PERFORMANCE EVALUATION OF THE SECURITY LEVEL OF OAUTH 2.0 IN THE IMPLEMENTATION OF AUTHORIZATION SYSTEMS FOR ACCESS TO WEB RESOURCES ON CLOUD-BASED PLATFORMS

DIEGO VELOZ CHÉRREZ[1], GUILLERMO VALENCIA[2], FABRICIO JAVIER SANTACRUZ SULCA[3], DIEGO RAMIRO ÑACATO ESTRELLA[4]

[1]Escuela Superior Politécnica del Chimborazo (ESPOCH)
Riobamba, Ecuador
diego.veloz@espoch.edu.ec
ORCID: https://orcid.org/0000-0002-6084-2213
[2]valepetroguillo@gmail.com
https://orcid.org/0000-0002-3938-2021
[3]Escuela Superior Politécnica del Chimborazo (ESPOCH)
Riobamba, Ecuador
fabricio.santacruz@espoch.edu.ec
https://orcid.org/0000-0001-7123-2552
[4]Escuela Superior Politécnica del Chimborazo (ESPOCH)
Riobamba, Ecuador
diego.nacato@espoch.edu.ec
https://orcid.org/0000-0002-7233-9076

Abstract— The demand for remote access has experienced exponential growth., making it difficult for users to maintain different accounts for each service they use. In the traditional client-server authentication model, clients enter their credentials, usually usernames and passwords, to request a restricted access resource from servers. However, there are some drawbacks with these processes: decreased confidentiality, user sensitivity to phishing, full access to resources and limited reliability. The purpose of this paper was to assess the security level of access control over resources on cloud-based platforms by implementing two real scenarios, one with a traditional authentication system and the other implementing an access authorization system using the OAuth2 framework. To reach this goal, an infrastructure has been created, using virtualization approaches, which sends requests to the server that owns the resources and this in turn communicates through APIs to a database server in AWS. The OWASP project was used to analyze the vulnerabilities in these scenarios, studying the exposure of confidential information, level of access to resources, alert control, as well as system response time parameters to measure their efficiencies. The results showed that the implementation of OAuth2, as the basis for authorization systems, improves security in the exchange of client-server messages through the implementation of tokens, reduces the exposure of confidential information, facilitates access to resources on different platforms and even makes it easy to assign roles and levels of access to resources.

Keywords—cloud computing, web resources, REST APIs, design pattern, telematic security

I. INTRODUCTION

Every smart electronic device connected to the Internet has access to Cloud computing servers, which allows information to be shared through computing platforms [2]. This exchange of information is transmitted through public or private infrastructures, which leads to the definition of methods of authentication and authorization to control access and who can reach data.

In traditional authentication models, clients enter their credentials (usually a username and password) to request a restricted or protected resource from the server. However, sending sensitive user information creates side effects such as decreased confidentiality, user sensitivity to phishing, full access to resources, limited reliability, and difficulty implementing stronger authentication. Additionally, user credentials are saved in plain text in third-party applications for future access to resources and without the possibility of restricting the duration of the authorization or limiting the amount of information that is reached and the difficulty to revoke that permission granted [6].

The demand for remote data access has grown exponentially making it difficult for users to maintain different accounts for every service they use [7]. As the Internet has developed, different web services have cooperated to create mash-up services. In this case, the owner of a resource must be authenticated by the server that stores the resource, and a third-party web application must be authorized to access the resource [20]. One solution to this problem is the OAuth 2.0 framework, which uses a single account to identify users across different services without sharing or transferring passwords. The aim of this work was the evaluation of security levels for authentication and authorization systems in clouding platforms by sending access requests and measuring the exchange of messages between servers and clients. For this purpose, two infrastructure models were implemented, a traditional user and password authentication approach and the other one based on a resource access approach through access tokens using the OAuth 2.0 protocol.

II. OAUTH 2.0

OAuth is an emerging authorization standard that has been adopted by an increasing number of websites such as Twitter, Facebook, Google, Yahoo!, Netflix, Flickr, and many other social media and resource providers. It has an open-web specification, so that organizations can access protected resources between different websites. This is achieved by giving users permission to access protected content in third-party applications without having to send credentials to these applications [4] [11].

OAuth 2.0 defines a secure access framework for Application Programming Interfaces (APIs), typically RESTful, to acquire protected resources. There are three primary participations in the OAuth flow: The client (an application that requests information), who sends an API query to a resource server (RS); The server resource, which hosts the desired resource or data and validate the authentication message that was sent by the client; and the access token provided by the authorization server (AS), which is included in client's API message [5].

OAuth 2.0 includes:

1. A web redirection mechanism that a resource owner can delegate authorizations to their resources (for example, their profile) to some third-party site.
2. Identity mechanisms that can be used by the resource owner to delegate their identity attributes for client applications such as desktop, mobile, and Internet of Things (IoT).
3. A restricted Security Token Service (STS) model notably based on REST principles rather than SOAP messaging. The STS issues tokens and updates expired tokens.
4. A set of mechanisms for REST-based HTTP APIs, including:
   - Protect the identity attributes of resource owners that require consent.
   - Protect the identity attributes of resource owners that are implied.
   - Protect specific data of the resource owner that do not require consent [5].

III. METHOD

The main objective of this work is to evaluate the security level provided by authorization systems in contrast to traditional authentication systems where the client enters their credentials (username and password) to request a restricted or protected access resource to the server [9].
Therefore, to establish the architecture to be implemented, the roles defined by the OAuth 2.0 protocol were considered and fulfilled in the design of this study:

- Resource Owner (RO): It is the owner of a resource, usually hosted on a resource server, who specifies the authorizations to be created on the authorization server. Authorizations are defined through an access token issued to the client [5] [13].
- Resource server: Actor in charge of protecting resources and making them available to authenticated and authorized clients.
- Authorization Server (AS): It issues access and updates tokens to clients.
- Access token: It is a string that represents an authorization issued to the client. They are credentials used to access protected resources. The tokens represent specific scope and duration of access, granted by the resource owner and enforced by the resource server and authorization server.
- Refresh Token: A long-lived token that the client can exchange with the authorization server to obtain a new access token with the same authorizations provided in the expired token.
- Client: It is the application that sends requests to access a resource protected by the resource server and interacts with an authorization server to obtain access tokens.

A. Environment design

The test environment consists of creating a client application, which sends requests to both the resource server and the authorization server in order to access restricted resources. These servers are in the cloud to allow customer access from anywhere and through any device that has Internet access.

Two test scenarios were defined to contrast the level of security between traditional access systems and authorization systems using tokens: The first scenario where the content server allows access to resources through user and password sent by the client. The other scenario has been created taking into account an authorization approach through token validation.

To test the performance of each authentication and authorization system, an Inventory System (IS) was used on the cloud resource server.

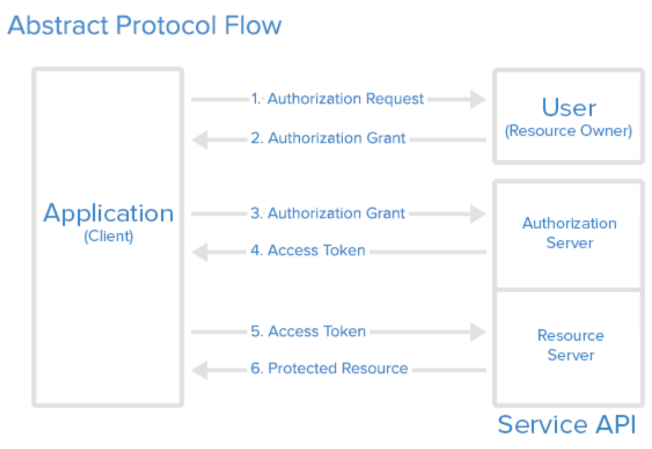


Figure 1. OAuth 2.0 Protocol flow [13]

B. *Classic authentication system*

In this scenario, the client application makes HTTP requests [16], with a basic authentication of user credentials (username and password) previously created in the PostgreSQL database hosted in Amazon Web Service (AWS), to request resources with read-only access permissions of the Inventory System (SI) located on the Heroku Resource Server (SR), as shown in Figure 2. The client application uses PHP programming language and consists mainly of the Aws.php class file and the test_basic.php file. Aws.php takes the user credentials and makes a number of HTTP requests to the resource server that is defined by the sample size as specified in the calculation at the end of this section. The programming code of the Client Application and the configuration of the AWS class are detailed in annex 2.

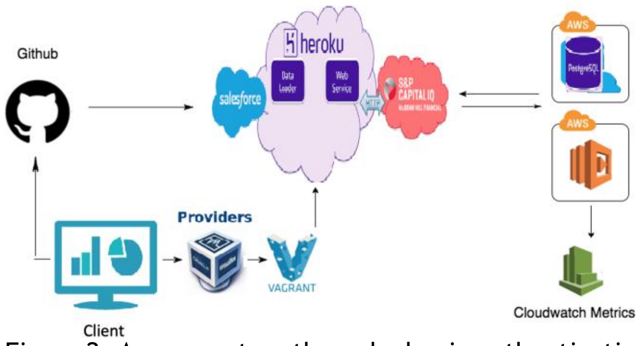


Figure 2. Access system through classic authentication

The function of each part of the system is described below:

- Github: Uses the Git version control system to host the client application and the inventory system to access shared resources. The software that runs GitHub is written in Ruby on Rails.
- Vagrant: The client uses the Vagrant virtualized environment tool that uses a VirtualBox instance as the image base and configures the software and hardware resources through the Vagrantfile file as shown in Annex 2. This virtualization approach allows for scalability and makes the system more efficient.
- PostgreSQL: It is an open source, object-oriented relational database management system used to store inventory system information that contains resources to be accessed.
- Inventory system: Web application located in the resource server to issue and control access to web resources. The inventory system uses Model View Controller (MVC), where the controller allows the access to the resources.
- Heroku (RS): Resource server that takes client requests and verifies their credentials to deliver or deny the requested resource. The characteristics of this server are:
  - Description: Canonical, Ubuntu, 18.04 LTS, amd64 bionic image build on 2019-07-22
  - Status: available
  - Platform details: Linux/UNIX

  - Platform: Ubuntu
  - Image Size: 8GB
  - Visibility: Public
  - Network interfaces: eth0
- Cloudwatch Metric: It measures AWS data base performance parameters.

C. *Authorization system using access tokens*

In this second scenario, the client application requests an access authorization token from the authorization server hosted at CEDIA through an HTTP Request [16], with client credentials previously created in the SQLite engine database, to request the resources with read-only access permissions located in the inventory system on the Heroku resource server, as shown in the following figure.

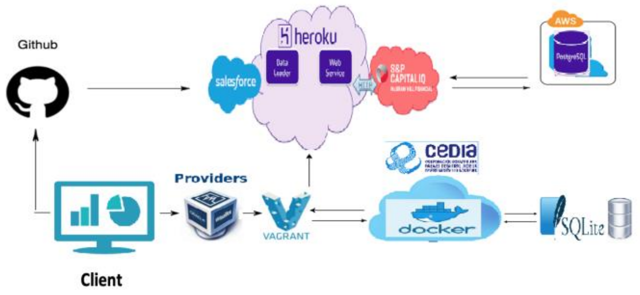
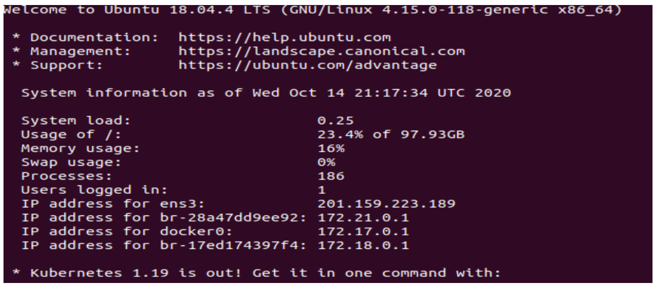


Figure 3. Access authorization system using tokens

Common parts of this system between the previous system performs the same function, with the following differences:

- CEDIA Server: It is a Platform as a Service (PAAS) where a virtual machine performs the function of authorization server for token generation. The setup of this server is shown in the following figure:



```
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-118-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Wed Oct 14 21:17:34 UTC 2020

  System load:                     0.25
  Usage of /:                      23.4% of 97.93GB
  Memory usage:                    16%
  Swap usage:                      0%
  Processes:                       186
  Users logged in:                 1
  IP address for ens3:             201.159.223.189
  IP address for br-28a47dd9ee92:  172.21.0.1
  IP address for docker0:          172.17.0.1
  IP address for br-17ed174397f4:  172.18.0.1

 * Kubernetes 1.19 is out! Get it in one command with:
```

Figure 4. CEDIA server resources

- Docker: CEDIA server is defined with a container approach for portability of the systems to be implemented. The function of Docker is authorization tokens generation of JSON Web Token type (JWT) for access to resources. The docker compose setup is as follows:

```
version: '3.7'
volumes:
  logs:
    driver: local
services:
  slim:
    image: php:7-alpine
    working_dir: /var/www
    command: php -S 0.0.0.0:8080 -t public
    environment:
      docker: "true"
    ports:
      - 8080:8080
    volumes:
```

```
- .:/var/www
- logs:/var/www/logs
```

- SQLite: It is a small relational database that has the function of registering the information of the authorization system as well as authorization and update tokens. The setup is detailed in annex 3.
- The client requests a JWT access token from the Access Server (SA) which runs in a Docker container. This token is verified in the Heroku SR through public and private keys.

D. *Application registry*

Before using OAuth, the application must be registered through authorization server tfilling a registration form in the service's website developer API. The fllowing information must be provided:

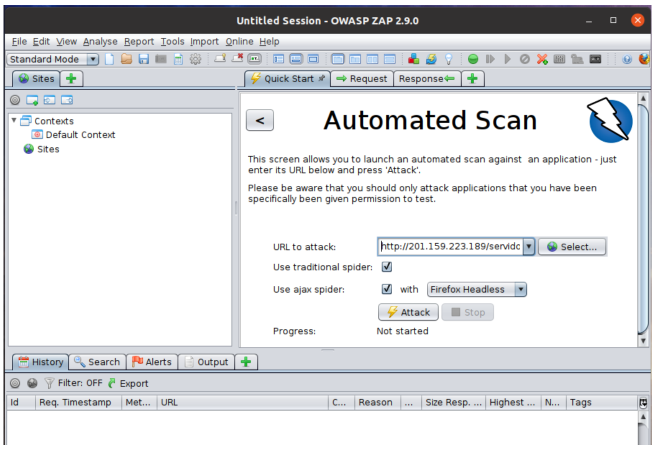
- The name of the application
- The application website
- Redirect URI or Callback URL

Redirect URI will redirect the user after the request is authorized or denied, then the application will handle authorization codes or access tokens [13].
After the application is registered, the service will issue the credentials to the client application in the form of:
Client ID: Specified as the client_id when interacting with the resource server.
Client Secret: Specified as client_secret when exchanging an authorization code for an access token and a refresh access token used on the server side of the application's web stream [14].
The authorization grant types depend on the method used by the application and the authorization types supported by the API. OAuth 2.0 defines three types of authorization, depending on each case:

- Authorization Code – Used in server-side applications.
- Permission based on the resource owner's password: this type of permission is used only for trusted clients.
- Client credentials: allows an application to obtain an access token from the authorization server by sending its credentials (client id and client secret). This type of permission is suitable for applications that need access APIs such as storage or database services [13].

E. *Vulnerability Analysis*

The security level of both scenarios was evaluated considering the Open Web Application Security Project (OWASP) by using its OWASP ZAP tool.


Untitled Session - OWASP ZAP 2.9.0

Automated Scan

This screen allows you to launch an automated scan against an application - just enter its URL below and press 'Attack'.

Please be aware that you should only attack applications that you have been specifically been given permission to test.

URL to attack: http://201.159.223.189/servidc
Use traditional spider: ☑
Use ajax spider: ☑ with Firefox Headless
Progress: Not started

Id	Req. Timestamp	Met...	URL	C...	Reason	...	Size Resp. ...	Highest ...	N...	Tags

Figure 5. Owasp zap autoscan

The resource and authorization servers URL was included in this tool, as shown in Figure 5, to determine the security information properties, such as confidentiality, integrity and availability were guaranteed by both scenarios as well as to measure the level of exposure each one has.

Therefore, the following variables were taken into consideration in the test plan detailed in the following table:

TABLE I. **VARIABLES OF ANALYSIS**

Variable	Measure unit
Flow communication control	Response time of data transmission measured in milliseconds
Authentication and authorization	Number of authorization and authentication mechanisms
Roles and scope	Availability and scope by differentiating the types of access to resources
Traffic encryption	Encryption types
Alerts and monitorization	Number of alert types allowed by cloud services
Threat protection	Number of protection mechanisms against threats and attacks

F. Sampling and statistic analysis

The amount of access attempts to web service resources in a cloud-computing environments was considered as statistical population for the study. Due to the difficulty to determine the number of access attempts to define the observation experiment that could lead to a very large number, thus an infinite population was considered.

The selection of the study sample was obtained through the statistical formula that determines its size considering the study parameter and the type of population.

$$n = \frac{z^2 * p * q}{e^2};$$

Where:
e: Margin of error => (0.05)
n: Sample size
z: Confidence level => (1.96)
p: Sample proportion => (0.5)
q: Uncertainty => (0.5)

$$n = \frac{z^2 * p * q}{e^2}$$

$$n = \frac{1.96^2 * 0.5 * 0.5}{0.05^2}$$

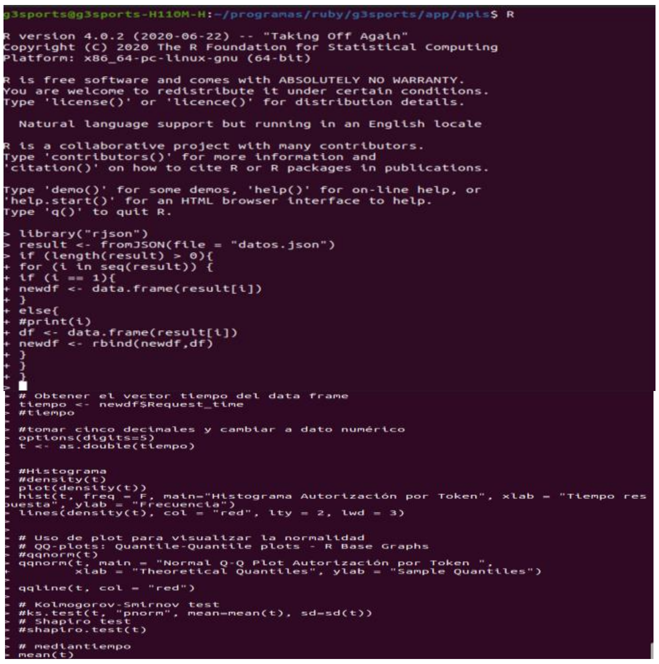
$$n = \frac{0.9604}{0.0025} = 384{,}16$$

n=384.16

Therefore, sample size is 385.

Accordingly, the number of requests that the Client Application must generate in each scenario is 385 HTTP request.

For the analysis of each variable considering the size of the sample, R software environment was used due to its easy interpretation through its command line, which facilitates the manipulation of the large amounts of data obtained.



```
g3sports@g3sports-H110M-H:~/programas/ruby/g3sports/app/apis$ R

R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> library("rjson")
> result <- fromJSON(file = "datos.json")
> if (length(result) > 0){
+ for (i in seq(result)) {
+ if (i == 1){
+ newdf <- data.frame(result[i])
+ }
+ else{
+ #print(i)
+ df <- data.frame(result[i])
+ newdf <- rbind(newdf,df)
+ }
+ }
+ }
> 
> # Obtener el vector tiempo del data frame
> tiempo <- newdf$Request_time
> #tiempo
>
> #tomar cinco decimales y cambiar a dato numérico
> options(digits=5)
> t <- as.double(tiempo)
>
>
> #Histograma
> #density(t)
> plot(density(t))
> hist(t, freq = F, main="Histograma Autorización por Token", xlab = "Tiempo res
puesta", ylab = "Frecuencia")
> lines(density(t), col = "red", lty = 2, lwd = 3)
>
>
> # Uso de plot para visualizar la normalidad
> # QQ-plots: Quantile-Quantile plots - R Base Graphs
> #qqnorm(t)
> qqnorm(t, main = "Normal Q-Q Plot Autorización por Token ",
+        xlab = "Theoretical Quantiles", ylab = "Sample Quantiles")
>
> qqline(t, col = "red")
>
> # Kolmogorov-Smirnov test
> #ks.test(t, "pnorm", mean=mean(t), sd=sd(t))
> # Shapiro test
> #shapiro.test(t)
>
> # mediantiempo
> mean(t)
```

Figure 6. Data framework creation

IV. RESULTS

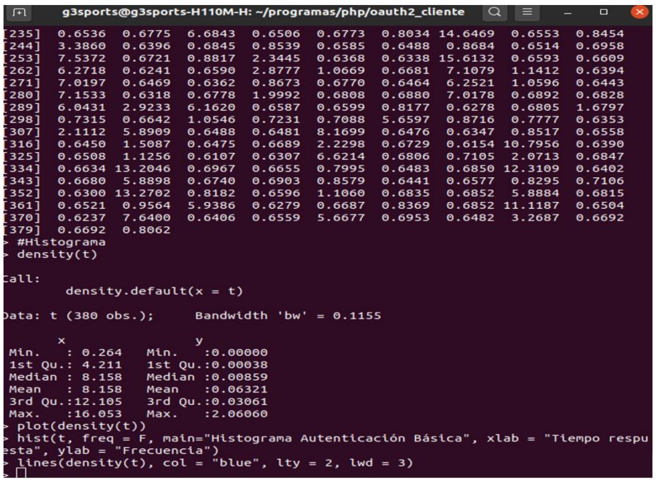
The measurement of the variables defined in Table 1 was carried out for each scenario in order to compare the findings and define the security level of each system.

To access the web resources in the cloud server, a REST API was created for each scenario, which allows establishing connection and message exchange between clients and servers

On the other hand, for the analysis of the measurements of the variables, for each case it was verified if there is a normal data distribution to define the analysis criteria.

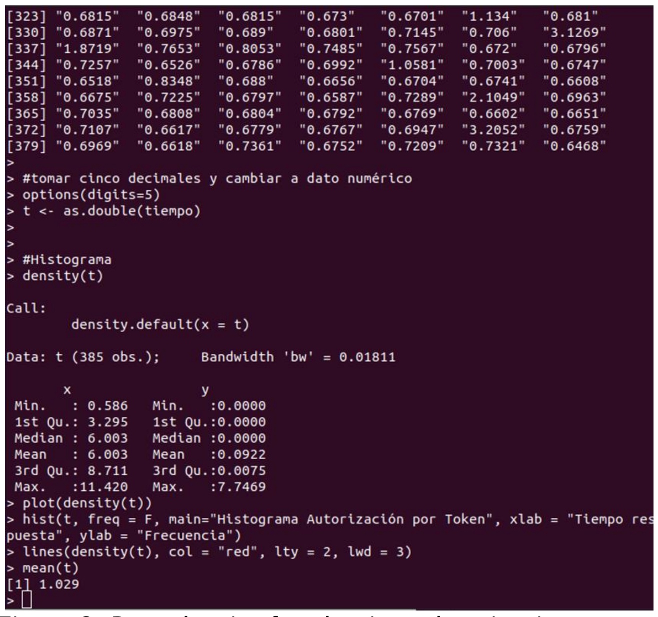
A. *Response time*

According to the central limit theorem, regardless of the content of the distribution, the sample distribution tends to be normal if the sample size is large enough (n>30) [17]. It can be ignored data distribution and use a parametric test. However, to demonstrate this approach, a visual inspection is showed in the next figures with a density plot and a quantile plot using R's ggplot and qqplot plotting tools..



```
g3sports@g3sports-H110M-H: ~/programas/php/oauth2_cliente
[235]  0.6536  0.6775  6.6843  0.6506  0.6773  0.8034 14.6469  0.6553  0.8454
[244]  3.3860  0.6396  0.6845  0.8539  0.6585  0.6488  0.8684  0.6514  0.6958
[253]  7.5372  0.6721  0.8817  2.3445  0.6368  0.6338 15.6132  0.6593  0.6609
[262]  6.2718  0.6241  0.6590  2.8777  1.0669  0.6681  7.1079  1.1412  0.6394
[271]  7.0197  0.6469  0.6362  0.8673  0.6770  0.6464  6.2521  1.0596  0.6443
[280]  7.1533  0.6318  0.6778  1.9992  0.6808  0.6880  7.0178  0.6892  0.6828
[289]  6.0431  2.9233  6.1620  0.6587  0.6599  0.8177  0.6278  0.6805  1.6797
[298]  0.7315  0.6642  1.0546  0.7231  0.7088  5.6597  0.8716  0.7777  0.6353
[307]  2.1112  5.8909  0.6488  0.6481  8.1699  0.6476  0.6347  0.8517  0.6558
[316]  0.6450  1.5087  0.6475  0.6689  2.2298  0.6729  0.6154 10.7956  0.6390
[325]  0.6508  1.1256  0.6107  0.6307  6.6214  0.6806  0.7105  2.0713  0.6847
[334]  0.6634 13.2046  0.6967  0.6655  0.7995  0.6483  0.6850 12.3109  0.6402
[343]  0.6680  5.8898  0.6740  0.6903  0.8579  0.6441  0.6577  0.8295  0.7106
[352]  0.6300 13.2702  0.8182  0.6596  1.1060  0.6835  0.6852  5.8884  0.6815
[361]  0.6521  0.9564  5.9386  0.6279  0.6687  0.8369  0.6852 11.1187  0.6504
[370]  0.6237  7.6400  0.6406  0.6559  5.6677  0.6953  0.6482  3.2687  0.6692
[379]  0.6692  0.8062
> #Histograma
> density(t)

Call:
        density.default(x = t)

Data: t (380 obs.);     Bandwidth 'bw' = 0.1155

       x                y
 Min.   : 0.264   Min.   :0.00000
 1st Qu.: 4.211   1st Qu.:0.00038
 Median : 8.158   Median :0.00859
 Mean   : 8.158   Mean   :0.06321
 3rd Qu.:12.105   3rd Qu.:0.03061
 Max.   :16.053   Max.   :2.06060
> plot(density(t))
> hist(t, freq = F, main="Histograma Autenticación Básica", xlab = "Tiempo respu
esta", ylab = "Frecuencia")
> lines(density(t), col = "blue", lty = 2, lwd = 3)
> 
```

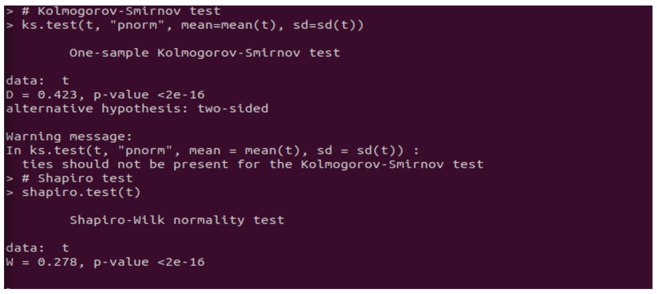
Figure 7. Data density for classic authentication system



```
[323] "0.6815"  "0.6848"  "0.6815"  "0.673"   "0.6701"  "1.134"   "0.681"
[330] "0.6871"  "0.6975"  "0.689"   "0.6801"  "0.7145"  "0.706"   "3.1269"
[337] "1.8719"  "0.7653"  "0.8053"  "0.7485"  "0.7567"  "0.672"   "0.6796"
[344] "0.7257"  "0.6526"  "0.6786"  "0.6992"  "1.0581"  "0.7003"  "0.6747"
[351] "0.6518"  "0.8348"  "0.688"   "0.6656"  "0.6704"  "0.6741"  "0.6608"
[358] "0.6675"  "0.7225"  "0.6797"  "0.6587"  "0.7289"  "2.1049"  "0.6963"
[365] "0.7035"  "0.6808"  "0.6804"  "0.6792"  "0.6769"  "0.6602"  "0.6651"
[372] "0.7107"  "0.6617"  "0.6779"  "0.6767"  "0.6947"  "3.2052"  "0.6759"
[379] "0.6969"  "0.6618"  "0.7361"  "0.6752"  "0.7209"  "0.7321"  "0.6468"
>
> #tomar cinco decimales y cambiar a dato numérico
> options(digits=5)
> t <- as.double(tiempo)
>
>
> #Histograma
> density(t)

Call:
        density.default(x = t)

Data: t (385 obs.);     Bandwidth 'bw' = 0.01811

       x                y
 Min.   : 0.586   Min.   :0.0000
 1st Qu.: 3.295   1st Qu.:0.0000
 Median : 6.003   Median :0.0000
 Mean   : 6.003   Mean   :0.0922
 3rd Qu.: 8.711   3rd Qu.:0.0075
 Max.   :11.420   Max.   :7.7469
> plot(density(t))
> hist(t, freq = F, main="Histograma Autorización por Token", xlab = "Tiempo res
puesta", ylab = "Frecuencia")
> lines(density(t), col = "red", lty = 2, lwd = 3)
> mean(t)
[1] 1.029
> 
```

Figure 8. Data density for classic authentication system

It is possible to use a test of significance comparing the sample distribution with a normal distribution to determine if the data presents a serious deviation from normality. There are several methods to perform normality tests such as Kolmogorov-Smirnov (K-S) normality and Shapiro-Wilk's. Shapiro-Wilk's is the most widely used method for normality testing, and provides better results than K-S. It is based on the correlation between the data and its normal results [18].



```
> # Kolmogorov-Smirnov test
> ks.test(t, "pnorm", mean=mean(t), sd=sd(t))

        One-sample Kolmogorov-Smirnov test

data:  t
D = 0.423, p-value <2e-16
alternative hypothesis: two-sided

Warning message:
In ks.test(t, "pnorm", mean = mean(t), sd = sd(t)) :
  ties should not be present for the Kolmogorov-Smirnov test
> # Shapiro test
> shapiro.test(t)

        Shapiro-Wilk normality test

data:  t
W = 0.278, p-value <2e-16
```

Figure 9. AWS Shapiro Test

As can be seen in the previous figure, p-value is less than 0.05, which implies the sample data has a significance of the normal distribution, it means that the data does not have a normal distribution. Therefore, a comparison is made based on time between all the measurements for the two scenarios, obtaining the average values shown in the following table.:

TABLE II. **RESPONSE TIME**

Classic authentication scenario	Authorization using tokens scenario
1.9098 ms	**1.0290 ms**

B. *Authentication and authorization*

Annex 7 shows the REST API application code that allows the exchange of messages between the client and servers for each scenario and defines the measurement mechanisms. The measurement results are shown below:

TABLE III. **AUTHENTICATION AND AUTHORIZATION MECHANISMS**

Classic authentication scenario	Authorization using tokens scenario
• Basic authentication (user/password) • Id session cookie token	• base64_encode (application/Secret) • Access Token JWT • JWT with scope and expiration
2 mechanisms	**3 mechanisms**

C. *Roles and scope of access control*

The authorization process starts when the client authentication process is correct. The detail of the REST API application code that defines the roles and access scope of each client is shown in Annex 8. The results of the measurements are shown below:

TABLE IV. **MECHANISMS USED IN THE DEFINITION OF CUSTOMER ROLES**

Classic authentication scenario	Authorization using tokens scenario
• Access role by controller	• Unique role based on OAuth2.0 access token • Role based on JWT token expiration • Role based on JWT token scope
1 mechanism	**3 mechanisms**

D. *Traffic encryption*

The following table shows the valuation of encryption and encryption based on the quantity and quality of mechanisms implemented in the exchange of messages for each of the scenarios. Annex 9 details the REST API code.

TABLE V. **ENCYPTION MECHANISMS**

Classic authentication scenario	Authorization using tokens scenario
• Base 64 user credential encoded	• SHA256 Algorithm in header • Base 64 payload encoded • HMACSHA256 sign
1 mechanism	**3 mechanisms**

E. *Alerts and monitoring*

The following table shows the mechanisms present in each system. The REST API code is shown in annex 10.

TABLE VI. **ALERT AND MONITORING MECHANISMS**

Classic authentication scenario	Authorization using tokens scenario
• Trigger after new session SQL in data base	• Monitoring of recent activities by access token • Configuration of alerts and notifications in the middleware • HTTP status codes
1 mechanism	**3 mechanisms**

F. Protection mechanisms against threats

The Open Web Application Security Project (OWASP) defines the top 10 vulnerabilities present in web applications. For this reason, this methodology was used to analyze the vulnerabilities detailed in the following table.

TABLE VII. OWASP VULNERABILITIES

OWASP Top 10 Web Application Security Risks 2017	Vulnerability code
Injection	A1
Broken Authentication	A2
Sensitive Data Exposure	A3
XML External Entities (XXE)	A4
Broken Access Control	A5
Security Misconfiguration	A6
Cross-Site Scripting (XSS)	A7
Insecure Deserialization.	A8
Using Components with Known Vulnerabilities	A9
Insufficient Logging&Monitoring	A10

Vulnerability analyzes for both the traditional authentication system and the token authorization system are shown in annexes 11 and 12 respectively. The following tables show the results of the vulnerabilities found in each system.

TABLE VIII. CLASSIC AUTHENTICATION SYSTEM VULNERABILITIES

Vulnerability	Vulnerability code
Base64_decode weak	A2
PHPSESSID can supplant the session	A5
Insufficient log y monitoring	A10
Sensitive data shown: Server: Apache, PHP	A3

TABLE IX. AUTHORIZATION SYSTEM VULNERABILITIES

Vulnerability	Vulnerability code
Man in the middle en HTTP	A2
Sensitive data shown: Server: Apache2.4., Powered-by:PHP7.3.18, OS:Debian	A3
No encryption from source to destination (TLS)	A5
IP address in URI response	A6

TABLE X. SECURITY MECHANISMS IN BOTH SYSTEMS

Classic authentication scenario	Authorization using tokens scenario
• A1 • A4	• A1 • A4

• A6 • A7 • A8 • A9	• A7 • A8 • A9 • A10
6 mechanisms	**6 mechanisms**

Table 11 shows all the results of the measurement of each variable when analyzing the exchange of messages to access web resources, taking into account a scale from 1 to 4, where 4 is satisfactory and 1 indicates the worst performance. Each of the weightings are detailed in Annex 13 for a better understanding.

TABLE XI. **MEASUREMENT OF VARIABLES**

Likert scale		
Indicators	Classic authenticatio n scenario	Authorization using tokens scenario
Flow communication control	2	3
Authentication and authorization	2	3
Roles and scope	1	3
Traffic encryption	1	3
Alerts and monitorization	1	3
Threat protection	3	3
TOTAL	10	18

V. DISCUSSION

When evaluating each system according to its own features and taking into account the architecture defined for each scenario, it is evident that the authorization system using tokens has a better access control to web resources in cloud-based platforms. Confidentiality and authentication properties are much more efficient in authorization systems since they implement more robust encryption algorithms such as SHA256 and more mechanisms are used to define user roles and to allow access to resources, which guaratees client prioritization and has better access security management. In the traditional authentication environment, the client credentials (username and password) in the header of each request are only encoded using BASE64 code, but not encrypted, which makes it extremely easy for an attacker to decode the header and obtain this sensitive information.

Regarding the level of data exposure, the results show that traditional systems have deficiencies to protect sensitive user information because the client must authenticate the request with a single basic method by sending their user name and password in order to access the resource, causing a high level of exposure, especially when data cannot be sent through private network infrastructures. On the other hand, through the implementation of access tokens, 2 additional security mechanisms are created, the distinction of users by roles and the scope token and its expiration time. In addition, this approach also allows the client to be authenticated on third-party servers without the need for user credentials and passwords to be sent to them, guaranteeing that even public network infrastructures can be used.

Additionally, the efficiency in the communication time has a difference of almost double the time in the average values between both systems. In the traditional authentication system with login, it was observed that the response time began to decrease as the requests for resources increased, this is due to the fact that the session states are stored on the server. The tokenized authorization system is stateless, therefore, a better response time was obtained.

The implementation of the OAuth 2.0 protocol defines a base model for the adoption of security token systems (STS) in the effective control of access to resources.

REFERENCES

[1] NIST, «National Institute of Standards and Tecnology,» 2020. [En línea]. Available: https://www.nist.gov/.

[2] R. H. L. L. P. &. M. S. Hill, Guide to cloud computing: principles and practice., Springer Science & Business Media., 2012.

[3] LinuxFoundationX, «Introduction to Cloud Foundry and Cloud Native Software Architecture (LFS132),» 2020. [En línea]. Available: https://training.linuxfoundation.org/training/introduction-to-cloud-foundry-and-cloud-native-software-architecture/.

[4] K. Kiani, «Four Attacks on OAuth – How to Secure Your OAuth Implementation,» 2020.

[5] Ping Identity, «The Essential OAuth Primer: Understanding OAuth for Securing Cloud APIs,» 2011.

[6] Richer, «OAuth 2 in Action,» 2017.

[7] Siriwardena, «Advanced API Security: OAuth 2.0 and Beyond,» 2019.

[8] O'Raw, «Security Evaluation of the OAuth 2.0 Framework,» 2015.

[9] E. C. H. T. D. T. P. &. B. K. Shernan, More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations., 2015.

[10] L. S. G. W. E. E. S. &. T. H. Seitz, Authentication and authorization for constrained environments (ACE) using the OAuth 2.0 framework (ACE-OAuth)., 2018.

[11] E. Hardt, «The OAuth 2.0 Authorization Framework,» 2012.

[12] A. Lopez, Learning PHP 7, 2016.

[13] DigitalOcean, «Una introducción a OAuth 2,» 2020. [En línea]. Available: https://www.digitalocean.com/community/tutorials/una-introduccion-a-oauth-2-es.

[14] Boyd, «Getting Started with OAuth 2.0,» 2012.

[15] Argyriou, «Security Flows in OAuth 2.0 Framework: A Case Study,» 2017.

[16] Mozilla, «Generalidades del protocolo HTTP,» 2020. [En línea]. Available: https://developer.mozilla.org/es/docs/Web/HTTP/Overview.

[17] Sheldon, M. R. (2009). INTRODUCTION TO PROBABILITY AND STATISTICS FOR ENGINEERS AND SCIENTISTS.

[18] STHDA. (2020). Normality Test in R. Obtenido de http://www.sthda.com/english/wiki/normality-test-in-r

[19] Google Developers. (2020). PageSpeed Insights. Obtenido de https://developers.google.com/speed/pagespeed/insights/?hl=es

[20] Shernan, E. C. (2015). More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations.