAuth Service will check the access token request to see if it is legitimate. The access code will then be sent to the client server.

e. API call: Now that it has access to the access token, the client application can now access it. To obtain the user information, submit to the /userinfoend point.

f. Resource Grant: The user information end point will verify and confirm that the token is authentic and not the product of a hacker when it receives it. As a result, the OAuth server will send the necessary data.

```
{
 "access_token":"z0y9x8w7v6u5",
 "token_type":"Bearer",
 "expires_in":3600,
 "scope":"openidprofile",
 …
}
{
 "username":"Carlos",
 "email":"carlos@carlos-montoya.net",
 …
}
GET    /userinfo  HTTP/1.1
Host:   oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

II Implicit grant type: - This grant type is essentially identical to "authorization code," but when the user approves the OAuth prompt, the access token is immediately granted to the client server, unlike "authorization code," where access code is granted before authorization code [7].

IV. EXPLOITING OAUTH 2.0

Now as we have learned about the basic concepts of OAuth. Let's finally see about exploitation techniques [8]:

a) **Open redirect:** An attacker may be able to steal the authorization tokens linked to other users' accounts if the OAuth provider does not correctly verify redirect uri. The code or access tokens may be forwarded to the website controlled by the attacker and then utilised to complete the flow [9]. To counteract this attack, client applications typically include a whitelist of their true call back URIs when registering with the OAuth service. However, there are still a variety of ways to go around this validation.

b) **Flawed scope validation:** A list of the data that the client application wishes to access is displayed to the user each time they log in to the authorization server (Like Email, profile picture). The user's information is requested and sent using secure server-to- server communication when using the authorisation code grant type. which the attacker finds nearly impossible to control. Attackers may nonetheless register their own client software with the OAuth service. The access token is transmitted through the browser for the implicit grant type. Additionally, an attacker can issue a regular browser-based request to an oAuth API while manually adding a new scope parameter in order to steal the tokens and utilise them.

c) **Improper implementation of the implicit grant type [10]:** Due to the risks associated with providing access tokens through the browser, the implicit grant type is often recommended for single-page web applications. However, due to its relative simplicity, it is also commonly used in traditional client and server web applications. During this process, the OAuth service sends a URL fragment containing an access token to the client application. The client application then uses JavaScript to access the token. The problem is that the application must store the current user data (usually the user ID and access token) somewhere in order to maintain the session when the user leaves the page. To solve this problem, client applications often send this data via POST to the server. This behavior can lead to serious vulnerabilities. In this case, an attacker can impersonate any user by changing the request parameters sent to the server. If you make a request after receiving the session cookie, you are effectively logged in. This request resembles a form submission request that might be made in the context of a typical password-based login. However, in this case the server is implicitly trusted as there is no secret or password corresponding to the data provided. An attacker can access this her POST request through a browser in an implicit flow. Against this background, if the client her application does not fully validate that the access token matches the other information in the token,

V. STRENGTHENING AND SECURING THE WHOLE PROCESS

OAuth2, a well-liked authorization framework that enables apps to safeguard resources from unwanted access, is frequently paired with OpenID-Connect (OIDC). It transfers control of user authentication to an authorization service, which then gives the user's consent for other apps to access the restricted resources. Web and mobile applications can use OAuth2's authorization flows.

The authorization code grant flow, which is used to provide confidential clients access to protected resources, is the most often used OAuth2 flow. Clients identify themselves to the authorization service using a client id, and they authenticate themselves using a client secret.

A. OAUTH2 AUTHORIZATION CODE GRANT FLOW

The frontend and backend flows of the authorization code grant are well separated. A user agent, often the system browser, is given control of the frontend flow. It checks the user's credentials and requests authorization permissions from them so that the client can

access protected resources. When successful, the client receives an authorization code. In the backend flow, the client trades the authorization code for access and refresh tokens after being verified using a client secret. In order to access secured backend resources on behalf of the user, the client uses an access token.

B. MOBILE OAUTH2 CODE GRANT

The client secret is only revealed to the authorization server during the code grant sequence. It is never made available through the frontend user agent, which can be less secure.

A confidential client in OAuth2 is one that has the ability to securely guard client secrets. Unfortunately, native apps are not seen as private clients but rather as public ones. They cannot safeguard static secrets. Secrets are more challenging to steal, but not impossible, thanks to obfuscation and code hardening approaches. Anyone could complete an authorization code exchange if the client secret could be stolen.

Many identification and authorization service providers simply drop the client secret because a public client secret is no client secret at all. AppAuth, a well-known open source OAuth2 SDK, advises against using client secrets especially for Android and iOS during using the cloud services [11].

C. MAKING USE OF CLIENT SECRETS (DANGEROUS)

When possible, we highly advise against utilising static client secrets in native applications. Dynamic client registration-derived client secrets are secure to employ, but static client secrets can be easily collected from your apps and give third parties access to your app's user data. If the OAuth2 service you are integrating with has to employ client secrets, we strongly advise carrying out the code exchange step on your backend so that the client secret can be kept secret.

Public clients are vulnerable to a variety of threats, including as client impersonation by malicious software and theft of authorisation codes and tokens. To restore the integrity of the OAuth2 code, either with a weak secret or none at all. Grant flow must increase from public to confidential client strength for native mobile app protection.

D. PROOF KEY FOR CODE EXCHANGE (PKCE)

Anyone who can see a frontend authorization code on a public client utilising the basic code grant flow can attempt to convert it into access and refresh tokens. To counteract this flaw, Proof Key for Code Exchange (PKCE) was incorporated into the fundamental flow. It tries to guarantee that the client who requests the frontend code exchange is the same client who requests the backend code exchange later.

The code verifier runtime secret is initially created by the client. The client hashes this secret and transmits this code challenge value as part of the frontend request in the stringer form of PKCE. This value is retained by the authorisation server. The client incorporates the code verifier into its ensuing backend code, client secret or not. Request for exchange. The code verifier's hash is compared to the code challenge it originally got via the authorization server. The service will handle the code exchange request as usual if they match. A malicious actor that steals the authorization code cannot swap the codes effectively with PKCE unless they are aware of the original code verifier. The runtime generated secret for the code verifier. The code verifier can be deemed private on the mobile client because it is transient and does not need to be stored on the client. It would be necessary to create a fake code verifier and insert the accompanying false code challenge hash into the client's initial frontend request in order to attack PKCE. The malicious actor can then submit its false code verifier to complete the exchange after examining the returning authorisation code.

These attacks can be avoided using methods like SSL/TLS and certificate pinning, but they do not stop repackaged apps from imitating the original app and its functionality.

VI. CONCLUSION:

This article discussed oAuth2.0, including what steps must be taken to create a system of this kind, how to set up grant types, and how tokens are transferred between servers. We talked about a few well-known vulnerabilities and how they can be used to undermine authentication. Finally, we can say that OAuth2.0 is a well-designed authorization mechanism, but if it is utilised carelessly or indiscriminately, it may lead to major security and data breach concerns.

REFERENCES

[1] Soumya Sharma, Sunil K. Singh (2022), IoT and its uses in Security surveillance, Insights2Techinfo, pp.1

[2] Argyriou, Marios&Dragoni, Nicola &Spognardi, Angelo. (2017). Security Flows in OAuth 2.0 Framework: A Case Study. 396-406. DOI: 10.1007/978-3-319-66284-8_33.

[3] Sucasas, V., Mantas, G., Radwan, A., & Rodriguez, J. (2016, May). An OAuth2-based protocol with strong user privacy preservation for smart city mobile e-Health apps. In 2016 IEEE International Conference on Communications (ICC) (pp. 1-6). IEEE.

[4] C Diwan, Sunil K Singh. An approach to revamp the data security using cryptographic techniques. International Journal of Advanced Research in Computer Science. Jul/Aug2017, Vol. 8 Issue 7, p476-479. 4p.

[5] Siddharth Gupta, Sunil Kumar Singh. Authenticating and Securing Mobile Applications Using Microlog. International Conference on Computer Science and Information Technology pp(258-267).

[6] Sunil Kr Singh, Rajni Jindal. Performance Enhancement of TCP over an Mobile Ad- Hoc National Conference on Energy, Communication and Computer (NC ECC 06),MAIT, New Delhi, India pp(242-251).

[7] Ferry, Eugene & O'Raw, John & Curran, Kevin. (2015). Security evaluation of the OAuth 2.0 framework. Information and Computer Security. 23. 73-101. DOI: 10.1108/ICS-12-2013-0089.

[8] Chatterjee, Ayan, et al. "SFTSDH: Applying Spring Security Framework With TSD- Based OAuth2 to Protect Microservice Architecture APIs." IEEE Access 10 (2022): 41914- 41934.

[9] Yang, Ronghai, Wing Cheong Lau, and Tianyu Liu. "Signing into one billion mobile app accounts effortlessly with oauth2. 0." Black Hat Europe (2016).

[10] Ferretti, Luca, Mirco Marchetti, and Michele Colajanni. "Verifiable delegated authorization for user-centric architectures and an OAuth2 implementation." 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). Vol. 2. IEEE, 2017.

[11] Peñalvo, F. J., Sharma, A., Chhabra, A., Singh, S. K., Kumar, S., Arya, V., & Gaurav, A. (2022). Mobile Cloud Computing and Sustainable Development: Opportunities, Challenges, and Future Directions. International Journal of Cloud Applications and Computing (IJCAC), 12(1), 1-20. http://doi.org/10.4018/IJCAC.312583