API Security

A guide to building and securing APIs from the developer team at Okta

with a foreword by

Les Hazlewood

API Security

A guide to building and securing APIs
from the developer team at Okta

Foreword by Les Hazlewood

With chapters by

Lee Brandt

Keith Casey

Brian Demers

Joël Franusic

Sai Maddali

Dave Nugent

Matt Raible

API Security

by the Developer Team at Okta

Published by Okta, Inc. 301 Brannan Street, San Francisco, CA, 94107

ISBN: 978-1-387-78313-7

18122.2124

First Edition

° ° ° ° °

Table of Contents

Acknowledgments

Hello World

Foreword

by Les Hazlewood

I entered the world of information security almost 20 years ago, as often occurs in our industry, by accident. I was a software engineer excited to be working on a very large software system that was being created for the New York Port Authority - right after 9/11. The system was complex and the problems we were solving were genuinely interesting and intellectually gratifying. And because of our customer, keeping the system and its users secure was of the utmost importance. I went from being somewhat naive about how applications were secured to being thrust into a team that was responsible for securing one of the most important computer systems in the country given events at the time. It made software security very real - in a very concrete in a way I hadn't experienced before. When the project was done, I was proud to have helped in some small way to making New York a safer place.

I learned immensely from that experience, and realized how much technology then and since was advancing at break-neck speeds. The learning curve was really steep and all the while, technology was advancing rapidly, and you constantly had to keep learning. It's easy to forget about security when you've got 20 other things to learn just to get an application out the door! But if I take a step back and look at our industry with a wider perspective, I can't help but be intrigued by this exponential growth and innovation - and how security will always play a part.

Humanity has always had the drive to innovate, but we've also been determined to undermine our own advancements for selfish gain through surreptitious means. As a result, it is incumbent on us, the builders and innovators of the world, to protect ourselves. From the Mesopotamian potter 3500 years ago who wanted to keep his glazing techniques secret from competitors to modern banks who safeguard the world's digital financial transactions - there has always been a need to keep information secret. And there have always been people trying to steal those secrets.

What's important about this dichotomic dance between information holder and information thief is that the dance never ends - a safeguard today will eventually be bypassed tomorrow, which then must be be supplanted by a better safeguard. Unfortunately, even smart, capable people and corporations forget this or even ignore it, which is why we have the Equifax and Yahoo data breaches of the world.

So what does this mean in the current climate of exploding connectivity between millions of devices in the world, and the HTTP APIs that are shared and consumed between them?

To put it simply, it's the Wild West out there!

Of course, no one expects you to wear leather chaps and ride a horse to work (but if that's your thing, you do you, and do it proudly!), but we software developers are constantly looking down the barrel of a hacker's metaphorical six-shooter.

And while a bit tongue-in-cheek, the Wild West metaphor is valid - the Western frontier in the United States' early years was expanding and changing quickly, and law enforcement often wasn't available. Individuals and companies had to protect themselves using the best strategies and technologies at their disposal. Similarly, our computer and information technology industry is soberingly new - the first digital computer was invented only 70 years ago, in the time of a single human life span! Our still-nascent industry clearly reflects the same opportunity to expand and build, and for some, to engage in nefarious activity.

So what about us? The web and mobile application developers? The API developers? How do we address this expanding frontier when even massive companies fall victim every day?

I believe the answer is that we be smart, informed, and proactive. We focus on known best practices and never stop looking for new ones. We implement modern approaches that have been proven successful in real, practical experience. We stay diligent and learn from those who have come before us, thus standing on the shoulders of giants, like the authors of the book in your hands right now.

So you don't have to be afraid of the Wild West. There are amazing opportunities ahead and the Internet is still the Great Equalizer for today's builders and innovators. And for the API builders among you? Armed with the wonderful information in this book, I'm confident you'll be ahead of (and safer than) 99% of all other APIs today.

Head 'em up, move 'em out!

Chapter 1

Transport Layer Security

By Dave Nugent

Chapter 2

DOS Mitigation Strategies

By Lee Brandt

Chapter 3

Sanitizing Data

By Brian Demers

The inputs to your application represent the most significant surface area of attack for any application. Does your API power forms for user input? Do you display data that didn't originate in your API? Do users upload files through your API?

Any time data crosses a trust boundary - the boundary between any two systems - it should be validated and handled with care. For example, a trust boundary would be any input from an HTTP request, data returned from a database, or calls to remote APIs.

Let's start with a simple example: a user submission to the popular internet forum, Reddit. A user could try to include a malicious string in a comment such as:

```
<img src onerror='alert("haxor")'>
```

If this were rendered as is, in an HTML page, it would pop up an annoying message to the user. However, to get around this, when Reddit displays the text to the user, it is escaped:

```
&lt;img src onerror=&#39;alert(&quot;haxor&quot;)&#39;&gt;
```

which will make the comment appear as visible text instead of HTML, as shown in Figure 3-1.

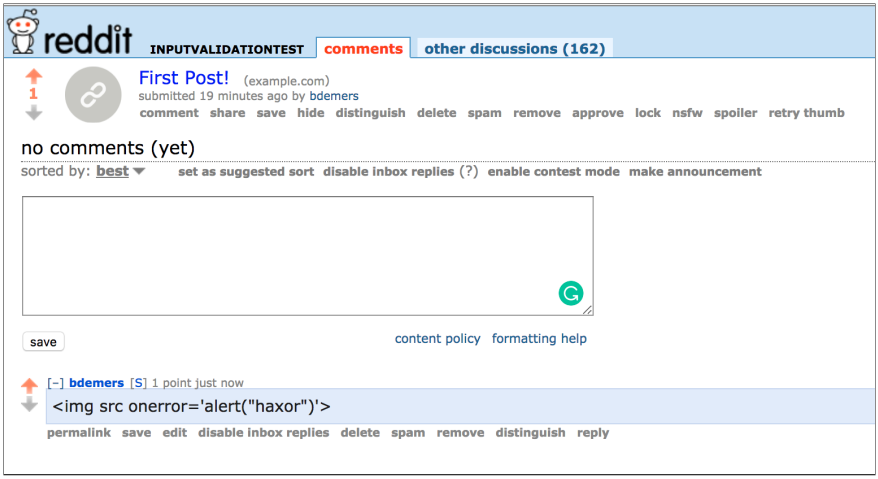

Figure 3-1: Reddit properly escapes user input

In this example the trust boundary is obvious as any user input should not be trusted.

There are a few different approaches you can use when validating input:

- Accept known good
- Reject bad
- Sanitize
- Do nothing

Accept Known Good

The known good strategy is often the easiest and most foolproof of the given options. With this approach each input is validated against an expected type and format:

- Data type, (Integers are Integers, booleans are booleans, etc)
- Numeric values fall within an expected range (for example: a person's age is always greater than 0 and less than 150)

- Field length is checked
- Specially formatted string fields such as zipcode, phone number, and social security number are valid

Most web frameworks have some type of declarative support to validate input fields built in. For example, in the Node.js world you can use the popular `validator` package to validate different types of input:

```
import validator from 'validator';
validator.isEmail('foobar@example.com');
```

Reject Bad Inputs

Rejecting known invalid inputs is more complicated than only accepting known good inputs (which we talked about above) and far less accurate. This strategy is typically implemented as a blacklist of strings or patterns. This technique may require many regular expressions to be run against a single field which may also affect the speed of your application. It also means that this blacklist will require updates any time a new pattern needs to be blocked.

Take a typical use-case of blocking 'bad-words'. This problem is incredibly complex as language usage varies across locale. These complexities can be demonstrated using the simple word: `ass`. It would be pretty easy to block this word alone, but doing so would also block the proper use of the word referring to donkeys. Then you need to think about both variations of the word and where those letters happen to come together: 'badass,' 'hard-ass,' 'amass,' 'bagasse', the first two are questionable while the second two are fine. Even if you block all of these and the thousands of other words that contain these three letters, there are still variations that would make it past: '4ss', 'as.s,' 'azz,' '@ss,' 'ās s,' or '\41\73\73' (escaped characters). As time goes on the list of blocked words would increase raising the total cost of the solution.

Another famous example of this technique is antivirus software. Your antivirus updates every few days to get a new blacklist of items to scan. And we all know how well that works ;)

Sanitize Inputs

Sanitizing inputs can be a good option when the input format is not strict but still somewhat predictable, such as phone numbers or other free-text fields. There are a few different ways to sanitize inputs, you could use a whitelist, a blacklist, or escape input.

Sanitize Input Using a Whitelist

When sanitizing data with a whitelist, only valid characters/strings matching a given pattern are kept. For example, when validating a phone number there are multiple formats people use, US phone numbers could be written as `555-123-1245`, `(555) 123-1245`, `555.123.1245`, or a similar combination. Running any of these through a whitelist that only allows numeric characters would leave `5551231245`.

Sanitize Input Using a Blacklist

A blacklist, of course, is the exact opposite of a whitelist. A blacklist can be used to strip HTML `<script>` tags or other non-conforming text from inputs before using input values. This technique suffers from the same shortcomings of the above section, Rejecting Bad Inputs on page 19. This type of sanitization must be done recursively until the value no longer changes. For example if the value `<scr<script>ipt foo bar` is only processed once the result would be still contain `<script`, but if done recursively, the result would be `foo bar`.

Sanitize Input Using Escaping

Escaping input is one of the easiest and best ways to deal with free-form text. Essentially, instead of trying to determine the parts of the input that are safe (as with the above strategies), you assume the input is unsafe. There are a few different ways to encode strings depending on how the value is used:

HTML/XML Encoding

Example Input:

```
<img src onerror='alert("haxor")'>
```

Result:

```
&lt;img src onerror=&#39;alert(&quot;haxor&quot;)&#39;&gt;
```

HTML/XML Attribute Encoding

Example Input:

```
<div attr="" injectedAttr="a value here"><div attr="">
```

Result:

```
<div attr="&quot; injectedAttr&#61;&quot;a value
 here&quot;&gt;&lt;div attr=&quot;">
```

JSON Encoding

Example Input:

```
{"key": "", anotherKey": "anotherValue"}
```

Result:

```
{"key": "\", anotherKey\": \"anotherValue\""}
```

Base64 Encoding

Example Input:

```
any random string or binary data
```

Result:

```
YW55IHJhbmRvbSBzdHJpbmcgb3IgYmluYXJ5IGRhdGE=
```

There are ways to escape just about any format you need SQL, CSV, LDAP, etc.

Do Nothing

The last type of input validation is the no-op. Along with being the easiest to implement it is the most dangerous and most strongly discouraged! Almost every application takes input from an untrusted source. Not validating inputs puts your application and users at risk.

Common Attacks

The examples in this chapter have discussed ways to validate inputs but have only hinted at the type of attacks used when inputs are not properly sanitized. Let's look at those potential attacks, and how to prevent them, now.

SQL Injection Attacks

SQL injection is by far the most common form of data sanitization attack, and remains number one in the OWASP Top 10 *(https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf)* (a popular list of the most commonly found and exploited software vulnerabilities). It's held the number one spot for over 10 years now.

SQL injection occurs when an attacker is able to query or modify a database due to poor input sanitization. Other query injection attacks are similar, as most are typically a result of string concatenation. In the following example, a simple user query string is built with concatenation.

```
userId = getFromInput("userId");
sql = "SELECT * FROM Users WHERE UserId = " + userId;
```

If the `userId` were `jcoder` the SQL query would be `"SELECT * FROM Users WHERE UserId = jcoder`, however, a malicious attacker might input `jcoder; DROP TABLE ImportantStuff` which would result in two statements being executed:

```
SELECT * FROM Users WHERE UserId = jcoder;
DROP TABLE ImportantStuff
```

Similarly, the user could enter `jcoder OR 1=1` which would query for a user with the ID of `jcoder` OR `true` (`1=1` is always true), this would return all of the users in the system.

The cause of this issue is the use of poor string concatenation. In the example above, the value of the `userId` input crosses a trust boundary and ends up getting executed. The best way around this is to use SQL prepared statements. The syntax for using prepared statements varies from language to language but the gist is that the above query would become `SELECT * FROM Users WHERE UserId = ?`. The question mark would be replaced with the input value and be evaluated as a string instead of changing the query itself.

Most web frameworks and ORM libraries provide tools to protect against SQL injection attacks, be sure to look through your developer library documentation to ensure you're using these tools properly.

XSS - Cross Site Scripting

A cross-site scripting attack (XSS) is an attack that executes code in a web page viewed by a user. There are three different types of XSS attacks:

- Stored XSS - A persisted (in a database, log, etc) payload is rendered to an HTML page. For example, content on an forum.
- Reflected XSS - Attack payload is submitted by a user, the rendered server response contains the executed code. This differs from Stored XSS where as the attack payload is not persisted, but instead delivered as part of the request, eg. a

link: `http://example.com/login?userId=<script>alert(document.cookie)</script>`

- **DOM based XSS** - The attack payload is executed as the result of an HTML page's DOM changing. With DOM based XSS the attack payload may not leave the victim's browser. The client side Javascript is exploited.

There are tons of resources online *(https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29)* that cover this topic in great detail, so I'll only provide a basic example here. Earlier in this chapter the string `<img src onerror='alert("haxor")'>` was posted as a Reddit comment. If this string isn't correctly escaped it would have resulted in an annoying popup, shown in Figure 3-2.

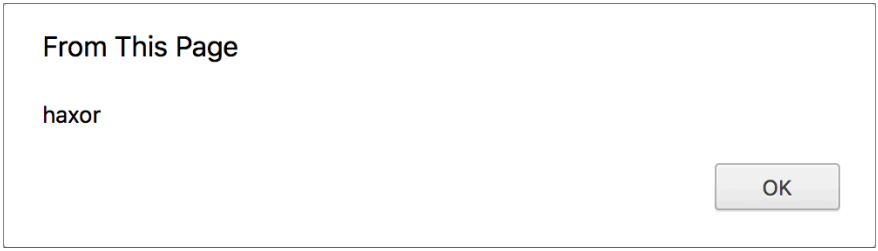

Figure 3-2: A JavaScript alert popup

You may see `alert()` used throughout examples when describing these attacks. The idea is if you can cause an alert to happen in the browser, you can execute other code that does something more malicious like sends your information (cookie, session ID, or other personal info) to a remote site.

File upload attacks

It is common for sites to support file uploads, particularly images such as profile avatars or photos. When uploading files, it is necessary to validate the type, size, and contents of these files. For example, if a user is uploading an avatar image, it's important to ensure the newly uploaded file is actually an image.

If an attacker can upload a PHP file named `avatar.php` instead of an image file, then later retrieve the file, unexpected and disastrous behavior may occur. Imagine what would happen if that file is executed on the server, you could have a remote code exploit on your hands. There are a few things you can do to prevent this type of attack:

Validate expected file types Check that file size is reasonable (if someone is uploading a 1GB image, you might have a problem) If storing the file to disk, do NOT use a user input field as part of the file name, eg: `../../../etc/config.file` Always serve the files with the correct Content-Type header (image/png, audio/mpeg) Run a virus scan on all uploaded files Do not allow uploads of web executed files: php, cgi, js, swf, etc. Process the files - rename, resize, remove exif data, etc - before displaying back to the user

Look For Other Attack Vectors

Inputs are everywhere, often only evident in hindsight. User input and file uploads are just the tip of the iceberg, but what if we consider more than input and instead the code itself? Here are a couple of examples to illustrate this point.

Your Dependencies

Do you trust all of your dependencies? How about all of the transitive dependencies of your application? It is not uncommon to for an application to have a page that lists its dependencies versions and licenses (the later might even be required depending on the license). The popular Node package manager (npm) has had a few projects which have contained maliciously formed license fields *(https://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers)*. In another npm incident, packages ran malicious scripts *(https://iamakulov.com/notes/npm-malicious-packages/)* upon installation automatically that uploaded the user's environment variables to a third party.

Every dependency is code you include from other systems across your trust boundary. Properly inspecting and validating your dependencies is a critical first step of any input sanitation plan. GitHub recently introduced automated security alerting *(https://blog.github.com/2017-11-16-introducing-security-alerts-on-github/)* to let you know when your dependencies might have security issues. Pay attention to these and you can prevent a lot of headaches.

Inbound HTML Requests

Almost all values from an HTTP request can be changed by the sender and need to be handled accordingly. To help illustrate this, here is a simple HTTP POST including numerous headers to `http://example.com/submit-me`:

```
POST /submit-me HTTP/1.1
Host: example.com
Accept: */*
Referer: http://example.com/fake.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: My Fake UserAgent <img src onerror='alert("haxor")'>
Content-Length: 37
Connection: Keep-Alive
Cache-Control: no-cache

foo=bar&key=value
```

You can see right away: request headers are user input too. Imagine for a moment that an HTTP client maliciously changes the User-Agent header. The logged User-Agent may falsely identify a request as coming from a different client application than the one in which it really originated.originated from. While that's unlikely to affect the current request, it might cause confusion in the application's logging and reporting system.

Further, the User-Agent could be visible from an internal web application that doesn't sanitize the User-Agent values before displaying them. In this case, an HTTP client could maliciously modify their User-Agent to any JavaScript code they want which would then be executed in an internal user's browser via XSS.

As these examples illustrate, even sanitizing relatively innocuous inputs is an important part of an overall security strategy.

Best Practices for Secure Data

While this chapter provides an overview of a few common types of attacks; there are many more out there.

First, you don't need to be an expert to prevent these attacks, but you do need to have some knowledge of them. The Open Web Application Security Project at *OWASP.org* is a great source information and examples on how to secure your application, often in multiple programming languages.

One of the most straightforward means of prevention is not to reinvent the wheel, and use an existing framework. Most frameworks contain tools to properly escape values, both on the frontend and backend, when used correctly.

Next, don't forget to monitor your application dependencies. There are mailing lists as well as open source and commercial tools to help you. New CVEs (Common Vulnerabilities and Exposures) are reported all of the time. For example, at the time of this writing a popular Java Web Container, Apache Tomcat 8, has about 60 CVEs *(https://tomcat.apache.org/security-8.html)* reported (and fixed). These reports, and the subsequent releases indicate that the project takes security seriously and updates regularly.

And finally, trust no one! As you have seen, any input into your API is an attack vector. Everything from an HTTP request to data returned from a database query to the files user upload could be

dangerous. Proper data validation and sanitization goes a long way to help mitigate risk.

Chapter 4

Managing API Credentials

By Joël Franusic

A critical part of designing an API is determining how to grant users access to sensitive or important parts of it. While many APIs have publicly accessible endpoints that don't require authentication, the vast majority of APIs require a user to authenticate before any request can be fulfilled.

You need to authenticate with Stripe to charge a credit card, you have to authenticate with Twilio to send an SMS, and you have to authenticate with SendGrid to send an email. There's really no way to avoid authentication.

This chapter will cover two main aspects of managing API tokens:

- Protecting tokens that you use to connect to other APIs
- Advice and suggestions for what sort of API token to use for an API that you are building

Keep Your Credentials Private

No matter whether you are using an API or building your own, the advice applies to you: Never put an API secret into your code. Never ever!

The biggest problem with storing credentials in source code is that once a credential is in source code it's exposed to any developers who have access to that code, including your version control provider.

Many people use GitHub to host their source code — what happens when you push company code containing sensitive credentials? That means GitHub staff can now view your credentials, it's a security risk. If that project is later shared with contractors, partners, customers, or even made public, your secret is no longer secret. The same is true for open source projects — accidentally publishing test API tokens or other sensitive data will cause enormous problems. There are attackers who scan GitHub commits looking for sensitive data like Amazon Web Services API tokens and then use these credentials to do things like mine cryptocurrencies, form botnets, and enable fraud.

If you can't store your credentials in source code, then how should you authenticate to your databases or 3rd party APIs? The short answer is to use environment variables instead. What that means is ignoring what you see in sample code (where secrets are entered directly into source) and instead loading secrets from environment variables which are managed from outside your source code and will not be stored in version control and every team member's text editor.

Below are two short snippets of sample code that demonstrate how important it is to use environment variables to store credentials.

Below we have some example Python code from Twilio. In this example, the `AC01a2bcd3e4fg567h89012i34jklmnop5` string is the "username" or "account ID" and the the `01234567890a12b34c567890de123fg4` string is the API token.

```
from twilio.rest import Client

account_sid = "AC01a2bcd3e4fg567h89012i34jklmnop5"
auth_token = "01234567890a12b34c567890de123fg4"
client = Client(account_sid, auth_token)
```

As you can see above, these API credentials are hard-coded into the program's source. This is a big no-no. Instead, as an example of what you *should* do, take a look at this example code from SendGrid, which uses the "os.environ.get" method in Python to grab the SendGrid API token from the `SENDGRID_API_KEY` environment variable.

```
import os
import sendgrid
from sendgrid.helpers.mail import *

apikey=os.environ.get('SENDGRID_API_KEY')
sg = sendgrid.SendGridAPIClient(apikey)
```

By pulling sensitive credentials from environment variables, which can be managed by configuration management and secret management software, your application code will remain credential-free.

Using environment variables to store secrets is an incredibly important first step to take in securing your code.

"What Kind of API Token Should I Use?"

Now that we've covered general advice for storing and using API tokens, let's review the common options for securing your API. Then we'll cover the different types of API tokens, touch on the advantages and disadvantages of each, and summarize with suggestions and a recommended approach.

Secure Your API

First, let's dig into some best practices:

- If your API is intended to be used to support an end user application, secure it using OAuth 2.0 to act as a Resource Server (RS) as defined by the OAuth 2.0 specification.
- If your API is intended to be used as a service by other

software, secure your API using an “API token” - a string that is unique for each client.

The first approach, implementing an OAuth 2.0 Resource Server, is intended to be used when an application is acting on behalf of the person using the app. The Instagram and Spotify APIs are great examples where every action of the API needs the user’s context to make sense.

The second approach, using an API token, is the best approach for automated software. Stripe, Twilio, and SendGrid are examples of this type of an API. While you certainly can, and eventually should consider, implementing OAuth 2.0 access tokens, doing so may be more overhead than telling your users to just use an API token.

Regardless of approach, the following patterns apply:

- Use the `Authorization: Bearer` header for authenticating to your API. The single most important reason is that URL parameters are captured in server access logs and caches. Any “secret” included as a parameter is no longer a secret. As a standard defined by RFC 6750, most HTTP clients have built-in methods for using Bearer tokens.
- Further, instead of using a simple unique string, use a JSON Web Token or JWT as the Bearer token. By using JWT and the claims defined by RFC 7519, you can support a wide range of scenarios using just one authentication method. And, since JWT is such a widely adopted standard, most programming languages and frameworks have first-class JWT decoding and validation built in. You get more flexibility and wider compatibility with less work!

Another important thing to keep in mind is that by using a JWT as a Bearer token, you can support both token types that we describe above. A JWT works equally well as an OAuth 2.0 access token or as an “opaque” generic API token. You could start by just offering authenticating with an API token, but later add support for OAuth 2.0 and give your customers the features that OAuth 2.0 easily

enables like automatic rotation of access tokens. You and your users get a clear, simple upgrade path for the life of your API.

Other Options for Authentication to Your API Service

Now, while I suggest that you have your clients authenticate to your API using JWTs as Bearer tokens, there are other patterns you will encounter. Below is a short list of other approaches to API authentication. Keep in mind that this is by no means an exhaustive list, just a list of the most common approaches and their tradeoffs:

Basic Authentication

This is the good 'ole "username and password" form of authentication method. Some APIs will use other words for "username" and "password" for example, Twilio calls the "username" the "Account SID" and the "password" the "Auth Token" but it works exactly the same.

If you decide to use Basic Auth to secure your API, keep in mind that your "username" and "password" should be random strings and not the same as the account username and password. You can generate these values by using an entropy source from your operating system (`/dev/random` on Unix-like systems or CryptGenRandom on Windows systems)

Opaque Tokens

These were the most common type of API token for APIs designed before OAuth 2.0 was standardized. Companies that use these types of token to secure their APIs include Stripe (with tokens that look like this: `sk_test_ABcdefGHiJkL0MnOpQ1rstU2`), and Okta (with tokens that look like this `00aBCdE0FGHijklmNO1pQ2RStuvWx34Y5z67ABCDEf`). These tokens have

no relationship with the account information and mean absolutely nothing outside those systems.

As with basic auth strings, we suggest that you generate opaque tokens using an entropy source from your operating system.

Another best practice for opaque tokens is to allow multiple tokens to be issued and used at the same time, as this will allow for key rotation.

Signed or Hashed Tokens

These tokens are cryptographically signed or hashed and can either be opaque tokens or contain carry information about themselves. One reason to use a signed or hashed token is to allow your API to validate tokens without the need for a database lookup. The best supported token type in this category is the JWT used for OpenID Connect. Other examples of tokens in this category include PASETO and Hawk. In general, we suggest using JWTs as described above.

Advanced API Token Considerations

Using OAuth 2.0 with OIDC, or just a JWT as a Bearer token is a significant milestone in the ongoing task of keeping your API secure. Depending on use case, type of data, and type of operations your API provides, you may need to consider additional steps to secure your API. Keep in mind that these extra considerations might not be appropriate for every kind of API. If your API is a fan-made API that gives programmatic access to Star Wars data (like swapi.co), simple API keys are probably sufficient. However, if your API deals with things in the real world or any form of sensitive data, you should consider the options below and choose the combination appropriate for your API and fits your compliance requirements.

Implement API Token Rotation

A great next step to take in securing an API is to rotate the API token automatically. Like passwords, regularly changing an API token will limit the damage a leaked or misplaced API token can cause. More importantly, by considering and implementing this at the beginning, if a token is leaked or when an employee leaves the team, you have a process for quickly responding and protecting your systems.

Additionally, one of the great side-effects of frequent API token rotation is that it forces best security practices. Sometimes, when a team is in a rush to deliver a critical feature, corners get cut and hard coding an API token instead of storing it properly may save a few minutes in the short term. If you rotate tokens on a regular basis, developers have follow the rules otherwise their code will stop working on the next rotation.

If you are using OAuth 2.0 to secure your API, token rotation is built-in to the OAuth 2.0 standard: An “access_token” always has a limited lifespan and must be rotated periodically using the “refresh_token”. As an additional benefit, if you’re using an OAuth server such as Okta, when you exchange the refresh_token for a new access_token, your authorization policies are re-evaluated. If a user’s API access has been limited, increased, or even revoked, your application will know.

Outside of OAuth 2.0, there isn’t an accepted best practice for implementing token rotation. Therefore your best and easiest option is to implement OAuth 2.0. Once you have a system in place to manage your API tokens, it makes sense to start rotating API tokens on a regular basis. Your specific rotation schedule will depend on the use case. For read/write operations in banking or healthcare, rotating every 5 or 10 minutes might be necessary. For read only access to a public Twitter feed, annually is probably sufficient. Regardless, you should always rotate keys after an employee leaves the team to protect against accidental or intentional misuse of API tokens by former employees.

Ideally, key rotation should also be paired with configuring your API to log events into a "Security and Information and Event

Management" (SIEM) system that you can use to monitor your API for suspicious events.

Monitor for Token Leaks

In addition to the use of SIEM systems as suggested above, an advanced technique is to scan sites like GitHub and S3 for leaked API keys. No best practices have emerged in this area yet, but a good technique should include automatically disabling and notifying end users when a token has been discovered in public as part of a scan.

Quite a few open source projects can be found that will scan for leaked tokens, a good way to find these services is to search for "github credential scan"

Bind Tokens To TLS Sessions

Finally, an interesting emerging technique that I'm keeping my eye on is the binding of tokens to TLS sessions. This technique is described in RFC 5056 *(https://tools.ietf.org/html/rfc5056)* and RFC 5929 *(https://tools.ietf.org/html/rfc5929)*.

The basic idea with "channel binding" is to tie an API token to a specific TLS session. In practice this would mean writing your API to issue tokens that can only be used in the same TLS session. This way, if an API token is compromised from a client, an attacker can't move that token to another client or machine because they would have a different TLS session for the initial issuer. This still isn't foolproof but the work and effort for the attacker just multiplied.

Key Takeaways for Managing API Credentials

In closing, here is my best advice for managing API credentials:

- Never paste a secret into your code. Never ever!
- Secure your API using OAuth 2.0 by writing your API to act

as an OAuth 2.0 “Resource Server”
- Use JSON Web Tokens (JWT) as your tokens to embed additional context
- Use the token as a Bearer token with the Authorization header to prevent leaking your token in logs and caches
- Implement regular token rotation to reduce the damage from leaked keys, poor practices, honest mistakes, and disgruntled employees.
- Monitor your source code for token leaks
- Implement “channel binding” to tie your API tokens to the TLS session they are requested over

Chapter 5

Authentication

By Matt Raible

Chapter 6

Authorization

By Sai Maddali

Chapter 7

API Gateways

By Keith Casey

° ° ° ° °

About the Authors

Lee Brandt

Keith Casey

Brian Demers

Joël Franusic

Sai Maddali

Dave Nugent

Matt Raible

API Security

The OAuth 2.0 authorization framework has become the industry standard in providing secure access to web APIs. It allows users to grant external applications access to their data, such as profile data, photos, and email, without compromising security.

OAuth 2.0 Simplified is a guide to building an OAuth 2.0 server. Through high-level overviews, step-by-step instructions, and real-world examples, you will learn how to take advantage of the OAuth 2.0 framework while building a secure API.

Who this book is written for

Whether you're a software architect, application developer, project manager, or a casual programmer, this book will introduce you to the concepts of OAuth 2.0 and demonstrate what is required when building a server. The practical examples in this book assume basic knowledge about HTTP communication, HTML, JSON, executing commands on a command line, and some PHP knowledge.

What you will learn from this book

- Discover how OAuth 2.0 facilitates authorization and authentication
- Understand how OAuth 2.0 and its extensions relate to each other
- Learn what it takes to build an OAuth server beyond what is written in the spec
- Learn how to securely handle OAuth in native and mobile apps
- Demystify how OAuth 2.0 relates to OpenID Connect

{okta}

developer.okta.com

ISBN 978-1-387-78313-7
90000
9 781387 783137