Journal of Theoretical and Applied Information Technology
15th May 2021. Vol.99. No 9

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

FRAMEWORK TO SECURE THE OAUTH 2.0 AND JSON WEB TOKEN FOR REST API

[1]EHAB RUSHDY, [2]WALID KHEDR, [3]NIHAL SALAH

[1]Associate Professor of Information Technology, Department of Information Technology, Faculty of Computers & Informatics, Zagazig University, Zagazig, Egypt

[2]Professor of Information Technology, Department of Information Technology, Faculty of Computers & Informatics, Zagazig University, Zagazig, Egypt

[3]Post Graduate, Department of Information Technology, Faculty of Computers & Informatics, Zagazig University, Zagazig, Egypt

E-mail: [1]ehab.rushdy@gmail.com, [2]khedrw@yahoo.com, [3]nihal.radwan@hotmail.com

ABSTRACT

The Open Authorization (OAuth 2.0) specification defines a delegation protocol that is helpful for conveying authorization decisions (via a token) across a network of web applications and APIs. OAuth 2.0 achieves this with the help of valid tokens issued by an authorization server which enables access to protected resources. For Achieving statelessness, access tokens in OAuth needs that resource server request to authorization server to get the user details related to the token and whether the token is valid or not for it each client request. OAuth 2.0 allows safe transfer of bearer tokens, but doesn't validate the processing party. If a token endpoint is wrong then entire system security is in danger. JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for transmitting data between various parties by encoding them as JSON objects which can be digitally signed or encrypted. This research proposes an approach that combines OAuth 2.0 with JWT in REST API to allow the resource server to validate the OAuth access token locally and only needs an interaction with the Authorization server to get a new OAuth access token. This combination reduces the interaction with the Auth server that yields the performance improvements. JWT has several options for using algorithms namely a symmetric algorithm HS256 (HMAC with SHA-256) and an asymmetric algorithm RS256 (RSA Signature with SHA-256).
Testing the most different cryptographic algorithms used to construct JWT for signing token are done based on the speed of generating tokens, the size of tokens, time data transfer tokens and security of tokens against attacks. The experimental results show the use of The HS256 has good result compare to using RS256 for framework of evaluation the performance of proposed algorithms in terms of generating tokens, the size of tokens and time data transfer tokens. In order to prove that proposed scheme can avoid phishing attacks especially user credentials by taking two-factor authentication service which makes user can guarantee the authenticity of client.

Keywords: *Authentication, Authorization, OAuth 2.0, JWT, Security*

1. INTRODUCTION

Application programming interfaces (APIs) [1]created using the Representational State Transfer (REST) protocol have become a nearly universal standard today for connecting mobile apps and websites with servers and third-party systems that the service providers expose several of services as web accessible APIs for users to build applications or consume services. One of the key aspects of designing an API platform is controlling who has access to data.
An API platform should be capable of permitting completely different levels of user access ways wherever authorized users could get unlimited access to secure APIs while non-authorized users will only access APIs that are public. From the API platform perspective, granting access to completely different levels of users should be created as easy as possible.

As a result, many organizations provide different authorization strategies to access their APIs on behalf of the user. This has created a major requirement for a common global standard for securing APIs. This is where OAuth2.0 stands out among other standards.

Every API request from the client to a server contain all the necessary information necessary to serve the request. The server maintains neither state nor context. To create a common API security model, suppose all endpoints require an OAuth 2.0 Access Token issued from a common identity provider and have the appropriate API security token checks in place and authorize access to protected REST APIs that OAuth 2.0 and JSON Web Token are two of the most widely used token frameworks or standards for authorizing access to APIs. OAuth2.0 and JWT can be combined to gain performance improvements and describing the most different cryptographic algorithms used to construct JWT for signing by comparison application of RS256 and HS256 algorithms and prevent the security issues attacks of JWT.

1.1 The Oauth2.0 Protocol:

OAuth2.0 [2] is an open standard for authorization and provides a method for clients to access server resources on behalf of a resource owner, such as a different client or an end-user.
It also provides a way for applications to gain limited access to a user's protected resources without the need for the user to disclose their login credentials to the application.
In OAuth2.0, the client requests access to resources controlled by the resource owner and hosted by the resource server by giving a different set of credentials (not the resource owner's credentials) to access the resource. In OAuth2.0, the client will obtain an access token from the authorization server, which can be used to access server resources on behalf of the resource owner. There are four types of roles specified in OAuth2.0 [3] illustrated in Figure 1 describes the interaction between these roles:

1.1.1 Client

Also known as "the app". It can be an application running on a mobile device or a web app. It is the application requesting access to protected resources keep on the resource server. It additionally obtains authorization from the resource owner.

1.1.2 Resource owner:

It's referred to as a person, it is called an end-user who is capable of authorizing access to a protected resource or a service.

1.1.3 Resource server:

Data owned by the resource owner such like sensitive data and it can accept and respond to protected resource requests.

1.1.4 Authorization server:

This server supplies access tokens to the client and it is responsible for validating (authorization grants) and issuing the (access tokens) that give the client application access to the end user 's data on the resource server.

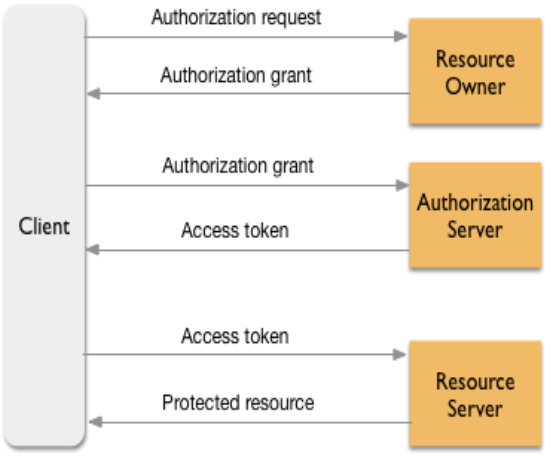

Figure 1: The OAuth 2.0 flow

1.1.5 OAuth 2.0 grant types:

An authorization grant type [4] is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. In the abstract protocol flow above, the first four steps cover getting an authorization grant and access token. OAuth defines four grant types, each of which is useful in different cases can be seen in Table 1.

Table 1: OAuth 2.0 Grant Types

Grant type	Used for
Client Credentials	When two machines need to talk to each other, e.g., two APIs
Authorization Code	This is the flow that occurs when login to a service using Facebook, Google, GitHub etc.
Implicit Grant	Like the Authorization Code Grant Type, but user-based.
Password Grant	The user's username and password are exchanged directly for the OAuth2 tokens. which is more suitable for server apps used by a single user account. This is by far the easiest authentication scheme to implement that this paper focuses on this type.

1.1.6 Access token

An access token [5] is a long sequence of characters that contains security credentials used to access protected resources or services and to provide authorization for API requests. Access tokens (also referred to as bearer tokens) are passed in authorization headers. It's represented credentials like username and password. Access tokens generally have a lifetime determined by server and once expired; a replacement token must be generated. Some grant types enable the authorization server to issue a refresh token that allows the application to get a new access token once the old one expires.

1.2 JSON Web Token

JSON Web Token (JWT) [6],[7] is an open standard that defines a compact and self-contained way for securely transmitting information between parties. It is an authentication protocol for allow encoded claims (tokens) to be transferred between two parties (client and server) and the token is issued upon the identification of a client.

JWT tokens are based on JSON and used in new authentication and authorization protocols in OAuth 2.0 because of their small size, JWT tokens can be sent via URL, HTTP POST parameters or inside the Header HTTP, and also because of its small size it can transmitted faster. Called independent information because the contents of the generated token have information from users needed, so no need to query to database more than once. The token can be verified and trusted because it has been digitally signed. JWT tokens can be signed using a secret key (HMAC algorithm) or public / private key pair (RSA algorithm). However, API only use the JWT concept which can be called "jot". JWT doesn't depend on the specific program language.

1.2.1 Structure of JWT

Every JWT is generated with the same structure[8]. There are three parts: The header, the payload and the signature, separated by period character (.). Each section (except signature) is comprised of base64url-encoded JSON containing specific information for that token as shown in Figure 2.

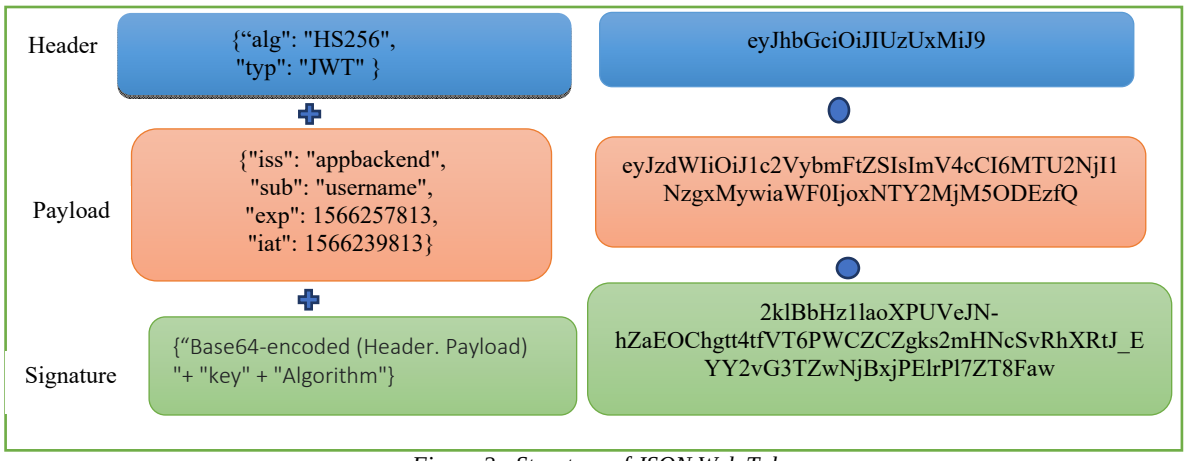

Figure 2: Structure of JSON Web Token

1.2.2 Common JWT signing algorithms

There are two main ways to sign JWTs cryptographically: using symmetric or asymmetric keys. Both types are commonly used, and provide the same guarantees of authenticity. Most JWTs in the wild are just signed. The most common algorithms are HMAC SHA-256 symmetric algorithm and RSA-256 an asymmetric algorithm can be seen in Table 2.

Table 2: Compression Between SHA256 and RSA256[9]

	SHA256	RSA256
Stand For	HMAC with SHA-256	RSA encryption plus SHA-256 hashing
Key type	Symmetrical secret key	Asymmetrical Public / private key pair RSA
Use cases	That's an algorithm which encrypts and hashes the message (a JSON data) at the same time using symmetrical secret key. The same key is used for encryption and decryption of the message.	RSA is an asymmetric encryption algorithm, which means it operates on a pair of keys – public and private. Private key is used to encrypt a token, and public one – to decipher it.

1.3 Two-Factor Authentication (2FA)

Two-factor authentication (2FA)[10] is a second layer of security to protect an account or system. Users must go through two layers of security before being granted access to an account or system. 2FA increases the safety of online accounts by requiring two types of information from the user, such as a password or PIN, an email account, an ATM card or fingerprint, before the user can log in. The first factor is the password; the second factor is the additional item.

2. RELATED WORK

Various performed researches and manufactured models have been proposed for Security Analysis of OAuth 2.0. Several works have been done regarding the OAuth security and its application in constrained environments.

The work [11] explained scenarios of OAuth 2.0 Protocol with example for establishing identity management standards across services, provides an alternative to sharing our usernames and passwords, and exposing ourselves to attacks on our online data and identities.

The work [12] focuses on security vulnerabilities of the OAuth 2.0 protocol that proposed an attacker model to perform systematic analysis of the root causes of common attacks like replay attacks, impersonation attacks and forced login CSRF attacks.

The work [13] this work provided an additional layer of protection against CSRF attacks for OAuth 2.0 services and proposed a new practical technique used to mitigate CSRF attacks against both OAuth 2.0 and OpenID Connect.

The work [14] proposed a scheme based on OAuth 2.0 and JSON Web Token for securing existing health care services in the IOT cloud platform.

The work [15] provided the stateless and compact feature of JWT for authentication and access authorizations for cloud users.

3. PROBLEM STATEMENT

OAuth2.0 provides Resource Owner Password Credentials Grant is used when the application exchanges the user's username and password for an access token. This grant type is appropriate for clients capable of getting the resource owner's credentials (username and password, typically using an interactive form) and eliminates the need for the OAuth2 client to store the resource owner's credentials for future use. Workflow [16] of Resource Owner Password Credentials Grant as shown Figure 3 :

1. Resource Owner/User launches the Client App to access the its own protected resource in the Resource Server.
2. The Client App shows login form for the user to enter their credentials into the App.
3. The user provides the client application its credentials that are the resource owner's user name and password.
4. The client application requests to the authorization server by including the credentials received from the user and either the client credentials or a client assertion.

5. The authorization server validates both the client identifier and secret, determines whether or not it's authorized for making this request, and validates that the resource owner credentials and other parameters are supplied and if valid, issues an access token in the response. This is described in successful authorization ,if the request failed or is invalid for any reason, then the authorization server returns an error response.
6. Thereafter, the authorization sever sends back response with access token.
7. The OAuth client makes a REST API call to the resource server using the access token to access the protected resource.
8. The resource server receives a request and access token from the client application. After sending any data/details/resources, the resource server validates access token and if access token is not valid, the resource server returns an error.
9. Repeat steps 7 and 8 until the access token expires and the resource server sends a response to authorization server to check whether or not the access token is valid.
10. The client application receives a response from the resource server and shows protected resource to the user.
11. The end-user can see and get access the protected resources from the client application.

In addition to OAuth 2.0 also relies completely on SSL (Secure Sockets Layer)/TLS (Transport Layer Security). SSL/TLS [17] provides end-to-end security, i.e. it allows safe transfer of bearer tokens, but the resource server asks the authorization server for the user details related to the token and whether the token is valid or not on each client request. Once unauthorized authorization server is deauthorized in all resource servers because resource servers are depending on the authorization server to check validity on every request. Before OAuth2 or JWT the Resource server would need to go to the Authorization server to validate the token and get a lot of information regarding the user (resource owner). That type of bearer token can't be validated by the resource server without direct communication with an authorization server.

One way to create self-encoded tokens is to create a JSON-serialized representation of all the data that included in the token, and sign the resulting string with a key known only to server.
One potential attack against OAuth servers is a phishing attack that most people are familiar with consist of fake login pages hosted on attacker-controlled web servers. This is where an attacker makes a web page that looks identical to the service's authorization page, which typically contain username and password fields that it will be possible for attackers to steal username and password of user.

In order to avoid original password being stolen and users can be protected from phishing their username and password, so using Two-factor authentication service, the added security step that requires user enter a code sent to phone SMS or email, has traditionally worked to keep usernames and passwords safe from phishing attacks.

Journal of Theoretical and Applied Information Technology
15th May 2021. Vol.99. No 9

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

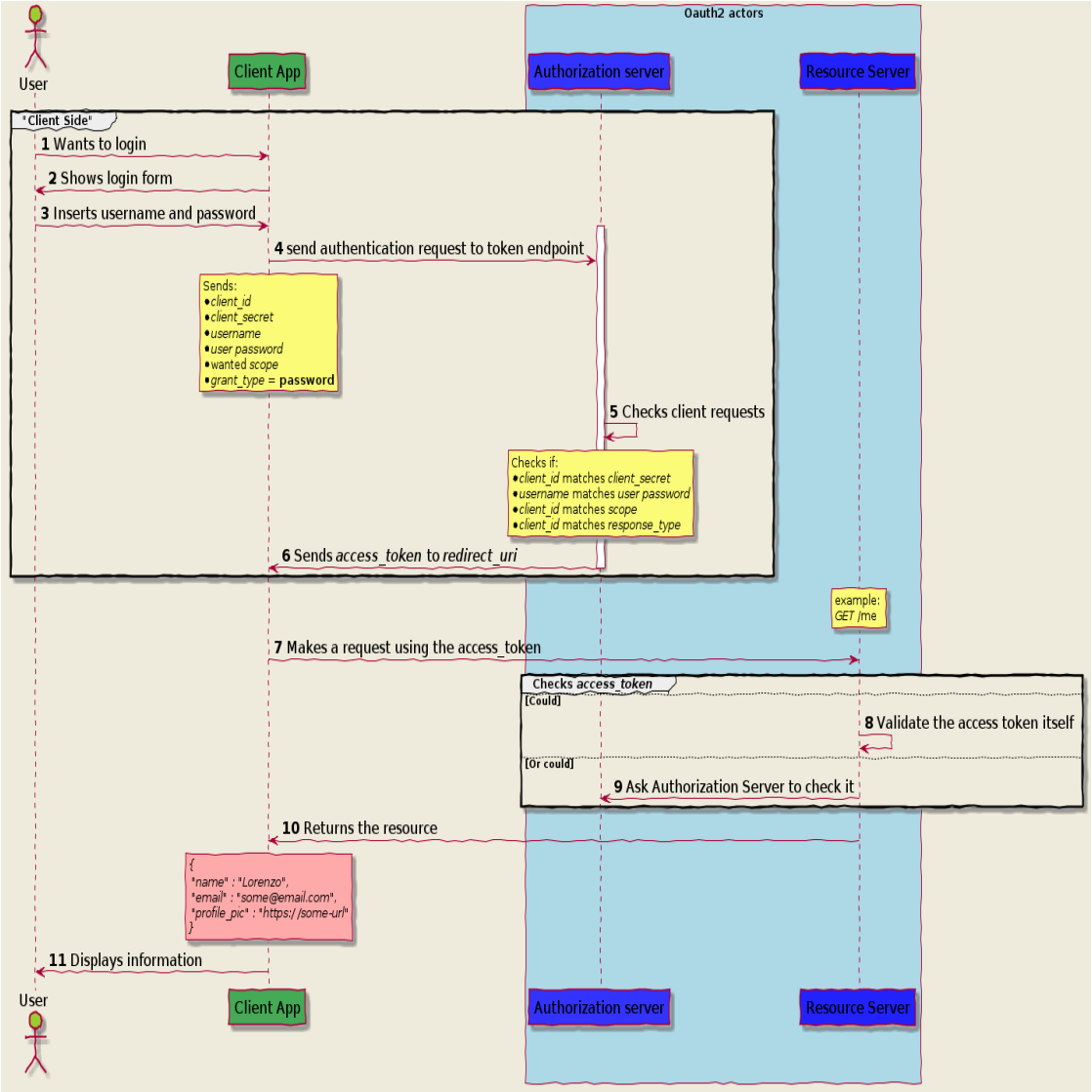

Figure 3: Workflow of Resource Owner Password Credentials

4. PROPOSED SCHEME

This section proposes scheme that combines OAuth 2.0 with JWT in REST API to ensure the integrity of the exchanged messages, the authentication of the sender and the non-repudiation as illustrated in Figure 4.
The procedure of proposed scheme is as follow:

1. Resource owner enters username and password via a client.
2. Client login by sending credentials to the authorization server.
3. The authorization server send request to two-factor authentication service.
4. The two-factor authentication send a security code by e-mail or SMS.
5. User send received code to the authorization server.
6. The authorization server will generate a JWT containing user details, permissions and data on the authentication request after successfully authenticating the client.
7. Client stores the JWT for a limited amount of time, depending on the expiration specified by authorization server then client sends the stored JWT in an Authorization header for each request to the resource server.

8. For every request, the resource server takes the JWT from the authorization header and it validates the JWT, decrypt the JWT, and parsing its contents. Based on this data only, and again without looking up further details in the database or contacting the authorization server, it can accept or reject the client request.
9. If the token is valid, the resource server allows the client access to the protected resources.

There are many types of signatures for JWT. In this research: HS256 and RS256 are used. Therefore, two proposed schemes are presented; the first scheme proposes deploying JWT using SHA-256 while RSA-256 is used with the second scheme.

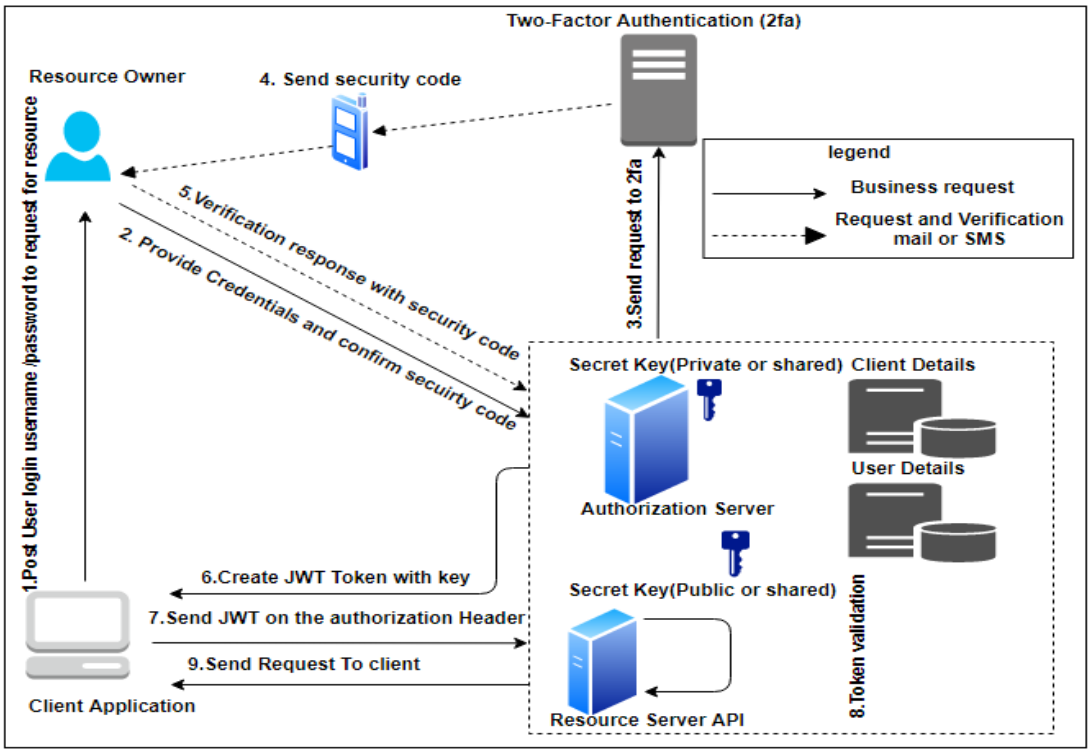

Figure 4: Combination OAuth and JWT in REST API

4.1 The HS256 JWT Digital Signature:

The HS256 digital signature is based on is cryptographic hashing function to produce a signature to take the Header, the Payload and ad also a secret key, and then hash everythin together, in Figure 6 illustrated generating an verifying JWT with SHA-256 where only th authentication server and the resource server know the secret key.

This step of proposed scheme are as follows:

1. The user sends request to the authorization server asking for authentication using credentials (username and password) that BCryptPasswordEncoder in Spring Security [18] is used for hashing algorithm to encode and hash a password and put it into a database, for perform a login authentication will hash the password and compare it with the hashed password from database.
2. The Client then sends a request to the authorization server end point of with following parameters: **The grant type** parameter should be set to password.

ClientID and client secret: This is the unique identification of the client. These two are required if the server has made client authentication mandatory that the Client App obtained during registration.

Scope: For which the Client is requesting authorization. These are space delimited list of scope string; for example – profile, email, location etc.

Username and Password: The resource owner's username and password.

3. If credentials exist in the database, the authorization server validates the user name and password and will call Two-factor authentication service to generate a security code for user.
4. The Two-factor authentication service receives the request from authorization server then it will generate a security code and send it to end user by email or SMS that code will be expire in two minutes.
5. User will get the Two-factor authentication form to enter code that can help prevent unauthorized access user' account by third party and protect against numerous other types of phishing threats.
6. User inputs received code and commits to callback URL.
7. The authorization server will verify user credentials and validate the security code retrieved previously then will generate and sign JWT with secret key that the secret key is stored on authorization server which will be received during login and then used to set the token as shown in Figure 5.

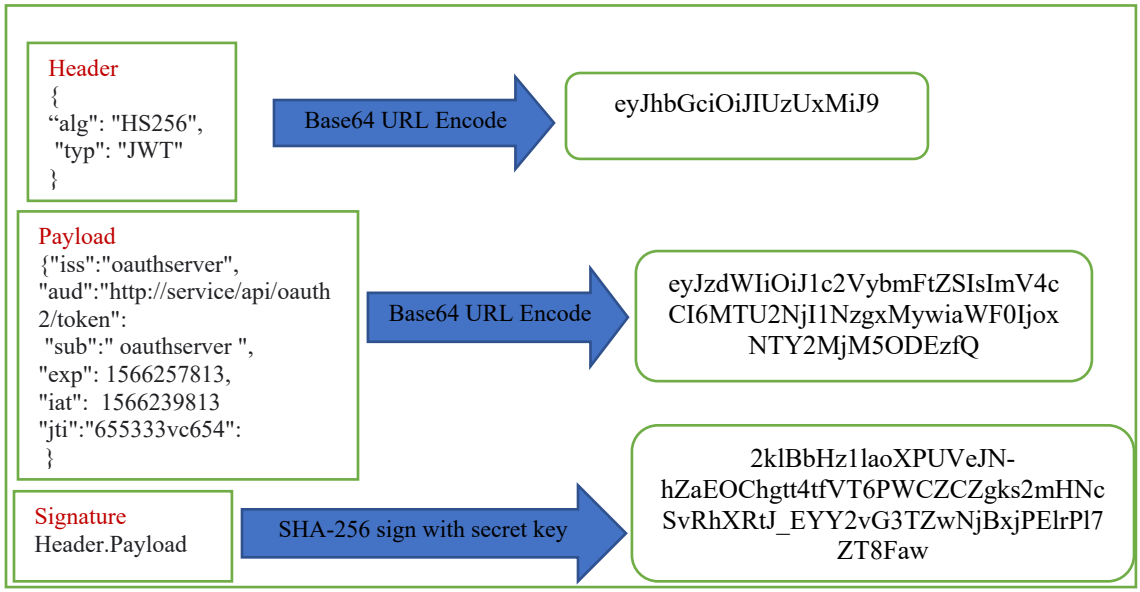

Figure 5: Content of the JWT Token

8. The authorization server returns JWT containing the user's information in the response body as shown in Fig. 7 that JWT in response is stored locally on the client system, it should be set on each request via JavaScript on client-side and the user is allowed inside the application. The access token is only valid for 5 minutes and refresh tokens can also expire but are rather long-lived. Otherwise, the operation in this step may be acknowledged with one of the following errors as shown in Table 3:

Table 3: Operation return errors

Response Code	Response Body	Description
400 Bad Request	{"error": consent_required"}	The user or administrator hasn't granted (or has revoked) their consent to be impersonated by the app.
400 Bad Request	{"error": invalid_grant"}	The JWT token was valid, but some of the claims were not.

9. The client needs both an access token (which it was obtained in the previous step), and base URI that is unique to the user on whose behalf client is making the API call.
Request syntax: *GET https://oauth/API*

Authorization: Bearer
ISSUED_ACCESS_TOKEN

Otherwise, the operation in this step may be acknowledged with one of the following errors as shown in Table 4:

Table 4: operation return errors

Response Code	Response Body	Description
401 Unauthorized	{"ErrorCode": "invalid_request"}	Response returned for requests that do not contain a proper bearer token.
401 Unauthorized	{"error":"AccessToken expired"}	The JWT was expired. it then refreshes it's token by submitting the refresh token to a specified end

10. The resource server validates for authenticity of the incoming valid JWT in the Authorization header Since the resource server knows the secret key, the resource server can perform the same signature algorithm as in Step 7 on the JWT.

A. The resource server takes the original Base64url-encoded Header and original Base64url-encoded Payload segments (Base64url-encoded Header + "." + Base64url-encoded Payload), and hash them with SHA-256.

B. The resource server uses SHA-256 secret key to encrypt.

The resource server can then verify that the generated signature obtained from its own hashing operation matches the signature on the JWT itself (it matches the JWT signature created by the authentication server).

If the generated signature match, then that means the JWT is valid which indicates that the API call is coming from an authentic source.

Otherwise, if the signatures don't match, then it means that the received JWT is invalid, and the request must be rejected which may be an indicator of a potential attack on the application.

11. If the token is valid, the resource server sends resources back to client.

Using the access token to make any authenticated API calls as long as the token is valid (it's not expired). When the access token expires, use the refresh token to generate a new one from the authorization server to keep the user's login status and not use to make an authenticated API call.

This mean that the user basically has 5 minutes to use the JWT before it expires. Once it expires, user will use current refresh token to try and get a new JWT.

Since the refresh token has been revoked, this operation will fail and user will be forced to login again.

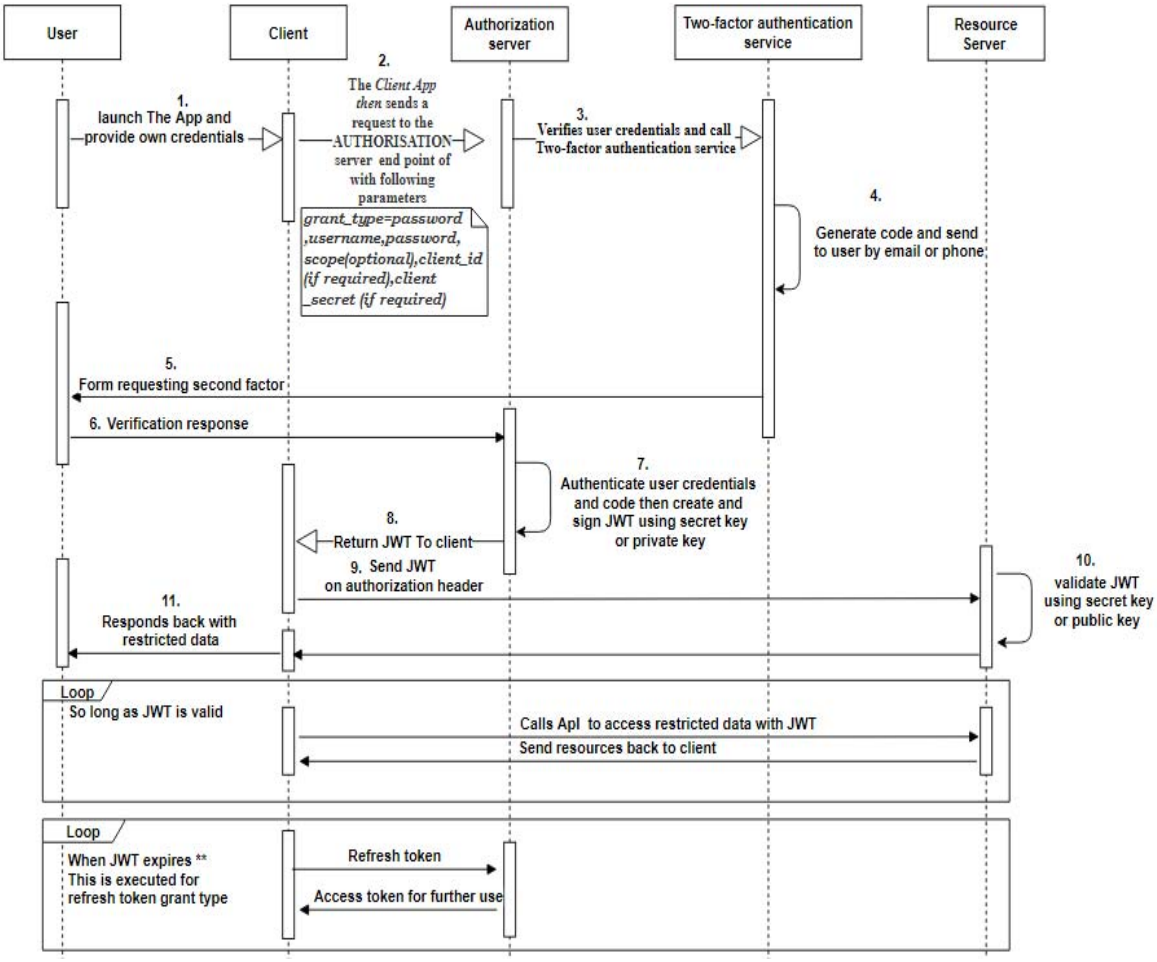

Figure 6: JWT Authentication Workflow with HS256 and RS256

4.2 The RS256 JWT Digital Signature:

In the second signature type RS256, an asymmetric signature is generated with two keys standard 256 bit Public/Private keypair that private key is used to sign the JWT and a different public key is used to verify the signature.

Unlike symmetric algorithms, RS256 offers confirmation that OAuth is the signer of a JWT since OAuth is the only party with the private key.

In this type of signature:

1. There is separation between the ability to create valid JWT and the ability to validate JWT.
2. The authentication server is only responsible for creating valid JWT.
3. The resource server is only responsible for validation JWT.

In RS256 algorithm two keys are generated instead of one:

There is a first key called private key but this time it will be owned only by the issuer, used only to sign JWT, but it can't be used to validate them and can be kept in a confidential location, only known to the issuer of the JWT. There is a second key called the public key, which is used by the resource server only to validate JWT, but it can't be used to sign new JWTs.

The authorization server authenticates the client, generates JWT and signs it using the private key. Any incoming requests will be then verified using the public key as shown below Figure 6.

1. The user makes a login request.

2. The Client then sends a request to the authorization server end point of with following parameters:
   The grant type parameter should be set to password.
   ClientID and client secret: This is the unique identification of the client. These two are required if the server has made client authentication mandatory that the Client App obtained during registration.
   Scope: For which the Client is requesting authorization. These are space delimited list of scope string; for example – profile, email, location etc.
   Username and Password: The resource owner's username and password.
3. If credentials exist in the database, the authorization server validates the user name and password and will call Two-factor authentication service to generate a security code for user.
4. The Two-factor authentication service receives the request from authorization server then it will generate a security code and send it to end user by email or SMS that code will be expire in two minutes.
5. User will get the Two-factor authentication form to enter code that can help prevent unauthorized access user' account by third party and protect against numerous other types of phishing threats.
6. User inputs received code and commits to callback URL.
7. The authorization server verifies if the user is legit and generates JWT and signs it using the private key then responds with JWT containing the identity of the user.
8. The JWT in response is sent to client that JWT is stored on Client side: Local Storage for the user's browser.
9. The user sends JWT to resource server.
10. The resource server first verifies if the request contains the JWT (responds with an error if not passed). The JWT is then verified using the public key. To verify the signature:
    A. The resource server checks RS256 Signatures that take the header and the payload, and hash everything with SHA-256.
    B. Decrypt the signature using the public key, and obtain the signature hash.
    C. The resource server compares the signature hash with the hash that he calculated based on the Header and the Payload.

    If the calculated signature does not match the original Signature included with the JWT, the token is considered invalid, and the request must be rejected.
11. The resource server allows the client access to the protected resources.

5. ATTACKS ON JWT

JWTs are encrypted or signed tokens that can be used to store claims on browsers and mobile clients. These claims can easily be verified by receivers through shared secrets or public keys. The JWTs specification allow for custom private claims, making JWTs a good tool for exchanging any sort of validated or encrypted data that can easily be encoded as JSON. JWTs have their own pitfalls and common attacks.

In this section proposes about the token attacks, and later how to prevent them. JWT tokens are stored on the frontend and the backend and are frequently sent over the network (depending on the session flow). As such, they are vulnerable to several types of attacks.

5.1 Cross Site Scripting (XSS) attacks:

Cross site scripting [19] [20] is the method where the attacker injects malicious script into trusted website application running on the victim's browser and attempt to inject JavaScript through form inputs. The injected code reads and transmits auth JWT tokens and cookies to the attacker. Local Storage can be read just as easily from the same injected code, so if the authentication data was stored in local storage, it could be easily grabbed. Web Storage (local Storage/session Storage) is accessible through JavaScript on the same domain. This means that any JavaScript running on application will have access to web storage, and because of this can be vulnerable to (XSS) attacks can be executed on the sessions that do not need any authentication.

5.2 Cross-site request forgery attack

Cross Site Request Forgery (CSRF/XSRF) [19] is one of the most popular ways of exploiting a server. It attacks the server by forcing the client to perform an unwanted action and this attack is not used to steal auth JWT tokens and attempts to trick users into performing an action using an existing session of a different

website. Since cookies with JWT are sent by the browser client by default, an attacker might trick user into clicking on a malicious link and submitting an authenticated request to client app. The CSRF is executed when the session is authenticated and the user or the browser is trusted by the website as shown Figure7.

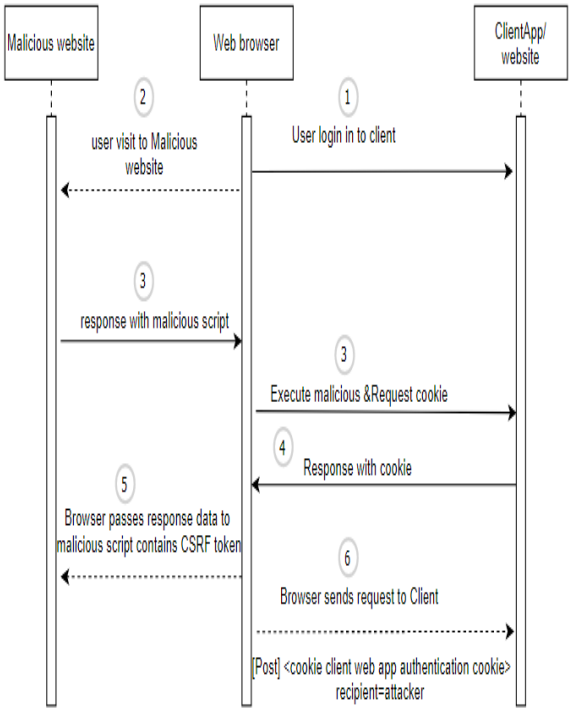

Figure 7: CSRF attack workflow

6. EXPERIMENTAL RESULTS AND DISCUSSION

Framework Implementation is done by integration OAuth with JWT. Implementations are performed by JAVA and MySQL server that is used as the database and configured with Spring Security OAuth2 [21] and the front-end is created with HTML, jQuery ,JavaScript and ajax and using Bootstrap and perform Form validation. The main program of the project is run in web application, web server and back-end API server (including Authentication Server and Resource Server), we treat web browser as OAuth Client and users must authorize the device or browser to access their resource on the resource server, as well as using ajax-request in JavaScript to obtain a JWT from authorization server and connect with REST API and, we build a two-factor authentication service as Authentication methods before granting access.

6.1 Performance Evaluation

In this framework, the performance evaluation of the algorithm SHA-256 and RSA-256 is based on terms of:

- The security of tokens to prevent the security issues attacks.
- The time to generate the token.
- The size of the token generated.
- Time of token data transfer from the client request to the resource server until the token response is received by the client.

The performance evaluation of the two algorithms is tested on JWT with experiment 10 times the process against number of users.

6.1.1 Security of tokens against attacks

Framework Implementation is done by making OAuth with JWT to prevent the security issues attacks in the parameter testing the security of tokens. Common attacks using JWTs have been discussed. All of these attacks can be successfully prevented by following these mitigations.

6.1.1.1 Preventing XSS attacks

JWT stored in local Storage passed Authentication Bearer header can help but is always still susceptible to XSS attacks. JWT is accessible via JavaScript and can be passed with any request. The proposed approach is to store the JWT in an HTTP Only cookie to prevent JavaScript access to the token. This can be prevented fairly easily by using HTTP Only or Secure cookies to store auth tokens. That way, in case of XSS, the JWT token can't be read by the malicious script. it is better to store JWT in cookies than in local storage or in a JavaScript variable.

6.1.1.2 Preventing CSRF attacks

Using Cross-Origin Resource Sharing (CORS) [22] to secure user data that CORS handles this vulnerability well, and disallows the retrieval and inspection of data from another Origin, CORS will prevent the third-party JavaScript from reading private data , and will fail AJAX requests with security errors. CORS is an important consideration for using developing browser applications with JavaScript because most requests to resources are sent to an external domain (including Authentication Server and Resource Server) for a web service API, a main web application gathered the information from these servers.

In the Proposed of approach as shown Figure 8 checking the Origin header this happens when end-user execute AJAX cross domain request using jQuery Ajax interface, Fetch API, or plain XMLHttpRequest is as follow:

1. The Victim user is logged the legitimate website.
2. The Victim Web Browser visits AttackerWebSite.com.
3. AttackerWebSite.com provides a malicious script that instructs the Victim Browser to make a request that includes cookies to server. The Origin of this malicious code is "https:// AttackerWebSite.com".
4. The Victim Browser submits the request that includes its legitimatewebsite.com session cookies.
5. legitimatewebsite.com responds with private data. The Origin of this response is "https:// legitimatewebsite.com".
   These are the headers that the server sends back in its response:
   a. Access-Control-Allow-Origin: <origin>: This is used to specify the origin allowed to access the resource on the server. It's possible to specify that only requests from a specific origin are allowed – Access-Control-Allow-Origin: https:// legitimatewebsite.com
   b. Access-Control-Max-Age: <seconds>: This indicates the duration for which the response of a preflight request can be cached.
   c. Access-Control-Allow-Methods: This indicates the methods that allowed when attempting to access a resource.
   d. Access-Control-Allow-Headers: This indicates the HTTP headers can be used in a request.
6. The Victim Browser receives the response and examines the CORS headers to see if legitimatewebsite.com indicates code from the origin "https://AttackerWebSite.com" is permitted to read the response.
7. If the CORS response headers from legitimatewebsite.com permit cross-origin communication to the origin https://AttackerWebSite.com, the Victim Browser will give the malicious script access to the response contents containing private data and responds by returning the requested resource. If the cross-origin communication is not allowed by the response headers, the browser will enforce and block the Malicious Script from reading response content.

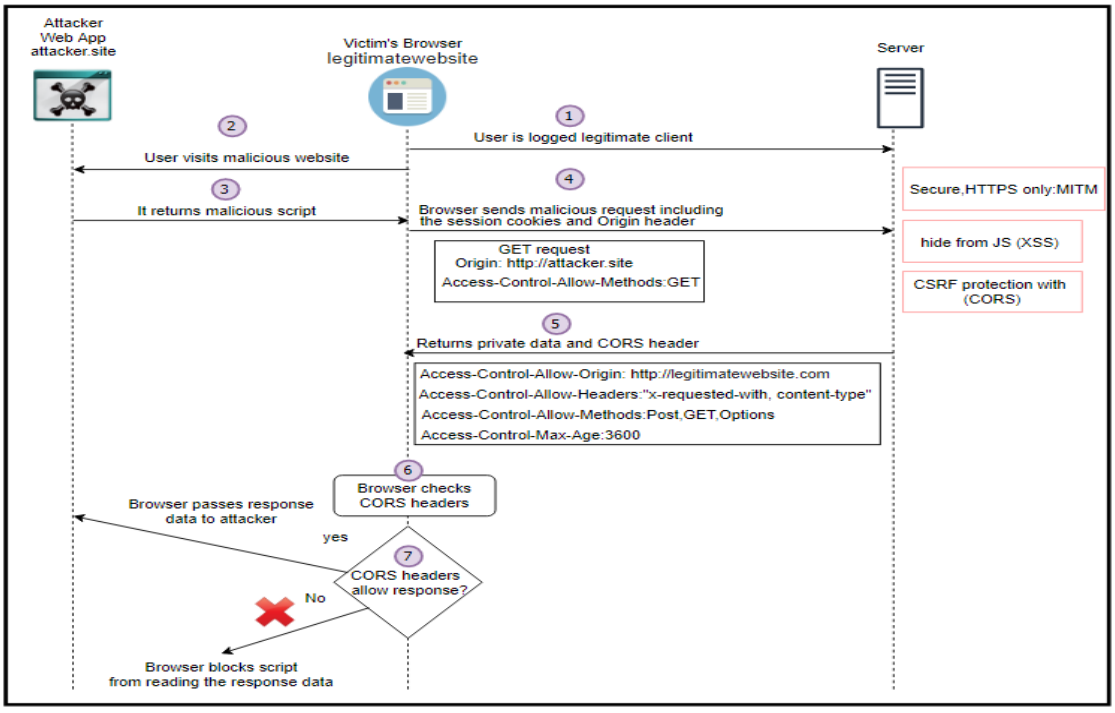

Figure 8: CORS process flow

6.1.1.3 Preventing Man in the middle attacks

Proposed scheme uses HTTPS to protect against this type of attack and secure cookies throughout application that Man in The Middle attack is

possible if unencrypted data is sent and a malicious attacker intercepts the requests sent from a client or response sent from a server. This can be prevented if the channel is encrypted. These have been resolved to prevent man-in-the-middle type of attacks, meaning it will never be possible to decode HTTPS traffic by getting a copy of it. The End-user on the client host wants to make a secure transaction to authorization server. An attacker host is connected to the same network as the client and will manipulate HTTPS traffic between the client host and server host but the data sent by the client-authorization server side encrypted and packet send is sent through secure channel and then its integrity is maintained.

6.1.1.4 Comparison vulnerability and protection of XSS and CSRF attack for JWT's Storage Security

In this section compares vulnerability and protection of XSS and CSRF attack for storing the JWT on the cookies or on the web storage as shown in Table 5:

Table 5: JWT's Storage Security

	XSS	XSS Protection	CSRF	CSRF Protection
JWT in Web storage	Vulnerable	• Ensure use HTTPS • Add X-Frame-options header to every response, and set it to Deny • X-XSS-Protection to 1 • X-Content-Type-Options →no sniff	Not vulnerable	---------
JWT Cookie	Vulnerable	• Set the HTTP only flag • Secure cookie • Ensure use HTTPS	Vulnerable	• CSRF mitigation with a cross origin request (CORS)

When implemented correctly it can represent a secure solution API and a reduced server load, as some part of data can be retrieved from the token's payload without extra DB hits.

6.1.2 Practical experiments

Testing is done with Postman[23] who has a function as application used for testing the REST API that was created. There are two the process carried out at Postman namely POST and GET:

1. Post to send parameters in the form of a username and a valid password to generate JWT.
2. This JWT as a key to gain access to make the next request.
3. The GET process is done with enter the key or JWT that what was obtained in the post process to obtain data.
4. When make requests, generally JWT are sent at HTTP header.
5. The default key for carrying JWT is "Authorization". Usually by default the JWT will always be updated after the HTTP request is made, and can be accessed in the response header.
6. Display the results of the application of HMAC SHA-256 and RSA-256.

6.1.2.1 Generate token time

The first test is to generate a token process and in Table 6 experiments are presented and the results of the test context as shown Figure 9.

Table 6. Generate token processing time

Testing	Signing Algorithm RSA-256 (Generate token processing time)	Signing Algorithm SHA-256 (Generate token processing time)
1	211	253
2	468	240
3	356	245
4	231	254
5	153	257
6	245	229
7	255	224
8	204	258
9	206	239
10	196	251
AVG	252.5	245

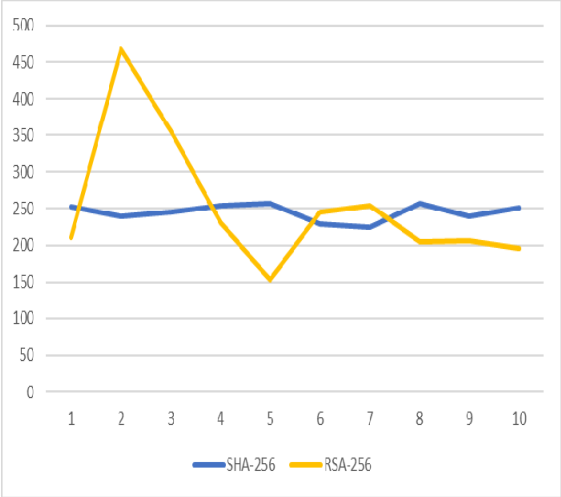

Figure 9: Evaluation performance of time process generating for tokens using RSA-256 and SHA-256 Algorithms

6.1.2.2 Token size

The second test is JWT Token size process and in Table 7 experiments are presented and the results of the test context as shown Figure 10.

The size of the token depends on adding claims in JWT, each request to the server must involve the JWT along with it.

Table 7. Token size

Testing	Signing Algorithm RSA-256 (bytes)	Signing Algorithm SHA-256 ((bytes)
1	2460	1960
2	2450	1770
3	2420	2250
4	2400	2250
5	2410	1740
6	2410	2250
7	2410	2250
8	2410	1740
9	2420	1740
10	1740	1290
AVG	2353	1924

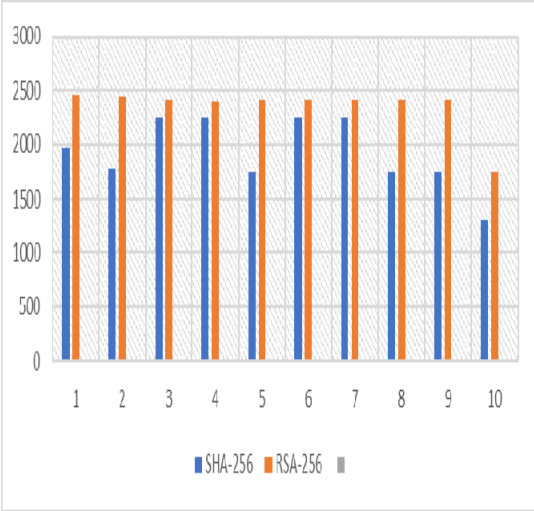

Figure 10: Evaluation of tokens size using RSA-256 and SHA-256 algorithms.

6.1.2.3 Time of token data transfer

Testing time transferring token data from client to resource server can be seen in Table 8 and Figure 11

Table 8. Time of token data transfer

Testing	Signing Algorithm RSA-256 (ms)	Signing Algorithm SHA-256 (ms)
1	170	40
2	122	46
3	106	96
4	120	102
5	134	64
6	137	106
7	70	87

8	42	74
9	126	96
10	79	55
AVG	110.6	76.6

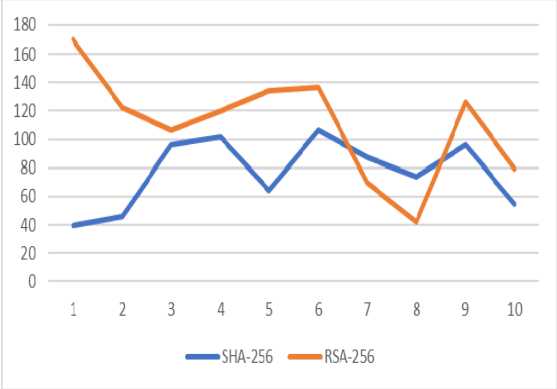

Figure 11: Evaluation performance time of token data transfer using RSA-256 and SHA-256 algorithms.

6.2 Performance evaluation of the proposed scheme OAuth2.0 with JWT and OAuth2.0 without JWT

In this section compares evaluation of the proposed scheme OAuth2.0 with JWT and OAuth2.0 according with parameters testing average the time to generate the token, the time data transfer speed of the token and the size of the token generated from the client request to the server can be seen in Table 9.

Evaluation the performance of the token generating time (table 6 and figure 9) that the SHA-256 algorithm to an average score of 245 ms, there is an increase of 2.97 % when compared to RS256.

According token size (table 7 and figure 10) SHA-256 produces 1924 bytes token size when compared to RS256 increased by 18.23 %. Test results (table 8 and figure 11) in SHA-256 token transfers received an average score of 76.6 ms, while an increase of 30.74 % in RS256.

In OAuth2.0 without JWT the resource server asks the authorization server for the user details related to the token and whether the token is valid or not on each client request that data token transfer received an average score of 112 ms, compared with proposed scheme OAuth2.0 JWT (Algorithm SHA-256) take less time with average 76.6 ms and (AlgorithmRSA-256) with average 110.6 ms.

The overall results show that the experimental results show the use of the SHA-256 has better performance than RS256 for generating tokens, the size of tokens and time of token data transfer.

Table 9: Comparison the proposed scheme Stateless OAuth2 using OAUTH2 + JWT and OAuth2.0 without JWT

Parameters	OAuth2.0 without JWT	Proposed scheme OAuth2.0 with JWT (AlgorithmRSA-256)	Proposed scheme OAuth2.0 with JWT (Algorithm SHA-256)
Generate token time	733.8 ms	252.5ms	245 ms
Token size	483 bytes	2353 bytes	1924 bytes
Data token transfer	122 ms	110.6 ms	76.6 ms

6.3 Preventing User Credentials Phishing Attack

Countermeasures for protection against phishing user credentials, proposes solution applies two-factor authentication service that is considered against various phishing attempts and to ensure only authorized user allow to access the application by verify the user identity before any accesses grant to the user for the protected services as follow:

a) The user enters the username and password and after validation, user is prompted for the 2-factor authentication code in the next form to increase the assurance that a correct user has been authorized to access a secure system.

b) A Security code was dynamically generated in Two-factor authentication service and sent to user by email or SMS so it was difficult to phish the User Credentials.

c) The Authorization server accepts the request and validates the username and password first

and then the 2-factor security code entered by the user that received by e-mail or SMS.

7. CONCLUSION AND FUTURE SCOPE

In this research, using OAuth 2.0 and JWT is proposed to achieve stateless and gain performance improvement. The statistic results of this research have been presented about the comparison of token-based authentication performance using JWT with algorithms SHA-256 and RS256 and performance evaluation of OAuth2.0 without JWT.

Analysis of security token attack have been presented and prevented these types of attacks on JWT such as XSS attacks and CSRF attacks. In order to avoid original password being stolen, two-factor authentication service is also introduced by verifying the user identity before any accesses grant for the protected services by sending a security code to user's e-mail or SMS.

When store JWT tokens in cookies, the size limit of a cookie or URL may be quickly exceeded that depends on the app's architecture and especially data that is stored in the token. There may be access tokens and refresh tokens for accessing users and getting the refresh token is a bit more complicated. We will implement other pattern to secure store JWT.so, it is the future scope to apply this proposed scheme in such a scenario.

REFERENCES:

[1] Fielding, R.T. and R.N. Taylor, *Principled design of the modern Web architecture.* ACM Transactions on Internet Technology (TOIT), 2002. **2**(2): p. 115-150.
[2] D. Hardt, E. *The OAuth 2.0 Authorization Framework*. October 2012; Available from: https://tools.ietf.org/html/rfc6749.
[3] Torroglosa-García, E., et al., *Integration of the OAuth and Web Service family security standards.* Computer networks, 2013. **57**(10): p. 2233-2249.
[4] auth0. *OAuth 2.0 Authorization Framework*. Available from: https://auth0.com/docs/protocols/oauth2.
[5] Hardt, D., *The OAuth 2.0 authorization framework*. 2012, RFC 6749, October.
[6] Peyrott, S., *The JWT Handbook.* Seattle, WA, United States, 2016.
[7] Bradley, J. *JSON Web Token (JWT)*. May 2015; Available from: https://tools.ietf.org/html/rfc7519.
[8] Auth0. *Introduction to JSON Web Tokens*. Available from: https://jwt.io/introduction/.
[9] connect2id:. *JOSE / JWT cryptographic algorithm*. Available from: https://connect2id.com/products/nimbus-jose-jwt/algorithm-selection-guide.
[10] KENTON, W. *Two-Factor Authentication*. May 9, 2019; Available from: https://www.investopedia.com/terms/t/twofactor-authentication-2fa.asp.
[11] Leiba, B., *Oauth web authorization protocol.* IEEE Internet Computing, 2012. **16**(1): p. 74-77.
[12] Yang, F. and S. Manoharan. *A security analysis of the OAuth protocol*. in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 2013. IEEE.
[13] Li, W., C.J. Mitchell, and T. Chen. *Mitigating CSRF attacks on OAuth 2.0 systems*. in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. 2018. IEEE.
[14] Solapurkar, P. *Building secure healthcare services using OAuth 2.0 and JSON web token in IOT cloud scenario*. in *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*. 2016. IEEE.
[15] Ethelbert, O., et al. *A JSON token-based authentication and access management schema for Cloud SaaS applications*. 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud) 2017; 47-53].
[16] Spyna, L. *Implicit Grant Flow*. 2018; Available from: https://itnext.io/an-oauth-2-0-introduction-for-beginners-6e386b19f7a9.
[17] Dierks, T. and E. Rescorla, *The transport layer security (TLS) protocol version 1.2*. 2008.
[18] Ottinger, J.B. and A. Lombardi, *Spring Security*, in *Beginning Spring 5*. 2019, Springer. p. 313-343.
[19] Wichers, D., *Owasp top-10 2013.* OWASP Foundation, February, 2013.
[20] Guo, X., S. Jin, and Y. Zhang. *XSS vulnerability detection using optimized attack vector repertory*. in *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. 2015. IEEE.
[21] Alex, B., et al., *Spring Security Reference.* URL https://docs. spring. io/spring-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

security/site/docs/current/reference/htmlsingle/. [utoljára megtekintve: 2017. 04. 21.], 2004.

[22] Sharing, W.C.C.-O.R. *Cross-Origin Resource Sharing*. 16 January 2014; Available from: https://www.w3.org/TR/cors/.

[23] Postman-tutorial. postman-tutorial. March 2020; Available from: https://www.guru99.com/postman-tutorial.html.