THE OVERFLOW

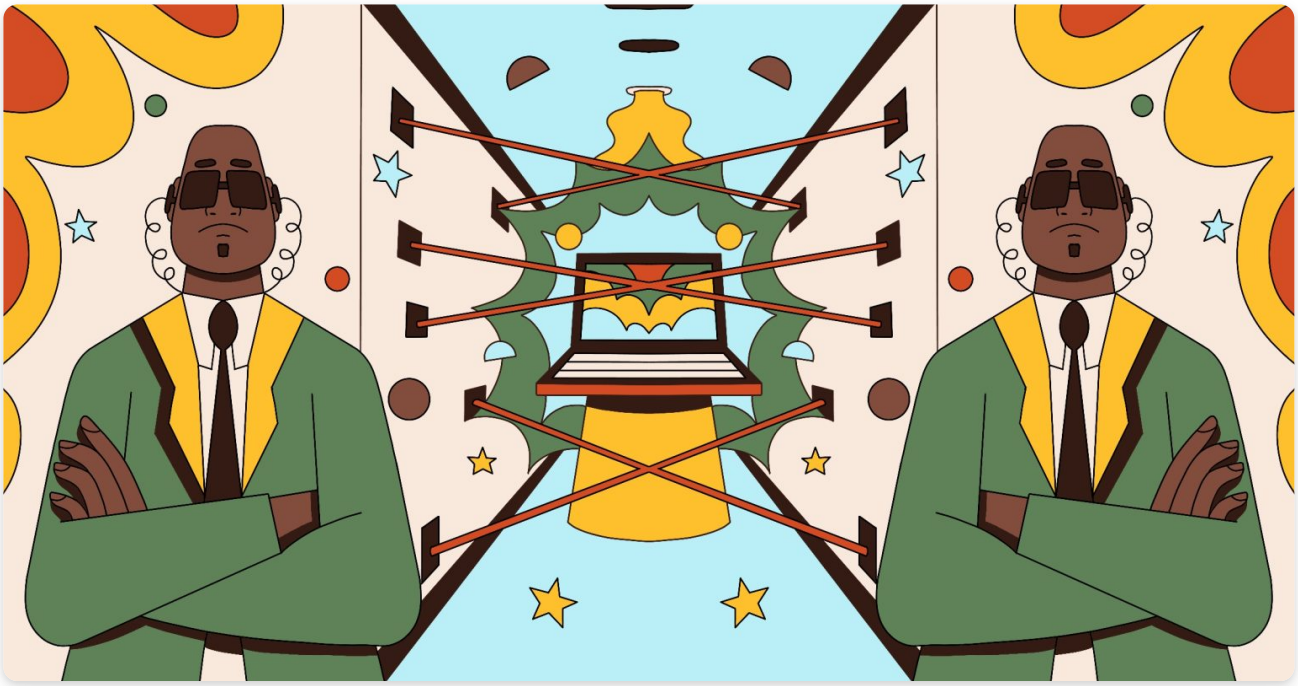
Essays, opinions, and advice on the act of computer programming from Stack Overflow.

code-for-a-living OCTOBER 6, 2021

Best practices for REST API security: Authentication and authorization

If you have a REST API accessible on the internet, you're going to need to secure it. Here's the best practices on how to do that.

Sam Scott and Graham Neray



Most apps that use a modern web framework will have one or more REST APIs. REST is a simple and flexible way of structuring a web API. It's not a standard or protocol, but rather a set of architectural constraints.

There are three reasons you might find yourself writing a REST API:

- To give a networked client that you built—for instance, a single-page app in the browser or on a mobile app on a phone—access to data on your server.
- To give end users, both people and programs, programmatic access to data managed by your application.
- To let the many services that make up your app's infrastructure communicate with each other.

Any API built for these reasons can be abused by malicious or reckless actors. Your app will need an access policy—who can view or modify data on your server? For instance, only the author [*Editor's note: the editors, too]* of a blog post should be able to edit it, and readers should only be able to view it. If anyone could edit the post you're reading, then we'd get vandals, link farmers, and others changing and deleting things willy nilly.

This process of defining access policies for your app is called authorization. In this article, we'll show you our best practices for implementing authorization in REST APIs.

Always use TLS

Every web API should use TLS (Transport Layer Security). TLS protects the information your API sends (and the information that users send to your API) by encrypting your messages while they're in transit. You might know TLS by its predecessor's name, SSL. You'll know a website has TLS enabled when its URL starts with `https://` instead of `http://` .

Without TLS, a third party could intercept and read sensitive information in transit, like API credentials and private data! That undermines any of the authentication measures you put in place.

TLS requires a certificate issued by a certificate authority, which also lets users know that your API is legitimate and protected. Most cloud providers and hosting services will manage your certificates and enable TLS for you. If you host a website on Heroku, enabling TLS is a matter of clicking a button. If you host on AWS, AWS Certificate Manager combined with AWS Cloudfront will take care of you. If you can, let your host manage your certificates for you—it means no hassle at all and every API call will be automatically secured.

If you're running your own web server without any third-party services, you'll have to manage your own certificates. The easiest way to do this is with Let's Encrypt, an automated certificate

authority. Let's Encrypt has a helpful getting started guide.

Use OAuth2 for single sign on (SSO) with OpenID Connect

Nearly every app will need to associate some private data with a single person. That means user accounts, and that means logging in and logging out. In the past, you may have written login code yourself, but there's a simpler way: use OAuth2 to integrate with existing single sign-on providers (which we'll refer to as "SSO").

SSO lets your users verify themselves with a trusted third party (like Google, Microsoft Azure, or AWS) by way of token exchange to get access to a resource. They'll log in to their Google account, for instance, and be granted access to your app.

Using SSO means that:

- You don't have to manage passwords yourself! This reduces the user data you store and therefore less data to be exposed in the event of a data breach.
- Not only do you avoid implementing login and logout, but you also avoid implementing multi-factor authentication.
- Your users don't need a new account and new password—they've already got an account with an SSO provider like Google. Less friction at signup means more users for you.

OAuth2 is a standard that describes how a third-party application can access data from an application on behalf of a user. OAuth2 doesn't directly handle authentication and is a more general framework built primarily for *authorization*. For example, a user might grant an application access to view their calendar in order to schedule a meeting for you. This would involve an OAuth2 interaction between the user, their calendar provider, and the scheduling application.

In the above example, OAuth2 is providing the mechanism to coordinate between the three parties. The scheduling application wants to get an access token so that it can fetch the calendar data from the provider. It obtains this by sending the user to the calendar provider at a specific URL with the request parameters encoded. The calendar provider asks the user to consent to this access, then redirects the user back to the scheduling application with an authorization code. This code can be exchanged for an access token Here's a good article on the details of OAuth token exchange..

You can implement authentication on top of OAuth2 by fetching information that uniquely identifies the user, like an email address.

However, you should prefer to use OpenID Connect. The OpenID Connect specification is built on top of OAuth2 and provides a protocol for authenticating your users. Here's a getting started guide on OAuth2 with OpenID Connect.

Unfortunately, not every identity provider supports OpenID Connect. GitHub, for instance, won't let you use OpenID Connect. In that case, you'll have to deal with OAuth2 yourself. But good news—there's an OAuth2 library for your programming language of choice and plenty of good documentation!

Tips for OAuth

You can use OAuth2 in either *stateless* or *stateful* modes. Here's a good summary on the differences.

The short version: it is typically easier to *correctly* implement a stateful backend to handle OAuth flows, since you can handle more of the sensitive data on the server and avoid the risk of leaking credentials. However, REST APIs are meant to be stateless. So if you want to keep the backend this way, you either need to use a stateless approach or add an additional stateful server to handle authentication.

If you opt to implement the stateless approach, make sure to use its Proof Key for Code Exchange mode, which prevents cross-site request forgery and code injection attacks.

You'll need to store users' OAuth credentials. Don't put them in local storage—that can be accessed by any JavaScript running on the page! Instead, store tokens as secure cookies. That will protect against cross-site scripting (XSS) attacks. However, cookies can be vulnerable to cross-site request forgery (CSRF), so you should make sure your cookies use `SameSite=Strict`.

Use API keys to give existing users programmatic access

While your REST endpoints can serve your own website, a big advantage of REST is that it provides a standard way for other programs to interact with your service. To keep things simple, don't make your users do OAuth2 locally or make them provide a username/password combo—that would defeat the point of having used OAuth2 for authentication in the first place. Instead, keep things simple for yourself and your users, and issue API keys. Here's how:

1. When a user signs up for access to your API, generate an API key:

```
var token = crypto.randomBytes(32).toString('hex');
```

2. Store this in your database, associated with your user.

3. Carefully share this with your user, making sure to keep it as hidden as possible. You might want to show it only once before regenerating it, for instance.
4. Have your users provide their API keys as a header, like

   ```
   curl -H "Authorization: apikey MY_APP_API_KEY"
   https://myapp.example.com
   ```
5. To authenticate a user's API request, look up their API key in the database.

When a user generates an API key, let them give that key a label or name for their own records. Make it possible to later delete or regenerate those keys, so your user can recover from compromised credentials.

Encourage using good secrets management for API keys

It's the user's responsibility to keep their secrets safe, but you can also help! Encourage your users to follow best practices by writing good sample code. When showing API examples, show your examples using environment variables, like `ENV["MY_APP_API_KEY"]`.

```
from requests.auth import HTTPBasicAuth
import requests
import os

api_key = os.environ.get("MY_APP_API_KEY")
auth = HTTPBasicAuth('apikey', api_key)
req = requests.get("<https://myapp.example.com>,
        headers={'Accept': 'application/json'},
        auth=auth)
```

(If you, like Stripe, write interactive tutorials that include someone's API key, make sure it's a key to a test environment and never their key to production.)

Choose when to enforce authorization with request-level authorization

We've been speaking about API authorization as if it will apply to every request, but it doesn't necessarily need to. You might want to add *request-level authorization*: looking at an incoming request to decide if the user has access to your resources or not. That way, you can let everyone see resources in `/public/`, or choose certain kinds of requests that a user needs to be authenticated to make.

The best way to do this is with request middleware. Kelvin Nguyen over at Caffeine Coding has a nice example here.

Configure different permissions for different API keys

You'll give users programmatic API access for many different reasons. Some API endpoints might be for script access, some intended for dashboards, and so on. Not every endpoint will need the user's full account access. Consider having several API keys with different permission levels.

To do this, store permissions in the database alongside the API keys as a list of strings. Keep this simple at first: "read" and "write" are a great start! Then, add a request middleware that fetches the user and the permissions for the key they've submitted and checks the token permissions against the API.

Leave the *rest* of the authorization to the app/business logic

Now that you've started adding authorization to your API, it can be tempting to add more and more logic to handle more checks. You may end up with nested if-statements for each resource and permission level. The problem with that is that you may end up duplicating application logic. You'll find yourself fetching database records in the middleware, which is not ideal!

Instead, leave that level of authorization logic to your application code. Any authorization checks made on resources should happen in the app, not in the middleware. If you need to handle complex authorization logic in your app, use a tool like Oso, which will let you reduce your authorization policy to a few simple rules.

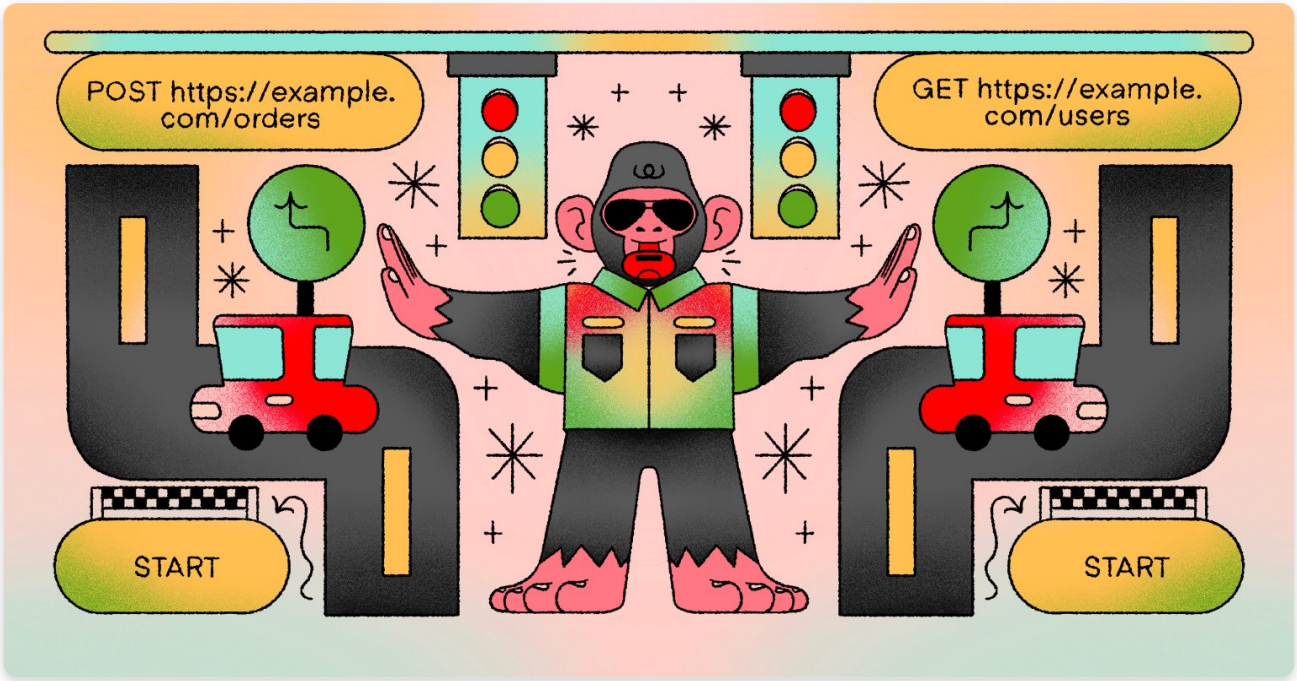
There's always more to discuss with authentication and authorization, but that's enough to get started! We hope these tips help you design useful and secure API endpoints.

In summary: use good libraries

We've given you plenty of specific advice, but it all comes back to one point—try to offload as much work as you can to trusted libraries. Authorization is tricky, and we'd like to minimize the number of places in which we can make a mistake. You have plenty of great tools at hand to help with authorization, so make the best use of them that you can! Much like with cryptography: study up, and then do as little as possible yourself.

Tags: api, authentication, authorization, rest api

Related



code-for-a-living MARCH 13, 2023

Building an API is half the battle: Q&A with Marco Palladino from Kong

API gateways, service mesh, and GraphQL, oh my!

Ryan Donovan

20 Comments

TylerH 6 Oct 21 at 3:10

“Store this in your database, associated with your user. You can keep this in plain text—it’s not re-used like passwords are.” and “Carefully share this with your user, making sure to keep it as hidden as possible. You might want to show it only once before regenerating it, for instance.” seem contradictory to one another, especially considering you then have an entire section on keeping API keys secret.

Reply

Wayne Conrad 7 Oct 21 at 8:33

I agree. API keys are usually persistent, so wouldn’t I want to store a salted hash, like with any password? But this statement: “it’s not re-used like passwords are” doesn’t seem true to me. That doesn’t jive with the client code example or with how API keys are usually used.

Reply

Carles 8 Oct 21 at 12:36

I think they mean that the same API key won’t be used in another site as a passwords would. I would store it as a salted hash too, better be safe than sorry…

Also, I still think you can use one of the useless oAuth flows rather than an API key. I think it’s better to rely on these open and trusted protocols.

Reply

Carles 10 Oct 21 at 3:14

I obviously meant userless, not useless 😀

Reply

Sam 11 Oct 21 at 1:01

Hey all,

Thanks for the comments/feedback!

The original rationale was: if someone has breached your app + DB, then leaking the API keys is probably no worse than all the other data the attacker would have access to! Although it does potentially make it a persistent breach, and it could be a path to escalating the attack to other services.. So depending on your threat model it's more or less necessary.

That's quite a nuanced point, so I was hoping to keep it simple here. But I can see that it was going too far the other way, in suggesting there is no need to secure them.

I agree that you might as well hash them (even unsalted is fine — API keys should be high entropy and not repeating anyway).

We've updated the article to remove the misleading sentence.

Reply

Wayne Conrad 12 Oct 21 at 2:57

Thanks for your response and update. You've given me something to think about re salting of high entropy keys.

n0body 8 Oct 21 at 3:32

>> Using SSO means that:: Your users don't need a new account and new password—they've already got an account with an SSO provider like Google. Less friction at signup means more users for you.

Force me to signup through google, or facebook == more friction and guarantees I won't be one of those users

Reply

Tennison 13 Oct 21 at 10:31

You would have to signup to the site either way, so whether it's custom signup or Google's doesn't change that. The rationale for using SSO such as Google is that most people already have Google accounts so makes it easier to provide SSO auth.

Reply

Acing 1 Jul 22 at 12:30

If you prefer not to use a third-party sso provider, I recommend you to take a look at the "Keycloak" which is an open-source sso provider that can be hosted on your server. That will be more private if you are supposed to keep the user's privacy not to be shared with other company e.g. Google or facebook

Reply

Bane Sekulic 13 Oct 21 at 4:26

I disagree. Facebook and all the other "giants" went offline for more than couple of hours in a span of a few days, and trust me when you client has a rush hour on their end, and your app has not provided them with a set of a credentials in your system, then good luck to you! It's a fool errand seeing how everybody forces this third party services for authentication, when simple JWT implementation is what will do the trick.

Reply

Sam 14 Oct 21 at 1:58

Hey Bane! Thanks for the comment.

It's tough writing anything on security best practices, because everything is a tradeoff.

Overall, I'd say that if you are capable of beating Google/Facebook on availability, and consider implementing an auth system with JWTs securely to be simple, then this probably isn't the post for you! The recommendations here were designed to allow someone without security expertise to get something working safely.

Unfortunately, in my opinion the tooling to do that yourself just isn't there yet. For example, JWTs themselves have an entire RFC of best practices to apply: https://datatracker.ietf.org/doc/html/rfc8725.txt

Reply

Rudiger 24 Dec 22 at 5:20

You don’t need to beat Google/Facebook on availability. You just have to be 100% available whenever your REST API is available. Chances are they run on the same backend anyway.

Reply

User 13 Oct 21 at 8:54

Why not use JWT with ECDDA signatures instead of API tokens that have to be stored? JWT provides ways to store extra info in a stateless way so not only can you avoid the database entirely you can also add scope and time to live values to better manage the token.

Reply

Sam 14 Oct 21 at 2:04

Hey! Thanks for the comment.

The short answer: I wanted to provide advice that people could follow without entering into a world of tradeoffs and implementation concerns! And I don’t think implementing JWTs securely is a simple matter (see my other comment on that).

The longer answer: there are a few reasons making JWTs work for session is not ideal. For example, I’d argue implementing token revocation for API tokens is an absolute _must_. Which means storing a revocation list and checking it… which leads to hitting the database. Here’s a nice (if a bit sarcastic!) article on that: http://cryto.net/~joepie91/blog/2016/06/19/stop-using-jwt-for-sessions-part-2-why-your-solution-doesnt-work/

Reply

Max 17 Oct 22 at 6:50

Thanks for the article (and the flow chart). I am curious on how you would solve this (my) problem:
I have a website, which (client side) connects to a backend using websockets. The website

should not need a login and is open for everyone to use, but I want to stop people from accessing the api without using the website. I thought about creating JWT on the webserver with the IP of the request or handing out short-lived API tokens. What would you suggest?
I thought about this after seeing websites with YT subsciber counters full of ads, but the backend with the acutal subscriber counter is reachable from any server/website

Reply

Aiton C 8 May 23 at 3:20

I think CORS can do the trick if you just want to make sure only your API only allows your website to connect to it you can simply restrict the cross-origin policy to only allow connections from your websites domain URL

Reply

pa7 6 Nov 21 at 7:35

Ok. all OK

The longer answer: there are a few reasons making JWTs work for session is not ideal

Reply

Raja R 30 Nov 22 at 2:56

Anyone please suggest- How to secure Dspace Asset url

Reply

Jignesh K 27 Dec 22 at 10:53

How storing credentials or keys in an environment variables (on the host) is safe as compared to storing these keys in the application itself? I am pointing to the below piece of code from the above tutorial:
`api_key = os.environ.get(“MY_APP_API_KEY”)`

Reply

Dennis 13 Jan 23 at 11:04

It keeps the credentials/key(s) from being stored in a repository. That way you decouple your codebase from your credentials. Say for example your source control server is breached, at least your credentials are still secure. (Then again, for security reasons you would change them anyway after such an event.)

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment