UDC 004.9+004.056

METHODS OF AUTHENTICATION WHEN INTERACTING WITH WEB SERVICES VIA API

DZMITRY SAUCHANKA, YAUHEN SUKHAREU
Polotsk State University, Belarus

The paper discusses the methods of authentication used when interacting with third-party web services via application programming interfaces on behalf of a user. The methods provide different level of authentication credentials security, different required level of mutual trust among the parties of authentication process and different capabilities in organizing fine-grained separation of access permissions.

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems [1]. HTTP, alongside with the traditional usage of hypertext transferring and other cases, is used as a transport for RESTful services. Representational State Transfer (REST) is an architectural style to build distributed hypermedia systems [2] that is widely used to design application programming interfaces (API) for rich internet applications (RIA). Such APIs need to provide means for third-party applications to act on behalf of a user via process of user authentication. The authentication method is required to ensure highest level of user credentials security and the least possible required level of mutual trust among the parties of authentication process.

HTTP provides a simple challenge-response authentication mechanism which may be used by a server to challenge a client request and by a client to provide authentication information [1]. This is a general higher-level mechanism that allows various specific implementations. Authentication information challenging is done by a server with the *WWW-Authenticate* header and response code *401* (unauthorized). Providing authentication information by a client is achieved with the *Authorization* header.

The simplest "basic" authentication scheme is based on the model that the client authenticates itself with a user identifier and a password [1]. The user identifier is concatenated with the password separated by a single colon character and the result is encoded within a Base64 scheme. Obtained string is prepended with word "Basic" and the result forms the *Authorization* header value. Thus, to authenticate on behalf of the user with identifier "john doe" and password "0iyrB7bhzZza" the following header should be sent to a server:

Authorization: Basic am9obl9kb2U6MGl5ckI3Ymh6Wnph

This authentication scheme is non-secure in terms of authentication credentials security and is based on assumption that the connection between the client and the server can be regarded as a trusted carrier. Furthermore, the scheme requires a client application to be aware of user credentials and thus it assumes the application to be trusted to by the user. This scheme also requires a server to store user passwords in plaintext or in encrypted with reversible algorithm form. There are highly limited ways to separate access permissions with a basic authentication scheme: the separation can be only achieved by the server by requesting different user passwords for different request URIs.

The Digest Access authentication scheme allows avoiding the most serious flaws of Basic authentication [3]; this is achieved through not sending a user password in plaintext in *Authorization* header when performing authentication. Instead, a hash of the string containing username, password, and unique server-generated nonce is sent as authentication information. MD5 is typically used as the hash algorithm. In its simplest case *Authorization* header value is formed in this manner:

A1 = MD5(username:realm:password)
A2 = MD5(method:digest-uri)
Hash =MD5(A1:nonce:A2)
Authorization-header = Authorization: "Digest"
"username"=username,
"realm"=realm,
"nonce"=nonce,
"uri"=digest-uri,
"reponse"=Hash

Here

username is user identifier;

realm is a string sent by a server in *XXX-Authenticate* header and displayed by client to users so they know which username and password to use;

password is user password;

method is the HTTP request method (GET, POST etc.) used in the current request;

digest-uri is the HTTP request URI used in the current request;

nonce is a server-specified data string which should be uniquely generated by a server each time and sent in *XXX-Authenticate* header.

Thus, to authenticate on behalf of the user with identifier "john_doe" and password "0iyrB7bhzZza" given that the server returned realm *auth@example.com* and nonce *59fb925ffbc8a83d8c0993ee264a946f* and http://example.com/index.html URI is requested with HTTP GET method the following actions should be done to form authentication request:

A1 = MD5(john_doe:auth@example.com:0iyrB7bhzZza) = 4bccf28e28cc61aae0ca0c35154931f0

A2 = MD5(GET:http://example.com/index.html) = 25ed971549a69ec6d4ed9c7884d2f785

Hash = MD5(4bccf28e28cc61aae0ca0c35154931f0:59fb925ffbc8a83d8c0993ee264a946f:25ed971549a69ec6d4ed9c7884d2f785) = c6428a734599e224606b9c13c22a73ef

Authorization-header = Authorization: Digest
username=john_doe,
realm=auth@example.com,
nonce=59fb925ffbc8a83d8c0993ee264a946f,
uri=http://example.com/index.html
reponse=c6428a734599e224606b9c13c22a73ef

Digest authentication scheme cannot be considered secure though it does not assume transferring user password as a plain text and does not require storing user password in reversible form on the server side (instead, *MD5(username:realm:password)* hash can be stored). It still requires a user to trust his or her credentials to a client application and it still provides highly limited means to organize fine-grained separation of access permissions.

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to a HTTP service on behalf of a user (resource owner) without knowing authorization credentials by orchestrating an approval interaction between the resource owner and the web service [4]. This is achieved through separating the role of the client from that of the resource owner; instead of using the resource owner's credentials to access protected resources, the client obtains an access token — a string denoting a specific scope of access, lifetime, and other access attributes; the access token is then used to access the protected resources hosted by the server. General abstract OAuth 2.0 flow is presented in figure 1 and includes the following steps:

A. the client requests authorization from the resource owner (the client); the request can be made indirectly via the authorization server as an intermediary as it is done in authorization code grant type;

B. the client receives an authorization grant, which is a special credential representing the resource owner's authorization;

C. the client requests an access token by authenticating with the authorization server and presenting the authorization grant;

D. the authorization server authenticates the client and validates the authorization grant and issues an access token for valid authorization grant;

E. the client requests the protected resource from the resource server and authenticates by presenting the access token;

F. the resource server validates the access token and serves the request for valid access token.

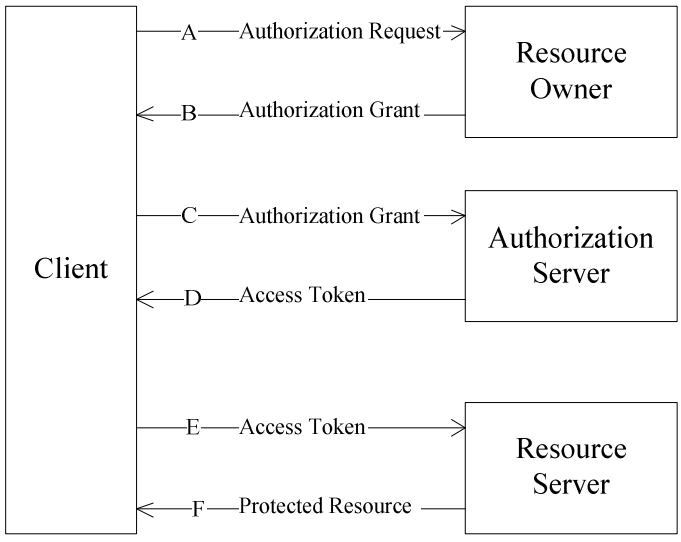

Fig. 1. OAuth 2.0 Protocol Flow

It is not necessary for authorization server and resource server to be separate entities; they may be the same server.

The authorization code grant type is used to obtain access tokens and is optimized for confidential clients [4]. This is a redirecting-based flow and thus the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server. Authorization code flow is illustrated in figure 2.

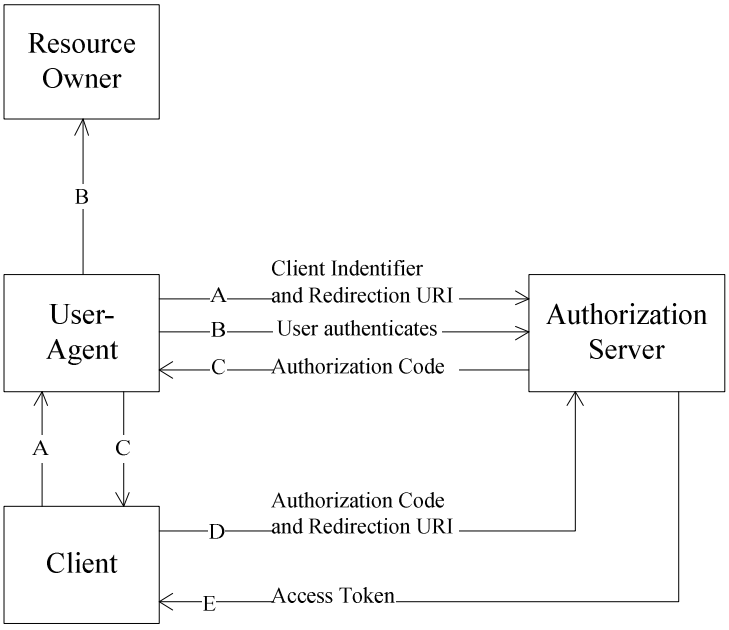

Fig. 2. Authorization Code Flow

The flow includes the following steps:

A. the client directs the resource owner's user-agent to the authorization endpoint; the client includes its client identifier, requested scope, local state, and a redirection URI for authorization server to send user-agent back to after access is granted or denied;

B. the authorization server authenticates the resource owner via the user-agent and establishes whether he or she grants or denies the client's access request;

C. assuming access has been granted by the resource owner, the authorization server redirects the user-agent to the client using the redirected URI provided earlier and including an authorization code and local state provided by the client earlier;

D. the client requests an access token from the authorization server by including the previously received authorization code and redirection URI previously used for verification;

E. the authorization server authenticates the client, validates the authorization code and redirection URI and, if valid, responds with an access token.

Obtained access token then can be used to form the *Authorization* header when accessing protected resources. "Bearer" authentication scheme is used to construct the header [5]. Assuming that access token obtained from the authorization server is *ddb2c93ebcf6098f6ccb08807ed66e02934248a5*, the following *Authorization* header should be passed to the server:

Authorization: Bearer ddb2c93ebcf6098f6ccb08807ed66e02934248a5

OAuth 2.0 authorization framework provides relatively high level of security since it does not require a user to trust his or her credentials to a client application, does not require storing of a user password in reversible form on a server, allows fine-grained separation of access permissions through means of access scopes and explicit user access grant, does not require plaintext password transferring when performing authentication since access token, bound to specific client application, is used. The framework, however, does not define any ways of secure user authentication via user-agent and it is assumed the security is provided by means of encrypted HTTP over SSL protocol.

Summary characteristics of authentication methods are presented in table 1.

OAuth 2.0 authentication method provides the maximum level of security compared to other methods, though it requires more complex client application implementation since the application should be able to interact with user-agent and receive requests via redirection.

Table 1 – Authentication methods summary characteristics

Authentication method property	Basic authentication	Digest authentication	OAuth 2.0 authentication
Client is required to have access to user credentials	Yes	Yes	No
Server is required to store user password in reversible state	Yes	No	No
Plain text credential transferring	Yes	No	No
Fine-grained separation of access permissions	Partial	Partial	Yes
Client is required to be capable of interacting with the user-agent receiving incoming requests via redirection	No	No	Yes

REFERENCES

1. RFC 1945 – Hypertext Transfer Protocol – HTTP/1.0 [Electronic Resource] / Tim Berners-Lee, Roy T. Fielding, Henrik Frystyk Nielsen. – Mode of access: http://tools.ietf.org/pdf/rfc1945.pdf. – Date of access: 15.01.2015.
2. Fielding, Roy T. Architectural Styles and the Design of Network-based Software Architectures / Roy Thomas Fielding; University of California. – Irvine, 2000. – 162 p.
3. RFC 2617 – HTTP Authentication: Basic and Digest Access Authentication [Electronic Resource] / John Franks, [et al.]. – Mode of access: http://tools.ietf.org/pdf/rfc2617.pdf. – Date of access: 15.01.2015.
4. Hardt, Dick. RFC 6749 – The OAuth 2.0 Authorization Framework [Electronic Resource] / Dick Hardt. – Mode of access: http://tools.ietf.org/pdf/rfc6749.pdf. – Date of access: 15.01.2015.
5. Jones, Michael B. RFC 6750 – The OAuth 2.0 Authorization Framework: Bearer Token Usage [Electronic Resource] / Michael B. Jones, Dick Hardt. – Mode of access: http://tools.ietf.org/pdf/rfc6750.pdf. – Date of access: 15.01.2015.

UDC 681.5

EVALUATION FUNCTION AS A KEY FACTOR IN SOLVING THE TASK OF UNIVERSITY SCHEDULE CREATION

ALEH TRAVKIN, DMITRY RUGOL
Polotsk State University, Belarus

The paper is devoted to further investigation of local evolutionary method for solving discrete optimization problems, namely, the applicability of this method to the problem of scheduling the university.

This problem is NP-hard and has a strong applied focus. It remains relevant, particularly, for universities in Belarus.[1]

Unsolved problems include the following:

a) effective restructuring of the schedule due to changes in the source data;

b) taking into account the specific constraints dictated by the organization of the process of scheduling in specific universities;

c) acceptable rate of the convergence of the optimization process of scheduling. [4]

The quality evaluation function plays the key role in the evaluation of the quality of the schedule and determining the limits of the optimization process.

The initial data for the drawing up of the educational schedule are:

$A=\{a_i\}$ – set of auditoriums;

$T=<t_i>$ – amount of time – arranged set of study time quanta (quantum – two academic hours);

$W=\{w_i\}$ – set of lecturers;

$P=\{p_i\}$ – set of classes;

$G=\{g_i\}$ – set of student groups;

$U=\{<p_i,g_i>\}PG$ – set of learning plans for student groups;