

The Buzz Media

Video Games, Movies and Technology

[Home](#) [Reviews](#) [Guides](#) [Software](#) [Advertise With Us](#) [Authors](#) [Contact](#)

[Subscribe by Email](#) [Subscribe to RSS](#)
[Subscribe by Email](#) [Subscribe to RSS](#)



Designing a Secure REST (Web) API without OAuth

by [Riyad Kalla](#) on APRIL 26, 2011 in [PROGRAMMING](#)

Situation

You want to develop a RESTful web API for developers that is secure to use, but doesn't require [the complexity of OAuth](#) and takes a simple "*pass the credentials in the query*" approach... or something equally-as-easy for people to use, but it needs to be **secure**.

You are a smart guy, so you start to think...

Problem

You realize that [literally passing the credentials over HTTP](#) leaves that data open to being sniffed in plain-text; After the [Gawker incident](#), you realize that plain-text or [weakly-hashed anything](#) is usually a bad idea.

You realize that hashing the password and sending the hash over the wire in lieu of the plain-text password still gives people sniffing *at least* the username for the account and a hash of the password that could (in a disturbing number of cases) be looked up in a [Rainbow Table](#).

That's not good, so you scratch your head some more...

Then you realize that a lot of [popular public APIs](#) seem to use a combination of two values passed along with each command request: one public value and one (hopefully) private value that only the account owner is *suppose* to know.



"Still not quite right!" you exclaim, because in this case (which is really a username/password scenario all over again) you still suffer from the same problems (sniffed traffic) that sending the username and password in plain text had.

At this point you are *about* to give up and concede to using OAuth, but you insist that there has to be a secure **but** relatively easy way to design a public web API that can keep credentials private.

Solution

After doing [Peyote](#) for 2 days straight (*you should find better ways to relax*) it finally dawns on you: [Amazon Web Services](#) has one of the [largest](#) and [most used](#) web APIs online right now, and they don't support OAuth at all!

Search...



POPULAR POSTS

- [Class Action Lawsuit against Sony for "Green ...](#)
- [Patrick's 20 Favorite Action Movies](#)
- [Designing a Secure REST \(Web\) API without OAuth...](#)
- [How to Install iPhone 2.2.1 Firmware on Jailb...](#)
- [Gears of War PC Problems](#)
- [Monkey Has His Way With a Frog](#)
- [Samsung TV Capacitor "Clicking" Issue and Fre...](#)
- [DDR2-800 vs DDR3-1333, Does Speed Matter?](#)
- [PS3 + HDMI = Black Screen](#)
- [How to Install iPhone 2.1 Firmware on Hacked ...](#)
- [Halo 3 Screenshots Analyzed](#)
- [Guitar Hero: World Tour Compatible with Rock ...](#)
- [Netflix Throttling Instant Video Streaming Pe...](#)
- [God, I Hate These Smarmy Atheists!](#)
- [FourSquare Mayor Checkin Program w/ Source Co...](#)
- [Gears of War Movie Treatment Summary](#)
- [HP Printer Ink Class Action Lawsuit](#)
- [How to Add Page-Link Tag \(Multiple Page Post ...](#)



After a long afternoon of fever-dreams, you finally come down enough to see how Amazon keeps it's [API requests secure](#).

You aren't sure why, but after reading the entire page on how to assemble a request for an AWS service, it still doesn't make total sense to you. What's with this "signature" thing? What is the *data* argument in the code examples?

So you keep searching for articles on "[secure API design](#)"...

You come across other people, asking the [exact same question](#) and see them [getting excellent replies](#) that point at this "[HMAC](#)" thing... or something, you aren't sure yet.

You find [other articles](#) that encourage you to use "[HMAC](#)" and you are H-FINE using it, if someone would H-EXPLAIN it in plain H-ENGLISH!

You do run across [a distillation of the basic concept](#) that makes sense (yay!) and it goes something like this in plain English:

A server and a client know a public and private key; only the server and client know the private key, but everyone can know the public key... who cares what they know.

A client creates a unique HMAC (hash) representing its request to the server. It does this by combining the request data (arguments and values or XML/JSON or whatever it was planning on sending) and hashing the blob of request data along with the private key.

The client then sends that HASH to the server, along with all the arguments and values it was going to send anyway.

The server gets the request and re-generates its own unique HMAC (hash) based on the submitted values using the same methods the client used.

The server then compares the two HMACs, if they are equal, then the server trusts the client, and runs the request.

That seems pretty straight forward. What was confusing you originally is that you thought the original request was being *encrypted* and sent, but really all the HMAC method does is create some unique **checksum** (hash) out of the arguments using a private key that only the client and server know.

Then it sends the checksum along with the original parameters and values to the server, and then the server **double-checks** the checksum (hash) to make sure it agrees with what the client sent.

Since, hypothetically, only the client and server know the private key, we assume that if their hashes match, then they can both trust each, so the server then processes the request normally.

You realize that in real-life, this is basically like someone coming up to you and saying: "*Jimmy told me to tell you to give the money to Johnny Two-toes*", but you have no idea who this guy is, so you hold out your hand and test him to see if he knows **the secret handshake**.

If he does, then he *must* be part of your gang and you do what he says... if he doesn't

Privacy Information

RECENT COMMENTS

- Wilma on [OtterBox iPhone 4 Defender Series Case Review](#)
- Gentlemen Radio: 61 | NERD TO THE CORE on [The Gentlemen Radio Episode 61: We Didn't Go To SDCC](#)
- Brian Ford on [WARNING: EdgeStar, CompactAppliance.com and Living Direct are a Terrible Shopping Experience](#)
- Highland Rock on [WARNING: EdgeStar, CompactAppliance.com and Living Direct are a Terrible Shopping Experience](#)
- Steve C on [Netflix Throttling Instant Video Streaming Performance for Viewers](#)
- Karl on [Netflix Throttling Instant Video Streaming Performance for Viewers](#)
- squarism on [The Triad Rises Brother.](#)
- Annie on [WARNING: EdgeStar, CompactAppliance.com and Living Direct are a Terrible Shopping Experience](#)
- Luciano on [WARNING: EdgeStar, CompactAppliance.com and Living Direct are a Terrible Shopping Experience](#)
- Nick in Castle Pines, CO on [Samsung TV Capacitor "Clicking" Issue and Free Repair](#)

CATEGORIES

- Entertainment (148)
- Humor & Fun (562)
- Life & World (392)
- Movies (241)
- Podcast (62)
- Programming (105)
- Shopping (67)
- Technology (1161)
- Uncategorized (15)
- Video Games (870)

STUFF WE LIKE

- Agoraquest
- AJAX Tools
- At Home with Cats

know the secret handshake, you decide to shoot him in the face (*you have anger issues*).



You sort of get it, but then you wonder: "*What is the best way to combine all the parameters and values together when creating the giant blob?*" and luckily the guy behind [tarasnaps](#) has your back and explains to you how [Amazon screwed this up with Signature Version 1](#).

Now you re-read how Amazon Web Services does authentication and it makes sense, it goes something like:

1. [CLIENT] Before making the REST API call, [combine a bunch of unique data together](#) (*this is typically all the parameters and values you intend on sending, it is the “data” argument in the code snippets on AWS’s site*)
2. [CLIENT] Hash ([HMAC-SHA1 or SHA256](#) preferably) the blob of data data (from Step #1) with your private key assigned to you by the system.
3. [CLIENT] Send the server the following data:
 - a. Some user-identifiable information like an “API Key”, client ID, user ID or something else it can use to identify who you are. This is the **public API key**, never the private API key. This is a public value that anyone (even evil masterminds can know and you don’t mind). It is just a way for the system to know WHO is sending the request, not if it should trust the sender or not (it will figure that out based on the HMAC).
 - b. Send the HMAC (hash) you generated.
 - c. Send all the data (parameters and values) you were planning on sending anyway. Probably unencrypted if they are harmless values, like “`mode=start&number=4&order=desc`” or other operating nonsense. If the values are private, you’ll need to encrypt them.
4. (**OPTIONAL**) *The only way to protect against “replay attacks” on your API is to include a timestamp of time kind along with the request so the server can decide if this is an “old” request, and deny it. The timestamp must be included into the HMAC generation (effectively stamping a created-on time on the hash) in addition to being checked “within acceptable bounds” on the server.*
5. [SERVER] Receive all the data from the client.
6. [SERVER] (see **OPTIONAL**) Compare the current server’s timestamp to the timestamp the client sent. Make sure the difference between the two timestamps is within an acceptable time limit (5-15mins maybe) to hinder [replay attacks](#).
 - a. **NOTE:** Be sure to compare the same timezones and watch out for issues that pop up with [daylight savings time](#) change overs.
 - b. **UPDATE:** As correctly pointed out by a few folks, just [use UTC time](#) and forget about the DST issues.
7. [SERVER] Using the user-identifying data sent along with the request (e.g. API Key) look the user up in the DB and load their private key.
8. [SERVER] Re-combine the same data together that the client did in the same way the client did it. Then hash (generate HMAC) that data blob using the private key you looked up from the DB.
 - a. (see **OPTIONAL**) If you are protecting against replay attacks, include the timestamp **from the client** in the HMAC re-calculation on the server. Since you already determined this timestamp was within acceptable bounds to be accepted, you have to re-apply it to the hash calculation to make sure it was

- [Binary JSON Spec](#)
- [HN Notify](#)
- [imgscalr.com](#)
- [Katie Mullaly](#)
- [TestSeek](#)
- [Today I Found Out](#)
- [Was it Up?](#)

ARCHIVES

- [August 2013](#)
- [July 2013](#)
- [June 2013](#)
- [May 2013](#)
- [April 2013](#)
- [March 2013](#)
- [February 2013](#)
- [January 2013](#)
- [December 2012](#)
- [November 2012](#)
- [October 2012](#)
- [September 2012](#)
- [August 2012](#)
- [July 2012](#)
- [June 2012](#)
- [May 2012](#)
- [April 2012](#)
- [March 2012](#)
- [February 2012](#)
- [January 2012](#)
- [December 2011](#)
- [November 2011](#)
- [October 2011](#)
- [September 2011](#)
- [August 2011](#)
- [July 2011](#)
- [June 2011](#)
- [May 2011](#)
- [April 2011](#)
- [March 2011](#)
- [February 2011](#)
- [January 2011](#)
- [December 2010](#)
- [November 2010](#)
- [October 2010](#)
- [September 2010](#)
- [August 2010](#)
- [July 2010](#)
- [June 2010](#)
- [May 2010](#)
- [April 2010](#)
- [March 2010](#)
- [February 2010](#)
- [January 2010](#)
- [December 2009](#)
- [November 2009](#)
- [October 2009](#)
- [September 2009](#)
- [August 2009](#)
- [July 2009](#)
- [June 2009](#)

the same timestamp sent from the client originally, and not a made-up timestamp from a [man-in-the-middle attack](#).

9. [SERVER] Run that mess of data through the HMAC hash, **exactly** like you did on the client.
10. [SERVER] Compare the hash you just got on the server, with the hash the client sent you; if they match, then the client is considered legit, so process the command. Otherwise reject the command!

REMINDER: Be consistent and careful with how you combine all parameters and values together. Don't do what Amazon did with Auth Signature version 1 and [open yourself up to hash-collisions!](#) (Suggestion: just hash the whole URL-encoded query string!)

SUPER-REMINDER: Your private key should **never** be transferred over the wire, it is just used to generate the HMAC, the server looks the private key back up itself and recalculates its own HMAC. The public key is the only key that goes across the wire to identify the user making the call; it is OK if a nefarious evil-doer gets that value, because it doesn't imply his messages will be trusted. They still have to be hashed with the private key and hashed in the same manner both the client and server are using (e.g. prefix, postfix, multiple times, etc.)

Update 10/13/11: Chris correctly [pointed out](#) that if you don't include the URI or HTTP method in your HMAC calculation, it leaves you open to more hard-to-track man-in-the-middle attacks where an attacker could modify the endpoint you are operating on as well as the HTTP method... for example change an HTTP POST to /issue/create to /user/delete. Great catch Chris!

Denouement

It's been a long few days, but you finally figured out a secure API design and you are proud of yourself. You are *super-extra* proud of yourself because the security method outlined above actually protects against another commonly popular way of hacking API access: [side-jacking](#).

Session sidejacking is where a man-in-the-middle sniffs network traffic and doesn't steal your credentials, but rather steals the temporary Session ID the API has given you to authenticate your actions with the API for a temporary period of time (e.g. 1hr). With the method above, because the individual methods themselves are checksummed, there is no Session ID to steal and re-use by a nefarious middle man.

You rock.

You also slowly realize and accept that at some point you will [have to implement OAuth](#), but it will probably be [OAuth 2.0 support](#) and that [isn't quite ready yet](#).

I am relatively new to the RESTful API game, focusing primarily on [client-side libraries](#). If I missed something please point it out and I'll fix it right up. If you have questions, suggestions or ideas that you think should go into the story above, please leave a comment below.

Alternatively you can email me and we can talk about friendship, life and canoeing.

Gotchas (Problems to Watch For)

<This section was removed, because by [using UTC time](#) you avoid the daylight-savings-time issue all together and my solution proposed here was stupid anyway.>

Additional Thoughts for APIs

- May 2009
- April 2009
- March 2009
- February 2009
- January 2009
- December 2008
- November 2008
- October 2008
- September 2008
- August 2008
- July 2008
- June 2008
- May 2008
- April 2008
- March 2008
- February 2008
- January 2008
- December 2007
- November 2007
- October 2007
- September 2007
- August 2007
- July 2007
- June 2007
- May 2007
- April 2007
- March 2007
- February 2007
- January 2007
- December 2006
- November 2006
- October 2006
- September 2006
- August 2006
- July 2006
- June 2006

 CLICKY ANALYTICS

What about the scenario where you are writing a public-facing API like Twitter, where you might have a mobile app deployed on thousands of phones and you have your public and [private keys embedded in the app?](#)

On a rooted device, those users could likely decompile your app and pull your private key out, doesn't that leave the private key open to being compromised?

Yes, yes it does.

So what's the solution?

Taking a hint from Twitter, it looks like to some degree you *cannot avoid this*. Your app needs to have its private key (they call it a secret key) and that means you are open to getting your private key compromised.

What you can do though is to issue private keys on a **per-application-basis**, instead of on a per-user-account basis. That way if the private key is compromised, that *version* of the application can be banned from your API until new private keys are generated, put into an updated version of the app and re-released.

What if the new set of keys get compromised *again*?

Well yes, that is very possible. You would have to combat this in some way on your own, like encrypting the keys with *another* private key... or praying to god people will stop hacking your software.

Regardless, you would have to come up with some 2nd layer of security to protect that new private key, but at least there is a way to get the apps deployed in the wild working again (new version) instead of the root account being locked and NONE of the apps being able to access the service again.

Update #1: There are some fantastic feedback and ideas on securing a web-API down in the comments, I would highly recommend reading them.

Some highlights are:

- Use “[nonce](#)” (1-time-use-server-generated) tokens to stop replay attacks AND [implement idempotency in your API](#).
- The algorithm above is “[95% similar to ‘two-legged’ OAuth 1.0](#)”, so maybe look at that.
- Remove all the security complexity by [sending all traffic to go over SSL \(HTTPS\)](#)!

Update #2: I have since looked at “2-legged OAuth” and it is, as a few readers pointed out, almost exactly the process described above. The advantage being that if you write your API to this spec, there are plenty of [OAuth client libraries](#) available for implementors to use.

The only OAuth-specific things of note being:

- OAuth spec is [super-specific](#) with how you need to encode your params, order them and then combine them all together when forming the HMAC (called the “method signature” in OAuth)
- OAuth, when using HMAC-SHA1 encoding, requires that you send along a *nonce*. The server or “provider” must keep the nonce value along with the timestamp associated with the request that used that nonce on-record to verify that no other requests come in with the SAME nonce and timestamp (indicating a “replay” attempt). Naturally you can expire these values from your data store eventually, but it would probably be a good idea to keep them on-file for a while.
 - The nonce doesn’t need to be a secret. It is just a way to associate some

unique token to a particular timestamp; the combination of the two are like a thumbprint saying “at 12:22pm a request with a nonce token of HdjS872djas83 was received”. And since the **nonce and timestamp** are included in the HMAC hash calculation, no nefarious middle-man can ever try and “replay” that previous message AND successfully hash his request to match yours without the server seeing the same timestamp + nonce combination come back in; at which point it would say “Hey! A request with this thumbprint showed up two hours ago, what are you trying to do?”

- Instead of passing all this as GET params, all these values get jammed into one giant “Authorization” HTTP header and coma-separated.

That is pretty much the high points of 2-legged OAuth. The HMAC generation using the entire request and all the params is still there, sending along the timestamp and a nonce is still there and sending along the original request args are all still there.

When I finally get around to implementing 2-legged OAuth from a server perspective, I'll write up another article on it.

Related Stories



[QN: imgscalr Java Image-Scaling Library 2.0 Released](#)

[Explanations of Common Java Exceptions](#)

[kallasoft SmugMug Java API - Beta 3 Released](#)

[SmugMug Java API Download Links Fixed](#)

[Added a SmugMug Java API FAQ](#)

[D-Link DIR-655 Updated Firmware Hijacks Your DNS](#)

Tags: [Amazon Web Services API](#), [API](#), [authentication](#), [credentials](#), [guide](#), [HMAC](#), [HMAC-SHA1](#), [HTTPS](#), [nonce](#), [OAuth](#), [Public / Private Key](#), [security](#), [session sidejacking](#), [SSL](#), [tarsnap](#), [timestamp](#), [tips](#), [two-legged](#), [web 2.0](#), [web application](#)

← Patrick's 10 Most Anticipated 2011 Summer Movies

[Understanding the Unix Epoch \(in Java\), Time Zones and UTC →](#)



About Riyad Kalla

Software development, video games, writing, reading and anything shiny. I ultimately just want to provide a resource that helps people and if I can't do that, then at least make them laugh.

[View all posts by Riyad Kalla →](#)

Tags: [Amazon Web Services API](#), [API](#), [authentication](#), [credentials](#), [guide](#), [HMAC](#), [HMAC-SHA1](#), [HTTPS](#), [nonce](#), [OAuth](#), [Public / Private Key](#), [security](#), [session sidejacking](#), [SSL](#), [tarsnap](#), [timestamp](#), [tips](#), [two-legged](#), [web 2.0](#), [web application](#)

← Patrick's 10 Most Anticipated 2011 Summer Movies

[Understanding the Unix Epoch \(in Java\), Time Zones and UTC →](#)

192 Responses to “Designing a Secure REST (Web) API without OAuth”



Jim April 27, 2011 at 12:40 pm #

As for your timestamp issue, if you use the # of seconds since the Unix Epoch (commonly referred to as unix-time) your timezone concern is completely gone, as the Unix Epoch is completely agnostic to timezones and daylight savings.

The concern then becomes whether or not your clients' times are set correctly. There's not much I can think of that would fix that, so you just have to say “set your time right!”

Also, your 10-15 minute time-window allows for replay attacks within that time period. Essentially, if a user could make a legitimate request once, they could repeat it N number of times for the next 10 minutes (or whatever your acceptable time-window is). I've seen methods around this, one of which was to include the 'microtime' for each request (basically the current number of milliseconds since the Unix Epoch) and any single micro-timed request could never be run again for any single API client.

Good read, though, it was light and explanatory, and definitely gives a high level overview of how private keys (and "secret api keys") work.

[REPLY](#)



Riyad Kalla April 27, 2011 at 3:55 pm #

Jim,

I had actually been wondering about how timezone was defined for epoch time, I didn't realize that per the spec it is UTC; that makes my life infinitely easier not only for this article, but for a data model I was specifying out.

Thanks for the heads up!

As for the replay attack, very true, the window does give an opportunity to slip an attack in, I was just following Amazon's window which slides "up to 15mins" in some services cases.

I don't quite follow the microtime correction; can you clarify that? (sounds like you are including the epoch time twice; once in seconds and once in milliseconds... not sure on how that helps or doesn't suffer the same issue of the existing timestamp check?)

[REPLY](#)



Jim Rubenstein April 28, 2011 at 8:24 am #

In reference to microtime, I was simply saying you could pass THAT instead of /just/ the unix time (in seconds). This would give you your time signature, time window limit, and could guard against a replay attack.

Basically my thought against the replay protection was that you'd log each request from the client with the microtime timestamp. When a new request came in and you determined that it was within the time-window constraint and passed hash checks (and whatever other security layers you have) you'd check to see if that microtime stamp had been used for another request. If it had been used in the past, you assume it's a repeated request and kick it back to the client with an error.

However, I really like Raphael's idea of an idempotent. That seems like a more significant undertaking though, you'd have to be more meticulous about what's happening so you never do the same exact thing twice. I'd imagine some services just can't work as an idempotent service. As an example, if you ran a web service that simply incremented a counter. Repeating the request would increment the counter each time.

Without any research (so this might be completely off-base, and is pretty much talking out of my butt, so take it with a grain of salt) I'd say the "simplest way" to make an idempotent web service requires a 2-request system. The client would initially make a request to the API to get a "request key." Once the client has received the request key, it uses that key to make *the* request to the API, with the request key it received from the first request. Each request key is single-time use, and you can put other restrictions on how long it's good for (infinite if you want?). That seems like a lot of overhead (twice as much, really!) but I'm not sure how else you'd create an idempotent feature into an API which is similar to "incrementing a counter."

I hope all that made sense, I kind of started to ramble, and I said the word "idempotent" a friggin lot! Sorry for that 😊

[REPLY](#)**Jim Rubenstein** April 28, 2011 at 8:32 am #

Dear Riyad, I need to be able to edit my comments because I suck at proof-reading before pressing post!

"I really like the Raphael's idea of an idempotent." – stupid. What I meant to say was, I really like Raphael's idea of an idempotent API.

Also, to clarify my first sentence even more:

In reference to microtime, I was simply saying you could pass THAT instead of /just/ the unix time (in seconds). This would give you your time signature, time window limit, and could guard against a replay attack.

Instead:

In reference to microtime, I was simply saying that you could pass it instead of the unix time in seconds. This would give you your time signature, time window limit, and could guard against a replay attack, all in one piece of data passed to the API.

Sorry for the post spam.

[REPLY](#)**Riyad Kalla** April 28, 2011 at 8:41 am #

Jim,

Thanks for the followup, I totally agree with you about Raphael's suggestion with regards to the nonce tokens.

You nailed my concern with that approach (doubling API traffic, and increasing data-layer access) BUT, just like you mentioned, I really really like how cleanly and exactly it solves the replay-attack issue AND something you pointed out that I didn't realize, is it can be used as part of a mechanism to implement idempotency on the server side... "Has this token been used? No, ok then re-try the command".

There is still the pain point of doing your own "transactions" with a data store that doesn't natively support them, but it is at least a step forward towards that goal.

Great feedback so far!

[REPLY](#)**Jim Rubenstein** April 28, 2011 at 8:57 am #

I don't know that transactions would really be a problem here. The response to the client doesn't necessarily have to be a failed one on an already used token. The transaction issue might come into play when you try to do something on the API over several API calls (in which case, ouch), but other than that – I think executing the API request and marking the token as used after the execution is successful is plenty.

If you're worried about part of the API execution failing, that should be handled on the application layer, provide an exception response to the client, and not mark the token as used. Let the client fix their request, and re-submit the request with the same token. If someone replays that same request, then they'll get the failed message all day long. If the client re-submits their request after they fix it, the token will be marked as used and no one will be able to use that

token in the future.

I might be missing the problem you're trying to solve with the transactional issue, but I just can't think of a use-case where I'd be worried about it.

[REPLY](#)



Riyad Kalla April 28, 2011 at 9:02 am #

You aren't missing anything, you are spot on with a tightly focused API (where each method represents a discrete action).

In my previous reply I was waving my hands around and inventing situations that don't exist trying to think of the downsides.

In my *specific* use-case, my API is extremely discrete and what you outlined would be a perfect fit for my needs.

[REPLY](#)



Ben Werdmuller January 18, 2012 at 7:24 pm #

This whole mechanism is pretty much exactly how the latako.com API works, and I can tell you from experience that there are a bunch of Windows users out there with their clock set incorrectly. The issue is usually that people set their base time to their timezone-adjusted time. It's annoying, and weirdly has mostly up in enterprise environments where users are unable to reset their clocks themselves (i.e., where IT has screwed up). Nonetheless, it's secure and the timestamp issue is usually surmountable.

[REPLY](#)



akhil handoo May 13, 2013 at 11:56 am #

The attacker will have to know the secret key to create new Hash including the new timestamp, which will be hypothetically impossible for it, since timestamp is a part of the checksum.

[REPLY](#)



Preston Lee April 27, 2011 at 2:06 pm #

@Riyad,

Great writeup! Silly question... if both the server and client know the private key, why does the client need to submit it? (Step 3b.)

[REPLY](#)



Riyad Kalla April 27, 2011 at 3:47 pm #

Preston, thanks for the heads up. That was an unfortunately "poorly formed sentence" bug. I removed the confusing info. Your "WTF alarm" was absolutely correct, the PK should never transfer over the wire.

That sentence should have only said "Send the HMAC (the one you made by combining the private key and your data blob)", but that parens section was redundant and confusing, so I nixed it.

Thanks!

[REPLY](#)



Raphael April 27, 2011 at 5:46 pm #

Great article!

Besides standardizing on UTC (which can still fail if the user's workstation has the time-zone setup incorrectly because they switch DST manually), there's another way to handle replay-attacks: If your API endpoints are idempotent, "replays" are turned into a feature that clients can take advantage of in case of timeouts or network disruptions. It's also considered a best practice in API design. I'm not sure if Amazon implemented it or not (I've always worked with higher-level libraries that shielded me from the actual low-level implementation details of their protocol).

Speaking of Amazon: They have an API for generating private API keys with reduced privileges, in case you need to share your private key with someone like an EC2 monitoring service (I forgot the exact name of the API) or an external DNS management service. Such keys can be easily revoked on demand (the tool to create these keys is a bit clunky). And finally, Google implemented a similar "revokable" feature as part of their two-factor authentication scheme for Google Accounts.

[REPLY](#)



Riyad Kalla April 28, 2011 at 7:46 am #

Raphael,

Good point about the UTC/DST issue... I haven't found a perfect way around the time-mismatch issue... I think besides crossing my fingers *really hard*, that's the best I can do to keep failing clients from connecting.

The idempotent behavior, as you correctly pointed out, is important and from what I've read/seen, a property of really well designed APIs for exactly the reason you gave (easy recovery for clients by simply retrying).

How would you go about implementing idempotent behavior in an API for a call that... let's say inserts a record with a NoSQL data store? With a SQL datastore I guess it's as easy as rolling back the transaction and either kicking a message back to the client or just letting the client time out and try again.

But with a NoSQL store like Mongo, Redis or SimpleDB, I'm not totally clear on a great way to do this... any ideas?

Ahhh! The IAM stuff over at Amazon, you are exactly right about the revokable keys; that is another great approach for massive APIs that will be adopted by lots of folks.

Didn't know about Google doing the same, thanks for the heads up!

[REPLY](#)



Raphael April 28, 2011 at 3:13 pm #

There's quite a bit of literature on how to implement idempotent receivers as this is widely used in the enterprise space. A simple solution would be for the client to attach a locally unique ID to each request (or a nonce), and send it to the server together with the request (like many suggested here to prevent replay attacks). The REST server would keep a key-value store of the hash of the client ID with the request ID (the request ID is only guaranteed to be unique per client, not globally) as the key in the store, and the request status/result as the value. After successfully authenticating a request, the REST server looks up the value from the store. An empty value means it's a new transaction that needs to be processed. A non-empty value represents the result that needs to be sent back to the client, without any further processing.

You might need to include a time-stamp within each value so that you could expunge stale items from the store over time, or use some other means to achieve the same goal. Redis is an ideal storage solution for this (it also solves the synchronization problem if you have more than a single REST server).

Transactions are not really a part of this protocol; this is just a fancy transport. It's left to the client and backend server to implement such things as the request parameters and results. If the request fails, the result object simply represents that.

Idempotent APIs lend themselves beautifully to situations where you have long requests which you process asynchronously. I just implemented something very similar to what I describe above in Node.js. I used a constant in the result value that signifies "in-progress", to represent that a task has been committed to the server but no results have yet been received for the client. This yields a very simple API, minimal server overhead and complexity (idempotency related code was less than fifty lines), and most importantly, most of the robustness of a middleware server without any of the bloat.

On a slightly different subject: I remember reading some treatment in a Schneier/Ferguson book about the issue of whether to sign a request with a nonce or leave the nonce out. Their recommendation was to sign (HMAC in our case) everything together, nonce included, and they had some sound security related arguments for doing that.

[REPLY](#)



Stevan Little April 27, 2011 at 6:33 pm <#>

First of all, nice writeup, I just recently had to design something like this for a whole suite of services we are developing for a client (<https://github.com/stevan/SAuth> if you are interested). It is always reassuring when you read that someone else has come to a similar conclusion as you have.

Second, you could also use a nonce instead of (or along with) a timestamp. This should still protect against man-in-the-middle attacks since a nonce can only be used once. It works well with REST as well since in HTTP digest authentication there is already a "nonce" field in the 401 response, and you can use the Authentication-Info header to include a "nextnonce". For me this was nicest since it left the payload free and clear of auth details.

[REPLY](#)



Riyad Kalla April 28, 2011 at 7:47 am <#>

Stevan, never heard of the nonce approach, thanks for the pointer!

Also I absolutely am interested in the example code, appreciate you sending that along as well so I can dig through and see what folks rolling to production are doing.

Do you have any pointers in hindsight after rolling out the API?

[REPLY](#)



Stevan Little April 28, 2011 at 7:59 am <#>

Riyad, no real pointers at this time, but we have only rolled it out for one application so far, I am sure as we add it to more apps things will come up.

Our system is very specifically tailored for our needs, which are a set of web-services that need to securely interact with any number of web-enabled applications and many different kinds of devices. It is very heavily based on OAuth (1 and 2), but specifically leaves out anything that has to do with human interaction/intervention, because we want this all to be transparent to the actual users. We can get away with this mostly because this will all be used

within a single company and is not for the general public.

[REPLY](#)



Riyad Kalla April 28, 2011 at 8:30 am #

Stevan,

I looked into [nonce](#) use – technically I really like the approach, have some token server laying around doling out 1-time use tokens (similar to how a lot of web frameworks protect forms against re-submits or invalid submits (POSTs) from external sources).

The only downside i see is literally doubling the API traffic (1 nonce for every API call) and increasing (doubling in the worst case) hits to your datastore layer for 1-time-use data that aren't that cacheable.

1 insert when it is generated and returned

1 query by the executing function to check

1 more insert to mark the nonce as "used"

BUT, that being said, unless you are trying to write the next twitter I really like this approach from a technical/security standpoint and appreciate the heads up on that. I may go with this myself because I am not writing the next twitter and can spare the overhead 😊

[REPLY](#)



Stevan Little April 28, 2011 at 10:10 am #

Actually there are no extra calls to be made. The initial nonce is sent with the first 401 response (or can be asked for explicitly though an API call if you want). And then after that nonce is simply passed to the server-side via the clients Authorization header and the next-nonce is passed to the client-side via the Authentication-Info header. The whole thing is pretty standard HTTP interaction, and really only adds one extra part to the already-being-sent Authorization header and one extra header on the way back.

As for the nonce storage, I am actually not storing them. We are using a cryptographic secure random number generator (strong if we can, but weak if we have too), which are pretty much guaranteed to not repeat (only under extreme circumstances will a weak generator repeat, which could likely be solved by just adding a high-res timestamp to it).

[REPLY](#)



Pradeep Sharma November 29, 2011 at 2:37 am #

It is not so necessary to do insert. The entire algorithm for generating nonce can use caching layer to cache the key for 5mins-1hour duration.

[REPLY](#)



Morten Fangel April 28, 2011 at 12:41 am #

What you just described is 95% of what two-legged OAuth (1.0) is. OAuth really isn't that complicated, so I personally wouldn't try to reinvent the wheel just to avoid figuring out how OAuth works.

However, use of two-legged OAuth isn't that well known so perhaps you just missed it. But try looking at it, and see if you don't come to conclusion that it is infact terribly similar to what you have outlined here.

[REPLY](#)



Riyad Kalla April 28, 2011 at 7:50 am #

Morten,

I have been pushing back on making heads or tails of OAuth since it was announced and the initial overview I read through made my eyes roll back in their head... that was years ago though when I expected all web auth to ever be done with a username and password; I think you are right I should take another look.

Thanks for the pointer to the "two-legged" OAuth; not heard that term before, but it gives me a focused starting point.

Any idea what changes are coming in OAuth 2 that make this stuff better? I read (from one of the linked OAuth articles above) that 2.0 is suppose to be easier to integrate for API devs and other type of service interfaces, but knowing nothing about OAuth, I didn't know in what ways that was the case.

[REPLY](#)



Stephen Sugden April 28, 2011 at 10:13 pm #

I have to 100% agree with Morten, and your response.

I was asked this last november to implement "OAuth using this client-provided private key". It took me an embarrassingly long time to figure out that three-legged (the one that 95% of my google results seemed to talk about) was not at all required, and in fact all I wanted was two-legged OAuth or signature-verification (those were the magic words google needed). After I sorted that out, the actual implementation was dead simple and sounds remarkably similar to what you describe in the article.

[REPLY](#)



Riyad Kalla May 2, 2011 at 8:46 am #

Stephen,

Thanks for adding a bit about your own experience; my limited googling around of OAuth a few months ago brought me many of the same types of results you are talking about (I guess that is what is called "3-legged"). I just scratched my head and moved on.

But now I have a more focused OAuth scenario to search for and appreciate the help from you guys digging through this approach, providing improvements (e.g. nonce) and even getting a good starting point with OAuth (2-legged).

[REPLY](#)



Amir April 28, 2011 at 8:05 am #

Why didn't the author just end up using SSL? Don't need to sign the request anymore. Essentially what OAuth 2.0 is doing.

[REPLY](#)

**Riyad Kalla** April 28, 2011 at 8:26 am #

Amir, SSL is certainly an option, but has downsides like more processor time on the server if your equipment budget isn't as high as you would like.

This was just looking at one of many ways to secure an online API; I think right now with most popular web-2 protocols, non-SSL APIs are probably a lot more common than SSL-protected ones (e.g. twitter, facebook, google, aws)

But if you have the option of using SSL, I agree, it is definitely a good choice.

REPLY**Pelle** February 16, 2012 at 6:45 am #

The SSL being slow and processor intensive was maybe true 10 years ago. Now not so much.

All serious web services now a days use SSL anyway and for good reasons.

OAuth 2 is essentially ready to be used, but unless you need to deal with delegation you don't even need that.

The Bearer Token spec is what most people call OAuth 2 and is just a single token in a http header or query string.

<http://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-16>

If you need to share url's such as Amazons signed urls where you need to give some access to a resource such as an image or download use the Mac token, which is receiving serious security analysis now:

<http://tools.ietf.org/html/draft-hammer-oauth-v2-mac-token-05>

Both of them are still officially drafts, but are mainly receiving wording changes now.

OAuth 1.0 should not be used for any new applications and was only complicated for Library authors. It was always recommended that you never attempt to implement OAuth 1.0 without a library.

REPLY**Riyad Kalla** February 16, 2012 at 10:22 am #

Pelle, while the overhead for encoding the communication does seem to be an issue for servers of yester-year, the initial negotiation with the API does still seem to take more time when done over HTTPS.

That being said, I'd agree with you that the trend absolutely looks to be moving towards "everything over HTTPS" though.

Also appreciate the link on the specs, didn't know about bearer tokens!

REPLY**Jim Rubenstein** April 28, 2011 at 8:27 am #

There is always that...haha.

REPLY**Riyad Kalla** April 28, 2011 at 8:37 am #

Amir,

I thought you brought up a fair point, so I did some more searching around HTTP/HTTPS and it looks like the [initial handshake is fairly expensive](#) with an HTTPS connection, so I imagine this would be a killer for a high-throughput API.

[REPLY](#)



Amir April 28, 2011 at 8:53 am #

I was looking at that link recently. I haven't found the initial handshake to actually degrade performance. I would be curious to see some performance comparisons between the two with recent technology. I am also curious to know how Android/iPhone handle the initial handshake and how much of a loss it really is.

All this aside, your solution is sound to me. I had a similar task a year ago and did something similar to yours but after much research, I found something called "WRAP Access token" now renamed as OAuth 2.0. So I thought to myself, if they are going that direction then I should do something similar. After benchmarking on iPhone and Android I concluded that there really isn't that much loss.

I guess the big question is: Would you rather keep your API simple so clients don't have to do such crazy "signing" tasks by using HTTPS or gain a little performance but keep the API harder to use. I leaned towards the first option because I felt keeping the API easy to use is a winner.

[REPLY](#)



Jim Rubenstein April 28, 2011 at 9:00 am #

Easy APIs for the win!

[REPLY](#)



Riyad Kalla April 28, 2011 at 9:05 am #

Amir,

I have nothing more than a few Google links and speculation to back up the non-HTTPS-approach and absolutely agree with your focus on ease-of-use priority.

It sounds like you have a lot more experience with this stuff than I do, especially in the mobile space which is what I would be targeting.

Out of curiosity, once you had the API running over HTTPS, do you just go back to basic authentication? Username/Password over-the-wire or did you use some other mechanism, like an API Key with no additional credentials?

[REPLY](#)



Amir April 28, 2011 at 9:17 am #

No basic authentication. The idea is simple and exactly as what you do above except no signing. This is how I implemented it

1. Client asks for a token by going to /REST/newToken?
user=Amir&pass=PASSWORD
- The above should return a token to the client and the application should save that token to associate with that user

2. To actually use the API, you do /REST/someservice?

token=TOKEN

Here the TOKEN is from step 1.

You don't need to sign anything because https is already taking care of that for you. All the app needs to do is look up the token to find the user.

Now you may ask, isn't sending username and password in step 1 still exposing the password. I thought that too and if you look at HTTPS closer, you will notice that even the URL is secured and any man in middle can not see the URL.

So as you can see, the client is simpler because it doesn't have to worry about signing anything. The server is simpler because it just relies on SSL doing the work for it. Everybody is happy. You don't even need to worry about replay attack because someone would need to be able to look inside the HTTP request to be able to resend everything.

The only hint I have is that Android is really fussy about SSL. For example, last time I worked with it, it failed to validate the difference between <http://www.foo.com> and foo.com in the SSL certificate. But there are ways around that. 😊

[REPLY](#)



Riyad Kalla April 28, 2011 at 10:15 am #

Amir,

I appreciate the impl details. Are you timing out the tokens eventually or are they infinite? Like if you deploy an app to someone's phone, and they login for the first time, they get a token (with their phoneID associated or something) and for the life of that app being on that phone, that app will keep using the same token?

Seems like a good way of identifying devices.

[REPLY](#)



Amir April 28, 2011 at 10:19 am #

Yep. We set a limit to 10 tokens per user. So if you had 10 devices you can still use all of them. You can also send the device info when you are requesting for a token. As you guessed it, this is an excellent way to track devices and let the user detach from a device using the web.

We specifically chose not to time out. But this is up to the implementer. I am not sure which is better. I imagine if you have financial data then you would want to timeout, else make it infinite.

[REPLY](#)



Gregg July 5, 2011 at 10:15 am #

While all URL data (except the hostname) is encrypted over-the-wire when using HTTPS, you still have to worry about the URL data being stored in plaintext in the server logs and in the user's browser history. For this reason, I suggest not including sensitive information in the URL.

[REPLY](#)**Riyad Kalla** July 5, 2011 at 10:28 am #

Gregg, great point. In discussing the idea of using HTTPS further with other friends, there was this false sense of security that "nothing can be hacked!" that I think makes your point even more important, because the tendency might be to trust the URL a little too much with secure data (e.g. login credentials).

[REPLY](#)**Amir** July 5, 2011 at 11:34 am #

This is true that logs are sometimes stored in clear text. I believe that operations and writing web application are two different domains. For our clients, we had to secure all the hard drives. Encrypt all mysql data. Encrypt all keys and user cookies. The list can go on. 😊 But good point, I agree with that. Always use post if you can. (Even though that can be logged)

**Nathan** June 19, 2012 at 3:16 pm #

More expensive than having to hash your entire POST or UPDATE payload *twice*?

[REPLY](#)**Nathan** June 19, 2012 at 6:13 pm #

I mean, making the HMAC on the client and the server isn't "free."

[REPLY](#)**Vinnie** May 13, 2011 at 1:42 pm #

What I still don't get is that it makes no difference if you use SSL for mobile or desktop apps. I am far from an expert, but simply monitoring the traffic with a tool like Fiddler allows anyone to see what the request looks like before they get encrypted with SSL. Then they could easily just replay the request any time they wanted to download the data outside the app.

Am I missing something? Yes, I understand they can't replay 15 minutes later, but that doesn't solve the problem of the client app being unverified. Essentially, how would one guarantee that the request to the web service is being sent by the binary app that the users have installed on their device/computer? I can see how all these methods will work from server to server, but I still don't get the client app security here, not even mentioning decompiling your app at this point.

[REPLY](#)**Riyad Kalla** May 13, 2011 at 6:39 pm #



Vinnie,

If an SSL tunnel has been established between the client and server all traffic between the two points is encrypted. What Fiddler does to "Decrypt HTTPS traffic" (per this page: <http://www.fiddler2.com/Fiddler/help/httpsdecryption.asp>) is dynamically generate an SSL cert that you have to accept in order for it to eaves-drop on your traffic.

In the real-world deployment, you wouldn't accept a random cert like that, so the man-in-the-middle trying to do what Fiddler does wouldn't be able to decrypt the encrypted data.

As for decompiling the app, that would give you access to the private key used to sign the requests; so you are correct, that would be an attack vector.

Twitter and other services combat this by accepting that your key can very well get compromised, so keys can be disabled on the fly until new keys are generated and used to replace the old ones.

I'm not crazy about this as you can just get decompiled again and again, but I haven't seen a way around this yet.

REPLY**Vinnie** May 16, 2011 at 6:23 am #

So forgive my ignorance here, but can't they view the request data using Fiddler and just replay that?

For instance, let's say I have a header named "Secret-api-key:" with a value of "123456". The evil genius will use Fiddler to see the request being sent through HTTPS, and just replay the request with the secret key from some self made client. Basically, making their own tunnel.

Am I missing something again?

REPLY**Amir** May 16, 2011 at 7:16 am #

With SSL you cannot look at any of the headers or request parameters. So how would they replay the request? With SSL everything is secured including the URL it self.

REPLY**Vinnie** May 16, 2011 at 10:14 am #

I guess I'm not understanding. So can Fiddler decrypt the request or not?

REPLY**Amir** May 16, 2011 at 10:36 am #

This is how fiddler does it
<http://www.fiddlertool.com/fiddler/help/httpsdecryption.asp>

In abstract, it mimics a server and forwards everything to the actual server. It does this by displaying many warnings to browsers such as IE and FF that state you cannot trust this certificate. So really it just used for testing purpose. In the real world, one would have to step between foo.com and the client by changing foo.com dns to actually point to fiddler. This is

pretty hard. After that, the attacker creates fake certificates which will actually be flagged by all browsers and probably rejected by all mobile devices.

[REPLY](#)



Vinnie May 16, 2011 at 11:14 am <#>

I think we're talking about two different things. Here is my concern. Suppose I have an API with private resources. For instance, I have picture that I want to serve to a mobile app. This picture can't be viewed in the app until the user passes some sort of milestone... let's say getting to level 8 of a game. In one sense the resource is public because it doesn't care "who" is accessing it, but in another sense it's private because it only wants my client app to access it. This is totally not a real example so picking apart my scenario and suggesting a different way is not what I'm looking for. I'm more interested in the security and keeping my API strictly for my mobile app.

So the app sends out a request with the secret API key encrypted using HTTPS. Using Fiddler, evil genius captures the request in raw form. So, evil genius goes and creates a .NET form app, sends the exact same request using HTTPS that he just saw in Fiddler. Then my server happily serves the image to his .NET form because after all, it's got the secret key in the request.

Go easy on me if there's just something not penetrating my skull. I understand your "in the real world" comments I think. But I'm still not getting how a REST web resource can be secured without a personal username and password for each person using a client app.

[REPLY](#)



Amir May 16, 2011 at 11:30 am <#>

I am still confused on this line –

"Using Fiddler, evil genius captures the request in raw form."

How would evil genius do that? he/she would need to reroute the server to the fiddler server. And even if they could access dns settings on some gateway, all the certs would fail. which means all mobile apps would reject it and the app would break.

Assuming he could take over the app by doing all those things above then yes what you say is true. But as you can see HTTPS was designed to not be able to have any man in the middle attacks. Fiddler simply just reroutes requests for dev and testing.



Vinnie May 16, 2011 at 1:12 pm <#>

I guess I am still confused too. I was under the impression that Fiddler could decrypt the request being sent from the client... headers and all. If you're saying nothing can do that, then I'm

good to go.

I'm not saying evil genius would do this for every single client, but just do it on his own iPhone using Fiddler as a proxy. Does that make sense?



Riyad Kalla May 16, 2011 at 8:15 pm #

Vinnie – right, Fiddler can't just randomly decrypt SSL traffic; only traffic where you have agreed to its cert, and your app will never do that, so no dice for the evil genius.



Eduardo Cereto February 16, 2012 at 2:19 am

#

Note that each fiddler installation has its own cert. So even if you also use Fiddler on your machine and accepted the fake cert another person on another machine routing the traffic through his fiddler won't be able to decrypt since he has a different cert on that machine. Fiddler certs are random and generated at install time.



Vinnie May 17, 2011 at 4:03 am #

Ahhh... Yes. That makes sense. Thanks for taking the time to get it through my skull. My app won't have the little popup that says "do you accept this suspicious Fiddler cert?" and therefore won't handshake. Got it.



Dennis December 27, 2012 at 12:41 pm #

I'm with you Vinnie. I've seen instructions on how to get a Windows Phone to install the Fiddler certificate and assume similar things exist on Android and iPhone. If the device has accepted the Fiddler cert, then how would I guarantee that my app won't ever accept a different certificate?

Wouldn't my app be relying on the device to handle the SSL handshake? I don't think I'd be doing that myself.

I'm not concerned with man-in-the-middle attacks so much as someone trying to replay their own request that came from the app on their own device. Let's say my API accepts a request with the user's new high score or best time. If someone ran the app and captured all the API traffic via fiddler, they could adjust the score and re-submit the request.

HMAC hash and nonce seem like a good way to guard against that, but simply using SSL doesn't sound to me like it would be enough protection.



Napo June 15, 2011 at 6:24 pm #

Regarding AWS's solution, private key is used to hash the content, that means Amazon has to store my private key without hashing. For a secure REST API(Browser/Server), I do not know where I can save the "private key" at client

side(browser). In case client uses customer's password as a hash key, does it mean I have to save his password in plain text at server?

[REPLY](#)



Riyad Kalla June 16, 2011 at 7:15 am #

Napo,

You don't need to store the key in plain-text necessarily, you could encrypt it, but SOMEWHERE in your app you will have a decryption key for that encrypted key, so that is just security-by-obscurity and if someone is digging that hard to pull the key out of your app then yes, they will probably have access to your private key whether you want them to or not.

I've not found a great solution to this on the client side, but on the server side I've decided to follow a model similar to Twitter's — each account can register multiple private API Keys and expire them when they become compromised then generate/embed new ones in an updated release of the app.

Not the best, but at least it is designed to cope with the eventual need to replace the key up-front and the software doesn't have to struggle with that portion down the road. It still feels insecure to me, but I haven't come across a great technique for adding that last little bit of security on the client side to help make it harder to compromise the keys.

If you find a method, please come back and share.

[REPLY](#)



Anonymous October 21, 2011 at 4:29 pm #

Why not have treat the private key as a hash of the password and some salt (perhaps a hash of the username). Then have the same algorithm on the client side of things. Nothing to be transferred as the user would just enter their new password on the client side and everything is updated and nothing stored in plaintext that the user might have not wanted to expose.

[REPLY](#)



Ramesh July 15, 2011 at 5:00 am #

Great post !!!

Regarding the timestamp, I think the timestamp has to be sent as a separate parameter (apart from it is being included in the signature)..since the server cannot fetch the time from the signature.correct me if I'm wrong.

Also, it would be great if you can point me to sample code where the rest API implementation with this authorization header. Just want to know how to read authorization info from the header..might need to user filter/handler to intercept and read the header I assume...

[REPLY](#)



Amir July 15, 2011 at 7:13 am #

Ramesh,

I think you are right. The client does have to send the timestamp with one of the parameters. The idea is also to reject the request if the timestamp is after some time period.

[REPLY](#)

**Riyad Kalla** July 16, 2011 at 7:56 am #

Ramesh,

Absolutely correct – everything included in the signature (except the private key) has to be sent to the server because it has to re-construct the same signature server side and compare the two hashes to see if they match.

So technically every parameter is optional, just make sure whatever you use, you send to the server (except private key) so it can re-produce. The signature generation the server and client agree on just has to be the same.

FWIW, I saw an API the other day that used this HMAC method, but all it had you do is use your private key to sign the api key and send that across as a signature.

The problem with this is that NONE of the params were included with the signature calculation, so technically a malicious 3rd party could intercept the call, change any and all params, and re-submit with the same signature hash and the server would still accept it.

This *technically* becomes the same problem of side-jacking/session-jacking in that a nefarious man-in-the-middle doesn't need to get your credentials, just use your authorized signature to be malicious.

That is why specs like OAuth define *Such* pedantic rules for how to combine all the parameters together in a very specific way, so the client and server can agree on all of it.

**Ramesh** July 20, 2011 at 1:24 am #

Riyad, totally agree with your points...also can you point me to actual web service implementation which uses this kind of authorization? I would like to know how to read the authorization header from the request. I'm using Spring Restful web service.

**Riyad Kalla** July 22, 2011 at 7:12 am #

Ramesh, lots of web services use OAuth.

You'll want to find the OAuth support in the Spring REST WS libraries so it can just do this stuff for you automatically, you wouldn't want to write this stuff by hand.

Jersey, for example, provides its own support for OAuth.

I just did a quick Google for "spring restful web service oauth" and got some results that might give you a starting place.

**Sid** July 19, 2011 at 7:35 am #

Excellent post. Describes exactly what I had been doing all last week (besides the peyote)

**Fered** July 20, 2011 at 4:46 am #



Thanks Riyad that's a good overview. Can you confirm if my understanding of these points is correct?:

- There is no need for a Login method or SessionId (since every call uses the hmac instead)
- SSL improves security, but hmac is still required when using SSL
- The hmac must be sent as a http header, not a parameter to the method being called
- Nonce (if used) must be sent as a http header
- Does it matter what the name of that header is?

Thanks 😊



Riyad Kalla July 22, 2011 at 7:21 am #

Thanks Fered, I'm glad it helped. To your questions:

- Yes
- No. HMAC is used to protect the integrity of the params; help stop man in the middle attacks and the like. Over SSL, there is no way for the man in the middle to read the params in the first place, so you don't need HMAC. When you use SSL, you can use "old fashion" or easier methods of API authentication by having a user do something like /login?username=bob&password=123abc and then give them a session ID they send with every subsequent request to identify themselves.
- No. It doesn't matter how you send the HMAC, all that matters is that the client and server agree on (a) where to find it and (b) how to calculate it. You could, for example, write the API such that the HMAC is always sent as a parameter named "hamSandwiches" and that you calculated it by counting the characters in ALL your parameters, adding them all together into 1 big number, like 1137, and then signing that number with your secret key and submitting it. (that's not a great idea, it's just an example). The point being that you can literally do whatever you want, but every client has to know how to do it the way the server expects it, because the server is re-creating the SAME hash (HMAC) after it gets the request to verify nothing changed.
- No. Same as question above, if you use a nonce, there is no magic here. It just has to be a unique string that the server can identify as being used before or not. So if I send you a request with "nonce=HelloSuzie", the server will compute the HMAC, store in the DB that "HelloSuzie" has been used and then process the request. Now if ANY other request ever comes in again with a nonce of "HelloSuzie", the server can assume it is a man-in-the-middle or some nefarious client trying to re-submit past requests again or trying to mess with the system some how and will reject it. nonce's are an absolute way to defend against "replay attacks" as opposed to timestamps which help, but still leave wiggle room.
- It doesn't matter what you name anything, as long as the server and client agree.

ok, now ALL that being said, if you were asking *specifically* about OAuth, then "yes" to all your questions about specific header names.

OAuth does define very specific names and locations for things to be (it has to, because it is a spec), you can see the names of those headers in the [OAuth spec](#).



Colin Wiseman August 18, 2011 at 11:14 pm #

So using SSL, and just like Facebook these days, you could just give the user the access token in the app they are trying to access (e.g. In their profile), to save a lot of fuss. And this access token could actually just be a Guid, a random Guid or even the Guid that is their profile Id? Cause if a developer of the other side isn't secure with their access token then really god help rhe for what ever happens 😊

But utterly awesome article, you wrote down everything that was in my head, but gave the

answer at the end! Nice one!

[REPLY](#)



Riyad Kalla August 19, 2011 at 2:31 pm #

Colin, it is true that a lot of this just melts away and gets simpler when you move to a SSL-secured stream, and thank you for the kind words!

[REPLY](#)



Tal Maizels August 22, 2011 at 10:48 pm #

Hi Riyad,

A kick-ass post! Found it very helpful and greatly written.

I'm writing a web app with RESTful API, which will serve all the data for the JavaScript & mobile client. Parts of the site can be browsed by everyone and parts are for authorized users (authenticated & Role checked).

At first i was troubled by the possibility that everyone can get to my REST API and read results in the parts that don't need special authorization, but then i realized that it's exactly the same as scanning the final html in the site.

The important thing is to verify that you don't serve private data which you don't show on the page, through the RESTful API. Specifically this privacy breach can occur when marshaling an entire object, which contains more data than needed in the page.

Be sure to do the marshaling wisely and avoid the fields you don't need.

Thanks again and Good day.

[REPLY](#)



Colin Wiseman August 23, 2011 at 12:32 am #

HAHAH! I have done that. Built a website, downloaded every via a generic handler and JSON – it had all the user's passwords in it. Thankfully it didn't go live straight away! DOH!

[REPLY](#)



Alex Couper August 25, 2011 at 2:33 am #

Fantastic article.

[REPLY](#)



Riyad Kalla September 3, 2011 at 10:07 am #

Thanks Alex.

[REPLY](#)



Chris October 12, 2011 at 11:08 am #

In your outline of the algorithm, you say "combine a bunch of unique data together (this is typically all the parameters and values you intend on sending, it is the "data" argument in the code snippets on AWS's site)" then HMAC that.

Does this approach open the door to a different kind of man-in-the-middle attack? If the hashed content is just the arguments, not the full URI of the request — wouldn't an

attacker be able to intercept your API call, and change the URI and/or REST verb while using the same valid arguments? In other words, change a CREATE CAT to a CREATE DOG, or change an UPDATE DOG to a DELETE DOG, assuming the parameters are the same?

Seems like the hash needs to be based on the entire request, not just params.

[REPLY](#)



Riyad Kalla October 13, 2011 at 11:31 am <#>

Chris, *excellent* catch, you are exactly right. I've updated the post to add a note about that.

[REPLY](#)



Venki October 18, 2011 at 7:57 pm <#>

Great article.. Well explained the intricate details in layman terms !! Looking forward for more such elegant articles !!

[REPLY](#)



Kele October 20, 2011 at 12:34 am <#>

Hi Riyad,

Great article and lots of great comments.

So in a pure javascript-based web app (the client) are there any best practices for storing the private key? Or do you just try to bury it somewhere in the source code and hope someone doesn't want to spend the time looking for it.

Also if you also need to authenticate user accounts not just authorize a client app it still does seem like SSL is the safest way to go. Or use OAuth and have the user authenticate directly on the server first and grant access to the client app to make calls on their behalf.

[REPLY](#)



Kevin October 26, 2011 at 9:58 pm <#>

Riyad,

Excellent article! Very informative for someone like myself trying to get a foundation on which to build my first API.

I have one question, however. Are there glaring problems with simply issuing a client a single "secret key" and then having each request (always sent over HTTPS/SSL) look something like api.com/service/?param1=1?param2=2?key=abcd1234 and then server-side simply compare that "key" value to something stored in the database? If the key matches, good to go! If not, 401.

Is the username/password -> issue a token necessary?

Thanks!

[REPLY](#)



Riyad Kalla October 29, 2011 at 9:11 am <#>

Kevin,

Your thinking is spot on, if you are doing everything over HTTPS the interaction with

the API becomes infinitely easier. If your customers are good about not sharing keys, then sure, just having them send along a key with each request (with no “login” action) is a perfectly viable usage model.

Adding the user/password “login” process to the API is just another level of security if you need/want it.

If you don't need it, the API key only is fine, you just want to make sure to educate users on how important it is that they don't share the key or inline URLs in their website with references to your API with their key sitting there in plain-text.

If you aren't ever going to have use-cases like these (e.g. an API for mobile app devs) then that should be fine.

But if it is possible you COULD end up with situations like this with users that aren't so security-minded, then forcing the “login” sequence helps keep them safer from themselves 😊

[REPLY](#)



Kevin November 2, 2011 at 9:33 pm #

Riyad,

Thanks for the reply! It seems like my original thinking is OK; however, I've realized another problem: we'd have to put that secret key in an AJAX call, which would be exposed to the browser in plain-text.

I imagine a solution wherein the AJAX passes the data instead to an intermediate page “callAPI.php” which then appends the secret key and in turn calls the actual API with all necessary information, without any of it being exposed client-side.

Is having this intermediate page too clunky? Is there a more obvious solution I've missed?

Thanks again for the excellent article and help!

[REPLY](#)



Riyad Kalla November 3, 2011 at 10:37 am #

Ahhh yea with AJAX this gets harrier. The problem with that intermediary page is that your AJAX code calls it with no way to verify that the call is from your web page — meaning I can watch your calls in my AJAX console and call “callAPI.php” directly myself and ruin your day, for example: callAPI.php?action=eraseUser&id=1,2,3,4,5....

You get the idea.

So the question is how do you know you can “trust” the JavaScript making the call because it resides, in plain text, on a machine outside your circle of trust and honestly, I don't know the right way to do this.

There was another developer working on a script-generation design framework that was doing all the generation client-side in JS, then in an AJAX call, uploading all the PHP to the server where it would run... I pointed out that *I* can call that same AJAX call and upload my own PHP, which executes “rm -rf /” and he wouldn't be able to tell a difference between my call and his call to protect from that sort of thing.

I don't think he ever ended up addressing the security hole so I am not sure what the “right way” to do this is.

I'll poke around and ask some smart folks and see what I can come up with. Let me know what you find as well!

[REPLY](#)

Amir November 3, 2011 at 11:20 am #



Riyad,

There is at least one way you can stop ajax request being made from an intruder. The idea is to do the same thing one would do to stop cross-site request forgery (CSRF). Some common approaches are to check for the referrer header or using some kind of token. Using any headers obviously won't work because they can't be modified even by curl!

Assuming the token is the right way to go, you could create a token just for that user's session and then put a hidden field with that token on the form. The ajax request will also need to post the token for it to work correctly. Another intruder cannot intercept and repeat the same call because they don't have access to the same session.

Now the important part is if the intruder was sitting at the user's machine then all hell will break loose because at that point you can do anything. The idea is not to stop someone virus or persons on the host.

Here are some readings:

http://en.wikipedia.org/wiki/Cross-site_request_forgery
<http://haacked.com/archive/2011/10/10/preventing-csrf-with-ajax.aspx>
<http://blog.stevensanderson.com/2008/09/01/prevent-cross-site-request-forgery-csrf-using-aspnet-mvc3-antiforgerytoken-helper/>

Let me know if it makes sense.



Riyad Kalla November 4, 2011 at 10:47 am #

Amir, great point about the session ID — I see more and more frameworks (at least in Java land) auto-generating those with forms now to avoid re-submissions for exactly this purpose.

I am still trying to figure out how to protect against a nefarious client, let's say Bob Jones, that is a jerk and wants to cause you problems. So he connects to your app and starts sniffing/analyzing your traffic with the intent of breaking your app and ruining your day.

How would you protect against that? I can't think of a scenario involving private/public keys, because you don't necessarily trust the client — so you can't give them a key. I suppose the same is true for any Android or iOS app that has been decompiled and the private key pulled out of it, then the person would be free to make API calls using the stolen key from any location and your service API would be none the wiser.

There may not be ways to protect against this stuff, I just wanted to see what you guys thought.



Amir November 4, 2011 at 11:03 am #

I had thought of this too and came to the conclusion that SSL fixes this. I am not sure what you mean by "connect to your app". There are a few things going on:

- Man in the middle attack. (Protected by SSL will take care of this because the handshake for SSL is strictly between the client and the server)
 - Having a user sent to a specific page for example delete.php?id=xyz is protected because the user has a different session token
- Are there any use cases I am missing?





Kevin March 7, 2013 at 9:09 am #

Hi Riyad,

An excellent article which provides clarity on many aspects of HMAC and secure REST API development.

Wanted to know your thoughts on designing RESTful APIs for Mobile applications.

1) How do the User {of an Mobile app} get the Public & Private keys to consume the REST services. During registration or first login?

2) Also, should we have Public & Private keys bundled for the Mobile App which is specific to OS {Android, iOS, Windows Mobile}. Then keys are common for the OS across all the users for the REST API.

Appreciate any suggestion or thoughts.

Thanks,

Kevin

[REPLY](#)



Riyad Kalla March 23, 2013 at 4:03 pm #

Kevin,

Great questions.

1. Your only concern is the private key and transferring that securely to the client. One way to do that is to have the client authenticate with the host server (over HTTPS) and then transfer the secret key for that user and have it stored on the mobile device in secure storage and used for future communication.

2. Yep; this is a problem that exists today. Twitter just had all their secret app keys compromised and posted online:

<https://gist.github.com/re4k/3878505> — this means that ANYONE can make twitter API requests and identify themselves as “the official twitter app” (for example) — with the current OAuth impl, there is no great way to fix this. You have to trust something and in this case trusting the host platform to keep the keys stored in a secure location is the best we can do. This is considered one of the “biggest flaws” with OAuth, but I put quotes around it, because it is actually a flaw with public/private keys in general.

[REPLY](#)



orftz November 4, 2011 at 9:47 am #

Well-written, funny, informative. Thanks a lot.

[REPLY](#)



Riyad Kalla November 4, 2011 at 10:50 am #

orftz, most welcome. Thank you for the kind words.

[REPLY](#)



Luca Degasperi November 8, 2011 at 4:17 am #

Hi Riyad,

I've found this article really interesting and informing. Still I have some questions and my application will behave as follows:

Trusted Mobile App (with public and private key) sends username and password to the server -> if the request is valid (hmac checksum, nonce and stuff) the app gets a token for the user.

How can I send the password in a way that it's not decriptable? Can i hmac it with the private app key?



Riyad Kalla November 9, 2011 at 2:00 pm #

Luca,

If you are sending credentials over the wire, you'll *have* to use HTTPS for that — it is the only way to ensure that no man-in-the-middle sniffs your traffic and ends up with information that is important user data (either login name and/or passwords).

Once you have "logged in" the user and given back a session ID, the hard part is that you probably should still keep doing your requests over HTTPS, because again, a man-in-the-middle can simply intercept your SessionID's and do what is called sidejacking... basically mimick an authenticated user and make requests to the system as them.

If you want to avoid "login" and "session" keys, what you do is distribute your public/private keys inside the client app, and then there is no authentication step... it just makes its request along with the HMAC, the server gets the request, re-calculates the HMAC based on some unique ID identifying the user (typically an "API key" registered to the user) and if the HMAC checks out, runs the command the user requested.

You can run this over HTTP and there is no need for the "login" step. This is how OAuth 2 API authentication works, AWS API auth, etc.

Let me know if you have more questions!



Luca Degasperi November 9, 2011 at 2:21 pm #

I understood what you are talking about. Still I don't think is my problem. I'll try to explain myself better. This is the flow I want to create:

- The mobile app has a public and a private key to make it identifiable by the server
- the mobile app requires the user a username and a password (this credentials should be sent in some secure way).
- the url gets hmac-ed with the private key
- if everything is fine, the server sends the mobile app an authorization token, again hmac-ed with the private key of the mobile app.
- the mobile app makes the subsequent requests with the authorization token.

My question is: having a private key on the mobile app side, can i use it to "securely" send the user credentials to the server?



Riyad Kalla November 12, 2011 at 11:54 am #

There are a few different functional parts to this question (and

to my answer).

Credentials ==> To safely exchange a username and a password, you have to send this over HTTPS. Keep in mind that the 2 pieces of data (username and password) are no different than “public key” and “private key” — you are sending two pieces of data, 1 that can be publicly known (username OR public key) and one that CANNOT be publicly known (password or private key) — the only way to exchange this safely is to use HTTPS.

Authorizing API Calls ==> The authorization of API calls requires two things (1) you know WHO is making the call and (2) you TRUST who is making the call.

For part 1 (WHO is making the call) you can identify the user in two ways: using a public api_key (standard) OR making them login and then giving them a temporary “session_id” or “auth_token” that they can give back to you, identifying themselves with each subsequent request until the session or auth key expires (e.g. in 1hr). NOTE: In either of these cases a malicious man-in-the-middle can intercept your identifying value and pretend to be the user by making its own requests with their api_key or auth_token. It is because of this that you must go on to part 2...

For part 2 (TRUST who is making the call) you need a mechanism by which you know to trust the requests coming into your API. You *have* to assume that a malicious man-in-the-middle has recorded an api_key or auth_token and will try and use it, so how do you protect yourself against bad calls? Well, this is where the Private Key comes in.

Because you cannot send the private key back to the server in plain text, you instead generate a checksum *of* each request and send it *with* each request back to the server (HMAC). Now the server recalculates the checksum and if it matches, realizes that YOU are someone who knows the private key and trusts you.

=====> That is why your “login” step and “auth_token” approach aren’t necessary — because each request back to the server needs more than just a simple auth token, you need to re-verify with every request that YOU are who you say you are.

If you just use an auth token, as a malicious man-in-the-middle, all I have to do is wait until you authorize, steal your auth-token and then make all the requests I want on your behalf. BUT, if every request you send to the server includes an HMAC, signed with the private key, of all the arguments you are sending me — as the man in the middle, there is no way I can spoof your requests until I know the private key.

I hope that helped.

[REPLY](#)



Luca Degasperi November 12, 2011 at 12:29 pm #

Sure, this helped a lot. The part of the HMAC is crisp and clear, now also the part of user authentication. Thank you a lot!

[REPLY](#)



Luca Degasperi November 12, 2011 at 12:33 pm #

But what if i HMAC the authentication request with booth the user password and the private key of the app? the password will not be sended in this case.

So I will have the trust of the application making the call and the trust of the user.

Makes sense?

[REPLY](#)**Riyad Kalla** November 13, 2011 at 3:21 pm #

Luca, functionally there is nothing wrong with that, it is just adding an unnecessary step. If you are already authenticating *every* request by way of an HMAC, then having the user "login" in order to get an authentication token is unnecessary, and then hashing the auth request itself after you get it, is a 2nd level of unnecessary steps.

To best understand this, take a step ALL THE WAY back to the beginning, what is the *1* thing we are trying to accomplish? A method for the server to ensure that the person making requests can be trusted.

An HMAC with a public/private key does this — it lets the server know that the client making the request has access to the private key (Which is a secret only the server and trusted client(s) know) — so if the server gets a request from a trusted source, that's it, it can execute the request.

There is no need to have the user login or get a token.

NOTE: If you are going to do *all* your traffic over HTTPS, then having a user login to get a temporary session ID that is used in subsequent requests (only over HTTPS) can make sense, but going over HTTP — you just need the HMAC and a public/private key.

Hope that helps!

If this gets confusing, don't be afraid to step back from all of it, and just start from that one goal: how does the server know how to trust the client?

If you just walk from that perspective forward, step by step, I think this all makes more sense.

FWIW, it took me like 3 months to understand all this nonsense, I knew *nothing* about security let alone OAUTH or HMAC's, it wasn't until I sat down and thought through it myself on a piece of paper that I finally understand WHY these things are designed this way.

Take care.

[REPLY](#)**Luca Degasperri** November 14, 2011 at 1:59 am #

"To best understand this, take a step ALL THE WAY back to the beginning, what is the *1* thing we are trying to accomplish? A method for the server to ensure that the person making requests can be trusted."

Exactly. What I need is to ensure both the application and the user can be trusted. So two signatures for the access_token request are required. one for trusting the app and one for trusting the user. Also the access_token the server sends back, will need to be signed?

[REPLY](#)**Riyad Kalla** November 15, 2011 at 2:50 pm #

Ohhh, Luca my appologies, I was so focused on the client-trust issue that I was missing the fact that you are trying to auth a user as well.

Unfortunately I am not sure of a good way to secure the access_token once it is granted... at any point a man-in-the-middle can intercept it, BUT, I think that is OK because your API calls themselves will be protected via the private_key.

You can either do *all* API communication over HTTPS and get rid of the

idea of an HMAC and securing the API queries — or if you want to do the login over HTTPS and all other followup calls over HTTP, then we just assume the access_token (it's really more like a session_id used to identify the user as "logged in" for a period of time) can be compromised at any time and actually just acts as a temporary alias for the username/password without you needing to send back the username/password every time you make an API call.

This helps keep the username AND password out of the hands of a nefarious man-in-the-middle, and instead gives them a temporary alias (the session_id/access_token) that is expired by the server as a set period of time... BUT, he still can't make API requests because he doesn't have the private key you used to generate the HMAC for each request.

So even if he gets the session_id he cannot sidejack your sessions.

So yes, I think that works. Good idea!

[REPLY](#)



Felix November 14, 2011 at 2:45 am #

This is (imho) by far the best article on this topic I've read.

And I like your funny style of writing.

Thanks for this article, it's already bookmarked 😊

[REPLY](#)



Riyad Kalla November 15, 2011 at 2:50 pm #

Felix, I really appreciate the kind words man; made my day.

[REPLY](#)



Aravind November 23, 2011 at 9:19 am #

Hey Riyad, Thanks for the wonderful article. You saved me big time.

BTW, I was wondering if there are any tutorials available for securing REST with HTTPS or just pointers to get me going. Thanks.

[REPLY](#)



Riyad Kalla November 23, 2011 at 10:19 am #

Aravind, glad it helped!

Securing a RESTful web service using HTTPS is a hell of a lot easier because you can trust the security of the connection, so you no longer have to worry about eaves-droppers or men-in-the-middle, you just need a way to identify the user.

If it is a simple API, you can get away with a simple api_key that you issue to the user. If it is a robust/secure web service like a banking API, you will probably need secret credentials from the user. To avoid them needing to send the credentials in *every* request, you can have them "login" by sending their credentials once to an endpoint like /login and you give back a session_id that you keep on the server for say 30mins registered to them. Then as requests come in from that session ID you know who is making requests to you.

[REPLY](#)

**Aravind** November 24, 2011 at 2:46 am #

I have another question here. Isn't the below a possibility ? Using the basic SSL authentication mechanism and using a CA certificate. By this, we can eliminate the session id being for each request. Sorry, If I'm getting too basic here. Just trying to understand the possibilities.

REPLY**Riyad Kalla** November 24, 2011 at 6:07 am #

Aravind, I don't understand the question; can you rephrase it?

REPLY**Aravind** November 23, 2011 at 10:26 pm #

Thanks for the response Riyad. Since I'm very much a newbie to security, I was wondering how to redirect all my API traffic to https. I'm using HttpBasicAuthentication in Spring for security. After making changes to Tomcat, What are the config details that I should modify in my context to reflect the SSL settings. Please LMK

REPLY**Riyad Kalla** November 24, 2011 at 6:06 am #

Aravind, sorry I don't know what the settings are off the top of my head. The good news is that it is a very common thing to do, so I imagine stackoverflow.com should have you covered there.

REPLY**Philippe** December 16, 2011 at 5:19 pm #

Hi,

first of all, congrats and thanks for this post! It's just what I was looking for to understand the authentication dance between client/server.

I'm developing a mobile app, so I need to set up an API and of course secure it. FYI, I intend to make all the API calls over HTTPS.

There is a thing I can't figure out, and I'm sure the answer is here somewhere, but I really can't find... It is about the private key storage, and the public key generation.

You mention that there are two ways of handling the private key, either one private key by user or one private key by app version (as Twitter does, if I've well understood). So I have questions for both cases.

Case 1: One private key per user

So how do you generate the private key, and give it to the user (or device)? How and when does the user (or device) know his private and public keys?

Is it at the account creation? Does the user create his account via the mobile app, then the answer from the server is his private and public keys that I have to store on the device?

Is it during the installation of the app on the device? Is this possible?

Case 2: One private key per app version

So let's say, my app version is 1.2.0, and the private key is somewhere in the code. If I understand well, the main risk is that someone finds this private key (e.g. by decompiling)

the app) because it is shared by all the app users using the same app version?

In this case, the “hacker” can make call on the behalf of others users, if he also knows their public keys? Is this the real issue? Or do I miss something?

What's the point of using this solution (one private key per app) instead of the first one (one private key per user)? This solution seems riskier than the first one.

Can we imagine a solution using both case? A private key per version which is used to generate a private key for each user on the client and server side without the need of sending this over the wire. Because both the client and server know the “app private key” and how to generate a “user private key” (e.g, from his email or username).

Also do you have any advice of how generate private and public keys?

Sorry, If my questions are not really clear. It is because, all this is a bit new for me. I'm used to the oauth protocol because I develop an app which interacts a lot with social networks, but it's the first time I need to set up my own API.

[REPLY](#)



Kevin March 7, 2013 at 8:45 am <#>

Hi Philippe,

Am in the same dilemma as you.

1. Were you able to figure out which one is better Private Key per User vs per App?
2. If App is decompiled then the risk of exposing the keys are high.

Any thoughts or ideas?

Thanks,
Kevin

[REPLY](#)



Andy December 19, 2011 at 1:59 pm <#>

Fantastic article, Riyad! Some great insight, and you actually made reading about securing REST services entertaining – something I would have never thought possible! 😊

[REPLY](#)



Mingus January 23, 2012 at 9:48 pm <#>

Wonderful write-up, this was exactly what I was looking for.

Thank you!

[REPLY](#)



Riyad Kalla February 2, 2012 at 1:42 pm <#>

Most welcome Mingus, glad it helped!

[REPLY](#)



prabu February 3, 2012 at 4:54 am <#>

hi Riyad.

Great writeup!

I will hope to dev security framwork for restfull webservice as my Masters of Information security degree(Research project).So i need to submit project proposal to that.can you give me some sugession to do that..

thanks,
prabath.

[REPLY](#)



Raphael February 9, 2012 at 5:04 pm #

Solid article, exactly was I was looking for! Thanks for sharing such great info!

[REPLY](#)



Cristiano Fontes February 16, 2012 at 5:27 am #

Hi, very nice article I was needing this.

But it would be nice if you could point out the solutions to send the private key to the user, because I cannot find a proper way to do it without sending it thru the wire when the user creates it's account.

Another thing, what is the safest way to store that data in the user computer ? using localstorage ? cookies ?

What if the user loses it ? how to proceed ?

Thanks for the article !

[REPLY](#)



ahmet alp balkan February 16, 2012 at 10:54 am #

In the following blog post I explained how we can create REST APIs with a very simple OAuth 2 authentication.

<http://news.ycombinator.com/item?id=3599744>

I appreciate your comments on it.

[REPLY](#)



Carl Partridge February 19, 2012 at 10:39 am #

Thank you so much for this – I was just beginning to consider how to secure my new REST server and this article was comprehensive, easy-to-understand and perfect for an introduction.

Not only that, but since my web service will be mainly dealing with mobile app clients, the comments were really useful too. If anyone does get some good benchmarks on iPhone/Android performance over SSL, please let us know here.

FWIW, I decided to go with SSL in the end – and I guess I'll just have an API key that can be passed, unencrypted, as part of the querystring each time. The costs for a cert are really reasonable these days.

C

[REPLY](#)



Dima Q February 21, 2012 at 4:19 am #

Why not use SSL?

All of these issues are already addressed. Get a free cert from e.g. startcom and you can even send plaintext passwords in request if you want too, although random secure long-lived cookie is simpler and safer.

And if you really really want, you can even disable encryption in SSL and bring stack to the level of integrity and transparency that you propose.

[REPLY](#)



nbari February 27, 2012 at 8:28 am #

Hi, I am trying to find a way of making an "auth + post" in one single shot.

Based on your excellent article and on how amazon API works, I think the way to go is to share a common secret key on the client/server and on every post from the client, sign (hash_mac) the parameters with the secret key, sending as a parameters the user key timestamp and data.

So far so good, but I am a little lost on how to guarantee the integrity of the data, I can authenticate, but how, from the serve side, could I check the integrity of the submitted data?.

let me put an example, I want to use a json-rpc API for posting articles to a blog, where the article can contain, text, images and in some cases some .zip files.

The flow I am thinking will be something like this:

1. when the editor/user submits the article, build a package, with all the text, images, zip files etc.

2. create a checksum of the package.

3. post the data to the main server with something like:

```
{ authorization: user key,
signature: hash_mac($parameteres, 'secrete_key'),
checksum: sha1_file('package')
}
```

4. on the main server, search the secret key based on the authorization field, and validate the signature plus the checksums.

The goal besides been authenticated, to store on the server the same data used for creating the article and avoid possible data corruption.

Any ideas ?

[REPLY](#)



Riyad Kalla February 27, 2012 at 9:20 am #

nbari,

You have the basic gist correct; the thing you have to remember is that the hash (HMAC) you generate with the secret key has to include EVERY value that you don't want to change.

So consider this, if you just hash the params for your submission and the article text/title but only send along checksums (e.g. MD5 or SHA-1) for your images and ZIPs separately, if I am a nefarious man-in-the-middle, I can grab your images or ZIPs, repackage them with my OWN content and then send them to you with correct (new) checksums.

On the server when you to re-check the checksums on the files, they all check out because you are just re-hasing the files *I* sent you.

What you would need to do is include the binary file checksums in your HMAC calculation as well; so if ANYTHING AT ALL changed on its way to the server, when you went to re-calculate all that information server side, something would be different and you would know that someone is messing with your data and reject it.

[REPLY](#)**Learner** March 4, 2012 at 11:35 pm #

I am trying to understand. Can you please explain;

1. Why oauth2 requires params to be ordered and encoded?. (for 2-legged)

All it has to worry about is the matching signature in both the end for the given data(query string).

We can just check the signature generated using the query string right?. (e.g ?a=1&b=2). Since the signature is generated based on the secret key which is known only to the client and provider, we can only consider the query string without any ordering/encoding.

Any advantage in doing ordering/encoding and then creating the signature?

[REPLY](#)**Learner** March 5, 2012 at 12:49 am #

one more 😊

How can this signature save me from man-in-the middle attack?

If I have to make a request like this to my server from client;

increaseUserPoints?userId=1&pointsToAdd=5&appId=x&token=XYZ

Now the token XYZ will be always same in the above case. The hacker can keep posting the same request to increase the points. The server will allow since the generated token is from the given appId is same. How to handle this case?.

[REPLY](#)**Riyad Kalla** March 7, 2012 at 7:27 pm #

Learner,

Good question; the reason for ordering to be required is to avoid ANY potential inconsistency in the method used to generate the signature. It is possible that an intermediary proxy or app server modifies the query string in some way; maybe changing the case or manipulating it by adding an additional parameter, in that case it is important that OAuth clarify *exactly* what be used to generate the sig.

The other reason you might not want to just sign the entire query string as a string String is because part of the query string itself is the signature and potentially other args you *don't* want to be part of the hash. So instead OAuth defines these pedantic rules about combination and generation to avoid issues.

[REPLY](#)**Maxi Wu** March 7, 2012 at 7:37 pm #

Hi, great discussion and article.

I have only one question, "never send the private key over the wire".

Then, how would you and the server have the same private key?

1. mobile app, maybe the key comes with app installation

2. other URL base app, how do you get your private from the "system"?

does this system means the RESTful API? or some Key issuing web page like AMAZON?

[REPLY](#)



Steven Stieng April 4, 2012 at 4:09 am #

In order to get a private key, you either have to sign up for the service and retrieve it when you are logged in, or the whoever has the web service creates one for you and sends it to you by e-mail.

REPLY



Elvis Ligu March 29, 2012 at 5:15 am #

After reading your post I would like to share with you how I solved a problem I had in the past in designing some secure REST services and their corresponding API (client, and server).

Disclaimer: it was a student project and no careful design was made to solve all security issues, however it solved a lot of issues (well at least it was a good design because it was conforming with the project requirements). The application was developed in Java.

I had some REST services and I wanted them to be secure, so this is what I did:

1 – Design a login REST service:

REST is stateless so I couldn't use any session oriented API the Java could offer me. So what I did was to invent (very dummy) a session. In my design a session was just a Session object which had a session ID (uuid) and a lastAccessTime property.

I designed a session manager which was simply an object (singleton) that would manage the session. This manager's job was to create a session and keep a map that would associate sessions with session ids. An other job was to periodically check whether a session was expired by checking the session's currentTime – lastAccessTime > fixedTime. The reason why the manager would check for expired sessions was just to keep low memory footprint.

I had a service/pubKey to which the client would perform a GET request to retrieve the server's public key. Once the client has the server's public key the client issues a POST (create) request to/login service with the user's credentials (password, username), along with a client's public key.

After the client makes a POST request to the /login service the server's check user credentials if they are ok, it creates a session and associates that session with the client's public key, and sends an OK response (actually a No Content) to client along with two headers: session id, secret token. The secret token is encrypted with client's public key.

What is secret token? Each time a client performs a request to server the client sends the session id and a secret token (encrypted with server's public key). When the server receives this secret token it requires from session manager the session with the specified ID, if so the server decrypts secret token using its private key and checks to see if the current secret token that this session has is the same with that that was returned from client. In other words for each request (associated with a session) the server generates a secret token and responds to the client with this secret token (encrypted with client's public key). This way if next time the client wants to issue a request to the server he must specify the secret token (the one he got from server the last time he talked to server).

After a successful login there are two things that are happening:

- a) a session is required, and it is associated with the client's public key. Also the session's last access time is put to the current time, and a secret token (a random string) is generated for this session.
- b) the session id and the secret token (encrypted with client's public key) are sent to client (as http headers).

When the client makes a call to any secure REST service he sends the session id and the secret token (encrypted with server's public key) along with other required (from service) data.

When a request is sent to server the server decrypts the secret token and compares it with the one of the session (with the specified session id). If the secret token is equal then he proceeds with the request and responds to client sending a new secret token (encrypted with client's public key), the same session id, and the other required data (if any). Also the session's last access time is put to current time.

Note that the secret token is sent from client to server using server's public key and from server to client using client's public key. So this means that the client in order to use the token for the next response must decrypt it with its private key and encrypt it with server's public key.

Using this mechanism I was able to avoid some of the security issues that you pointed out in your post.

1) There is no way one to perform man in the middle attack: the secret token is encrypted.
 2) There is no way one to perform reply attack: for each request there is always one token so if the client already did a request an attacker can not reply the same request because the expected token (for the specified id) is a new one.

3) Secret token, is always secure and can not be sniffed, not or used from any other except the client: the secret token, when it is sent from server to client, is encrypted with the client public key, so only the client who has the corresponding private key can decrypt it.

4) No security issues with time stamp and different time zones: there is no time stamp exchanged. A session will expire only if the time elapsed from the last access exceed a fixed amount of time (let say 30 mins).

5) What in case a client wants to keep a session open for a long time, a functionality like browser-cookie where the browser preserve the session cookie? We can add another layer of negotiation (let say another REST service) where a client can instruct the server to never expire (or to chose the amount the session can stay alive) the session, so the server can persist the session and current secret token, and the client can do the same.

6) REST in peace: REST services are kept intact because they are not tight to security, their URI and or query parameters are not altered, not or the resource representation. So a developer can change service URI and query parameters, or can change the resource representation preserving the security constraints. In the same manner he can changes a service from secured to unsecured ignoring the http headers. In this case I have to say that the client doesn't presume a service is secured, he just send those headers to any service along with its request, and when the server sends back a response he just check for those headers, if they are present they are kept in order to make the next call to server.

A backward of this protocol is that it is not a standard so a browser can not directly speak to a REST service, however, if there is a javascript client available one can use AJAX to perform a request to a service. I don't know how can be done but I assume it is possible.

Conclusions: I just wanted to give my two cents about my (dummy) implementation of this specific task (securing REST services), and wanted to share that with you. I don't think my solution is the best out there, but at least it helped me to achieve what the project requirements were. However, if this solution is good or not, I would really like to see your thoughts. Thank you in advance.

Elvis.



 **CaptRespect** April 10, 2012 at 10:13 am #

So after all that about not using OAuth you end up with OAuth. 😊



 **Riyad Kalla** April 30, 2012 at 5:47 pm #

Technically 2-legged OAuth, but yes. I still thought the writeup showing the journey would be helpful since I wasn't familiar with OAuth before hand and didn't really understand the problem it was solving. So I figured I'd write the whole thing up and hope it might help someone else.



DreamMerchant September 7, 2012 at 8:36 am #



I must say very nice article and the explanation is excellent, crisp, clear and Simple. Kudos to Riyad.

What do you suggest in a scenario where you have multiple types of client like mobile/.net/php and your service is exposed using SOAP/wsHttpBinding(WCF).



AA April 21, 2012 at 4:19 pm #

Thanks, you have a gift for writing, and explained in 10mins reading what I didn't get from reading a book on OAuth! If you wrote a Kindle book on how to build a secure API using OAuth 2, I'd buy it!



Ollie Armstrong May 8, 2012 at 3:08 pm #

Would just like to thank you for this brilliant post. Was just about to pull my hair out trying to work out how to make it secure! You truly are a gem of the internet!



Yadu June 24, 2012 at 9:27 pm #

Nice post! However IMHO we should start with problem statements before putting solution.

If protocol is HTTPS then it's not possible to know real message, even if someone tries to sniff the data, because whole data in HTTP message (except hostname) is encrypted.

Said that we need to address

- Authentication
- Data Integrity (malicious user cannot understand data but can tamper it blindly)
- Replay Attacks

Solution proposed above addresses these issues.

Apart from this I got a question.

What if someone hacks the server database and finds private key (used to get HMAC hash) in plain text, unlike passwords which are supposed to be stored as salted hash?



PBS June 28, 2012 at 12:09 am #

I am designing REST based APIs and was looking for something which can throw some light on security. I found article quite interesting and detailed on same topic. Thanks for the article.

But I have a question when the article said:

"Before making the REST API call, combine a bunch of unique data together (this is typically all the parameters and values you intend on sending, it is the "data" argument in the code snippets on AWS's site)"

I have an API call like:

"DELETE /post/498578 HTTP/1.1"

There is no request parameter or body just the URI to delete.(Yes there is authentication key also but that is part of the request headers and not body!!)

I am not sure how and what data combinations can be taken from this request. Is there

any way to secure the authentication key in this case?

[REPLY](#)



Peter Wolanin June 30, 2012 at 9:43 am #

We've implemented a couple HTTP APIs that use HMAC authentication and something that's missing here and also missing in the OAuth1 2-leg and OAuth 2 MAC spec is validation of the response from the server to the client.

You can essentially just run the process in reverse for the response – the server should calculate a HMAC on the response body (or on a sha2 hash of the response body). We've set it up so the response uses the same nonce as the request, and a timestamp which is different. Send that info in a custom response header, and then the client can use the secret key to check that the response is authentically from the service and has not been tampered in transit.

A specific use case is for a service that provides a hosted search server. If an attacker could inject spam or links to malware in the search response, and those were displayed to the end user, it would be very harmful to the client.

[REPLY](#)



John July 12, 2012 at 8:18 am #

This was an excellent read, I must say. But, I am confused over one thing. The entire article revolves around HMAC hash that is sent along with the data by the client. On the server the data is re-hashed using the same private key that the client used. Then the hash generated by the server is checked to be the same as the hash that was received with the data from the client. This appears to be overly complicated.

Now, wouldn't it be easier to simply use something like openssl_encrypt (PHP) to encrypt the data using a private key (known only to client and the server) and then later decrypt it on the server using the same private key via openssl_decrypt? If needed the timestamp could be added during encryption and decryption to prevent replays. Correct me if I am wrong?

[REPLY](#)



Kanmeh West August 12, 2012 at 12:09 am #

Yo Taylor, I'm happy for you and I'mma let you finish, but Amazon Web Services has one of the largest and most used web APIs online right now!

[REPLY](#)



OSteEL September 11, 2012 at 10:17 am #

You're a hero.

[REPLY](#)



Moritz Zander September 27, 2012 at 2:32 am #

Great article! Awesome writing style.

Thanks a lot!

[REPLY](#)



Taz October 1, 2012 at 8:38 am #

An excellent article, scarily close to my thought pattern on this one.

I'm currently developing an HTML5/jQuery/Ajax WCF JSON calls,CSS mobile app that uses Kendo UI Mobile to style the interface to look like a native app, it's a neat solution although Kendo Mobile isn't quite finished yet 😊

I have the same issue, since it's a single page model the secretkey is exposed via the javascript code and session variables are horribly exposed (sessionStorage.getItem etc).

I have two WCF Rest services, one for authorization and one for the data services, it's all SSL and currently the login validation which is done against AD returns a token, just a guid.

The idea was to hold that key as a session variable and submit it to the data service in the headers and validate, it's already messy as I had to take care of CORS request after ditching JSONP calls.

I plan to hold the key and the users ID in a table on the server and check against that on data service requests and maybe have an expiry time for the key.

Would this be enough using SSL to reasonably protect the WCF service? all the calls are read-only thankfully in this app.

I've I'm leaving a big security hole here then please let me know, this is the 1st type of app using WCF Rest and a mobile client I've done.

Great blog and great post.



Jay Gridley October 24, 2012 at 6:46 am #

Very nice article. Thanks for it. But I have some questions... I would like to know, how long the private key and hashed payload have to be to hmacSHA256 hash algorithm work as designed. Or it does not matter? Thanks



Riyad Kalla October 27, 2012 at 5:32 am #

Jay, good question, it doesn't matter. The SHA256 algorithm will output a fixed-length result regardless of input.



Quinn November 15, 2012 at 3:07 pm #

bookmarked!! , I like your blog!

my web-site; [Ratings And Reviews](#)



JPSavard December 8, 2012 at 11:02 am #

In a situation where I have a Web Client to my Public API, and lets say I use the member password of a site as a private key and doesn't want my client to explicitly login and type his/her username/password everything they open the client application, how should I deal with the secret key(password) ?

Should I store it in the cookies at least for the first login(this is a vulnerable time frame)?

Thanks in advance!

[REPLY](#)



JPSavard December 8, 2012 at 11:03 am <#>

In a situation where I have a Web Client to my Public API, and lets say I use the member password of a site as a private key and doesn't want my client to explicitly login and type his/her username/password everything they open the client application, how should I deal with the secret key(password) ?

Should I store it in the cookies at least for the first login(this is a vulnerable time frame)?

Thanks in advance!

[REPLY](#)



Shafi_trumboo December 24, 2012 at 1:06 am <#>

I have the same question as asked by learner???

How can this signature save me from man-in-the middle attack?

If I have to make a request like this to my server from client;

increaseUserPoints?userId=1&pointsToAdd=5&appId=x&token=XYZ

Now the token XYZ will be always same in the above case. The hacker can keep posting the same request to increase the points. The server will allow since the generated token is from the given appId is same. How to handle this case?.

[REPLY](#)



Lalit January 2, 2013 at 7:04 am <#>

Excellent work!

I like your style of explaining...its simple yet effective...what a writeup.

You have just explained the whole security thing so nicely, that even a newbie can secure his endpoints without any issue and that even using any technology....Great work!!!

Really helped me a lot

[REPLY](#)



Jason McCreary January 2, 2013 at 11:04 am <#>

I noticed this asked a few times without a clear answer. I too have the same question:

How do we *send* the user their private key. Consider a distributed mobile app. Until the user logs in, it won't have the private key. Clearly I don't want to *send* the private key back in plain text. I could use their credentials as the private key. But now login is a weak link.

As noted, SSL seems to solve most of these issues. But I'm curious how a private key is *sent* in this case...

[REPLY](#)



JSDesigner January 22, 2013 at 8:46 am <#>

To my belief you should add the content of an http request to your signature.

We are talking about POST/PUT scenarios obviously.

Unfortunately I'm really really unable to read the content out of a request neither in a DelegatingHandler nor in a Filter. This is due to the fact that the request can only be read once – and apparently it already has been read. So here's my question.

1. What am I doing wrong here... am I supposed to read the content in a DelegatingHandler?
2. As a workaround I imagine you could build yet another hash (solely) over the content and add that one to the header too. This way I could add the content hash to my HMAC signature.

[REPLY](#)



Darren Mart January 29, 2013 at 3:42 pm <#>

You have an excellent writing style Riyad. I've got a long way to go before I've wrapped my head around OAuth and/or other options for securing my web APIs, but your article was the first I've found that was a pleasure to read from beginning to end. The comments have been very valuable also; the questions and follow-ups have echoed my own questions and concerns along the way.

Very well done, kudos!

[REPLY](#)



Yoshi February 27, 2013 at 8:11 pm <#>

I know I'm late to the party on this one, but how are the private keys best distributed in the 1st place? You say they can be disabled based on a compromise, and need to be updated, so say it's a mobile device how are they updated?

Do you just inherently take the risk here and make sure it's over SSL?

[REPLY](#)



Dallin Skinner March 1, 2013 at 1:13 pm <#>

Thank you for the article. I found it very helpful in securing my own API. Well-written and broken down in easy to digest steps. Once again, thank you.

In addition to this I also wrote my own blog post using an analogy with cupcakes that may help explain the concept to a non-technical user.

<http://verisage.us/en/blog/2013/02/27/securing-restful-api-hmac/>

[REPLY](#)



Kondur March 8, 2013 at 10:33 am <#>

Twitter API keys & secrets have been compromised. Verified all the keys that are mentioned in the Gist. They all work. Again brings the interesting debate whether it is safe to bundle the keys & secrets within a mobile or desktop clients. As they can be decompiled & exposed.

Will be interesting to see how Twitter respond to this.

<http://www.readability.com/articles/27khdlzy>

[REPLY](#)

Mike Miller April 10, 2013 at 3:15 pm <#>



Great posting, and I really enjoyed the writing style – so much better than a lot of the dry technical stuff you find other places.

[REPLY](#)



Riyad Kalla April 30, 2013 at 6:52 am #

Thanks Mike!

[REPLY](#)



fvisticot April 18, 2013 at 12:52 am #

Really great article !! and very useful

I think I have understood the mechanism when used with “standart application (mobile or not) BUT it seems that this mechanism can not work for a pure Javascript client (what I would like to do...)

- How to keep the private key “secure” in a pure HTML5 app ? ... everybody can open a console and read the private key (who is part of the javascript code 😕)
- Is it possible to detail a “Javascript solution” and explain how to secure the API in a pure Javascript client... My first impression is that using HTTPS is the more simple and secure approach.... Is it correct ?
- Another approach is to use the presented solution (private and public keys) and obfuscate the javascript (by using JScrambler or equivalent) .. does it make sense ?

[REPLY](#)



Riyad Kalla April 30, 2013 at 6:51 am #

You are exactly right; in a pure HTML5 (JS/CSS/HTML) app, there is no protecting the key. You would do all communication over HTTPS in which case you wouldn't need a key since you could safely identify a client using a standard API_KEY or some other friendly identifier without the need or complexity of an HMAC.

Obfuscating the JS won't help if someone wants to get to your key; so just use HTTPS and you are all set.

[REPLY](#)



Nik Martin May 2, 2013 at 11:05 am #

You handle this by not storing the API key anywhere but memory. Provide a Login form that does a round trip to the server like a normal login, but use the user's password as the secret API key, and store in the client's browser as a variable. This fiddle shows the use of HTML5 localStorage to achieve this:

<http://jsfiddle.net/Sdx4F/>

[REPLY](#)



Miqdad May 2, 2013 at 5:43 am #

Hi,

Could you please help me to write a class which will communicate with linkedin api without oauth class ?

Thanks is advance

[REPLY](#)**Nik Martin** May 2, 2013 at 10:56 am <#>

FYI, Cloudstack's API is exactly like this. One thing that you might mention, when generating hashes to sort all fields alphabetically before hashing with the key on both sides (client and server) this will ensure that you are actually hashing the same data on both ends. Some Javascript code (requires a crypto module):

<https://gist.github.com/nikmartin/5499838>

[REPLY](#)**Nik Martin** May 2, 2013 at 10:58 am <#>

FYI, Cloudstack's API is exactly like this. One thing that you might mention, when generating hashes to sort all fields alphabetically before hashing with the key on both sides (client and server) this will ensure that you are actually hashing the same data on both ends. Some Javascript code (requires a crypto module):

<https://gist.github.com/nikmartin/5499838>

[REPLY](#)**Matt D** May 13, 2013 at 7:08 am <#>

Great article.

[REPLY](#)**Matt G** June 11, 2013 at 8:47 am <#>

Great article – looks like something that I need, but I'm rather new to all of this...

One confusion for me though: if my client is in javascript, how can I securely combine the parameters with the private key and keep it private? Anyone will be able to read the javascript and discover the private key...

[REPLY](#)**Khan** June 12, 2013 at 4:09 am <#>

I am on the same boat, can some body reply to the question raised please?

[REPLY](#)**Yoshi** June 13, 2013 at 9:15 pm <#>

I had similar questions when I first read this. A lot of what I've read since then is that this sort of thing is great for app to server connections where a predefined key can be put in the code and protected somehow, or via the use of an encrypted keychain of some sort, but that it can tend to fall down when used for client uses where the private key can be not so private, or even variable.

Basically the private key needs to be secure and known to both parties before the communication begins.

If it's in a web browser or something then you could investigate setting a localstorage variable when they put in their password (if that's where you're heading) that is scoped for your site and then having the code reference that variable rather than hard coding the key in. If it's not a password but rather a proper pre-shared key that is static then that's not a problem unique to this. At least mobile devices have some sort of secure keychain for this sort of thing.

[REPLY](#)



Matt G June 13, 2013 at 10:15 pm #

Many thanks for those pointers Yoshi, I will have a look and see if I can do something along those lines.

[REPLY](#)



JM May 16, 2013 at 3:40 pm #

Hi guys! Great article. Some of you lost me, though, when you wrote about how using two-legged OpenAuth is easy. I'm struggling with that part.

Is there some sample code out there for that? Everything I see is three-legged, and uses SSL certs (was looking at DotNetOpenAuth, but the authorization server component loads certs. From what I understand, if I only need two-legged auth, and I already have to have certs, shouldn't I just go ahead with SSL (if I already have to have the certs anyway?)

[REPLY](#)

Trackbacks/Pingbacks

1. [Understanding the Unix Epoch \(in Java\), Time Zones and UTC](#) - April 27, 2011

[...] I get different values, does that mean when I am writing a web service or REST API, I need the client to send me its timestamp as well as its timezone so I can properly [...]

2. [My WCF4 REST Service and Entity Framework with POCO trip \(5\)-identity authentication](#) - September 22, 2011

[...] put it realize perfect, you will find yourself a and OAuth exactly the same things out, for exampleThe elder brothersAnd he was very excited to declare their invented a need not OAuth identity verification, results in [...]

3. [AWS Elastic Load Balancer sends 2 Million Netflix API Requests to Wrong Customer](#)

- October 29, 2011

[...] example, if you are deploying a RESTful web service, consider securing it using 2-legged OAuth or an HMAC or for a standard web application, only allow logins over [...]

4. [Designing a Secure REST API with OAuth2 You Can Be Proud Of | ahmet alp balkan : blog](#) - February 16, 2012

[...] Designing a Secure REST API with OAuth2 You Can Be Proud Of Yesterday there has been a popular post on Hacker News about Designing Secure REST API without OAuth. I don't agree that OAuth is that difficult to use and I'll introduce my way shortly. This post is intended to be a reply on this topic. [...]

5. [Interesting Finds: February 16, 2012 « Hank Wallace](#) - February 16, 2012

[...] Getting started with designing a Secure REST (Web) API – Riyad Kalla Modern Web Development – Part 5 – Shawn Wildermuth Camels, XAML and Silverlight is dead- Long live Silverlight 5! – Surf4Fun SQL AZURE LOST ITS LEASE! EVERYTHING MUST GO! – Brent Ozar What If You're Not CEO Material? – David Cohen What makes the Boston startup scene special – Jeff Bussgang Share this:FacebookTwitterStumbleUponDiggRedditLike this:LikeBe the first to like this post. [...]

6. [My most important Twitter Messages #13 - Flash, Programming, Interaction | der](#)

hess - February 21, 2012

[...] RT Great article summing up how to implement a secure authentication scheme for REST based web services [...]

7. API - Techniques | Pearltrees - March 19, 2012

[...] Designing a Secure REST (Web) API without OAuth [...]

8. Learning | Pearltrees - April 4, 2012

[...] You realize that hashing the password and sending the hash over the wire in lieu of the plain-text password still gives people sniffing at least the username for the account and a hash of the password that could (in a disturbing number of cases) be looked up in a Rainbow Table . Designing a Secure REST (Web) API without OAuth [...]

9. 无需 OAuth, 设计一个安全的 Rest (Web) API (未完待续) | MyOwnSpace - May 8, 2012

[...] 原文在此 : <http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/> Situation [...]

0. Simply using HTTPS to secure your APIs | MyOwnSpace - May 10, 2012

[...] <http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/>
[...]

1. Quick and Dirty REST Security (or Hashes For All!) at blog.phpdeveloper.org - May 14, 2012

[...] for something a bit more lightweight (and simpler to implement a bit more quickly) I came across this older article with a suggestion of a private key/hash combination. I figured that could do the job nicely for a [...]

2. ushi » atropa API & Key based authentication - May 24, 2012

[...] spannender Punkt an einer API ist die Authentifizierung – Nach ein wenig stöbern, bin ich auf diesen Artikel gestoßen, der die grundsätzliche Funktionsweise einer schlüsselbasierten [...]

3. Securing my Node.js app's REST API? | PHP Developer Resource - May 28, 2012

[...] <http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/>
[...]

4. Securing access to PHP API | active questions php - July 5, 2012

[...] have read this article but I am unsure what they mean when they [...]

5. » How to REST and not die trying – Writing API's the REST way Commented Out -

August 8, 2012

[...] Custom techniques [...]

6. REST API for website which uses Facebook for authentication - feed99 - August 21, 2012

[...] methods as Basic Auth over HTTPS, as there are no credentials for a user as such. Something like this seems to be only for authenticating applications using the [...]

7. Secure API | Code Link Archive - August 28, 2012

[...] <http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/>
[...]

8. How do I secure my REST API? | Jisku.com - Developers Network - September 9, 2012

[...] second option is to follow the path of Amazon Web Services, where i use public/private key authentication. When a user registers i use a https api to save [...]

9. La veille du week-end (quarante-troisième) | LoïcG - September 14, 2012

[...] Designing a Secure REST (Web) API without OAuth : via @OStéEL [...]

0. RESTful APIs with Node.js and Express.js, and Backbone.js client side « Picanteverde - October 30, 2012

[...] <http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/>
[...]

1. Distributed Weekly 184 — Scott Banwart's Blog - December 7, 2012

[...] Designing a Secure REST (Web) API without OAuth [...]

2. 2-legged OAuth PHP implementasyonu | Aykut Farsak - December 24, 2012

[...] geldi bu örnek uygulamaya 2-legged OAuth implementasyonuna. Designing a Secure REST (Web) API without OAuth adresinde yer alan çalışma yöntemini uygulayalım. Öncelikle akışı [...]

3. IneatConseil » RESTful Authentication - January 29, 2013

[...] Designing a Secure REST (Web) API without OAuth
: <http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/> [...]

4. Mobile Apps | Verisage | Blog - March 1, 2013

[...] Designing a Secure REST api without Oauth [...]

5. Authenticating RESTful web applications | on Code - April 5, 2013

[...] additional security (such as timestamps to mitigate relay attacks) to this, and I recommend reading this excellent tutorial which sets out many of [...]

6. Restful web service API user authentication on every single call : Android

Community - For Application Development - April 7, 2013

[...] use HMAC-SHA256 to send a hash of the URL params over to the server. Amazon does that and there is an article that covers how this is done. It is not as complicated as [...]

7. Secured RESTful API that can be used by Web App (angular), iOS and Android |

BlogoSfera - May 6, 2013

[...] the API like Amazon S3 as described in this great article. In short, he says that both server and client would know of a private key, which they would use to [...]

8. Two legged OAuth authentication with Web API | BlogoSfera - May 12, 2013

[...] my approach to be similar to how Amazon Web Services does their authentication, and there's a great article on The Buzz Media that specifically describes how they do their authentication at a high level. I [...]

9. Secured RESTful API that can be used by Web App (angular), iOS and Android -

Tech Forum Network - July 22, 2013

[...] the API like Amazon S3 as described in this great article. In short, he says that both server and client would know of a private key, which they would use to [...]

0. HMAC REST API Security | 9bit Studios - July 29, 2013

[...] Designing a Secure REST (Web) API without OAuth [...]

Leave a Reply

Name (required)

Mail (will not be published) (Required)

Are you human? (This helps block spam) *

- 3 = 6

Notify me of new comments

MOST POPULAR TAGS

Android announcement Apple
awesome bizarre buggy cell
phones deals E3 funny game
gameplay Google Guitar Hero
HDTV health Humor and
Fun iPhone Java law Linux
Microsoft movie Nintendo
opinion performance podcast
politics PS3 release
review screenshots
security shenanigans
software development Sony T-
Mobile the Gentlemen Radio tip trailer
Video Video Games Wii Xbox
Xbox 360

© 2013 The Buzz Media. All Rights Reserved.