REST API Token-based Authentication

Asked 12 years, 9 months ago Modified 9 years ago Viewed 191k times

126

I'm developing a REST API that requires authentication. Because the authentication itself occurs via an external webservice over HTTP, I reasoned that we would dispense tokens to avoid repeatedly calling the authentication service. Which brings me neatly to my first question:

Is this really any better than just requiring clients to use HTTP Basic Auth on each request and caching calls to the authentication service server-side?

The Basic Auth solution has the advantage of not requiring a full round-trip to the server before requests for content can begin. Tokens can potentially be more flexible in scope (i.e. only grant rights to particular resources or actions), but that seems more appropriate to the OAuth context than my simpler use case.

Currently tokens are acquired like this:

```
curl -X POST localhost/token --data
"api_key=81169d80...

&verifier=2f5ae51a...

&timestamp=1234567
```

```
                                        &user=foo
                                        &pass=bar"
```

The `api_key`, `timestamp` and `verifier` are required by all requests. The "verifier" is returned by:

```
sha1(timestamp + api_key + shared_secret)
```

My intention is to only allow calls from known parties, and to prevent calls from being reused verbatim.

Is this good enough? Underkill? Overkill?

With a token in hand, clients can acquire resources:

```
curl localhost/posts?api_key=81169d80...
                    &verifier=81169d80...
                    &token=9fUyas64...
                    &timestamp=1234567
```

For the simplest call possible, this seems kind of horribly verbose. Considering the `shared_secret` will wind up being embedded in (at minimum) an iOS application, from which I would assume it can be extracted, is this even offering anything beyond a false sense of security?

authentication **rest**

Share

Improve this question

Follow

asked Mar 19, 2012 at 16:09

cantlin

3,226 ● 3 ● 22 ● 22

3 Instead of using sha1(timestamp+api_key+shard_secret) you should use hmac(shared_secret, timpestamp+api_key) for a better security hashing en.wikipedia.org/wiki/Hash-based_message_authentication_code – Miguel A. Carrasco Oct 27, 2013 at 2:17

@MiguelA.Carrasco And in seems to be the consensus in 2017 that bCrypt is the new hashing tool. – Shawn May 17, 2017 at 15:54

3 Answers

Sorted by: Highest score (default)

96

Let me seperate up everything and solve approach each problem in isolation:

Authentication

For authentication, baseauth has the advantage that it is a mature solution on the protocol level. This means a lot of *"might crop up later"* problems are already solved for you. For example, with BaseAuth, user agents know the password is a password so they don't cache it.

Auth server load

If you dispense a token to the user instead of caching the authentication on your server, you are still doing the same

thing: Caching authentication information. The only difference is that you are turning the responsibility for the caching to the user. This seems like unnecessary labor for the user with no gains, so I recommend to handle this transparently on your server as you suggested.

Transmission Security

If can use an SSL connection, that's all there is to it, the connection is secure*. To prevent accidental multiple execution, you can filter multiple urls or ask users to include a random component ("nonce") in the URL.

```
url = username:key@myhost.com/api/call/nonce
```

If that is not possible, and the transmitted information is not secret, I recommend securing the request with a hash, as you suggested in the token approach. Since the hash provides the security, you could instruct your users to provide the hash as the baseauth password. For improved robustness, I recommend using a random string instead of the timestamp as a "nonce" to prevent replay attacks (two legit requests could be made during the same second). Instead of providing seperate "shared secret" and "api key" fields, you can simply use the api key as shared secret, and then use a salt that doesn't change to prevent rainbow table attacks. The username field seems like a good place to put the nonce too, since it is part of the auth. So now you have a clean call like this:

```
nonce = generate_secure_password(length: 16);
one_time_key = nonce + '-' +
sha1(nonce+salt+shared_key);
url = username:one_time_key@myhost.com/api/call
```

It is true that this is a bit laborious. This is because you aren't using a protocol level solution (like SSL). So it might be a good idea to provide some kind of SDK to users so at least they don't have to go through it themselves. If you need to do it this way, I find the security level appropriate (just-right-kill).

Secure secret storage

It depends who you are trying to thwart. If you are preventing people with access to the user's phone from using your REST service in the user's name, then it would be a good idea to find some kind of keyring API on the target OS and have the SDK (or the implementor) store the key there. If that's not possible, you can at least make it a bit harder to get the secret by encrypting it, and storing the encrypted data and the encryption key in seperate places.

If you are trying to keep other software vendors from getting your API key to prevent the development of alternate clients, only the encrypt-and-store-seperately approach *almost* works. This is whitebox crypto, and to date, no one has come up with a truly secure solution to problems of this class. The least you can do is still issue a single key for each user so you can ban abused keys.

(*) EDIT: *SSL connections [should no longer be considered secure](#) without [taking additional steps to verify](#) them.*

Share Improve this answer Follow

edited Apr 10, 2014 at 17:28

answered Mar 19, 2012 at 17:09

cmc

4,413 ● 2 ● 38 ● 37

Thanks cmc, all good points and great food for thought. I've ended up taking a token/HMAC approach similar to the one you discussed above, rather like the S3 REST API authentication mechanism. – cantlin Mar 26, 2012 at 18:00

If you cache the token on the server, then isn't it essentially the same as the good old session id? Session id is short-lived and it is also attached to fast cache storage (if you implement it) to avoid hitting your DB on every request. True RESTful & stateless design should not have sessions, but if you are using a token as an ID and then still hitting the DB, then wouldn't it better just use session ID instead? Alternatively, you can go for JSON web tokens that contain encrypted or signed information for entire session data for true stateless design. – JustAMartin Sep 8, 2017 at 7:16

15

A pure RESTful API should use the underlying protocol standard features:

1. For HTTP, the RESTful API should comply with existing HTTP standard headers. Adding a new



HTTP header violates the REST principles. Do not re-invent the wheel, use all the standard features in HTTP/1.1 standards - including status response codes, headers, and so on. RESTFul web services should leverage and rely upon the HTTP standards.

2. RESTful services MUST be STATELESS. Any tricks, such as token based authentication that attempts to remember the state of previous REST requests on the server violates the REST principles. Again, this is a MUST; that is, if you web server saves any request/response context related information on the server in attempt to establish any sort of session on the server, then your web service is NOT Stateless. And if it is NOT stateless it is NOT RESTFul.

Bottom-line: For authentication/authorization purposes you should use HTTP standard authorization header. That is, you should add the HTTP authorization / authentication header in each subsequent request that needs to be authenticated. The REST API should follow the HTTP Authentication Scheme standards.The specifics of how this header should be formatted are defined in the RFC 2616 HTTP 1.1 standards – section 14.8 Authorization of RFC 2616, and in the RFC 2617 HTTP Authentication: Basic and Digest Access Authentication.

I have developed a RESTful service for the Cisco Prime Performance Manager application. Search Google for the REST API document that I wrote for that application for more details about RESTFul API compliance here. In that implementation, I have chosen to use HTTP "Basic"

Authorization scheme. - check out version 1.5 or above of that REST API document, and search for authorization in the document.

Share Improve this answer Follow

edited Sep 2, 2014 at 16:24

answered Sep 2, 2014 at 2:13

Rubens Gomes
482 ● 5 ● 10

9 *"Adding a new HTTP header violates the REST principles"* How so? And if you are at it you may be so kind to explain what exactly is the difference (regarding principles) between a password that expires after a certain period and a token that expires after a certain period. – a better oliver Nov 27, 2015 at 10:49

6 User name + password is a token(!) that is exchanged between a client and a server on every request. That token is maintained on the server and has a time-to-live. If the password expires I have to acquire a new one. You seem to associate "token" with "server session", but that's an invalid conclusion. It's even irrelevant because it would be an implementation detail. Your classification of tokens other than user name / password as being stateful is purely artificial, imho. – a better oliver Nov 28, 2015 at 14:56

1 I think you should motivate why to do an implementation with RESTful over Basic Authentication which is part of the original question. Maybe you could also link to some good examples with code included. As a beginner in this subject the theory seems clear enough with lots of good resources but the implementation method is not and the examples are convoluted. I find it frustrating that it seems to take custom coding to implement in a timely manner something that has

been done thousands of times. – JPK Sep 14, 2016 at 9:53

15 -1 "Any tricks, such as token-based authentication that attempt to remember the state of previous REST requests on the server violates the REST principles." **token based authentication has nothing to do with the state of previous REST requests and does not violate statelessness of REST**. – Kerem Baydoğan Oct 24, 2016 at 13:25

1 So, according to this, JSON Web Tokens are REST violation because they can store user's state (claims)? Anyway, I prefer to violate REST and use good old session ID as a "token", but initial authentication is performed with username+pass, signed or encrypted using shared secret and very short-lived timestamp (so it fails if anybody tries to replay that). In an "enterprise-ish" application it is difficult to throw away session benefits (avoiding hitting the database for some data needed in almost every request), so sometimes we have to sacrifice true statelessness. – JustAMartin Sep 8, 2017 at 7:26

2

In the web a stateful protocol is based on having a temporary token that is exchanged between a browser and a server (via cookie header or URI rewriting) on every request. That token is usually created on the server end, and it is a piece of *opaque* data that has a certain time-to-live, and it has the sole purpose of identifying a specific web user agent. That is, the token is temporary, and becomes a STATE that the web server has to maintain on behalf of a client user agent during the duration of that conversation. Therefore, the communication using a token in this way is STATEFUL.

And if the conversation between client and server is STATEFUL it is not RESTful.

The username/password (sent on the Authorization header) is usually persisted on the database with the intent of identifying a user. Sometimes the user could mean another application; however, the username/password is *NEVER* intended to identify a specific web client user agent. The conversation between a web agent and server based on using the username/password in the Authorization header (following the HTTP Basic Authorization) is STATELESS because the web server front-end is not creating or maintaining any STATE *information* whatsoever on behalf of a specific web client user agent. And based on my understanding of REST, the protocol states clearly that the conversation between clients and server should be STATELESS. Therefore, if we want to have a true RESTful service we should use username/password (Refer to RFC mentioned in my previous post) in the Authorization header for every single call, NOT a sension kind of token (e.g. Session tokens created in web servers, OAuth tokens created in authorization servers, and so on).

I understand that several called REST providers are using tokens like OAuth1 or OAuth2 accept-tokens to be be passed as "Authorization: Bearer " in HTTP headers. However, it appears to me that using those tokens for RESTful services would violate the true STATELESS meaning that REST embraces; because those tokens are

temporary piece of data created/maintained on the server side to identify a specific web client user agent for the valid duration of a that web client/server conversation. Therefore, any service that is using those OAuth1/2 tokens should not be called REST if we want to stick to the TRUE meaning of a STATELESS protocol.

Rubens

edited Nov 28, 2015 at 16:36

community wiki
6 revs
Rubens Gomes