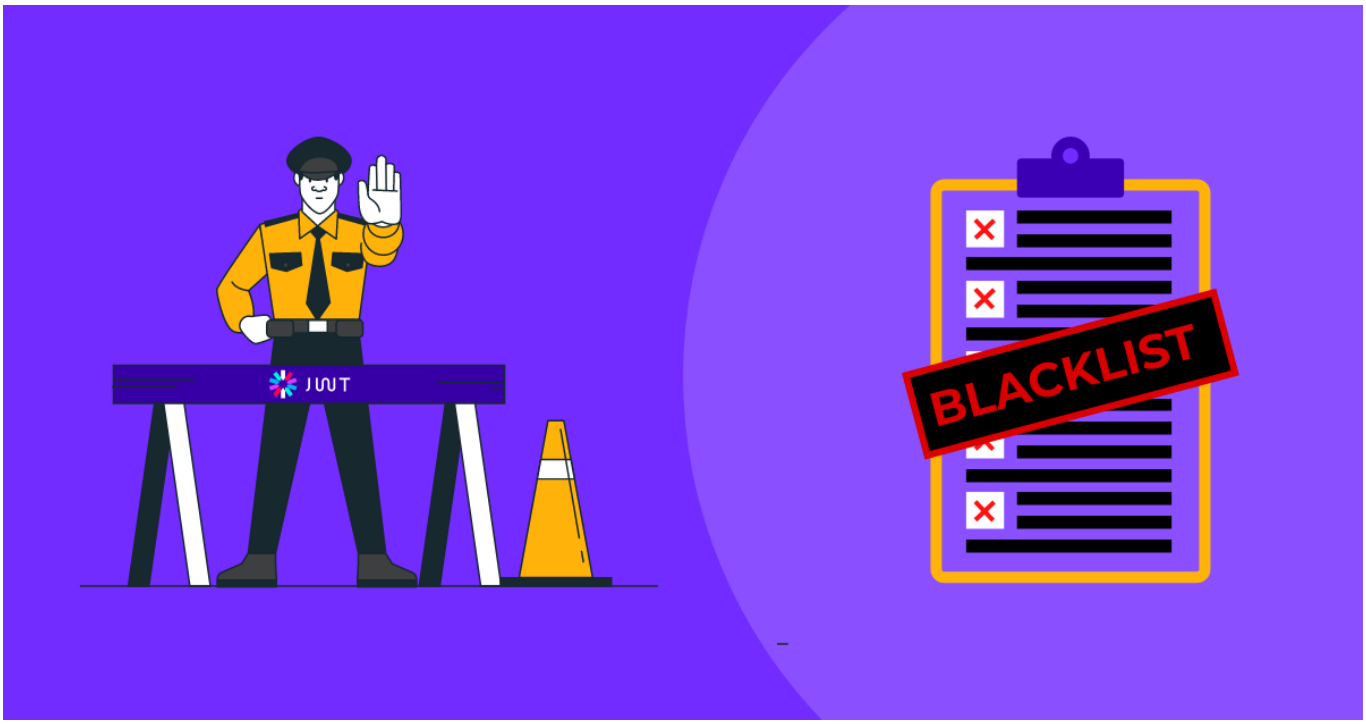

February 10, 2022

Revoking Access to JWT tokens with a Blacklist/Deny List

By Advait Ruia

Depending on who you listen to, JWTs are either a panacea for all your authentication problems or should be avoided like the plague. We're in two minds here at SuperTokens.

What is a JWT token?

A JWT, or JSON Web Token, is a string / token issued by the server that asserts properties contained in its “payload”. Its most common use case is for authentication (OAuth 2.0 + Open ID Connect) and session management.

As the name suggests, a JWT can contain any information inside it in JSON form. This is also known as “JWT claims”. For example, for session management, the JSON would at least need to contain the logged in user’s `userId`:

```
{
    "userId": "...",
    "expiry": 1646472008501,
    ...
}
```

If the JWT containing this information needs to expire, the JSON can also contain the token’s expiry time (as shown above). There is also a convention for the names of the keys to these fields. In our example above those are:

- `userId` -> `sub`
- `expiry` -> `exp`

Taking an example of the login process, a JSON containing the above information will be sent to the client, and then on each request, the client can send this JSON back to the server. This way, the server knows which userID is querying its APIs. If the JSON has expired, then the server can reject the request, and the user must login again.

However, from a security point of view, it’s very easy for the client to change the `userId` value in the JSON and spoof being another user. To prevent this, the server sends the original JSON’s “signature” along with the JSON. This “signature” is created using a secret

that is known only by the server, so the client cannot create a signature of a JSON by itself. Therefore, if the client changes the JSON, the server will fail to match the original JSON's signature with the incoming / changed JSON's signature (this is known as verifying the signature) and can then reject the request.

There are several algorithms that a server can use to generate a signature for a JSON:

- HMAC + SHA256
- RSASSA-PKCS1-v1_5 + SHA256
- ECDSA + P-256 + SHA256

The signing method chosen to create the signature must somehow be encoded in the JWT so that the same method is used when verifying the signature.

All in all, a JWT contains three parts:

- The header string containing information about the signing algorithm used.
- The body string containing the actual JSON.
- The signature string that can be used to verify that the JWT has not been changed by the client.

These three sections are concatenated with a `.` separator to form the full JWT Token. An example JWT can be later seen in this blog post.

Advantages of JWT Tokens

The JWT approach certainly has its advantages over opaque tokens. JWTs are:

- **Self-contained**: The JWT can contain the user's details (not just a session ID, like a cookie but other custom data such as user name and even permissions), together

with the token's expiry time so you don't need to query a database for that information. This is completely unlike an opaque token which, by its very nature, is just a meaningless jumble of alphanumeric characters.

- **Secure**: JWTs are digitally signed using either a secret (HMAC) or a public/private key pair (RSA or ECDSA) which safeguards them from being modified by the client or an attacker.
- **Stored only on the client**: You generate JWTs on the server and send them to the client. The client then submits the JWT with every request. This saves database space.
- **Efficient**: It's quick to verify a JWT, because it doesn't require a database lookup.

Disadvantages of JWT Tokens

The fact that JWTs are only stored client-side leads to a fundamental disadvantage with JWTs. And that is: how do you revoke a user's access?

Sure, JWTs have an expiry time and this can be as short-lived as you like. As soon as the access token expires however, the JWT is invalid and the client must re-authenticate with your server. This, of course, has a negative effect on the user experience..

But suppose the user intentionally logs out of your system? Or you want to kick them out because you fear that security has been compromised? You can't[1]: if they still have an unexpired token, they still have access.

There is a solution to this problem, but it does involve some extra work on your part. It is a technique that requires maintaining a JWT blacklist/deny list. In this article we'll show you what a JWT blacklist/deny list is, how to implement one, and discuss whether it is a good solution to this problem or not.

What is a JWT blacklist/deny list?

A JWT blacklist/deny list is a list of tokens that should no longer grant access to your system.

Where you maintain this list is up to you. You could use a traditional database, but a much better approach is to use an in-memory data cache, like Redis. An in-memory data cache provides much faster and more predictable seek times than data stored on disk.

That's what we'll use in this example, and we'll code our solution using Node.js and Express. If that's not your chosen technology stack then fear not: the fundamental approach is the same regardless of how you choose to build it.

OK, so how do I do it?

First, you'll need to instantiate your Express server application and set up Redis so that you can maintain a list of active JWTs:

```
import express from "express";
import bodyparser from "body-parser";
import jwt from "jsonwebtoken";
import redis from "redis";

const JWT_SECRET = "Ultra-secure-secret";

const app = express();
app.use(bodyparser.urlencoded({ extended: false }));
app.use(bodyparser.json());

let redisClient = null;

(async () => {
    redisClient = redis.createClient();
```

```
    redisClient.on("error", (error) => {
    console.log(error);
    });
    redisClient.on("connect", () => {
    console.log("Redis connected!");
    });

    await redisClient.connect();
})();

// listen for requests
const listener = app.listen(3000, () => {
    console.log("Server running");
});
```

To test the storage and subsequent revocation of a JWT, you'll need a way to create a user and generate a JWT for that user. Let's add the `createUser` endpoint for this. You can make a POST request to this endpoint with the name of the new user:

```
import jwt from "jsonwebtoken";

app.post("/createUser", (request, response) => {
    const token = generateAccessToken({ username: request.body.username });
    response.json(token);
});

const generateAccessToken = (username) => {
    return jwt.sign(username, JWT_SECRET, { expiresIn: "3600s" });
};
```

Hit that endpoint, and you'll see a JWT being issued for the user:

```
curl --location --request POST 'http://localhost:3000/createUser' \
--header 'Content-Type: application/json' \
--data-raw '{
    "username": "Derek"
}'
```

```
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IkRlcmVrIiwiaWF0IjoxNjQxMzA3MTgxI
```

If you paste that token into the online debugger, you'll see that the JWT has the user name encoded within it, along with other details such as the time it was issued (`iat`) and the time it expires (`exp`):

```
{
    "username": "Derek",
    "iat": 1641307181,
    "exp": 1641310781
}
```

The next thing you need to do is authenticate the user with that JWT whenever that user attempts to do anything with your application. This is a three-step process. First, you'll want to ensure that the JWT has been supplied. Then, you'll want to test whether the JWT supplied is in the blacklist/deny list stored in Redis. If it's not, then you'll finally want to verify that the JWT is valid.

We can encode these steps as Express middleware:

```
// JWT middleware
const authenticateToken = async (request, response, next) => {
    const authHeader = request.headers["authorization"];
    const token = authHeader && authHeader.split(" ")[1];

    // token provided?
    if (token == null) {
        return response.status(401).send({
            message: "No token provided",
        });
    }

    // token in deny list?
    const inDenyList = await redisClient.get(`bl_${token}`);
    if (inDenyList) {
        return response.status(401).send({
```

```
            message: "JWT Rejected",
        });
    }

    // token valid?
    jwt.verify(token, JWT_SECRET, (error, user) => {
        if (error) {
            return response.status(401).send({
            status: "error",
            message: error.message,
            });
        }

        request.userId = user.username;
        request.tokenExp = user.exp;
        request.token = token;

        next();
    });
};
```

Now what will happen is that any time the user you created earlier attempts to visit your home route (`/`), the JWT they supply will be checked against the blacklist/deny list first and then verified before being granted access.

That's all well and good, but at the moment you have no way to revoke a JWT, so there's nothing in the blacklist/deny list.

Let's create another route that will simulate a user logging out. When this endpoint is hit, the user's JWT is persisted to Redis, using the format `bl_<token>` for the key, and the value being the actual token. We'll set the key to expire when the token itself expires, so as not to fill up Redis with lots of expired tokens.

```
app.post("/logout", authenticateToken, async (request, response) => {
    const { userId, token, tokenExp } = request;

    const token_key = `bl_${token}`;
    await redisClient.set(token_key, token);
```

```
    redisClient.expireAt(token_key, tokenExp);

    return response.status(200).send("Token invalidated");
});
```

Issue a `POST` request to the `/logout` endpoint:

```
curl --location --request GET 'http://localhost:3000/' \
--header 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6I
```

Now the user's JWT is technically still valid (until it expires), but now it's in the blacklist/deny list and will therefore be intercepted on subsequent requests. Repeat the `POST` request to the home route above and you'll get an HTTP 403 Forbidden error, with a message `"JWT Rejected"`.

However, this won't help if you want to deny access to a user who hasn't logged out and whose token is still valid. The problem is that you won't know what the user's access token is at that time, and therefore you won't know what to put in the cache.

To solve this issue, you should add the ability to blacklist a user's `userID` via redis and check that blacklist as well during JWT verification. Unlike the JWT blacklist, this entry won't have an expiry time associated with it.

> *You can find a working version of this example on Github.*

Is this a good idea?

So now you see that this concept of a JWT blacklist/deny list is relatively easy to implement. But is it the best way of handling this situation?

In our opinion, it's not.

The biggest advantage of JWTs is that they make session verification fast. If you maintain a blacklist/deny list and have to query it on every API call, then you've lost that advantage.

Instead, what we advocate is a solution for session management that combines the respective strengths of both JWTs and opaque tokens.

In this workflow, when the user logs in, the server issues a both short-lived JWT (the access token), and a long-lived opaque token (the refresh token). When a user is granted access, both of these tokens are sent to the client.

When the user attempts to access a resource, they send the JWT access token along with every request. When the JWT expires, the client then uses the opaque refresh token to request a new JWT and a new opaque refresh token. This process is known as refresh token rotation.

The client then uses this new JWT to make subsequent requests and the process continues.

The advantages and disadvantages of issuing two tokens and of implementing refresh token rotation

The benefits of this approach is that if you want to revoke access, then all you need to do is invalidate the opaque token on the server side. Then, when the refresh endpoint is called, the server looks up the opaque token, sees that it has expired, and logs the user out.

Note that this doesn't solve the problem of still-valid JWTs existing on the client. But, because you can make those JWTs much more short-lived (even just a few mins), you should find that it's not too big a problem.

In this way, you keep one of the key benefits of using JWTs and that's the fact that you don't have to keep accessing the database on each API call to verify the JWT - you only need to do a database lookup when refreshing the session - which happens relatively rarely.

If this sounds ideal for your use case, check out SuperTokens' implementation of this session flow. SuperTokens uses this method to harness all the benefits of using JWTs while mitigating many of their disadvantages by combining them with opaque tokens.

Furthermore, changing the refresh token on each use adds additional security benefits like being able to detect session hijacking.

Written by the Folks at SuperTokens — hope you enjoyed!

JWT Decoder

SuperTokens

View Report