Securing HHTP and RMC requests

With JWT and Passport

Exposing api routes and event patterns creates the risk of inviting malicious activity from third parties. Data can be modified by anyone who has access to the route url so validating that the user is authorized to use the exposed functions is of crucial importance.

There are many ways in which an API can be secured but for the solution in my project I have chosen to use the classic JWT token security in combination with the Passport Javascript library.

JasonWebToken

JWTs consist of three parts: Header, Payload, and Signature.

* Header: Describes the type of token and the signing algorithm being used.
* Payload: Contains claims or statements about the entity (user) and additional data.
* Signature: Created by combining the encoded header, encoded payload, and a secret key using a specified algorithm.

When the user attempts to log into the system, his log in information will be used by the server in order to generate a JWToken using a secret key hosted on the server. This token is then returned to the user browser where it is then stored as a cookie. The server never has to host any data about this token except for the secret key with which the Signature has been hashed. This enables servers that share the user base but do not share the user data to validate if the user is authorized to use the system by only holding in memory the secret key.

Each time the user makes a request, the route guard is set up in order to validate the JWT token and see if the original hashed data has been tempered with.

Passport.Js

Passport is a middleware library for Node.js that amplifies the control the developer has over the authentication mechanism. It allows the developer to implement hand-written validation strategies that will be integrated into the validation router.

* Passport.js authenticates requests by delegating the authentication process to specific strategies.
* Strategies handle the details of verifying credentials and, if successful, Passport.js attaches the user object to the request.

In our case (my project) this delegated custom validation method entails that the system checks weather the security middleware trigger was caused by an HTTP request or by an RPC (RabbitMQ) request.

**Implementation:**

First, we need to describe what we intend to secure. My project uses a REST API used to call CRUD functionalities that manage a book library database. This means that we need to secure the HTTP requests that come through from the front end ( this also accounts for the initial log in that the user has to make). Also, each time a CRUD request is resolved, a message is sent to an analytics microservice via RabbitMQ. The RPC routes that the queue uses also have to be secured. With this in mind, let us proceed to the code implementation.

Let us begin with the user creation. An user is composed of 2 information fields: email and password. When a new user is created, it’s password is hashed and store in the database. Any future log ins will have the credentials compared against this initial hashing.

Next, for user authentication. I have created 2 validation strategies that will be integrated with the Passport.Js interfaces. One strategy for local (email and password authentication), and one for JWT authentication.

@Injectable()

export class LocalStrategy extends PassportStrategy(Strategy) {

  constructor(private readonly *usersService*: UsersService) {

*super*({ usernameField: 'email' });

  }

  async validate(*email*: string, *password*: string) {

    return this.usersService.validateUser(*email*, *password*);

  }

}

This code block will be run by the middleware when an http log in request is made and uses a boolean function in the user service in order to validate the data.

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

  constructor(

*configService*: ConfigService,

    private readonly *usersService*: UsersService,

  ) {

*super*({

      jwtFromRequest: ExtractJwt.fromExtractors([

        (*request*: any) => {

          return *request*?.Authentication;

        },

      ]),

      secretOrKey: *configService*.get('JWT\_SECRET'),

    });

  }

  async validate({ *userId* }: TokenPayload) {

    try {

      return await this.usersService.getUser({

        \_id: new Types.ObjectId(*userId*),

      });

    } catch (err) {

      throw new UnauthorizedException();

    }

This code block represents the JWT authentication guard. The constructor extracts the JWT payload information and attempts to use the decrypted id in order to retrieve the user.

These strategies would function well enough on a monolithic application but since our application is deployed using microservices then the JWT key shouldn’t be exposed to services that are not responsible for authentication. In order to compensate for this issue we must implement another middleware guard that protects the microservices communication paths. This middleware will not validate the JWT token per se but it will use an RPC route to send the token to our original strategy described before.

@Injectable()

export class JwtAuthGuard implements CanActivate {

    constructor(@Inject(AUTH\_SERVICE) private *authClient*: ClientProxy) { }

    canActivate(*context*: ExecutionContext): boolean | Promise<boolean> | Observable<boolean> {

        const authentication = this.getAuthentication(*context*);

        return this.authClient.send('validate\_user', {

            Authentication: authentication

        }).pipe(

            tap((*res*) => {

                this.addUser(*res*, *context*);

            }),

            catchError(() => {

                throw new UnauthorizedException('Invalid authentication');

            })

        );

    }

    private getAuthentication(*context*: ExecutionContext) {

        let authentication: string;

        if (*context*.getType() === 'rpc') {

            authentication = *context*.switchToRpc().getData().Authentication;

        } else if (*context*.getType() === 'http') {

            authentication = *context*.switchToHttp().getRequest()?.cookies.Authentication;

        }

        if (!authentication) {

            throw new UnauthorizedException('No authentication provided');

        }

        return authentication;

    }

This code block represents a common strategy (that will be shared in a library accessible for all microservices) that will be implemented for the CRUD and analytics requests. The function determines weather the request is HTTP or RPC, extracts the cookie and uses an RPC route to send the token to the Authentication microservice controller, and then that request will be picked up by the JWT strategy we set up previously.

@UseGuards(JwtAuthGuard)

  @MessagePattern('validate\_user')

  async validateUser(@CurrentUser() *user*: User) {

    return *user*;

  }

}

(JwtAuthGurad here is the guard we set up for Passport.Js)

This way, both HTTP and RPC routes are secured.

Passport will the attach the decrypted payload data to the request so this makes the user information immediately available for analytics after validating.