INFORMATION
SYSTEMS IN
MANAGEMENT

Information Systems in Management (2016) Vol. 5 (3) 358–369

REST API SAFETY ASSURANCE BY MEANS OF HMAC MECHANISM

GRZEGORZ NOWAKOWSKI

Department of Automatic Control and Information Technology,
Cracow University of Technology (PK)

The HMAC mechanism that enables authentication REST services and assures their integrity, non-repudiation and confidentiality, has been presented in this article. A demonstration Restful API has been implemented using Slim Framework, in which several endpoints for login, test route available only for registered users and authenticated by means of HMAC mechanism, have been assigned. The solution proposed here suggests an alternative that is easy to implement compared to other well-known methods of authentication and authorization.

Keywords: REST (Representational State Transfer), HMAC (Keyed-Hash Message Authentication Code), API (Application Programming Interface), cryptography

1. Introduction

Current applications use or share increasingly their API (Application Programming Interface) in the form of web-services. In the case where services are available to the public or for applications with limited trust, it is important to ensure the appropriate level of security of these services. The mechanisms that provide the security level are mainly based on a secret key and are commonly called message authentication codes – MAC (Message Authentication Code). They are used mainly between two parts that share a secret key to authenticate the information transmitted between these parts. In modern cryptography a shortcut function with a secret key called HMAC (Keyed-Hash Message Authentication Code),

in short, ensuring both the protection of the integrity and authenticity of data, is used as MAC codes.

The HMAC mechanism that enables authentication of REST (Representational State Transfer) services and assures their integrity, non-repudiation and confidentiality, has been presented in this article. A demonstration of Restful API has been implemented using Slim Framework, in which several endpoints for login, test route available only for registered users and authenticated by means of HMAC mechanism, have been assigned. The solution proposed here suggests an alternative that is easy to implement comparing to other well-known methods of authentication and authorization, is easy to implement.

2. Rest

REST [1] is a standard software architecture that describes how to handle queries to the API and introduces a set of good practices. REST simplifies request and response operation in a new and easier way, without resorting to complex documentation. It is based on URI addresses (Uniform Resource Identifier) and HTTP, without using an additional encapsulation such as in the SOAP (Simple Object Access Protocol) protocol, for example. The condition and functionality of the application is divided into units defined as *resources*. All *resources* use a uniform interface for changing the state, which consists of a limited set of well-defined operations and a limited set of data representation. In practice, the data are represented in JSON (JavaScript Object Notation) [5], XML (Extensible Markup Language) [6] and HTML (HyperText Markup Language) format, for example, or in text format. Additional request details are sent as HTTP header parameters.

This standard was developed by Roy T. Fielding, who wrote his doctoral dissertation on this topic [4]. Roy Fielding is one of the principal authors of the HTTP specification, an authority on computer network architecture and co-founder of the Apache HTTP Server project.

2.1. Richardson Maturity Model

The Richardson Maturity Model (presented in Figure 1) [2], developed by Leonard Richardson, describes the bases of REST in terms of resources, verbs, and hypermedia controls. The starting point for the maturity model is to use the HTTP layer as transport.

Level 0 – Remote Procedure Invocation. Level 0 includes sending data by means of SOAP or XML-RPC technology as POX (Plain Old XML). Only the POST methods are used. This is the easiest way of building SOA (Service-Oriented Architecture) application using a single POST method and XML format to communicate between services.

Level 1 – REST resources. Level 1 deals with the POST methods and the REST URIs are used instead of function and passing arguments. Only one HTTP method is being used. The advantage of level 1 in relation to level 0 is to break a complex functionality into multiple resources with the use of one POST method serving to communicate between services.

Level 2 – Additional HTTP verbs. Level 2 deals with other HTTP verbs such as GET, POST, PUT, DELETE. It represents the real use of REST technology in which different HTTP verbs are being used in order to request methods and the system can have multiple resources.

Level 3 – Hypermedia as the Engine of Application State. HATEOAS is the most mature level of Richardson's model. The responses to the clients requests contain hypermedia controls, which can help a client to decide what to do next. Level 3 encourages easy discoverability and makes it easy to understand.

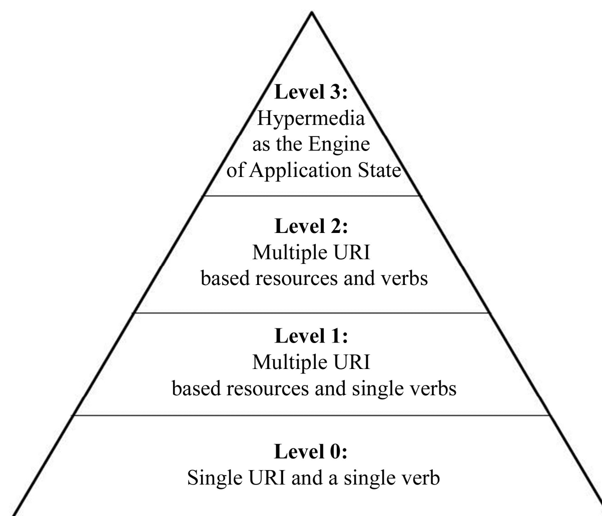

Figure 1. Richardson Maturity Model [2]

2.2. Safety and idempotence [2, 4]

A method that does not change the state on the server is considered to be a safe method. An idempotent method is a method that will produce the same results irrespective of how many times it is called. They are shown in Table 1.

The GET method is safe because it enables to get access to a resource, and thus does not change the state on the server. Requests sent by this method are cached and can contain parameters. POST and PUT methods are not safe as they can create or modify a resource on the server. The DELETE method is not safe either as it deletes a resource on the server.

The GET method is idempotent because no matter how many times the method is tested, the response is always the same. The PUT method is idempotent as well. Calling the PUT method multiple times will update the same resource and not change the outcome. POST method is not idempotent and calling it multiple times may have different results and create new resources. The DELETE method is idempotent because based on the RFC 2616, the side effects of N > 0 requests are the same as for a single request. This means that once the resource deleted, calling DELETE multiple times will get the same response.

Table 1. Safety and idempotence methods

Method	Safe		Idempotent	
	YES	NO	YES	NO
GET	X		X	
POST		X		X
PUT		X	X	
DELETE		X	X	

2.3. How did the RESTful Web Services come into existence?

There are two ways to build architecture (whether software or not):

- *from scratch*, by taking some known components, such as Slim Framework, and assembling them, with full freedom of action and creating what is needed respectively,
- *from the whole*, by building the whole and imposing some limitations.

The latter approach was adopted at creating REST concept. It started with hypertext, multimedia, WWW (World Wide Web) information system and imposed the following restrictions [4]:

- *client - server* (separation of interface and space where data are stored),
- *statelessness* (each request from the client to the server has to contain all details to understand the request. It helps to improve visibility, reliability, and scalability of requests. Visibility is improved, as the system monitoring the requests does not have to look beyond one request to obtain details. Reliability is improved as there are no check-points and resuming in case of partial failures. Scalability is improved because the number of requests that can be processed by the server increases, as the server is not responsible for storing any state information),
- *cache vulnerability* (buffering some resources we can avoid unnecessary interaction and make the application work faster),
- *uniform interface* (using: GET, POST methods, etc., gives some simplification and reliable performance increase).

2.4. Design principles for building RESTful services

The entire process of designing, developing, and testing RESTful API demo (in which several endpoints for login, test route available only for registered users and authenticated by means of HMAC mechanism, have been assigned) was divided into several stages, and then created general scheme design principles that should occur during such a process.

Stage 1 *- identifying the resource URIs*. Resources [2, 3, 4] are the key concept in REST technology. The resource represents everything that is interesting enough so that we want to relate to it. RESTful resources are identified by resource URIs. REST is extensible due to the use of URIs for identifying resources by resource URIs. REST is extensible due to the use of URIs for identifying resources. There are two types of addresses to resources:

- *collection address*, which usually ends with a descriptive name, eg. auctions,
- *a single address* item of a particular resource that usually ends with some identifier.

Table 2. Sample URIs, which can represent different resources in the system

URI	Description of the URI
/v1/users	this is used to represent all users
/v1/users/1234	this is used to represent a user in a system identified by ‘1234’
/v1/users/1234/auctions	this is used to represent all the auctions for a user identified by ‘1234’

All the preceding samples (presented in Table 2) show a clear readable pattern, which can be interpreted by the client.

Stage 2 *- identifying the methods supported by the resource* [2, 3, 4]. HTTP verbs comprise a major portion of the uniform interface constraint, which defines the association between the actions identified by the verb and the noun-based REST resource. A summary of HTTP methods and description of its actions have been presented in Table 3.

Table 3. Summary of HTTP methods and description of its actions

HTTP method	Resource URI	Description
GET	/users	gets a list of users
GET	/users/1234	gets a user identified by '1234'
POST	/users	creates a new user
PUT	/users/1234	updates a user identified by ‘1234’
DELETE	/users	deletes all users
DELETE	/users/1234	deletes a user identified by ‘1234’

HTTP verbs in the context of REST [3, 4]. HTTP verbs inform the server how to deal with the data sent as part of the URL. A summary of HTTP methods in the context of REST has been presented in Table 4.

Table 4. Summary of HTTP methods in the context of REST

HTTP method	Method description
GET	enables access to a resource. Whenever the client clicks a URL in the browser, the latter sends a GET request to the address specified by the URL. The GET requests are cached and can contain parameters.
POST	is used to create resources. Multiple invocations of the POST requests can create multiple resources.
PUT	is used to update resources. Multiple invocations of the PUT requests should produce the same results by updating the resource. The PUT requests should invalidate the cache entry if it exists.
POST versus PUT	the difference between PUT and POST methods concerns URI Request. The URI identified by POST defines the entity that handles the POST request. The URI in the PUT request includes the entity in the request. PUT and POST can both be used to create or update resources. The usage of the corresponding method depends on the idempotence behavior expected from the method as well as the location of the resource to identify it.
DELETE	is used to delete resources. The delete resource disappears and re-calling the same method multiple times does not change the outcome.

Stage 3 - *identifying the different resource representations*. RESTful resources are abstract entities that need to be serialized into a presentable format before sending to the client. The most common resource representations are XML, JSON, HTML, or plain text. A resource can provide the representation to the client depending on what the latter can handle. A client can specify their preferred languages and media types.

Sage 4 - *implementation of RESTful services using Slim Framework and authentication using HMAC mechanism*. There are many different methods or protocols that might be used (by a programmer) to secure REST API and all of them have advantages and disadvantages. One of the programmer's challenges when handling REST API security is the fact that it is a principle of REST architecture to remain stateless. The server does not maintain any record of whether or not a user is authenticated / authorized. In order to determine who sends the request (and whether it is authorized to access a particular resource) from the server side, all the information needed to operate has to be contained within the request coming from the client.

The authentication method using HMAC mechanism (proposed below) is a very good solution for securing a REST API.

Generally, the idea of HMAC is based on the fact that a client and a server know a secret key. This secret key is never sent directly during authentication.

It is only used in combination with other data elements and then transmitted. In this way, when we use a secret key and any other transmitted data: a public key that identifies the user (in the form of a header or cookie), the current Unix timestamp, or other elements that we want to use - and pass this data through encryption algorithm, such as SHA-1 (one-way hash function) for example, we can create the same hash, both on the client's and the server's side. However, the server assumes that the message is authentic and comes from the client, because only they know the key used to generate the HMAC.
HMAC uses the following parameters [10 - 13] (presented in Table 5):

Table 5. HMAC parameters

B	Block size (in bytes) of the input to the Approved hash function.
H	An Approved hash function
ipad	Inner pad; the byte x'36' repeated B times.
K	Secret key shared between the originator and the intended receiver(s).
K_0	The key K after any necessary pre-processing to form a B byte key
L	Block size (in bytes) of the output of the Approved hash function.
opad	Outer pad; the byte x'5c' repeated B times.
text	The data on which the HMAC is calculated; text does not include the padded key. The length of text is n bits, where 0 < n < 2B - 8B.
x 'N'	Hexadecimal notation, where each symbol in the string 'N' represents 4 binary bits.
//	Concatenation.
$\oplus$	Exclusive-Or operation.

Source: [10 - 13]

To compute a MAC over the data 'text' using the HMAC function, the following operation is performed:

$$MAC(text) = HMAC(K, text) = H((K0 \oplus opad) \,||\, H((K0 \oplus ipad) \,||\, text)) \quad (1)$$

Table 6. illustrates the step by step process in the HMAC algorithm.

The authentication mechanism applied to a specific example has been presented in Figure 2. Restful API has been implemented using Slim Framework, in which several endpoints for login, test route available only for registered users and authenticated by means of HMAC mechanism, have been assigned.

Slim [7] is a PHP micro framework that helps you quickly write simple yet powerful web applications and APIs. It includes routing mechanism and a simple template system, session maintenance and cookies. It allows building a website using essentially one file *index.php* and several endpoints.

Table 6. HMAC Algorithm

Steps	Description
1	If the length of *K* = B: set K_0 = K. Go to step 4.
2	If the length of *K* > B: hash *K* to obtain an *L* byte string, then append (B-L) zeros to create a B-byte string K_0 (i.e., K_0 = h(k) || 00...00). Go to step 4.
3	If the length of *K* < B: append zeros to the end of *K* to create a B-byte string K_0 (e.g., if *K* is 20 bytes in length and *B* = 64, then *K* will be appended with 44 zero bytes x'00').
4	Exclusive-Or K_0 with *ipad* to produce a B-byte string: $K_0 \oplus ipad$.
5	Append the stream of data *'text* to the string resulting from step 4: $(K_0 \oplus ipad) \| text$.
6	Apply *H* to the stream generated in step 5: $H((K_0 \oplus ipad) \| text)$.
7	Exclusive-Or K_0 with *opad*: $K_0 \oplus opad$.
8	Append the result from step 6 to step 7: $(K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text)$.
9	Apply *H* to the result from step 8: $H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))$.

Source: [10 - 13]

STEP 1: A client and a server know a secret key. This secret key is never directly sent during authentication. On the client side, a login form using JavaScript has been created, which contains two fields: username and password (after typing encrypted). On the server side, an endpoint has been set [*POST /login*], which will be used to verify whether the specified user login and encrypted password are in the SQLite database. If the user verification is successful, the public key (*apiPublicKey*) will be retrieved from the database (individual for each user) on the basis of username and encrypted password, and it will be sent to the client.

This public key will be used to identify the user. Conversely, if the verification fails, the information will be returned: Access denied. Data from the login form will be sent to the server endpoint [*POST /login*] by means of jQuery mechanisms.

During transmission of a public key via the server, it is possible that an unauthorized person (hacker) using tools such as Firesheep (or something similar) for example, might be able to sniff network traffic and steal the key. However, this unauthorized person will not have the secret API key and thus will not be able to recreate the same HMAC HASH, which might be done by the client and the server.

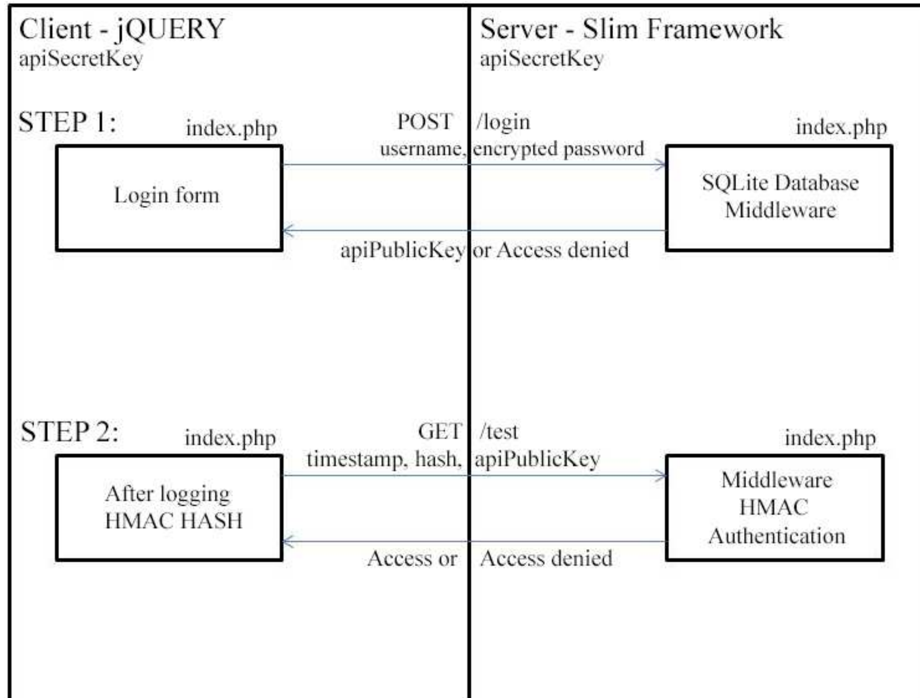

Figure 2. HMAC authorization

STEP 2*:* HMAC HASH will be generated after correct verification and logging on the client side. A particular function and the following parameters were implemented in this purpose:

- current UNIX timestamp - essentially, to determine whether the data sent to the API have been sent in a short period of time. JavaScript does not have a built-in function that retrieves time so such function has been implemented,
- user public key (*apiPublicKey*) returned in the previous step by the server,
- secret key (*apiSecretKey*) only known to the client and server,
- Javascript library - *cryptoJS*, which offers encryption and hashing algorithms (in this example a one-way hash function SHA1 has been used).

Then the following data will be sent from the client (*within the single HTTP request*) to the server side: HMAC generated HASH, the user's public key, the current time UNIX.

On the server side the endpoint, [*GET /test*] which will receive the data sent by the client, has been set.

It is worth mentioning that Slim Framework serving to authenticate requests uses middleware. Basically, this is the code that executes before the request actually reaches the intended route. As it is described in the Slim Framework middleware

documentation [8], when the middleware class runs the entry point, the *call()* method is activated. The first thing being examined by this method is whether the specific route is on a list of 'allowed routes'. Any routes that we want to declare as 'open', that is not needing authentication, can be added to the array in the class constructor.

The route [*GET /test*] is available only by authenticated user on the server side. In the *call()* method the verification, whether the timestamp sent in a header (current Unix time) has been sent within a given period of time, takes place.

If not, the access is blocked immediately. If so, then the server generates on its side HMAC HASH. A function available in PHP *hash_hmac()* [9] has been used for this purpose, and the following parameters were transferred:

- current UNIX timestamp,
- user public key (*apiPublicKey*) sent by the client,
- secret key (*apiSecretKey*) only known to the client and server.

If the HMAC HASH generated by the server corresponds to the HMAC HASH generated on the client side, the request is trusted and, in consequence, we get full access to it.

***Stage 5** – testing the RESTful services* [2]. There are different ways to access the REST resources and testing them by clients. We may use a cURL tool or Postman.

cURL is a command-line tool for testing REST APIs. The cURL library and the cURL command give the user the possibility to create requests and explore the response.

Postman is an alternative tool that may be installed in addition to Chrome. It includes a JSON and XML viewer for rendering the data. It allows previewing HTTP 1.1 requests as well as replaying, and organizing requests for the future use. Postman shares the same environment as the browser and can display browser cookies, too.

An advantage of Postman in comparison with cURL is a user-friendly interface for entering parameters so that the user does not need to deal with commands or scripts. Moreover, various authorization schemes such as basic user authentication and digest access authentication are operated by Postman.

3. Conclusion

One weakness of many web-services that require authentication is that the username and password of the user making the request are simply included as request parameters [15]. Alternatively, some use basic authentication, which transmits the username and password in an HTTP header encoded using Base64. Basic authentication obscures the password, but does not encrypt it. There is

a better way - using a HMAC (Keyed-Hash Message Authentication Code) to sign service requests with a secret key. HMAC provides the server and the client each with a public and secret key. The public key is known, but the secret key is known only to that specific server and that specific client [14]. The client creates a unique HMAC, or hash, per request to the server by combing the request data and hashing that data, along with a secret key and sending it as part of a request. The server receives the request and regenerates its own unique HMAC. The server compares the two HMACs, and, if they're equal, the client is trusted and the request is executed. What makes HMAC more secure than MAC (Message Authentication Code) is that the key and the message are hashed in separate steps.

There are two big advantages. The first is that the HMAC allows to verify the password (or secret key) without requiring the user to embed it in the request, and the second is that the HMAC also verifies the basic integrity of the request. If an unauthorized person manipulated the request in any way in transit, the signatures would not match and the request would not be authenticated [15].

The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output, and on the size and quality of the key. This means that completely random key, where every bit is randomly generated, is far better than set of characters. The optimum size of the key is equal to block size. If the key is too short then it is padded usually with zeroes (which are not random). If the key is too long then its hash function is used. The length of hash output is anyway block size.

The solution proposed in this article suggests an alternative that is easy to implement compared to other well-known methods of authentication and authorization.

REFERENCES

[1] Webber J., Parastatidis S., Robinson I. (2010) *REST in Practice: Hypermedia and Systems Architecture*, O'Reilly Media, 1 edition.

[2] Mehta B. (2014) *RESTful Java Patterns and Best Practices*, Packt Publishing.

[3] Richardson L., Amundsen M, Ruby S. (2013) *RESTful Web APIs*, O'Reilly Media.

[4] Fielding R.T. (2000) *Architectural Styles and the Design of Network-based Software Architectures*, Chapter 5, Dissertation, University Of California, Irvine.

[5] JSON, (online) homepage: http://json.org/ (date of access: 2016-02-05)

[6] XML, (online) homepage: http://www.w3.org/XML/ (date of access: 2016-02-05)

[7] Slim Framework, a micro framework for PHP (online) homepage: http://www.slimframework.com/ (date of access: 2016-02-05)

[8] Slim Framework, Middleware-Overview (online) homepage: http://docs.slimframework.com/#Middleware-Overview (date of access: 2016-02-05)

[9] hash_hmac(), (online) homepage: http://php.net/manual/en/function.hash-hmac.php (date of access: 2016-02-05)

[10] Krawczyk H., Bellare M., and Canetti R. (1997) *HMAC: Keyed-Hashing for Message Authentication*, Internet Engineering Task Force, Request for Comments (RFC) 2104.

[11] National Institute of Standards and Technology (2008) *Secure Hash Standards (SHS)*, Federal Information Processing Standards Publication 180-3.

[12] NIST Special Publication (SP) 800-57 (2007) *Recommendation for Key Management – Part 1: General (Revised).*

[13] NIST Special Publication (SP) 800-107 (2009) *Recommendation for Applications Using Approved Hash Algorithms*.

[14] Hash-based Message Authentication Code (HMAC) definition, (online) homepage: http://searchsecurity.techtarget.com/definition/Hash-based-Message-Authentication-Code-HMAC (date of access: 2016-02-05)

[15] Using HMAC to authenticate Web service requests, (online) homepage: http://rc3.org/2011/12/02/using-hmac-to-authenticate-web-service-requests/ (date of access: 2016-02-05)